



Built February 12, 2018

Contents:

- User's Guide
 - Welcome to PhysX
 - About this User Guide
 - Physics vs. PhysX
 - World and Objects
 - Snippets
 - What are PhysX Snippets?
 - HelloWorld: PhysX Basics
 - Using PhysX Visual Debugger with SnippetHelloWorld
 - Building with PhysX
 - Build Settings
 - Build Configurations
 - Header Files
 - Libraries
 - Redistribution
 - The PhysX API
 - Introduction
 - Memory Management
 - Error Reporting
 - Math Classes

- Connecting PhysX Objects with User Application Objects
- Type Casting
- Reference Counting
- Using Different Units
- Assertions
- Determinism
- Startup and Shutdown
 - Introduction
 - Foundation and Physics
 - Cooking
 - Extensions
 - Optional SDK Components
 - Delay-Loading DLLs
 - Shutting Down
- Threading
 - Introduction
 - Data Access from Multiple Threads
 - Asynchronous Simulation
 - Multithreaded Simulation
- Geometry
 - Introduction
 - Geometry Types
 - Mesh Scaling
 - PxGeometryHolder
 - Vertex and Face Data
- Rigid Body Overview
 - Introduction
 - Rigid Body Object Model
 - The Simulation Loop
- Rigid Body Collision

- Introduction
- Shapes
- Simulation Shapes and Scene Query Shapes
- Kinematic Triangle Meshes (Planes, Heightfields)
- Broad-phase Algorithms
- Regions of Interest
- Broad-phase Callback
- Collision Filtering
- Aggregates
- Creating an Aggregate
- Populating an Aggregate
- Releasing an Aggregate
- Amortizing Insertion
- Trigger Shapes
- Interactions
- Rigid Body Dynamics
 - Velocity
 - Mass Properties
 - Applying Forces and Torques
 - Gravity
 - Friction and Restitution
 - Sleeping
 - Kinematic Actors
 - Active Transforms
 - Active Actors
 - Dominance
 - Solver Iterations
 - Immediate Mode
 - Enhanced Determinism
 - Axis locking

- Simulation
 - Callback Sequence
 - Simulation memory
 - Completion Tasks
 - Synchronizing with Other Threads
 - Substepping
 - Split sim
 - Split fetchResults
- Advanced Collision Detection
 - Tuning Shape Collision Behavior
 - Contact Modification
 - Contact reporting
 - Extracting Contact information
 - Continuous Collision Detection
 - Raycast CCD
 - Speculative CCD
 - Persistent Contact Manifold (PCM)
- Joints
 - Joint Basics
 - Fixed Joint
 - Spherical Joint
 - Revolute Joint
 - Prismatic Joint
 - Distance Joint
 - D6 Joint
 - Custom Constraints
- Articulations
 - Creating an Articulation
 - Articulation Joints
 - Driving an Articulation

- Articulation Projection
- Articulations and Sleeping
- Scene Origin
 - Shifting The Scene Origin
- GPU Rigid Bodies
 - Introduction
 - Using GPU Rigid Bodies
 - What is GPU accelerated?
 - Tuning
 - Performance Considerations
- Geometry Queries
 - Introduction
 - Raycasts
 - Overlaps
 - Penetration Depth
 - Sweeps
 - Additional PxGeometryQuery functions
 - PxMeshQuery
- Spatial Queries
 - PxSpatialIndex
- Scene Queries
 - Introduction
 - Basic queries
 - Touching and blocking hits
 - Query modes
 - Filtering
 - User defined hit callbacks for unbounded results
 - Batched queries
 - Volume Caching
 - Single Object Caching

- PxPruningStructureType
- PxPruningStructure
- Vehicles
 - Introduction
 - Algorithm
 - First Code
 - Snippets
 - Advanced Concepts
 - More Advanced Concepts
 - Tuning Guide
 - Troubleshooting
 - References
- Character Controllers
 - Introduction
 - Kinematic Character Controller
 - Creating a character controller
 - Overlap Recovery Module
 - Character Volume
 - Volume Update
 - Moving a Character Controller
 - Graphics Update
 - Auto Stepping
 - Climbing Mode
 - Up Vector
 - Walkable Parts & Invisible Walls
 - Obstacle Objects
 - Hit Callback
 - Behavior Callback
 - Character Interactions: CCT-vs-dynamic actors
 - Character Interactions: CCT-vs-CCT

- Hidden Kinematic Actors
- Time Stepping
- Invalidating Internal Geometry Caches
- Runtime Tessellation
- Troubleshooting
- Particles (deprecated)
 - Introduction
 - Creating Particle Systems
 - Particle Management
 - Parameter Guide
 - Collision Handling
 - Best Practices / Troubleshooting
 - GPU/CUDA Acceleration
 - Sample Discussion
 - References
- Cloth
 - Introduction
 - Simulation Algorithm
 - Best Practices / Troubleshooting
 - Snippet Discussion
 - References
- Debug Visualization
 - Introduction
 - Usage
- PhysX Visual Debugger (PVD)
 - PVD
 - Basic Setup (SDK Side)
 - Advanced Setup
- Simulation Statistics
 - Interface

- Usage
- Serialization
 - Introduction
 - First Code
 - In-depth Discussion
 - Common Use Cases
 - Snippet Discussion
 - Best practices / Troubleshooting
 - PVD
- Extending Serialization
 - Introduction
 - Overview
 - Binary Serialization of Custom Classes
 - RepX Serialization of Custom Classes
- Best Practices Guide
 - Introduction
 - Debugging
 - Performance Issues
 - Character Controller Systems using Scene Queries and Computation
 - Quantizing HeightField Samples
 - Reducing memory usage
 - Behavior issues
 - GPU Rigid Bodies
 - Determinism
- Migrating From PhysX SDK 2.x to 3.x
 - Removed Features
 - Basics
 - API Design Changes
 - Material Indexes

- Continuous Collision Detection
- Pose Description
- Shape Description
- Joints
- Time Stepping
- Collision Filtering
- Scene Queries
- Migrating From PhysX SDK 3.2 to 3.3
 - Math Classes
 - Scene Query API
 - SPU batch queries
 - Core PhysX
 - PhysX Extensions
 - PhysX Character Controller
 - PhysX Vehicles
 - CCD
 - PhysX Visual Debugger
 - PhysX Cloth
 - RepX Serialization
 - Binary Serialization
 - PhysX TaskManager
- Migrating From PhysX SDK 3.3 to 3.4
 - Deprecated APIs
 - Core PhysX
 - Contact Generation
 - PhysX Cooking
 - Reference Counting
 - PhysX Visual Debugger
 - Scene queries
 - PxExtensions

Welcome to the NVIDIA® PhysX® SDK version 3! With this second n SDK, we are excited to bring you a great number of enhancements, i API improvements. Because so much has changed in the API, we experienced PhysX users to read through this guide to familiarize them: programming interface.

About this User Guide

This Guide will help the reader to understand the PhysX-3 SDK and this Guide presents an overview of the features and implementation of the SDK performance in general use as well as in specific cases.

That is, this Guide covers:

- what PhysX does;
- how PhysX works;
- how well PhysX is expected to perform;
- how to use PhysX by example, and performance in those use cases.

The Guide does not attempt to explain the details of the API, and the reader should refer to the PhysX API Reference Documentation. (See PhysX API Reference Documentation directory under the main directory where the PhysX SDK is unpacked.) Users migrating from PhysX-2 will find the *Migrating From PhysX 2.x to 3.x* chapter of particular interest.

Physics vs. PhysX

Physics is a rich and broad scientific pursuit, an attempt to explain with the behavior of all matter, everything in the entire universe, using concepts like time, energy, inertia, momentum and force. In physics, space is a continuous infinitely in three dimensions, and can be divided into infinitely small units with fine precision. In other words, positions in physics space are described by real numbers in a 3-dimensional Cartesian coordinate system. In contrast, simulation space are vectors of single-precision floating point numbers.

Like the dimensions of space, time in physics is described by a real duration divisible into arbitrarily small intervals. Physics promises that if forces imposed on a system are known throughout some period of time, and the system is known precisely at some instant of time in that period, then the system can be determined precisely for any other instant throughout that period. For example, if one observes a ball falling towards the ground, and measures its position and velocity, one can calculate what the position and velocity of the ball must have been at an earlier time, as well as what they must become at a later time. In contrast, simulation is discrete, not continuous, and it runs only 'forwards'. That is, a simulated system is known only at specific instants in time, usually relative to the start and the simulation may only step forwards in time, never backwards. The system in between time steps is not precisely determined.

Because of such approximations a PhysX simulation is subject to limitations not seen in ordinary physics, and later sections in this Guide will highlight these limitations wherever they are likely to concern the user. PhysX is best suited for interactive 3D applications where performance and scalability are more important than precision. Here "quasi-real time" means that advancing a PhysX simulation by one time step, say 1/60 second, will take less than that amount of time on a real computer if the performance of the hardware platform is sufficient for the simulation. That the PhysX SDK is more widely used in computer and video game scientific or engineering applications is both a cause and an effect of these choices. Consequently this Guide usually refers to PhysX in the context of 'the game world', 'rigid body game objects', 'the character', etc.

World and Objects

The basic concepts of the world within a PhysX simulation are easy to v

- The PhysX world comprises a collection of Scenes, each conta
Actors;
- Each Scene defines its own reference frame encompassing all (
- Actors in different Scenes do not interact with each other;
- The three major types of Actors are rigid bodies, particles and c
- Characters and vehicles are complex specialized objects made
- Actors have physical state : position and orientation; veloc
energy; etc,
- Actor physical state may evolve over time due to applied force
as joints or contacts, and interactions between Actors.

Games are a very visual medium and audible and games usually pl
requirements on their graphics and sound. Production quality graph
outside the scope of PhysX, but it is enormously valuable to be ab
otherwise hidden world. Some of our example programs come with r
visualization, and we also provide a stand-alone debugging tool ca
Debugger (PVD). PVD provides a graphical view of the PhysX scene to
tools to inspect and visualize variables of every PhysX object. Addi
record and visualize memory and timing data. See [PhysX Visual D](#)
details.

What are PhysX Snippets?

In the context of the PhysX SDK, a 'Snippet' is a simple, minimalistic code sample. The PhysX SDK version 3.3.0 offers a collection of Snippets to illustrate usage of the SDK in a concise format, free from the complexity of a sample framework or application. The Snippets folder is in the top-level directory of the PhysX SDK, along with Documentation, Include, Samples, etc.

The folder {SDK Root}/Snippets/compiler/{platform} contains the Snippets solution files. For example, the folder Snippets/compiler/vc14win64/Snippets.sln

Although a few of the Snippets support rendering, (Win32, Win64, OS dependent), most Snippets do not provide rendering, require no input, and provide output through messages. Although Snippets can be run from a command prompt or by clicking the executable icon, the best way to explore Snippets is by view them in the Visual Studio IDE, and running the program in the debugger.

HelloWorld: PhysX Basics

SnippetHelloWorld illustrates basic use of PhysX, from startup to shutdown scene, and is a good place to start learning the PhysX API. The code files comprise a single source file, but SnippetHelloWorld, among others, uses rendering through a second source file. SnippetHelloWorld creates a scene with stacks on a plane, and if rendering is enabled, allows the user to create and fire a ball from the camera position.

The primary code for SnippetHelloWorld is found in `Root}/Snippets/SnippetHelloWorld/SnippetHelloWorld.cpp`.

On Windows, PhysX requires Visual Studio 2013 or later versions.

Build Settings

The PhysX headers should compile cleanly at the highest typical warning settings. For Visual Studio, `-Wall -Wextra -pedantic` for gcc- and clang-based compilers, warning settings may result in a small number of benign informational warnings.

The PhysX source projects and snippets will compile cleanly using the makefiles supplied.

Build Configurations

The SDK has four build configurations available, designed for development and deployment.

- the *debug* build can be useful for error analysis, but contains assertions and development which some customers may find too intrusive. Optimizations are turned off for this configuration.
- the *checked* build contains code to detect invalid parameters, API misuse and other incorrect uses of the API which might otherwise cause errors or failures in simulation.
- the *profile* build omits the checks, but still has PVD and memory instrumentation.
- the *release* build is built for minimal footprint and maximum speed, with no checks and instrumentation.

Simulation works the same way in all of them, and all are compiled with the same optimization levels (except debug configuration).

Note: We strongly recommend that you use the checked build as the configuration for day-to-day development and QA.

Note: PhysX libraries of different build configurations (e.g. the DEBUG version of PhysXVehicle and the CHECKED version of PhysXVisualDebuggerSD) should not be mixed in an application because this will result in a CRT conflict.

Header Files

To build your own PhysX app, you will need to add some include paths to your project makefile or IDE.

Users should specify the root "Include" and "Lib" folders in the additional library directories respectively. There is a combined include header available

```
#include "PxPhysicsAPI.h"
```

This will include the entire PhysX API including core, extensions, and vehicle. It is possible to include subsets of the SDK if preferred, for example:

```
#include "vehicle/PxVehicleSDK.h"
```

Libraries

At a minimum, applications need to link against the following libraries with platform extension (e.g. ".lib" or ".a") and with * being a x86 or x64 for Windows or Linux respectively.

- PhysX3_*.lib
- PhysX3Common_*.lib
- PxFoundation_*.lib

Note: The static libraries we provide with the Windows binary distribution are linked against the Multi-Threaded static C Run-Time (CRT) libraries. This means that your application must also use the same CRT flavor. If you need to use a different version, you must upgrade to our source license. The source distribution can be recompiled using different CRT settings.

Redistribution

On the Windows platform, you need to redistribute some of our DLLs to your application:

- PhysX3Common_*.dll - will always be needed.
- PhysX3_*.dll - will always be needed.
- PxFoundation_*.dll - will always be needed.
- PhysX3Cooking_*.dll - you only need to bundle if your application has data on the fly.
- PhysX3GPU_*.dll - is only needed if your application runs some GPU.
- PhysX3CharacterKinematic_*.dll - is only needed if your application uses a character controller.
- PXPvdSDK_*.dll - is only needed if your application uses PVD.

Where * is a platform specific suffix, e.g. x86 or x64. You will need to choose the correct version depending on whether your application is built in 64 bit mode.

Introduction

This chapter covers the basic patterns common to the PhysX application interface (API.) We are committed to keeping this API stable and backward compatible from one minor release to the next, to protect the investment you make in your code.

The PhysX API is composed primarily of abstract interface enumerations and functions defined by the API have the prefix Px.

Note: There is currently one section of the public API which does not have the Px prefix: the PhysX Visual Debugger connection library which has the prefix Pxv.

The PhysX libraries also expose some classes and functions that are not part of the public API. These are primarily containers and platform abstractions that are used internally by the PhysX libraries which are distributed as source, and are also used in some third-party applications. Even though they can be recognized because they do not have the Px prefix. Even though they are in principle accessible to users, they are largely undocumented and we strongly recommend against their use in applications. For that reason, we strongly recommend against their use in applications.

Memory Management

PhysX performs all allocations via the *PxAllocatorCallback* interface. You use this interface in order to initialize PhysX:

```
class PxAllocatorCallback
{
public:
    virtual ~PxAllocatorCallback() {}
    virtual void* allocate(size_t size, const char* typeName, const
        int line) = 0;
    virtual void deallocate(void* ptr) = 0;
};
```

The size of the request is specified in bytes, and PhysX requires that the memory returned be 16-byte aligned. On many platforms `malloc()` returns memory aligned, and on Windows the system function `_aligned_malloc()` provides the other parameters to `allocate()` are a string which identifies the type, the `__FILE__` and `__LINE__` location inside PhysX code where the allocation occurred. Refer to `PxAllocatorCallback::allocate()` to find out more about them.

A simple implementation of the allocator callback class can be found in the PhysX Extensions library, see class *PxDefaultAllocatorCallback*.

Note: On some platforms PhysX uses system library calls to determine the type name, and the system function that returns the type name may call the allocator. If you are instrumenting system memory allocations, you may observe unexpected behavior. To prevent PhysX requesting type names, disable allocation reporting by calling `PxFoundation::setReportAllocationNames()`.

You can place PhysX objects in memory owned by the application using the `Serialize` and `Deserialize` methods. See [Serialization](#) for details.

As an alternative to instrumenting the allocator, you can obtain detailed information about memory allocation in the PhysX Visual Debugger (see: [PhysX Visual Debugger](#)).

Error Reporting

PhysX logs all error messages through the *PxErrorCallback* interface. You must implement this interface in order to initialize PhysX:

```
class UserErrorCallback : public PxErrorCallback
{
public:
    virtual void reportError(PxErrorCode::Enum code, const char*
        int line)
    {
        // error processing implementation
        ...
    }
};
```

There is only a single function to implement, *reportError*. This function receives a message, or print it on the application's output console. For the error codes *eABORT*, *eINVALID_PARAMETER*, *eINVALID_VERSION*, *eINTERNAL_ERROR* and *eOUT_OF_MEMORY*, breaking into the debugger is a more appropriate choice. Whatever you do, do not just ignore the message.

A simple implementation of the error callback class can be found in the PhysX library, see class *PxDefaultErrorCallback*.

Math Classes

The common math classes used in PhysX are `PxVec2`, `PxVec3`, `PxMat44`, `PxTransform`, `PxPlane` and `PxQuat`, which are defined in header files, e.g. `(SDKRoot)/Include/foundation/PxVec3.h`. The types support operator overloads and typical math operations. Zero and identity values can be constructed by passing the arguments `PxZero` and `PxIdentity` respectively.

Some points to note are:

- `PxTransform` is a representation of a rigid body transform as a rotation and a position vector, and PhysX functions which take transforms all use this type.
- `PxPlane` is a homogeneous plane equation: that is, the constructor `PxPlane(n, d)` represents the equation $n \cdot x + d = 0$.

`PxMat33` and `PxMat44` matrices represent transformations with `columns` (pre-multiply with matrix on the left hand side) and are stored in column-major order. This format is layout compatible with popular graphics APIs such as `Direct3D`. For example, to set the model transformation for a rigid body

```
// retrieve world space transform of rigid body
PxTransform t = rigidActor.getGlobalPose();

// convert to matrix form
PxMat44 m = PxMat44(t);

// set to OpenGL
glMatrixMode(GL_MODELVIEW);
glPushMatrix();

// PxMat44::front() returns a pointer to the first matrix element
glMultMatrixf(m.front());

// draw model

glPopMatrix();
```

DirectX uses row-major storage for matrices by default (D3DMATRIX basis vectors in rows (post-multiply on the right), so PxMat44 may be D3DXMATRIX types directly.

Connecting PhysX Objects with User Application Objects

Often an application needs to associate PhysX objects with application logic or rendering purposes. An easy way to connect a single user application object to a PhysX object is to use the *userData* member provided by the most PhysX classes (*PxActor::userData*, *PxShape::userData*, *PxMaterial::userData*, etc.). This member is a *void** pointer which is reserved for application use. Each class has a *userData* field, so to manage multiple associations another mechanism

Type Casting

PhysX API interface classes inherit from a top-level interface call provides mechanisms for type-safe down-casting between interface type. To cast from a PxActor to a PxRigidDynamic, use the following idiom:

```
PxActor* actor = <...>
PxRigidDynamic* myActor = actor->is<PxRigidDynamic>();

const PxActor* actor = <...>
const PxRigidDynamic* myActor = actor->is<PxRigidDynamic>();
```

This pattern can be used to cast to intermediate types in the hierarchy, such as PxRigidActor, but this is somewhat slower than casting to concrete types. PxBase provides the following capabilities:

- `getConcreteType()` provides an integer value which corresponds to the type of an object
- `getConcreteTypeName()` provides a string name of the concrete type
- `isKindOf()` provides string-based testing of inheritance

Reference Counting

Some PhysX objects are designed to be shared and referenced multiple times in the scene graph. For example, a `PxConvexMesh` may be referenced by multiple objects, each sharing the same geometry but associated with different material types. These types are `PxTriangleMesh`, `PxHeightField`, `PxConvexMesh`, `PxMaterial` and `PxShape`. Each object of these types has a reference count. The rules for reference counting are as follows:

- when an object is created from `PxPhysics`, it has a reference count of 1.
- when an object's reference count reaches 0, the object is destroyed.
- when a new counted reference is created, the reference count is incremented. Counted references are as follows:
 - when a `PxShape` references a `PxConvexMesh`, `PxTriangleMesh`.
 - when a `PxShape` references a `PxMaterial`.
 - when a `PxRigidActor` references a `PxShape`.
 - when a `PxCloth` references a `PxClothFabric`.
- when a counted reference is destroyed, or the object's `release()` method is called, the reference count is decremented.
- when an object is created through deserialization, its reference count is the number of counted references that exist to the object.

The initial reference count of 1 ensures the object is not destroyed immediately. It allows it by calling `release()` - thereafter it will be destroyed when no more references to it exist.

For example, if you create a shape using `PxPhysics::createShape()` and attach it to an actor with `PxRigidActor::attachShape()`, it has a reference count of 2. When you call the shape's `release()` method, it has a reference count of 1. When the actor is destroyed and the shape is detached from the actor, the reference count is decremented to now 0, the shape is destroyed.

The `acquireReference()` method increments the reference count. For example, when a spatial query returns a reference to a mesh shape, and that result is used by another thread for deferred processing, incrementing the reference count ensures that even if the shape referencing the mesh is released, the mesh still exists.

Note: subtypes of `PxGeometry` do not have counted references to the objects they point to, e.g. when `PxConvexMeshGeometry` points to a `PxConvexMesh`, a reference exists only when the geometry is within a `PxShape`.

Note: shapes are often created using the utility method `PxRigidActorExt::createExclusiveShape()`. Take special care when deserializing actors (see *Shapes* and *Reference Counting of Deserialized Objects*)

Using Different Units

PhysX is designed to produce correct results regardless of the units of long as inputs use those units consistently. However, there are certain whose defaults need to be adjusted depending on the units. In order to tolerances default to reasonable values, adjust the values in PxTolerancesScale when creating the PxPhysics and PxCooking interfaces. Tolerances for creation time, and may then be overridden by the application.

You should set tolerances based on the typical size of objects in your example, if you are working with objects of size approximately one meter centimeters, you should set the scale as follows:

```
PxFoundation* foundation = ...;
PxTolerancesScale scale;
scale.length = 100;           // typical length of an object
scale.speed = 981;           // typical speed of an object, gravity
PxPhysics *p = PxCreatePhysics(PX_PHYSICS_VERSION, *foundation, s
```

This will result in the defaults for values like PxShape::contactDist appropriately for your objects.

You can also set the typical object mass in PxTolerancesScale.

It is important to use the same PxTolerances value for initialization PxPhysics, and also when creating PxSceneDesc objects.

Assertions

PhysX uses the `PX_DEBUG` macro to enable or disable assertions. It is defined in the `PhysXCore` and `PhysXCommon` libraries, and so by default the assertions are enabled, however you may configure the libraries to disable them. When an assert is triggered, PhysX calls an assert handler. By default, the handler will trigger a debug breakpoint. However, you may use `PxSetAssertHandler()` to customize the assert handler.

Determinism

PhysX is deterministic in the sense it will produce identical simulation same sequence of API calls applied from the point where a scene is (and the same responses from simulation callbacks which modify removing all the objects from a scene is not in general sufficient to r purpose.

PhysX simulation behavior is not sensitive to the number of CPU worke

An important caveat to determinism is the state of the x87 FPU o platforms. Some compilers produce x87 floating point instructions eve to prefer SSE instructions, and the results of those operations may dep the x87 control word. Since it is too expensive to modify the x87 FPU st entry point, this is delegated to the application if necessary. PhysX oper in changes to the x87 control word, but certain other libraries (inclu modify it.

Configurations in which this is known to be a issue are all 32 configurations, and all MSVC 32-bit checked, release and profile cor Visual Studio 2012.

Introduction

The first step in using the PhysX SDK in a program is the initialization of PhysX objects. These objects can be released when PhysX is no longer needed. This chapter describes how to do this.

Foundation and Physics

First, in some startup code, create a *PxFoundation* object:

```
static PxDefaultErrorCallback gDefaultErrorCallback;
static PxDefaultAllocator gDefaultAllocatorCallback;

mFoundation = PxCreateFoundation(PX_FOUNDATION_VERSION, gDefaultA
    gDefaultErrorCallback);
if(!mFoundation)
    fatalError("PxCreateFoundation failed!");
```

Every PhysX module requires a *PxFoundation* instance to be available. The parameters are a version ID, an allocator callback and a *PX_PHYSICS_VERSION*, is a macro predefined in our headers to enable for a version mismatch between the headers and the corresponding SD

Usually, the allocator callback and error callback are specific to the application. PhysX provides default implementations that make it easy to get started. See [Memory Management](#) and [Error Reporting](#) for more details of these callbacks. (The code supports an advanced memory allocator that tracks allocations internally but we have omitted that detail here.)

Now create the top-level *PxPhysics* object:

```
bool recordMemoryAllocations = true;

mPvd = PxCreatePvd(*gFoundation);
PxPvdTransport* transport = PxDefaultPvdSocketTransportCreate(PVD
mPvd->connect(*transport, PxPvdInstrumentationFlag::eALL);

mPhysics = PxCreatePhysics(PX_PHYSICS_VERSION, *mFoundation,
    PxTolerancesScale(), recordMemoryAllocations, mPvd);
if(!mPhysics)
    fatalError("PxCreatePhysics failed!");
```

Again, the version ID has to be passed in. The *PxTolerancesScale* p

easier to author content at different scales and still have PhysX work :
get started simply pass a default object of this type. The record
parameter specifies whether to perform memory profiling. The optic
enables the debugging and profiling with the PhysX Visual Debugger.

Cooking

The PhysX cooking library provides utilities for creating, converting, and saving cooking data. Depending on your application, you may wish to link to the cooking process such data at runtime. Alternatively you may be able to precompute the data in advance and just load it into memory as required. Initialize the cooking library as follows:

```
mCooking = PxCreateCooking(PX_PHYSICS_VERSION, *mFoundation, PxCo
if (!mCooking)
    fatalError("PxCreateCooking failed!");
```

The `PxCookingParams` struct configures the cooking library to target a specific platform, use non-default tolerances or produce optional outputs. It is important to use the `PxTolerancesScale` values everywhere in your application (see [Using PxTolerancesScale](#) for more details).

The cooking library generates data through a streaming interface. Implementations of streams are provided in the `PxToolkit` library to read from files and memory buffers. Heightfield or Trianglemesh cooked mesh can be inserted into `PxPhysics` without serialization using the `PxPhysicsInsertionCallback` default callback must be used and can be obtained from `PxPhysics::getPhysicsInsertionCallback()`.

Extensions

The extensions library contains many functions that may be useful to users, but which some users may prefer to omit from their application for various reasons or to avoid use of certain subsystems, such as those pertaining to visualization. Initializing the extensions library requires the PxPhysics object:

```
if (!PxInitExtensions(*mPhysics, mPvd))
    fatalError("PxInitExtensions failed!");
```

Optional SDK Components

When linking PhysX as a static library on memory constrained platform avoid linking the code of some PhysX features that are not always used in memory. Currently the optional features are:

- Articulations
- Height Fields
- Cloth
- Particles

If your application requires a subset of this functionality, it is recommended to use `PxCreateBasePhysics` as opposed to `PxCreatePhysics` and then manually register the components you require. Below is an example that registers some of the

```
physx::PxPhysics* customCreatePhysics(physx::PxU32 version,
    physx::PxFoundation& foundation,
    const physx::PxTolerancesScale& scale,
    bool trackOutstandingAllocations
    physx::PxPvd* pvd)
{
    physx::PxPhysics* physics = PxCreateBasePhysics(version, foundation, scale, trackOutstandingAllocations, pvd);

    if(!physics)
        return NULL;

    PxRegisterArticulations(*physics);
    PxRegisterHeightFields(*physics);

    return physics;
}
```

Note that this will only save memory when linking PhysX as a static library. You must also instruct the linker to strip out the unused code.

Delay-Loading DLLs

The PhysXCommon DLL, PxFoundation DLL and PxPvdSDK DLL are loaded inside of the PhysX, PhysXCooking, PhysXCommon and PxPvd. It is possible to have delay-loaded PxFoundation, PxPvdSDK, PhysXCommon and PhysXCooking DLLs.

PhysXCommon DLL and PsFoundation DLL load

The application links against PhysXCommon DLL, and will usually load PxPvdSDK and PhysXCommon.dll before any other PhysX DLL. The application must be the same one that will be used by the PhysX DLLs. In the PhysX and PhysXCooking DLLs, the choice of PxFoundation and PxPvdSDK is made as follows:

- If delay load hook is specified the PhysXCommon name, and PxPvdSDK name provided by user is used
- If delay load hook is not specified, the corresponding PhysXCooking PsFoundation DLL or PxPvdSDK DLL is used

PxDelayLoadHook

The PxDelayLoadHook class supports loading of different versions of PhysX DLL, PxFoundation DLL or PxPvdSDK DLL. This can be achieved by providing DLL names to the PhysX SDK through a custom subclass of PxDelayLoadHook following example:

```
class SampleDelayLoadHook: public PxDelayLoadHook
{
    virtual const char* getPhysXCommonDEBUGDllName() const
        { return "PhysX3CommonDEBUG_x64_Test.dll"; }
    virtual const char* getPhysXCommonCHECKEDDllName() const
        { return "PhysX3CommonCHECKED_x64_Test.dll"; }
    virtual const char* getPhysXCommonPROFILEDllName() const
        { return "PhysX3CommonPROFILE_x64_Test.dll"; }
```

```

virtual const char* getPhysXCommonDllName() const
    { return "PhysX3Common_x64_Test.dll"; }
virtual const char* getPxFoundationDEBUGDllName() const
    { return "PxFoundationDEBUG_x64_Test.dll"; }
virtual const char* getPxFoundationCHECKEDDllName() const
    { return "PxFoundationCHECKED_x64_Test.dll"; }
virtual const char* getPxFoundationPROFILEDllName() const
    { return "PxFoundationPROFILE_x64_Test.dll"; }
virtual const char* getPxFoundationDllName() const
    { return "PxFoundation_x64_Test.dll"; }
virtual const char* getPxPvdSDKDEBUGDllName() const
    { return "PxPvdSDKDEBUG_x64_Test.dll"; }
virtual const char* getPxPvdSDKCHECKEDDllName() const
    { return "PxPvdSDKCHECKED_x64_Test.dll"; }
virtual const char* getPxPvdSDKPROFILEDllName() const
    { return "PxPvdSDKPROFILE_x64_Test.dll"; }
virtual const char* getPxPvdSDKDllName() const
    { return "PxPvdSDK_x64_Test.dll"; }
} gDelayLoadHook;

```

Now the hook must be set for PhysX, PhysXCooking, PhysXGpuPxPvdSDK:

```

PxSetPhysXDelayLoadHook(&gDelayLoadHook);
PxSetPhysXCookingDelayLoadHook(&gDelayLoadHook);
PxSetPhysXGpuDelayLoadHook(&gDelayLoadHook);
PxSetPhysXCommonDelayLoadHook(&gDelayLoadHook);
PxPvdSetFoundationDelayLoadHook(&gDelayLoadHook);

```

PxGpuLoadHook

The PxGpuLoadHook class supports loading of different versions of Pfr can be achieved by providing different DLL names to the PhysX SDK subclass of PxGpuLoadHook, see the following example:

```

class SampleGpuLoadHook: public PxGpuLoadHook
{
    virtual const char* getPhysXGpuDEBUGDllName() const
        { return "PhysX3GpuDEBUG_x64_Test.dll"; }
    virtual const char* getPhysXGpuCHECKEDDllName() const
        { return "PhysX3GpuCHECKED_x64_Test.dll"; }
    virtual const char* getPhysXGpuPROFILEDllName() const

```

```
        { return "PhysX3GpuPROFILE_x64_Test.dll"; }  
    virtual const char* getPhysXGpuDllName() const  
        { return "PhysX3Gpu_x64_Test.dll"; }  
} gGpuLoadHook;
```

Now the hook must be set for PhysX:

```
PxSetPhysXGpuLoadHook(&gGpuLoadHook);
```

PhysXCommon Secure Load

All PhysX DLLs distributed by NVIDIA are signed. The PhysXCommon is checked, when it is loaded by PhysX or PhysXCooking. If signature application is terminated.

Shutting Down

To dispose of any PhysX object, call its `release()` method. This will destroy all contained objects. The precise behavior depends on the object type; refer to the reference guide for details. To shut down the extension function `PxCloseExtensions()`. To shut down physics, call `release()` on the `mPhysics` object, and this will clean up all of the physics objects:

```
mPhysics->release();
```

Do not forget to release the foundation object as well, but only after all modules have been released:

```
mFoundation->release();
```


Introduction

This chapter explains how to use PhysX in multithreaded application main aspects to using PhysX with multiple threads:

- how to make read and write calls into the PhysX API from multiple threads, causing race conditions.
- how to use multiple threads to accelerate simulation processing.
- how to perform asynchronous simulation, and read and write simulation is being processed.

Data Access from Multiple Threads

For efficiency reasons, PhysX does not internally lock access to its data. Your application, so be careful when calling the API from multiple application threads. The rules are as follows:

- API interface methods marked 'const' are read calls, other API interface methods are write calls.
- API read calls may be made simultaneously from multiple threads.
- Objects in different scenes may be safely accessed by different threads.
- Different objects outside a scene may be safely accessed from different threads. Be aware that accessing an object may indirectly cause access to a persistent reference (such as joints and actors referencing one another, or a shape referencing a mesh.)

Access patterns which do not conform to the above rules may result in deadlocks, or crashes. Note in particular that it is not legal to perform a write operation to an object in a scene concurrently with a read operation to an object in the same scene. The checked build contains code which tracks access by application thread within a scene, to try and detect problems at the point when the illegal access occurs.

Scene Locking

Each PxScene object provides a multiple reader, single writer lock to control access to the scene by multiple threads. This is useful for situations where a PhysX scene is shared between more than one system, for example a game engine's physics code. The scene lock provides a way for these systems to communicate with each other.

It is not mandatory to use the lock. If all access to the scene is from a single thread, the lock adds unnecessary overhead. Even if you are accessing the scene from multiple threads, you may be able to synchronize the threads using a simple application-specific mechanism that guarantees your application's access to the scene.

conditions. However, using the scene lock has two potential benefits:

- If the `PxSceneFlag::eREQUIRE_RW_LOCK` is set, the checked warning for any API call made without first acquiring the lock, or if when the lock has only been acquired for read,
- The APEX SDK uses the scene lock to ensure that it shares the your application.

There are four methods for for acquiring / releasing the lock:

```
void PxScene::lockRead(const char* file=NULL, PxU32 line=0);
void PxScene::unlockRead();

void PxScene::lockWrite(const char* file=NULL, PxU32 line=0);
void PxScene::unlockWrite();
```

Additionally there is an RAII helper class to manage these locks, see [PxSceneLock](#)

Locking Semantics

There are precise rules regarding the usage of the scene lock:

- Multiple threads may read at the same time.
- Only one thread may write at a time, no thread may write if any thread has a write lock.
- If a thread holds a write lock then it may call both read and write APIs.
- Re-entrant read locks are supported, meaning a `lockRead()` on already acquired a read lock is permitted. Each `lockRead()` must be followed by a `unlockRead()`.
- Re-entrant write locks are supported, meaning a `lockWrite()` on already acquired a write lock is permitted. Each `lockWrite()` must be followed by a `unlockWrite()`.
- Calling `lockRead()` by a thread that has already acquired the write lock is permitted. The thread will continue to have read and write access. Each lock must be followed by an associated `unlock*()` that occurs in reverse order.

- Lock upgrading is *not* supported - a *lockWrite()* by a thread that has a read lock is *not* permitted. Attempting this in checked builds will result in a runtime error, while in release builds it will lead to deadlock.
- Writers are favored - if a thread attempts a *lockWrite()* while the resource is held by a reader, it will be blocked until all readers leave. If new readers arrive while blocked they will be put to sleep and the writer will have first chance to acquire the lock. This prevents writers being starved in the presence of multiple readers.
- If multiple writers are queued then the first writer will receive priority and subsequent writers will be granted access according to OS scheduling.

Note: *PxScene::release()* automatically attempts to acquire the write lock. It is not necessary to acquire it manually before calling *release()*.

Locking Best Practices

It is often useful to arrange your application to acquire the lock a single time for multiple operations. This minimizes the overhead of the lock, and in cases such as a sweep test in one thread seeing a rag doll that has been inserted by another thread.

Clustering writes can also help reduce contention for the lock, as a single write will stall any other thread trying to perform a read access.

Asynchronous Simulation

PhysX simulation is asynchronous by default. Start simulation by calling

```
scene->simulate(dt);
```

When this call returns, the simulation step has begun in a separate thread. While the simulation is running, you can still make calls into the API. When the simulation state is updated, the results will be buffered and reconciled with the current state when the simulation step completes.

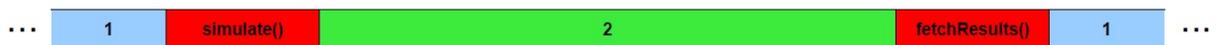
To wait until simulation completes, call:

```
scene->fetchResults(true);
```

The boolean parameter to `fetchResults` denotes whether the call should wait for the simulation to complete, or return immediately with the current completion state. See the [documentation](#) for more detail.

It is important to distinguish two time slots for data access:

1. After the call to `PxScene::fetchResults()` has returned and before the `PxScene::simulate()` call (see figure below, blue area "1").
2. After the call to `PxScene::simulate()` has returned and before the `PxScene::fetchResults()` call (see figure below, green area "2").



In the first time slot, the simulation is not running and there are no reads or writes to object properties. Changes to the position of an object, for example, are made instantaneously and the next scene query or simulation step will take account of them.

In the second time slot the simulation is running and in the process, reads

the state of objects. Concurrent access from the user might corrupt the or lead to data races or inconsistent views in the simulation code. The user's view of the objects is protected from API writes, and any attribute updates are buffered to allow API reads. The consequences will be discussed in the next section.

Note that *simulate()* and *fetchResults()* are write calls on the scene, and it is illegal to access any object in the scene while these functions are running.

Double Buffering

While a simulation is running, PhysX supports read and write access to the scene (with some exceptions, see further below). This includes adding and removing objects to/from a scene.

From the user perspective, API changes are reflected immediately. For example, if the velocity of a rigid body is set and then queried, the new velocity will be returned. However, if an object is created while the simulation is running, it cannot be accessed until the next simulation step. Similarly, if an object is destroyed while the simulation is running, it cannot be accessed until the next simulation step. However, these changes are buffered so that the simulation always sees the object state as it was when *PxScene::simulate()* was called. For instance, if a user changes the filter data of an object while the simulation is running, the changes are ignored until the next simulation step. Similarly, if a user generates a new object during the running step, it will only be visible for the next simulation step.

When *PxScene::fetchResults()* is called, any buffered changes are flushed to the simulation and are reflected in the API view of the objects, and API queries are visible to the simulation code for the next step. User changes take effect immediately. For example, a user change to the position of an object while the simulation is running will overwrite the position which resulted from the simulation.

The delayed application of updates does not affect scene queries, which always account for the latest changes.

Events involving removed objects

Deleting objects or removing them from the scene while the simulation is running is allowed, but the objects will not be visible to the simulation until the next simulation step.

affect the simulation events sent out at *PxScene::fetchResults()*. This follows:

- *PxSimulationEventCallback::onWake()*, *::onSleep()* events will not be sent if an object is involved which got deleted/removed during the running simulation.
- *PxSimulationEventCallback::onContact()*, *::onTrigger()* events will not be sent if an object is involved which got deleted/removed during the running simulation. Deleted/removed objects will be marked as *PxContactPairHeaderFlag::eREMOVED_ACTOR_0*, *PxContactPairFlag::eREMOVED_SHAPE_0*, *PxTriggerPairFlag::eREMOVED_SHAPE_TRIGGER*, *PxTriggerPairFlag::eREMOVED_SHAPE_TRIGGERED*, *PxPairFlag::eNOTIFY_TOUCH_LOST*, *::eNOTIFY_THRESHOLD* events were requested for the pair containing the deleted/removed object. Further events will be created.

Support

Not all PhysX objects have full buffering support. Operations which cannot be performed while simulation is in process are mentioned in the API documentation and their return codes. The most important exceptions are as follows:

- Particles: The particle bulk data can not be read or modified while simulation is running, this includes operations like reading/writing particle data, creating/deleting particles, adding forces, etc.
- Cloth: The only allowed double buffered operation is to create/add/remove it to/from the scene.

Memory Considerations

The buffers to store the object changes while the simulation is running are limited by memory demand. If memory usage concerns outweigh the advantage of reading/writing parallel with simulation, do not write to objects while the simulation is running.

Multithreaded Simulation

PhysX includes a task system for managing CPU and GPU compute requests created with dependencies so that they are resolved in a given order, which are then submitted to a user-implemented dispatcher for execution.

Middleware products typically do not want to create CPU threads for themselves, especially true on consoles where execution threads can have significant overhead. In a task model, the computational work is broken into jobs that are submitted to a thread pool as they become ready to run.

The following classes comprise the CPU task management.

TaskManager

A TaskManager manages inter-task dependencies and dispatches requests to its respective dispatcher. There is a dispatcher for CPU tasks and GPU tasks. TaskManager.

TaskManagers are owned and created by the SDK. Each PxScene has a TaskManager instance which users can configure with dispatchers through PxSceneDesc or directly through the TaskManager interface.

CpuDispatcher

The CpuDispatcher is an abstract class the SDK uses for interfacing with a thread pool. Typically, there will be one single CpuDispatcher for the application since there is rarely a need for more than one thread pool. A CpuDispatcher can be shared by more than one TaskManager, for example if multiple scenes are simulated.

PhysX includes a default CpuDispatcher implementation, but we prefer to let users implement this class themselves so PhysX and APEX can efficiently share the thread pool with the application.

Note: The TaskManager will call `CpuDispatcher::submitTask()` from either API calls (aka: `scene::simulate()`) or from other running tasks, so the function is thread-safe.

An implementation of the `CpuDispatcher` interface must call the following for each submitted task for it to be run correctly:

```
baseTask->run();    // optionally call runProfiled() to wrap with
baseTask->release();
```

The `PxExtensions` library has default implementations for all dispatchers. The following code snippets are taken from `SampleParticles` and `SampleBodies` where the default dispatchers are created. `mNbThreads` which is defined in `PxDefaultCpuDispatcherCreate` defines how many worker threads the dispatcher has.

```
PxSceneDesc sceneDesc(mPhysics->getTolerancesScale());
[...]
// create CPU dispatcher which mNbThreads worker threads
mCpuDispatcher = PxDefaultCpuDispatcherCreate(mNbThreads);
if(!mCpuDispatcher)
    fatalError("PxDefaultCpuDispatcherCreate failed!");
sceneDesc.cpuDispatcher = mCpuDispatcher;
#ifdef PX_WINDOWS
// create GPU dispatcher
PxCudaContextManagerDesc cudaContextManagerDesc;
mCudaContextManager = PxCreateCudaContextManager(cudaContextManagerDesc,
sceneDesc.gpuDispatcher = mCudaContextManager->getGpuDispatcher();
#endif
[...]
mScene = mPhysics->createScene(sceneDesc);
```

Note: Best performance is usually achieved if the number of threads is equal to the available hardware threads of the platform you are running on. More worker threads than hardware threads will often lead to worse performance. On platforms with a single execution core, the CPU dispatcher can be created with 0 worker threads (`PxDefaultCpuDispatcherCreate(0)`). In this case all work is executed on the thread that calls `PxScene::simulate()`, which can be modified to use a thread pool.

using multiple threads.

Note: CudaContextManagerDesc support appGUID now. It only work build. If your application employs PhysX modules that use CUDA you r GUID so that patches for new architectures can be released for your g; obtain a GUID for your application from NVIDIA. The application should into a file which can be sent to NVIDIA for support.

CpuDispatcher Implementation Guidelines

After the scene's TaskManager has found a ready-to-run task and appropriate dispatcher it is up to the dispatcher implementation to decide the task will be run.

Often in game scenarios the rigid body simulation is time critical and the latency from simulate() to the completion of fetchResults(). The lowest will be achieved when the PhysX tasks have exclusive access to CPU the update. In reality, PhysX will have to share compute resources with Below are some guidelines to help ensure a balance between throughput when mixing the PhysX update with other work.

- Avoid interleaving long running tasks with PhysX tasks, this will help
- Avoid assigning worker threads to the same execution core as high If a PhysX task is context switched during execution the rest of the may be stalled, increasing latency.
- PhysX occasionally submits tasks and then immediately waits for because of this, executing tasks in LIFO (stack) order may perform (queue) order.
- PhysX is not a perfectly parallel SDK, so interleaving small to r tasks will generally result in higher overall throughput.
- If your thread pool has per-thread job-queues then queuing tasks were submitted may result in more optimal CPU cache coherence,

required.

For more details see the default `CpuDispatcher` implementation that comes with the `PxExtensions` package. It uses worker threads that each have their own queue and can steal tasks from the back of other worker's queues (LIFO order) to improve distribution.

BaseTask

`BaseTask` is the abstract base class for all task types. All tasks are run on application threads, so they need to be careful with their own little stack as possible, and they should never block for any reason.

Task

The `Task` class is the standard task type. Tasks must be submitted to the `TaskManager` at each simulation step for them to be executed. Tasks may be named and this allows them to be discoverable. Tasks will be given a reference count when they are submitted, and the `TaskManager::startSimulation()` function decrements the reference count of all tasks and dispatches all `Tasks` whose reference count reaches zero. After `TaskManager::startSimulation()` is called, `Tasks` can set dependencies to control the order in which they are dispatched. Once simulation has started, it is possible to submit new tasks and add dependencies, but it is up to the user to avoid race hazards. You cannot add dependencies to tasks that have already been dispatched, and newly submitted `Tasks` must have their reference count set to zero before that `Task` will be allowed to execute.

Synchronization points can also be defined using `Task` names. The user can assign the name a `TaskID` with no `Task` implementation. When all of the tasks with dependencies are met, it will decrement the reference count of all `Tasks` with that name.

APEX uses the `Task` class almost exclusively to manage CPU resource allocation. It defines a number of named `Tasks` that the modules use to schedule their work. For example, the `PhysX` scene starts after LOD calculations are complete, finish before the `PhysX` scene is simulated.

LightCpuTask

LightCpuTask is another subclass of BaseTask that is explicitly programmer. LightCpuTasks have a reference count of 1 when they are created. The reference count must be decremented before they are dispatched. When a task increments their continuation task reference count when they are initialized, the reference count will be decremented when they are released (after completing their run()).

PhysX 3.x uses LightCpuTasks almost exclusively to manage CPU tasks. For example, each stage of the simulation update may consist of multiple parallel tasks. When each of these tasks has finished execution it will decrement the reference count of the next task in the update chain. This will then be automatically dispatched when its reference count reaches zero.

Note: Even when using LightCpuTasks exclusively to manage CPU tasks, TaskManager startSimulation() and stopSimulation() calls must be made as a first step to keep the GpuDispatcher synchronized.

The following code snippets show how the crabs' A.I. in SampleSubmarine is implemented as a CPU Task. By doing so the Crab A.I. is run as a background Task during the PhysX simulation update.

For a CPU task that does not need handling of multiple continuations, you can subclass LightCpuTask. A LightCpuTask subclass requires that the getName and run methods be defined:

```
class Crab: public ClassType, public physx::PxLightCpuTask, public BaseTask
{
public:
    Crab(SampleSubmarine& sample, const PxVec3& crabPos, RenderManager* mgr) : mSample(sample), mCrabPos(crabPos), mMgr(mgr) {}
    ~Crab();
    [...]

    // Implements LightCpuTask
    virtual const char* getName() const { return "Crab AI Task"; }
    virtual void run();

    [...]
}
```

```
}
```

After `PxScene::simulate()` has been called, and the simulation started, `removeReference()` on each Crab task, this in turn causes it to be `CpuDispatcher` for update. Note that it is also possible to submit task directly (without manipulating reference counts) as follows:

```
PxLightCpuTask& task = &mCrab;  
mCpuDispatcher->submitTask(task);
```

Once queued for execution by the `CpuDispatcher`, one of the thread pool will eventually call the task's run method. In this example the Crab raycasts against the scene and update its internal state machine:

```
void Crab::run()  
{  
    // run as a separate task/thread  
    scanForObstacles();  
    updateState();  
}
```

It is safe to perform API read calls, such as scene queries, from multiple threads while `simulate()` is running. However, care must be taken not to overlap API read and write calls from multiple threads. In this case the SDK will issue an error, see [7.1.1.1](#) for more information.

An example for explicit reference count modification and task dependencies is provided below:

```
// assume all tasks have a refcount of 1 and are submitted to the  
// 3 task chains a0-a2, b0-b2, c0-c2  
// b0 shall start after a1  
// the a and c chain have no dependencies and shall run in parallel  
//  
// a0-a1-a2  
//      \  
//      b0-b1-b2  
// c0-c1-c2  
  
// setup the 3 chains
```

```
for(PxU32 i = 0; i < 2; i++)
{
    a[i].setContinuation(&a[i+1]);
    b[i].setContinuation(&b[i+1]);
    c[i].setContinuation(&c[i+1]);
}

// b0 shall start after a1
b[0].startAfter(a[1].getTaskID());

// setup is done, now start all task by decrementing their refcou
// tasks with refcount == 0 will be submitted to the dispatcher (
for(PxU32 i = 0; i < 3; i++)
{
    a[i].removeReference();
    b[i].removeReference();
    c[i].removeReference();
}
```


Introduction

This section discusses the PhysX geometry classes. Geometries are used for rigid bodies, as collision triggers, and as volumes in PhysX's solver. PhysX also provides standalone functions for testing intersection by raycasting against them, and sweeping one geometry against another.

Geometries are value types, and inherit from a common base class, `PxGeometry`. Each geometry class defines a volume or surface with a fixed position and transform. The transform specifies the frame in which the geometry is interpreted. For each geometry type PhysX provides helper functions to construct these geometries from common alternative representations.

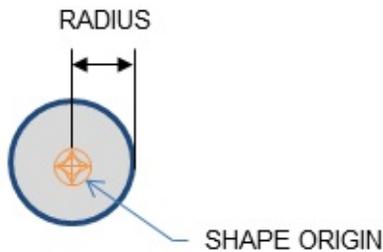
Geometries fall into two classes:

- primitives (`PxBoxGeometry`, `PxSphereGeometry`, `PxCylinderGeometry`, `PxPlaneGeometry`) where the geometry object contains all of the data.
- meshes or height fields (`PxConvexMeshGeometry`, `PxTriangleMeshGeometry`, `PxHeightFieldGeometry`), where the geometry object contains a reference to a larger object (`PxConvexMesh`, `PxTriangleMesh`, `PxHeightField`). These objects use these objects with different scales in each `PxGeometry` type. The larger objects must be created using a *cooking* process, described in the next section.

When passed into and out of the SDK for use as simulation geometry, it is copied into and out of a `PxShape` class. It can be awkward in this case to use a geometry without knowing its type, so PhysX provides a union-like class (`PxGeometryHolder`) that can be used to pass any geometry type by value. The `PxHeightField` has a reference count that tracks the number of `PxShapes` that reference the mesh.

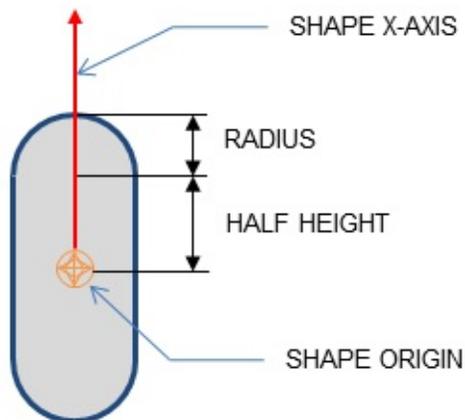
Geometry Types

Spheres



A `PxSphereGeometry` is specified by one attribute, its radius, and is centered at the origin.

Capsules



A `PxCapsuleGeometry` is centered at the origin. It is specified by a radius value by which its axis extends along the positive and negative X-axis.

To create a dynamic actor whose geometry is a capsule standing upright a relative transform that rotates it around the Z-axis by a quarter-circle capsule will extend along the Y-axis of the actor instead of the X-axis and actor is otherwise the same as for the sphere:

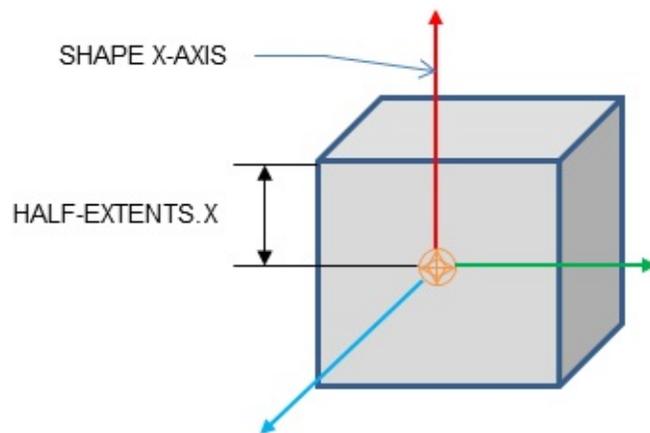
```

PxRigidBody* aCapsuleActor = thePhysics->createRigidBody(Px
PxTransform relativePose(PxQuat(PxHalfPi, PxVec(0,0,1)));
PxShape* aCapsuleShape = PxRigidBodyExt::createExclusiveShape(*a
    PxCapsuleGeometry(radius, halfHeight), aMaterial);
aCapsuleShape->setLocalPose(relativePose);
PxRigidBodyExt::updateMassAndInertia(*aCapsuleActor, capsuleDensi
aScene->addActor(aCapsuleActor);

```

The function `PxTransformFromSegment()` converts from a line segment capsule axis to a transform and halfheight.

Boxes



A `PxBoxGeometry` has three attributes, the three extents halved:

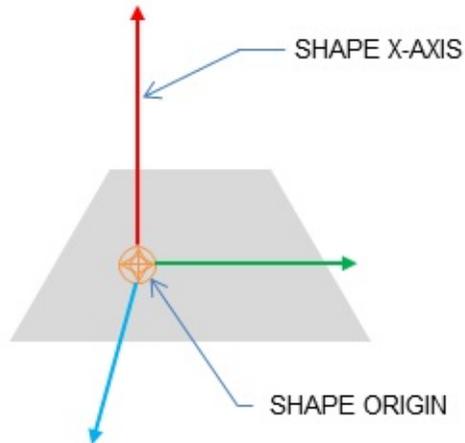
```

PxShape* aBoxShape = PxRigidBodyExt::createExclusiveShape(*aBoxA
    PxBoxGeometry(a/2, b/2, c/2), aMaterial);

```

Where a , b and c are the side lengths of the resulting box.

Planes



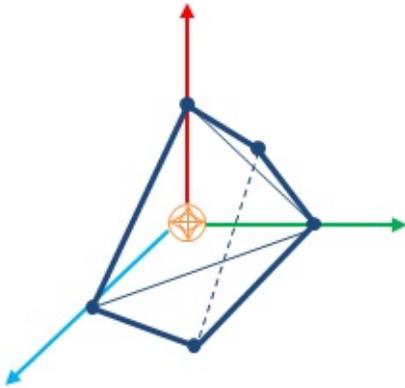
Planes divide space into "above" and "below" them. Everything "below" them will collide with it.

The Plane lies on the YZ plane with "above" pointing towards positive X. To convert a plane equation to an equivalent transform, use `PxTransformFromPlaneEquation()`. To convert a transform to a plane equation, use `PxPlaneEquationFromTransform()`.

A `PxPlaneGeometry` has no attributes, since the shape's pose entirely defines its collision volume.

Shapes with a `PxPlaneGeometry` may only be created for static actors.

Convex Meshes



A shape is convex if, given any two points within the shape, the shape between them. A `PxConvexMesh` is a convex polyhedron represented by vertices and polygonal faces. The number of vertices and faces of a convex mesh is limited to 255.

Creating a `PxConvexMesh` requires cooking. It is assumed here that the cooking library has already been initialized (see [Startup and Shutdown](#).) The following code shows how to create a simple square pyramid.

First, define the vertices of the convex object:

```
static const PxVec3 convexVerts[] = {PxVec3(0,1,0),PxVec3(1,0,0),
    PxVec3(0,0,-1)};
```

Then construct a description of the convex data layout:

```
PxConvexMeshDesc convexDesc;
convexDesc.points.count      = 5;
convexDesc.points.stride    = sizeof(PxVec3);
convexDesc.points.data      = convexVerts;
convexDesc.flags             = PxConvexFlag::eCOMPUTE_CONVEX;
```

Now use the cooking library to construct a `PxConvexMesh`:

```
PxDefaultMemoryOutputStream buf;
PxConvexMeshCookingResult::Enum result;
if(!cooking.cookConvexMesh(convexDesc, buf, &result))
```

```

return NULL;
PxDefaultMemoryInputData input(buf.getData(), buf.getSize());
PxConvexMesh* convexMesh = physics->createConvexMesh(input);

```

Finally, create a shape using a `PxConvexMeshGeometry` which instanc

```

PxShape* aConvexShape = PxRigidActorExt::createExclusiveShape(*aC
PxConvexMeshGeometry(convexMesh), aMaterial);

```

Alternatively the `PxConvexMesh` can be cooked and directly insert without stream serialization. This is useful if real-time cooking is req recommended to use offline cooking and streams. Here is an examp improve cooking speed if needed:

```

PxConvexMeshDesc convexDesc;
convexDesc.points.count      = 5;
convexDesc.points.stride    = sizeof(PxVec3);
convexDesc.points.data      = convexVerts;
convexDesc.flags            = PxConvexFlag::eCOMPUTE_CONVEX | PxC

#ifdef _DEBUG
    // mesh should be validated before cooking without the mesh c
    bool res = theCooking->validateConvexMesh(convexDesc);
    PX_ASSERT(res);
#endif

PxConvexMesh* aConvexMesh = theCooking->createConvexMesh(convexDe
thePhysics->getPhysicsInsertionCallback());

```

Please note that mesh validation is required for debug and checked meshes from unvalidated input descriptors may result in undefined I `PxConvexFlag::eFAST_INERTIA_COMPUTATION` flag the volume in SIMD code path which does faster computation but with lesser precision

The user can optionally provide a per-instance `PxMe: PxConvexMeshGeometry`. The scale defaults to identity. Negative sca for convex meshes.

`PxConvexMeshGeometry` also contains flags to tweak some aspects of

By default the system computes approximate (loose) bounds around Using `PxConvexMeshGeometryFlag::eTIGHT_BOUNDS` enables shapes which are more expensive to compute but could result in improved simulation when a lot of convex objects are interacting with each other.

`PxConvexMeshGeometry` also contains a variable called `maxMargin`. By default it is `3.4e38f`. If the `maxMargin` is smaller than the margin amount calculated by the contact generation, it will choose the smallest margin for the shrunk shape to perform the GJK algorithm. In this case, the application might notice some artifacts from the vertex collision. If the `maxMargin` is set to be a small value, the visibility of these artifacts is reduced. If `maxMargin` is set to zero, PCM will use the GJK algorithm. This will result in no artifacts for this approach. It is a trade off between performance and accuracy.

Convex Mesh cooking

Convex Mesh cooking transforms the mesh data into a form which performs efficient collision detection. The input to cooking is defined by `PxConvexMeshDesc`.

There are different ways to fill in this structure, depending on whether you are starting with a convex mesh starting from just a cloud of vertices, or whether you have the faces of a polyhedron already.

If Only Vertex Points are Provided

When providing only vertices, set the `PxConvexFlag::eCOMPUTE_MESH` to compute the mesh:

```
PxConvexMeshDesc convexDesc;
convexDesc.points.count      = 20;
convexDesc.points.stride    = sizeof(PxVec3);
convexDesc.points.data      = convexVerts;
convexDesc.flags             = PxConvexFlag::eCOMPUTE_MESH;
convexDesc.maxVerts         = 10;

PxDefaultMemoryOutputStream buf;
```

```
if(!cooking.cookConvexMesh(convexDesc, buf))
    return NULL;
```

The algorithm tries to create a convex mesh from the source \ convexDesc.vertexLimit specifies the limit for the maximum number resulting hull.

This routine can sometimes fail when the source data is geometrica example if it contains a lot of vertices close to each-other. If cookin reported to the error stream and the routine returns false.

If PxConvexFlag::eCHECK_ZERO_AREA_TRIANGLES is used, the ; include triangles with an area less than PxCookingParams::area algorithm cannot find 4 initial vertices without a PxConvexMeshCookingResult::eZERO_AREA_TEST_FAILED is retu that the provided vertices were in a very small area and the cooker c valid hull. The toolkit helper function PxToolkit::createConvexMeshSafe robust strategy for convex mesh cooking. First it tries to create the hull that fails it tries inflation, and if that also fails, uses an AABB or OBB.

It is recommended to provide vertices around origin and put transform otherwise additional PxConvexFlag::eSHIFT_VERTICES flag for the mes

If huge amount of input vertices are provided, it might be useful to vertices, in this case use PxConvexFlag::eQUANTIZE_INPUT and PxConvexMeshDesc::quantizedCount.

Convex cooking supports two different algorithms:

Quickhull Algorithm

This algorithm does not use inflation. It creates a convex mesh wh subset of the original vertices, and the number of vertices is guarante than the specified maximum.

The Quickhull algorithm performs these steps:

- Cleans the vertices - removes duplicates etc.
- Finds a subset of vertices, no more than vertexLimit, that enclose the input vertices.
- If the vertexLimit is reached, expand the limited hull around the input vertices to ensure we encapsulate all the input vertices.
- Compute a vertex map table. (Requires at least 3 neighbor polygons)
- Checks the polygon data - verifies that all vertices are on or inside the hull.
- Computes mass and inertia tensor assuming density is 1.
- Saves data to stream.

When the hull is constructed each new vertex added must be within `PxCookingParams::planeTolerance` from the hull, if not that vertex is dropped.

Inflation Based Incremental Algorithm

This algorithm always uses the `PxConvexFlag::eINFLATE_CONVEX` flag to generate hull planes by `PxCookingParams::skinWidth`.

The Inflation Incremental Algorithm performs these steps:

- Cleans the vertices - removes duplicates etc.
- Finds a subset of vertices, no more than vertexLimit, that enclose the input vertices.
- Creates planes from the produced enclosed hull.
- Inflates planes by defined `PxCookingParams::skinWidth`.
- Crops the AABB by the inflated planes and produces a new hull.
- Computes vertex map table. (Requires at least 3 neighbor polygons)
- Checks polygon data - verifies all vertices are on or inside the hull,
- Computes mass and inertia tensor assuming density is 1.
- Saves data to stream.

Note that the inflation based algorithm can produce hulls with more detail but the algorithm is significantly slower than the quickhull and produces significant artifacts. It is recommended to use the quickhull algorithm.

Vertex Limit Algorithms

If a vertex limit has been provided, there are two algorithms that handle

The default algorithm computes the full hull, and an OBB around the OBB is then sliced with the hull planes until the vertex limit is reached. This algorithm requires the vertex limit to be set to at least 8, and typically produces much better quality than are produced by plane shifting.

When plane shifting is enabled (`PxConvexFlag::ePLANE_SHIFTING`), computation stops when vertex limit is reached. The hull planes are the planes defined by all input vertices, and the new plane intersection points are then used to compute the hull with the given vertex limit. Plane shifting may produce sharp edges away from the input cloud, and does not guarantee that all input vertices are on the resulting hull. However, it can be used with a vertex limit as low as 4. Plane shifting is a better choice for cases such as small pieces of debris with very low vertex counts.

Vertex Points, Indices and Polygons are Provided

To create a `PxConvexMesh` given a set of input vertices (`convexVerts`) and hull polygons (`hullPolygons`):

```
PxConvexMeshDesc convexDesc;
convexDesc.points.count      = 12;
convexDesc.points.stride    = sizeof(PxVec3);
convexDesc.points.data      = convexVerts;
convexDescPolygons.polygons.count = 20;
convexDescPolygons.polygons.stride = sizeof(PxHullPolygon);
convexDescPolygons.polygons.data = hullPolygons;
convexDesc.flags             = 0;

PxDefaultMemoryOutputStream buf;
if(!cooking.cookConvexMesh(convexDesc, buf))
    return NULL;
```

When points and polygons are provided, the SDK validates the mesh and creates a `PxConvexMesh` directly. This is the fastest way to create a convex mesh. The SDK requires at least 3 neighbor polygons for each vertex. Other

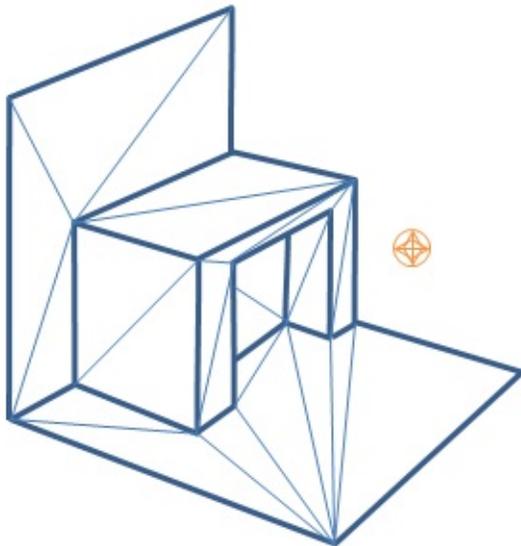
structure for PCM is not created and it does result in performance enabled.

(NOTE: the SDK should reject such a mesh as invalid)

Internal steps during convex cooking:

- Compute vertex map table, requires at least 3 neighbor polygons for
- Check polygons data - check if all vertices are on or inside the hull,
- Compute mass and inertia tensor assuming density 1.
- Save data to stream.

Triangle Meshes



Like graphical triangle meshes, a collision triangle mesh consists of a c and the triangle indices. Triangle mesh creation requires use of the c assumed here that the cooking library has already been initialized *Shutdown*.):

```
PxTriangleMeshDesc meshDesc;  
meshDesc.points.count           = nbVerts;  
meshDesc.points.stride         = sizeof(PxVec3);
```

```

meshDesc.points.data          = verts;

meshDesc.triangles.count      = triCount;
meshDesc.triangles.stride     = 3*sizeof(PxU32);
meshDesc.triangles.data       = indices32;

PxDefaultMemoryOutputStream writeBuffer;
PxTriangleMeshCookingResult::Enum result;
bool status = cooking.cookTriangleMesh(meshDesc, writeBuffer, resu
if(!status)
    return NULL;

PxDefaultMemoryInputData readBuffer(writeBuffer.getData(), writeB
return physics.createTriangleMesh(readBuffer);

```

Alternatively *PxTriangleMesh* can be cooked and directly inserted into stream serialization. This is useful if real-time cooking is required. It is recommended to use offline cooking and streams. Example how to speed it up if needed:

```

PxTolerancesScale scale;
PxCookingParams params(scale);
// disable mesh cleaning - perform mesh validation on development
params.meshPreprocessParams |= PxMeshPreprocessingFlag::eDISABLE_
// disable edge precompute, edges are set for each triangle, slow
params.meshPreprocessParams |= PxMeshPreprocessingFlag::eDISABLE_
// lower hierarchy for internal mesh
params.meshCookingHint = PxMeshCookingHint::eCOOKING_PERFORMANCE;

theCooking->setParams(params);

PxTriangleMeshDesc meshDesc;
meshDesc.points.count          = nbVerts;
meshDesc.points.stride         = sizeof(PxVec3);
meshDesc.points.data           = verts;

meshDesc.triangles.count       = triCount;
meshDesc.triangles.stride      = 3*sizeof(PxU32);
meshDesc.triangles.data        = indices32;

#ifdef _DEBUG
    // mesh should be validated before cooked without the mesh cl
    bool res = theCooking->validateTriangleMesh(meshDesc);
    PX_ASSERT(res);
#endif

```

```
PxTriangleMesh* aTriangleMesh = theCooking->createTriangleMesh(me  
    thePhysics->getPhysicsInsertionCallback());
```

Indices can be 16 or 32 bit. The strides used here assume that vertex arrays of `PxVec3s` and 32bit integers respectively with no gaps in the data.

Returned result enum `PxTriangleMeshCookingResult::eLARGE_TRIANGLES` if the user if the mesh contains large triangles, which should be tessellated for simulation and CCT stability.

Like height fields, triangle meshes support per-triangle material instances. To create triangle materials for a mesh, provide per-triangle indices to the cooking mesh descriptor. Later, when creating the `PxShape`, supply a table of material values in the mesh to material instances.

Triangle Mesh cooking

Triangle mesh cooking proceeds as follows:

- Check validity of input vertices.
- Weld vertices and check triangle sizes.
- create acceleration structure for queries.
- Compute edge convexity information and adjacencies.
- Save data to stream.

Note that mesh cleaning may result in the set of triangles produced being a subset different from the original input set. Mesh cleaning removes triangles (containing out-of-range vertex references), duplicate triangles, and zero-area triangles. When this happens, PhysX optionally outputs a mesh remapping table mapping each internal triangle to its source triangle in the user's data.

There are multiple parameters to control mesh creation.

In `PxTriangleMeshDesc`:

- *materialIndices* defines per triangle materials. When a triangle collides with another object, a material is required at the collision point. If material is not specified, then the material of the PxShape instance is used.

In *PxCookingParams*:

- *scale* defines Tolerance scale is used to check if cooked triangles are overlapping. This check will help with simulation stability.
- *suppressTriangleMeshRemapTable* specifies whether the face remapping table is used. If not, this saves a significant amount of memory, but the SDK does not provide information about which original mesh triangle is hit in collision raycasts hits.
- *buildTriangleAdjacencies* specifies if the triangle adjacency information is built. The adjacent triangles can be retrieved for a given triangle using the *getAdjacentTriangles* method.
- *meshPreprocessParams* specifies mesh pre-processing parameters
 - *PxMeshPreprocessingFlag::eWELD_VERTICES* enables vertex welding during triangle mesh cooking.
 - *PxMeshPreprocessingFlag::eDISABLE_CLEAN_MESH* disables mesh cleaning process. Vertices duplicities are not searched, huge triangle meshes are allowed. Vertices welding is not done. Does speed up the cooking.
 - *PxMeshPreprocessingFlag::eDISABLE_ACTIVE_EDGES_PRECOMPUTATION* disables vertex edge precomputation. Makes cooking faster but may affect contact generation.
- *meshWeldTolerance* - If mesh welding is enabled, this controls the distance between vertices are welded. If mesh welding is not enabled, this value defines the maximum distance for mesh validation. Provided no two vertices are within this distance, mesh is considered to be clean. If not, a warning will be emitted. Higher tolerance is required to achieve the best possible performance.

- *midphaseDesc* specifies the desired midphase acceleration structure
 - *PxBVH33MidphaseDesc* - *PxMeshMidPhase::eBVH33* structure. It was the one used in recent PhysX versions and has great performance and is supported on all platforms.
 - *PxBVH34MidphaseDesc* - *PxMeshMidPhase::eBVH34* implementation introduced in PhysX 3.4. It can be significant in terms of cooking performance and runtime performance, only available on platforms supporting the SSE2 instruction

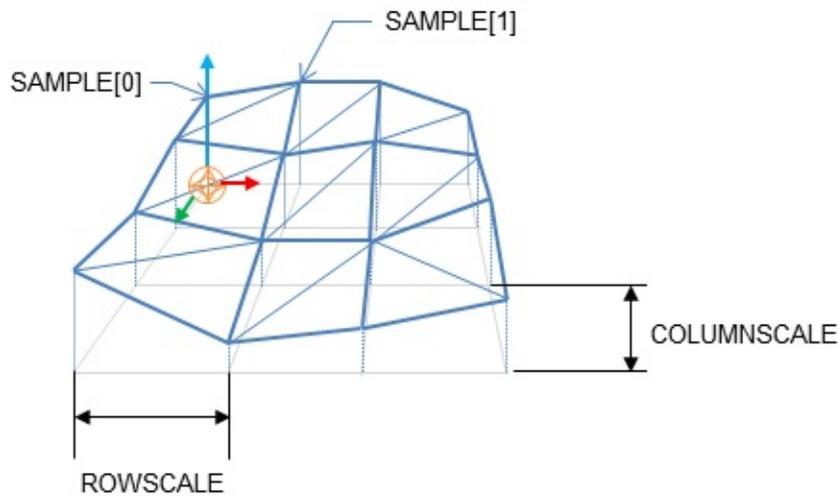
PxBVH33MidphaseDesc params:

- *meshCookingHint* specifies mesh hierarchy construction preference: cooking performance over collision performance, for application performance is more important than best quality mesh creation.
- *meshSizePerformanceTradeOff* specifies the trade-off between cooking and runtime performance.

PxBVH34MidphaseDesc params:

- *numTrisPerLeaf* specifies the number of triangles per leaf. Less triangles produces larger meshes with general better runtime performance.

Height Fields



Local space axes for the height fields are:

- Row - X axis
- Column - Z axis
- Height - Y axis

As the name suggests, terrains can be described by just the height v rectangular sampling grid:

```
PxHeightFieldSample* samples = (PxHeightFieldSample*)alloc(sizeof
    (numRows*numCols));
```

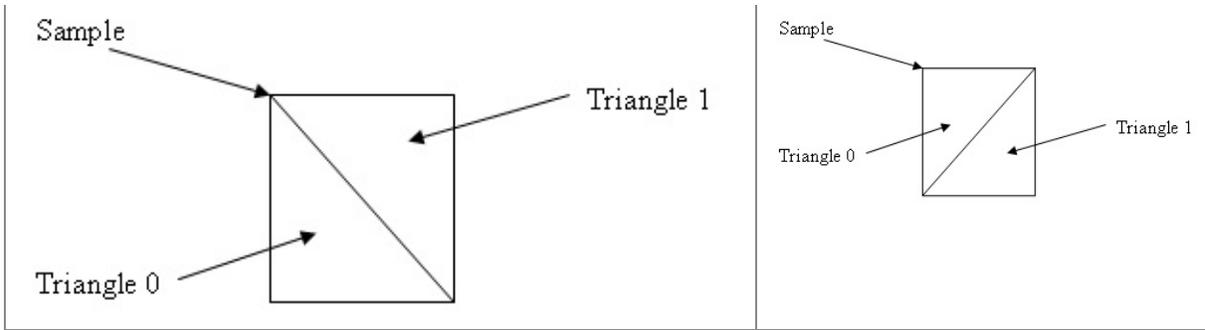
Each sample consists of a 16 bit integer height value, two materials (f in the samples rectangle) and a tessellation flag.

The flag and materials refer to the cell below and to the right of the indicate along which diagonal to split it into triangles, and the materials A special predefined material `PxHeightFieldMaterial::eHOLE` specifies a field. See the reference documentation for `PxHeightFieldSample` for mo

Tessellation flag set

Tessellation flag not set

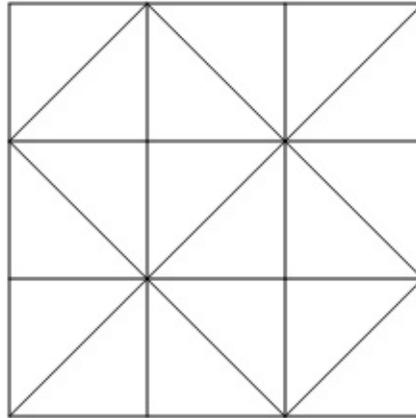
--	--



Examples:

Tessellation flags	Result
0,0,0 0,0,0 0,0,0	
1,1,1 1,1,1 1,1,1	

```
0,1,0  
1,0,1  
0,1,0
```



To tell the system the number of sampled heights in each direction, instantiate a `PxHeightField` object:

```
PxHeightFieldDesc hfDesc;  
hfDesc.format      = PxHeightFieldFormat::eS16_TM;  
hfDesc.nbColumns   = numCols;  
hfDesc.nbRows      = numRows;  
hfDesc.samples.data = samples;  
hfDesc.samples.stride = sizeof(PxHeightFieldSample);  
  
PxHeightField* aHeightField = theCooking->createHeightField(hfDesc,  
    thePhysics->getPhysicsInsertionCallback());
```

Now create a `PxHeightFieldGeometry` and a shape:

```
PxHeightFieldGeometry hfGeom(aHeightField, PxMeshGeometryFlags(),  
    colScale);  
PxShape* aHeightFieldShape = PxRigidActorExt::createExclusiveShape(  
    hfGeom, aMaterialArray, nbMaterials);
```

The row and column scales tell the system how far apart the samples are in each associated direction. The height scale scales the integer height values to a range.

The variant of `createExclusiveShape()` used here specifies an array of height values, which will be indexed by the material indices of each cell to be used with that cell. The single-material variant may be used instead, but the height

indices must all be a single value or the special value `eHOLE`.

Contact generation with triangle edges at the terrain's borders can be `PxHeightFieldFlag::eNO_BOUNDARY_EDGES` flag, allowing more generation when there are multiple heightfield shapes arranged so that

Heightfield cooking

Heightfield data can be cooked in offline and then used to create `HeightField`. `HeightField` does precompute and store the edge information. This allows much faster contact generation for a heightfield, since the edges are already precomputed. It is very useful to create heightfields in the runtime, since it does improve the speed of contact generation significantly.

Heightfield cooking proceeds as follows:

- Load heightfield samples into internal memory.
- Precompute edge collision information.
- Save data to stream.

Unified Heightfields

PhysX provides two contact generation approaches for heightfields. The

- Default unified heightfield contact generation.
- Legacy heightfield contact generation.

The default unified heightfield contact generation approach extracts the heightfield and utilizes the same low-level contact generation code that is used for contact generation against triangle meshes. This approach ensures equivalent performance if triangle meshes or heightfields are used interchangeably. With this approach, the heightfield surface has no thickness so fast-moving objects can pass through it if CCD is not enabled.

The legacy heightfield collision code, which was default in previous versions,

works differently from triangle mesh contact generation. In addition to generating contacts with shapes touching the surface of the heightfield, it generates contacts that are beneath the surface. The heightfield's "thickness" is used to control how many surface contacts are generated. This works by extruding the AABB of the broad phase by the "thickness" along the vertical axis. Contacts are generated for shapes below the surface whose bounds intersect the heightfield's extruded volume.

Unified heightfield contact generation is enabled by calling:

```
PxRegisterHeightFields(PxPhysics& physics);
```

Legacy heightfield contact generation is enabled by calling:

```
PxRegisterLegacyHeightFields(PxPhysics& physics);
```

These calls must be made before any scenes have been created, otherwise they will be ignored. The heightfield collision setting is a global setting, and it applies to all physics objects.

If `PxCreatePhysics(...)` is called, this will automatically call `PxRegisterHeightFields` to register the default, unified heightfield collision approach. If `PxCreatePhysics` is called without the unified approach, no heightfield contact generation is registered by default. If heightfield contact generation is desired, the application must call the appropriate heightfield registration function.

Mesh Scaling

A shared `PxTriangleMesh` or `PxConvexMesh` may be stretched or compressed by a geometry. This allows multiple instancing of the same mesh with different scale factors applied. Scaling is specified with the `PxMeshScale` class, which allows scale factors to be applied along 3 orthogonal axes. A factor greater than 1.0 results in stretching, while a factor less than 1.0 results in compression. The directions are governed by a quaternion, and specified in the local frame of the shape.

Negative mesh scale is supported, with negative values producing a reflection along the corresponding axis. In addition PhysX will flip the normals for meshes where $\text{scale.x} * \text{scale.y} * \text{scale.z} < 0$.

The following code creates a shape with a `PxTriangleMesh` scaled by the x-axis, y along the y-axis, and z along the z-axis:

```
// created earlier
PxRigidActor* myActor;
PxTriangleMesh* myTriMesh;
PxMaterial* myMaterial;

// create a shape instancing a triangle mesh at the given scale
PxMeshScale scale(PxVec3(x,y,z), PxQuat(PxIdentity));
PxTriangleMeshGeometry geom(myTriMesh, scale);
PxShape* myTriMeshShape = PxRigidActorExt::createExclusiveShape(*
```

Convex meshes are scaled using the `PxMeshScale` class in a similar way. The following code creates a shape with a `PxConvexMesh` scaled by a factor of $(\sqrt{1/2}, 1.0, -\sqrt{1/2})$, by a factor of y along $(0,1,0)$ and a factor of z along $(\sqrt{1/2}, 1.0, \sqrt{1/2})$:

```
PxMeshScale scale(PxVec3(x,y,z), PxQuat quat(PxPi*0.25f, PxVec3(0,1,0)));
PxConvexMeshGeometry geom(myTriMesh, scale);
PxShape* myConvexMeshShape = PxRigidActorExt::createExclusiveShape(*
```

Height fields can also be scaled, using scale factors stored in `PxHeightField`.

this case the scale is assumed to be along the axes of the rows, columns and depth directions of the height field. The scaling of is demonstrated in `SampleNorthPoleBuilder.cpp`:

```
PxHeightFieldGeometry hfGeom(heightField, PxMeshGeometryFlags(),  
PxShape* hfShape = PxRigidActorExt::createExclusiveShape(*hfActor
```

In this example, the coordinates along the x and z axes are scaled by sample heights are scaled by `heightScale`.

PxGeometryHolder

When a geometry is provided for a shape, either on `PxShape::setGeometry()`, the geometry is copied into the SDK's internal cache. If you know the type of a shape's geometry you may retrieve it directly:

```
PxBoxGeometry boxGeom;  
bool status = shape->getBoxGeometry(geometry);
```

The status return code is set to false if the shape's geometry is not of the specified type.

However, it is often convenient to retrieve a geometry object from a shape without knowing its type - for example, to call a function which takes a `PxGeometry` as an argument.

`PxGeometryHolder` is a union-like class that allows the return of a `PxGeometry` value, regardless of type. Its use is illustrated in the `createRenderActor` function in `PhysXSample.cpp`:

```
PxGeometryHolder geom = shape->getGeometry();  
  
switch(geom.getType())  
{  
case PxGeometryType::eSPHERE:  
    shapeRenderActor = SAMPLE_NEW(RenderSphereActor)(renderer, geom);  
    break;  
case PxGeometryType::eCAPSULE:  
    shapeRenderActor = SAMPLE_NEW(RenderCapsuleActor)(renderer, geom, geom.capsule().halfHeight);  
    break;  
...  
}
```

The function `PxGeometryHolder::any()` returns a reference to a `PxGeometry` object. For example, to compare two shapes in a scene for overlap:

```
bool testForOverlap(const PxShape& s0, const PxShape& s1)  
{
```


Vertex and Face Data

Convex meshes, triangle meshes, and height fields can all be queried data. This is particularly useful, for example, when rendering the mesh. The function:

```
RenderBaseActor* PhysXSample::createRenderObjectFromShape(PxShape  
RenderMaterial* material)
```

in `PhysXSample.cpp` contains a switch statement with a case for illustrating the steps required to query the vertices and faces.

It is possible to get information about triangle from a triangle mesh or `PxMeshQuery::getTriangle` function. You can also retrieve adjacent triangles given triangle (`triangle triangleNeighbour[i]` shares the edge `vertex[i]`-`vertex[i+1]` of `triangle` indexed as `'triangleIndex'`, where `vertex` is in the range from 0 to `triangleNumVertices`). To feature the triangle mesh is cooked with `buildTriangleAdjacencies` parameter.

Convex Meshes

A convex mesh contains an array of vertices, an array of faces, and an array that concatenates the vertex indices for each face. To unpack a convex mesh to extract the shared convex mesh:

```
PxConvexMesh* convexMesh = geom.convexMesh().convexMesh;
```

Then obtain references to the vertex and index buffers:

```
PxU32 nbVerts = convexMesh->getNbVertices();  
const PxVec3* convexVerts = convexMesh->getVertices();  
const PxU8* indexBuffer = convexMesh->getIndexBuffer();
```

Now iterate over the array of faces to triangulate them:

```
PxU32 offset = 0;
```

```

for(PxU32 i=0;i<nbPolygons;i++)
{
    PxHullPolygon face;
    bool status = convexMesh->getPolygonData(i, face);
    PX_ASSERT(status);

    const PxU8* faceIndices = indexBuffer + face.mIndexBase;
    for(PxU32 j=0;j<face.mNbVerts;j++)
    {
        vertices[offset+j] = convexVerts[faceIndices[j]];
        normals[offset+j] = PxVec3(face.mPlane[0], face.mPlane[1]
    }

    for(PxU32 j=2;j<face.mNbVerts;j++)
    {
        *triangles++ = PxU16(offset);
        *triangles++ = PxU16(offset+j);
        *triangles++ = PxU16(offset+j-1);
    }

    offset += face.mNbVerts;
}

```

Observe that the vertex indices of the polygon begin at `indexBuffer[face` the count of vertices is given by `face.mNbVerts`.

Triangle Meshes

Triangle meshes contain arrays of vertices and index triplets which def indexing into the vertex buffer. The arrays can be accessed directl triangle mesh:

```

PxTriangleMesh* tm = geom.triangleMesh().triangleMesh;
const PxU32 nbVerts = tm->getNbVertices();
const PxVec3* verts = tm->getVertices();
const PxU32 nbTris = tm->getNbTriangles();
const void* tris = tm->getTriangles();

```

The indices may be stored with either 16-bit or 32-bit values, specifi was originally cooked. To determine the storage format at runtime, use t

```
const bool has16bitIndices = tm->has16BitTriangleIndices();
```

Assuming that the triangle indices are stored in 16-bit format, find the triangle by:

```
const PxU16* triIndices = (const PxU16*)tris;  
const PxU16 index = triIndices[3*i +j];
```

The corresponding vertex is:

```
const PxVec3& vertex = verts[index];
```

Height Fields

The storage of height field data is platform-dependent, and therefore a height field samples is not provided. Instead, calls are provided to render to a user-supplied buffer.

Again, the first step is to retrieve the geometry for the height field:

```
const PxHeightFieldGeometry& geometry = geom.heightField();
```

The height field has three scaling parameters:

```
const PxReal    rs = geometry.rowScale;  
const PxReal    hs = geometry.heightScale;  
const PxReal    cs = geometry.columnScale;
```

And a shared data structure, which stores the row and column count:

```
PxHeightField* hf = geometry.heightField;  
const PxU32    nbCols = hf->getNbColumns();  
const PxU32    nbRows = hf->getNbRows();
```

To render the height field, first extract the samples to an array:

```
const PxU32 nbVerts = nbRows * nbCols;
```

```
PxHeightFieldSample* sampleBuffer = new PxHeightFieldSample[nbVer  
hf->saveCells(sampleBuffer, nbVerts * sizeof(PxHeightFieldSample)
```

The samples are stored in row-major order; that is, row0 is stored first then row2, and so on. Thus the sample corresponding to the i th row and j th column is at index $i * \text{nbCols} + j$.

Evaluate the scaled vertices of the height field as follows:

```
PxVec3* vertices = new PxVec3[nbVerts];  
for(PxU32 i = 0; i < nbRows; i++)  
{  
    for(PxU32 j = 0; j < nbCols; j++)  
    {  
        vertices[i * nbCols + j] = PxVec3(PxReal(i) * rs, PxReal(  
            (i * nbCols + j) * height) * hs, PxReal(j) * cs);  
    }  
}
```

Then tessellate the field from the samples as required.

Heightfield Modification

Heightfield samples can be modified at runtime in rectangular blocks. In the snippet we create a HF and modify its samples:

```
// create a 5x5 HF with height 100 and materials 2,3  
PxHeightFieldSample samples1[25];  
for (PxU32 i = 0; i < 25; i++)  
{  
    samples1[i].height = 100;  
    samples1[i].materialIndex0 = 2;  
    samples1[i].materialIndex1 = 3;  
}  
  
PxHeightFieldDesc heightFieldDesc;  
heightFieldDesc.nbColumns = 5;  
heightFieldDesc.nbRows = 5;  
heightFieldDesc.thickness = -10;  
heightFieldDesc.convexEdgeThreshold = 3;  
heightFieldDesc.samples.data = samples1;  
heightFieldDesc.samples.stride = sizeof(PxHeightFieldSample);
```

```

PxPhysics* physics = getPhysics();
PxHeightField* pHeightField = cooking->createHeightField(heightFi

// create modified HF samples, this 10-sample strip will be used
// Source samples that are out of range of target heightfield wil
PxHeightFieldSample samplesM[10];
for (PxU32 i = 0; i < 10; i ++)
{
    samplesM[i].height = 1000;
    samplesM[i].materialIndex0 = 1;
    samplesM[i].materialIndex1 = 127;
}

PxHeightFieldDesc desc10Rows;
desc10Rows.nbColumns = 1;
desc10Rows.nbRows = 10;
desc10Rows.samples.data = samplesM;
desc10Rows.samples.stride = sizeof(PxHeightFieldSample);

pHeightField->modifySamples(1, 0, desc10Rows); // modify row 1 wi

```

PhysX does not keep a mapping from the heightfield to heightfield shape. Call `PxShape::setGeometry` on each shape which references the heightfield. This ensures that internal data structures are updated to reflect the new geometry:

```

PxShape *hfShape = userGetHfShape(); // the user is responsible for
// shapes associated with modifying
hfShape->setGeometry(PxHeightFieldGeometry(pHeightField, ...));

```

Please also note that `PxShape::setGeometry()` does not guarantee behavior when objects are resting on top of old or new geometry.

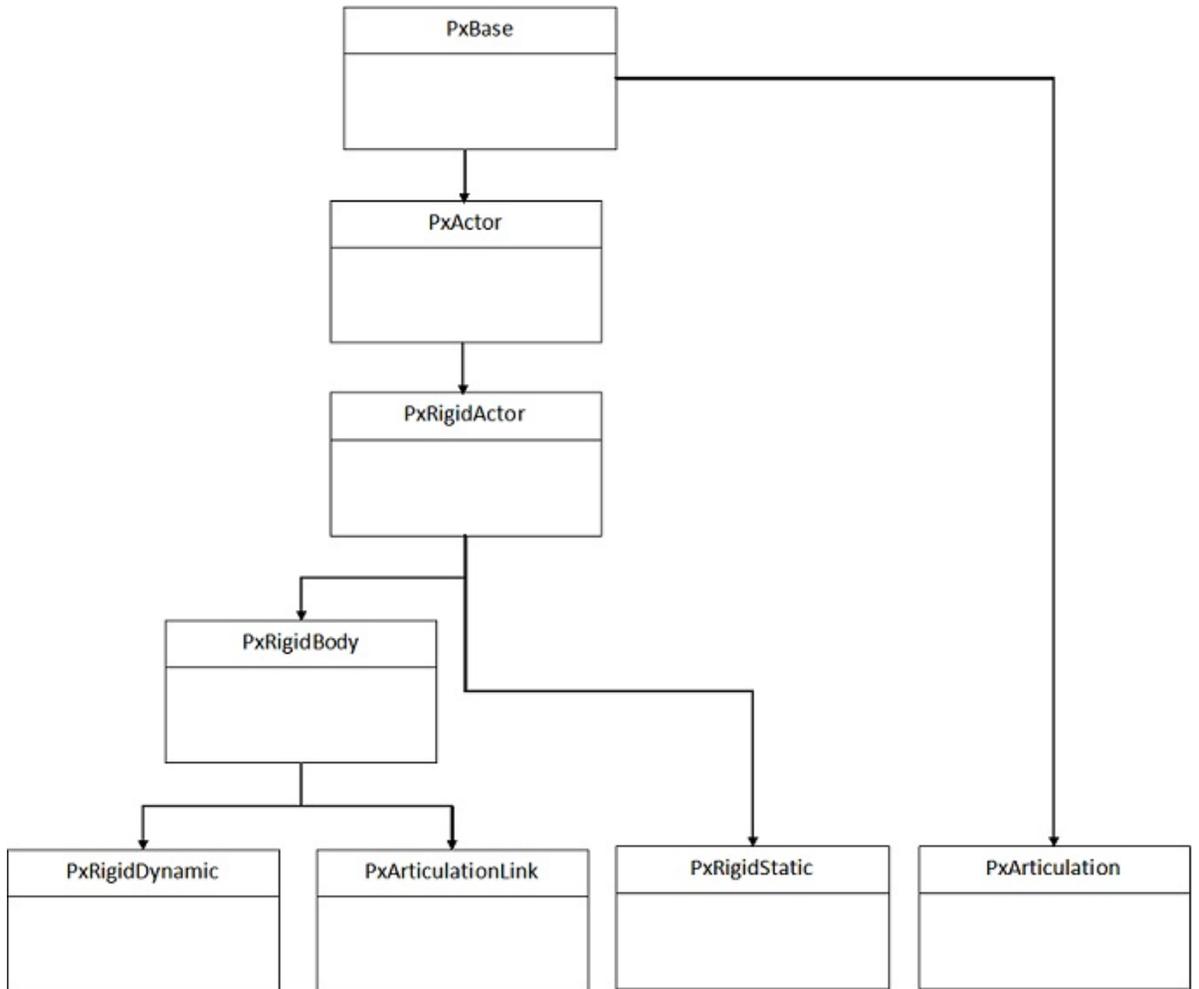
The method `PxHeightField::getTimestamp()` returns the number of times the heightfield has been modified.

Introduction

This chapter will introduce the fundamentals of simulating rigid body c
NVIDIA PhysX engine.

Rigid Body Object Model

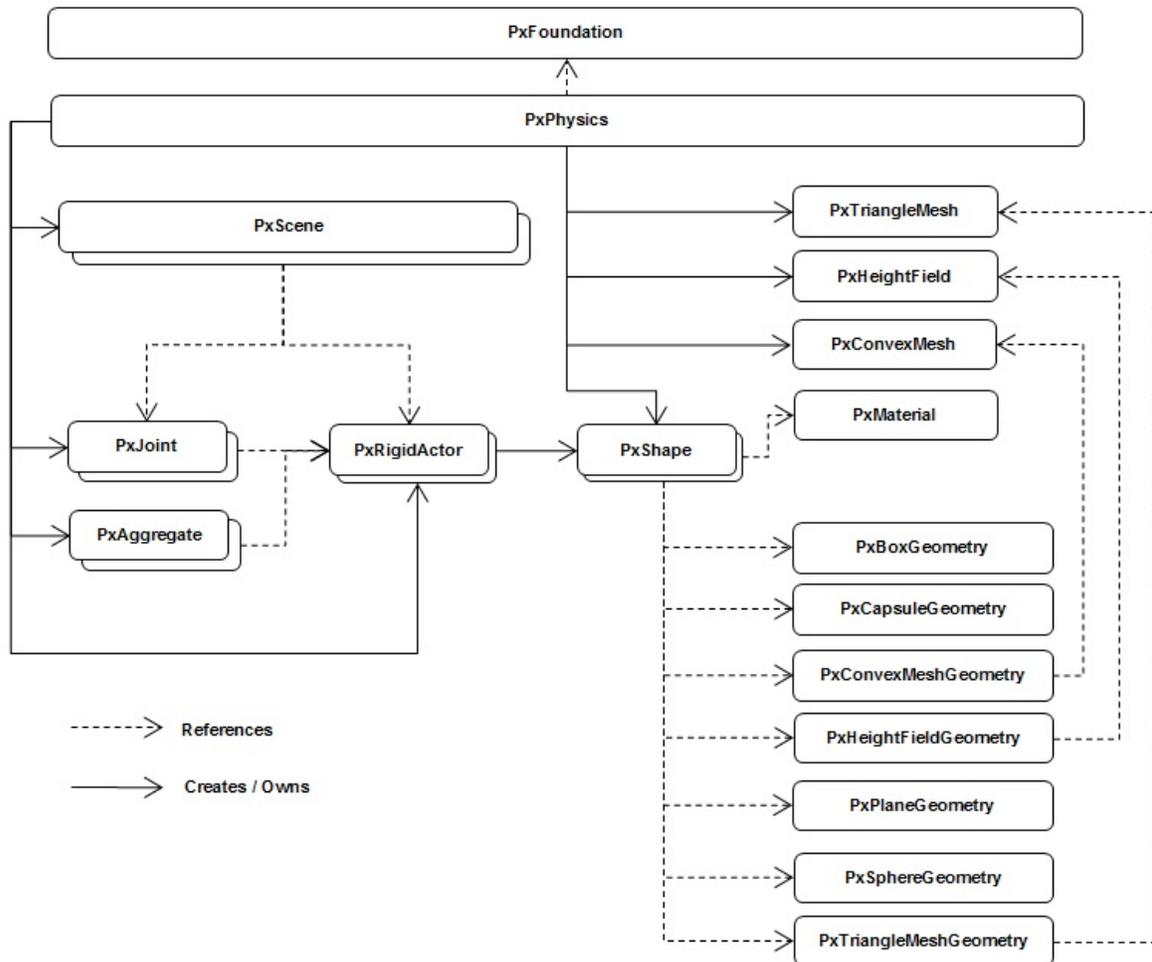
PhysX uses a hierarchical rigid body object/actor model, which looks like



Class	Extends	Functionality
<i>PxBASE</i>	N/A	Reflection/querying object types.
<i>PxACTION</i>	PxBASE	Actor name, actor flags, dominance, aggregates, query world bounds.
<i>PXRIGIDACTION</i>	PxACTION	Shapes and transforms.
<i>PXRIGIDBODY</i>	PXRIGIDACTION	Mass, inertia, velocities, body flags.
<i>PXRIGIDSTATIC</i>	PXRIGIDACTION	Interface for static body in the scene body has implicit infinite mass/inertia
<i>PXRIGIDDYNAMIC</i>	PXRIGIDBODY	Interface for dynamic rigid body in th Introduces support for kinematic targ

		sleeping.
<i>PxArticulationLink</i>	PxRigidBody	Interface for a dynamic rigid body link PxArticulation. Introduces support for articulation and adjacent links.
<i>PxArticulation</i>	PxBase	Defines interface for a PxArticulation contained referencing multiple PxArt bodies.

The following diagram shows the relationship between the main types in the body pipeline:



The Simulation Loop

Now use the method `PxScene::simulate()` to advance the world forward. Here is a simplified code from the samples' fixed stepper class:

```
mAccumulator = 0.0f;
mStepSize = 1.0f / 60.0f;

virtual bool advance(PxReal dt)
{
    mAccumulator += dt;
    if(mAccumulator < mStepSize)
        return false;

    mAccumulator -= mStepSize;

    mScene->simulate(mStepSize);
    return true;
}
```

This is called from the sample framework whenever the app is done with events and is starting to idle. It accumulates elapsed real time until it reaches a sixtieth of a second, and then calls `simulate()`, which moves all objects forward by that interval. This is probably the simplest of very many different ways to step with time when stepping the simulation forward.

To allow the simulation to finish and return the results, simply call:

```
mScene->fetchResults(true);
```

True indicates that the simulation should block until it is finished, so the results are guaranteed to be available. When `fetchResults` completes, the event callback functions that you defined will also be called. See the [Sequence](#).

It is possible to read and write from the scene during simulation. One disadvantage of this to perform rendering work in parallel with physics. When `fetchResults` returns, the results of the current simulation step are not available. So rendering

parallel with simulation renders the actors as they were when simulate fetchResults() returns, all these functions will return the new, post-simulation chapter [Threading](#) for more details about reading and writing while running.

For the human eye to perceive animated motion as smooth, use at least 30 frames per second, with each frame corresponding to a physics time step. For a smooth, realistic simulation of more complex physical scenes, use at least 60 frames per second.

Note: If you are making a real-time interactive simulation, you may use different sized time steps which correspond to the amount of real time elapsed since the last simulation frame. Be very careful if you do this, rather than constant-sized time steps: The simulation code is sensitive to both very small time steps, and also to too much variation between time steps. In these cases, you are likely to produce jittery simulation.

See [Simulation memory](#) for details of how memory is used in simulation.

Introduction

This section will introduce the fundamentals of rigid body collision.

Shapes

Shapes describe the spatial extent and collision properties of actors. They have three purposes within PhysX: intersection tests that determine the collision between rigid objects, scene query tests such as raycasts, and defining triggers that generate notifications when other shapes intersect with them.

Shapes are reference counted, see [Reference Counting](#).

Each shape contains a PxGeometry object and a reference to a PxMaterial, both of which must be specified upon creation. The following code creates a shape with a sphere geometry and a specific material:

```
PxShape* shape = physics.createShape(PxSphereGeometry(1.0f), myMaterial, true);
myActor.attachShape(*shape);
shape->release();
```

The method `PxRigidActorExt::createExclusiveShape()` is equivalent to the code above.

Note: for reference counting behavior of deserialized shapes refer to [Counting of Deserialized Objects](#).

The parameter 'true' to `createShape()` informs the SDK that the shape is shared with other actors. You can use shape sharing to reduce the memory usage in a simulation when you have many actors with identical geometry, but shape sharing has a very strong restriction: you cannot update the attributes of a shared shape once it is attached to an actor.

Optionally you may configure a shape by specifying shape flags of type `PxShapeFlags`. By default a shape is configured as

- a simulation shape (enabled for contact generation during simulation)
- a scene query shape (enabled for scene queries)

- being visualized if debug rendering is enabled

When a geometry object is specified for a shape, the geometry object is attached to the actor. There are some restrictions on which geometries may be specified depending on the shape flags and the type of the parent actors.

- TriangleMesh, HeightField and Plane geometries are not supported for dynamic shapes that are attached to dynamic actors, unless the dynamic actor is set to be kinematic.
- TriangleMesh and HeightField geometries are not supported for trigger shapes.

See the following sections for more details.

Detach the shape from the actor as follows:

```
myActor.detachShape(*shape);
```

Note: in previous versions of PhysX, `release()` was used to detach a shape from an actor and destroy it. This use of `release()` is deprecated in PhysX 3.3 and will not be supported in future versions of PhysX.

Simulation Shapes and Scene Query Shapes

Shapes may be independently configured to participate in either or both simulation and contact tests. By default, a shape will participate in both.

The following pseudo-code configures a PxShape instance so that it does not participate in shape pair intersection tests:

```
void disableShapeInContactTests(PxShape* shape)
{
    shape->setFlag(PxShapeFlag::eSIMULATION_SHAPE, false);
}
```

A PxShape instance can be configured to participate in shape pair intersection tests as follows:

```
void enableShapeInContactTests(PxShape* shape)
{
    shape->setFlag(PxShapeFlag::eSIMULATION_SHAPE, true);
}
```

To disable a PxShape instance from scene query tests:

```
void disableShapeInSceneQueryTests(PxShape* shape)
{
    shape->setFlag(PxShapeFlag::eSCENE_QUERY_SHAPE, false);
}
```

Finally, a PxShape instance can be re-enabled in scene query tests:

```
void enableShapeInSceneQueryTests(PxShape* shape)
{
    shape->setFlag(PxShapeFlag::eSCENE_QUERY_SHAPE, true);
}
```

Note: If the movement of the shape's actor does not need to be controlled in simulation at all, i.e., the shape is used for scene queries only and gets

if necessary, then memory can be saved by additionally disabling simu
(see the API documentation on PxActorFlag::eDISABLE_SIMULATION

Kinematic Triangle Meshes (Planes, Heighfields)

It is possible to create a kinematic `PxRigidDynamic` which can have (plane, heightfield) shape. If this shape has a simulation shape flag, it is kinematic. If you change the flag to not simulated, you can switch even

To setup kinematic triangle mesh see following code:

```
PxRigidDynamic* meshActor = getPhysics().createRigidDynamic(PxTra  
PxShape* meshShape;  
if(meshActor)  
{  
    meshActor->setRigidDynamicFlag(PxRigidDynamicFlag::eKINEMATIC  
  
    PxTriangleMeshGeometry triGeom;  
    triGeom.triangleMesh = triangleMesh;  
    meshShape = PxRigidActorExt::createExclusiveShape(*meshActor,  
        defaultMaterial);  
    getScene().addActor(*meshActor);  
}
```

To switch a kinematic triangle mesh actor to a dynamic actor:

```
PxRigidDynamic* meshActor = getPhysics().createRigidDynamic(PxTra  
PxShape* meshShape;  
if(meshActor)  
{  
    meshActor->setRigidDynamicFlag(PxRigidDynamicFlag::eKINEMATIC  
  
    PxTriangleMeshGeometry triGeom;  
    triGeom.triangleMesh = triangleMesh;  
    meshShape = PxRigidActorExt::createExclusiveShape(*meshActor,  
        defaultMaterial);  
    getScene().addActor(*meshActor);  
  
    PxConvexMeshGeometry convexGeom = PxConvexMeshGeometry(convex  
    convexShape = PxRigidActorExt::createExclusiveShape(*meshActo  
        defaultMaterial);
```

Broad-phase Algorithms

PhysX supports several broad-phase algorithms:

- *sweep-and-prune (SAP)*
- *multi box pruning (MBP)*

PxBroadPhaseType::eSAP was the default algorithm used until PhysX 3.2. It is a generic choice with great performance when many objects are sleeping, but it degrades significantly though, when all objects are moving, or when objects are added to or removed from the broad-phase. This algorithm requires world bounds to be defined in order to work.

PxBroadPhaseType::eMBP is a new algorithm introduced in PhysX 3.3. It is a broad-phase algorithm that does not suffer from the same performance degradation when all objects are moving or when inserting large numbers of objects. Its generic performance when many objects are sleeping might be inferior to *eSAP*, but it requires users to define world bounds in order to work.

The desired broad-phase algorithm is controlled by the *PxBroadPhaseType* member of the *PxSceneDesc* structure.

Regions of Interest

A region of interest is a world-space AABB around a volume of space in a broad-phase. Objects contained inside those regions are properly handled in the broad-phase. Objects falling outside of those regions lose all collision detection. Regions should cover the whole game space, while limiting the amount of space.

Regions can overlap, although for maximum efficiency it is recommended to have as little amount of overlap between regions as much as possible. Note that touching AABBs just touch are not considered overlapping. For the `PxBroadPhaseExt::createRegionsFromWorldBounds` helper function creates non-overlapping region bounds by simply subdividing a given world AABB into a 2D grid.

Regions can be defined by the `PxBroadPhaseRegion` structure, along with a pointer to the broad-phase they are assigned to them. They can be defined at scene creation time or at a later time using the `PxScene::addBroadPhaseRegion` function. The SDK returns a handle to the newly created regions, that can be used later to remove a region using the `PxScene::removeBroadPhaseRegion` function.

A newly added region may overlap already existing objects. The SDK will automatically add those objects to the new region, if the `populateRegion` parameter of the `PxScene::addBroadPhaseRegion` call is set. However this operation might have a high impact on performance, especially when several regions are added in the same frame. Thus, it is recommended to disable it whenever possible. A region would then be created empty, and it would only be populated either with new objects added to the scene after the region has been created, or with previously existing objects that are updated (i.e. when they move).

Note that only `PxBroadPhaseType::eMBP` requires regions to be populated. The `PxBroadPhaseType::eSAP` algorithm does not. This information is contained in the `PxBroadPhaseCaps` structure, which lists information and capabilities for each broad-phase algorithm. This structure can be retrieved by the `PxScene::getBroadPhaseCaps` function.

function.

Runtime information about current regions can be retrieved using the methods *PxScene::getNbBroadPhaseRegions* and *PxScene::getBroadPhaseRegions*.

The maximum number of regions is currently limited to 256.

Broad-phase Callback

A callback for broad-phase-related events can be defined within a structure. This *PxBroadPhaseCallback* object will be called when objects enter the specified regions of interest, i.e. "out of bounds". The SDK disables collisions for those objects. It is re-enabled automatically as soon as the object re-enters the region.

It is up to users to decide what to do with out-of-bounds objects. Typical

- delete the objects
- let them continue their motion without collisions until they re-enter a valid region
- artificially teleport them back to a valid place

Collision Filtering

In almost all applications beyond the trivial, the need arises to exempt objects from interacting, or to configure the SDK collision detection behavior for an interacting pair. In the submarine sample, like indicated above, the submarine is notified when the submarine touched a mine, or the chain of a mine, so that they can blow up. The crab's AI also needs to know when crabs touch the floor.

Before we can understand what the sample does to achieve this, we need to explore the possibilities of the SDK filtering system. Because filtering potentially happens in the deepest parts of the simulation engine, and needs to be done for objects that come near each other, it is particularly performance sensitive. One way to implement it would be to always call a callback function for every interacting pair, where the application, based on the two object pointers, decides whether to interact using some custom logic -- like consulting its game data base -- when they interact. Unfortunately this quickly becomes too slow if done for a very large number of objects, especially if the collision detection processing happens on a remote GPU or an other kind of vector processor with local memory, which would interrupt its parallel computations, interrupt the main processor that runs game logic, and execute the callback before it can continue. Even if it were to be executed on a GPU, it would likely be done so simultaneously on multiple cores or hyperthreads, and synchronization code would have to be put in place to make sure that concurrent accesses to the game data base are safe. Far better is to use some kind of fixed function logic that can be executed on the GPU processor. This is what we did in PhysX 2.x -- unfortunately the simple filtering rules we provided were not flexible enough to cover all applications. In PhysX 3.0 we introduce both a shader system, which lets the developer implement custom filtering rules using code that runs on the vector processor (and is therefore not limited by the eventual game data base in main memory), which is more flexible than the simple filtering, but just as efficient, and a totally flexible callback mechanism. The shader calls a CPU callback function that is able to access any application data at the cost of performance -- see `PxSimulationFilterCallback` for details. The application can decide on a per-pair basis to make this speed vs. flexibility trade-off.

Let us look at the shader system first: Here is the filter shade

SampleSubmarine:

```
PxFilterFlags SampleSubmarineFilterShader(  
    PxFilterObjectAttributes attributes0, PxFilterData filterData  
    PxFilterObjectAttributes attributes1, PxFilterData filterData  
    PxPairFlags& pairFlags, const void* constantBlock, PxU32 cons  
{  
    // let triggers through  
    if(PxFilterObjectIsTrigger(attributes0) || PxFilterObjectIsTr  
    {  
        pairFlags = PxPairFlag::eTRIGGER_DEFAULT;  
        return PxFilterFlag::eDEFAULT;  
    }  
    // generate contacts for all that were not filtered above  
    pairFlags = PxPairFlag::eCONTACT_DEFAULT;  
  
    // trigger the contact callback for pairs (A,B) where  
    // the filtermask of A contains the ID of B and vice versa.  
    if((filterData0.word0 & filterData1.word1) && (filterData1.w  
        pairFlags |= PxPairFlag::eNOTIFY_TOUCH_FOUND;  
  
    return PxFilterFlag::eDEFAULT;  
}
```

SampleSubmarineFilterShader is a simple shader function that is an implementation of the `PxSimulationFilterShader` prototype declared in `PxFiltering.h`. The shader (called `SampleSubmarineFilterShader` above) may not reference any arguments of the function and its own local stack variables -- because it is compiled and executed on a remote processor.

`SampleSubmarineFilterShader()` will be called for all pairs of shapes that intersect -- more precisely: for all pairs of shapes whose axis aligned bounding space are found to intersect for the first time. All behavior beyond that is determined by what `SampleSubmarineFilterShader()` returns.

The arguments of `SampleSubmarineFilterShader()` include `PxFilterObjectAttributes` for the two objects, and a constant block of memory. Note that the two objects are NOT passed, because those pointers refer to the memory, and that may, as we said, not be available to the shader, so the arguments are not very useful, as dereferencing them would likely

PxFilterObjectAttributes and PxFilterData are intended to contain all the information that one could quickly glean from the pointers. PxFilterObjectAttributes encodes information that encodes the type of object: For example PxFilterObjectType::eRIGID_DYNAMIC, or even ::ePARTICLE_SYSTEM. Additionally, it indicates if the object is kinematic, or a trigger.

Each PxShape and PxParticleBase object in PhysX has a member PxFilterData. This is 128 bits of user defined data that can be used to store specific information related to collision filtering. This is the other variable SampleSubmarineFilterShader() for each object.

There is also the constant block. This is a chunk of per-scene global information that an application can give to the shader to operate on. You will want to use it to determine about what to filter and what not.

Finally, SampleSubmarineFilterShader() also has a PxPairFlags parameter as an output, like the return value PxFilterFlags, though used slightly differently. It tells the SDK if it should ignore the pair for good (eKILL), ignore it if the objects are overlapping, but ask again, when filtering related data changes for the pair (eSUPPRESS), or call the low performance but more flexible CPU callback if the SDK cannot decide (eCALLBACK).

PxPairFlags specifies additional flags that stand for actions that the simulation should take in the future for this pair. For example, eNOTIFY_TOUCH_FOUND means to notify the application when the pair really starts to touch, not just potentially.

Let us look at what the above shader does:

```
// let triggers through
if(PxFilterObjectIsTrigger(attributes0) || PxFilterObjectIsTrigger(attributes1))
{
    pairFlags = PxPairFlag::eTRIGGER_DEFAULT;
    return PxFilterFlag::eDEFAULT;
}
```

This means that if either object is a trigger, then perform default trigger filtering (the application about start and end of touch), and otherwise perform

detection between them.

```
// generate contacts for all that were not filtered above
pairFlags = PxPairFlag::eCONTACT_DEFAULT;

// trigger the contact callback for pairs (A,B) where
// the filtermask of A contains the ID of B and vice versa.
if((filterData0.word0 & filterData1.word1) && (filterData1.word0 &
    pairFlags |= PxPairFlag::eNOTIFY_TOUCH_FOUND;

return PxFilterFlag::eDEFAULT;
```

This says that for all other objects, perform 'default' collision handling. It is a rule based on the filterDatas that determines particular pairs where notifications. To understand what this means, we need to know the sample the sample gives to the filterDatas.

The needs of the sample are very basic, so we will use a very simple sample of it. The sample first gives named codes to the different object types in an enumeration:

```
struct FilterGroup
{
    enum Enum
    {
        eSUBMARINE      = (1 << 0),
        eMINE_HEAD      = (1 << 1),
        eMINE_LINK      = (1 << 2),
        eCRAB            = (1 << 3),
        eHEIGHTFIELD    = (1 << 4),
    };
};
```

The sample identifies each shape's type by assigning its PxFilterGroup type. Then, it puts a bit mask that specifies each type of shape that should generate a report when touched by an object of type word0 into word1. This is done in the samples whenever a shape is created, but because shape creation is encapsulated in SampleBase, it is done after the fact, using this function

```

void setupFiltering(PxRigidActor* actor, PxU32 filterGroup, PxU32
{
    PxFilterData filterData;
    filterData.word0 = filterGroup; // word0 = own ID
    filterData.word1 = filterMask; // word1 = ID mask to filter
                                   // contact callback;

    const PxU32 numShapes = actor->getNbShapes();
    PxShape** shapes = (PxShape**)SAMPLE_ALLOC(sizeof(PxShape*)*n
actor->getShapes(shapes, numShapes);
    for(PxU32 i = 0; i < numShapes; i++)
    {
        PxShape* shape = shapes[i];
        shape->setSimulationFilterData(filterData);
    }
    SAMPLE_FREE(shapes);
}

```

This sets up the PxFilterDatas of each shape belonging to the pass some examples how this is used in SampleSubmarine:

```

setupFiltering(mSubmarineActor, FilterGroup::eSUBMARINE, FilterGr
    FilterGroup::eMINE_LINK);
setupFiltering(link, FilterGroup::eMINE_LINK, FilterGroup::eSUBMA
setupFiltering(mineHead, FilterGroup::eMINE_HEAD, FilterGroup::eS

setupFiltering(heightField, FilterGroup::eHEIGHTFIELD, FilterGrou
setupFiltering(mCrabBody, FilterGroup::eCRAB, FilterGroup::eHEIGH

```

This scheme is probably too simplistic to use in a real game, but it shows the filter shader, and it will ensure that SampleSubmarine::onContact returns interesting pairs.

An alternative group based filtering mechanism is provided with source function PxDefaultSimulationFilterShader. And, again, if this shader becomes inflexible, consider using the callback approach provided with PxSimulationFilterShader.

Aggregates

An aggregate is a collection of actors. Aggregates do not provide extra features, but allow you to tell the SDK that a set of actors will be clustered in turn allows the SDK to optimize its spatial data operations. A typical ragdoll, made of multiple different actors. Without aggregates, this gives broad-phase entries as there are shapes in the ragdoll. It is typically represent the ragdoll in the broad-phase as a single entity, and performs tests in a second pass if necessary. Another potential use case is a large number of attached shapes.

Creating an Aggregate

Create an aggregate from the *PxPhysics* object:

```
PxPhysics* physics; // The physics SDK object

PxU32 nbActors;      // Max number of actors expected in the aggregate
bool selfCollisions = true;

PxAggregate* aggregate = physics->createAggregate(nbActors, selfCollisions);
```

The maximum number of actors is currently limited to 128, and for efficiency as low as possible.

If you will never need collisions between the actors of the aggregate at creation time. This is much more efficient than using the scene filtering, bypasses all internal filtering logic. A typical use case would be an aggregate of kinematic actors.

Note that both the maximum number of actors and the self-collision flag are immutable.

Populating an Aggregate

Adds an actor to an aggregate as follows:

```
PxActor& actor;    // Some actor, previously created
aggregate->addActor(actor);
```

Note that if the actor already belongs to a scene, the call is ignored. Either add the actor to an aggregate and then add the aggregate to the scene, or add the scene and then the actors to the aggregate.

To add the aggregate to a scene (before or after populating it):

```
scene->addAggregate(*aggregate);
```

Similarly, to remove the aggregate from the scene:

```
scene->removeAggregate(*aggregate);
```

Releasing an Aggregate

To release an aggregate:

```
PxAggregate* aggregate;    // The aggregate we previously created  
aggregate->release();
```

Releasing the PxAggregate does not release the aggregated actors. If the aggregate belongs to a scene, the actors are automatically re-inserted in that scene. To delete both the PxAggregate and its actors, it is most efficient to release the aggregate then release the PxAggregate when it is empty.

Amortizing Insertion

Adding many objects to a scene in one frame can be a costly operation. A good case for a ragdoll, which as discussed is a good candidate for PxAggregate is localized debris, for which self-collisions are often disabled. To amortize object insertion into the broad-phase structure over several frames, spawn the objects into the scene over those frames, then remove each actor from the aggregate and and

Trigger Shapes

Trigger shapes play no part in the simulation of the scene (though they do participate in scene queries). Instead, their role is to report that they overlap with another shape. Contacts are not generated for the intersection of trigger shapes. Further, because they play no part in the simulation, the SDK will not allow the eSIMULATION_SHAPE and eTRIGGER_SHAPE flags to be raised simultaneously; that is, if one attempts to raise the other will be rejected, and an error will be passed to the user.

Trigger shapes have been used in SampleSubmarine to determine if the submarine has reached the treasure. In the following code the PxAActor representing the treasure is configured as a trigger shape:

```
PxShape* treasureShape;
gTreasureActor->getShapes(&treasureShape, 1);
treasureShape->setFlag(PxShapeFlag::eSIMULATION_SHAPE, false);
treasureShape->setFlag(PxShapeFlag::eTRIGGER_SHAPE, true);
```

The overlaps with trigger shapes are reported in SampleSubmarine's implementation of PxSimulationEventCallback::onTrigger in the PxAActor class, a sub-class of PxSimulationEventCallback:

```
void SampleSubmarine::onTrigger(PxTriggerPair* pairs, PxU32 count)
{
    for(PxU32 i=0; i < count; i++)
    {
        // ignore pairs when shapes have been deleted
        if (pairs[i].flags & (PxTriggerPairFlag::eREMOVED_SHAPE_TRIGGER |
            PxTriggerPairFlag::eREMOVED_SHAPE_OTHER))
            continue;

        if ((&pairs[i].otherShape->getActor() == mSubmarineActor) &&
            (&pairs[i].triggerShape->getActor() == gTreasureActor))
        {
            gTreasureFound = true;
        }
    }
}
```

The code above iterates through all pairs of overlapping shapes that touch the treasure shape. If it is found that the treasure has been touched by the submarine, `gTreasureFound` is set true.

Interactions

The SDK internally creates an interaction object for each overlapping pair of objects in the broad-phase. These objects are not only created for pairs of colliding rigid bodies, but also for pairs of overlapping triggers. Generally speaking users should not be concerned with how many interaction objects are created regardless of the involved objects' types (rigid body or trigger) and regardless of involved *PxFilterFlag* flags.

There is currently a limit of 65535 such interaction objects for each actor. If more than 65535 interactions involve the same actor, then the SDK outputs an error and the extra interactions are ignored.

In this chapter we cover a number of topics that are also important to us if you are comfortable with setting up a basic rigid body simulation world.

Velocity

A rigid body's motion is separated into linear and angular velocity. In a simulation, PhysX will modify the velocity of an object in accordance with applied forces and torques and as a result of various constraints, such as joints.

A body's linear and angular velocities can be read using the following methods:

```
PxVec3 PxRigidBody::getLinearVelocity();  
PxVec3 PxRigidBody::getAngularVelocity();
```

A body's linear and angular velocities can be set using the following methods:

```
void PxRigidBody::setLinearVelocity(const PxVec3& linVel, bool autoSolve);  
void PxRigidBody::setAngularVelocity(const PxVec3& angVel, bool autoSolve);
```

Mass Properties

A dynamic actor needs mass properties: the mass, moment of inertia, and mass frame which specifies the position of the actor's center of mass and the inertia axes. The easiest way to calculate mass properties is the `PxRigidBodyExt::updateMassAndInertia()` helper function, which calculates mass properties based on the actor's shapes and a uniform density value. Custom mass functions allow combinations of per-shape densities and manual specification of mass properties. See the reference for `PxRigidBodyExt` for more details.

The Wobbly Snowmen in the North Pole Sample illustrate the use of mass properties. The snowmen act like roly-poly toys, which are usually just a hollow shell with the bottom filled with some heavy material. The low centers of mass cause them to fall back to an upright position after they have been tilted. They come in three varieties depending on how the mass properties are set:

The first is basically massless. There is just a little sphere with a relatively small mass at the bottom of the Actor. This results in a quite rapid movement due to the low moments of inertia. The snowman feels light.

The second uses the mass of the bottom snowball only, resulting in a low moment of inertia. In addition, the center of mass is moved to the bottom of the actor. This approach is physically correct, but the resulting snowman feels a bit more filled.

The third and fourth snowman use shapes to calculate the mass. The third calculates the moments of inertia first (from the real center of mass), and then the center of mass is moved to the bottom. The other calculates the moments of inertia from the low center of mass that we pass to the calculation routine. Note how the wobbling is for the second case although both have the same mass. The large head accounts for much more in the moment of inertia (the distance from the center of mass squared).

The last snowman's mass properties are set up manually. The sample uses a large moment of inertia to create a specific desired behavior. The dia-

low value in X, and high values in Y and Z, producing a low resistance the X-axis and high resistance around Y and Z. As a consequence, wobble back and forth only around the X axis.

If you have a 3x3 inertia matrix (for example, you have real-life inert objects) use the `PxDiagonalize()` function to obtain principal axes and tensors to initialize `PxRigidDynamic` actors.

When manually setting the mass/inertia tensor of bodies, PhysX requires for the mass and each principal axis of inertia. However, it is legal to provide zero values. When provided with a 0 mass or inertia value, PhysX interprets it as infinite mass or inertia around that principal axis. This can be used to resist all linear motion or that resist all or some angular motion. Examples that could be achieved using this approach are:

- Bodies that behave as if they were kinematic.
- Bodies whose translation behaves kinematically but whose rotation is dynamic.
- Bodies whose translation is dynamic but whose rotation is kinematic.
- Bodies which can only rotate around a specific axis.

Some examples of what could be achieved are detailed below. First, let's create a common structure - a windmill. The code to construct the parts of the windmill are provided below:

```
PxRigidDynamic* dyn = physics.createRigidDynamic(PxTransform(PxVec3(0.f, 0.f, 0.f), PxRigidActorExt::createExclusiveShape(*dyn, PxBBoxGeometry(2.f, 0.5f, 0.5f)));
PxRigidActorExt::createExclusiveShape(*dyn, PxBBoxGeometry(0.2f, 2.f, 2.f));
dyn->setActorFlag(PxActorFlag::eDISABLE_GRAVITY, true);
dyn->setAngularVelocity(PxVec3(0.f, 0.f, 5.f));
dyn->setAngularDamping(0.f);
PxRigidStatic* st = mPhysics.createRigidStatic(PxTransform(PxVec3(0.f, 0.f, 0.f), PxRigidActorExt::createExclusiveShape(*st, PxBBoxGeometry(0.5f, 1.5f, 1.5f)));
scene.addActor(dyn);
scene.addActor(st);
```

The above code creates a static box frame for the windmill and a rotating box for the blades of the turbine. We turn off gravity and angular damping on the

give it an initial angular velocity. As a result, this turbine blade will rotate with an angular velocity indefinitely. However, if another object collided with the turbine blade, the windmill would cease to function correctly because the turbine blade would be knocked out of place. There are several options to make the turbine blade behave as if it were in a fixed position when other bodies interact with it. One such approach might be to make the turbine blade have infinite mass and inertia. In this case, any interactions with other bodies would not affect the turbine at all:

```
dyn->setMass(0.f);  
dyn->setMassSpaceInertiaTensor(PxVec3(0.f));
```

This example retains the previous behavior of the turbine spinning at a constant angular velocity indefinitely. However, now the body's linear velocities cannot be changed by constraints because the body has infinite mass and inertia. If a body collided with the turbine blade, the collision would behave as if the turbine blade was a perfectly rigid body.

Another alternative would be to make the turbine have infinite mass and inertia only around the body's local z-axis. This would provide the same effect as a revolute joint between the turbine and the static windmill frame:

```
dyn->setMass(0.f);  
dyn->setMassSpaceInertiaTensor(PxVec3(0.f, 0.f, 10.f));
```

In both examples, the body's mass was set to 0, indicating that the body's linear velocity cannot be changed by any constraints. However, in the second example, the body's inertia is configured to permit the body's angular velocity to be changed by constraints around one principal axis of inertia. This provides a similar effect to a revolute joint. The value of the inertia around the z-axis can be increased or decreased to make the turbines more/less resistive to motion.

Applying Forces and Torques

The most physics-friendly way to interact with a body is to apply a force. In classical mechanics, most interactions between bodies are typically solved because of the law:

$$f = m \cdot a \text{ (force = mass * acceleration)}$$

Forces directly control a body's acceleration, but its velocity and position are also affected. For this reason, control by force may be inconvenient if you need immediate control over position. One advantage of forces is that regardless of what forces you apply to the bodies, the simulation will be able to keep all the defined constraints (joints and springs) intact. For example, gravity works by applying a force to bodies.

Unfortunately, applying large forces to articulated bodies at the resonance frequency of the system may lead to ever-increasing velocities, and eventually to the failure to maintain the joint constraints. This is not unlike a real-world system, which will ultimately break.

The forces acting on a body are accumulated before each simulation frame, then applied during the simulation, and then reset to zero in preparation for the next frame. The list of `PxRigidBody` and `PxRigidBodyExt` methods are listed below. Please refer to the documentation for more detail:

```
void PxRigidBody::addForce(const PxVec3& force, PxForceMode::Enum mode);
void PxRigidBody::addTorque(const PxVec3& torque, PxForceMode::Enum mode);

void PxRigidBodyExt::addForceAtPos(PxRigidBody& body, const PxVec3& force,
    const PxVec3& pos, PxForceMode::Enum mode, bool wakeup);
void PxRigidBodyExt::addForceAtLocalPos(PxRigidBody& body, const PxVec3& force,
    const PxVec3& pos, PxForceMode::Enum mode, bool wakeup);
void PxRigidBodyExt::addLocalForceAtPos(PxRigidBody& body, const PxVec3& force,
    const PxVec3& pos, PxForceMode::Enum mode, bool wakeup);
void PxRigidBodyExt::addLocalForceAtLocalPos(PxRigidBody& body, const PxVec3& force,
    const PxVec3& pos, PxForceMode::Enum mode, bool wakeup);
```

The `PxForceMode` member defaults to `PxForceMode::eFORCE` to apply a constant force. There are other possibilities. For example `PxForceMode::eIMPULSE` will apply an impulsive force. `PxForceMode::eVELOCITY_CHANGE` will do the same as `eFORCE` but scaled by the mass of the body, effectively leading to an instantaneous velocity change. See the documentation of `PxForceMode` for the other possibilities.

Note: The methods in `PxRigidBodyExt` support only the force modes `eIMPULSE`.

There are further extension functions that compute the linear and angular velocity delta that would arise in the next simulation frame if an impulsive force or impulse is to be applied:

```
void PxRigidBodyExt::computeVelocityDeltaFromImpulse(const PxRigidBodyExt& body,
const PxVec3& impulsiveForce, const PxVec3& impulsiveTorque,
PxVec3& deltaAngularVelocity);
```

A use case for this function might be to predict an updated velocity for an asset that asset loading may be initiated in advance of the simulation frame to exceed a threshold velocity at the end of the frame. The impulsive force is simply the force and torque that are to be applied to the body multiplied by the simulation frame. Neglecting the effect of constraint and contact for the moment, the linear and angular velocity that are expected to arise in the next simulation frame are returned in `deltaLinearVelocity` and `deltaAngularVelocity`. The predicted velocity can then be computed with `body.getLinearVelocity() + deltaLinearVelocity`, and the predicted angular velocity can be computed with `body.getAngularVelocity() + deltaAngularVelocity`. If required, it is possible to immediately update the velocity of the body with `body.setLinearVelocity(body.getLinearVelocity() + deltaLinearVelocity)` and `body.setAngularVelocity(body.getAngularVelocity() + deltaAngularVelocity)`.

Gravity

Gravity is such a common force in simulations that PhysX makes it particularly easy to apply. For a scene-wide gravity effect, or any other uniform force field, you can set the scene's gravity vector using `PxScene::setGravity()`.

The parameter is the acceleration due to gravity. In meters and second, it should have a magnitude of about 9.8 on earth, and should point downwards. The force applied at the center of mass of each body in the scene is this acceleration times the actor's mass.

Certain special effects can require that some dynamic actors are not influenced by gravity. To specify this set the flag:

```
PxActor::setActorFlag(PxActorFlag::eDISABLE_GRAVITY, true);
```

Note: Be careful when changing gravity (or enabling/disabling it) during a simulation. For performance reasons the change will not wake up sleeping actors. Thus it may be necessary to iterate through all actors and call `PxRigidDynamic::wakeUp()` manually.

An alternative to `PxActorFlag::eDISABLE_GRAVITY` is to use a zero gravity scene, then apply your own gravity force to rigid bodies, each using `PxRigidDynamic::setGravity()`. This is useful to create radial gravity fields, as demonstrated in `SampleCustomGravity`.

Friction and Restitution

All physical objects have at least one material, which defines the friction properties used to resolve a collision with the objects.

To create a material, call `PxPhysics::createMaterial()`:

```
PxMaterial* mMaterial;  
  
mMaterial = mPhysics->createMaterial(0.5f, 0.5f, 0.1f); // static friction  
                                                    // restitution  
  
if(!mMaterial)  
    fatalError("createMaterial failed!");
```

Materials are owned by the `PxPhysics` object, and can be shared across multiple scenes. The material properties of two objects involved in a collision are combined in various ways. See the reference documentation for `PxMaterial` for details.

PhysX objects whose collision geometry is a triangle mesh or a height field can have a material per triangle.

Friction uses the coulomb friction model, which is based around two coefficients: the static friction coefficient and the dynamic friction coefficient (called kinetic friction). Friction resists relative lateral motion of two surfaces in contact. These two coefficients define a relationship between the normal force on each surface on the other and the amount of friction force that is applied to resist motion. Static friction defines the amount of friction that is applied between surfaces that are not moving lateral to each-other. Dynamic friction defines the amount of friction between surfaces that are moving relative to each-other.

The coefficient of restitution of two colliding objects is a fractional value representing the ratio of speeds after and before an impact, taken along the line of impact. A coefficient of restitution of 1 is said to collide elastically, while a coefficient of restitution less than 1 is said to collide inelastically.

Sleeping

When an actor does not move for a period of time, it is assumed that it will stay in that state in the future either until some external force acts on it that throws it out of equilibrium. If an actor is no longer simulated in order to save resources, this state is called sleeping. To query an actor's sleep state with the following method:

```
bool PxRigidDynamic::isSleeping() const;
```

It is however often more convenient to listen for events that the SDK generates when actors fall asleep or wake up. To receive the events, the flag `PxActorFlag::eSEND_SLEEP_NOTIFICATIONS` must be set for the actor:

```
void PxSimulationEventCallback::onWake(PxActor** actors, PxU32 count);  
void PxSimulationEventCallback::onSleep(PxActor** actors, PxU32 count);
```

See the section [Callback Sequence](#) and the subsection [Sleep state](#) for more information.

An actor goes to sleep when its kinetic energy is below a given threshold for a given time. Basically, every dynamic rigid actor has a wake counter which gets incremented at the simulation time step when the kinetic energy of the actor is below the threshold. However, if the energy is above the threshold after a simulation time step, the counter gets reset to a minimum default value and the whole process starts over. When the wake counter reaches zero, it does not get decremented any further until the actor is ready to go to sleep. Please note that a zero wake counter does not necessarily mean the actor has to be asleep, it only indicates that it is ready to go to sleep. There are several factors that might keep an actor awake for a while longer.

The energy threshold as well as the minimum amount of time an actor has to be awake before it can go to sleep can be manipulated using the following methods:

```
void PxRigidDynamic::setSleepThreshold(PxReal threshold);  
PxReal PxRigidDynamic::getSleepThreshold() const;
```

```
void PxRigidDynamic::setWakeCounter(PxReal wakeCounterValue);  
PxReal PxRigidDynamic::getWakeCounter() const;
```

Note: For kinematic actors, special sleep rules apply. A kinematic actor is considered sleeping unless a target pose has been set (in which case it will stay awake until the next simulation step where no target pose has been set anymore). As a result, `setWakeCounter()` is not allowed to use `setWakeCounter()` for kinematic actors. The wake state of a kinematic actor is solely defined based on whether a target pose has been set.

If a dynamic rigid actor is sleeping, the following state is guaranteed:

- The wake counter is zero.
- The linear and angular velocity is zero.
- There is no force update pending.

When an actor gets inserted into a scene, it will be considered asleep if the wake counter is above hold, else it will be treated as awake.

In general, a dynamic rigid actor is guaranteed to be awake if at least one of the following conditions holds:

- The wake counter is positive.
- The linear or angular velocity is non-zero.
- A non-zero force or torque has been applied.

As a consequence, the following calls will wake the actor up automatically:

- `PxRigidDynamic::setWakeCounter()`, if the wake counter value is less than or equal to hold.
- `PxRigidBody::setLinearVelocity()`, `::setAngularVelocity()`, if the velocity is non-zero.
- `PxRigidBody::addForce()`, `::addTorque()`, if the force or torque is non-zero.

In addition, the following calls and events wake an actor up:

- `PxRigidDynamic::setKinematicTarget()` in the case of a kinematic actor (this call also sets the wake counter to a positive value).

- `PxRigidActor::setGlobalPose()`, if the `autowake` parameter is set to `true`.
- Simulation gets disabled for a `PxRigidActor` if the actor has the flag `PxActorFlag::eDISABLE_SIMULATION`.
- `PxScene::resetFiltering()`.
- `PxShape::setSimulationFilterData()`, if the subsequent re-filtering of the shape pair to transition between suppressed, trigger and contact.
- Touch with an actor that is awake.
- A touching rigid actor gets removed from the scene (this is the default behavior, but can be specified by the user, see note further below).
- Contact with a static rigid actor is lost.
- Contact with a dynamic rigid actor is lost and this actor is awake in the next simulation step.
- The actor gets hit by a two-way interaction particle.

Note: When removing a rigid actor from the scene or a shape from a scene, it is possible to specify whether to wake up the objects that were touching the removed object in the previous simulation step. See the API comments in `PxScene::removeActor()` and `PxRigidActor::detachShape()` for details.

To explicitly wake up a sleeping object, or force an object to sleep, use:

```
void PxRigidDynamic::wakeUp();
void PxRigidDynamic::putToSleep();
```

Note: It is not allowed to use these methods for kinematic actors. The kinematic actor is solely defined based on whether a target pose has been reached.

The API reference documents exactly which methods cause an actor to wake up or go to sleep.

Sleep state change events

As mentioned above, PhysX provides an event system that reports changes in the sleep state of actors.

state of dynamic rigid bodies during *PxScene::fetchResults()*:

```
void PxSimulationEventCallback::onWake(PxActor** actors, PxU32 count) const  
void PxSimulationEventCallback::onSleep(PxActor** actors, PxU32 count) const
```

It is important to understand the correct usage of these events, and their implications:

- A body added since the previous *fetchResults()* or *flushSimulation()* generate an event, even if no sleep state transition occurred.
- If there have been multiple changes in a body's sleep state since the previous *fetchResults()* or *flushSimulation()*, PhysX will report only the most recent transition.

Sometimes it is desirable to detect transitions between awake and sleeping bodies. Suppose a sleeping body *B* wakes up during a simulation application, the counter is incremented, and during the next simulation application, *B* goes back to sleep. Even though *B*'s sleep state did not change during simulation, since the previous *fetchResults()*, and so an *onWake()* event will be generated. When the counter is incremented again in response to this event, its value will be incremented by two.

To use sleep state events to detect transitions, a record of the sleep state of each body of interest has to be kept, for example in a hash. When processing an event, the hash can be used to check whether there has been a transition.

Kinematic Actors

Sometimes controlling an actor using forces or constraints is not precise or flexible. For example moving platforms or character controls manipulate an actor's position or have it exactly follow a specific path. A scheme is provided by kinematic actors.

A kinematic actor is controlled using the `PxRigidBodyDynamic::setKinematicTarget()`. Each simulation step PhysX moves the actor to its target position, regardless of forces, gravity, collision, etc. Thus one must continually call `setKinematicTarget()` for each kinematic actor, to make them move along their path. The movement of a kinematic actor affects dynamic actors with which it collides. A kinematic actor constrained with a joint. The actor will appear to have infinite mass and push dynamic actors out of the way.

To create a kinematic actor, simply create a regular dynamic actor and set the `eKINEMATIC` flag:

```
PxRigidBody::setRigidBodyFlag(PxRigidBodyFlag::eKINEMATIC, true);
```

Use the same function to transform a kinematic actor back to a regular dynamic actor. While you do need to provide a mass for the kinematic actor as for all dynamic actors, the mass will not actually be used for anything while the actor is in kinematic mode.

Caveats:

- It is important to understand the difference between `PxRigidBodyDynamic::setKinematicTarget()` and `PxRigidBodyActor::setGlobalPose()`. While `setGlobalPose()` would also move the actor to the desired position, it would not make that actor properly interact with other objects. In particular, while the kinematic actor would not push away other dynamic actors in its path, it would go right through them. The `setGlobalPose()` function can still be used if one simply wants to teleport a kinematic actor to a new position.

- A kinematic actor can push away dynamic objects, but nothing prevents a kinematic actor from pushing a dynamic actor against a static actor or another kinematic actor. As a result, the squished dynamic actor can penetrate the geometry it has been pushed into.
- There is no interaction or collision between kinematic actors. However, it is possible to request contact information for kinematic pairs by setting the `PxSceneFlag::eENABLE_KINEMATIC_PAIRS` flag. The `PxSceneFlag::eENABLE_KINEMATIC_STATIC_PAIRS` flag also gets set.

Active Transforms

Note: the active transforms are currently deprecated. See next paragraph for its replacement.

The active transforms API provides an efficient way to reflect actor transform in a PhysX scene to an associated external object such as a render mesh.

When a scene's `fetchResults()` method is called an array of *PxActiveTransform* is generated, each entry in the array contains a pointer to the actor that moved and its new transform. Because only actors that have moved will be included in the array, this approach is potentially much more efficient than, for example, analyzing the scene individually.

The example below shows how to use active transforms to update a render object.

```
// update scene
scene.simulate(dt);
scene.fetchResults();

// retrieve array of actors that moved
PxU32 nbActiveTransforms;
PxActiveTransform* activeTransforms = scene.getActiveTransforms(nbActiveTransforms);

// update each render object with the new transform
for (PxU32 i=0; i < nbActiveTransforms; ++i)
{
    MyRenderObject* renderObject = static_cast<MyRenderObject*>(a
    renderObject->setTransform(activeTransforms[i].actor2World);
}
```

Note: `PxSceneFlag::eENABLE_ACTIVETRANSFORMS` must be set for the active transforms array to be generated.

Note: Since the target transform for kinematic rigid bodies is set by the actor's `setKinematicTargetTransform` method, they can be excluded from the list by setting the flag `PxSceneFlag::eEXCLUDE_KINEMATICS_FROM_ACTIVE_ACTORS`.

Active Actors

The active actors API provides an efficient way to reflect actor transforms from a PhysX scene to an associated external object such as a render mesh.

When a scene's `fetchResults()` method is called an array of active *Px* actors is returned. Because only actors that have moved will be included in the list this approach is much more efficient than, for example, analyzing each actor in the scene.

The example below shows how to use active actors to update a render

```
// update scene
scene.simulate(dt);
scene.fetchResults();

// retrieve array of actors that moved
PxU32 nbActiveActors;
PxActor** activeActors = scene.getActiveActors(nbActiveActors);

// update each render object with the new transform
for (PxU32 i=0; i < nbActiveActors; ++i)
{
    MyRenderObject* renderObject = static_cast<MyRenderObject*>(a
    renderObject->setTransform(activeActors[i]->getGlobalPose());
}
```

Note: `PxSceneFlag::eENABLE_ACTIVE_ACTORS` must be set on the active actors array to be generated.

Note: Since the target transform for kinematic rigid bodies is set by the scene, they can be excluded from the list by setting the flag `PxSceneFlag::eEXCLUDE_KINEMATICS_FROM_ACTIVE_ACTORS`.

Dominance

Dominance is a mechanism to enable dynamic bodies to dominate each other effectively. It imbues the dominant body in a pair with infinite mass. This mass modification occurs within the constraint solver and, as such, can override the effects of contact modification on the bodies in a pair. Similar effects can be achieved through local contact modification but dominance has the advantage of being handled within the SDK so does not incur the additional memory and performance cost of contact modification.

Each actor must be assigned a dominance group ID. This is a 5-bit value in the range [0, 31]. As such, you are restricted to at-most 32 dominance groups. By default, all actors are placed in dominance group 0. An actor can be assigned to a dominance group using the following method on PxActor:

```
virtual void setDominanceGroup(PxDominanceGroup dominanceGroup) =
```

Dominance is defined by 2 real numbers in the following struct:

```
struct PxDominanceGroupPair
{
    PxDominanceGroupPair(PxReal a, PxReal b)
        : dominance0(a), dominance1(b) {}
    PxReal dominance0;
    PxReal dominance1;
};
```

And dominance between two dominance groups can be configured using the following method on PxScene:

```
virtual void setDominanceGroupPair(PxDominanceGroup group1, PxDominanceGroup group2,
    const PxDominanceGroupPair& dominance) = 0;
```

The user can define 3 different states for a given PxDominanceGroupPair. A value of 0 indicates that both bodies have equal dominance. This is the default behavior.

indicates that body B dominates body A. * 0 : 1. This indicates that body B.

Any values other than 0 and 1 are not valid in a PxDominanceGroupPair both sides of the PxDominanceGroupPair is also invalid. These values to be scales applied to the bodies' respective inverse mass and dominance value of 0 would therefore equate to an infinite mass body.

The following example sets two actors, actorA and actorB, into different and configures the dominance group to make actorA dominate actorB:

```
PxRigidDynamic* actorA = mPhysics->createRigidDynamic(PxTransform
PxRigidDynamic* actorB = mPhysics->createRigidDynamic(PxTransform

actorA->setDominanceGroup(1);
actorB->setDominanceGroup(2);

mScene->setDominanceGroupPair(1, 2, PxDominanceGroupPair(0.f, 1.f
```

Dominance values will not affect joints. Local mass modification is performed using the following methods on PxJoint:

```
virtual void setInvMassScale0(PxReal invMassScale) = 0;
virtual void setInvMassScale1(PxReal invMassScale) = 0;
virtual void setInvInertiaScale0(PxReal invInertiaScale) = 0;
virtual void setInvInertiaScale1(PxReal invInertiaScale) = 0;
```

As previously mentioned, dominance does not permit values other than dominance values are applied uniformly to both the inverse mass and Joints and contacts through contact modification permit defining separate and inverse inertia scales, which accept any values within the range [0, 1] so can be used to achieve a wider range of effects than dominance can

Dominance can produce some very peculiar results if misused. For example, A, B and C configured in the following way:

- Body A dominates body B
- Body B dominates body C

- Body C dominates body A

In this situation, body A cannot push body C directly. However, it can push body B into body C.

Solver Iterations

When the motion of a rigid body is constrained either by contacts or joints, the solver comes into play. The solver satisfies the constraints on the body by iterating over all the constraints restricting the motion of the body a certain number of times. The more iterations, the more accurate the results become. The solver iteration counts are 10 position iterations and 1 velocity iteration. Those counts may be set for a body using the following function:

```
void PxRigidBodyDynamic::setSolverIterationCounts(PxU32 minPositionIt
```

Typically it is only necessary to significantly increase these values for joints and a small tolerance for joint error. If you find a need to use a tolerance of 30, you may wish to reconsider the configuration of your simulation.

The solver groups contacts into friction patches; friction patches are groups of contacts which share the same materials and have similar contact normals. The solver permits a maximum of 32 friction patches per contact manager (pair of contacts). If more than 32 friction patches are produced, which may be due to very complex geometry or very large contact offsets, the solver will ignore the remaining contacts. A warning will be issued in checked/debug builds when this happens.

Immediate Mode

In addition to simulation using a PxScene, PhysX offers a low-level simulation "immediate mode". This provides an API to access the low-level contact constraint solver. This approach currently supports only CPU rigid bodies, support articulations, clothing or particles.

The immediate mode API is defined in PxImmediateMode.h and demonstrating its usage in "SnippetImmediateMode".

The API provides a function to perform contact generation:

```
PX_C_EXPORT PX_PHYSX_CORE_API bool PxGenerateContacts(const PxGeo
                const PxReal contactDistance, const PxReal meshContactMar
```

This function takes a set of pairs of PxGeometry objects located at : performs collision detection between the pairs. If the pair of geometries are generated, which are reported to contactRecorder. In addition, it is cached in contactCache to accelerate future queries between these pairs. Any memory required for this cached information will be allocated using

In addition, the immediate mode provides APIs for the constraint solver functions to create bodies used by the solver:

```
PX_C_EXPORT PX_PHYSX_CORE_API void PxConstructSolverBodies(const
PX_C_EXPORT PX_PHYSX_CORE_API void PxConstructStaticSolverBody(co
```

In addition to constructing the bodies, PxConstraintSolverBodies also provides gravitational acceleration into the bodies velocities.

The following function is optional and is used to batch constraints:

```
PX_C_EXPORT PX_PHYSX_CORE_API PxU32 PxBatchConstraints(PxSolverCo
                PxSolverConstraintDesc* outOrderedConstraintDescs);
```

Batching constraints reorders the provided constraints and produces batches that can be used by the solver to accelerate constraint solving by solving independent constraints and solving them in parallel using multiple registers. This process is entirely optional and can be bypassed if not desired. It will change the order in which constraints are processed, which can affect the performance of the solver.

The following methods are provided to create contact constraints:

```
PX_C_EXPORT PX_PHYSX_CORE_API bool PxCreateContactConstraints(PxConstraintAllocator& allocator, PxReal invDt, PxReal bo
```

This method can be provided with the contacts produced by PxGenerateContacts or contacts produced by application-specific contact generation approaches.

The following methods are provided to create joint constraints:

```
PX_C_EXPORT PX_PHYSX_CORE_API bool PxCreateJointConstraints(PxConstraintAllocator& allocator, PxVec3* linearMotionVelocity, PxVec3* angularMotionVelocity)  
PX_C_EXPORT PX_PHYSX_CORE_API bool PxCreateJointConstraintsWithSh
```

The methods provide a mechanism for the application to define joint constraints. The application can make use of PhysX PxConstraint objects, which create the constraints.

The following method solves the constraints:

```
PX_C_EXPORT PX_PHYSX_CORE_API void PxSolveConstraints(PxConstraintAllocator& allocator, PxVec3* linearMotionVelocity, PxVec3* angularMotionVelocity)
```

This method performs all required position and velocity iterations and updates the delta velocities and motion velocities, which are stored in F.linearMotionVelocity/linear/angularMotionVelocity respectively.

The following method is provided to integrate the bodies' final positions and velocities.

bodies' velocities to reflect the motion produced by the constraint solver

An example of how the immediate mode can be used
SnippetImmediateMode.

Enhanced Determinism

PhysX provides limited deterministic simulation. Specifically, the result will be identical between runs if simulating the exact same scene (same order) using the same time-stepping scheme and same PhysX on the same platform. The simulation behavior is not influenced by the threads that are used.

However, the results of the simulation can change if actors are inserted. In addition, the overall behavior of the simulation can change if additional actors are added or if some actors are removed from the scene. This means that the particular collection of actors can change depending on whether other actors are in the scene or not, irrespective of whether these actors actually interact with each other. This behavioral property is usually tolerable but there are cases where it is not acceptable.

To overcome this issue, PhysX provides the flag `PxSceneFlag::eENABLE_ENHANCED_DETERMINISM`, which provides enhanced determinism. Specifically, provided the application inserts the actors in a fixed order, with this flag raised, the simulation of an island will be identical to other islands in the scene. However, this mode sacrifices some performance for additional determinism.

Axis locking

It is possible to restrict motion along or around specific world-space axes using the `PxRigidDynamicLockFlag`. For example, the below code snippet demonstrates how to restrict a `PxRigidDynamic` to a two-dimensional simulation. In this case, the `PxRigidDynamic` can rotate only around the Z-axis and translate only along the X and Y axes:

```
PxRigidDynamic* dyn = physics.createRigidDynamic(PxTransform(PxVec3(1, 1, 1), PxQuat::IDENTITY));
...
//Lock the motion
dyn->setRigidDynamicLockFlags(PxRigidDynamicLockFlag::eLOCK_LINEAR_X | PxRigidDynamicLockFlag::eLOCK_LINEAR_Y);
```

It is legal to restrict movement or rotation around any combination of axes, giving you a range of freedom.

Callback Sequence

The simplest type of simulation callbacks are the events. Using callbacks can simply listen for events and react as required, provided the callback that SDK state changes are forbidden. This restriction may be a bit surprising since the SDK permits writes to an inactive back-buffer while the simulation is running. The callbacks, however, are not called from within the simulation thread, but from a separate thread that processes the results of the simulation (via `fetchResults()`). The key point here is that `fetchResults()` processes the results of the simulation meaning that writing to the SDK from an event callback can be a particular problem. To avoid this fragility it is necessary to impose the rule that SDK state changes are not permitted from an event callback.

Inside `fetchResults()`, among other things, the buffers are swapped. This means that properties of each object's internal simulation state are visible to the user. Some event callbacks happen before this swap, and some happen after. The events that happen before are:

- `onTrigger`
- `onContact`
- `onConstraintBreak`

When these events are received in the callback, the shapes, actors, etc. are in the state they were in immediately before the simulation started. This is particularly important if these events were detected early on during the simulation, before objects have moved (or even moved) forward. For example, a pair of shapes that get an `onContact()` event while they are in contact will still be in contact when the call is made, even though they have since bounced apart again after `fetchResults()` returns.

On the other hand, these events are sent after the swap:

- `onSleep`
- `onWake`

Sleep information is updated after objects have been integrated, so

send these events after the swap.

To 'listen' to any of these events it is necessary to subclass `PxSimulationEventCallback` so that the various virtual functions may be overridden as desired. An instance of this subclass can then be registered per `PxScene::setSimulationEventCallback` or `PxSceneDesc::simulationEventCallback`. Following these steps alone will ensure that constraint break events are reported. One further step is required to report sleep and wake events. In the expense of reporting all sleep and wake events, actors identified as wakeable require the flag `PxActorFlag::eSEND_SLEEP_NOTIFICATIONS` to receive `onContact` and `onTrigger` events it is necessary to set a flag `onContact` callback for all pairs of interacting objects for which events are required. The filter shader callback can be found in Section [Collision Filtering](#).

Simulation memory

PhysX relies on the application for all memory allocation. The primary `PxAllocatorCallback` interface required to initialize the SDK:

```
class PxAllocatorCallback
{
public:
    virtual ~PxAllocatorCallback() {}
    virtual void* allocate(size_t size, const char* typeName, con
        int line) = 0;
    virtual void deallocate(void* ptr) = 0;
};
```

After the self-explanatory function argument describing the size of the three function arguments are an identifier name, which identifies the type the `__FILE__` and `__LINE__` location inside the SDK code where the allocation occurred. More details of these function arguments can be found in the PhysX API

Note: An important change since 2.x: The SDK now requires that the memory returned be 16-byte aligned. On many platforms `malloc()` returns memory aligned, but on Windows the system function `_aligned_malloc()` provides

Note: On some platforms PhysX uses system library calls to determine the type name, and the system function that returns the type name may call the allocator. If you are instrumenting system memory allocations, you may observe unexpected behavior. To prevent PhysX requesting type names, disable allocation names by calling the method `PxFoundation::setReportAllocationNames()`.

Minimizing dynamic allocation is an important aspect of performance. PhysX provides several mechanisms to control and analyze memory usage, which are discussed in turn.

Scene Limits

The number of allocations for tracking objects can be minimized by adjusting the capacities of scene data structures, using either `PxSceneDesc::limits` or the function `PxScene::setLimits()`. It is useful to note that these limits represent hard limits, meaning that PhysX will automatically perform fallback allocations if the number of objects exceeds the scene limits.

16K Data Blocks

Much of the memory PhysX uses for simulation is held in a pool of blocks. The initial number of blocks allocated to the pool can be controlled by `PxSceneDesc::nbContactDataBlocks`, while the maximum number of blocks that can be in the pool is governed by `PxSceneDesc::maxNbContactDataBlocks`. If PhysX internally needs more blocks than `nbContactDataBlocks` then it will automatically allocate further blocks to the pool until the number of blocks reaches `maxNbContactDataBlocks`. If PhysX subsequently needs more blocks than the maximum number of blocks, it will simply start dropping contacts and joint constraints. When this happens, the error is passed to the error stream in the `PX_CHECKED` configuration.

To help tune `nbContactDataBlocks` and `maxNbContactDataBlocks` it is possible to query the number of blocks currently allocated to the pool using `PxScene::getNbContactDataBlocksUsed()`. It can also be useful to query the maximum number of blocks that can ever be allocated to the pool using `PxScene::getMaxNbContactDataBlocksUsed()`.

Unused blocks can be reclaimed using `PxScene::flushSimulation()`. When `flushSimulation()` is called any allocated blocks not required by the current scene state will be freed. Additionally, a number of other blocks may be reused by the application. Additionally, a number of other blocks are freed by shrinking them to the minimum size required by the scene.

Scratch Buffer

A scratch memory block may be passed as a function argument to `PxScene::simulate`. As far as possible, PhysX will internally allocate from the scratch memory block, thereby reducing the need to perform fallback allocations from `PxAllocatorCallback`. The block may be reused by the application.

PxScene::fetchResults() call, which marks the end of simulation. One scratch memory block is that it must be a multiple of 16K, and it must be

In-place Serialization

PhysX objects can be stored in memory owned by the application using a serialization and deserialization mechanism. See [Serialization](#) for details.

PVD Integration

Detailed information about memory allocation can be recorded and displayed in the Visual Debugger. This memory profiling feature can be configured using the `trackOutstandingAllocations` flag when calling `PxCreatePhysics()`, or the `PxVisualDebuggerConnectionFlag::eMEMORY` when connecting to the Visual Debugger using `PxVisualDebuggerExt::createConnection()`.

Completion Tasks

A completion task is a task that executes immediately after `PxScene::simulate()`. If PhysX has been configured to use worker threads then `PxScene::simulate()` runs simulation tasks on the worker threads and will likely exit before the work is completed. The completion task must complete the work necessary to complete the scene update. As a consequence, a completion task would first need to call `PxScene::fetchResults(true)`. `fetchResults(true)` blocks until all worker threads started during `simulate()` have finished their work. After calling `fetchResults(true)`, the completion task can perform any physics work deemed necessary by the application:

```
scene.fetchResults(true); game.updateA(); game.updateB(); ... game.update();
```

The completion task is specified as a function argument in `PxScene::simulate()`. Details can be found in the PhysAPI documentation.

Synchronizing with Other Threads

An important consideration for substepping is that `simulate()` and `classed` as write calls on the scene, and it is therefore illegal to read scene while those functions are running. For the `simulate()` function it is the distinction between running and ongoing. In this context, it is illegal to read a scene before `simulate()` exits. It is perfectly legal, however, to read after `simulate()` has exited but before the worker threads that started during the call have completed their work.

Note: PhysX does not lock its scene graph, but it will report an error if it detects that multiple threads make concurrent calls to the same scene. All read calls.

Substepping

For reasons of fidelity simulation or better stability it is often desired frequency of PhysX be higher than the update rate of the application. To do this is just to call `simulate()` and `fetchResults()` multiple times:

```
for(PxU32 i=0; i<substepCount; i++)
{
    ... pre-simulation work (update controllers, etc) ...
    scene->simulate(substepSize);
    scene->fetchResults(true);
    ... post simulation work (process physics events, etc) ...
}
```

Sub-stepping can also be integrated with the completion task feature function. To illustrate this, consider the situation where the scene is graphics component signals that it has completed updating the render. Here, the completion task will naturally run after `simulate()` has exited. I block with `fetchResults(true)` to ensure that it waits until both `simulate()` have completed their sequential work. When the completion task is a next work item will be to query the graphics component to check if a required or if it can exit. In the case that another `simulate()` step is required to pass a completion task to `simulate()`. A tricky point here is that cannot submit itself as the next completion task because it would recursion. A solution to this problem might to be to have two completion stores a reference to the other. Each completion task can then pass `simulate()`:

```
scene.fetchResults(true);
if(!graphics.isComplete())
{
    scene.simulate(otherCompletionTask);
}
```

Split sim

As an alternative to `simulate()`, you can split the simulation into two phases: `collide()` and `advance()`. For some properties, called write-through properties, modifications during the `collide()` phase will be seen immediately by the `advance()` phase. This allows `collide()` to begin before the data required for `advance()` is available and to run in parallel with game logic that generates inputs to the simulation, particularly useful for animation logic generating kinematic targets, applying forces to bodies. The write-through properties are listed below:

```
addForce()/addTorque()/clearForce()/clearTorque()  
setAngularVelocity()/setLinearVelocity()  
setKinematicTarget()  
wakeUp()  
setWakeCounter()
```

When using the split sim, a physics simulation loop would look like this:

```
scene.collide(dt)  
scene.fetchCollision()  
scene.advance()  
scene.fetchResults()
```

Any other sequence of API calls is illegal. The SDK will issue error messages if you can't. You can interleave the physics-dependent game logic between `collide()` and

```
scene.collide(dt)  
physics-dependent game logic(animation, rendering)  
scene.fetchCollision()
```

`fetchCollision()` will wait until `collide()` has finished before it updates properties in the SDK. Once `fetchCollision()` has completed, any game logic performed on the objects in the executing scene will be buffered and will not be performed until the simulation and a call to `fetchResults()` has completed. The SDK will take the write-through properties into account when computing the new sets of poses for the actors being simulated.

Split fetchResults

The `fetchResults()` method is available in both a standard and split form offers some advantages over the standard `fetchResult()` method because it allows the user to parallelize processing of contact reports, which can be expensive in complex scenes.

A simplistic way to use `split fetchResults` would look something like this:

```
gSharedIndex = 0;

gScene->simulate(1.0f / 60.0f);

//Call fetchResultsStart. Get the set of pair headers
const PxContactPairHeader* pairHeader;
PxU32 nbContactPairs;
gScene->fetchResultsStart(pairHeader, nbContactPairs, true);

//Set up continuation task to be run after callbacks have been processed
callbackFinishTask.setContinuation(*gScene->getTaskManager(), NULL);
callbackFinishTask.reset();

//process the callbacks
gScene->processCallbacks(&callbackFinishTask);

callbackFinishTask.removeReference();

callbackFinishTask.wait();

gScene->fetchResultsFinish();
```

The user is free to use their own task/threading system to process the contacts. The PhysX scene provides a utility function that processes the callbacks in parallel threads, which is used in this code snippet. This method takes a continuation task to be run when the tasks processing callbacks have completed. In completion task raises an event that can be waited upon to notify that callback processing has completed.

This feature is demonstrated in `SnippetSplitFetchResults`. In order to

approach, contact notification callbacks must be thread-safe. For approach to be beneficial, contact notification callbacks need to be amount of work to benefit from multi-threading them

[NVIDIA PhysX SDK 3.4.1 Documentation »](#)
[User's Guide »](#)

[previous](#) | [next](#)

Tuning Shape Collision Behavior

Shapes used for contact generation influence the motion of the dynamics. They are attached to the bodies through contact points. The constraint solver generates the contact points to keep the shapes resting or moving without passing through each other. Shapes have two important parameters that control how the solver generates contact points between them, which in turn are central for the behavior of colliding or stacking: `contactOffset` and `restOffset`. They are set using `PxShape::setContactOffset()` and `PxShape::setRestOffset()` respectively. The `restOffset` value is used directly. Collision detection always operates on a pair of shapes, and it always considers the sum of the offsets of the two shapes as the `contactDistance` and `restDistance` respectively.

Collision detection is able to generate contact points between two shapes even if they are still a distance apart, when they are exactly touching, or when they are overlapping. To make the discussion simpler, we treat interpenetration as a negative distance between two shapes can be positive, zero, or negative. Separation distance at which collision detection will start to generate contacts. If `contactOffset` is less than zero, meaning that PhysX will always generate contacts when the shapes are penetrating (unless collision detection between the two shapes is in some way disabled, such as with filtering). By default, when using metric units and the `PxTolerancesScale`, `contactOffset` is 0.02, which means `contactDistance` is 0.02 centimeters. So when two shapes approach each other within 4 centimeters, contact points will be generated until they are again moved further apart than 4 centimeters.

The generation of contact points does not however mean that a force will be immediately applied at these locations to separate the shapes, or to prevent further motion in the direction of penetration. This would make the simulation time step selected to be tiny, which is not desirable for performance. Instead, we want the force at the contact to smoothly increase as penetration increases until it reaches a value sufficiently high to stop the penetrating motion. The distance at which this maximum force is applied is `restDistance`, because at this distance two shapes stacked on each other will be in equilibrium and come to rest. When the shapes are for some reason p

much that they have a distance below `restDistance`, an even greater push them apart until they are at `restDistance` again. The variation of distance changes is not necessarily linear, but it is smooth and continuous, resulting in a pleasing simulation even at large time steps.

There are a few different things to consider when choosing `contactOffset` for shapes. Typically the same values can be used for all shapes in a simulation. The goal is typically to have the shapes appear to stack such that they are exactly touching, like bodies do in the real world. If the collision shapes are sized to be the exact same size as the graphics shapes, a `contactOffset` of zero is needed. If the collision shapes are an epsilon bigger than the graphics shapes, a positive `contactOffset` is needed. If the collision shapes are an epsilon smaller than the graphics shapes, a negative `contactOffset` is correct. This will let the larger collision shapes touch each other until the smaller graphics shapes touch too. `contactOffset` values of zero are practical for example if there are problems with sliding on surfaces where the penetration based contact generation has more trouble than a separation one, resulting in a smoother slide.

Once the `restOffset` is determined, the `contactOffset` should be chosen slightly larger. The rule of thumb is to make the difference between the two as large as possible that still effectively avoids jitter at the time step size the simulation uses. A larger time step will need the difference to be larger. The drawback of setting a large `contactOffset` is that contacts will be generated sooner as two shapes approach, which increases the number of contacts that the simulation has to worry about. This can impact performance. Also, the simulation code often makes the assumption that the contact points are close to the convex shapes' surface. If the contact offset is very large, this assumption breaks down which could lead to behavior artefacts.

Contact Modification

Under certain circumstances, it may be necessary to specialize collision solver for example to implement sticky contacts, give objects the appearance of swimming inside each other, or making objects go through apparent simple approach to achieve such effects is to let the user change contact properties after they have been generated by collision detection, but before the solver. Because both of these steps occur within the scene simulation, `Scene::simulate()` must be used.

The callback occurs for all pairs of colliding shapes for which the user has set the pair flag `PxPairFlag::eMODIFY_CONTACTS` in the filter shader.

To listen to these modify callbacks, derive from the class `PxContactModifyCallback`

```
class MyContactModification : public PxContactModifyCallback
{
    ...
    void onContactModify(PxContactModifyPair* const pairs, PxU32 count);
};
```

And then implement the function `onContactModify` of `PxContactModifyCallback`

```
void MyContactModification::onContactModify(PxContactModifyPair * pairs, PxU32 count)
{
    for(PxU32 i=0; i<count; i++)
    {
        ...
    }
}
```

Every pair of shapes comes with an array of contact points, that have properties that can be modified, such as position, contact normal, and time being, restitution and friction properties of the contacts cannot be modified. `PxModifiableContact` and `PxContactSet` for properties that can be modified.

In addition to modifying contact properties, it is possible to:

- Set target velocities for each contact
- Limit the maximum impulse applied at each contact
- Adjust inverse mass and inverse inertia scales separately for each

Conveyor belt-like effects can be achieved by setting target velocities achieved by having target velocities running in tangential directions to but the solver does also support target velocities in the direction of the c

The user can limit the impulse applied at each contact by limiting the applied at each contact. This can be useful to produce "soft" contact (the impression of energy dissipation due to compression or to limit the a dynamic body due to a kinematic collision. Note that limiting the max potentially lead to additional penetration and bodies passing through ea

Adjusting mass and inertia scales can be used to tune how contacts bodies affect the bodies' linear and angular velocities respectively. contact pair has a separate inverse mass and inverse inertia scale. initialized to 1 and can be adjusted as part of the callback. Note that th local mass modification within the contact pair and affect all contacts wi

Uniformly scaling a body's inverse mass and inverse inertia by the sa the body behaving like a body that is either heavier or lighter dependen used. Providing inverse mass/inverse inertia scales < 1 results in th heavier; providing scales > 1 result in the body appearing lighter. Fo mass/inertia scales of 0.5 result in the body appearing to have double t inverse mass/inertia scales of 4 would result in the body appearing to h original mass. Providing inverse mass/inertia scale of 0 results in the bc has infinite mass.

However, it is also possible to non-uniform scale a body's inverse mass by providing different values to a body's inverse mass and inverse example, it is possible to reduce or increase the amount of angular ve result of contacts by adjusting just the inverse inertia scale. The use-ca modification are extremely game-dependent but may involve, for

interactions between a player's vehicle and traffic vehicles in an a game, where the player's car is expected to be bumped by traffic ve would be extremely frustrating to the player if the car was to spin-out collision. This could also be achieved by making the traffic vehicles mu player's vehicle but this may make the traffic vehicles appear "too li damage the player's immersion.

When performing local mass modification, the impuls `PxSimulationEventCallback::onContact()` will be relative to the locally the bodies involved in that contact. Therefore, this reported impuls accurately reflect the change in momentum caused by a given contact. this issue, we have provided the following methods in the rigid body ex the linear and angular impulse and velocity change caused by a conta modification:

```
static void computeLinearAngularImpulse(const PxRigidBody& body,
    const PxVec3& point, const PxVec3& impulse, const PxReal invM
    const PxReal invInertiaScale, PxVec3& linearImpulse, PxVec3&

static void computeVelocityDeltaFromImpulse(const PxRigidBody& bo
    const PxTransform& globalPose, const PxVec3& point, const PxV
    const PxReal invMassScale, const PxReal invInertiaScale, PxVe
    PxVec3& deltaAngularVelocity);
```

These methods return separate linear and angular impulse and velocit reflect the fact that the mass and inertia may have been non-uniformly mass modification has been used, it may be necessary to extract s angular impulses for each contact point, for each body in the pair. Ple helper functions are provided to provide users with accurate impulse va means mandatory. For simple use-cases, e.g. triggering effects or impulse thresholds, the single impulse value reported by the contac perfectly acceptable even when local mass modification has been use mass modification has been used and the impulse values are bei complex behaviors, e.g. balance control for a ragdoll, then these h most-likely be required to achieve correct behavior. Please note th articulations, `computeLinearAngularImpulse` will return the correct ir

respective articulation link. However, `computeVelocityDeltaFromImpulse` correct velocity changes for an articulation link because it does not take other links of the articulation into account.

In addition, the following considerations must be made when using local modification:

- Force thresholding for callbacks will be based on the scaled contact reports. This was calculated using the scaled mass/inertia so using mass scaling may require these thresholds to be re-tuned.
- Maximum impulse clamping occurs in the solver on an impulse based on the scaled masses/inertias. As a result, the magnitude of the impulse calculated from `computeLinearAngularImpulse(...)` may exceed the clamped value in situations where mass scaling was used. In situations where un-scaled masses were used, the magnitude of the magnitude of linear impulse will be $\text{massScale} * \text{maxImpulse}$ and angular impulse will not exceed `maxImpulse`.

There are a couple of special requirements for the callback due to the fact that it is called from deep inside the SDK. In particular, the callback should be thread-safe. In other words, the SDK may call `onContactModify()` from any thread concurrently (i.e., asked to process sets of contact modification pairs simultaneously).

The contact modification callback can be set using the `contactModifyCallback` property of `PxSceneDesc` or the `setContactModifyCallback()` method of `PxScene`.

Contact reporting

Here is an example for a contact event function from SampleSubmarine

```
void SampleSubmarine::onContact(const PxContactPairHeader& pairHeader,
    const PxContactPair* pairs, PxU32 nbPairs)
{
    for(PxU32 i=0; i < nbPairs; i++)
    {
        const PxContactPair& cp = pairs[i];

        if(cp.events & PxPairFlag::eNOTIFY_TOUCH_FOUND)
        {
            if((pairHeader.actors[0] == mSubmarineActor) ||
                (pairHeader.actors[1] == mSubmarineActor))
            {
                PxActor* otherActor = (mSubmarineActor == pairHeader.actors[0] ? pairHeader.actors[1] : pairHeader.actors[0]);
                Seamine* mine = reinterpret_cast<Seamine*>(otherActor);
                // insert only once
                if(std::find(mMinesToExplode.begin(), mMinesToExplode.end(), mine) == mMinesToExplode.end())
                    mMinesToExplode.push_back(mine);

                break;
            }
        }
    }
}
```

SampleSubmarine is a subclass of PxSimulationEventCallback. onContact is called for each pair for which the requested contact events have been triggered. The function is only interested in eNOTIFY_TOUCH_FOUND events, which are raised when two shapes start to touch. In fact it is only interested in touch events of the type TOUCH_FOUND. This is checked in the second if-statement. It then goes on to assume that the other actor is a mine (which works in this example because the sample is configured to report contact reports will get sent when a submarine actor is involved). After that, the mine is added to a set of mines that should explode during the next update.

Note: By default collisions between kinematic rigid bodies and kinematic rigid bodies will not get reported. To enable these reports raise the `PxSceneFlag::eENABLE_KINEMATIC_PAIRS` or `::eENABLE_KINEMATIC_STATIC_PAIRS` flag respectively by calling `PxScene::setFlag()`.

Frequently, users are only interested in contact reports, if the force of it is above a certain threshold. This allows to reduce the amount of reported pairs processed. To take advantage of this option the following additional settings are necessary:

- Use `PxPairFlag::eNOTIFY_THRESHOLD_FORCE_FOUND`, `::eNOTIFY_THRESHOLD_FORCE_PERSISTS`, `::eNOTIFY_THRESHOLD_FORCE_LOST` instead of `::eNOTIFY_TOUCH_FOUND` etc.
- Specify the threshold force for a dynamic rigid body using `PxRigidDynamic::setContactReportThreshold()`. If the body collides with a static object and the contact force is above the threshold, a report will be generated (if enabled according to the `PxPairFlag` setting of the pair). If two dynamic bodies both have a force threshold specified then the lower one is used.

Note: If a dynamic rigid body collides with multiple static objects, the sum of all those contacts will get summed up and used to compare against the threshold. In other words, even if the impact force against each individual object is below the threshold, the contact reports will still get sent for each pair if the sum of forces exceeds the threshold.

Contact Reports and CCD

If continuous collision detection (CCD) with multiple passes is enabled, an object might bounce on and off the same object multiple times during

step. By default, only the first impact will get reported as a *eNOTIFY* event in this case. To get events for the other impacts too *eNOTIFY_TOUCH_CCD* has to be raised for the collision pair. *eNOTIFY_TOUCH_CCD* events for the non primary impacts. For per the system can not always tell whether the contact pair lost touch in a CCD pass and thus can also not always tell whether the contact persisted. *eNOTIFY_TOUCH_CCD* just reports when the two collisions detected as being in contact during a CCD pass.

Extracting Contact information

The onContact simulation event permits read-only access to all contact information. In previous releases, this information was available as a PxContactPair object. However, PhysX 3.3 introduces a new format for compressed contact streams. The contact information is now provided in a compressed format for a given PxContactPair depending on certain conditions. Depending on the shapes involved, the properties of the contacts, material properties, and whether the contacts are modifiable.

As there are a large number of combinations of different formats, there are two built-in mechanisms to access the contact data. The first approach is a mechanism to extract contacts from a user buffer and can be used as follows:

```
void MySimulationCallback::onContact(const PxContactPairHeader& p
    const PxContactPair* pairs, PxU32 nbPairs)
{
    const PxU32 bufferSize = 64;
    PxContactPairPoint contacts[bufferSize];
    for(PxU32 i=0; i < nbPairs; i++)
    {
        const PxContactPair& cp = pairs[i];

        PxU32 nbContacts = pairs[i].extractContacts(contacts, buf
        for(PxU32 j=0; j < nbContacts; j++)
        {
            PxVec3 point = contacts[j].position;
            PxVec3 impulse = contacts[j].impulse;
            PxU32 internalFaceIndex0 = contacts[j].internalFaceIn
            PxU32 internalFaceIndex1 = contacts[j].internalFaceIn
            //...
        }
    }
}
```

This approach requires copying data to a temporary buffer in order to access it. The second approach allows the user to iterate over the contact information directly from their own copy:

```

void MySimulationCallback::onContact(const PxContactPairHeader& p
    const PxContactPair* pairs, PxU32 nbPairs)
{
    for(PxU32 i=0; i < nbPairs; i++)
    {
        const PxContactPair& cp = pairs[i];

                PxContactStreamIterator iter(cp.contactPatche

        const PxReal* impulses = cp.contactImpulses;

        PxU32 flippedContacts = (cp.flags & PxContactPairFlag::eI
        PxU32 hasImpulses = (cp.flags & PxContactPairFlag::eINTER
        PxU32 nbContacts = 0;

        while(iter.hasNextPatch())
        {
            iter.nextPatch();
            while(iter.hasNextContact())
            {
                iter.nextContact();
                PxVec3 point = iter.getContactPoint();
                PxVec3 impulse = hasImpulses ? dst.normal * impul

                PxU32 internalFaceIndex0 = flippedContacts ?
                    iter.getFaceIndex1() : iter.getFaceIndex0();
                PxU32 internalFaceIndex1 = flippedContacts ?
                    iter.getFaceIndex0() : iter.getFaceIndex1();
                //...
                nbContacts++;
            }
        }
    }
}

```

This approach is slightly more involved because it requires the user to r all of the data but also consider conditions like whether the pair h; whether impulses have been reported with the pair. However, this ap over the data in-place may be more efficient because it doesn't require

Extra Contact Data

Since pointers to the actors of a contact pair are provided in contact properties, they can be read directly within the callback. However, the pose of an actor usually refers to the time of impact. If for some reasons the velocity response is of interest, then the actor can not provide that information. It is possible to get the actor velocity or the pose at impact if those properties were requested by the user while the simulation was running (in such a case the newly requested information will be returned). Last but not least, if CCD with multiple passes is used, a moving object might bounce on and off the same object multiple times and velocities for each such impact can not get extracted from the actor velocity callback. For these scenarios, the PhysX SDK provides an additional callback. This callback can hold all sorts of extra information related to the contact pair. This information can be requested per pair through the pair flags *PxPairFlags* (see the API documentation for *PxPairFlag::ePRE_SOLVER_VELOCITY*, *PxPairFlag::ePOST_SOLVER_VELOCITY*, *PxPairFlag::eCONTACT_EVENT_POSE* for details). If requested, the extra data is available as a member of the *PxContactPairHeader* structure. The stream is parsed by using the predefined iterator *PxContactPairExtraDataIterator* or custom parsing code (see the implementation of *PxContactPairExtraDataIterator* for details about the format of the stream).

Example code:

```
void MySimulationCallback::onContact(const PxContactPairHeader& pairHeader,
    const PxContactPair* pairs, PxU32 nbPairs)
{
    PxContactPairExtraDataIterator iter(pairHeader.extraDataStream,
        pairHeader.extraDataStreamSize);
    while(iter.nextItemSet())
    {
        if (iter.postSolverVelocity)
        {
            PxVec3 linearVelocityActor0 = iter.postSolverVelocity[0];
            PxVec3 linearVelocityActor1 = iter.postSolverVelocity[1];
            ...
        }
    }
}
```

Continuous Collision Detection

When continuous collision detection (or CCD) is turned on, the affect not go through other objects at high velocities (a problem also known as tunneling). To enable CCD, three things need to happen:

1. CCD needs to be turned on at scene level:

```
PxPhysics* physx;
...
PxSceneDesc desc;
desc.flags |= PxSceneFlag::eENABLE_CCD;
...
```

2. Pairwise CCD needs to be enabled in the pair filter:

```
static PxFilterFlags filterShader(
    PxFilterObjectAttributes attributes0,
    PxFilterData filterData0,
    PxFilterObjectAttributes attributes1,
    PxFilterData filterData1,
    PxPairFlags& pairFlags,
    const void* constantBlock,
    PxU32 constantBlockSize)
{
    pairFlags = PxPairFlag::eSOLVE_CONTACT;
    pairFlags |= PxPairFlag::eDETECT_DISCRETE_CONTACT;
    pairFlags |= PxPairFlag::eDETECT_CCD_CONTACT;
    return PxFilterFlags();
}

...

desc.filterShader = testCCDFilterShader;
physx->createScene(desc);
```

3. CCD need to be enabled for each PxRigidBody that requires CCD:

```
PxRigidBody* body;
...
```

```
body->setRigidBodyFlag(PxRigidBodyFlag::eENABLE_CCD, true);
```

Once enabled, CCD only activates between shapes whose relative speed is less than the sum of their respective CCD velocity thresholds. These velocities are automatically calculated based on the shape's properties and support n

Contact Notification and Modification

CCD supports the full set of contact notification events that are supported by discrete collision detection. For details on contact notification, see the [Callbacks](#).

CCD supports contact modification. To listen to these modify callbacks, you must implement the class `PxCCDContactModifyCallback`:

```
class MyCCDContactModification : public PxCCDContactModifyCallback
{
    ...
    void onCCDContactModify(PxContactModifyPair* const pairs, PxU
};
```

And then implement the function `onContactModify` of `PxContactModifyCallback`:

```
void MyContactModification::onContactModify(PxContactModifyPair *
{
    for(PxU32 i=0; i<count; i++)
    {
        ...
    }
}
```

This `onContactModify` callback operates using the same semantic as discrete collision detection contact modification callbacks. For further details, see the [documentation on Callbacks](#).

As with discrete collision detection, CCD will only emit contact modification events for a given pair if the user has specified the pair flag `PxPairFlag::eMODIFY_`

filter shader.

Triggers

Currently, shapes flagged with `PxShapeFlag::eTRIGGER_SHAPE` will not be processed by the CCD. However, it is possible to get trigger events from CCD by not flagging shapes as `PxShapeFlag::eTRIGGER_SHAPE` and instead configuring the filter to detect the following state for pairs involving trigger shapes:

```
pairFlags = PxPairFlag::eTRIGGER_DEFAULT |  
PxPairFlag::eDETECT_CCD_CONTACT; return PxFilterFlag::eDEFAULT;
```

It should be noted that not flagging shapes as `PxShapeFlag::eTRIGGER_SHAPE` result in the triggers being more expensive. Therefore, this work is reserved for use only in situations where important trigger events will be detected by the CCD.

Tuning CCD

The CCD should generally work without any tuning. However, there are several parameters that can be adjusted:

1. `PxSceneDesc.ccdMaxPasses`: This variable controls the number of passes the CCD will perform. This is defaulted to 1, meaning that all objects are attempted to be processed to the TOI of their first contact. Any remaining time after the TOI of the first contact will be dropped. Increasing this value permits the CCD to run multiple passes, which reduces the likelihood of time being dropped but can increase the cost of the CCD.
2. `PxRigidBody::setMinCCDAdvanceCoefficient(PxReal advanceCoefficient)`: This method allows you to adjust the amount by which the CCD advances per pass. By default, this value is 0.15, meaning that CCD will advance by $0.15 * \text{ccdThreshold}$, where `ccdThreshold` is a value computed from the scene. This value acts as a lower-bound of the maximum amount of time that could be spent on a pair before there is a chance that the object could have tunneled. This value can be adjusted to increase the amount of time spent on a pair, but it should be used with caution as it can increase the cost of the CCD.

0.15 improves the fluidity of motion without risking missed collisions. A value of 0.0 can negatively impact fluidity but will reduce the likelihood of tunneling at the end of a frame. Increasing this value may increase the likelihood of tunneling. This value should only be set in the range [0,1].

3. Enabling the flag `PxSceneFlag::eDISABLE_CCD_RESWEEP` (`PxSceneDesc.flags`): Enabling this flag disables CCD resweeps. This prevents missed collisions as the result of ricochets but has the potential overhead of the CCD. In general, enabling this advancement means that objects will not pass through the static environment but no longer dynamic objects with CCD enabled will not pass through each other.
4. `PxRigidBody::setRigidBodyFlag(PxRigidBodyFlag::eENABLE_CCD, true)`: Enabling this flag enables the application of friction forces if they are disabled by default. As the CCD operates using only linear motion, friction inside CCD can cause visual artefacts.

Performance Implications

Enabling CCD on a scene/all bodies in a scene should be relatively efficient. Some performance impact even when all the objects in the scene are moving slowly. A great deal of effort has been put into optimizing the CCD and additional overhead should only constitute a very small portion of the time when the objects are moving slowly. As the objects' velocities increase, overhead will increase, especially if there are a lot of high-speed objects. Increasing the number of CCD passes can make the CCD more expensive. CCD will terminate early if the additional passes aren't required.

Limitations

The CCD system is a best-effort conservative advancement scheme. It uses a fixed number of CCD substeps (defaulted to 1) and drops any remaining time only dropped on high-speed objects at the moment of impact so it is not. However, this artefact can become noticeable if you simulate an object

small/thin relative to the simulation time-step that the object could be accelerated by gravity from rest for 1 frame, i.e. a paper-thin rigid body would always be moving at above its CCD velocity threshold and could result in a proportion of simulation time being dropped for that object and any other objects on that island as it (any objects whose bounds overlap the bounds of that object) cause a noticeable slow-down/stuttering effect caused by the object becoming noticeably out-of-sync with the rest of the simulation. It is recommended that paper-thin/tiny objects should be avoided if possible.

It is also recommended that you filter away CCD interactions between objects that are constrained together, e.g. limbs in the same ragdoll. Allowing CCD interactions between limbs of the same ragdoll could increase the cost of CCD and also potentially result in some CCD interactions being dropped unnecessarily. CCD interactions are automatically disabled for objects that are part of an articulation.

Raycast CCD

The PhysX SDK supports an alternative CCD implementation based on raycasts. This "raycast CCD" algorithm is available in PhysX Extensions, and it is implemented as a code snippet ("SnippetRaycastCCD"). Contrary to the built-in CCD algorithm within the PhysX SDK, this cheaper and simpler alternative version is located outside of the SDK itself.

After the traditional `simulate/fetchResults` calls, the system performs raycasts from the shapes' center positions to double-check that they did not tunnel. If tunneling is detected for an object, it is moved back to a previous position along the ray, in a position that is safe from tunneling. Then next frame, the SDK's contact generation takes over and generates contacts for the object's motion. There are some subtle details not described here, but this is the gist of the nutshell.

Since it is raycast-based, the solution is not perfect. In particular, small objects can still go through the static world if the ray goes through a crack between static shapes or a small hole in the world (like the keyhole from a door). Also, dynamic-to-dynamic contacts are very approximate. It only works well for fast-moving dynamic objects and not for slow-moving dynamic objects. Other known limitations are that it is only implemented for `PxRigidDynamic` objects (not for `PxArticulationLink` objects) and only for actors with one shape (not for "compounds").

However the implementation should be able to prevent important objects from tunneling in a game world, provided the world is watertight. The code is very small and easy to modify, and its performance is often better overall than for the built-in CCD algorithm. It is a valuable alternative if the default CCD becomes too expensive.

Speculative CCD

In addition to sweep-based CCD, PhysX also provides a cheaper but less accurate approach called speculative CCD. This approach functions differently to the sweep-based CCD in that it operates entirely as part of the discrete simulation by inflating collision spheres on object motion and depending on the constraint solver to ensure that objects do not pass through each-other.

This approach generally works well and, unlike the sweep-based CCD, does not require setting speculative CCD on kinematic actors. However, there are cases where objects do not pass through each-other. As an example, if the scene accelerates an actor (as a result of a collision or joint) such that the actor passes through objects during that time-step, speculative CCD can result in tunneling.

To enable this feature, raise `PxRigidBodyFlag::eENABLE_SPECULATIVE_CCD` on the rigid body that requires CCD:

```
PxRigidBody* body;
...
body->setRigidBodyFlag(PxRigidBodyFlag::eENABLE_SPECULATIVE_CCD,
```

Unlike the sweep-based CCD, this form of CCD does not require setting it on either the scene or on the pair in the filter shader.

Note that this approach works best with PCM collision detection. It may also work well if the legacy SAT-based collision detection approach is used.

This feature can work in conjunction with the sweep-based CCD, e.g. a scene where a kinematic actor has speculative CCD enabled but dynamic rigid bodies use sweep-based CCD. However, if speculative CCD is used on kinematics in conjunction with sweep-based CCD, it is important to ensure that interactions between the kinematic actor and the CCD-enabled dynamic actors do not also enable sweep-based CCD interactions otherwise the sweep-based CCD may overrule the speculative CCD, leading to poor behavior.

Persistent Contact Manifold (PCM)

The PhysX SDK provides two types of collision detection:

1. Default collision detection

The default collision detection system uses a mixture of SAT (Separating Axis Theorem) and distance-based collision detection to generate full contact manifold. It finds the potential contacts in one frame, so it lends itself better to static contacts. This approach is stable for small contact offsets and rest offsets but may not be correct contact points when large offsets are used because it approximates contact points in these situations by plane shifting.

2. Persistent Contact Manifold (PCM)

PCM is a fully distance-based collision detection system. PCM generates contacts when two shapes first come into contact. It recycles and updates contacts from the previous frame in the manifold and then it generates the subsequent frame if the shapes move relative to each other more than a certain amount or if a contact was dropped from the manifold. If too many contacts are dropped from the manifold due to a large amount of relative motion in a frame, contact generation is re-run. This approach is quite efficient in terms of performance. However, because PCM potentially generates fewer contacts than the default collision detection, it might reduce stacking stability when simulating tall stacks. As this approach is distance-based, it will generate contact points for arbitrary contact offsets/rest offsets.

To enable PCM, set the flag in the `PxSceneDesc::flags`:

```
PxSceneDesc sceneDesc;  
  
sceneDesc.flags |= PxSceneFlag::eENABLE_PCM;
```


Joint Basics

A joint constrains the way two actors move relative to one another. A typical example would be to model a door hinge or the shoulder of a character. Joints are provided in the PhysX extensions library and cover many common scenarios, but in some cases that are not met by the joints packaged with PhysX, you can implement your own. Since joints are implemented as extensions, the pattern for creating them is different from other PhysX objects.

Creation of simple joints and limits is demonstrated in the `SnippetJoint` example.

To create a joint, call the joint's creation function:

```
PxRevoluteJointCreate(PxPhysics& physics,
                    PxRigidActor* actor0, const PxTransform& localFrame0,
                    PxRigidActor* actor1, const PxTransform& localFrame1)
```

This has the same pattern for all joints: two actors, and for each actor a local frame.

One of the actors must be movable, either a *PxRigidDynamic* or a *PxActor*. The other may be of one of those types, or a *PxRigidStatic*. Use a `NULL` to indicate an implicit actor representing the immovable global reference frame.

Each `localFrame` argument specifies a constraint frame relative to the actor. Each joint defines a relationship between the global positions and origins of the two frames that will be enforced by the PhysX constraint solver. In this example, the joint constrains the origin points of the two frames to be coincident and to share a common axis, but allows the two actors to rotate freely relative to one another.

PhysX supports six different joint types:

- a **fixed** joint locks the orientations and origins rigidly together
- a **distance** joint keeps the origins within a certain distance range
- a **spherical** joint (also called a *ball-and-socket*) keeps the origins together

the orientations to vary freely.

- a **revolute** joint (also called a *hinge*) keeps the origins and x-axes together, and allows free rotation around this common axis.
- a **prismatic** joint (also called a *slider*) keeps the orientations identical and allows the origin of each frame to slide freely along the common x-axis.
- a **D6** joint is a highly configurable joint that allows specification of degrees of freedom either to move freely or be locked together. It can be used to model a wide variety of mechanical and anatomical joints, but is somewhat more difficult to configure than the other joint types. This joint is covered in detail below.

All joints are implemented as plugins to the SDK through the `PxJoint` interface. A number of the properties for each joint are configured using the `PxJointFlags` enumeration.

Note: As in the rest of the PhysX API, all joint angles for limits and drives are specified in radians.

Visualization

All standard PhysX joints support debug visualization. You can visualize each actor, and also any limits the joint may have.

By default, joints are not visualized. To visualize a joint, set its visualization flag and the appropriate scene-level visualization parameters:

```
scene->setVisualizationParameter(PxVisualizationParameter::eJOINT_LIMITS, true);
scene->setVisualizationParameter(PxVisualizationParameter::eJOINT_DRIVES, true);
...
joint->setConstraintFlag(PxConstraintFlag::eVISUALIZATION)
```

Force Reporting

The force applied at a joint may be retrieved after simulation with a call

```
scene->fetchResults(...)  
joint->getConstraint().getForce(force, torque);
```

The force is resolved at the origin of actor1's joint frame.

Note that this force is only updated while the joint's actors are awake.

Breakage

All of the standard PhysX joints can be made *breakable*. A maximum torque may be specified, and if the force or torque required to maintain exceeds this threshold, the joint will break. Breaking a joint generates (see `PxSimulationEventCallback::onJointBreak`), and the joint no longer participates in simulation, although it remains attached to its actors until it is deleted.

By default the threshold force and torque are set to `FLT_MAX`, making joints unbreakable. To make a joint breakable, specify the force and torque thresholds:

```
joint->setBreakForce(100.0f, 100.0f);
```

A constraint flag records whether a joint is currently broken:

```
bool broken = (joint->getConstraintFlags() & PxConstraintFlag::eBROKEN) != 0;
```

Breaking a joint causes a callback via `PxSimulationEventCallback::onConstraintBreak`. In this callback, a pointer to the joint and its type are specified in the `externalReference` field of the `PxConstraintInfo` struct. If you have implemented your own `PxSimulationEventCallback`, use the `PxConstraintInfo::type` field to determine the dynamic type of the joint. Otherwise, simply cast the `externalReference` to a `PxJoint`:

```
class MySimulationEventCallback  
{  
    void onConstraintBreak(PxConstraintInfo* constraints, PxU32 count)  
    {  
        for(PxU32 i=0; i<count; i++)  
        {  
            PxJoint* joint = (PxJoint*) constraints->externalReference[i];  
            // Do something with the joint  
        }  
    }  
};
```

```

        PxJoint* joint = reinterpret_cast<PxJoint*>(constrain
        ...
    }
}
}

```

Projection

Under stressful conditions, PhysX' dynamics solver may not be able to the constraints specified by the joint. PhysX provides kinematic *projection* bring violated constraints back into alignment even when the solver fails a physical process and does not preserve momentum or respect collisions. It is best avoided if practical, but can be useful in improving simulation separation results in unacceptable artifacts.

By default projection is disabled. To enable projection, set the linear and angular tolerance values beyond which a joint will be projected, and set the constraint projection flag.

```

joint->setProjectionLinearTolerance(0.1f);
joint->setConstraintFlag(PxConstraintFlag::ePROJECTION, true);

```

Very small tolerance values for projection may result in jittering around the constraint.

A constraint with projection enabled can be part of a graph of rigid bodies and constraints. If this graph is acyclic, the algorithm will choose a root node, traverse the graph, and project the bodies towards the root. If the constraint graph has cycles, the algorithm will split the graph into subgraphs, dropping edges that create cycles, and do the projection separately on each subgraph. Please note that having more than one constraint attached to a fixed/kinematic rigid body in a graph does count as a cycle (for example, a rigid body connected with constraints and both ends attached to world or a rigid body with two constraints fight over the same body or conflicting projection directions). The projection direction will be chosen based on the following priorities (high to low):

- world attachment or a rigid static actor with a projecting constraint
- kinematic actor with a projecting constraint

- all dominant dynamic actor (has projecting constraints and all of projecting towards this dynamic)
- dominant dynamic actor (same as above but there is at least one constraint as well)
- partially dominant dynamic actor (has at least one one-way pr towards this dynamic and at least one one-way projecting constrain actor)
- world attachment or a rigid static actor without any projecting const
- kinematic actor without any projecting constraints
- dynamic actor with or without two-way projecting constraints 1 (among these, the one with the highest constraint count wins)

Limits

Some PhysX joints constrain not just relative rotation or translation, but *limits* on the range of that motion. For example, in its initial configuration allows free rotation around its axis, but by specifying and enabling a limit bounds may be placed upon the angle of rotation.

Limits are a form of collision, and like collision of rigid body shapes, s requires a *contactDistance* tolerance specifying how far from t configuration may be before the solver tries to enforce it. Note that e starts before the limit is violated, so the role played by contactDis analogous to the role a positive contactDistance value plays in collision contact makes the limit less likely to be violated even at high relative because the limit is active more of the time, the joint is more expensive

Limit configuration is specific to each type of joint. To set a limit, geometry and set the joint-specific flag indicating that the limit is enable

```
revolute->setLimit(PxJointAngularLimitPair(-PxPi/4, PxPi/4, 0.1f)
revolute->setRevoluteJointFlag(PxRevoluteJointFlag::eLIMIT_ENABLE
```

Limits may be either *hard* or *soft*. When a hard limit is reached, relative stop dead if the limit is configured with zero restitution, or bounce if the zero. When a soft limit is violated, the solver will pull the joint back toward a spring specified by the limit's spring and damping parameters. By default and without restitution, so when the joint reaches a limit motion will simulate softness for a limit, declare the limit structure and set the spring and damping directly:

```
PxJointAngularLimitPair limitPair(-PxPi/4, PxPi/4, 0.1f));
limitPair.spring = 100.0f;
limitPair.damping = 20.0f;
revolute->setRevoluteJointLimit(limitPair);
revolute->setRevoluteJointFlag(PxRevoluteJointFlag::eLIMIT_ENABLE
```

Note: Limits are not projected.

When using spring limits, the `eACCELERATION` flag is strongly recommended. It will automatically scale the strength of the spring according to the masses of the objects that the limit is acting upon, and can substantially reduce the CPU time required for good, stable behavior.

Actuation

Some PhysX joints may be actuated by a motor or a spring implicitly by the PhysX solver. While driving simulations with actuated joints is more expensive than applying forces, it can provide much more stable control of simulation. See [Prismatic Joint](#), and [Revolute Joint](#) for details.

Note: The force generated by actuation is not included in the force resolution solver, nor does it contribute towards exceeding the joint's breakage force.

Note: Changing the drive parameters for a joint, or activating or deactivating it does not wake sleeping bodies attached to the joint. If required, wake them manually.

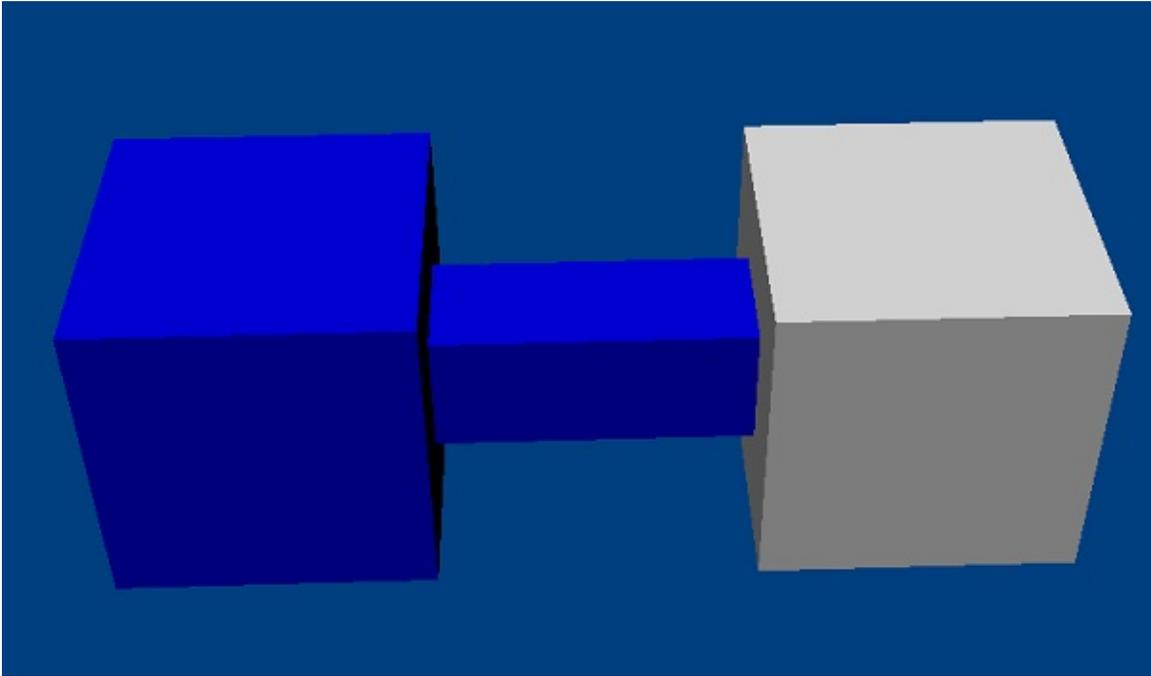
When using spring drives (in particular, drives on the D6 joint), the eAC is strongly recommended. This flag will automatically scale the strength according to the masses and inertias of objects that the limit is acting on, which can substantially reduce the amount of tuning required for good, stable behavior.

Mass Scaling

PhysX joints may apply scale to the mass and moment of inertia of bodies for the purposes of resolving a joint. For example, if you have a ragdoll of masses 1 and 10, PhysX will typically resolve the joint by changing the velocity of the lighter body much more than the heavier one. You can apply a mass scaling factor to the first body to make PhysX change the velocity of both bodies by an equal amount. To ensure the same property holds for both linear and angular velocity, you can also scale the inertia in accordance with the bodies' inertias as well. Applying mass scaling so that the joint sees similar effective masses and inertias makes the solver converge faster, which can make individual joints seem less rubbery or separated, and the overall behavior of the bodies appear less twitchy.

Many applications that prioritize visual behavior over adherence to physical laws can benefit from tuning these scale values. But if you use this feature, bear in mind that mass and inertia scaling is fundamentally nonphysical. In general, momentum is conserved, the energy of the system may increase, the force reported by the solver may be incorrect, and non-physical tuning of breakage thresholds and force limits may be required.

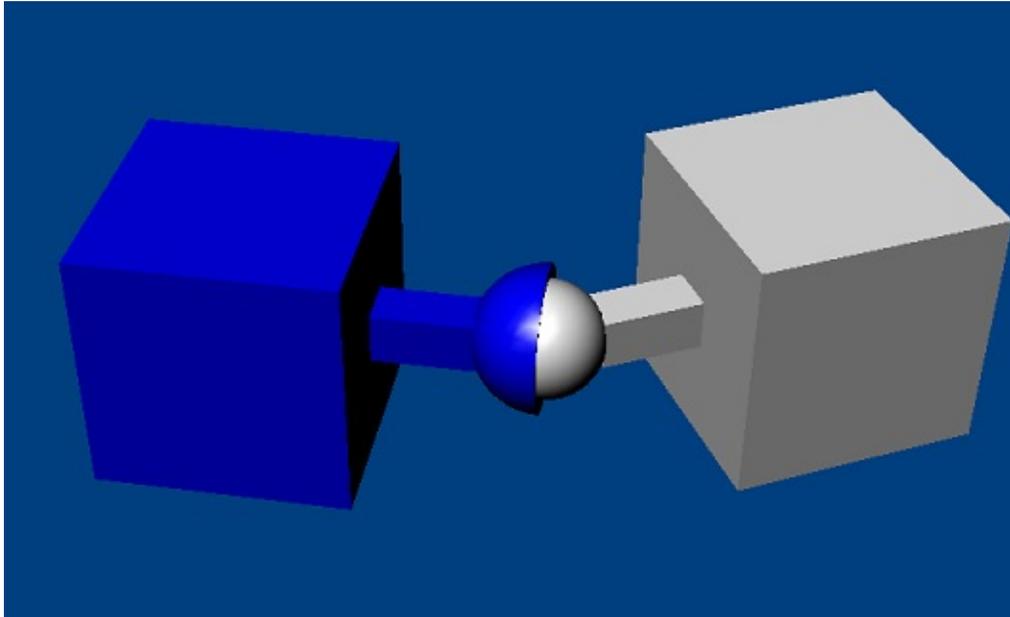
Fixed Joint



The fixed joint constrains two objects so that the positions and orientations of the constraint frames are the same.

Note: All joints are enforced by the dynamics solver, so although under simulation the objects will maintain their spatial relationship, there may be some numerical drift. A better alternative, which is cheaper to simulate and does not suffer from drift, is a single actor with multiple shapes. However fixed joints are useful, for example, if a joint must be breakable or report its constraint force.

Spherical Joint



A spherical joint constrains the origins of the actor's constraint frames to

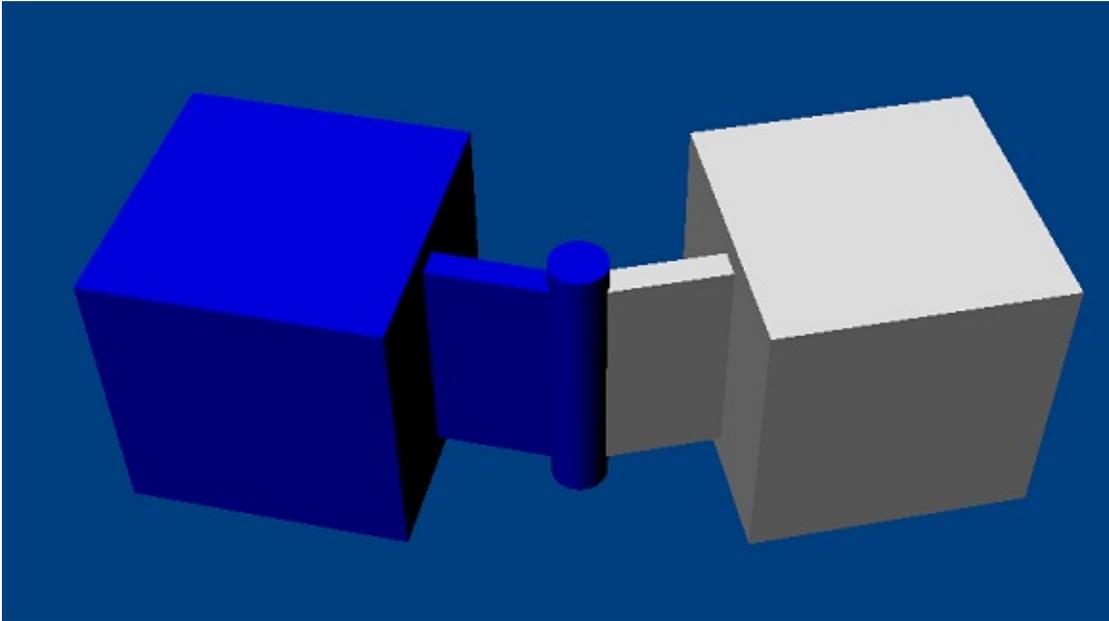
The spherical joint supports a cone limit, which constrains the angle between the axes of the two constraint frames. Actor1's X-axis is constrained by a limit cone around the x-axis of actor0's constraint frame. The allowed limit values are the angle around the y- and z- axes of that frame. Different values for the y- and z- axes are specified, in which case the limit takes the form of an elliptical angular cone.

```
joint->setLimitCone(PxJointLimitCone(PxPi/2, PxPi/6, 0.01f);  
joint->setSphericalJointFlag(PxSphericalJointFlag::eLIMIT_ENABLED
```

Note that very small or highly elliptical limit cones may result in solver jitter.

Note: *Visualization of the limit surface can help considerably in understanding its shape.*

Revolute Joint



A revolute joint removes all but a single rotational degree of freedom from a system. The axis along which the two bodies may rotate is specified by the common origin points of the joint frames and their common x-axis. In theory, all origin points along the axis are equivalent, but simulation stability is best in practice when the point of connection between the two bodies are closest.

The joint supports a rotational limit with upper and lower extents. The angle is measured about the x-axis of the joint frames, and increases from the y- and z- axes of the joint frames are coincident, and increases towards the z-axis:

```
joint->setLimit(PxJointLimitPair(-PxPi/4, PxPi/4, 0.01f);  
joint->setRevoluteJointFlag(PxRevoluteJointFlag::eLIMIT_ENABLED,
```

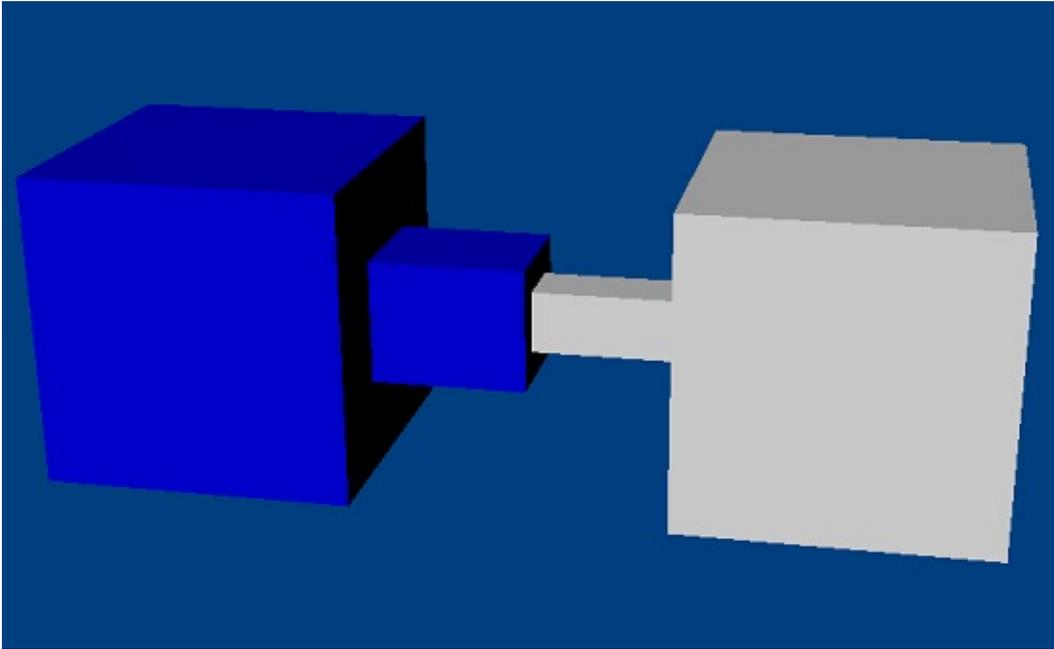
The joint also supports a motor which drives the relative angular velocity towards a user-specified target velocity. The magnitude of the force applied may be limited to a specified maximum:

```
joint->setDriveVelocity(10.0f);  
joint->setRevoluteJointFlag(PxRevoluteJointFlag::eDRIVE_ENABLED,
```

By default, when the angular velocity at the joint exceeds the target velocity as a brake; a freespinner flag disables this braking behavior.

The drive force limit for a revolute joint may be interpreted either as a force or torque depending on the value of `PxConstraintFlag::eDRIVE_LIMITS_ARE_FORCE`.

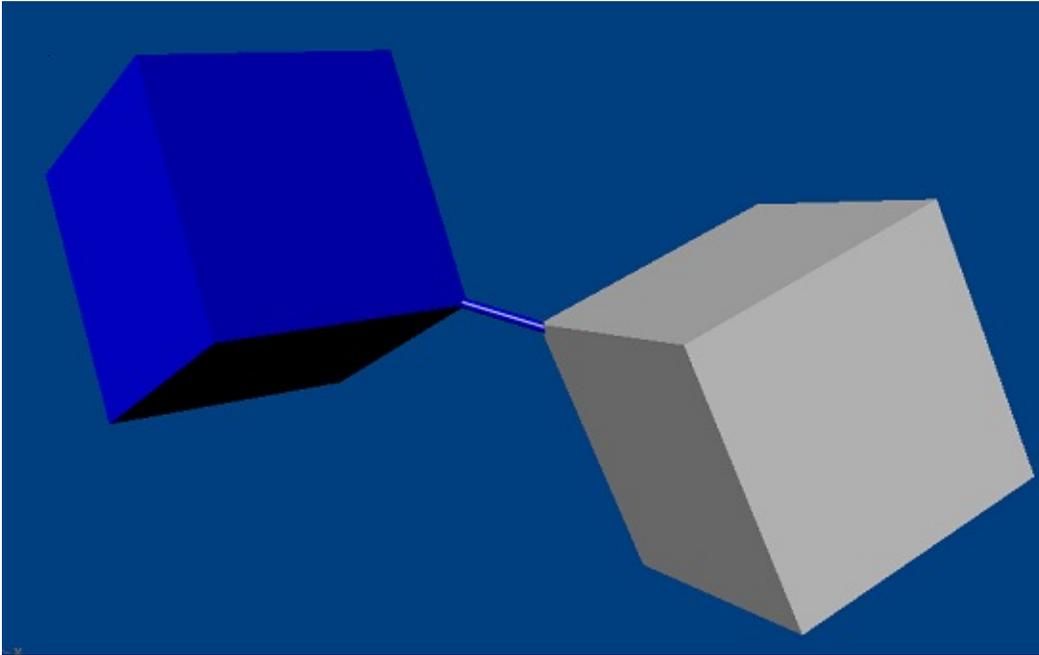
Prismatic Joint



A prismatic joint prevents all rotational motion, but allows the origin of frame to move freely along the x-axis of actor0's constraint frame. It supports a single limit with upper and lower bounds on the distance constraint frames' origin points:

```
joint->setLimit(PxJointLimitPair(-10.0f, 20.0f, 0.01f);  
joint->setPrismaticJointFlag(PxPrismaticJointFlag::eLIMIT_ENABLED
```

Distance Joint



The distance joint keeps the origins of the constraint frames within distance. The range may have both upper and lower bounds, v separately by flags:

```
joint->setMaxDistance(10.0f);  
joint->setDistanceJointFlag(eMAX_DISTANCE_ENABLED, true);
```

In addition, when the joint reaches the limits of its range motion beyond either be entirely prevented by the solver, or pushed back towards implicit spring, for which spring and damping parameters may be specif

D6 Joint

The D6 joint is by far the most complex of the standard PhysX joints. It behaves like a fixed joint - that is, it rigidly fixes the constraint frame. However, individual degrees of freedom may be unlocked to permit rotation around the x-, y- and z- axes, and translation along these axes.

Locking and Unlocking Axes

To unlock and lock degrees of freedom, use the joint's `setMotion` function:

```
d6joint->setMotion(PxD6Axis::eX, PxD6Motion::eFREE);
```

Unlocking translational degrees of freedom allows the origin point of the constraint frame to move along a subset of the axes defined by actor0's coordinate system. For example, unlocking just the X-axis creates the equivalent of a prismatic joint.

Rotational degrees of freedom are partitioned as *twist* (around the X-axis of the constraint frame) and *swing* (around the Y- and Z- axes). Different effects can be achieved by unlocking various combinations of twist and swing.

- if just a single degree of angular freedom is unlocked, the result is a revolute joint. It is recommended that if just one angular degree of freedom should be unlocked, it should be the twist degree, because the joint has various configuration optimizations that are designed for this case.
- if both swing degrees of freedom are unlocked but the twist degree is locked, the result is a *zero-twist* joint. The x-axis of actor1 swings freely about actor0 but twists to minimize the rotation required to align the constraint frame. This creates a kind of isotropic universal joint which avoids the problem of the 'engineering style' universal joint (see below) that is sometimes used for a ball-and-socket constraint. There is a nasty singularity at π radians (180 degrees).

limit should be used to avoid the singularity.

- if one swing and one twist degree of freedom are unlocked but the other is kept locked, a *zero-swing* joint results (often also called a *uni-swing* joint). For example the SWING1 (y-axis rotation) is unlocked, the x-axis of actor0 is free to remain orthogonal to the z-axis of actor0. In character applications, this joint is used to model an elbow swing joint incorporating the twist freedom of the lower arm or a knee swing joint incorporating the twist freedom of the lower leg. In other applications, these joints can be used as 'steered wheel' joints in which the wheel is free to rotate about its twist axis, while the free swing degree of freedom acts as the steering axis. Care must be taken with this combination due to its anisotropic behavior and singularities (beware the dreaded gimbal lock at $\pi/2$ radians (90 degrees), making the zero-twist joint a better behavior in most use cases.
- if all three angular degrees are unlocked, the result is equivalent to a ball-and-socket joint.

Three of the joints from PhysX 2 that have been removed from PhysX 3 and are not implemented as follows:

- The cylindrical joint (with axis along the common x-axis of the two actors) is given by the combination:

```
d6joint->setMotion(PxD6Axis::eX, PxD6Motion::eFREE);  
d6joint->setMotion(PxD6Axis::eTWIST, PxD6Motion::eFREE);
```

- the point-on-plane joint (with plane axis along the x-axis of actor0) is given by the combination:

```
d6joint->setMotion(PxD6Axis::eY, PxD6Motion::eFREE);  
d6joint->setMotion(PxD6Axis::eZ, PxD6Motion::eFREE);  
d6joint->setMotion(PxD6Axis::eTWIST, PxD6Motion::eFREE);  
d6joint->setMotion(PxD6Axis::eSWING1, PxD6Motion::eFREE);  
d6joint->setMotion(PxD6Axis::eSWING2, PxD6Motion::eFREE);
```

- the point-on-line joint (with axis along the x-axis of actor0's constraint frame) is implemented by the combination:

```
d6joint->setMotion(PxD6Axis::eX, PxD6Motion::eFREE);
d6joint->setMotion(PxD6Axis::eTWIST, PxD6Motion::eFREE);
d6joint->setMotion(PxD6Axis::eSWING1, PxD6Motion::eFREE);
d6joint->setMotion(PxD6Axis::eSWING2, PxD6Motion::eFREE);
```

Note: Angular projection is implemented only for the cases when two degrees of freedom are locked.

Limits

Instead of specifying that an axis is free or locked, it may also be specified that an axis is limited. D6 supports three different limits which may be used in any combination.

A single linear limit with only an upper bound is used to constrain any degrees of freedom. The limit constrains the distance between the origin of the constraint frame and the origin of the actor's frame when projected onto these axes. For example, the combination:

```
d6joint->setMotion(PxD6Axis::eX, PxD6Motion::eFREE);
d6joint->setMotion(PxD6Axis::eY, PxD6Motion::eLIMITED);
d6joint->setMotion(PxD6Axis::eZ, PxD6Motion::eLIMITED);
d6joint->setLinearLimit(PxJointLinearLimit(1.0f, 0.1f));
```

constrains the y- and z- coordinates of actor1's constraint frame to lie within a cylinder of radius 1 extending along the x-axis of actor1's frame. Since the x-axis is unconstrained, the effect is to constrain the origin of actor1's constraint frame to lie within a cylinder of radius 1 extending along the x-axis of actor1's frame.

The twist degree of freedom is limited by a pair limit with upper and lower bounds. The limit is identical to the limit of the revolute joint.

If both swing degrees of freedom are limited, a limit cone is generated, which is a subset of the spherical joint. As with the spherical joint, very small or highly overlapping limits may result in solver jitter.

If only one swing degree of freedom is limited, the corresponding angle is used to limit rotation. If the other swing degree is locked, the maximum is π radians (180 degrees). If the other swing degree is free, the maximum limit is $\pi/2$ radians (90 degrees).

Drives

The D6 has a linear drive model, and two possible angular drive models: *proportional derivative* drive, which applies a force as follows:

$$\text{force} = \text{spring} * (\text{targetPosition} - \text{position}) + \text{damping} * (\text{targetVelocity} - \text{velocity})$$

The drive model may also be configured to generate a proportional acceleration force, factoring in the masses of the actors to which the joint is attached. An acceleration drive is often easier to tune than force drive.

The linear drive model for the D6 has the following parameters:

- target position, specified in actor0's constraint frame
- target velocity, specified in actor0's constraint frame
- spring
- damping
- forceLimit - the maximum force the drive can apply (note that it is an impulse, depending on PxConstraintFlag::eDRIVE_LIMITS_ARI)
- acceleration drive flag

The drive attempts to follow the desired position input with the configured spring and damping properties. A physical lag due to the inertia of the driven body will occur; therefore, sudden step changes will result over several steps. Physical lag can be reduced by stiffening the spring or supplying

With a fixed position input and a zero target velocity, a position drive will drive position with the specified springing/damping characteristics:

```
// set all translational degrees free
```

```

d6joint->setMotion(PxD6Axis::eX, PxD6Motion::eFREE);
d6joint->setMotion(PxD6Axis::eY, PxD6Motion::eFREE);
d6joint->setMotion(PxD6Axis::eZ, PxD6Motion::eFREE);

// set all translation degrees driven:

PxD6Drive drive(10.0f, -20.0f, PX_MAX_F32, true);
d6joint->setDrive(PxD6JointDrive::eX, drive);
d6joint->setDrive(PxD6JointDrive::eY, drive);
d6joint->setDrive(PxD6JointDrive::eZ, drive);

// Drive the joint to the local(actor[0]) origin - since no angular
// dofs are free, the angular part of the transform is ignored

d6joint->setDrivePosition(PxTransform(1.0f));
d6joint->setDriveVelocity(PxVec3(PxZero));

```

Angular drive differs from linear drive in a fundamental way: it does not have an intuitive representation free from singularities. For this reason, the D6 angular drive models - twist and swing and SLERP (Spherical Linear Interpolation) differ in the way they estimate the path in quaternion space between the current orientation and the target orientation.

The two models differ in the way they estimate the path in quaternion space between the current orientation and the target orientation. In a SLERP drive, the path is estimated directly. In a twist and swing drive, it is decomposed into separate components and each component is interpolated separately. Twist and swing drives are useful in many situations; however, there is a singularity when driven to 180 degrees. In addition, the drive will not follow the shortest arc between two orientations. In contrast, SLERP drive will follow the shortest arc between a pair of orientations but may cause unintuitive changes in the joint's twist and swing.

The angular drive model has the following parameters:

- An angular velocity target specified relative to actor0's constraint
- An orientation target specified relative to actor0's constraint frame
- drive specifications for SLERP (slerpDrive), swing (swingDrive), and twist (twistDrive):
- spring - amount of torque needed to move the joint to its target orientation, proportional to the angle from the target (not used for a velocity constraint)

- damping - applied to the drive spring (used to smooth out oscillations around the drive target).
- forceLimit - the maximum torque the drive can apply (note that this is an impulsive torque, depending on the value of the parameter PxConstraintFlag::eDRIVE_LIMITS_ARE_FORCES)
- acceleration drive flag. If this flag is set the acceleration (rather than the torque) applied by the drive is proportional to the angle from the target.

Best results will be achieved when the drive target inputs are considered in terms of freedom and limit constraints.

Note: if any angular degrees of freedom are locked, the SLERP drive is ignored. If all angular degrees of freedom are unlocked, and parameter multiple angular drives, the SLERP parameters will be used.

Configuring Joints for Best Behavior

The behavior quality of joints in PhysX is largely determined by the attributes of the PXRigidBody which controls the solver iteration count. Better convergence can be achieved simply by adjusting the solver iteration count. The solver iteration count can also be configured to produce better convergence.

- the solver can have difficulty converging well when a light object is between two heavy objects. Mass ratios of higher than 10 are better scenarios.
- when one body is significantly heavier than the other, make the lighter body the second actor in the joint. Similarly, when one of the objects is static (the actor pointer is NULL) make the dynamic body the second actor.

A common use for joints is to move objects around in the world. Best results are achieved when the solver has access to the velocity of motion as well as the character's position.

- if you want a very stiff controller that moves the object to specific p consider jointing the object to a kinematic actor and use the function to move the actor.
- if you want a more springy controller, use a D6 joint with a driv desired position and orientation, and control the spring paran stiffness and damping. In general, acceleration drive is much easie drive.

When using mass scaling or when constraining bodies with infinite i axes, the reduction in degrees of freedom of the rigid bodies co inaccuracies in floating point calculation can produce arbitrarily stiff cc trying to correct unnoticeably small errors. This can appear, for exampl to perform 2D-simulation using infinite inertia to suppress velocity c simulation. In these cases, set the flag `PxConstraintFlag::eDISABLE_P` and set the `minResponseThreshold` on the constraint to a small value, result in such stiff constraint rows being ignored when encountered, an improve simulation quality.

Custom Constraints

It is also possible to add new joint types to PhysX. Use the existing PhysXExtensions library as a reference, and also the source for `Solver` which shows how to implement a Pulley Joint. Serializing custom objects is discussed in the chapter [Serialization](#), so the discussion here is limited to how to define the behavior in simulation. This is an advanced topic, and assumes familiarity with the mathematics underlying rigid body simulation. The presentation here is for a joint that constrains two bodies; the case for a static body is equivalent to a joint with infinite mass whose transform is the identity.

The functions which implement dynamic behavior of joints are PhysX shaders. They are similar in nature to the `PxFilterShader` (see [Collision Filtering](#)). In particular, they execute in parallel and asynchronously, and should not access any data not passed in as parameters.

To create a custom joint class, define the following:

- the functions which implement the behavior of the constraint. The functions are stateless, because they may be called simultaneously from multiple threads. Each function is called, PhysX passes a *constant block* which contains the joint configuration parameters (offsets, axes, limits etc).
- a static instance of `PxConstraintShaderTable` containing pointers to the functions.
- a class implementing the `PxConstraintConnector` interface, that connects the joint to PhysX.

Defining Constraint Behavior

There are two functions that define the joint behavior: the *solver pre-solve* function, which generates inputs to PhysX' velocity-based constraint solver, and the *solve* function, which allows direct correction of position error.

The processing sequence during simulation is as follows:

- in the `simulate()` function, before starting simulation the scene uses a copy of the joint's constant block (so that the joint's copy may be used during simulation without causing races).
- collision detection runs, and may wake bodies. If the joint connects two bodies, the simulation will ensure that either both bodies are awake, or neither.
- for every joint connected to an awake body, the simulation calls the `simulate` function.
- the solver updates body velocities and positions.
- if the constraint's `ePROJECTION` flag is set, the simulation calls the `project` function.

The Solver Preparation Function

The solver preparation function for a joint has the following signature:

```
PxU32 prepare(Px1DConstraint* constraints,
             PxVec3& bodyAWorldOffset,
             PxU32 maxConstraints,
             PxConstraintInvMassScale &invMassScale,
             const void* constantBlock,
             const PxTransform& bA2w,
             const PxTransform& bB2w);
```

The parameters are as follows:

- *constraints* is the output buffer of constraint rows.
- *bodyAWorldOffset* is the point, specified in world space as an offset from the origin of bodyA, at which the constraint forces act to enforce the joint. The solver ignores this value as the information is already encoded in the constraint's `bodyAWorldOffset` field. When reporting forces it is necessary to choose a point at which the forces are considered to act. For PhysX joints, the attachment point of the joint is used.
- *maxConstraints* is the size of the buffer, which limits the number of constraints that can be prepared.

that may be generated.

- *invMassScale* is the inverse mass scales which should be applied for the purpose of resolving the joint. In the standard joints, these are joint scaling parameters (see [Mass Scaling](#)).
- *constantBlock* is the simulation's copy of the joint constant block.
- *bA2w* is the transform of the first body. It is the identity transform if a NULL pointer was supplied in constraint creation.
- *bB2w* is the transform of the second body. It is the identity transform if a NULL pointer was supplied in constraint creation.

The role of the solver preparation function is to populate the buffer and provide the point of application for force reporting, and provide other properties. The return value is the number of Px1DConstraints generated in the buffer.

Notice that although the joint parameters (relative pose etc) are typical of an actor, the solver preparation function works with the transforms of rigid bodies. The constraint infrastructure (see [Data Management](#)) maintains consistency when, for example, the application modifies the transform of an actor.

Each Px1D constraint constrains one degree of freedom between two bodies. Its structure looks like this:

```
struct Px1DConstraint
{
    PxVec3          linear0;
    PxReal         geometricError;
    PxVec3          angular0;
    PxReal         velocityTarget;

    PxVec3          linear1;
    PxReal         minImpulse;
    PxVec3          angular1;
    PxReal         maxImpulse;

    union
```

```

{
    struct SpringModifiers
    {
        PxReal      stiffness;
        PxReal      damping;
    } spring;
    struct RestitutionModifiers
    {
        PxReal      restitution;
        PxReal      velocityThreshold;
    } bounce;
} mods;

PxReal      forInternalUse;
PxU16      flags;
PxU16      solveHint;
}

```

Each Px1DConstraint is either a hard constraint (for example, one axis soft constraint (for example, a spring). A joint may have a mixture constraint rows - for example, the actuated joint at a rag doll shoulder of

- 3 hard 1D-constraints which prevent the shoulder from separating.
- 3 hard 1D-constraints constraining the angular degrees of freedom
- 3 soft constraints simulating resistance to angular motion from mus

The constraint is treated as hard unless the Px1DConstraintFlag::eSPR

For both soft and hard constraints, the *solver velocity* for each row is the

$$v = \text{body0vel}.\text{dot}(\text{lin0}, \text{ang0}) - \text{body1vel}.\text{dot}(\text{lin1}, \text{ang1})$$

Hard Constraints

For a hard constraint, the solver attempts to generate:

- a set of motion solver velocities v_{Motion} for objects which, when the constraint errors, represented by the equation:

$$v_{\text{Motion}} + (\text{geometricError} / \text{timestep}) = \text{velocityTarget}$$

- a set of post-simulation solver velocities v_{Next} for the objects constraints:

$$v_{\text{Next}} = \text{velocityTarget}$$

The motion velocities are used for integration and then discarded. The velocities are the values that `getLinearVelocity()` and `getAngularVelocity`

There are two special options for hard constraints, both most often limits: restitution and velocity biasing. They are set by the constraint flag and `eKEEPBIAS`, are mutually exclusive, and restitution takes priority (restitution is set, biasing is ignored).

Restitution simulates bouncing (off a limit, for example). If the input velocity v_{Current} at the start of simulation exceeds the restitution velocity threshold, the velocity of the constraint will be set to:

$$\text{restitution} * -v_{\text{Current}}$$

and the input `velocityTarget` field will be ignored. To use `Px1DConstraintFlag::eRESTITUTION`.

Velocity biasing generates post-simulation velocities to satisfy the same as the motion velocities:

$$v_{\text{Next}} + (\text{geometricError} / \text{timestep}) = \text{velocityTarget}$$

This can be useful if, for example, the joint is approaching a limit but hasn't reached it. If the target velocity is 0 and the geometric error is the distance remaining to the limit, the solver will constrain the velocity below that required to violate the limit. The joint should then converge smoothly to the limit.

Soft Constraints

Alternatively, the solver can attempt to resolve the velocity constraint at the end of the simulation. In this case, the motion velocity v_{Motion} and post-simulation velocity v_{Post} are used. The solver solves the equation:

$$F = \text{stiffness} * -\text{geometricError} + \text{damping} * (\text{velocityTarget} - v)$$

where F is the constraint force.

Springs are fully implicit: that is, the force or acceleration is a function of the velocity after the solve. There is one special option that applies only to acceleration springs (`PxConstraintFlag::eACCELERATION`). With this option, the solver scales the magnitude of the force in accordance with the response of the system. Effectively it implicitly solves the equation:

$$\text{acceleration} = \text{stiffness} * -\text{geometricError} + \text{damping} * (\text{velocityTarget} - v)$$

Force Limits and Reporting

All constraints support limits on the minimum or maximum impulse applied. There is a special flag for force limits: `eHAS_DRIVE_FORCE_LIMIT`. If this flag is set, force limits will be scaled by the tire stiffness. The flag `PxConstraintFlag::eLIMITS_ARE_FORCES` is set for the constraint.

The flag `eOUTPUT_FORCE` on a 1D constraint determines whether the force for this row should be included in the constraint force output. The flag is used internally to determine joint breakage. For example, if creating a drive that breaks when the stress on the linear part exceeds a limit, set the flag for the linear equality rows but not the angular drive rows.

Solver Preprocessing

The joint solver attempts to preprocess hard constraints to improve solve time. The `solveHint` value controls preprocessing for each row:

- if the constraint is a hard equality constraint with unbounded impulse

impulse limits are `-PX_MAX_FLT` and `PX_MAX_FLT`)
`PxConstraintSolveHint::eEQUALITY`.

- If one of the force limits is zero and the other unbo
`PxConstraintSolveHint::eINEQUALITY`.
- for all soft constraints, and hard constraints with impulse limits oth
set it to `PxConstraintSolveHint::eNONE`.

The solver does not check that the hint value is consistent with
`Px1DConstraint`. Using inconsistent values may result in undefined beha

The Projection Function

The other behavior that joints may specify for simulation is *projectio*
positional correction designed to act when the velocity-based solver fa
function has the following signature:

```
typedef void (*PxConstraintProject)(const void* constantBlock,  
                                   PxTransform& bodyAToWorld,  
                                   PxTransform& bodyBToWorld,  
                                   bool projectToA);
```

It receives the constant block and the two body transforms. It s
bodyBToWorld transform if the projectToA flag is set, and otherwise
transform. See the implementations in the extensions library for exampl
projection functions.

The Constraint Shader Table

After coding the behavior functions, define a structure of type `PxCons`
which holds the pointers to the constraint functions. This structure wil
argument to `PxPhysics::createConstraint`, and is shared by all instances

```
struct PxConstraintShaderTable  
{  
    PxConstraintSolverPrep    solverPrep;  
    PxConstraintProject       project;
```

```
PxConstraintVisualize        visualize;
};
```

The constraint visualizer allows the joint to generate visualization info through the `PxConstraintVisualizer` interface. The functionality of this interface is limited to the standard joints; examples of its use can be found in the extensions library.

Data Management

Next, define the class which lets PhysX manage the joint. This class implements the `PxConstraintConnector` interface.

To create a joint, call `PxPhysics::createConstraint`. The arguments to this function are the constrained actors, the connector object, the shader table, and the constant block. The return value is a pointer to `PxConstraint` object.

`PxConstraintConnector` has a number of data management callbacks:

```
virtual void*      prepareData();
virtual void       onConstraintRelease();
virtual void       onComShift(PxU32 actor);
virtual void       onOriginShift(const PxVec3& shift);
virtual void*      getExternalReference(PxU32& typeId);
```

These functions are usually boilerplate; sample implementations can be found in the extensions library:

- The `prepareData()` function requests a pointer to the joint constant block from PhysX. The joint uses this data to update any state caches etc. When the function returns, PhysX makes an internal copy of this data, so that the joint may be modified without race conditions. The function is called at the start of the simulation, when the joint is inserted into the scene, and on a subsequent simulation step when the joint is informed that the joint's state has changed. To inform PhysX that the state has changed, call `PxConstraint::markDirty()`.
- `onConstraintRelease()` is associated with joint deletion. To delete a joint, call `PxConstraint::release()`.

`PxConstraint::release()` on the constraint. When it is safe to destroy no internal references are being held by currently executing simulation code will call `PxConstraint::onConstraintRelease()`. This runs the destructor and releases the joint's memory etc.

- `onComShift()` is called when the application calls `setCMassLocalP` actors connected by the joint. This is provided because the solve projection functions are defined using the frame of the underlying joint configuration is typically defined in terms of the actors.
- `onOriginShift()` is called when the application shifts the origin of necessary because some joints may have a NULL actor, signifying attached to the world frame.
- `getExternalReference()` is used by PhysX to report simulation constraints, particularly breakage. The returned pointer is passed to the application in the event callback, along with the `typeID` which the application uses in order to cast the pointer to the appropriate type. The `typeID` should be unique for each custom joint type, and different from any of the values in `PxJointType`. If the joint also implements the `PxBase` interface, use the concrete `PxBase` for the `typeID`.

An articulation is a single actor comprising a set of links (each of which is a rigid body) connected together with special joints. Every articulation is a tree structure - so there can be no loops or breaks. Their primary use is in non-actuated characters. They support higher mass ratios, more accurate collision detection, better dynamic stability and a more robust recovery from joint separation than PhysX joints. However, they are considerably more expensive to simulate.

Although articulations do not directly build on joints, they use very similar underlying mechanisms. In this section we assume familiarity with PhysX joints.

Creating an Articulation

To create an articulation, first create the articulation actor without links:

```
PxArticulation* articulation = physics.createArticulation();
```

Then add links one by one, each time specifying a parent link (NULL for initial link), and the pose of the new link:

```
PxArticulationLink* link = articulation->createLink(parent, linkPose, linkP  
PxRigidActorExt::createExclusiveShape(*link, linkGeometry, materi  
PxRigidBodyExt::updateMassAndInertia(*link, 1.0f);
```

Articulation links have a restricted subset of the functionality of rigid bodies: they can be kinematic, and they do not support damping, velocity clamping, or contact thresholds. Sleep state and solver iteration counts are properties of the articulation rather than the individual links.

Each time a link is created beyond the first, a *PxArticulationJoint* is created between the link and its parent. Specify the joint frames for each joint, in exactly the same way as for a rigid body joint.

```
PxArticulationJoint* joint = link->getInboundJoint();  
joint->setParentPose(parentAttachment);  
joint->setChildPose(childAttachment);
```

Finally, add the articulation to the scene:

```
scene.addArticulation(articulation);
```

Articulation Joints

The only form of articulation joint currently supported is an anato properties are similar to D6 joint configured for a typical rag dc Specifically, the joint is a spherical joint, with angular drive, a twist limit joint frame's x-axis, and an elliptical swing cone limit around the parent. The configuration of these properties is very similar to a D6 or spher options provided are slightly different.

The swing limit is a hard elliptical cone limit which does not support s from movement perpendicular to the limit surface. You can set the limit follows:

```
joint->setSwingLimit(yAngle, zAngle);
```

for the limit angles around y and z. Unlike the PxJoint cone limit the tangential spring to limit movement of the axis along the limit surface enable the swing limit:

```
joint->setSwingLimitEnabled(true);
```

The twist limit allows configuration of upper and lower angles:

```
joint->setTwistLimit(lower, upper);
```

and again you must explicitly enable it:

```
joint->setTwistLimitEnabled(true);
```

As usual with joint limits, it is good practice to use a sufficient limit cor that the solver will start to enforce the limit before the limit threshold is e

Articulation joints are not breakable, and it is not possible to retrieve t applied at the joint.

Driving an Articulation

Articulations are driven through joint acceleration springs. You can target, an angular velocity target, and spring and damping parameters strongly the joint drives towards the target. You can also set compliance how strongly a joint resists acceleration. A compliance near zero indicates resistance, and a compliance of 1 indicates no resistance.

Articulations are driven in two phases. First the joint spring forces are a term *internal* forces for these) and then any *external* forces such as forces. You may supply different compliance values at each joint for each

Note that with joint acceleration springs, the required strength of the using just the mass of the two bodies connected by the joint. By contrast springs account for the masses of all the bodies in the articulation, and actuation at other joints. This estimation is an iterative process, call *externalDriveIterations* and *internalDriveIterations* properties of the PxArticulation

Instead of setting the target quaternion for the joint drive, it is possible to set orientation error term directly as a rotation vector. The value is set as the target quaternion, with the real part set to 0.

```
joint->setDriveType(PxArticulationJointDriveType::eERROR); joint->setTargetOrientation(PxQuat(error.x, error.y, error.z, 0));
```

This allows the spring to be driven with a larger positional error than could be obtained by the difference between 2 quaternions. Obtain the same behavior by computing the error from the target quaternion, link frames as follows:

```
PxTransform cA2w = parentPose.transform(joint.parentPose);
PxTransform cB2w = childPose.transform(joint.childPose);
transforms.cB2cA = transforms.cA2w.transformInv(transforms.cB2w);
if(transforms.cB2cA.q.w<0)
    transforms.cB2cA.q = -transforms.cB2cA.q;
```

```
// rotation vector from relative transform to drive pose  
PxVec3 error = log(j.targetPosition * cB2cA.q.getConjugate());
```

◀

Articulation Projection

When any of the joints in an articulation separate beyond a specified distance, the articulation is projected back together automatically. Projection is an iterative process. The `PxArticulation` functions `PxArticulation::setSeparationTolerance()` and `PxArticulation::setMaxProjectionIterations()` control when projection occurs and for how many iterations for robustness.

Articulations and Sleeping

Like rigid dynamic objects, articulations are also put into a sleep state below a certain threshold for a period of time. In general, all the [pc](#) *Sleeping* apply to articulations as well. The main difference is that articulation to sleep if each individual articulation link fulfills the sleep criteria.

[NVIDIA PhysX SDK 3.4.1 Documentation »](#)
[User's Guide »](#)

[previous](#) | [next](#)

The further away objects move from the origin, the larger the chance to point precision issues. This can cause troubles especially in scenario worlds. To avoid these problems, a straightforward solution seems to be to move objects towards the origin in certain intervals. However, this is not only cumbersome but also pretty expensive due to the invalidation of cached data and persistence. To solve some of these issues, PhysX offers an API to shift the origin of a scene.

Shifting The Scene Origin

The following method will shift the origin of a scene by a translation vec:

```
PxScene::shiftOrigin(const PxVec3& shift)
```

The positions of all objects in the scene and the corresponding data adjusted to reflect the new origin location (basically, the shift vector from all object positions). The intended use pattern for this API is to s that object positions move closer towards zero. Please note the responsibility to keep track of the summed total origin shift and adj to/from PhysX accordingly. Even though this method preserves some cached data, it is still an expensive operation and we recommend to use where distance related precision issues may arise in areas far from the modules of PhysX are used like the character controller or vehicle lib necessary to propagate the scene shift to those modules as well. Please see the documentation of these modules for details.

Introduction

GPU Rigid Bodies is a new feature introduced in PhysX 3.4. It supports the GPU rigid body pipeline feature-set but currently does not support articulations. Accelerated rigid bodies can be modified and queried using the exact same APIs as CPU rigid bodies. GPU rigid bodies can interact with particles in the same way that CPU rigid bodies can and can easily be used with character controllers (CCTs) and vehicles.

Using GPU Rigid Bodies

GPU rigid bodies are no more difficult to use than CPU rigid bodies. GPU rigid bodies use the exact same API and same classes as CPU rigid bodies. GPU rigid body simulation is enabled on a per-scene basis. If enabled, all rigid bodies occupying the scene will be processed by the GPU. This feature is implemented in CUDA and requires a NVIDIA or later compatible GPU. If no compatible device is found, simulation will fall back to CPU and corresponding error messages will be provided.

This feature is split into two components: rigid body dynamics and broadphase. Both are enabled using `PxSceneFlag::eENABLE_GPU_DYNAMICS` and `PxSceneDesc::broadPhaseType` to `PxBroadPhaseType::eGPU` respectively. These properties are immutable properties of the scene. In addition, you must create a `PxCudaContextManager` context manager and set the GPU dispatcher on the `PxSceneDesc`. A code example demonstrating how to enable GPU rigid body simulation is provided in the code example below. The code example below serves as a brief reference:

```
PxCudaContextManagerDesc cudaContextManagerDesc;

gCudaContextManager = PxCreateCudaContextManager(*gFoundation, cudaContextManagerDesc);

PxSceneDesc sceneDesc(gPhysics->getTolerancesScale());
sceneDesc.gravity = PxVec3(0.0f, -9.81f, 0.0f);
gDispatcher = PxDefaultCpuDispatcherCreate(4);
sceneDesc.cpuDispatcher = gDispatcher;
sceneDesc.filterShader = PxDefaultSimulationFilterShader;
sceneDesc.gpuDispatcher = gCudaContextManager->getGpuDispatcher();

sceneDesc.flags |= PxSceneFlag::eENABLE_GPU_DYNAMICS;
sceneDesc.broadPhaseType = PxBroadPhaseType::eGPU;

gScene = gPhysics->createScene(sceneDesc);
```

Enabling GPU rigid body dynamics turns on GPU-accelerated collision shape/body management and the GPU-accelerated constraint solver. The majority of the discrete rigid body pipeline.

Turning on GPU broad phase replaces the CPU broad phase with a GPU broad phase.

Each can be enabled independently so, for example, you may enable GPU broad phase with CPU rigid body dynamics, CPU broad phase (either SAP or MF) with GPU rigid body dynamics or combine GPU broad phase with GPU rigid body dynamics.

What is GPU accelerated?

The GPU rigid body feature provides GPU-accelerated implementations

- Broad Phase
- Contact generation
- Shape and body management
- Constraint solver

All other features are performed on the CPU.

There are several caveats to GPU contact generation. These are as follows

- GPU contact generation supports only boxes, convex hulls, triangles and heightfields. Any spheres, capsules or planes will have contact generation for those shapes processed on the CPU, rather than GPU.
- Convex hulls require `PxCookingParam::buildGRBDData` to be set to `GPU` to be required to perform contact generation on the GPU. If a hull with more than 32 vertices or more than 32 vertices per-face is used, it will be processed on the CPU. The `PxConvexFlag::eGPU_COMPATIBLE` flag is used when the convex hull limits are applied to ensure the resulting hull can be used on GPU.
- Triangle meshes require `PxCookingParam::buildGRBDData` to be set to `GPU` to be required to process the mesh on the GPU. If this flag is not set, the GPU data for the mesh will be absent and any contact generation for the mesh will be processed on CPU.
- Any pairs requesting contact modification will be processed on the CPU.
- `PxSceneFlag::eENABLE_PCM` must be enabled for GPU contact generation to be performed. This is the only form of contact generation implemented. If `eENABLE_PCM` is not raised, contact generation will be processed on the CPU using the non distance-based legacy contact generation.

Irrespective of whether contact generation for a given pair is processed, the GPU solver will process all pairs with contacts that request collision filter shader.

As mentioned above, GPU rigid bodies currently do not support `ENABLE_GPU_DYNAMICS`. If `ENABLE_GPU_DYNAMICS` is enabled on the scene, any attempts to add art to the scene will result in an error message being displayed and the art not added to the scene.

The GPU rigid body solver provides full support for joints and contact performance is achieved using D6 joints because D6 joints are native to GPU, i.e. the full solver pipeline from prep to solve is implemented on the GPU. Joint types are supported by the GPU solver but their joint shaders are currently run on the CPU, which will incur some additional host-side performance overhead compared to native GPU joints.

Tuning

Unlike CPU PhysX, the GPU rigid bodies feature is not able to dynamically grow buffers. Therefore, it is necessary to provide some fixed buffer sizes for the GPU rigid body feature. If insufficient memory is available, the system will discard contacts/constraints/pairs, which means that behavior may be unpredictable. The following buffers are adjustable in `PxSceneDesc::gpuDynamicsConfiguration`.

```
struct PxDynamicsMemoryConfig
{
    PxU32 constraintBufferCapacity; //!< Capacity of constraint buffer
    PxU32 contactBufferCapacity;    //!< Capacity of contact buffer
    PxU32 tempBufferCapacity;       //!< Capacity of temp buffer
    PxU32 contactStreamCapacity;    //!< Capacity of contact stream
    PxU32 patchStreamCapacity;      //!< Capacity of the contact patch stream
    PxU32 forceStreamCapacity;      //!< Capacity of force buffer
    PxU32 heapCapacity;             //!< Initial capacity of heap
    PxU32 foundLostPairsCapacity;   //!< Capacity of found and lost pairs

    PxDynamicsMemoryConfig() :
        constraintBufferCapacity(32 * 1024 * 1024),
        contactBufferCapacity(24 * 1024 * 1024),
        tempBufferCapacity(16 * 1024 * 1024),
        contactStreamCapacity(6 * 1024 * 1024),
        patchStreamCapacity(5 * 1024 * 1024),
        forceStreamCapacity(1 * 1024 * 1024),
        heapCapacity(64 * 1024 * 1024),
        foundLostPairsCapacity(256 * 1024)
    {}
};
```

The default values are generally sufficient for scenes simulating applications with rigid bodies.

- `constraintBufferCapacity` defines the total amount of memory that can be used for constraints in the solver. If more memory is required, a warning will be issued and further constraints will be created.

- `contactBufferCapacity` defines the size of a temporary contact constraint solver. If more memory is required, a warning is issued and contacts are dropped.
- `tempBufferCapacity` defines the size of a buffer used for miscellaneous memory allocations used in the constraint solver.
- `contactStreamCapacity` defines the size of a buffer used to store contact stream. This data is allocated in pinned host memory and GPU. If insufficient memory is allocated, a warning will be issued and contacts are dropped.
- `patchStreamCapacity` defines the size of a buffer used to store correction contact stream. This data is allocated in pinned host memory and GPU. If insufficient memory is allocated, a warning will be issued and contacts are dropped.
- `forceStreamCapacity` defines the size of a buffer used to report applied forces to the user. This data is allocated in pinned host memory. If insufficient memory is allocated, a warning will be issued and contacts will be dropped.
- `heapCapacity` defines the initial size of the GPU and pinned host memory heap. Additional memory will be allocated if more memory is required. The amount of allocating memory can be relatively high so a custom heap allocator is used to reduce these costs.
- `foundLostPairsCapacity` defines the maximum number of found or lost pairs that GPU broad phase can produce in a single frame. This does not limit the total number of pairs but only limits the number of new or lost pairs that can be detected in a frame. If more pairs are detected or lost in a frame, an error is issued and contacts are dropped by the broad phase.

Performance Considerations

GPU rigid bodies can provide extremely large performance advantages in scenes with several thousand active rigid bodies. However, there are several performance considerations to be taken into account.

GPU rigid bodies currently only accelerate contact generation involving boxes (against convex hulls, boxes, triangle meshes and heightfields). The use of other shapes, e.g. capsules or spheres, contact generation involving them will only be processed on CPU.

D6 joints will provide best performance when used with GPU rigid bodies. D6 joints will be partially GPU-accelerated but the performance advantages will be less than the performance advantage exhibited by D6 joints.

Convex hulls with more than 64 vertices or with more than 32 vertices on their contacts processed by the CPU rather than the GPU, so, if possible, keep counts within these limits. Vertex limits can be defined in cooking to ensure that convex hulls do not exceed these limits.

If your application makes heavy use of contact modification, this may impact performance. Pairs that have contact generation performed on the GPU.

Modifying the state of actors forces data to be re-synced to the GPU, so actors must be updated if the application adjusts global pose, velocities, or if the application modifies the bodies' velocities etc.. The associated cost to the GPU is relatively low but it should be taken into consideration.

Features such as joint projection, CCD and triggers are not GPU accelerated and are processed on the CPU.

Introduction

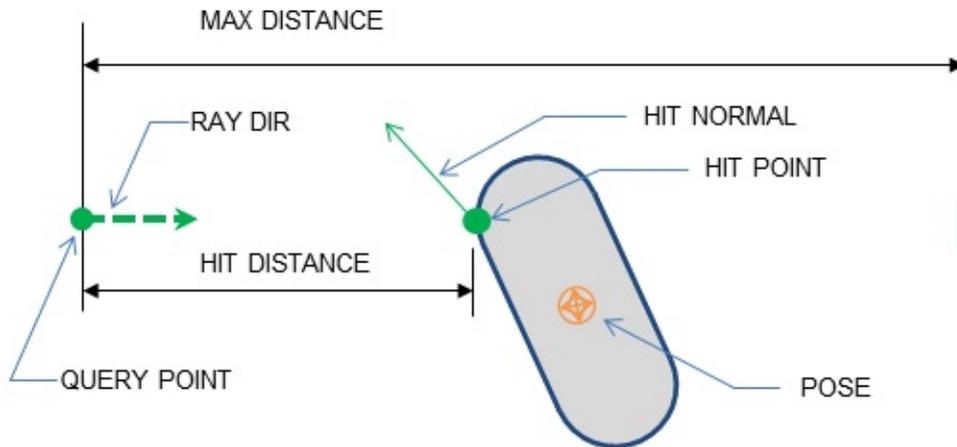
This chapter describes how to use PhysX' collision functionality with i objects. There are four main kinds of geometry queries:

- raycasts ("raycast queries") test a ray against a geometry object.
- sweeps ("sweep queries") move one geometry object along a line t of intersection with another geometry object.
- overlaps ("overlap queries") determine whether two geometry objec
- penetration depth computations ("minimal translational distance qu here to "MTD") test two overlapping geometry objects to find the dii they can be separated by the minimum distance.

In addition, PhysX provides helpers to compute the AABB of a geom compute the distance between a point and a geometry object.

In all of the following functions, a geometry object is defined by its sha structure) and its pose (a *PxTransform* structure). All transforms interpreted as being in the same space, and the results are also returne

Raycasts



A raycast query traces a point along a line segment until it hits a geom supports raycasts for all geometry types.

The following code illustrates how to use a raycast query:

```
PxRaycastHit hitInfo;
PxU32 maxHits = 1;
PxHitFlags hitFlags = PxHitFlag::ePOSITION|PxHitFlag::eNORMAL|PxH
PxU32 hitCount = PxGeometryQuery::raycast(origin, unitDir,
                                           geom, pose,
                                           maxDist,
                                           hitFlags,
                                           maxHits, &hitInfo);
```

The arguments are interpreted as follows:

- *origin* is the start point of the ray.
- *unitDir* is a unit vector defining the direction of the ray.
- *maxDist* is the maximum distance to search along the ray. It mu: range. If the maximum distance is 0, a hit will only be returned if the shape, as detailed below for each geometry.

- *geom* is the geometry to test against.
- *pose* is the pose of the geometry.
- *hitFlags* specifies the values that should be returned by the query processing the query.
- *maxHits* is the maximum number of hits to return.
- *hitInfo* specifies the *PxRaycastHit* structure(s) into which the ray is stored.
- The *anyHit* parameter is **deprecated**. It is equivalent to *PxHitFlags* which should be used instead.

The returned result is the number of intersections found. For each *PxRaycastHit* is populated. The fields of this structure are as follows:

```
PxRigidActor*   actor;
PxShape*        shape;
PxVec3          position;
PxVec3          normal;
PxF32           distance;
PxHitFlags      flags;
PxU32           faceIndex;
PxF32           u, v;
```

Some fields are optional, and the flags field indicates which members have result values. The query will fill fields in the output structure if the corresponding flag is set in the input - for example, if the *PxHitFlag::ePOSITION* is set in the input, the query will fill in the *PxRaycastHit::position* field, and set the *PxHitFlag::ePOSITION* in the *PxRaycastHit::flags*. If the input flag is not set for a specific member, that member may or may not contain valid data for that member. Omitting the *ePOSITION* flags in the input can sometimes result in faster queries.

For a raycast which is not initially intersecting the geometry object, the fields as follows (optional fields are listed together with the flag that controls them):

- *actor* and *shape* are not filled (these fields are used only in scene-[Scene Queries](#)).

- *position* (*PxHitFlag::ePOSITION*) is the position of the intersection.
- *normal* (*PxHitFlag::eNORMAL*) is the surface normal at the point of intersection.
- *distance* (*PxHitFlag::eDISTANCE*) is the distance along the ray from the origin to the intersection point.
- *flags* specifies which fields of the structure are valid.
- *faceIndex* is the index of the face which the ray hit. For triangle mesh intersections, it is a triangle index. For convex mesh intersections, it is a convex mesh index. For other shapes it is always set to 0xffffffff.
- *u* and *v* (*PxHitFlag::eUV*) are the barycentric coordinates of the intersection point. These fields (and the flag) are supported only for meshes and heightfields.

The position field is related to the barycentric coordinates via the following formula. *v0*, *v1* and *v2* are the vertices from the hit triangle:

$$\text{position} = (1 - u - v) * v0 + u * v1 + v * v2;$$

This mapping is implemented in *PxTriangle::pointFromUV()*.

See [Geometry](#) for details of how to retrieve face and vertex data from convex meshes and height fields using face and vertex indices.

Exceptions to the above behavior may apply if a ray starts inside an object. PhysX may not be able to compute meaningful output values for some cases the field will remain unmodified and the corresponding flag will be 0. Details vary by geometry type, and are described below.

The exact conditions for raycast intersections are as follows:

Raycasts against Spheres, Capsules, Boxes and Convex Meshes

For solid objects (sphere, capsule, box, convex) at most 1 result is returned. If the ray origin is inside a solid object:

- the reported hit distance is set to zero, and the *PxHitFlag::eDISTANCE* flag is set.

the output.

- the hit normal is set to be the opposite of the ray's c
PxHitFlag::eNORMAL flag is set in the output.
- the hit impact position is set to the ray's origin and the *PxHitFlag::*
set in the output.

If the start or end point of a ray is very close to the surface of the object as being on either side of the surface.

Raycasts against Planes

For raycasts, a plane is treated as an infinite single-sided quad that intersects (note that this is not the same as for overlaps). At most one result is reported if the ray origin is behind the plane's surface, no hit will be reported even if the ray intersects the plane.

If the start or end point of a ray is very close to the plane, it may be reported on either side of the plane.

Raycasts against Triangle Meshes

Triangle meshes are treated as thin triangle surfaces rather than solid volumes. They can be configured to return either an arbitrary hit, the closest hit, or multiple hits.

- if *maxHits* is 1 and *PxHitFlag::eMESH_ANY* is not set, the query will return the first intersection.
- if *maxHits* is 1 and *PxHitFlag::eMESH_ANY* is set, the query will return the first intersection. Use this when it is sufficient to know whether or not there is an intersection. e.g. for line-of-sight queries or shadow rays.
- if *maxHits* is greater than 1, the query will return multiple intersections. If more than *maxHits* intersection points exist, there is no guarantee that the results will include the closest. Use this for e.g. wall-piercing bullets that hit multiple surfaces or where special filtering is required. Note that *PxHitFlag::eMESH*

be used in this case.

In general "any hit" queries are faster than "closest hit" queries, and "any hit" queries are faster than "multiple hits" queries.

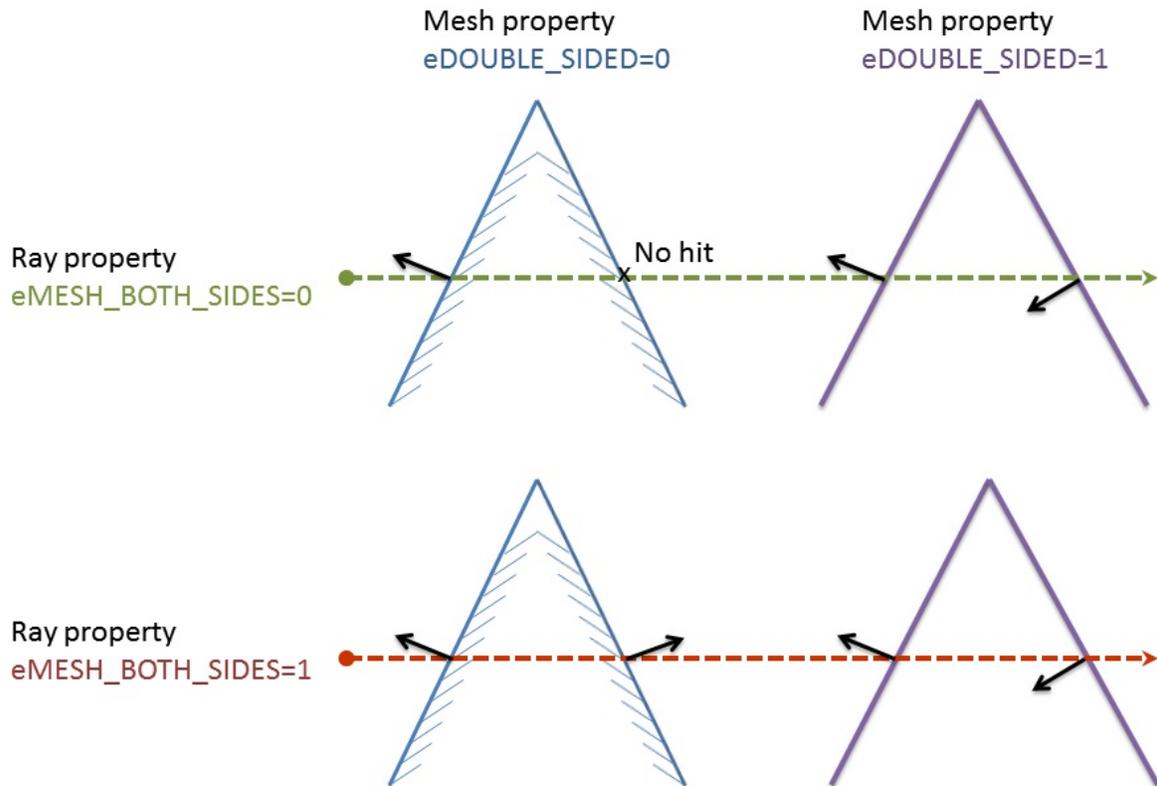
By default, back face hits (where the triangle's outward-facing normal product with the ray direction) are culled, and so for any triangle hit the reported normal will have a negative dot product with the ray direction. This behavior can be disabled on the mesh instance's *PxMeshGeometryFlag::eDOUBLE_SIDED* flag or the *PxHitFlag::eMESH_BOTH_SIDES* flag:

- if either *PxMeshGeometryFlag::eDOUBLE_SIDED* or *PxHitFlag::eMESH_BOTH_SIDES* is set, culling is disabled.
- if *PxMeshGeometryFlag::eDOUBLE_SIDED* is set, the reported normal is reversed for a back face hit.

For example a transparent glass window could be modeled as a double-sided mesh so that a ray would hit either side with the reported normal facing the ray direction. A raycast tracing the path of a bullet that may penetrate the front and emerge from the back could use *eMESH_BOTH_SIDES* to find both front and back facing triangles even when the mesh is single-sided.

The following diagram shows what happens with different flags, for a ray intersecting a mesh in several places.

Returned hit normals (with eMESH_MULTIPLE set)



To use `PxHitFlag::eMESH_BOTH_SIDES` for selected meshes rather than for all meshes, set the `eMESH_BOTH_SIDES` flag inside the `PxQueryFilterCallback`.

If the start or end point of a ray is very close to the surface of a triangle, the ray is treated as being on either side of the triangle.

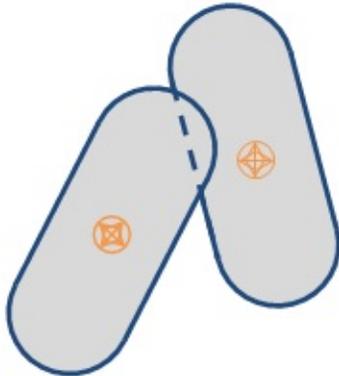
If the start or end point of a ray is very close to the surface of a triangle, the ray is treated as being on the either side of the triangle.

Raycasts against Heightfields

- Heightfields are treated the same way as triangle meshes with `n` (shape space) in `+y` direction when thickness is `<=0` and in `-y` direction when thickness is `>0`.
- Double-sided heightfields are treated the same way as double sided triangles.



Overlaps



Overlap queries simply check whether two geometry objects overlap. The supported geometries must be a box, sphere, capsule or convex, and the other must be a plane.

The following code illustrates how to use an overlap query:

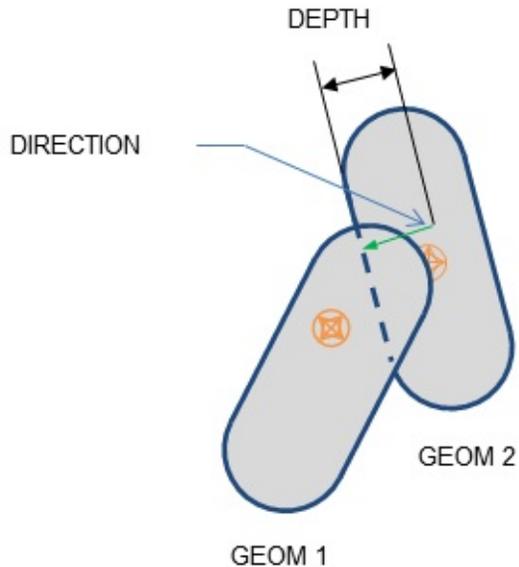
```
bool isOverlapping = overlap(geom0, pose0, geom1, pose1);
```

Overlaps do not support hit flags and return only a boolean result.

- A plane is treated as a solid half-space: that is, everything below the plane is considered part of the volume.
- Triangle meshes are treated as thin triangle surfaces rather than solid volumes.
- Heightfields are treated as triangle surface extruded by their height. Geometries that do not intersect with the heightfield surface but are within the extruded space will report a hit.

If more than a boolean result is needed for meshes and heightfields, use the [PxMeshQuery](#) API instead (see [PxMeshQuery](#)).

Penetration Depth



When two objects are intersecting, PhysX can compute the minimal distance by which the objects must be translated to separate them (this quantity is referred to as MTD, for *minimum translational distance*, as it is the vector by which translation will separate the shapes). One geometry object may be a sphere, capsule or convex mesh, and the other may be of any type.

The following code illustrates how to use a penetration depth query:

```
bool isPenetrating = PxGeometryQuery::computePenetration(direction, p0, p1, geom0, p0, geom1, p1)
```

The arguments are interpreted as follows:

- *direction* is set to the direction in which the first object should be translated to separate from the second.
- *distance* is set to the distance by which the first object should be translated to separate from the second.

- *geom0* is the first geometry.
- *pose0* is the transform of the first geometry.
- *geom1* is the second geometry.
- *pose2* is the transform of the second geometry.

The function returns true if the objects are penetrating, in which case *direction* and *depth* fields. Translating the first object by the *depenetration* vector and *depth* will separate the two objects. If the function returns true, the values are always be positive or zero. If objects do not overlap, the function returns false and the values of the *direction* and *distance* fields are undefined.

For simple (convex) shapes, returned results are accurate.

For meshes and heightfields, an iterative algorithm is used and dedicated functions are exposed in *PxExtensions*:

```
PxVec3 direction = PxComputeMeshPenetration(direction, depth,
                                           geom, geomPose,
                                           meshGeom, meshPose,
                                           maxIter, nb);

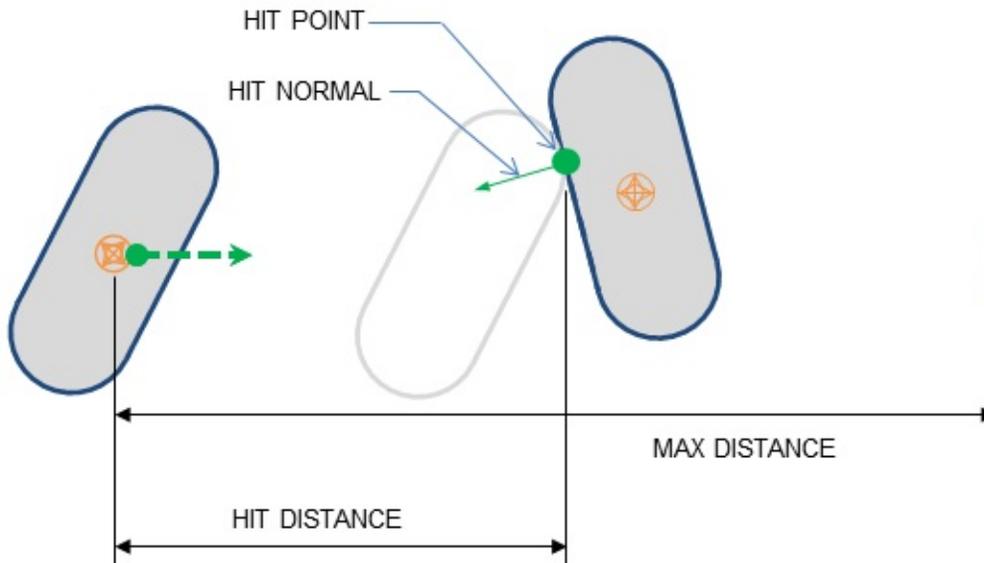
PxVec3 direction = PxComputeHeightFieldPenetration(direction, depth,
                                                    geom, geomPose,
                                                    heightFieldGeom,
                                                    maxIter, nb);
```

Here, *maxIter* is the maximum number of iterations for the algorithm, and *nb* is an output argument which will be set to the number of iterations performed. If no penetration is detected, *nb* is set to zero. The code will attempt at most *maxIter* iterations, but will stop earlier if a *depenetration* vector is found. Usually *maxIter* = 4 gives good results.

These functions only compute an approximate *depenetration* vector, and do not compute the amount of overlap between the geometry object and the mesh/heightfield. In particular, an intersection with a triangle will be ignored when the object is not inside the triangle, and if this holds for all intersecting triangles then no overlap is detected. The functions do not compute an MTD vector.



Sweeps



A sweep query traces one geometry object through space to find the second geometry object, and reports information concerning the imp found. PhysX only supports sweep queries where the first geometry object traced through space) is a sphere, box, capsule or convex geometry object may be of any type.

The following code illustrates how to use a sweep query:

```
PxSweepHit hitInfo;  
PxHitFlags hitFlags = PxHitFlag::ePOSITION|PxHitFlag::eNORMAL|PxH  
PxReal inflation = 0.0f;  
PxU32 hitCount = PxGeometryQuery::sweep(unitDir, maxDist,  
                                          geomToSweep, poseToSweep,  
                                          geomSweptAgainst, poseSwe  
                                          hitInfo,  
                                          hitFlags,  
                                          inflation);
```

The arguments are interpreted as follows:

- *unitDir* is a unit vector defining the direction of the sweep.
- *maxDist* is the maximum distance to search along the sweep. It must be in the range [0, 1], and is clamped by SDK code to at most `PX_MAX_SWEEP_LENGTH`. A sweep of length 0 is equivalent to an overlap check.
- *geomToSweep* is the geometry to sweep. Supported geometries are capsule or convex mesh.
- *poseToSweep* is the initial pose of the geometry to sweep.
- *geomSweptAgainst* is the geometry to sweep against (any geometry used here).
- *poseSweptAgainst* is the pose of the geometry to sweep against.
- *hitInfo* is the returned result. A sweep will return at most one hit.
- *hitFlags* determines how the sweep is processed, and which data fields in the output structure are filled. An impact is found if `PX_HIT_FLAG_IMPACT` is set.
- *inflation* inflates the first geometry with a shell extending outward from its surface, making any corners rounded. It can be used to ensure a minimum clearance space is kept around the geometry when using sweeps to test for intersections.

As with raycasts, fields will be filled in the output structure if the corresponding bit is set in the input *hitFlags*. The fields of `PxSweepHit` are as follows:

```
PxRigidActor*  actor;
PxShape*      shape;
PxVec3        position;
PxVec3        normal;
PxF32        distance;
PxHitFlags    flags;
PxU32        faceIndex;
```

- *actor* and *shape* are not filled (these fields are used only in scene queries).
- *position* (`PxHitFlag::ePOSITION`) is the position of the intersection point. For multiple impact points, such as two boxes meeting face-to-face, `PxHitFlag::ePOSITION` is the position of the first impact point.

point arbitrarily. More detailed information for meshes or height field using the functions in *PxMeshQuery*.

- *normal* (*PxHitFlag::eNORMAL*) is the surface normal at the point of vector, pointing outwards from the hit object and backwards along the sweep direction (in the sense that the dot product between the sweep direction and the normal is negative).
- *distance* (*PxHitFlag::eDISTANCE*) is the distance along the sweep direction from the start of the sweep to the intersection was found.
- *flags* specifies which fields of the structure are valid.
- *faceIndex* is the index of the face hit by the sweep. This is a face index not from the swept object. For triangle mesh and height field it is a triangle index. For convex mesh intersections it is a polygon index. For raycasts it is always set to 0xffffffff. For convex meshes the face index computation can be disabled by not providing the face index query hit flag *PxHitFlag::eFACE_INDEX*. If needed the face index can be computed externally using the function *PxFindFaceIndex* which is in the *PxExtensions* library.

Unlike raycasts, u,v coordinates are not supported for sweeps.

For the geometry object swept against:

- A plane is treated as a solid half-space: that is, everything behind the plane is considered part of the volume to sweep against.
- The same backface-culling rules as for raycasts apply for sweeps. The difference is that *eMESH_MULTIPLE* is not supported.

Initial Overlaps

Similarly to a raycast starting inside an object, a sweep may start with an object initially intersecting. By default PhysX will detect and report *PxSweepHit::hadInitialOverlap()* to see if the hit was generated by an initial overlap.

For triangle meshes and height fields, backface culling is performed, and thus no initial overlap is reported if a triangle is culled.

Depending on the value of *PxHitFlag::eMTD*, PhysX may also calculate *PxHitFlag::eMTD* is not set:

- the distance is set to zero, and the *PxHitFlag::eDISTANCE* flag is set in the *PxSweepHit* result structure.
- the normal is set to be the opposite of the sweep direction, and the *PxHitFlag::eNORMAL* flag is set in the *PxSweepHit* result structure
- the position is undefined, and the *PxHitFlag::ePOSITION* flag is not set in the *PxSweepHit* result structure.
- the *faceIndex* is a face from the second geometry object. For a triangle mesh, it is the index of the first overlapping triangle found. For other geometry types, the index is set to 0xffffffff.

If *PxHitFlag::eMTD* is set, the hit results are defined as follows:

- the distance is set to the penetration depth, and the *PxHitFlag::eDISTANCE* flag is set in the *PxSweepHit* result structure.
- the normal is set to the depenetration direction, and the *PxHitFlag::eNORMAL* flag is set in the *PxSweepHit* result structure.
- the position is a point on the sweep geometry object (i.e. the first geometry object) and the *PxHitFlag::ePOSITION* flag is set in the *PxSweepHit* result structure.
- the *faceIndex* is a face from the second geometry object:
 - For triangle meshes and heightfields it is the last penetrated triangle from the last iteration of the depenetration algorithm.
 - For other geometry types, the index is set to 0xffffffff.

This flag will incur additional processing overhead in the case of a hit. In addition, the following restrictions apply:

- `PxHitFlag::eMTD` is incompatible with `PxHitFlag::ePRECISE` and `PxHitFlag::eASSUME_NO_INITIAL_OVERLAP` (see below). Using `PxHitFlag::eMTD` in conjunction with either of these flags will result in a warning being issued that the flags are incompatible with `PxHitFlag::eMTD` being ignored.

Testing for initial overlaps sometimes uses a specialized code path to avoid a performance penalty. If it is possible to guarantee that geometry objects are not overlapping, the check for overlaps can be skipped by using the `PxHitFlag::eASSUME_NO_INITIAL_OVERLAP`. There are some restrictions when using this flag (also, see [Pitfalls](#))

- Using `PxHitFlag::eASSUME_NO_INITIAL_OVERLAP` flag when objects initially overlap produces undefined behavior.
- `PxHitFlag::eASSUME_NO_INITIAL_OVERLAP` in combination with `PxHitFlag::eMTD` and a non-zero distance produces a warning and undefined behavior.

Note: Sweeps with `PxHitFlag::eMTD` use two kinds of backface culling. First, the triangles are culled based on sweep direction to determine which side they overlap. If an overlap is detected, they are further culled by whether they are behind the triangle, and if no triangles are found, the direction will be swapped to the sweep direction and the distance to 0.

Note: In most cases, translating the first geometry object by `-normal * distance` will separate the objects. However, an iterative depenetration algorithm is used for `PxHitFlag::eMTD` for triangle meshes and height fields, and the MTD result may not guarantee complete depenetration from the mesh in extreme cases. In this case, the sweep will be called a second time after the translation has been applied.

Note: A known issue in PhysX 3.3 is that the face index for a sweep against a mesh is undefined when the `eMTD` flag is not set.

Precise Sweeps

PxHitFlag::ePRECISE_SWEEP enables more accurate sweep (a potentially faster but less accurate solution is used). The *ePRECISE_* is compatible with the inflation parameter, or with the flag *PxHitFlag::eMTL*

Sweeps against Height Fields

- Height fields are treated as thin triangle surfaces rather than solid c
- Thickness magnitude has no effect on initial overlap detection or p
- For single-sided height fields the normal of the hit will face in +y lo if thickness is < 0 and -y when thickness is > 0 .
- Height fields are treated as double sided if either one of *eD* or *eMESH_BOTH_SIDES* flags are used.
 - The returned hit normal will always face the sweep direction
- *eMESH_ANY* flag has no effect.
- *ePRECISE_SWEEP* flag has no effect.

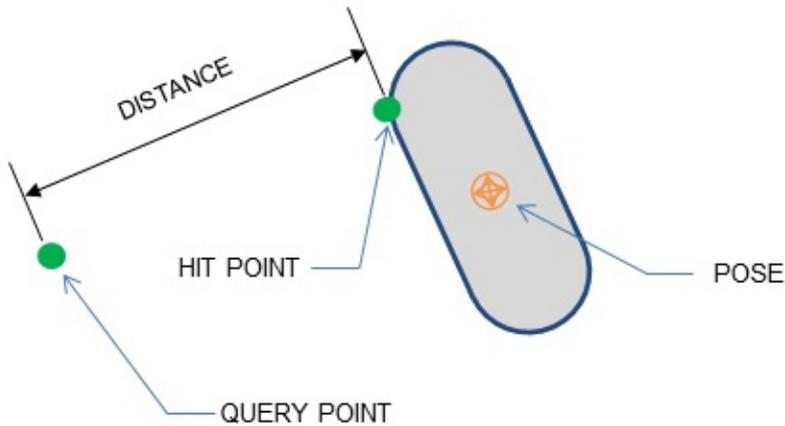
Pitfalls

There are some pitfalls to be aware of when using sweeps:

- Due to numerical precision issues, incorrect results may be returned and have very large size disparities.
- Due to algorithmic differences, a sweep query may detect a different set of overlapping shapes than an overlap query. In particular, it is not sufficient to perform an overlap check in order to determine the hit. Applications that require overlap/sweep/penetration depth information should use sweep

overlap testing and the *PxHitFlag::eMTD* flag.

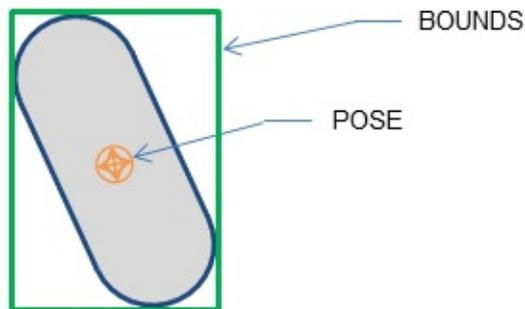
Additional PxGeometryQuery functions



The following function computes the distance between a point and a geometry object. Only solid objects (box, sphere, capsule, convex) are supported:

```
PxReal dist = PxGeometryQuery::pointDistance(point, geom, pose, closestPoint);
```

closestPoint is an optional output argument which returns the closest point on the geometry object.



The following function computes the axis-aligned bounding box (AABB) of a geometry object, given its pose:

```
PxBounds3 bounds = PxGeometryQuery::getWorldBounds(geom, pose, in
```

The bounding box is scaled by the *inflation* value, which defaults to 1 specified.

PxMeshQuery

PhysX provides additional functionality for obtaining multiple results for height field overlaps, and for sweeping against arrays of triangles. Or and capsules may be tested against meshes or heightfields using these

Mesh Overlaps

The following code illustrates how to process the mesh triangles spherical volume:

```
PxU32 triangleIndexBuffer[bufferSize];
PxU32 startIndex = 0;
bool bufferOverflowOccured = false;
PxU32 nbTriangles = PxMeshQuery::findOverlapTriangleMesh(sphereGeom
                                                         meshGeom
                                                         triangle
                                                         startInd

for(PxU32 i=0; i < nbTriangles; i++)
{
    PxTriangle tri;
    PxU32 vertexIndices[3];
    PxMeshQuery::getTriangle(meshGeom, meshPose, triangleIndexBuf

    ... // process triangle info
}
```

The *findOverlapTriangleMesh* method is used to extract the indices of th

- *sphereGeom* and *spherePose* specify the region to test for overlap
- *meshGeom* and *meshPose* specify the mesh and its pose.
- *triangleIndexBuffer* and *triangleSize* specify the output buffer and it
- *startIndex* is used to restart the query if the buffer size is exceed query for more triangles set this parameter to the number retrieved
- *bufferOverflowOccured* is set if more triangles would be returned f

would fit in the buffer.

Similar query functionality exists for height fields.

Sweeps against Triangles

Sometimes, for example, when using the mesh overlap API, it is convenient to sweep against groups of triangles. PhysX provides a function specifically for this with the following signature:

```
bool sweep(const PVec3& unitDir,
           const PReal distance,
           const PGeometry& geom,
           const PTransform& pose,
           PxU32 triangleCount,
           const PxTriangle* triangles,
           PSweepHit& sweepHit,
           PHitFlags hitFlags = PHitFlag::eDEFAULT,
           const PxU32* cachedIndex = NULL,
           const PReal inflation = 0.0f,
           bool doubleSided = false);
```

The arguments are interpreted as follows:

- *unitDir*, *distance*, *geom* and *pose* function identically to the first PxGeometryQuery::sweep(). *distance* is clamped to PX_MAX_SWI
- *triangleCount* is the number of triangles contained in the buffer sweep.
- *triangles* is the buffer of triangles.
- *hitFlags* specifies the required information in the output.
- *cachedIndex*, if set, specifies the index of a triangle to test first. This is an optimization when repeatedly sweeping against the same set of triangles.
- *inflation* functions identically to the inflation parameter of PxGeometryQuery::sweep().
- *doubleSided* indicates whether the input triangles are double-sided. If true, it is equivalent to the PMeshGeometryFlag::eDOUBLE_SIDED flag. If false, it suppresses backface culling, and for any hit the returned normal faces the direction of the sweep.

sweep direction (see *Raycasts against Triangle Meshes*).

This function has extra limitations compared to the other sweep queries

- the geometry type must be either a sphere, a capsule or a box. C not supported.
- the function returns a single hit. Multiple hits (and PxBroadPhaseHitFlag::eMESH_MULTIPLE) are not supported.
- The function always returns the closest hit.
- The only supported flags are PxBroadPhaseHitFlag::eASSUME_NO_INITIAL_OVERLAP, PxBroadPhaseHitFlag::eMESH_BOTH_SIDES and PxBroadPhaseHitFlag::eMESH_ANY.

The function tests each input triangle in the order they are given. By default, it will test all triangles and return the closest sweep hit (if a hit has PxBroadPhaseHitFlag::eMESH_ANY is used, the function will return as soon as a hit is found among the remaining untested triangles). This flag can also be used to return PxBroadPhaseHitFlag::eMESH_MULTIPLE, by calling the function PxBroadPhaseHitFlag::eMESH_ANY, using as a starting point the previously returned hit (whose index, between 0 and 'triangleCount', is available in sweepHit.flags).

Applications commonly need to efficiently query volumes in space or traverse objects through space to determine what might be there. PhysX supports this, one for objects already in a scene, and one for querying against AABBs. The scene query system is discussed in [Scene Queries](#).

PxSpatialIndex

PxSpatialIndex is a BVH data structure that allows spatial queries to be the need to instantiate a PxScene. It supports insertion, removal of objects defining a bounding box, and raycasts, sweeps, and overlap queries.

Spatial index has been marked as deprecated in 3.4 and will be removed in a future release.

SnippetSpatialIndex shows an example of how to use this class.

PxSpatialIndex has no internal locking, and there are special considerations from multiple threads. Query operations (marked const in the interface) can be performed in parallel with update (non-const) operations, or update operations in other threads. When issuing query operations in parallel, it is important that PxSpatialIndex defers some updates to its internal data structures until the end of the query. In a single-threaded context this does not affect correctness or safety, but from multiple threads simultaneously the internal updates may cause race conditions. In order to avoid these, call the flush() method to force the updates to be flushed immediately. Between a call to flushUpdates() and any subsequent queries may be safely issued in parallel.

A query against a PxSpatialIndex structure will result in a callback for the query, allowing filtering or precise intersection as desired. The PxGeometryQuery class can be used to perform these intersection queries. The results typically be in approximately sorted order, and when looking for the nearest hit for a raycast or sweep query against PxSpatialIndex, a useful optimization is to perform the query inside the callback. For example, in SnippetSpatialIndex:

```
PxAgain onHit(PxSpatialIndexItem& item, PxReal distance, PxReal& hitData)
{
    PX_UNUSED(distance);

    Sphere& s = static_cast<Sphere&>(item);
    PxRaycastHit hitData;
```

```

// the ray hit the sphere's AABB, now we do a ray-sphere inte
// the ray hit the sphere

PxU32 hit = PxGeometryQuery::raycast(position, direction,
                                     PxSphereGeometry(s.radius
                                     1e6, PxHitFlag::eDEFAULT
                                     1, &hitData);

// if the raycast hit and it's closer than what we had before
// of the raycast

if(hit && hitData.distance < closest)
{
    closest = hitData.distance;
    hitSphere = &s;
    shrunkDistance = hitData.distance;
}

// and continue the query

return true;
}

```

Note: Methods in `PxGeometryQuery` may report positive results when within a numerical tolerance of intersection or impact. To obtain results when using `PxSpatialIndex` and when not using a culling hierarchy, the must be slightly padded. `PxGeometryQuery::getWorldBounds` adds this default.

`PxSpatialIndex` has the same performance characteristics as the scene using the `PxPruningStructureType::eDYNAMIC_AABB_TREE` option correspond to moving objects, or there are many insertions and deletions the tree may degrade over time. In order to prevent this, the tree may be rebuilt using the function `rebuildFull()`. Alternatively, a second tree may be built in the background over many small steps, using the function `rebuildStep()` using the same incremental rebuild step as performed by the scene's dynamic pruning during `fetchResults()`. See [PxPruningStructureType](#) for details.

Introduction

PhysX provides methods in `PxScene` to perform collision queries on attached shapes in the scene. There are three types of queries: ray overlaps, and each can return either a single result, or multiple results. Each query traverses a culling structure containing the scene objects, test using the `GeometryQuery` functions (see [Geometry Queries](#)), and returns results. Filtering may occur before or after precise testing.

The scene uses two different query structures, one for `PxRigidStatic` actors and one for `PxRigidBody` actors (`PxRigidDynamic` and `PxArticulationLink`.) They can be configured to use different culling implementations depending on their speed/space characteristics (see [PxPruningStructureType](#).)

Basic queries

Raycasts

A `PxScene::raycast()` query intersects a user-defined ray with the simplest use case for a `raycast()` query is to find the closest hit also follows:

```
PxScene* scene;
PxVec3 origin = ...;           // [in] Ray origin
PxVec3 unitDir = ...;         // [in] Normalized ray direction
PxReal maxDistance = ...;    // [in] Raycast max distance
PxRaycastHit hit;             // [out] Raycast results

// Raycast against all static & dynamic objects (no filtering)
// The main result from this call is the closest hit, stored in t
bool status = scene->raycast(origin, unitDir, maxDistance, hit);
if (status)
    applyDamage(hit.block.position, hit.block.normal);
```

In this code snippet a `PxRaycastHit` object is used to receive result query. A call to `raycast()` returns true if there was a hit. `hit.hadBlock` is there was a hit. The distance for raycasts has to be in the $[0, \infty)$ range.

Raycasts results include position, normal, hit distance, shape and actor with UV coordinates for triangle meshes and heightfields. Before using `PxHitFlag::ePOSITION`, `eNORMAL`, `eDISTANCE`, `eUV` flags first, as they are not set.

Sweeps

A `PxScene::sweep()` query is geometrically similar to a `raycast()`: a `PxSweep` swept from a specified initial pose in a direction `unitDir` with specified `radius` find the points of impacts of the geometry with scene objects. The maximum sweeps has to be in the $[0, \infty)$ range, and will be

PX_MAX_SWEEP_DISTANCE, defined in file PxScene.h.

Allowed shapes are box, sphere, capsule and convex.

A PxSweepBuffer object is used to receive results from sweep() queries

```
PxSweepBuffer hit; // [out] Sweep results
PxGeometry sweepShape = ...; // [in] swept shape
PxTransform initialPose = ...; // [in] initial shape pose (at di
PxVec3 sweepDirection = ...; // [in] normalized sweep directio
bool status = scene->sweep(sweepShape, initialPose, sweepDirectio
```

Sweeps results include position, normal, hit distance, shape and actor for triangle meshes and heightfields.

Overlaps

PxScene::overlap() query searches a region enclosed by a specific overlapping objects in the scene. The region is specified as a transform capsule or convex geometry.

A PxOverlapBuffer object is used to receive results from overlap() queries

```
PxOverlapBuffer hit; // [out] Overlap results
PxGeometry overlapShape = ...; // [in] shape to test for overlap
PxTransform shapePose = ...; // [in] initial shape pose (at di

PxOverlapBuffer hit;
bool status = scene->overlap(overlapShape, shapePose, hit);
```

Overlaps results only include actor/shape and faceIndex since there is intersection.

Touching and blocking hits

For queries with multiple results we distinguish between *touching* and choice of whether a hit is touching or blocking is made by the user-ir logic. Intuitively a blocking hit prevents further progress of a raycast or path, and a touching hit is recorded but allows the ray or sweep to continue. A hit query will return the closest blocking hit if one exists, together with any touching hits that are closer. If there are no blocking hits, all touching hits will be returned.

See the [Filtering](#) section for details.

Query modes

Closest hit

The default mode of operation for all three query types is "closest hit". For all blocking hits, picks the one with the minimum distance and PxBoundingBox::block member.

- For overlap() queries an arbitrary blocking hit is chosen as the result (distance is treated as zero for all overlap() hits).

Any hit

All three query types can operate in "any hit" mode. This is a performance query system indicating that there is no need to look for the closest hit encountered will do. This mode is most often used for boolean blocking queries. Performance improvement may be a factor of 3 or more, depending on the scene. To activate this mode use PxQueryFlag::eANY_HIT filter data field in PxQueryFilterData object, for instance:

```
PxQueryFilterData fd;
fd.flags |= PxQueryFlag::eANY_HIT; // note the OR with the default
bool status = scene->raycast(origin, unitDir, maxDistance, hit,
                             PxHitFlags(PxHitFlag::eDEFAULT), fdA
```

Multiple hits

All three query types (raycast, overlap, sweep) can also report multiple hits in the scene.

- To activate this mode for raycasts use the PxRaycastHitBuffer constructor provided buffer for touching hits.
- In this mode all hits default to 'touching' type and are

PxRaycastBuffer::touches array.

For instance:

```
PxScene* scene;
PxVec3 origin = ...; // [in] Ray origin
PxVec3 unitDir = ...; // [in] Normalized ray direc
PxReal maxDistance = ...; // [in] Raycast max distance

const PxU32 bufferSize = 256; // [in] size of 'hitBuffer'
PxRaycastHit hitBuffer[bufferSize]; // [out] User provided buffe
PxRaycastBuffer buf(hitBuffer, bufferSize); // [out] Blocking and

// Raycast against all static & dynamic objects (no filtering)
// The main result from this call are all hits along the ray, sto
scene->raycast(origin, unitDir, maxDistance, buf);
for (PxU32 i = 0; i < buf.nbTouches; i++)
    animateLeaves(buf.touches[i]);
```

The same mechanism is used for overlaps (use PxOverlapBuffer with F sweeps (PxSweepBuffer with PxSweepHit[]).

Multiple hits with blocking hit

In the snippet for multiple hits above we only expected touching hits. If encountered along with touching hits, it will be reported in PxHitBuffer:: the touch buffer will contain only touching hits which are closer. This cc in scenarios such as bullets going through windows (breaking them on t of a tree (making them rustle) until they hit a blocking object (a concrete

```
// same initialization code as in the snippet for multiple hits
bool hadBlockingHit = scene->raycast(origin, unitDir, maxDistance
if (hadBlockingHit)
    drawWallDecal(buf.block);
for (PxU32 i = 0; i < buf.nbTouches; i++)
{
    assert(buf.touches[i].distance <= buf.block.distance);
    animateLeaves(buf.touches[i]);
}
```

- By default, hits are assumed to be touching when a touch buffer is filter callback should return `PxQueryHitType::eBLOCK` to denote th
See *Filtering* for details.
- For `overlap()` queries all touching hits will be recorded even if encountered and `PxQueryFlag::eNO_BLOCK` flag is set.

Filtering

Filtering controls how shapes are excluded from scene query results as reported. All three query types support the following filtering parameters

- a *PxQueryFilterData* structure, containing both *PxQueryFlags* and *PxQueryFilterData*
- an optional *PxQueryFilterCallback*

PxQueryFlag::eSTATIC, PxQueryFlag::eDYNAMIC

PxQueryFlag::eSTATIC and *PxQueryFlag::eDYNAMIC* flags control whether a query should include shapes from the static and/or dynamic query structure. This is an efficient way to filter out all static/dynamic shapes. For example an engine that applies forces to all dynamics in a region could use a spherical *overlapSphere* query with the *PxQueryFlag::eDYNAMIC* flag to exclude all statics since forces can only be applied to dynamic objects. By default both statics and dynamics are included in queries.

For instance:

```
PxScene* scene;
PxVec3 origin = ...; // [in] Ray origin
PxVec3 unitDir = ...; // [in] Normalized ray direction
PxReal maxDistance = ...; // [in] Raycast max distance
PxRaycastHit hit; // [out] Raycast results

// [in] Define filter for static objects only
PxQueryFilterData filterData(PxQueryFlag::eSTATIC);

// Raycast against static objects only
// The main result from this call is the boolean 'status'
bool status = scene->raycast(origin, unitDir, maxDistance, hit, P
```

PxQueryFlag::ePREFILTER, PxQueryFlag::ePOSTFILTER

Scene queries are performed in three phases: broad phase, midphase and narrow phase.

- Broad phase traverses the global scene spatial partitioning structure to find candidates for mid and narrow phases.
- Midphase traverses the triangle mesh and heightfield internal culling to find a smaller subset of the triangles in a mesh reported by the broad phase.
- Narrow phase performs exact intersection tests (ray test for raycast, exact sweep shape tests or overlap tests for sweep() and overlap()).

To implement custom filtering in queries, set the *PxQueryFlag::eFILTER* and *PxQueryFlag::ePOSTFILTER* flags and subclass *PxQueryFilterCallback* to implement filtering logic.

- Pre-filtering happens before midphase and narrow phase and objects that are efficiently discarded before the potentially expensive exact collision tests are more expensive for triangle meshes, heightfields, convexes than raycast and overlap tests involving only simple shapes (capsules and boxes.)
- Post-filtering happens after the narrow phase test and can therefore use the test (such as *PxRaycastHit.position*) to determine whether the hit is discarded or not. These results can be accessed via the *hit* input parameter of the post-filtering callback (*PxQueryFilterCallback::postFilter*). For raycasts, use `static_cast<PxRaycastHit&>(hit)`, access data specific to a raycastHit similarly for overlaps (*PxOverlapHit*) and sweeps (*PxSweepHit*.)

The implementation of a filtering callback returns a *PxQueryHitType* result.

- *eNONE* indicates that the hit should be discarded.
- *eBLOCK* indicates that the hit is blocking.
- *eTOUCH* indicates that the hit is touching.

Whenever a raycast(), sweep() or overlap() query was called, the *PxHitCallback::nbTouches* and *PxHitCallback::touches* parameters, which are the number of hits that are no further (touchDistance <= blockDistance) than the closest hit.

will be reported. For example, to record all hits from a raycast query, return `eTOUCH`.

Note: Returning `eTOUCH` from a filter callback requires the hit buffer to have a non-zero `::touches` array, otherwise PhysX will generate an error and discard any touching hits.

Note: `eBLOCK` should not be returned from user filters for `overlap()`. It will result in undefined behavior, and a warning will be issued. If the `PxQueryFlag::eNO_BLOCK` flag is set, the `eBLOCK` will instead be automatically converted to an `eTOUCH` and the warning suppressed.

PxQueryFlag::eANY_HIT

Use this flag to force the query to report the first encountered hit (which is closest) as a blocking hit. Performance may be more than three times faster on the scenario. Best gains can be expected for long raycasts/swept spheres intersecting object, or overlaps with multiple intersecting objects.

- Also see `PxHitFlag::eMESH_ANY`

PxQueryFlag::eNO_BLOCK

Use this flag when you want to override the `eBLOCK` value returned by a filter. It will return `eTOUCH` or in cases when no blocking hits are expected (in this case the performance hint.) All hits will then be reported as touching regardless of the return value. The hit callback/buffer object provided to the query is required to have a non-zero `PxHitBuffer::touches` buffer when this flag is used. Significant performance gains should only be expected for scenarios where the touching hit buffer overflows.

Note: this flag overrides the return value from pre and post-filter functions. If `eBLOCK` were previously returned as blocking will instead be returned as touching.

PxFilterData fixed function filtering

A fast, fixed-function filter is provided by *PxFilterData*, a 4*32-bit bitmask in filtering equation. Each shape has a bitmask (set via `PxShape::set` and the query also has a bitmask.

The query data is used differently by batched and unbatched queries (batched queries). For unbatched queries, the following rules are applied:

- If the query's bitmask is all zeroes, custom filtering and intersection as normal.
- Otherwise, if the bitwise-AND value of the query's bitmask and the zero, the shape is skipped

Or in other words:

```
PxU32 keep = (query.word0 & object.word0)
             | (query.word1 & object.word1)
             | (query.word2 & object.word2)
             | (query.word3 & object.word3);
```

This hardcoded equation can provide simple filtering while avoiding overhead of the filtering callback. For example, to emulate the behavior of groups, define the groups as follows:

```
enum ActiveGroup
{
    GROUP1    = (1<<0),
    GROUP2    = (1<<1),
    GROUP3    = (1<<2),
    GROUP4    = (1<<3),
    ...
};
```

When shapes are created, they can be assigned to the a group, for example:

```
PxShape* shape; // Previously created shape
```

```
PxFilterData filterData;  
filterData.word0 = GROUP1;  
shape->setQueryFilterData(filterData);
```

Or to multiple groups, for example GROUP1 and GROUP3:

```
PxShape* shape; // Previously created shape  
  
PxFilterData filterData;  
filterData.word0 = GROUP1|GROUP3;  
shape->setQueryFilterData(filterData);
```

When performing a scene query, select which groups are active for example GROUP2 and GROUP3 - as follows:

```
PxScene* scene;  
PxVec3 origin = ...; // [in] Ray origin  
PxVec3 unitDir = ...; // [in] Normalized ray direction  
PxReal maxDistance = ...; // [in] Raycast max distance  
PxRaycastBuffer hit; // [out] Raycast results  
  
// [in] Define what parts of PxRaycastHit we're interested in  
const PxHitFlags outputFlags = PxHitFlag::eDISTANCE | PxHitFlag::eGROUPS;  
  
// [in] Raycast against GROUP2 and GROUP3  
PxQueryFilterData filterData = PxQueryFilterData();  
filterData.data.word0 = GROUP2|GROUP3;  
  
bool status = scene->raycast(origin, unitDir, maxDistance, hit, outputFlags);
```

User defined hit callbacks for unbounded results

Queries can sometimes return a very large number of results (for example very large objects or in areas with high object density), and it can be expensive to reserve a sufficiently large memory buffer. The classes `PxSweepCallback` and `PxOverlapCallback` provide efficient callback methods for such scenarios. For instance a *raycast* query with a `PxRaycastCallback` return all touch hits via multiple virtual `PxHitCallback::processTouches()`

```
struct UserCallback : PxRaycastCallback
{
    UserData data;
    virtual PxAgain processTouches(const PxRaycastHit* buffer, Px
        // This callback can be issued multiple times and can be
        // to process an unbounded number of touching hits.
        // Each reported touching hit in buffer is guaranteed to
        // the final block hit after the query has fully executed
    {
        for (PxU32 i = 0; i < nbHits; i++)
            animateLeaves(buffer[i], data);
    }
    virtual void finalizeQuery()
    {
        drawWallDecal(this->block, data);
    }
};

PxScene* scene;
PxVec3 origin = ...; // [in] Ray origin
PxVec3 unitDir = ...; // [in] Normalized ray direction
PxReal maxDistance = ...; // [in] Raycast max distance

UserCallback cb; cb.data = ...;
scene->raycast(origin, unitDir, maxDistance, cb); // see UserCall
```

In this code snippet the `raycast()` query will potentially invoke `processTouches()` multiple times, with all touching hits already clipped to the globally nearest block

- Note that the query can be up to twice as expensive in case all e

not fit in the provided touches buffer and a blocking hit was also fou

- Also see PxQueryFlag::eNO_BLOCK

Batched queries

PhysX supports batching of scene queries via the *PxBatchQuery* interface, which may simplify multi-threaded implementations.

The batched query feature has been deprecated in PhysX version 3.4.

- *PxBatchQuery* interface facilitates batching and execution of multiple queries together. *PxBatchQuery* buffers raycast, overlap and sweep queries. When *PxBatchQuery::execute()* is called.
- Use *PxScene::createBatchQuery(const PxBatchQueryDesc& desc, PxBatchQuery)* to create a *PxBatchQuery* object.
- The hardcoded filtering equation is not used for batched queries. Instead, you can use two filter shaders, respectively running before (*PxBatchQueryPreFilterShader*) and after (*PxBatchQueryPostFilterShader*) the exact per-shape filtering equation. *PxBatchQueryDesc::preFilterShader* and *PxBatchQueryDesc::postFilterShader*.
- *BatchQueryFilterData::filterShaderData* will be copied and passed to the filter shaders via the *constantBlock* parameter.
- Results are written to user-defined buffers *PxBatchQueryResults* and *PxBatchQueryHits*. *PxBatchQueryDesc*, in the same order queries were queued in the *PxBatchQuery* object.
- The results and hits buffers for the each query type used (raycast, overlap or sweep) are specified separately.
- These buffers can be changed before each batch query execute. If they are not, PhysX will produce a warning for batched queries with NULL results or hits buffers for the corresponding query type (raycast, overlap or sweep).

Volume Caching

PxVolumeCache provides a mechanism for accelerating scene queries. It implements caching for objects within a specified volume and provides *PxScene* for executing raycasts, overlaps, and sweeps. *PxVolumeCache* provides a performance boost when objects within the same localized region of space are queried multiple times, either within the same simulation frame or on a later frame.

The volume cache feature has been deprecated in PhysX version 3.4.

Some expected use cases for *PxVolumeCache* are:

- A particle system with many raycasts performed for each particle within a localized cloud.
- Multiple short range character controller raycasts within the same character.
- Caching query results across multiple frames, the cache can be filled with a volume on previous frame (possibly extruded in the anticipated movement) and then queried with a smaller volume.

The cache has a maximum capacity, specified separately for dynamic and static objects in *PxScene::createVolumeCache()*.

For purposes of multithreaded access, any operation on the cache occurs on the scene.

Filling the Cache

To fill the cache, call *PxVolumeCache::fill()*. This will query the scene for objects overlapping with the volume defined by the geometry and transform array. The results are stored in an internal buffer up to the maximum sizes for static and dynamic objects. For *PxBoxGeometry*, *PxSphereGeometry* and *PxCapsuleGeometry* a cache volume. The call will always refill both the static and dynamic internal buffers.

if the new volume lies entirely within the previous cached volume. It returns `PxVolumeCache::FillStatus`.

Subsequent queries against the cache (raycasts, overlaps, sweeps, for cache automatically using the same volume if the scene query successfully updated since the last fill. The update status is tracked independently of dynamics, so a query might only refill the cache for dynamics while returning results for statics. If any attempt to fill or refill fails, the cache is invalid and the subsequent query will attempt to fill it.

Querying the Cache

`PxVolumeCache` provides an API for raycasts, sweeps and overlaps that is similar to the scene query API. The main difference in signatures is that *Single Objects* are supported for `PxVolumeCache` queries. Query results are returned via the `PxVolumeCache::Iterator::shapes()` callback, and the query may be issued multiple times to deliver multiple batches of results.

- Raycasts, overlaps and sweeps against a valid cache will return results that overlap the cache volume, but is guaranteed to return all such volumes.
- Raycasts, overlaps and sweeps against an invalid cache will return no results. In this case results may be returned which do not overlap the cache volume.

Since the cache refills automatically on any query where the scene has been updated, two conditions guarantee that a query against the cache that lies entirely within the cache volume will always return exactly the same shapes as querying the scene directly. If a query does not lie entirely within the cache volume (and the cache is valid) only the shapes which overlap the cache volume will be returned. If a query is issued against an invalid cache which `fill()` has never been called, an error is reported.

The cache also provides a low-level `forEach()` mechanism that iterates over scene objects. If `forEach()` is executed on a cache for which `fill()` has never been called, it will return results that overlap the cached volume directly from the scene. This procedure

allocation of a temporary buffer, and if the allocation fails, *forEach()* message and return.

This code snippet shows how to use `PxVolumeCache`:

```
PxScene* scene;
PxVec3 poi = ...; // point of interest
PxVec3 origin = ...; // [in] Ray origin
PxVec3 unitDir = ...; // [in] Normalized ray direc
PxReal maxDistance = ...; // [in] Raycast max distance
PxRaycastBuffer hit; // [out] Raycast results
const PxU32 maxStatics = 32, maxDynamics = 8;

// persistent cache, valid until invalidated by object movement,
// insertion or deletion
PxVolumeCache* cache = scene->createVolumeCache(maxStatics, maxDy
cache->setMaxNbStaticShapes(64); cache->setMaxNbDynamicShapes(16)

// fill the cache using a box geometry centered around the point
cache->fill(PxBoxGeometry(PxVec3(1.0f)), PxTransform(position));

...

// Perform multiple raycast queries using the cache
PxRaycastBuffer hit;
const bool status = cache->raycast(origin, unitDir, maxDistance,

// low level iterator for stored actor/shape pairs
struct UserIterator : PxVolumeCache::Iterator
{
    UserData userData;
    virtual void shapes(PxU32 count, const PxActorShape* actorSha
    {
        for (PxU32 i = 0; i < count; i++)
            doSomething(actorShapePairs[i].actor, actorShapePairs[
    }
} iter;

// invoke UserIterator::shapes() callback for all actor/shape pai
cache->forEach(iter);
```



```

PxQueryFilterData(), NULL, cac
if(status)
{
    // We hit a shape. Cache it for next frame.
    persistentCache.shape = hit.block.shape;
    persistentCache.faceIndex = hit.block.faceIndex;
}
else
{
    // We did not hit anything. Reset the cache for next frame.
    persistentCache = PxQueryCache();
}

```

Caching can also be useful in queries looking for the closest blocking hit with the `eANY_HIT` flag. In this case, testing the previously closest object first can shorten the query distance very early, leading to fewer total narrow phase tests and early out from the traversal.

Note: PhysX does not detect stale pointers, so the application is responsible for ensuring the validity of cached object pointers when shapes are deleted.

Note: Overlaps do not support single hit blocking caches.

PxPruningStructureType

PhysX SDK offers different pruning structures which are used to accelerate queries. This paragraph describes the differences between them.

Generalities

The Scene Query system uses two different acceleration structures, and an AABB tree.

The grid builds quickly, in $O(n)$ time, with queries executing in between $O(1)$ depending on how uniformly the objects are distributed in space, with worst case performance of $O(N)$ when all objects are clustered in the same grid cell.

The tree builds in $O(n \log(n))$ time, but queries with a single result typically take $O(1)$ time. Queries returning multiple results will traverse more of the tree, though a query returning all of the objects in the scene in $O(n)$ time. The tree degenerates when the same topology is maintained too long as objects are added and in pathological cases query performance may degrade to $O(n)$ time.

Acceleration structures must be continually modified in accordance with objects added or removed, or object AABB updates due to changes in position. To minimize the cost, modifications are deferred for as long as possible. When adding or removing objects or updating AABBs occurs in amortized constant time. Modifications are deferred until the changes 'commit'. This happens on the next query or the next `fetchResults()`. To force an immediate commit, call `PxScene::flushQueryUpdates()` function.

The exact details of the commit process depend on the values of `dynamicStructure` specified in `PxSceneDesc`.

To avoid automatic resizing triggered by insertions into internal structures, reserve the space in advance. See `PxSceneDesc::maxNbDynamicShapes` and `PxSceneDesc::maxNbDynamicShapes`.

PxPruningStructureType::eNONE

The acceleration structure is similar to a hierarchical grid. Committing changes requires a full rebuild. This is a good choice if you expect to rarely or never update the acceleration structure.

PxPruningStructureType::eSTATIC_AABB_TREE

The acceleration structure is a tree. Committing changes requires a full rebuild, which is generally recommended, but can be a good choice for staticStructure if your scene objects are created on initialization, and not modified thereafter. If you add or remove static geometry, the default eDYNAMIC_AABB_TREE setting is a better choice, although it has a higher memory footprint than that of eSTATIC_AABB_TREE.

PxPruningStructureType::eDYNAMIC_AABB_TREE

In this case, both the tree and the grid are used, and each query searches both the tree and the grid.

The tree is initially built by the first commit. Once a tree is built, commit operations proceed as follows:

- * the tree is refitted in accordance with updates to the objects it contains.
- * added objects are inserted into the grid. Such additions are not reflected in the tree.
- * removed objects currently in the grid, or changes to AABBs of objects in the grid, are reflected in the grid, but not in the tree.

The grid is rebuilt.

In addition, a new tree is incrementally built during fetchResults(), over a number of frames controlled by PxScene's dynamicTreeRebuildRateHint attribute. When the rebuild starts, it includes all of the objects in the current tree and grid. When the rebuild is complete, the new tree is refitted in accordance with any AABB changes since the build started, and then replaces the current tree. Any objects added since the start of the build remain in the grid.

To force a full immediate rebuild, call PxScene::forceDynamicTreeRebuild(). This is useful in cases such as the following:

- a slow rebuilt rate is typically desirable, but occasionally a large additions creates high occupancy in the grid, especially if the add so as to put pressure on just a few of the grid cells.
- you are moving many objects across large distances, since refittir degrade the quality of the current tree

PxPruningStructure

Provides access to precomputed pruning structure used to accelerate against newly added actors.

A pruning structure can be provided to `PxScene::addActors`. The shapes will then be directly merged into the scene's AABB tree, with AABB tree recompute:

```
// Create pruning structure from given actors.
PxPruningStructure* ps = PxPhysics::createPruningStructure(&actor
// Add actors into a scene together with the precomputed pruning
PxScene::addActors(*ps);
ps->release();
```

A `PxPruningStructure` object can be serialized into a collection together

For usage of `PxPruningStructure` please refer to the snippet `SnippetPru`

A typical use case for `PxPruningStructure` is a large world scenario where positioned actors get streamed in.

Merge process

The merge process into the scene query acceleration structure `PxPruningStructureType`: * `eSTATIC_AABB_TREE` - the pruning structure is merged directly into scene's AABBTREE. This might unbalance the tree and it is recomputed the static tree at some point. * `eDYNAMIC_AABB_TREE` structure is merged into a temporary pruning structure until the scene's AABB tree is computed.

Introduction

PhysX support for vehicles has been significantly reworked in 3.0. The new `NxWheelShape` class of 2.8.x, a more optimal integration of the core vehicle simulation code has been developed. More specifically, the vehicle simulation code now sits outside the core SDK in a manner similar to `PhysXExtensions`. This allows vehicles to be updated in a single pass as well as promotes a more modular approach to modeling vehicle data. Vehicle support has been extended from simple suspension/wheel/tire modeling of 2.8.x to a more complete model that includes vehicle components including engine, clutch, gears, gearbox, differential, suspensions, and chassis. A quick glance at the data structure `PxVehicleComponents.h` will provide a flavor of the behaviors supported for vehicles.

Algorithm

The PhysX Vehicle SDK models vehicles as collections of sprung masses. A sprung mass represents a suspension line with associated wheel and a collection of sprung masses have a complementary representation as a rigid body actor whose mass, center of mass, and moment of inertia matches exactly the coordinates of the sprung masses. This is illustrated below.

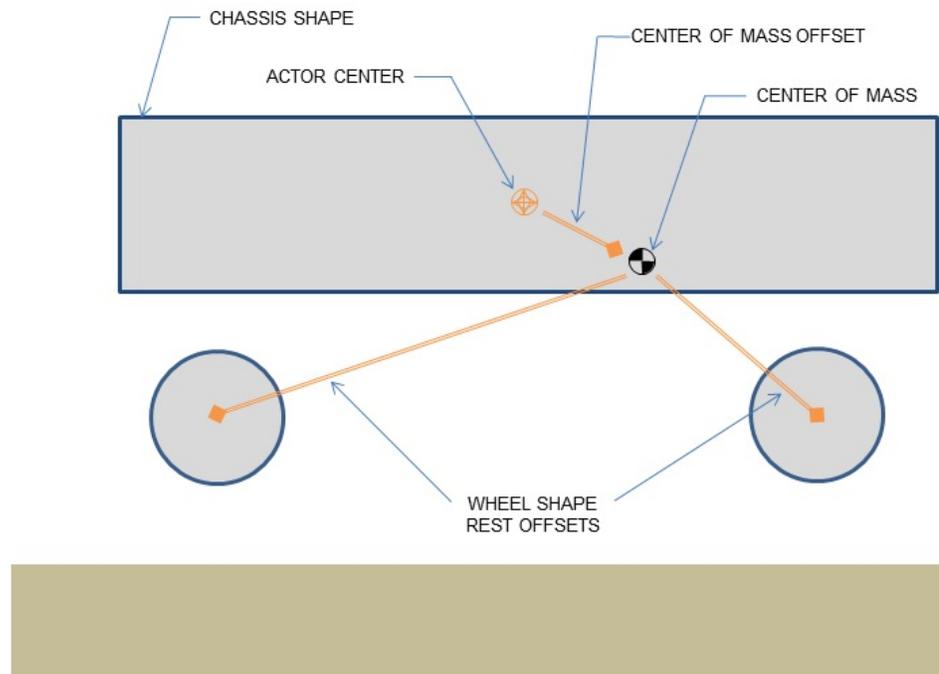


Figure 1a: Vehicle representation as a rigid body actor with shapes for wheels. Note that the wheel rest offsets are specified relative to the

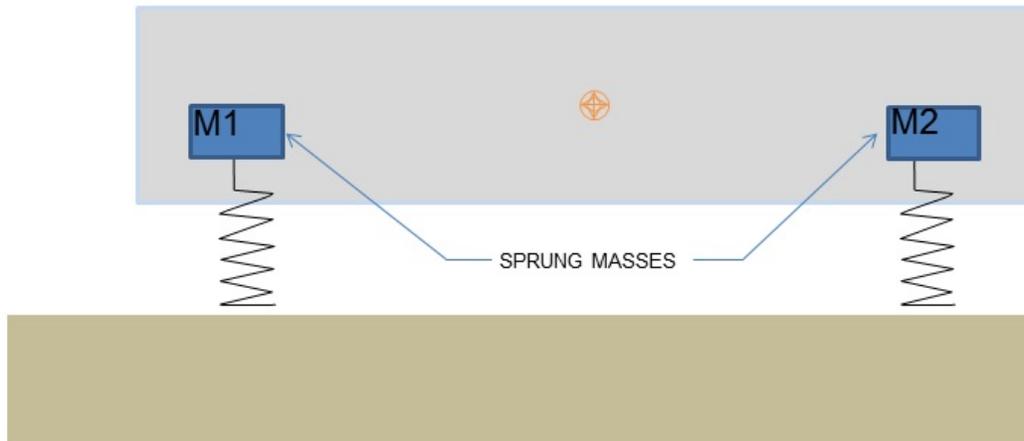


Figure 1b: Vehicle representation as a collection of sprung masses of

The relationship between the sprung mass and rigid body vehicle representation is mathematically formalized with the rigid body center of mass equations:

$$M = M1 + M2$$

$$X_{cm} = (M1 \times X1 + M2 \times X2) / (M1 + M2)$$

where M1 and M2 are the sprung masses; X1 and X2 are the sprung mass positions in actor space; M is the rigid body mass; and Xcm is the rigid body center of mass.

The purpose of the PhysX Vehicle SDK update function is to compute suspension forces using the sprung mass model and then to apply the aggregate forces to the PhysX SDK rigid body representation in the form of a modified velocity. Interaction of the rigid body actor with other scene objects and collision update is then managed by the PhysX SDK.

The update of each vehicle begins with a raycast for each suspension point. The raycast starts just above the top of the tire at maximum compression and travels along the direction of suspension travel to a position just below the bottom of the tire at maximum droop. This is shown in the diagram below.

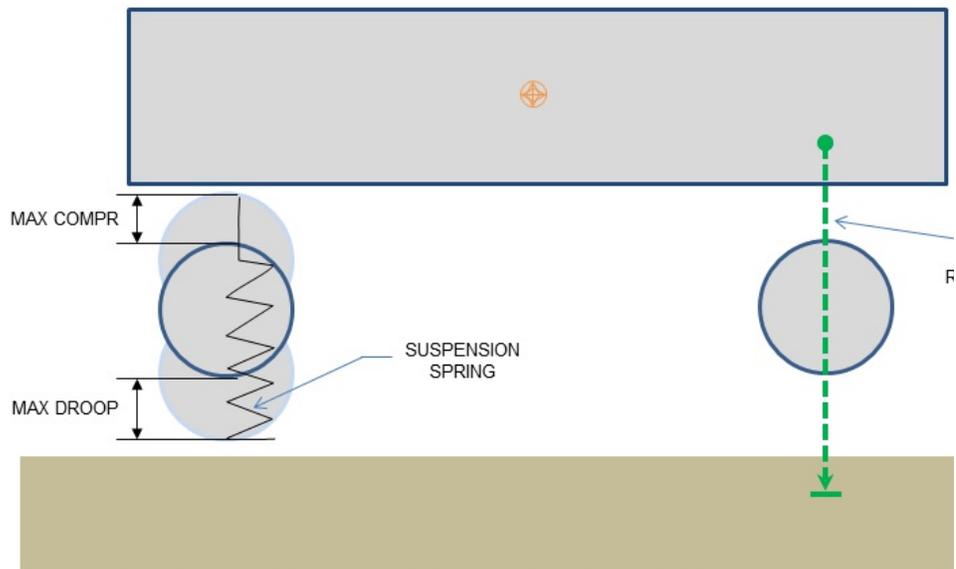


Figure 2: Suspension limits and suspension raycasts

The suspension force from each elongated or compressed spring is computed and applied to the rigid body. Additionally, the suspension raycast is used to compute the load that is bearing down on the tire. This load is used to compute the tire forces that will be generated in the contact plane and then added to the rigid body. The tire force computation actually takes into account a number of factors including steer angle, camber angle, friction, wheel radius, and rigid body momentum. The aggregated force of all tire and suspension forces is applied to the rigid body actor associated with the vehicle so that the suspension limits are modified accordingly in the next PhysX SDK update.

In addition to being collections of sprung masses, PhysX vehicles also support drive models. The center of the drive model is a torsion clutch, which couples the wheels and the engine via forces that arise from differences in rotational speeds on either side of the clutch. At one side of the clutch is the engine, which is powered by the accelerator pedal. The engine is modeled as a rigid body whose rotation is limited to a single degree of rotational freedom. At the other side of the clutch are the gearing system, the differential and the wheels. The rotational speed of the other side of the clutch can be computed directly from the rotational speed of the wheels that are coupled to the clutch through the clutch.

This model naturally allows engine torques to propagate to the wheels to propagate back to the engine, just as in a standard car.

The data describing each component of the PhysX vehicle can be found [Guide](#).

First Code

Vehicle SDK Initialization

Before using the vehicle SDK it must first be initialized in order to set threshold values from various tolerance scales. This is as straightforward as the following function:

```
PX_C_EXPORT bool PX_CALL_CONV PxInitVehicleSDK
    (PxPhysics& physics, PxSerializationRegistry* serializationRegistry);
```

This function should be called after setting up the required PxPhysics instances. If vehicle serialization is required a PxSerializationRegistry instance should be specified. A PxSerializationRegistry instance can be created by calling PxSerialization::createSerializationRegistry(), see [Serialization](#).

The basis vectors of the vehicle simulation must also be configured so that longitudinal and lateral tire slips may be unambiguously computed:

```
void PxVehicleSetBasisVectors(const PxVec3& up, const PxVec3& forward,
```

This function can be called at any time prior to the first execution of PxVehicle::simulate.

The rigid body actors associated with vehicles can be updated either with velocity modifications or updated with an acceleration that is applied to the vehicle. The following function can be used to select the required update mode for the vehicle SDK simulate call.

```
void PxVehicleSetUpdateMode(PxVehicleUpdateMode::Enum vehicleUpdateMode);
```

As expected, the vehicle SDK also has a shutdown process which needs to be called before the vehicle SDK simulate call.

```
PX_C_EXPORT void PX_CALL_CONV PxCloseVehicleSDK
    (PxSerializationRegistry* serializationRegistry = NULL);
```

This needs to be called before the PxPhysics instance and PxVehicle::simulate call.

released; that is, the order of shutdown is the reverse of the initialization. If serialization is required the `PxSerializationRegistry` specified for `PxInitVehicleSDK` to be passed to `PxCloseVehicleSDK`. If vehicle serialization is used, `PxCloseVehicleSDK` should be called before closing the `PhysXExtensions`.

As an illustration of the usage of these functions, `SnippetVehicle4W` initialization code:

```
PxInitVehicleSDK(*gPhysics);
PxVehicleSetBasisVectors(PxVec3(0,1,0), PxVec3(0,0,1));
PxVehicleSetUpdateMode(PxVehicleUpdateMode::eVELOCITY_CHANGE);
```

The shutdown code in `SnippetVehicle4W` is as follows:

```
PxCloseVehicleSDK();
```

Introduction To Vehicle Creation

The following pseudo-code illustrates the basic process of setting up an instance:

```
const PxU32 numWheels = 4;

PxVehicleWheelsSimData* wheelsSimData = PxVehicleWheelsSimData::allocate(numWheels);
wheelsSimData->setupWheelsSimulationData(wheelsSimData);

PxVehicleDriveSimData4W driveSimData;
driveSimData->setupDriveSimData(driveSimData);

PxRigidDynamic* vehActor = myPhysics.createRigidDynamic(startPose);
vehActor->setupVehicleActor(vehActor);
myScene.addActor(*vehActor);

PxVehicleDrive4W* vehDrive4W = PxVehicleDrive4W::allocate(numWheels);
vehDrive4W->setup(physics, veh4WActor, *wheelsSimData, driveSimData);
wheelsSimData->free();
```

The code above first instantiates a `PxVehicleWheelsSimData` instance and a `PxVehicleDrive4W` instance that are large enough to store configuration data for four wheels. This

includes fields such as suspension strength and damping rate, wheel and suspension travel direction. The next step is to create a PxVehicle instance. This structure stores the configuration of the drive model and such as engine peak torque, clutch strength, gearing ratios, and A correction. Following this, a PxRigidBodyActor is instantiated with geometry for the wheels and chassis as well as dynamic properties such as moment of inertia, and center of mass. The final step is to instantiate a PxVehicle instance and associate it with the actor and the vehicle configuration data.

The functions `setupWheelsSimulationData`, `setupDriveSimData` and `setupVehicleActor` are actually quite involved and shall be discussed in [setupWheelsSimulationData](#), [setupDriveSimData](#) and [setupVehicleActor](#).

Introduction To Vehicle Update

The PhysX Vehicles SDK utilizes batched scene queries to query the scene for each tire. A more detailed discussion of PhysX batched scene queries is in Section [Batched queries](#).

The following pseudo-code initializes a batched scene query with buffers for a single vehicle with four wheels:

```
PxRaycastQueryResult sqResults[4];
PxRaycastHit sqHitBuffer[4];
PxBatchQueryDesc sqDesc(4, 0, 0);
sqDesc.queryMemory.userRaycastResultBuffer = sqResults;
sqDesc.queryMemory.userRaycastTouchBuffer = sqHitBuffer;
sqDesc.queryMemory.raycastTouchBufferSize = 4;
sqDesc.preFilterShader = myFilterShader;
PxBatchQuery* batchQuery = scene->createBatchQuery(sqDesc);
```

The `PxBatchQuery` instance is typically instantiated as part of the initialization of a vehicle and then reused each frame. It is possible to instantiate a `PxBatchQuery` instance for a single vehicle or to instantiate a single `PxBatchQuery` instance with buffers for multiple wheels of a batched array of vehicles. The only restriction is that all buffers and associated buffers configured at the start of a vehicle simulation and are not changed until the end of the vehicle simulation frame.

PhysX vehicles make use of scene query filter shaders to eliminate int vehicle issuing the raycast and with any geometry that is not to b drivable surface. More details for how to set up "myFilterShader" abo Section *Filtering*.

For a batch containing just a single 4-wheeled vehicle the suspensio performed with the following pseudo-code:

```
PxVehicleWheels* vehicles[1] = {myVehicle};  
PxVehicleSuspensionRaycasts(batchQuery, 1, vehicles, 4, sqResults
```

The function PxVehicleSuspensionRaycasts performs suspension rayca in the batched array of vehicles. Each element in the sqResults array raycast report for a single suspension. Pointers to contiguous blocks w stored by each vehicle in turn as the function iterates through the vel memory blocks are stored by each vehicle so that they may easily qu raycast results in PxVehicleUpdates. As a consequence, the sqResults until at least the end of PxVehicleUpdates and must have length at le total number of wheels in the vehicles array.

The vehicles are updated with the following function call:

```
PxVehicleUpdates(timestep, gravity, frictionPairs, 1, vehicles, N
```

The function PxVehicleUpdates updates the internal dynamics of each wheel shapes of the vehicle's actor and applies either velocity or accel the actor, depending on the update mode chosen with PxVehicleSett details can be found in Section *Wheel Pose* and Section *Vehicle Upda*; frictionPairs is basically a lookup table that associates unique fr combinations of tire type and PxMaterial. The idea here is to allow t tuned for each surface type. This shall be discussed in more depth in S *on Drivable Surfaces*.

Snippets

Four snippets are currently implemented to illustrate the operation of the SDK. These are:

1. SnippetVehicle4W
2. SnippetVehicleTank
 3. SnippetNoDrive
3. SnippetVehicleScale
4. SnippetVehicleMultiThreading
 5. SnippetVehicleWheelContactMod

Code snippets from each of these is used throughout the guide.

SnippetVehicle4W

SnippetVehicle4W demonstrates how to instantiate and update `PxVehicleDrive4W`. It creates a vehicle on a plane and then controls the vehicle to perform a number of choreographed maneuvers such as accelerate, handbrake, and turn.

SnippetVehicleTank

SnippetVehicleTank demonstrates how to instantiate and update `PxVehicleDriveTank`. It creates a tank on a plane and then controls the tank to perform a number of choreographed maneuvers such as accelerate, and hard turns.

SnippetVehicleNoDrive

SnippetVehicleNoDrive demonstrates how to instantiate and update `PxVehicleNoDrive`. It creates a vehicle on a plane and then controls the vehicle to perform a number of choreographed manoeuvres such as accelerate, and hard turns.

SnippetVehicleScale

SnippetVehicleScale demonstrates how to configure a PhysX vehicle with the chosen length scale. The snippet sets up a vehicle with meters as scale and then modifies the vehicle parameters so that they represent but with centimeters as the chosen length scale.

SnippetVehicleMultiThreading

SnippetVehicleMultiThreading demonstrates how to implement multi-threading. It creates multiple vehicles on a plane and then concurrently simulates them across multiple threads.

Advanced Concepts

Vehicle Creation

This Section discusses the configuration of vehicle simulation data and set up an actor that will represent the vehicle in the PhysX SDK. Section [Vehicle Creation](#) identified three distinct phases of vehicle configuration: wheel simulation data, configuration of drive simulation data and actor creation. Each of these phases is discussed in turn.

setupWheelsSimulationData

The following code, taken from SnippetVehicle4W, PxVehicleWheelsSimData:

```
void setupWheelsSimulationData(const PxF32 wheelMass, const PxF32
    const PxF32 wheelRadius, const PxF32 wheelWidth, const PxU32
    const PxVec3* wheelCenterActorOffsets, const PxVec3& chassisC
    const PxF32 chassisMass, PxVehicleWheelsSimData* wheelsSimDat
{
    //Set up the wheels.
    PxVehicleWheelData wheels[PX_MAX_NB_WHEELS];
    {
        //Set up the wheel data structures with mass, moi, radius
        for(PxU32 i = 0; i < numWheels; i++)
        {
            wheels[i].mMass = wheelMass;
            wheels[i].mMOI = wheelMOI;
            wheels[i].mRadius = wheelRadius;
            wheels[i].mWidth = wheelWidth;
        }

        //Enable the handbrake for the rear wheels only.
        wheels[PxVehicleDrive4WheelOrder::eREAR_LEFT].mMaxHandBr
        wheels[PxVehicleDrive4WheelOrder::eREAR_RIGHT].mMaxHandB
        //Enable steering for the front wheels only.
        wheels[PxVehicleDrive4WheelOrder::eFRONT_LEFT].mMaxSteer
        wheels[PxVehicleDrive4WheelOrder::eFRONT_RIGHT].mMaxStee
    }
}
```

```

//Set up the tires.
PxVehicleTireData tires[PX_MAX_NB_WHEELS];
{
    //Set up the tires.
    for(PxU32 i = 0; i < numWheels; i++)
    {
        tires[i].mType = TIRE_TYPE_NORMAL;
    }
}

//Set up the suspensions
PxVehicleSuspensionData suspensions[PX_MAX_NB_WHEELS];
{
    //Compute the mass supported by each suspension spring.
    PxF32 suspSprungMasses[PX_MAX_NB_WHEELS];
    PxVehicleComputeSprungMasses
        (numWheels, wheelCenterActorOffsets,
         chassisCMOffset, chassisMass, 1, suspSprungMasses);

    //Set the suspension data.
    for(PxU32 i = 0; i < numWheels; i++)
    {
        suspensions[i].mMaxCompression = 0.3f;
        suspensions[i].mMaxDroop = 0.1f;
        suspensions[i].mSpringStrength = 35000.0f;
        suspensions[i].mSpringDamperRate = 4500.0f;
        suspensions[i].mSprungMass = suspSprungMasses[i];
    }

    //Set the camber angles.
    const PxF32 camberAngleAtRest=0.0;
    const PxF32 camberAngleAtMaxDroop=0.01f;
    const PxF32 camberAngleAtMaxCompression=-0.01f;
    for(PxU32 i = 0; i < numWheels; i+=2)
    {
        suspensions[i + 0].mCamberAtRest = camberAngleAtRest
        suspensions[i + 1].mCamberAtRest = -camberAngleAtRes
        suspensions[i + 0].mCamberAtMaxDroop = camberAngleAtM
        suspensions[i + 1].mCamberAtMaxDroop = -camberAngleAt
        suspensions[i + 0].mCamberAtMaxCompression = camberAn
        suspensions[i + 1].mCamberAtMaxCompression = -camberA
    }
}

//Set up the wheel geometry.
PxVec3 suspTravelDirections[PX_MAX_NB_WHEELS];
PxVec3 wheelCentreCMOffsets[PX_MAX_NB_WHEELS];

```

```

PxVec3 suspForceAppCMOffsets[PX_MAX_NB_WHEELS];
PxVec3 tireForceAppCMOffsets[PX_MAX_NB_WHEELS];
{
    //Set the geometry data.
    for(PxU32 i = 0; i < numWheels; i++)
    {
        //Vertical suspension travel.
        suspTravelDirections[i] = PxVec3(0, -1, 0);

        //Wheel center offset is offset from rigid body center
        wheelCentreCMOffsets[i] =
            wheelCenterActorOffsets[i] - chassisCMOffset;

        //Suspension force application point 0.3 metres below
        //rigid body center of mass.
        suspForceAppCMOffsets[i] =
            PxVec3(wheelCentreCMOffsets[i].x, -0.3f, wheelCentreCMOffsets[i].y);

        //Tire force application point 0.3 metres below
        //rigid body center of mass.
        tireForceAppCMOffsets[i] =
            PxVec3(wheelCentreCMOffsets[i].x, -0.3f, wheelCentreCMOffsets[i].y);
    }
}

//Set up the filter data of the raycast that will be issued by the wheels
PxFilterData qryFilterData;
setupNonDrivableSurface(qryFilterData);

//Set the wheel, tire and suspension data.
//Set the geometry data.
//Set the query filter data
for(PxU32 i = 0; i < numWheels; i++)
{
    wheelsSimData->setWheelData(i, wheels[i]);
    wheelsSimData->setTireData(i, tires[i]);
    wheelsSimData->setSuspensionData(i, suspensions[i]);
    wheelsSimData->setSuspTravelDirection(i, suspTravelDirections[i]);
    wheelsSimData->setWheelCentreOffset(i, wheelCentreCMOffsets[i]);
    wheelsSimData->setSuspForceAppPointOffset(i, suspForceAppCMOffsets[i]);
    wheelsSimData->setTireForceAppPointOffset(i, tireForceAppCMOffsets[i]);
    wheelsSimData->setSceneQueryFilterData(i, qryFilterData);
    wheelsSimData->setWheelShapeMapping(i, i);
}
}

```

The function `PxVehicleComputeSprungMasses` computes the sprung suspension so that they collectively match the rigid body center of mass in the frame of the actor. It makes sense to perform `PxVehicleCompute` the frame of the actor because the rigid body center of mass is always in the actor's frame. The vehicle suspension system, on the other hand, is specified in the mass frame. As a consequence, the functions `setSuspForceAppPointOffset` and `setTireForceAppPointOffset` all describe the rigid body center of mass. The directness of this approach can make the rigid body center of mass a bit more involved than might be expected. To solve this, the function `PxVehicleUpdateCMassLocalPose` has been introduced, though its code is not shown above. This function recomputes and sets all suspension offsets for the sprung masses and sets them in a way that preserves the natural frequency ratio of each spring.

Details of many of the parameters and functions above can be found in the [Vehicle Setup Guide](#). The function `setupNonDrivableSurface`, which sets up scene collision for each suspension raycast, shall be discussed in more detail in Section [Collision](#). The link between `TIRE_TYPE_NORMAL` and tire friction shall be made in the [Tire Friction on Drivable Surfaces](#). Finally, the use of the function `setWheelPose` shall be clarified in Section [Wheel Pose](#).

setupDriveSimData

The following code, taken from `SnippetVehicle4W`, shows the `PxVehicleDriveSimData4W`:

```
PxVehicleDriveSimData4W driveSimData;
{
    //Diff
    PxVehicleDifferential4WData diff;
    diff.mType=PxVehicleDifferential4WData::eDIFF_TYPE_LS_4WD;
    driveSimData.setDiffData(diff);

    //Engine
    PxVehicleEngineData engine;
    engine.mPeakTorque=500.0f;
    engine.mMaxOmega=600.0f;//approx 6000 rpm
    driveSimData.setEngineData(engine);
}
```

```

//Gears
PxVehicleGearsData gears;
gears.mSwitchTime=0.5f;
driveSimData.setGearsData(gears);

//Clutch
PxVehicleClutchData clutch;
clutch.mStrength=10.0f;
driveSimData.setClutchData(clutch);

//Ackermann steer accuracy
PxVehicleAckermannGeometryData ackermann;
ackermann.mAccuracy=1.0f;
ackermann.mAxleSeparation=
    wheelsSimData->getWheelCentreOffset(PxVehicleDrive4Wheel
    wheelsSimData->getWheelCentreOffset(PxVehicleDrive4Wheel
ackermann.mFrontWidth=
    wheelsSimData->getWheelCentreOffset(PxVehicleDrive4Wheel
    wheelsSimData->getWheelCentreOffset(PxVehicleDrive4Wheel
ackermann.mRearWidth=
    wheelsSimData->getWheelCentreOffset(PxVehicleDrive4Wheel
    wheelsSimData->getWheelCentreOffset(PxVehicleDrive4Wheel
driveSimData.setAckermannGeometryData(ackermann);
}

```

Details of many of the parameters and functions above can be found in the [Vehicle Configuration Guide](#).

Configuring PxVehicleDriveSimDataNW and PxVehicleDriveSimDataTa is a very similar procedure, albeit with slightly different components. More examples can be found, for example, in SnippetVehicleTank.

setupVehicleActor

The following code, common to all vehicle snippets, sets up a rigid body actor with collision geometry, filter and dynamics data:

```

PxRigidDynamic* createVehicleActor
(const PxVehicleChassisData& chassisData,
PxMaterial** wheelMaterials, PxConvexMesh** wheelConvexMeshes, c
PxMaterial** chassisMaterials, PxConvexMesh** chassisConvexMeshe

```

```

PxPhysics& physics)
{
    //We need a rigid body actor for the vehicle.
    //Don't forget to add the actor to the scene after setting
    PxRigidDynamic* vehActor = physics.createRigidDynamic(PxT

    //Wheel and chassis query filter data.
    //Optional: cars don't drive on other cars.
    PxFilterData wheelQryFilterData;
    setupNonDrivableSurface(wheelQryFilterData);
    PxFilterData chassisQryFilterData;
    setupNonDrivableSurface(chassisQryFilterData);

    //Add all the wheel shapes to the actor.
    for(PxU32 i = 0; i < numWheels; i++)
    {
        PxConvexMeshGeometry geom(wheelConvexMeshes[i]);
        PxShape* wheelShape=PxRigidActorExt::createExclus
        wheelShape->setQueryFilterData(wheelQryFilterData
        wheelShape->setSimulationFilterData(wheelSimFilterData)
        wheelShape->setLocalPose(PxTransform(PxIdentity))
    }

    //Add the chassis shapes to the actor.
    for(PxU32 i = 0; i < numChassisMeshes; i++)
    {
        PxShape* chassisShape=PxRigidActorExt::createExclus
        chassisShape->setQueryFilterData(chassisQryFilterData)
        chassisShape->setSimulationFilterData(chassisSimFilterData)
        chassisShape->setLocalPose(PxTransform(PxIdentity))
    }

    vehActor->setMass(chassisData.mMass);
    vehActor->setMassSpaceInertiaTensor(chassisData.mMOI);
    vehActor->setCMassLocalPose(PxTransform(chassisData.mCMOF

    return vehActor;
}

```

The significance of wheelSimFilterData, chassisSimFilterData, wheelQryFilterData shall be discussed in Section *Filtering*. Further, the ordering of the wheel shapes in the above code at PxVehicleWheelsSimData::setWheelShapeMapping is clarified in Section

Filtering

In this Section the concepts behind vehicle query and vehicle simulation are described.

The key goal of scene query and simulation filtering for vehicles is to ensure that vehicles are supported by suspension spring forces without interference from other objects or self-intersection. The requirements for filtering are then as follows:

1. wheel shapes must not hit drivable surfaces
2. suspension raycasts can hit drivable surfaces
3. suspension raycasts must not hit the shapes of the vehicle itself

Ensuring that wheel shapes don't hit drivable surfaces can be achieved through simulation filtering. This is discussed in more detail in Section [Collision Filtering](#). To use the following simulation filter shader:

```
PxFilterFlags VehicleFilterShader
(PxFilterObjectAttributes attributes0, PxFilterData filterData0,
 PxFilterObjectAttributes attributes1, PxFilterData filterData1,
 PxPairFlags& pairFlags, const void* constantBlock, PxU32 constantBlockSize)
{
    PX_UNUSED(attributes0);
    PX_UNUSED(attributes1);
    PX_UNUSED(constantBlock);
    PX_UNUSED(constantBlockSize);

    if( (0 == (filterData0.word0 & filterData1.word1)) && (0 == (filterData0.word2 & filterData1.word2)) )
        return PxFilterFlag::eSUPPRESS;

    pairFlags = PxPairFlag::eCONTACT_DEFAULT;
    pairFlags |= PxPairFlags(PxU16(filterData0.word2 | filterData1.word2));

    return PxFilterFlags();
}
```

The snippets also apply simulation filter data to wheel shapes as follows:

```
PxFilterData wheelSimFilterData;
wheelSimFilterData.word0 = COLLISION_FLAG_WHEEL;
```

```
wheelSimFilterData.word1 = COLLISION_FLAG_WHEEL_AGAINST;

...

wheelShape->setSimulationFilterData(wheelSimFilterData);
```

Finally, the following simulation filter data is applied to drivable surfaces

```
PxFilterData simFilterData;
simFilterData.word0 = COLLISION_FLAG_GROUND;
simFilterData.word1 = COLLISION_FLAG_GROUND_AGAINST;

...

shapes[0]->setSimulationFilterData(simFilterData);
```

The combination of collision flags (COLLISION_FLAG_GROUND_AGAINST etc) and filter shader ensures don't collide with drivable surfaces.

A remarkably similar process may be employed to configure the query filters. This is accomplished in the vehicle snippets with the follow

```
void setupDrivableSurface(PxFilterData& filterData)
{
    filterData.word3 = (PxU32)DRIVABLE_SURFACE;
}

void setupNonDrivableSurface(PxFilterData& filterData)
{
    filterData.word3 = UNDRIVABLE_SURFACE;
}

PxQueryHitType::Enum WheelRaycastPreFilter
(PxFilterData filterData0, PxFilterData filterData1,
const void* constantBlock, PxU32 constantBlockSize,
PxHitFlags& queryFlags)
{
    //filterData0 is the vehicle suspension raycast.
    //filterData1 is the shape potentially hit by the raycast.
    PX_UNUSED(constantBlockSize);
    PX_UNUSED(constantBlock);
    PX_UNUSED(filterData0);
```

```

PX_UNUSED(queryFlags);
return ((0 == (filterData1.word3 & DRIVABLE_SURFACE)) ?
        PxQueryHitType::eNONE : PxQueryHitType::eBLOCK);
}

```

Each vehicle wheel is given filter data configured with `setupNonDrivableSurface` passed to the vehicle with:

```

wheelsSimData->setSceneQueryFilterData(i, qryFilterData);

```

The parameter `filterData0` in `WheelRaycastPreFilter` corresponds to the `qryFilterData` passed to the vehicle's `PxVehicleWheelsSimData::setSceneQueryFilterData`. The parameter `filterData1`, on the other hand, corresponds to the query filter data of a shape potentially hit by the vehicle snippets the shape of the drivable ground plane has been previously configured with the function `setupDrivableSurface`. This satisfies the suspension raycasts can hit drivable surfaces. Vehicle shapes, on the other hand, are configured with `setupNonDrivableSurface`. This satisfies the restriction that raycasts must not hit the vehicle issuing the raycasts but also prevent driving on any other vehicles that might be added to the scene. This exclusion can readily be avoided by employing a more complex filter shader that perhaps uses unique IDs encoded in both the shape filter data and the filter data applied to the vehicle. Care must be taken, however, to configure the filters to ensure that shapes only interact with the shapes of other vehicles.

Note: It is vital that `WheelRaycastPreFilter` returns `PxQueryHitType::eNONE` if no raycast hit is allowed for the filter data pair. Using `PxQueryHitType::eBLOCK` means that each raycast returns either no hits or just the hit closest to the start of the raycast. This is important because `PxVehicleSuspensionRaycasts` and `PxVehicleUpdates` expect a one-to-one correspondence between each element in the `PxRaycastQueryResult` and `PxRaycastHit` arrays passed to the query.

Tire Friction on Drivable Surfaces

In this Section setting up tire types, drivable surface types, and combinations of tire and surface type shall be discussed.

To implement a unique friction value for each combination of tire type and surface type, it is first necessary to assign tire types to tires. In Section [setupWheelsSi](#) type was assigned to each tire:

```
//Set up the tires.
PxVehicleTireData tires[PX_MAX_NB_WHEELS];
{
    //Set up the tires.
    for(PxU32 i = 0; i < numWheels; i++)
    {
        tires[i].mType = TIRE_TYPE_NORMAL;
    }
}
```

Assigning a type to each surface is a little more complex. The basic suspension raycast hit returns the PxMaterial of the shape hit by knowledge of a PxMaterial array it is possible to associate the type of a surface with an index of the PxMaterial array element that matches the material hit by a lookup and the table of friction values is managed by PxVehicleDrivableSurfaceToTireFrictionPairs. To make the feature more flexible, an element of the PxMaterial array is actually associated with a PxVehicleDrivableSurfaceType instance. This allows multiple PxMaterials to share the same surface type.

In the vehicle snippets the following code makes the association between surface type and then associates each combination of tire and surface type with a friction value:

```
PxVehicleDrivableSurfaceToTireFrictionPairs* createFrictionPairs
(const PxMaterial* defaultMaterial)
{
    PxVehicleDrivableSurfaceType surfaceTypes[1];
    surfaceTypes[0].mType = SURFACE_TYPE_TARMAC;

    PxMaterial* surfaceMaterials[1];
    surfaceMaterials[0] = defaultMaterial;
}
```

```

PxVehicleDrivableSurfaceToTireFrictionPairs* surfaceTirePairs
    PxVehicleDrivableSurfaceToTireFrictionPairs::allocate(MAX
        MAX_NUM_SURFACE_TYPES);

surfaceTirePairs->setup(MAX_NUM_TIRE_TYPES, MAX_NUM_SURFACE_T
    surfaceMaterials, surfaceTypes);

for(PxU32 i = 0; i < MAX_NUM_SURFACE_TYPES; i++)
{
    for(PxU32 j = 0; j < MAX_NUM_TIRE_TYPES; j++)
    {
        surfaceTirePairs->setTypePairFriction(i,j,gTireFricti
    }
}
return surfaceTirePairs;
}

```

Note: It is not necessary to provide an exhaustive array of all materials. `PxVehicleDrivableSurfaceToTireFrictionPairs` has no knowledge of the materials and assumes a value of zero for the surface type.

There is no upper bound on the friction values used in the PhysX vehicle simulation. The maximum value of friction that obeys the laws of physics is 1.0, but the PhysX SDK purposefully does not enforce this rule. One reason for this is that the current model is far from a complete description of a real vehicle, meaning that some values are taken with friction values to generate the desired behavior. A more complete model would certainly provide greater accuracy given a specific set of vehicle parameters, but it is all clear that it would provide a greater range of editability and control to allow users to have the performance characteristics required for games. Another reason friction is not clamped at 1.0 is that games typically simulate the physics update at a lower frequency, which comes at a cost to numerical accuracy, especially when there are large oscillations in tire effects that require KHz update frequencies. One source of numerical instability is the amplitude of oscillation of the suspension, which is governed in turn by the frequency of the vehicle falling under gravity between each update. At 60Hz update rates, the simulation artifact is acceptably small, but not at 30Hz. The last reason for this is simply no need to impose the strict rules of friction on the vehicles. Some of the most interesting behaviors to be generated that would perhaps be impossible to achieve with a strict friction limit.

by the laws of rigid body and tire dynamics. Having said all this, however model simulated at 60Hz ought to have enough integrity that only small should be necessary. If very large friction values are required, say greater is likely that something is wrong with the update order or perhaps very data has been used.

A `PxVehicleDrivableSurfaceToTireFrictionPairs` instance is passed as a for each call to `PxVehicleUpdates`. Each `PxVehicleDrivableSurfaceToTireFrictionPairs` need only persist for `PxVehicleUpdates`. It is perfectly legal to edit the tire types, materials and between calls to `PxVehicleUpdates`. Editing any of these values while `P` still executing will lead to undefined behavior.

Vehicle Controls

In this Section setting the control values used to drive a vehicle shall be

The simplest and most direct way to set vehicle control values is to function:

```
void PxVehicleDriveDynData::setAnalogInput(const PxReal analogVal
```

One of the difficulties with vehicle dynamics in games is knowing how controller data in a way that results in pleasing handling. Players, accelerate by pressing very quickly on the accelerator trigger in a happen in a real car. This rapid acceleration can have a counter-productive the resulting wheel spin reduces the lateral and longitudinal forces that by the tire. To help overcome some of these problems some options provided to filter the control data from keyboard and gamepad.

A solution to the problem of filtering controller input data is to assign a each button or pad. For analog values under digital control it is possible to or decrease the analog value at a specified rate depending on whether on or off. For analog values under analog control it makes more sense

previous input value to the current input at a specified rate. A slight (simple model is that the difficulty of achieving a large steer angle at large steering rates can be modeled. One technique to achieve this would be to model the forces aligning moments and apply these to a steering linkage model. This is complicated and quite difficult to tune. A simpler solution might be to scale the steer value by another value in range (0,1) that decreases at high steering rates. This method has been implemented in the helper classes and functions.

Rise and fall rates for digital and analog control have been defined in SnippetVehicle4W:

```
PxVehicleKeySmoothingData gKeySmoothingData=
{
    {
        3.0f,    //rise rate eANALOG_INPUT_ACCEL
        3.0f,    //rise rate eANALOG_INPUT_BRAKE
        10.0f,   //rise rate eANALOG_INPUT_HANDBRAKE
        2.5f,    //rise rate eANALOG_INPUT_STEER_LEFT
        2.5f,    //rise rate eANALOG_INPUT_STEER_RIGHT
    },
    {
        5.0f,    //fall rate eANALOG_INPUT__ACCEL
        5.0f,    //fall rate eANALOG_INPUT__BRAKE
        10.0f,   //fall rate eANALOG_INPUT__HANDBRAKE
        5.0f,    //fall rate eANALOG_INPUT_STEER_LEFT
        5.0f    //fall rate eANALOG_INPUT_STEER_RIGHT
    }
};

PxVehiclePadSmoothingData gPadSmoothingData=
{
    {
        6.0f,    //rise rate eANALOG_INPUT_ACCEL
        6.0f,    //rise rate eANALOG_INPUT_BRAKE
        12.0f,   //rise rate eANALOG_INPUT_HANDBRAKE
        2.5f,    //rise rate eANALOG_INPUT_STEER_LEFT
        2.5f,    //rise rate eANALOG_INPUT_STEER_RIGHT
    },
    {
        10.0f,   //fall rate eANALOG_INPUT_ACCEL
        10.0f,   //fall rate eANALOG_INPUT_BRAKE
        12.0f,   //fall rate eANALOG_INPUT_HANDBRAKE
        5.0f,    //fall rate eANALOG_INPUT_STEER_LEFT
    }
};
```

```

    5.0f    //fall rate eANALOG_INPUT_STEER_RIGHT
  }
};

```

A look-up table has also been specified to describe the maximum steering speed:

```

PxFloat32 gSteerVsForwardSpeedData[2*8]=
{
    0.0f,      0.75f,
    5.0f,      0.75f,
    30.0f,     0.125f,
    120.0f,    0.1f,
    PX_MAX_F32, PX_MAX_F32,
    PX_MAX_F32, PX_MAX_F32,
    PX_MAX_F32, PX_MAX_F32,
    PX_MAX_F32, PX_MAX_F32
};
PxFixedSizeLookupTable<8> gSteerVsForwardSpeedTable(gSteerVsForwardSpeedData);

```

Using a `PxVehicleDrive4WRawInputData` instance it is straightforward to set the inputs in the event a keyboard is used:

```

gVehicleInputData.setDigitalAccel(true);
gVehicleInputData.setDigitalBrake(true);
gVehicleInputData.setDigitalHandbrake(true);
gVehicleInputData.setDigitalSteerLeft(true);
gVehicleInputData.setDigitalSteerRight(true);
gVehicleInputData.setGearUp(true);
gVehicleInputData.setGearDown(true);

```

or in the event that a gamepad is used:

```

gVehicleInputData.setAnalogAccel(1.0f);
gVehicleInputData.setAnalogBrake(1.0f);
gVehicleInputData.setAnalogHandbrake(1.0f);
gVehicleInputData.setAnalogSteer(1.0f);
gVehicleInputData.setGearUp(1.0f);
gVehicleInputData.setGearDown(1.0f);

```

Here, `gVehicleInputData` is an instance of the vehicle steering input `PxVehicleDrive4WRawInputData`.

The vehicle SDK offers two optional functions to smooth the keyboard and apply the smoothed input values to the PhysX vehicle. If the vehicle has digital inputs then the following function is used:

```
PxVehicleDrive4WSmoothDigitalRawInputsAndSetAnalogInputs(gKeySmoothed, gSteerVsForwardSpeedTable, carRawInputs, timestep, isInAir, (PxV
```

while gamepad controllers employ the following code:

```
PxVehicleDrive4WSmoothAnalogRawInputsAndSetAnalogInputs(gCarPadSmoothed, gSteerVsForwardSpeedTable, carRawInputs, timestep, (PxVehicleDr
```

The code above smooths the controller inputs and applies them to a vehicle instance. For other vehicle types the process is remarkably similar. There are also complementary classes and functions designed for each vehicle type.

Vehicle Update

It has already been mentioned that vehicles are updated in two stages:

1. specific vehicle code that updates the vehicle internal dynamic forces/torques to apply to the vehicle's rigid body representation
2. an SDK update that accounts for the applied forces/torques as well as with other scene bodies.

In Section [Introduction To Vehicle Update](#) the functions used to perform vehicle updates were introduced. In this Section these separate updates are discussed in more detail.

Raycast and Update Ordering

Prior to the first time that a vehicle is updated in `PxVehicleUpdates`, it performs suspension line raycasts at least once with `PxVehicleSuspension`. In subsequent updates it is not strictly necessary to issue fresh raycasts.

vehicle caches raycast hit planes that can be re-used. It is recommended one-to-one correspondence between raycast completion and update except for the case of vehicles that only require a low level of detail. cars that are far from the camera or where it is known that the vehicle geometry with high spatial coherence. Support for vehicles that require detail is discussed in Section *Level of Detail*.

There is some freedom in the order in which raycasts can be issued relative to dynamics update. In a real-world situation it might be that raycasts on a separate thread at the end of the update loop so that they are ready for the next. However, this really all depends on the threading environment of rigid body updates.

Wheel Pose

`PxVehicleUpdates` poses the wheels shapes of the vehicle's actor to the steer, camber, and rotation angles. The computed pose also attempts to place the geometry exactly on the contact plane identified by the raycast that was used to find the suspension line. To perform this function the PhysX Vehicles SDK needs the shapes of the actor correspond to each wheel of the vehicle. This is done by the function `PxVehicleWheelsSimData::setWheelShapeMapping`.

Note: The vehicle SDK has a default mapping for each wheel that is `PxVehicleWheelsSimData::setWheelShapeMapping(i,i)`. This needs care if the layout of the shapes is different from the default pattern.

Note: `PxVehicleWheelsSimData::setWheelShapeMapping(i,-1)` can be used for setting the local wheel pose. This is particularly useful if a wheel has no actor geometry.

The wheel pose is always within the limits `PxVehicleSuspensionData::mMaxDroop` and `PxVehicleSuspensionData::mMaxCompression`. If the suspension requires the wheel to be placed beyond the compression limit the wheel

the compression limit and a rigid body constraint will handle the differer simulate() call.

Vehicle State Queries

Each vehicle stores persistent simulation data that is up PxVehicleUpdates is called. Examples of persistent data include whe wheel rotation angle, and wheel rotation speed. Additionally, a larg persistent data is computed during each update. This non-persistent d the vehicle's own data structures. Instead, a data buffer is passed to and queried after PxVehicleUpdates completes. Examples of non-pers suspension jounce, tire force and raycast hit actor. The combination types allows an almost complete snapshot of the state of the vehicle a trigger secondary effects such as skid marks, engine and clutch particles.

Persistent wheel data is stored in PxVehicleWheelsDynData, while per data is stored in PxVehicleDriveDynData. The most useful functions are

```
PX_FORCE_INLINE PxReal PxVehicleDriveDynData::getEngineRotationSp  
PxReal PxVehicleWheelsDynData::getWheelRotationSpeed(const PxU32  
PxReal PxVehicleWheelsDynData::getWheelRotationAngle(const PxU32
```

To record non-persistent simulation data so that it may be later be function argument must be passed to PxVehicleUpdates. The follo records non-persistent data for a single 4-wheeled car:

```
PxWheelQueryResult wheelQueryResults[4];  
PxVehicleWheelQueryResult vehicleWheelQueryResults[1] = {{wheelQu  
PxVehicleUpdates(timestep, gravity, frictionPairs, 1, vehicles, v
```

Here, a PxVehicleWheelQueryResult array, whose length equals at le vehicles in the batched vehicles array, is passed to PxVehi PxVehicleWheelQueryResult instance has a pointer to a PxWheelC whose length equals at least the number of wheels in the vehicle. After

is complete the state of each each vehicle wheel may be inspected.

It is not obligatory to record non-persistent data for later query. Indeed, associate a vehicle with a NULL data block to avoid storing non-per: This feature allows memory budgets to be targeted at the vehicles of hig

More Advanced Concepts

Vehicle Telemetry

The purpose of telemetry data is to expose the inner dynamics of the car for tuning through the use of telemetry graphs. In this Section initialization and rendering of telemetry data shall be discussed.

Telemetry data is recorded by calling the following function:

```
void PxVehicleUpdateSingleVehicleAndStoreTelemetryData
(const PxReal timestep, const PxVec3& gravity,
 const PxVehicleDrivableSurfaceToTireFrictionPairs& vehicleDrivableSurfaces,
 PxVehicleWheels* focusVehicle, PxVehicleWheelQueryResult* vehicleWheelQueryResult,
 PxVehicleTelemetryData& telemetryData);
```

The function above is identical to `PxVehicleUpdate` with the exception that it only updates a single vehicle at a time and takes an extra function argument `telemetryData`.

Setting up the telemetry data is relatively straightforward. In addition to the telemetry data streams, the `PxVehicleTelemetryData` structure also stores the size, position, and color scheme of the graph. The following pseudocode initializes and configures telemetry data for a 4-wheeled vehicle:

```
PxVehicleTelemetryData* myTelemetryData = PxVehicleTelemetryData::Create();

const PxVec3 graphSizeX(0.25f);
const PxVec3 graphSizeY(0.25f);
const PxVec3 engineGraphPosX(0.5f);
const PxVec3 engineGraphPosY(0.5f);
const PxVec3 wheelGraphPosX[4]={0.75f, 0.25f, 0.75f, 0.25f};
const PxVec3 wheelGraphPosY[4]={0.75f, 0.75f, 0.25f, 0.25f};
const PxVec3 backgroundColor(255, 255, 255);
const PxVec3 lineColorHigh(255, 0, 0);
const PxVec3 lineColorLow(0, 0, 0);
myTelemetryData->setup
    (graphSizeX, graphSizeY,
     engineGraphPosX, engineGraphPosY,
```

```
wheelGraphPosX, wheelGraphPosY,  
backgroundColor, lineColorHigh, lineColorLow);
```

The sizes, positions, and colors are all values that will be used to render the graph. The exact values of these fields will depend on the coordinate system and how you want to visualize the telemetry data.

In the above example, the coordinates have been configured to render the graph in the center of the screen under the assumption that (1,1) is the top-left corner of the screen and (0,0) the bottom right-hand side of the screen. Screen dimensions have also been specified for rendering data associated with each of the four wheels.

The following enumerated lists detail the telemetry data that is collected

```
enum  
{  
    eCHANNEL_JOUNCE=0,  
    eCHANNEL_SUSPFORCE,  
    eCHANNEL_TIRELOAD,  
    eCHANNEL_NORMALISED_TIRELOAD,  
    eCHANNEL_WHEEL_OMEGA,  
    eCHANNEL_TIRE_FRICTION,  
    eCHANNEL_TIRE_LONG_SLIP,  
    eCHANNEL_NORM_TIRE_LONG_FORCE,  
    eCHANNEL_TIRE_LAT_SLIP,  
    eCHANNEL_NORM_TIRE_LAT_FORCE,  
    eCHANNEL_NORM_TIRE_ALIGNING_MOMENT,  
    eMAX_NUM_WHEEL_CHANNELS  
};  
  
enum  
{  
    eCHANNEL_ENGINE_REVS=0,  
    eCHANNEL_ENGINE_DRIVE_TORQUE,  
    eCHANNEL_CLUTCH_SLIP,  
    eCHANNEL_ACCEL_CONTROL,  
    eCHANNEL_BRAKE_CONTROL,  
    eCHANNEL_HANDBRAKE_CONTROL,  
    eCHANNEL_STEER_CONTROL,  
    eCHANNEL_GEAR_RATIO,  
    eMAX_NUM_ENGINE_CHANNELS  
};
```

Data is collected for suspension jounce, suspension force, tire load, normal wheel rotation speed, tire friction, tire longitudinal slip, tire longitudinal slip, tire lateral force, and tire aligning moment. Data is also collected for engine revs, engine drive torque, clutch slip, applied acceleration/braking, and gear ratio. For each graph all associated data is collected in separate arrays that can be accessed after the update is complete.

Prior to rendering the graph of a particular wheel and channel the following information is required:

```
PxF32 xy[2*PxVehicleGraph::eMAX_NB_SAMPLES];  
PxVec3 color[PxVehicleGraph::eMAX_NB_SAMPLES];  
char title[PxVehicleGraph::eMAX_NB_TITLE_CHARS];  
myTelemetryData->getWheelGraph(wheel).computeGraphChannel(PxVehicleGraph::eChannelType, xy, color, title);
```

This code computes a sequence of screen coordinates $[x_0, y_0, x_1, y_1, x_2, y_2, \dots, x_n, y_n]$ that represent the points of the specified graph channel's data. It also stores a color for each sample by `lineColorHigh` and `lineColorLow` depending on the value of the sample. The `channel` stores the last 256 samples so that a history of each parameter is available on the screen.

The PhysX Vehicles SDK does not render the graphs. This is an application's responsibility because each has its own system for rendering debug information.

Vehicle Update Multi-Threaded

The PhysX Vehicles SDK can be used in a multi-threaded environment to take advantage of performance improvements arising from parallelism. The update step is performed exactly as described in Section [Vehicle Update](#) but with an extra step: a function `PxVehiclePostUpdates` after all concurrent calls to `PxVehicleUpdate` and `PxVehicleUpdates` are complete. `PxVehiclePostUpdates` performs operations that are normally executed in `PxVehicleUpdates` but which are not possible to call when concurrency is employed.

PxVehicleSuspensionRaycasts is a thread-safe function and can be used without any modifications to the calling code with the exception, of course, of managing the tasks and threads that will execute the raycasts concurrently. PxVehicleUpdates as used in Section *Vehicle Update* is not concurrent and requires an extra PxVehicleConcurrentUpdateData array to be specified to be executed concurrently. When this extra data is specified PxVehicleUpdates writes to PhysX actors that are involved in the vehicle update and writes are stored in the PxVehicleConcurrentUpdateData array during PxVehicleUpdates and then executed sequentially in PxVehiclePostUpdate.

Sample code can be found in SnippetVehicleMultiThreading.

Tire Shaders

It is possible to replace the default tire model used by PhysX vehicles with a custom one. This requires a shader function that can be set per-vehicle along with a PxVehicleConcurrentUpdateData array that must be set per-wheel:

```
void PxVehicleWheelsDynData::setTireForceShaderFunction
    (PxVehicleComputeTireForce tireForceShaderFn)
void PxVehicleWheelsDynData::setTireForceShaderData
    (const PxU32 tireId, const void* tireForceShaderData)
```

The shader function must implement this function prototype:

```
typedef void (*PxVehicleComputeTireForce)
(const void* shaderData,
 const PxF32 tireFriction,
 const PxF32 longSlip, const PxF32 latSlip, const PxF32 camber,
 const PxF32 wheelOmega, const PxF32 wheelRadius, const PxF32 rec
 const PxF32 restTireLoad, const PxF32 normalisedTireLoad, const
 const PxF32 gravity, const PxF32 recipGravity,
 PxF32& wheelTorque, PxF32& tireLongForceMag, PxF32& tireLatForce
```

The vehicle update code will call the shader function for each wheel wheel for that wheel.

Vehicle Types

The PhysX Vehicle SDK supports four types of vehicle: `PxVehicleDriveNW`, `PxVehicleDriveTank` and `PxVehicleNoDrive`. `PxVehicleDrive4W` will be the best choice for rally cars, street cars. `PxVehicleDriveNW` is very similar to `PxVehicleDrive4W` except that it that it allows all wheels to be coupled to the differential. This general differential models of `PxVehicleDriveNW` cannot match the range or `PxVehicleDrive4W`. `PxVehicleDriveTank` implements a simple but effective constraining the left and right wheel speeds to mimic the effect of a tank. `PxVehicleNoDrive` implements a vehicle that is simply a rigid body with wheels and tires. The idea here is to allow custom drive models such as a hovercraft to be implemented using PhysX vehicles.

`PxVehicleDrive4W`

The class `PxVehicleDrive4W` has already been discussed in some discussion so far has focused on 4-wheeled vehicles. In the `PxVehicleDrive4W` shall be discussed with special reference to instancing and more than 4 wheels.

3-Wheeled Cars

Utility functions have been provided to quickly configure 3-wheeled cars to start with a 4-wheeled car and then disable one of the wheels:

```
void PxVehicle4WEnable3WTadpoleMode(PxVehicleWheelsSimData& wheels,
    PxVehicleWheelsDynData& wheelsDynData, PxVehicleDriveSimData4W& driveData)
void PxVehicle4WEnable3WDeltaMode(PxVehicleWheelsSimData& wheels,
    PxVehicleWheelsDynData& wheelsDynData, PxVehicleDriveSimData4W& driveData)
```

These functions ensure that no raycast hits are returned for the disabled wheel and additionally do some other work to decouple the disabled wheel from the rest of the vehicle, disable ackermann correction, re-position the opposite remaining wheel to the axle, and adjust the suspension of the opposite remaining wheel to

missing suspension of the disabled wheel. Further wheels could in the custom code to create a vehicle with 1 or 2 effective wheels. At that point balancing code would be required to prevent the vehicle falling over.

Some care must be taken when removing a wheel because the function has a number of requirements that must be satisfied for all requirement is that any wheel that has been disabled must not be PxShape. This is a safety feature that prevents PxVehicleUpdates at local pose of a PxShape that may no longer be valid. PxVehicleWheelsSimData::setWheelShapeMapping can be used requirement. The second requirement is that any wheel that has been zero wheel rotation speed. This can be satisfied PxVehicleWheelsDynData::setWheelRotationSpeed for the relevant requirement is that disabled wheels must receive no drive torque requirement can actually be ignored because it is automatically enforced code called by the PxVehicleUpdates function. For vehicles of type Px requirement on drive torque is fulfilled by ensuring that PxVehicleNoDrive is never called with a non-zero torque value. Further, the drive torque requirement is readily fulfilled for vehicles of type PxVehicleDriveNW by ensuring that disconnected from the disabled wheel. This is achieved using PxVehicleDifferentialNWDData::setDrivenWheel.

Configuring the differential of a PxVehicle4W to ensure that no drive torque to a disabled wheel is a little more complex because there are many ways to achieve this. If the wheel is not a driven wheel then disabling the wheel satisfies the drive torque requirement because such wheels can never be connected to the differential. On the other hand, if the wheel has index eFRONT_LEFT or eREAR_LEFT or eREAR_RIGHT then the differential does need to enforce the requirement. One way to do this is to set up the differential to only the rear(front) wheels if a front(rear) wheel has been disabled. This is readily implemented by selecting front-wheel drive mode or rear-wheel drive mode as appropriate:

```
PxVehicleDifferential4WData diff = myVehicle.getDiffData();  
if(PxVehicleDrive4WheelOrder::eFRONT_LEFT == wheelToDisable ||
```

```

PxVehicleDrive4WWheelOrder::eFRONT_RIGHT == wheelToDisable)
{
    if(PxVehicleDifferential4WData::eDIFF_TYPE_LS_4WD == diff.mType
        PxVehicleDifferential4WData::eDIFF_TYPE_LS_FRONTWD == diff.mType
        PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_4WD == diff.mType
        PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_FRONTWD == diff.mType)
    {
        diff.mBias = 1.3f;
        diff.mRearLeftRightSplit = 0.5f;
        diff.mType = PxVehicleDifferential4WData::eDIFF_TYPE_LS_FRONTWD;
        //could also be PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_FRONTWD
    }
}
else if(PxVehicleDrive4WWheelOrder::eREAR_LEFT == wheelToDisable
        PxVehicleDrive4WWheelOrder::eREAR_RIGHT == wheelToDisable)
{
    if(PxVehicleDifferential4WData::eDIFF_TYPE_LS_4WD == diff.mType
        PxVehicleDifferential4WData::eDIFF_TYPE_LS_REARWD == diff.mType
        PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_4WD == diff.mType
        PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_REARWD == diff.mType)
    {
        diff.mBias = 1.3f;
        diff.mFrontLeftRightSplit = 0.5f;
        diff.mType = PxVehicleDifferential4WData::eDIFF_TYPE_LS_REARWD;
        //could also be PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_REARWD
    }
}
myVehicle.setDiffData(diff);

```

In some situations limiting the drive torque to just the front or rear wheel is acceptable. If only a single wheel has been disabled then it is possible to have a mode where 3 wheels are driven. This can be achieved by modifying the logic so that torque is only delivered to 3 wheels:

```

PxVehicleDifferential4WData diff = myVehicle.getDiffData();
if(PxVehicleDrive4WWheelOrder::eFRONT_LEFT == wheelToDisable ||
    PxVehicleDrive4WWheelOrder::eFRONT_RIGHT == wheelToDisable)
{
    if(PxVehicleDifferential4WData::eDIFF_TYPE_LS_4WD == diff.mType
        PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_4WD == diff.mType)
    {
        if(PxVehicleDrive4WWheelOrder::eFRONT_LEFT == wheelToDisable)
        {

```

```

        diff.mFrontLeftRightSplit = 0.0f;
    }
    else
    {
        diff.mFrontLeftRightSplit = 1.0f;
    }
}
else if(PxVehicleDrive4WheelOrder::eREAR_LEFT == wheelToDisable
        PxVehicleDrive4WheelOrder::eREAR_RIGHT == wheelToDisable)
{
    if(PxVehicleDifferential4WData::eDIFF_TYPE_LS_4WD == diff.mTy
        PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_4WD == diff.m
    {
        if(PxVehicleDrive4WheelOrder::eREAR_LEFT == wheelToDis
        {
            diff.mRearLeftRightSplit = 0.0f;
        }
        else
        {
            diff.mRearLeftRightSplit = 1.0f;
        }
    }
}
myVehicle.setDiffData(diff);

```

In some situations it will make sense to disable Ackermann steer correction if a wheel was able to steer. In particular, if the remaining wheel of the front axle is positioned so that it is at the center of the axle then it would almost completely cancel out the Ackermann correction. This can be achieved by setting the accuracy to zero (`PxVehicleAckermannGeometryData::mAccuracy`). The role of Ackermann correction, however, really needs to be determined on a case by case basis.

N-Wheeled Cars

In addition to removing wheels from a vehicle, it is also possible to use `PxVehicleDrive4W` with more than 4 wheels but with the caveat that only the first 4 wheels are driven. As a consequence of this caveat the functionality of the extra wheels is limited compared to the first 4 wheels. More specifically, only the first 4 wheels are connected to the differential or the steering; that is, only the first block of 4 wheels experience a drive torque or a steer angle and only the first block of 4 wheels

the Ackermann steering correction. As a consequence, the extra wheel role to the rear wheels of a 4-wheeled car that has front-wheel drive or a 4-wheeled car that has rear-wheel drive. Adding extra wheels does ability to call `PxVehicle4WEnable3WTadpoleMode` or `PxVehicle4WEn` These functions, however, are hard-coded to disable one of the 4 wheels connected to the steering and driven through the differential.

The following pseudo-code illustrates the key steps in the creation of a `PxVehicleDrive4W` vehicle:

```
PxVehicleWheelsSimData* wheelsSimData=PxVehicleWheelsSimData::all
PxVehicleDriveSimData4W driveSimData;
setupSimData(wheelsSimData,driveSimData);
PxVehicleDrive4W* car = PxVehicleDrive4W::allocate(6);
PxRigidDynamic* vehActor=createVehicleActor6W();
car->setup(&physics,vehActor,*wheelsSimData,driveSimData,2);
```

PxVehicleDriveNW

While the `PxVehicleDrive4W` allows cars with any number of wheels simulated it only allows 4 of those wheels to be driven by engine through a differential. The vehicle type `PxVehicleDriveNW` has been introduced to overcome this limitation. This vehicle class makes use of the differential type `PxVehicleDriveNW` class that allows any or all of the vehicle's wheels to be coupled to the differential. The limitation that the torque available at the differential is always divided equally among the wheels that are coupled to the differential. The generality of `PxVehicleDriveNW` allows advanced features such as limited slip differentials and Ackermann steering correction, meaning that only a simple equal-split differential model can be provided.

The following pseudo-code illustrates the key steps in the creation of a `PxVehicleDriveNW` vehicle:

```
PxVehicleWheelsSimData* wheelsSimData=PxVehicleWheelsSimData::all
PxVehicleDriveSimDataNW driveSimData;
setupSimData(wheelsSimData,driveSimData);
PxVehicleDriveNW* car = PxVehicleDriveNW::allocate(6);
PxRigidDynamic* vehActor=createVehicleActorNW();
car->setup(&physics,vehActor,*wheelsSimData,driveSimData,6);
```

PxVehicleDriveTank

The PhysX vehicle SDK also supports tanks through the use of the `PxVehicleDriveTank` class. Tanks are different to multi-wheeled vehicles in that the wheels are driven through the differential in a way that ensures that all the wheels on the same side have the same speed, and all the wheels on the right-hand have the same speed. This constraint on wheel speed mimics the effect of the caterpillar track system, without the expense of simulating the jointed track structure. Adding the geometry for tank tracks is as easy as adding an actor shape down each side and setting the appropriate query filters as appropriate for the tracks. The motion of the caterpillar track is rendered with a scrolling texture, safe in the knowledge that all wheels have the same speed, just as though they were properly constrained by the track rotation.

Creating a `PxVehicleDriveTank` instance is very similar to creating a `PxVehicleDrive4W` instance with the exception that tanks have no concept of extra wheels connected to the differential: all tank wheels are driven. The following code shows how to set up a 12-wheeled tank:

```
PxVehicleWheelsSimData* wheelsSimData = PxVehicleWheelsSimData::allocate(12);
PxVehicleDriveSimData4W driveSimData;
setupTankSimData(wheelsSimData, driveSimData);
PxVehicleDriveTank* tank = PxVehicleDriveTank::allocate(12);
PxRigidDynamic* vehActor=createVehicleActor12W();
tank->setup(&physics, vehActor, *wheelsSimData, tankDriveSimData, 12)
```

Controlling a tank is quite different to controlling a car because tanks have a different steering mechanism: the turning action of a tank arises from the difference in left and right wheel speeds, while cars turn by the action of a steering wheel. This requires a set of helper classes and functions to smooth the control inputs:

1. `PxVehicleDriveTankRawInputData`
2. `PxVehicleDriveTankSmoothDigitalRawInputsAndSetAnalogInputs`
3. `PxVehicleDriveTankSmoothAnalogRawInputsAndSetAnalogInputs`

PhysX tanks currently support two drive models: `eSTANDARD` and `eS`

model eSPECIAL allows the tank tracks to rotate in different eSTANDARD does not. These two modes result in quite different turning actions. In model eSTANDARD simulates the usual turning action of a tank: pushing left(right) stick drives the left(right) wheels forward, while pulling back or applying the brake to the right(left) wheels. eSPECIAL, on the other hand, simulates an exotic turning action where pushing back on the right(left) stick drives the wheels backwards. This can result in a turning circle focused at the center of the tank. The smallest possible turning circle of a tank in eSTANDARD will have a focus on one of the caterpillar tracks, depending on whether the tank is turning left or right.

PxVehicleNoDrive

The class PxVehicleNoDrive has been introduced to provide a close backwards compatibility with the interface to the 2.8.x NxWheels. It is essentially a rigid body with N suspension/wheel/tire units attached, identical to that of a PxVehicleDrive4W which is permanently in neutral. The engine has no influence on the wheels and the wheels are coupled to the motion of the rigid body. This comes, of course, without the steering Ackermann steering correction data, engine torque curve data etc. The user can develop their own drive model on top of already existing vehicle control suspension raycasts, tire and suspension force computation, and Physics.

The key functions are the application of per wheel drive and brake torque and steer angles:

```
/**
 * \brief Set the brake torque to be applied to a specific wheel
 */
void setBrakeTorque(const PxU32 id, const PxReal brakeTorque);

/**
 * \brief Set the drive torque to be applied to a specific wheel
 */
void setDriveTorque(const PxU32 id, const PxReal driveTorque);

/**
 * \brief Set the steer angle to be applied to a specific wheel
 */
```

```
void setSteerAngle(const PxU32 id, const PxReal steerAngle);
```

SI Units

The discussion so far has assumed that distance is measured in metres, mass is measured in kilograms, and that time is measured in seconds. Further, the values of all relevant vehicle components have been set under the assumption that SI units will be adopted. An example of such a default parameter is the maximum wheel torque. Inspection of the constructor for `PxVehicleWheelData` reveals a parameter `mMaxBrakeTorque`. This number actually represents a value of 1500 "Newton-Squared Per Second-Squared" (an alternative way of expressing this is "Metres"). An important question is how to set up a vehicle with measurements if SI units are not adopted. The purpose of this Section is to illustrate the problem, in particular, the case where distance is measured in centimeters rather than metres, used as an example. This particular deviation from the adoption of SI Units is the most common one in game development, arising from the units of distance used in a 3D modeling package.

Vehicle parameters whose value is dependent on the length scale fall into two categories: those that can theoretically be measured with a ruler and those with values that are not involving combinations of other properties such as mass or time constant. The former category includes data fields such as wheel radius, wheel suspension droop, while the latter category includes data fields such as wheel torque or wheel moment of inertia.

The following is an exhaustive list of vehicle parameters that can be measured solely from vehicle geometry:

```
PxVehicleChassisData::mCMOffset  
PxVehicleAckermannGeometryData::mFrontWidth  
PxVehicleAckermannGeometryData::mRearWidth  
PxVehicleAckermannGeometryData::mAxeSeparation  
PxVehicleWheelData::mRadius
```

```
PxVehicleWheelData::mWidth  
PxVehicleSuspensionData::mMaxCompression  
PxVehicleSuspensionData::mMaxDroop  
PxVehicleWheelsSimData::setSuspForceAppPointOffset  
PxVehicleWheelsSimData::setTireForceAppPointOffset  
PxVehicleWheelsSimData::setWheelCentreOffset
```

It is useful to note that all the above parameters have default values independent of length scale they must always be set with measurements corresponding length scale if a legal vehicle is to be successfully instantiated.

Setting parameters that involve more complex combinations of length scales require more thought than those featured in the list above. A simple rule of thumb is that a parameter that has units linear with distance must be scaled by the number of length units that is equivalent to 1 meter, while any parameter that has units involving distance squared must be scaled by the square of the number of length units that is equivalent to 1 meter. A wheel braking torque of 1500 kilograms metres-squared per second squared, for example, is equivalent to $1500 \times 100 \times 100$ kilograms centimeters-squared per second squared. Consequently, when centimeters is used as the length scale a good initial value for wheel braking torque is 15000000 [kilograms centimeters-squared per second squared]. If inches are used as the length scale then a good initial value for wheel braking torque would be $1500 \times 39.37 \times 39.37$ (= 2324995.35) [kilograms inches-squared per second squared].

Each non-dimensionless parameter has been described with the corresponding units in PxVehicleComponents.h. The following is an exhaustive list of vehicle parameters and their indirect expressions of distance scale:

```
PxVehicleEngineData::mMOI (kg m^2)  
PxVehicleEngineData::mPeakTorque (kg m^2 s^-2)  
PxVehicleEngineData::mDampingRateFullThrottle (kg m^2 s^-1)
```

```
PxVehicleEngineData::mDampingRateZeroThrottleClutchEngaged (kg m^2 s^-1)
PxVehicleEngineData::mDampingRateZeroThrottleClutchDisengaged (kg m^2 s^-1)
PxVehicleClutchData::mStrength (kg m^2 s^-1)
PxVehicleWheelData::mDampingRate (kg m^2 s^-1)
PxVehicleWheelData::mMaxBrakeTorque (kg m^2 s^-2)
PxVehicleWheelData::mMaxHandBrakeTorque (kg m^2 s^-2)
PxVehicleWheelData::mMOI (kg m^2)
PxVehicleChassisData::mMOI (kg m^2)
```

All but the last three of the above parameters have non-zero initial values associated with their constructors. This means that a good guess for their initial values can be made by multiplying the value expressed in SI Units with either the number of length units equivalent to 1 meter or the square of the number of length units that meter.

It is important to note that the wheel handbrake torque has a default value that is not zero. Not all wheels respond to the handbrake torque. A good guess for the handbrake torque is simply the value of the wheel braking torque, perhaps multiplied by between 0.5 and 1.0 to ensure that the handbrake is stronger than the brake.

The wheel moment of inertia and chassis moment of inertia are typically proportional to the wheel radius and chassis dimensions so naturally reflect the length scales in the simulation. If values are taken from manufacturer data it is important to ensure the units of the manufacturer data are commensurate with the remaining data fields or to perform the appropriate unit conversion.

A number of functions also have parameters that are functions of time. The following is an exhaustive list of such functions:

```
PxVehicleWheelsSimData::setSubStepCount
PxVehicleWheelsSimData::setMinLongSlipDenominator
```

PxVehicleSetMaxHitActorAcceleration

Some care is required to set the threshold speed in PxVehicleWheels: Here, it is the case that the default threshold speed is 5.0 metres per second. If centimeters is the chosen length scale a value of 500 [centimeters per second] is required to achieve the equivalent behavior, or with inches as the chosen length scale a value of 5×39.37 (= 196.85) [inches per second] is required. The same logic can be applied to PxVehicleWheelsSimData::setMinLongSlipDenominator. If the default is 4.0 metres per second. If centimeters is the adopted scale then the equivalent value is 400 [centimeters per second], while 4×39.37 (=157.48) [inches per second] is required if inches is the chosen scale. PxVehicleSetMaxHitActorAcceleration takes a value that scales linearly with the length scale. If the desired maximum acceleration is 1000 metres per second per second then that would be scaled to 100000 centimetres per second per second in centimetres scale. With inches as the length scale the equivalent value would be 1000×39.37 inches per second per second.

The PhysX Vehicle SDK supports any system of units with the caveat that all values supplied must conform to the same unit system. Further, the default data is strictly expressed in the SI unit system, can be used as a guide to choose values in any unit system for almost any conceivable vehicle. A quick way to be able to decide if, say, a truck would have a stronger handbrake than a family car. Now, the default data approximates that of a standard family car with a good estimate to start with the truck having a handbrake that is perhaps 10 times that is, 5000 kilograms metres-squared per second-squared. If centimeters is the chosen length scale then a quick conversion can be performed by noting that 100 centimeters, leading to the brake torque being set as 5000×100 centimeters-squared per second-squared. If the natural unit of mass is grams, noting that 1 kilogram is 1000 grams leads to an equivalent value of 5000000 grams-squared per second-squared. This rule can be repeated for all other units by simply noting the default value and the SI units in the relevant clause and then performing the conversion to the chosen unit system.

The PhysX Vehicle SDK depends on a number of threshold values that are scaled to the length scale. These are set with the function PxInitVehicleSD

PxTolerancesScale values that have already been already configured for
If PxInitVehicleSDK is not called prior to the first call to PxVehicleUpdate
be passed to the PhysX error stream.

Level of Detail

It seems sensible to attempt to save valuable clock cycles for vehicles visible on the screen or are sufficiently far from the camera that its motion is exactly in step with the world geometry. The PhysX vehicle number of options for reducing the computational load for vehicles through levels of detail.

Extrapolation

The most obvious strategy for a vehicle that requires only a low level of stop performing raycasts (PxVehicleSuspensionRaycasts) (PxVehicleUpdates) for that vehicle. Instead of computing the ground vehicle's tires and computing the suspension and tire forces each and every frame, it may be acceptable to avoid these steps completely and let the PhysX SDK rigid body with the legacy momentum of the rigid body. After several frames, wheels will likely either be hovering above the ground or intersecting the ground. A strategy needs to be a strategy to decide how many PhysX SDK updates a vehicle is once more updated properly by including it in the vehicle PxVehicleSuspensionRaycasts/PxVehicleUpdates. The details of any extrapolation are left to users of the vehicles SDK because it depends on a number of factors: the distance from the camera; the spatial coherence of the world geometry; the speed of the vehicle; and whether the audio or graphics fix for this is an important role.

Disable Wheels

If there exist vehicles with large wheel counts it might also be possible to have a number of wheels that participate in the simulation. This can be done by calling PxVehicleWheelsSimData::disableWheel. An example might be a truck

Now, such a truck will clearly need to perform 18 raycasts, 18 tire force updates, and 18 updates of wheel rotation speed in order to complete the vehicle update. If the number of enabled wheels can be reduced to just 4 enabled wheels then it is clear that less computation is required. It is important to note that when wheels are disabled they no longer support the mass of the vehicle's rigid body. In the extreme case of a truck reduced to just 4 active wheels this will mean that the remaining 14 springs are only configured to support approximately 4/18 of the mass of the rigid body. To remedy this problem the mass of the rigid body will need to be distributed among the enabled wheels and suspensions, `PxVehicleComputeSprungMasses`. A more complete description of the disabled wheels can be found in Section [3-Wheeled Cars](#).

Swapping Multiple Vehicle Versions

Instead of disabling wheels, perhaps a simpler and more effective way to reduce computational cost is to instantiate two versions of the vehicle with different numbers of wheels. The two vehicles can be easily swapped in the vehicles `PxVehicleSuspensionRaycasts/PxVehicleUpdates` as the required level of detail increases and decreases. It is worth considering how this might work in the case of the truck mentioned earlier. The simplest strategy would be to first construct a rigid body and attach a `PxShape` instance for each of the 18 wheels of the 18-wheeled truck. Instantiating the required 18-wheeled version of the truck with `PxVehicleNW::setup` will automatically pose the shapes of all 18 wheels. The next step is to choose 4 of the 18 wheels to form the 4-wheeled version of the truck. Many choices are available but the most obvious choice would be the front-left/front-right/rear-left/rear-right wheels of the 18-wheeled truck. The 4-wheeled version can be instantiated using the same rigid body as for the 18-wheeled version. The rest poses of the `PxShape` instances to the rest pose of the 4-wheeled truck. If the 18-wheeled version have been set up correctly the rest poses ought to be the same as their counterparts in the 18-wheeled version. A key point to note is that both versions of the vehicle apply forces to the same rigid body. Another key point to note is that when the 4-wheeled vehicle is chosen only 4 of the 18 `PxShape` instances will be updated, leaving 14 `PxShape` instances at either the rest local pose or the rest pose given to them when the 18-wheeled version was last used. In terms of

these unposed shapes are the main disadvantage of the lower differences in handling are much harder to gauge.

A number of useful functions are available to make it easy to swap between versions of the same vehicle:

```
void PxVehicleComputeSprungMasses(const PxU32 nbSprungMasses,
    const PxVec3* sprungMassCoordinates, const PxVec3& centreOfMass,
    const PxU32 gravityDirection, PxReal* sprungMasses);
void PxVehicleWheelsSimData::copy(const PxVehicleWheelsSimData& src,
    const PxU32 trgWheel);
void PxVehicleSuspensionData::setMassAndPreserveNaturalFrequency(
    const PxU32 wheelIndex, PxReal mass, PxReal frequency);
void PxVehicleCopyDynamicsData(const PxVehicleCopyDynamicsMap& src,
    const PxVehicleWheels& src, PxVehicleWheels* trg);
```

The following pseudo-code hopefully makes clear how to apply these functions. We first construct the lower LOD vehicle and then swap between the different versions.

```
PxVehicleDriveNW* instantiate4WVersion(const PxVehicleDriveNW& ve
{
    //Compute the sprung masses of the 4-wheeled version.
    PxReal sprungMasses[4];
    {
        const PxReal rigidBodyMass = vehicle18W.getRigidDynamicAc
        const PxVec3 wheelCoords[4] =
        {
            vehicle18W.mWheelsSimData.getWheelCentreOffset(0),
            vehicle18W.mWheelsSimData.getWheelCentreOffset(1),
            vehicle18W.mWheelsSimData.getWheelCentreOffset(2),
            vehicle18W.mWheelsSimData.getWheelCentreOffset(3)
        };
        const PxU32 upDirection = 1;
        PxVehicleComputeSprungMasses(4, wheelCoords, PxVec3(0,0,0),
            sprungMasses);
    }

    //Set up the wheels simulation data.
    PxVehicleWheelsSimData* wheelsSimData4W = PxVehicleWheelsSimD
    for(PxU32 i = 0; i < 4; i++)
    {
        wheelsSimData4W->copy(vehicle18W.mWheelsSimData, i, i);

        PxVehicleSuspensionData suspData = wheelsSimData4W->getSu
```

```

        suspData.setMassAndPreserveNaturalFrequency(sprungMasses[
        wheelsSimData4W->setSuspensionData(i, suspData);
    }
wheelsSimData4W->setTireLoadFilterData(vehicle18W.mWheelsSimD

//Make sure the correct shapes are posed.
wheelsSimData4W->setWheelShapeMapping(0,0);
wheelsSimData4W->setWheelShapeMapping(1,1);
wheelsSimData4W->setWheelShapeMapping(2,2);
wheelsSimData4W->setWheelShapeMapping(3,3);

//Set up the drive simulation data.
PxVehicleDriveSimDataNW driveSimData4W = vehicle18W.mDriveSim
PxVehicleDifferentialNWData diff4W;
diff4W.setDrivenWheel(0, true);
diff4W.setDrivenWheel(1, true);
diff4W.setDrivenWheel(2, true);
diff4W.setDrivenWheel(3, true);
driveSimData4W.setDiffData(diff4W);

//Instantiate the 4-wheeled version.
PxRigidDynamic* rigidDynamic =
    const_cast<PxRigidDynamic*>(vehicle18W.getRigidDynamicAct
PxVehicleDriveNW* vehicle4W =
    PxVehicleDriveNW::create(&physics, rigidDynamic, *wheelsS

//Delete the wheels simulation data now that we have copied t
//vehicle.
wheelsSimData4W->free();

//Finished.
return vehicle4W;
}

void swapToLowLodVersion(const PxVehicleDriveNW& vehicle18W, PxVe
PxVehicleWheels** vehicles, PxU32 vehicleId)
{
    vehicles[vehicleId] = vehicle4W;

    PxVehicleCopyDynamicsMap wheelMap;
    wheelMap.sourceWheelIds[0]=0;
    wheelMap.sourceWheelIds[1]=1;
    wheelMap.sourceWheelIds[2]=2;
    wheelMap.sourceWheelIds[3]=3;
    wheelMap.targetWheelIds[0]=0;
    wheelMap.targetWheelIds[1]=1;
    wheelMap.targetWheelIds[2]=2;

```

```

wheelMap.targetWheelIds[3]=3;

PxVehicleCopyDynamicsData(wheelMap, vehicle18W, vehicle4W);
}

void swapToHighLowVersion(const PxVehicleDriveNW& vehicle4W, PxVe
PxVehicleWheels** vehicles, PxU32 vehicleId)
{
    vehicles[vehicleId] = vehicle18W;

    PxVehicleCopyDynamicsMap wheelMap;
    wheelMap.sourceWheelIds[0]=0;
    wheelMap.sourceWheelIds[1]=1;
    wheelMap.sourceWheelIds[2]=2;
    wheelMap.sourceWheelIds[3]=3;
    wheelMap.targetWheelIds[0]=0;
    wheelMap.targetWheelIds[1]=1;
    wheelMap.targetWheelIds[2]=2;
    wheelMap.targetWheelIds[3]=3;

    PxVehicleCopyDynamicsData(wheelMap, vehicle4W, vehicle18W);
}

```

Disable Raycasts

In some scenes it might be possible not to issue raycasts for each vehicle update. Depending on the geometry, this can lead to significant gains.

The PhysX vehicles SDK provides a simple mechanism to disable or enable raycasts per update and per vehicle by specifying an array of booleans as a full array of `PxVehicleSuspensionRaycasts`. An alternative to disabling raycasts would be to alter the array of vehicles passed to `PxVehicleSuspensionRaycasts` that some vehicles scheduled for update in `PxVehicleUpdates` do not perform a batched raycast prior to the update. It is anticipated that using the boolean array to be passed to both the raycast and update functions will allow for simpler vehicle management.

Vehicles that participate in the batched raycast automatically store raycast results which are re-used each subsequent update until they are replaced by the next raycast. This means that it is not necessary to perform raycasts for

especially if the vehicle is moving slowly or the vehicle is far from the ground. If a vehicle remains on the same plane for several updates in a row. As the number of updates preceded by a raycast decreases, the accuracy of the cached hit planes decreases, meaning that the likelihood of visibly poor wheel placement or intersecting the ground if raycasts are not performed prior to each update. It is up to users of the SDK to develop their own strategy to decide whether to perform a fresh raycast or not.

If a raycast is not performed prior to an update then the vehicle will only have a partial description of its interaction with the scene. For example, as objects are deleted the actor or shape or material hit by the last suspension raycast may no longer exist in the scene several updates later. For this reason, the vehicle reports the shapes/actors/materials if a cached plane is used instead of a fresh raycast. The documentation for `PxWheelQueryResult` describes this in more detail.

The first update of any vehicle requires that a raycast is performed prior to the first update. If a raycast is not performed prior to the first update then the vehicle will not have an opportunity to cache its raycast hit planes. Further, after each call to `update` the vehicle also needs to perform a raycast prior to the next update. The `setToRestState` method clears the cached hit planes, meaning that the next update will need to compute them once more.

Use The Clutch in Estimate Mode

The vehicle SDK implements a mathematical model for the clutch that has two modes of operational accuracy: `eESTIMATE` and `eBEST_POSSIBLE`. If `eBEST_POSSIBLE` is chosen the SDK attempts to accurately update rotation speeds from their coupling through the clutch. It is worth noting that the clutch model in `PxVehicleDriveTank` reduces to a particularly simple set of equations that have a fast analytical solution. As a consequence, the vehicle SDK uses the `eESTIMATE` accuracy model for tanks and instead always opts to compute the best possible solution. In the case of `PxVehicle4W` only marginal performance gains can be achieved by switching to `eESTIMATE` because at most only 4 wheels can ever be in clutch. The real performance gains from the estimated solution are seen in the `PxVehicleDriveTank` model.

PxVehicleNW instances with high wheel count.

If `eESTIMATE` is chosen the quality of the estimate can be tuned by `PxVehicleClutchData::mEstimateIterations`. As the value of this variable increases the computational cost also increases and the estimated solution approaches the best possible solution. At particularly large values of `mEstimateIterations` the estimated solution might even exceed that of the best possible solution in terms of precision. On the other hand, particularly low values such as 1 might result in an inaccurate coupling between the engine and wheels. This can be particularly noticeable after a gear change or at standing starts or when the brakes are applied. In such situations large angular velocity differences at the clutch result in torque spikes that require computational effort to resolve. A poor estimate might, for example, result in oscillating engine rotation speeds after a gear change instead of the smooth transitions. The magnitude of accuracy loss and its subsequent effect on vehicle behavior are very difficult to quantify and really need to be tested for each vehicle and engine.

It is recommended that `eBEST_POSSIBLE` is chosen for vehicles that require a high level of detail and that `eESTIMATE` is only chosen for vehicles that require low detail. Care must be taken when tuning `PxVehicleClutchData::mEstimateIterations` as the loss of accuracy is acceptable for the required level of detail. In many cases a possible value of 1 will turn out to provide perfectly acceptable. Smooth and believable behavior, however, is only guaranteed if `eBEST_POSSIBLE` is chosen.

Wheel Contact Beyond Raycasts

This Section describes the steps required to simulate wheel volumes, contact sweeps and contact modification. Sample code can be found in `SnippetVehicleContactMod`.

Section [Algorithm](#) described how scene query raycasts are used to calculate suspension forces. Expanding on this theme, Section [Filtering](#) describes scene query and simulation filtering to categorise scene shapes as either drivable surfaces: drivable surfaces interact only with suspension raycasts, non-drivable surfaces interact with wheels only through rigid body contact.

A variety of issues arise from the the system of raycasts and filtering. One problem is that it may be impractical to author every shape in t either drivable or non-drivable: it is easy to imagine a landscape moc mesh that is partially drivable and partially non-drivable. Another probl ignore the extent of the wheel in the lateral and longitudinal directions. Figures 2a and 2b.

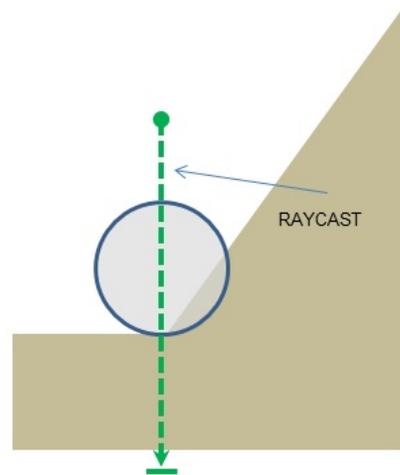


Figure 2a: The raycast ignores the overlap of the wheel's volume with plane.

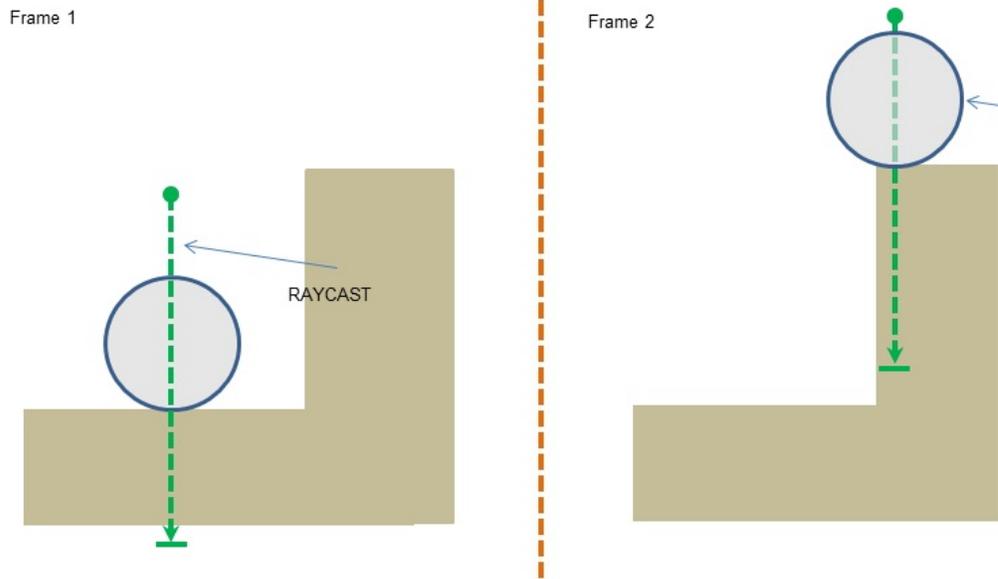


Figure 2b: The wheel rolls towards a wall in Frame 1 and is immediately elevated on a surface in Frame 2.

The problem illustrated in Figure 2a can be solved by replacing raycast. Instead of performing a raycast along the suspension direction through the wheel, the shape representing the wheel is swept from its compressed state to its state at maximum elongation. Sweeping a volume in a scene means that all possible contact planes are considered. This is illustrated in Figure 2c.

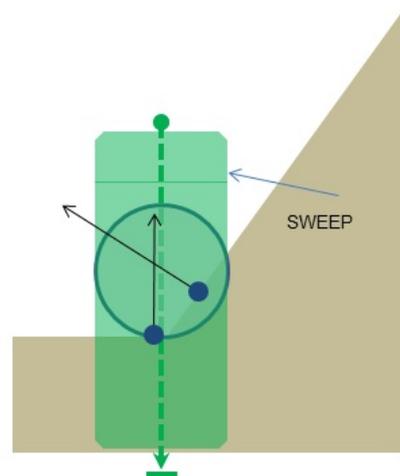


Figure 3: Sweeps pick up all contact planes under the wheel

In Figure 3, it is easy to see that there are multiple contact points and with a different normal. A decision needs to be made about which one to accept as the driving surface and which to ignore. In some scenarios it takes the first contact encountered by the sweep and ignore all others. It is recommended to issue a blocking sweep. PhysX supports two types of blocking and non-blocking. A detailed description of blocking and non-blocking can be found in Section [Filtering](#). In summary, however, a blocking sweep will ignore any contact encountered by the swept volume, while non-blocking sweeps will accept all contacts encountered by the sweep. The scenario in Figure 3 suggests that a non-blocking sweep would be sufficient because it will return the inclined plane rather than the horizontal plane. As a consequence, the vehicle will start to drive on the inclined plane. Some of the scenarios depicted in Figure 2b, are more complex and require a non-blocking sweep.

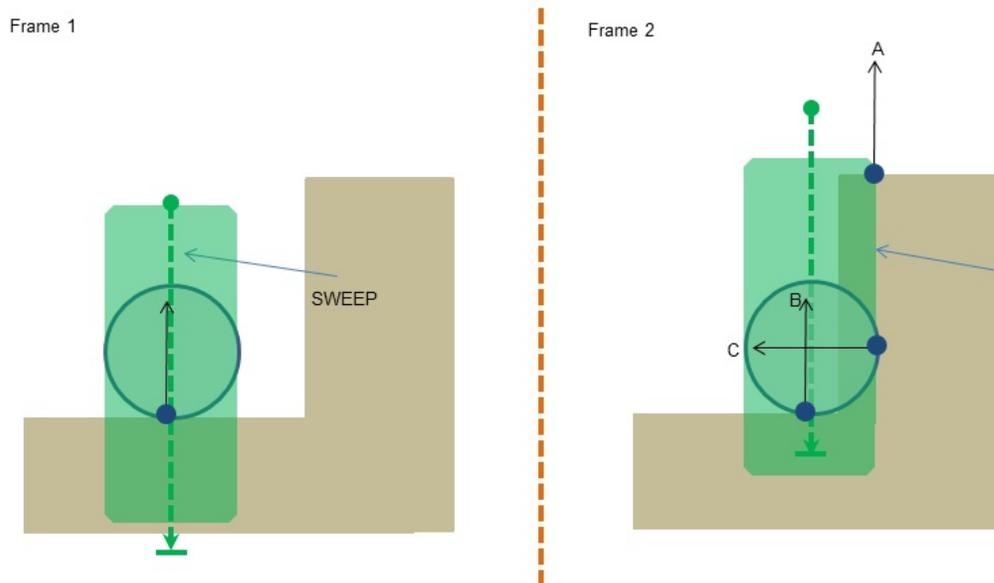


Figure 4: Judicious selection of sweep contacts and rigid body contacts can help a vehicle navigate a wheel through a complex scene.

Figure 4 shows a wheel rolling along a horizontal plane towards a vertical wall. The expected behavior is that the wheel continues to drive on the horizontal surface until it is blocked by the vertical plane. It turns out that this can be readily achieved by a judicious choice of sweep contacts and rigid body contacts. The first thing to not

will return the three contact planes labelled A, B and C in Figure 4. In rigid body contact between the wheel and the environment we will see contact planes B and C as rigid body contacts. The next step is to develop a method that accepts contact plane B for the sweep and contact plane C for rigid body contact. This combination will ensure that the wheel bounces off the vertical plane and drives on the lower horizontal plane. The strategy adopted by Physics Engine will be to categorise sweep and rigid body contacts by comparing contact normals with the suspension direction. The aim is to divide contact with the environment into drivable contact planes and non-drivable contact planes. This can be achieved by setting threshold angles to categories contact points and normals.

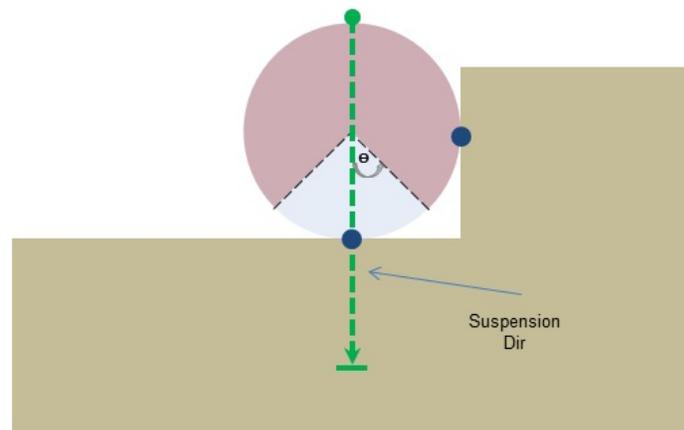


Figure 5: The position of sweep and rigid body contact points relative to the suspension direction is used to filter the sweep and rigid body contacts. Sweep contacts in the blue zone are accepted as driving planes, while rigid body contacts in the red zone are accepted as rigid body contact planes.

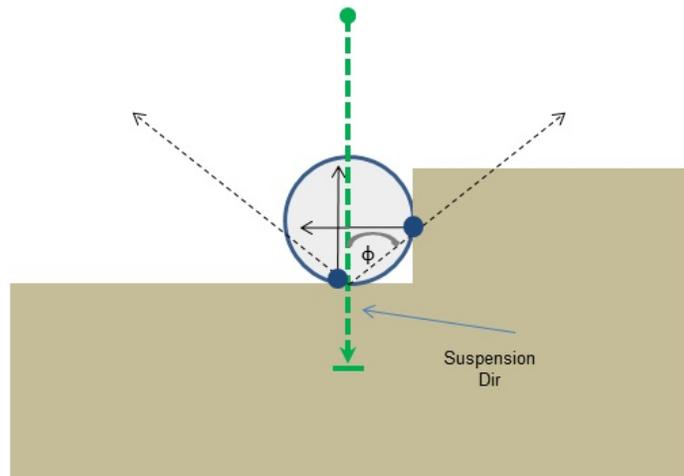


Figure 6: The angle between contact normal and the suspension direction categorised contact planes as either rigid body contacts or sweep contacts. Contact normals close to the suspension direction are accepted as drivable planes, while contact normals far from the suspension direction are accepted as rigid body contacts.

Figures 5 and 6 introduced two threshold angles that together allow sweep contacts to be categorised using their position and normal. Having a drivable and non-drivable contact points and normals allows a relationship between the contact normal and suspension direction. Filtering rules described in Section *Filtering*. The idea now is to set up so that wheel shapes sweep against and collide with pretty much everything. The two threshold angles will filter and categorise sweep and rigid contacts to generate the desired behavior.

The threshold angles shown in Figure 5 and Figure 6 are configured in the following function call:

```
void PxVehicleSetSweepHitRejectionAngles(const PxF32 pointRejectA
```

The code snippet `SnippetVehicleContactMod` demonstrates how to configure non-blocking sweeps. This snippet can be configured to run with either `BLOCKING_SWEEPS` by modifying the `BLOCKING_SWEEPS` define. Running the

BLOCKING_SWEEPS demonstrates that the situation depicted in Figure 10.10 requires blocking sweeps to ensure that the elevated horizontal plane is not chattering on the surface.

Suspension sweeps are issued with the following code:

```
//Suspension sweeps (instead of raycasts).  
//Sweeps provide more information about the geometry under the wheel.  
PxVehicleWheels* vehicles[NUM_VEHICLES] = {gVehicle4W[0], gVehicle4W[1], gVehicle4W[2], gVehicle4W[3]};  
PxSweepQueryResult* sweepResults = gVehicleSceneQueryData->getSweepQueryResults();  
const PxU32 sweepResultsSize = gVehicleSceneQueryData->getQueryResultsSize();  
PxVehicleSuspensionSweeps(gBatchQuery, NUM_VEHICLES, vehicles, sweepResults, sweepResultsSize);
```

In the event that non-blocking sweeps are implemented, the function `PxVehicleSweepHitRejectionAngles` rejects and accepts sweep hits using the threshold angles. When blocking sweeps are used, a single sweep contact is recorded. As a consequence, `PxVehicleUpdateSweepHitRejectionAngles` automatically works with the blocking sweep. Whether to use blocking or non-blocking sweeps is left to the developer and depends on knowledge about the kinds of geometry that will be encountered by the vehicle. In some applications it will be sufficient to opt for the compilation option of blocking sweeps, while other applications may expect the vehicle to encounter complex geometry and are prepared to accept the extra cost of non-blocking sweeps.

Categorisation of rigid body contacts is implemented using contact modification. Contact modification is described in Section [Contact Modification](#). The SDK provides the function `PxVehicleModifyWheelContacts` to accept contact modification callbacks, which is owned by the application. Configuring contact modification callbacks involves a combination of simulation filter data and a shader. The implementation details, therefore, are left to the application developer. Snippet `VehicleContactMod` illustrates one way to implement a contact modification callback using simulation filter data and the userdata pointers `PxRigidBody`. Other techniques are available using local knowledge of the scene. In addition to adding sweeps and contact modification, the snippet also adds collision detection (CCD) to the wheel shapes. CCD is introduced in Section [Collision Detection](#).

Collision Detection.

Tuning Guide

This Section describes the effect of the editable vehicle parameters of in PxVehicleComponents.h.

PxVehicleWheelData

mRadius:

This is the distance in metres between the center of the wheel and the tire. It is important that the value of the radius closely matches the render mesh of the wheel. Any mismatch will result in the wheels either above the ground or intersecting the ground. Ideally, this parameter will be exported from the 3D modeler.

mWidth:

This is the full width of the wheel in metres. This parameter has no bearing on handling but is a very useful parameter to have when trying to render the wheel/tire/suspension. Without this parameter it would be difficult to compute coordinates for render points and lines that ensure their visibility parameter will be exported from the 3D modeler.

mMass:

This is the combined mass of the wheel and the tire in kg. Typically, a mass between 20Kg and 80Kg but can be lower and higher depending on the vehicle.

mMOI:

This is the component of the wheel's moment of inertia about the roll axis. Higher values make it harder for the wheel to rotate about this axis, while lower values make it easier for the wheel to rotate about the rolling axis. Another way to think of it is that a high MOI will result in less wheel spin when stamping or because it is harder to make the wheel spin. Conversely, lower values will result in more wheel spin when stamping on the accelerator.

If the wheel is approximately cylindrical then a simple formula compute MOI:

$$\text{MOI} = 0.5 * \text{Mass} * \text{Radius} * \text{Radius}$$

There is no reason, however, to rely on equations to compute this strategy for tuning this number might to be start with the equation make small tweaks to the value until the handling is as desired.

mDampingRate:

This value describes how quickly a freely spinning wheel will come to damping rate describes the rate at which a freely spinning wheel lose speed. Here, a freely spinning wheel is one that experiences no force damping forces arising from the wheel's internal bearings. Higher damping result in the wheel coming to rest in shorter times, while lower damping the wheel maintaining speed for longer. Values in range (0.25, 2) seem values. Experimentation is always a good idea, even outside this range exercise some caution with very small damping rates. In particular, a exactly 0 should be avoided.

mMaxBrakeTorque:

This is the value of the torque applied to the wheel when the brake applied. Higher torques will lock the wheel quicker when braking, while will take longer to lock the wheel. This value is strongly related to because the MOI determines how quickly the wheel will react to applied

A value of around 1500 is a good starting point for a vanilla wheel but will reveal typical braking torques. One difficulty is that these are often manufacturers as braking horsepower or in "pounds inches". The here are in "Newton metres".

mMaxHandBrakeTorque:

This is the same as the max brake torque except for the handbrake rate brake. Typically, for a 4-wheeled car, the handbrake is stronger than the

only applied to the rear wheels. A value of 4000 for the rear wheels is a good point, while a value of 0 is necessary for the front wheels to make sure they react to the handbrake.

mMaxSteer:

This is the value of the steer angle of the wheel (in radians) when the wheel is at full lock. Typically, for a 4-wheeled car, only the front wheels respond to steering. In this case, a value of 0 is required for the rear wheels. More exotic cars might wish front and rear wheels to respond to steering. A value in radians somewhere between 30 degrees and 90 degrees seems like a good value but it really depends on the vehicle being simulated. Larger values of mMaxSteer result in tighter turns, while smaller values will result in wider turns. But that large steer angles at large speeds are likely to result in the car losing control and spinning out of control, just as would happen with a real car. A good value is to filter the steer angles passed to the car at run-time to generate smaller steer angles at larger speeds. This strategy will simulate the difficulty of achieving large steer angles at high speeds (at high speeds the wheels resist the turn applied by the steering wheel).

mToeAngle:

This is the angle of the wheel (in radians) that occurs with no steer angle. A small toe angle can be used to help the car straighten up after coming out of a turn. A good number to experiment with but is best left at 0 unless details are required.

To help the car straighten up apply a small negative angle to one of the front wheels and a small positive angle to the other front wheel. By choosing which wheel has the positive angles, and which the negative, it is straightforward to create either "toe-in" or "toe-out". A "toe-in" configuration, where the front wheels point towards each other, should help the car straighten up after a turn but at the cost of making it a little harder to turn in the first place. A "toe-out" configuration has the opposite effect. Toe angles greater than a few degrees are best avoided.

PxVehicleWheelsSimData

`void setSuspTravelDirection(const PxU32 id, const PxVec3& dir):`

This is the direction of the suspension in the downward direction in the configuration of the vehicle. A vector that points straight downwards is point.

`void setSuspForceAppPointOffset(const PxU32 id, const PxVec3& offset):`

This is the application point of the suspension force, expressed as a vector from the center of mass of the vehicle's rigid body. Another way of thinking about it is to start at the center of mass of the rigid body, then move along the offset vector to a point at the end of the offset vector is the point at which suspension force is applied.

In a real vehicle the suspension forces are mediated through the suspension struts. These are often incredibly complex mechanical systems that are expensive to simulate. As a consequence, instead of modeling the suspension strut, it makes sense to assume that the suspension strut applies the force to the rigid body. Choosing that point at which it applies the force to the rigid body. Choosing that point needs careful consideration. At the same time, it opens up all simulation possibilities, freed from the constraints of the real world.

Deciding on the suspension force application point requires some thought. In a real vehicle suspension is very close to the wheel so the wheel center is a good starting point. Consider a line through the wheel center and along the suspension direction. Somewhere along this line seems like an even better idea for the application point, albeit not completely scientific. For a standard 4-wheeled car it makes sense that the application point is somewhere above the wheel center but below the center of mass of the rigid body. It is probably above the wheel center because the suspension is mostly above this point. It can be assumed that it is somewhere below the center of mass because otherwise vehicles would lean out of the turn during the turn. This narrows down the application point to really quite a small known line.

When editing the suspension force application point it is important to be careful. Lowering the app point too far will result in cars leaning more into the turn.

have a negative effect on handling because the inner wheel can take that the response saturates, while the outer wheel ends up with reduced turning force. The result is poor cornering. Conversely, setting too high will result in cornering that looks unnatural. The aim is to balance.

```
void setTireForceAppPointOffset(const PxU32 id, const PxVec3& offset);
```

This is almost the same as the suspension force app point except for longitudinal forces that develop on the tire. A good starting point is to use the suspension force application point. Only for really detailed editing is it worth tweaking the tire force app offset independently of the suspension force app point.

```
void setWheelCentreOffset(const PxU32 id, const PxVec3& offset);
```

This is the center of the wheel at rest position, expressed as an offset from the vehicle's center of mass.

PxVehicleSuspensionData

mSprungMass:

This is the mass in kg that is supported by the suspension spring.

A vehicle with rigid body center of mass at the center of the footprint typically be equally supported by each of the suspension springs. Each suspension spring supports 1/4 of the total vehicle mass. If the center of mass moved forward then it would be expected that the front wheels would support more mass than the rear wheels. Conversely, a center of mass near the rear ought to result in the rear suspension springs supporting more mass than the front.

Note: In order to achieve stability at the desired rest pose it is recommended that the collection of sprung masses matches the mass and center of mass of the rigid body. There are two strategies that can be employed to achieve this. The first approach is to decide upon values for the individual sprung masses and work forwards to compute an equivalent value for the rigid body center of mass.

center of mass. More specifically, the rigid body mass and center of mass can be computed using the equations presented in Section [Algorithm 2](#) and applied to the vehicle's `PxRigidBody` instance. The second approach uses the rigid body mass and center of mass of the vehicle's `PxRigidBody` instance and works backwards to compute and set the sprung mass. This approach makes use of the function `PxVehicleComputeSprungMasses` that was introduced in Section [setupWheelsSimulationData](#).

`mMaxCompression`:

`mMaxDroop`:

These values describe the maximum compression and elongation the spring can support. The total travel distance along the spring direction is the sum of `mMaxCompression` and `mMaxDroop`.

A simple way to illustrate the maximum droop and compression values is to suspend a car that is suspended in mid-air so that none of the wheels are touching the ground. The wheels will naturally fall downwards from their rest position until the droop is reached. The spring cannot be elongated beyond this point because that the wheel is pushed upward, first to its rest position, then further until the spring can no longer be compressed. The displacement from the rest position to the maximum compression of the spring.

It is important to choose the maximum compression value so that the wheel is placed where the visual mesh of the wheel intersects the visual mesh of the chassis. Ideally, these values will be exported from the 3d modeler.

`mSpringStrength`:

This value describes the strength of the suspension spring. The spring has a profound influence on handling by modulating the time it takes for the car to respond to bumps in the road and on the amount of load experienced.

Key to understanding the effect of spring strength is the concept of natural frequency. Consider a simple spring system, such as a pendulum system.

forth. The number of trips per second that the pendulum makes forward and then back again is called the natural frequency of the pendulum. A powerful pendulum spring will result in the pendulum swinging with a higher natural frequency. Conversely, increasing the pendulum's mass will result in a slower oscillation, thereby reducing the natural frequency.

In the context of a suspension spring supporting a fixed portion of a vehicle's weight, the strength of the spring will affect the natural frequency; that is, the spring can respond to changes in load distribution. Consider a car taking a turn: as the car corners it leans into the turn, putting more weight on the suspension on the outside of the turn. The speed at which the spring reacts by adjusting the load is controlled by the natural frequency. Very high natural frequencies, such as those on a racing car, will naturally produce a stiff ride because the load on the tires, and therefore the forces they can generate, changes very rapidly. Very low natural frequencies, on the other hand, will result in sluggish and unresponsive handling.

Another effect of strength and natural frequency is the response to a bump in the road. High natural frequencies can result in the car reacting strongly and quickly to the bump, with the wheel possibly even leaving the ground for a short while. This not only creates a bumpy ride but also periods of time where the wheel is generating no forces. Weaker springs will result in a smoother trip with weaker but more constant tire forces. A balance must be found between the expected types of turn and terrain.

The natural frequency of the spring presents a challenge for computer simulation. A smooth and stable simulation requires that the spring is updated at a frequency greater than the spring's natural frequency. An alternative way of expressing this is to consider the period of the spring relative to the timestep of the simulation. The period of the spring is the time the spring takes to complete a single oscillation, which is mathematically equal to the reciprocal of the natural frequency. In order for a stable simulation the spring must be sampled at several points during each period. A natural consequence of this observation is that the simulation timestep must be significantly smaller than the period of the spring. To discuss this further,

introduce a ratio that describes the number of simulation updates during each spring oscillation. This ratio is simply the spring period / timestep

$$\alpha = \text{sqrt}(m\text{SprungMass}/m\text{SpringStrength})/\text{timestep}$$

where $\text{sqrt}(m\text{SprungMass}/m\text{SpringStrength})$ is the period of the spring. A value of 1.0 means that the chosen timestep and spring properties are one sample of the spring during each oscillation. As described above, $\alpha < 1.0$ is guaranteed to produce unstable behavior. In fact, the argument $\alpha < 1.0$ suggests a value of alpha significantly greater than 1.0 is essential for a smooth simulation. The exact value of alpha at which stability emerges is hard to predict and depends on many other parameters. As a guide, it is recommended that the timestep and spring properties are chosen such that they produce an alpha value greater than 5.0; that is, a minimum of five samples per spring cycle.

When tuning a suspension spring it can be very useful to use manufacturer data to discover typical values used across a range of vehicle types. This data is readily available. An alternative strategy would be to think in terms of the natural frequency of the spring by imagining how quickly the car would oscillate if it was dropped onto the ground from a height of, say, 0.5m. The spring constant of a family car have natural frequency somewhere between 5 and 10; that is, the car would make 5-10 oscillations per second if gently dropped to the ground. If the mass supported by the spring is already known then the spring strength can be calculated from the following equation

$$m\text{SpringStrength} = \text{naturalFrequency} * \text{naturalFrequency} * m\text{SprungMass}$$

Note: To achieve a spring that is theoretically correct, the values $m\text{SprungMass}$, $m\text{SpringStrength}$ and $m\text{MaxDroop}$ should be chosen such that they obey the equation $m\text{SpringStrength} * m\text{MaxDroop} = m\text{SprungMass} * \text{gravitationalAcceleration}$. When this equation is satisfied, the spring is guaranteed to provide exactly zero force at maximum elongation and is also able to support the sprung mass at the rest pose (the rest pose is where the spring force equals the weight of the mass).

PxVehicleWheelsSimDta::setWheelCentreOffset). It is often the case that the visual requirements of the car are in conflict with its handling requirements. An example might be a visual requirement, imposed on both the rest pose and the suspension travel limits. In order to satisfy a visual requirement and achieve a theoretically correct spring, the value of `mSpringStrength` must be equivalent to $mSprungMass * gravitationalAcceleration / mMaxDroop$. If this value of `mSpringStrength` does not meet the handling requirements of the vehicle, there is a conflict that cannot be easily resolved. For this reason, the Vehicles module does not require the spring to be a theoretically perfect spring. The consequences of an imperfect spring are that the spring either does not provide upward force before it hits maximum elongation or that it provides a non-zero force at maximum elongation. The effect on handling or the visual appearance of the vehicle is often quite difficult to spot. In practice, load filtering, discussed in Section [PxVehicleTireLoadFilterData](#), further disguises any imperfection.

`mSpringDamperRate`:

This describes the rate at which the spring dissipates the energy stored in it.

Key to the understanding of damper rate are the concepts of under-damping, and critical damping. An over-damped pendulum is unable to make a single back-and-forth trip before it dissipates all its energy. An under-damped pendulum would be able to make at least a single back-and-forth trip. A critically damped pendulum makes exactly a single back-and-forth trip before expending all its energy.

For vehicle suspension springs, it is typically important to make sure the spring has a damper rate that produces over-damping but not by too much. For example, it is important that the spring doesn't over-respond by sloshing from the left suspension to the right suspension then back again. If the tire load, and the forces generated, would be extremely variable, resulting in uncontrollable handling. A very heavily over-damped spring, on the other hand, will feel sluggish and unresponsive.

The concept of critical damping can be used to help tune the damper spring. It is helpful to introduce a value known as the damping ratio which mathematically describes the under-damping, critical damping and over-damping regimes.

$$\text{dampingRatio} = \text{mSpringDamperRate} / [2 * \text{sqrt}(\text{mSpringStrength} * \text{mSprungMass})]$$

A dampingRatio with a value greater than 1.0 produces over-damping, a value exactly 1.0 generates critical damping, and a value less than 1.0 is under-damping. It can be useful to first think about whether the spring will be under-damped, then think about how far it will be from critical damping. This is a number to be subjectively applied to the damping ratio. From here the damping ratio can be directly computed by rearranging the equation above

$$\text{mSpringDamperRate} = \text{dampingRatio} * 2 * \text{sqrt}(\text{mSpringStrength} * \text{mSprungMass})$$

A typical family car is probably slightly over-damped, having a damping ratio perhaps just over 1.0. A guideline would be that values very far from 1.0 are likely to be unrealistic and will either produce sluggish or twitchy handling. It is difficult to put an exact figure on this but somewhere between 0.8 and 1.2 is a good starting point for the damping ratio.

mCamberAtRest:

mCamberAtMaxCompression:

mCamberAtMaxDroop:

These values describe the camber angle of the wheels as a function of spring compression. It is typical for the wheels of extended springs to have negative camber; that is, the left and right wheels almost seem to form the edges of a triangle when viewed from the front or rear along the forward axis of the vehicle. On the other hand, typically camber is positive; that is, they form the outer edges of an A shape when viewed from the front or rear along the forward axis of the vehicle.

axis of the vehicle.

These three values allow the camber angle to be computed for any compression using simple linear interpolation. At rest, when the spring is elongated or compressed, the camber angle is equal to `mCamberAtRest` and `mCamberAtMaxCompression`. When the spring is compressed the camber is computed as a linear interpolation between `mCamberAtRest` and `mCamberAtMaxCompression`. When the spring is elongated the camber is computed as a linear interpolation between `mCamberAtRest` and `mCamberAtMaxDroop`.

The camber angle is used by the default tire model and is passed as an argument to the tire shader. It is also used to set the local pose of the wheel which geometrically represents the wheel.

PxVehicleAntiRollBar

When a vehicle takes a corner the turning force causes the car to roll. The suspension springs on the outside of the turn are compressed while the springs on the inside of the turn are elongated. If the roll is so severe that the wheels completely leave the ground then there is a danger that the driver loses control of the vehicle. In such cases, there is even a danger that the car will roll onto its side. For less severe rolls there still remains a handling problem due to the distribution of load between the inside and outside tires. The roll of the car that the imbalance of the vehicle can lead to under-steer or over-steer.

Anti-roll bars are commonly used to reduce the roll that naturally occurs during cornering. They typically work as a torsion spring that applies a torque to minimise the difference in spring displacement for a pair of wheels. A car might feature a front and rear anti-roll bar. The front bar applies a torque to the difference between the front-left and front-right wheels. Similarly, the rear bar applies a torque to reduce the difference between the rear-left and rear-right wheels.

The magnitude of the anti-roll torque is proportional to the difference in spring displacement of the two wheels that are connected by the bar. The roll stiffness is proportional to a stiffness parameter: stiffer bars generate more anti-roll torque.

As a general rule, under-steer can be reduced by increasing the stiffness of the front anti-roll bar. Increasing the stiffness of the front anti-roll bar typically reduces under-steer.

mWheel0: mWheel1:

The anti-roll bar connects two wheels described by the indices mWheel0 and mWheel1.

mStiffness:

This parameter describes the stiffness of the anti-roll bar.

PxVehicleTireData

The tire force computation is performed in two conceptual stages. In the first stage, the computation independently computes the lateral and longitudinal tire forces using linear equations. These independent forces are combined to view the tire as a linear system so that the force in each direction can be viewed as the product of a tire strength per unit slip and the slippage of the tire. The second stage of the computation applies the rule that the total force is limited by the product of the tire load and friction. Just as all tires are able to resist greater horizontal forces when they experience a greater load on a surface with high friction value. With this in mind the maximum force for a tire can be approximated as the product of the normal load and the friction value. The default PhysX Vehicle tire model employs a series of smooth functions to implement the normalization of the combined tire forces.

In addition to the lateral and longitudinal components of force a component arising from the camber angle of the tire, is also computed. Typically only a small correction to the effect of the lateral and longitudinal components of camber force participates in the normalization process.

The following tire parameters describe the computation of the independent lateral, longitudinal and camber components; that is, the first conceptual stage of the computation. Reference is made throughout to the handling cons

normalization process.

mLongitudinalStiffnessPerUnitGravity:

The longitudinal stiffness describes the longitudinal forces that develop when the tire slips (in radians). Here, a variable that represents the longitudinal stiffness per unit gravity has been introduced in order to make the variable more dependent on the value of gravitational acceleration. The longitudinal stiffness is approximately the product of the longitudinal stiffness per unit gravity and the magnitude of gravitational acceleration:

$$\text{longitudinalTireForce} = \text{mLongitudinalStiffnessPerUnitGravity} * \text{gravity};$$

Increasing this value will result in the tire attempting to generate more longitudinal force when the tire is slipping. Typically, increasing longitudinal stiffness will help the car accelerate and brake. The total tire force available is limited by the tire load so be aware that increases in this value might have no effect or even be at the expense of reduced lateral force.

mLatStiffX:

mLatStiffY:

These values together describe the lateral stiffness per unit lateral slip of the tire. The lateral stiffness of a tire has a role similar to the longitudinal stiffness (mLongitudinalStiffnessPerUnitGravity), except that it governs the lateral tire forces, and is a function of tire load. Typically, increasing lateral stiffness will help the car turn more quickly. The total tire force available is limited by the tire load so be aware that increases in this value might have no effect or even be at the expense of reduced longitudinal force.

Lateral stiffness is a little more complicated than longitudinal stiffness. Lateral stiffness typically provide poor response under heavy load. Typical for car tires is that lateral force against load that has linear response close to zero load and decreases at greater loads. This means that at low tire loads the lateral stiffness

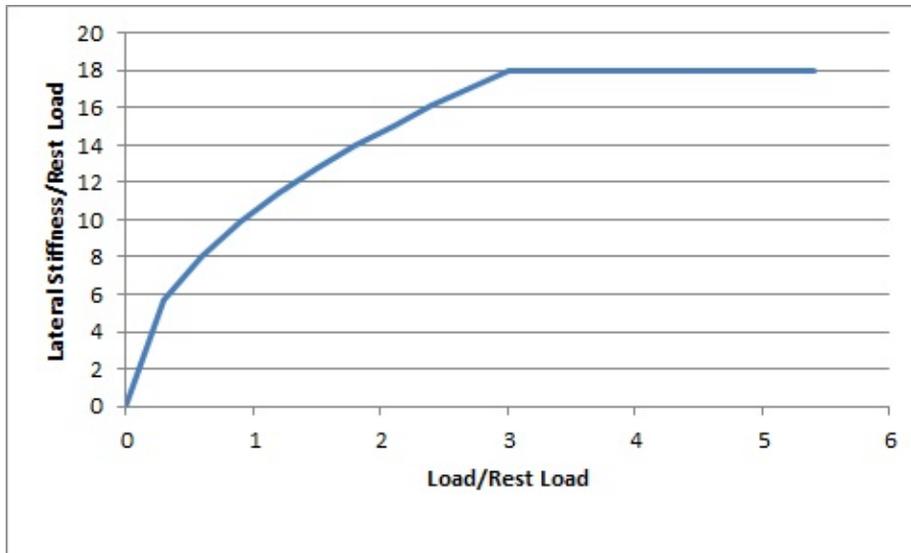
response to load; that is, more load results in more stiffness and more force. At higher tire loads the tire has a saturated response and if you are applying more load will not result in more tire stiffness. In this latter regime it is expected that the tire would start slipping.

The combination of two values $mLatStiffX$ and $mLatStiffY$ describe a stiffness per unit load as a function of normalized tire load. The tire force employs a smoothing function which requires knowledge of the normalized tire load at which the tire has a saturated response to tire load along with the lateral unit load that occurs at this saturation point. A typical curve can be seen below.

The parameter $mLatStiffX$ describes the normalized tire load above which there is a saturated response to tire load. The normalized tire load is simply the tire load divided by the load that is experienced when the vehicle is perfectly aligned. A value of 2 for $mLatStiffX$, for example, means that when the tire has a load that is twice its rest load it can deliver no more lateral stiffness no matter how much load is applied to the tire. In the graph below $mLatStiffX$ has value 3.

The parameter $mLatStiffY$ describes the maximum stiffness per unit load (in radians) per unit rest load. The maximum stiffness is delivered when the tire is in a saturated load regime, governed in turn by $mLatStiffX$. In the graph below $mLatStiffY$ has value 18.

The computation of the lateral stiffness begins by computing the lateral load, then computing the normalized load in order to compute the normalized lateral load experienced by the tire. This places the tire somewhere along the X-axis of the graph below. The corresponding value on the Y-axis of the curve $pLatStiff$ for a given $mLatStiffX$ and $mLatStiffY$ is queried to provide the lateral stiffness per unit load. The final value for the lateral stiffness is then computed by multiplying the graph value by the rest load. This final value describes the lateral stiffness at the lateral slip.



A good starting value for mLatStiffX is somewhere between 2 and 3
 value for mLatStiffY is around 18 or so.

mFrictionVsSlipGraph:

These six values describe a graph of friction as a function of longitudinal slip. Vehicle tires have a complicated response to longitudinal slip attempts to approximate this relationship.

Typically, tires have a linear response at small slips. This means that when a tire is only slightly slipping, it is able to generate a response force. As the slip increases, the force can increase to a peak value and then decrease from the peak value that occurs at the optimum slip. Beyond the optimum slip, the tire eventually starts behaving less and less efficiently, reaching a plateau of inefficiency.

The friction value for the combination of surface type and tire type has been discussed in Section *Tire Friction on Drivable Surfaces*. The friction versus longitudinal slip is used as a correction to the combination friction value. In particular, a final friction value is computed from the combination friction value and the graph's correction value. The final friction value responds to the final friction value.

The first two values describe the friction at zero tire slip: $mFrictionVsSlipGraph[0][0] = 0$, and $mFrictionVsSlipGraph[0][1] = \text{friction at zero slip}$.

The next two values describe the optimum slip and the friction at optimum slip: $mFrictionVsSlipGraph[1][0] = \text{optimum slip}$, $mFrictionVsSlipGraph[1][1] = \text{friction at optimum slip}$.

The last two values describe the slip at which the plateau of inefficiency begins and the value of the friction available at the plateau of inefficiency: $mFrictionVsSlipGraph[2][0] = \text{slip at the start of the plateau of inefficiency}$, $mFrictionVsSlipGraph[2][1] = \text{the friction available at the plateau of inefficiency}$.

In the graph below the following values have been used:

$$mFrictionVsSlipGraph[0][0] = 0.0$$

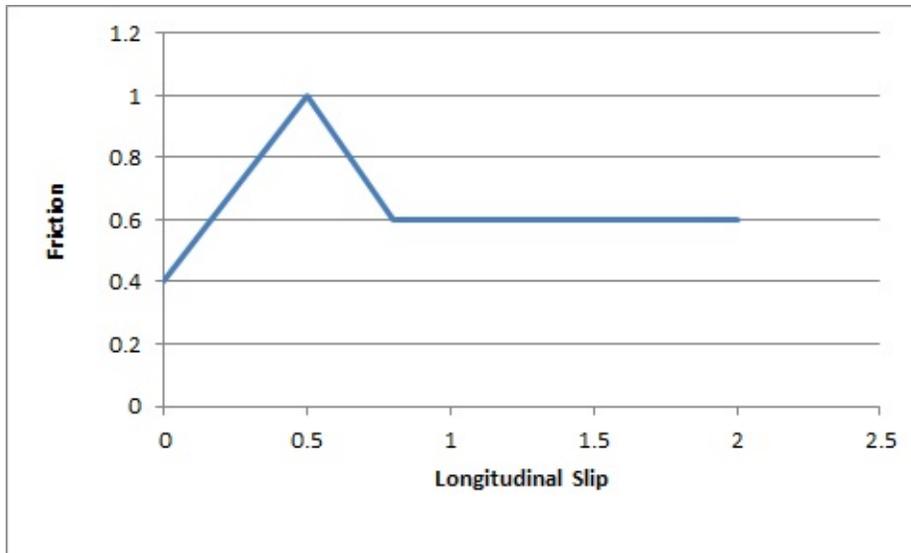
$$mFrictionVsSlipGraph[0][1] = 0.4$$

$$mFrictionVsSlipGraph[1][0] = 0.5$$

$$mFrictionVsSlipGraph[1][1] = 1.0$$

$$mFrictionVsSlipGraph[2][0] = 0.75$$

$$mFrictionVsSlipGraph[2][1] = 0.60$$



The friction values described here are used to scale the friction of the This means they should be in range (0,1) but this is not a strict requir the friction from the graph would be close to 1.0 in order to provide a to the ground surface friction.

A good starting point for this is a flat graph of friction vs slip with these

`mFrictionVsSlipGraph[0][0]=0.0`

`mFrictionVsSlipGraph[0][1]=1.0`

`mFrictionVsSlipGraph[1][0]=0.5`

`mFrictionVsSlipGraph[1][1]=1.0`

`mFrictionVsSlipGraph[2][0]=1.0`

`mFrictionVsSlipGraph[2][1]=1.0`

`mCamberStiffnessPerUnitGravity:`

The camber stiffness is analogous to the longitudinal and lateral stiffi it describes the camber thrust force arising per unit camber angle (in to the longitudinal stiffness, a camber stiffness per unit gravity has be

make the camber stiffness robust across different values of gravitational acceleration. The independent camber force is computed as the camber angle multiplied by the camber stiffness multiplied by the gravitational acceleration:

$$\text{camberTireForce} = \text{mCamberStiffnessPerUnitGravity} * \text{camberAngle}$$

mType:

This parameter has been explained in Section [Tire Friction on Drivable](#).

PxVehicleEngineData

mMOI:

This is the moment of inertia of the engine around the axis of rotation. Lower values make it harder to accelerate the engine, while higher values make it easier to accelerate the engine. A starting value of 1.0 is a good choice.

mPeakTorque:

This is the maximum torque that is ever available from the engine. The unit is in Newton metres. A starting value might be around 600.

mMaxOmega:

This is the maximum rotational speed of the engine expressed in radians per second.

mDampingRateFullThrottle:

mDampingRateZeroThrottleClutchEngaged:

mDampingRateZeroThrottleClutchDisengaged:

These three values are used to compute the damping rate that is applied to the engine. If the clutch is engaged then the damping rate is an interpolation between `mDampingRateFullThrottle` and `mDampingRateZeroThrottleClutchEngaged`. If the clutch is disengaged then the damping rate is `mDampingRateZeroThrottleClutchDisengaged`. The interpolation is governed by the acceleration control value `g`.

gamepad or keyboard. At full throttle `mDampingRateFullThrottle` is applied. At zero throttle `mDampingRateZeroThrottleClutchEngaged` is applied. At zero throttle with the clutch disengaged `mDampingRateZeroThrottleClutchDisengaged` is applied. The damping rate is an interpolation between `mDampingRateFullThrottle` and `mDampingRateZeroThrottleClutchDisengaged`.

The three values allow a range of effects to be generated: good acceleration, hampered by strong damping forces, tunable damping forces when in neutral gear during a gear change, and strong damping forces that bring the vehicle quickly to rest when it is no longer being driven by the player.

Typical values in range (0.25,3). The simulation can become unstable if the damping rates are 0.

`mTorqueCurve`:

This is a graph of peak torque versus engine rotational speed. Cars have a range of engine speeds that produce good drive torques, and other engine speeds that produce poor torques. A skilled driver will make good use of the engine to ensure that the car remains in the "good" range where the engine is most efficient. Tuning this graph can have profound effects on gameplay.

The x-axis of the curve is the normalized engine speed; that is, the engine speed divided by the maximum engine speed. The y-axis of the curve is a normalized torque (0,1) that is used to scale the peak torque.

PxVehicleGearsData

`mNumRatios`:

This is the number of the gears of the vehicle, including reverse and neutral. A standard car with 5 forward gears would, therefore, have a value of 7 for reverse and neutral.

`mRatios`:

Each gear requires a gearing ratio. Higher gear ratios result in more torque

It is worth noting that if the autobox initiates a gear change then the clutch is automatically disconnected from the engine for the entire duration of the gear change. Manual gear changes (`PxVehicleDriveDynData::startGearChange`, `PxVehicleDriveDynData::mGearUpPressed`, `PxVehicleDriveDynData::mGearDownPressed`) are not subject to this behavior, which is in keeping with typical real-world autobox behavior. The idea behind this is to avoid the engine wildly accelerating during the neutral phase of the gear change, thus avoiding damaging clutch slip when the clutch re-engages at the end of the change.

The autobox will not try to initiate a gear change while an automatic gear change is still active.

If the autobox is too simplistic for the application's requirements, `PxVehicleGearsData` can be readily disabled. The choices following `PxVehicleGearsData::mUseAutoGears` revert to a manual gear model or to implement a custom autobox in `PxVehicleDriveDynData::startGearChange`, while single gear changes are initiated with `PxVehicleDriveDynData::mGearUpPressed` and `PxVehicleDriveDynData::mGearDownPressed`.

The autobox can be enabled or disabled via `PxVehicleDriveDynData::mUseAutoGears`.

`PxReal mUpRatios[PxVehicleGearsData::eGEARSRATIO_COUNT];`

The autobox will initiate a gear increment if the ratio of the engine rotation speed to the maximum allowed engine rotation speed:

$$\frac{\text{PxVehicleDriveDynData::getEngineRotationSpeed()}}{\text{PxVehicleEngineData::mMaxOmega}}$$

is greater than the value `mUpRatios[PxVehicleDriveDynData::getCurrentGear()]`.

`PxReal mDownRatios[PxVehicleGearsData::eGEARSRATIO_COUNT];`

The autobox will initiate a gear decrement if the ratio of the engine to the maximum allowed engine rotation speed:

```
PxVehicleDriveDynData::getEngineRotationSpeed() /  
PxVehicleEngineData::mMaxOmega
```

is less than the value stored in `mUpRatios[PxVehicleDriveDynData::g`

```
void setLatency(const PxReal latency):
```

After the autobox has initiated a gear change it will not attempt to initiate another change until the latency time has passed. It is a good idea to set this latency significantly higher than `PxVehicleGearsData::mSwitchTime`. If the latency is less than the gear switch time then the autobox might decide to initiate a gear change immediately after an upward gear shift has been completed. This can leave the car cycling between neutral and first gear with very short intervals between 2nd gear.

PxVehicleClutchData

`mStrength`:

This describes how strongly the clutch couples the engine to the wheels. A value of 10 quickly differences in speed are eliminated by distributing torque to all four wheels.

Weaker values will result in more clutch slip, especially after clutch release and stamping on the accelerator. Stronger values will result in reduced clutch slip and more engine torque delivered to the wheels.

This value is to be edited only for very fine tweaking of the vehicle. A value of 10 can be attributed to the numerical issues in the simulation at large values. A value of 10 is a natural consequence of driving the car in an overly aggressive manner. A value of 10 is a good starting point.

PxVehicleAckermannGeometryData

mAccuracy:

Ackermann correction allows better cornering by steering the left and right wheels with slightly different steer angles, as computed from simple trigonometry. It is impossible to engineer a steering linkage that will achieve the perfect steering correction. This value allows the accuracy of the Ackermann correction to be controlled. Choosing a value of 0 completely disables steering correction. A value of 1.0, on the other hand, achieves the perfect Ackermann correction.

mFrontWidth:

This is the distance in metres between the two front wheels.

mRearWidth:

This is the distance in metres between the two rear wheels.

mAxleSeparation:

This is the distance in metres between the center of the front axle and the rear axle.

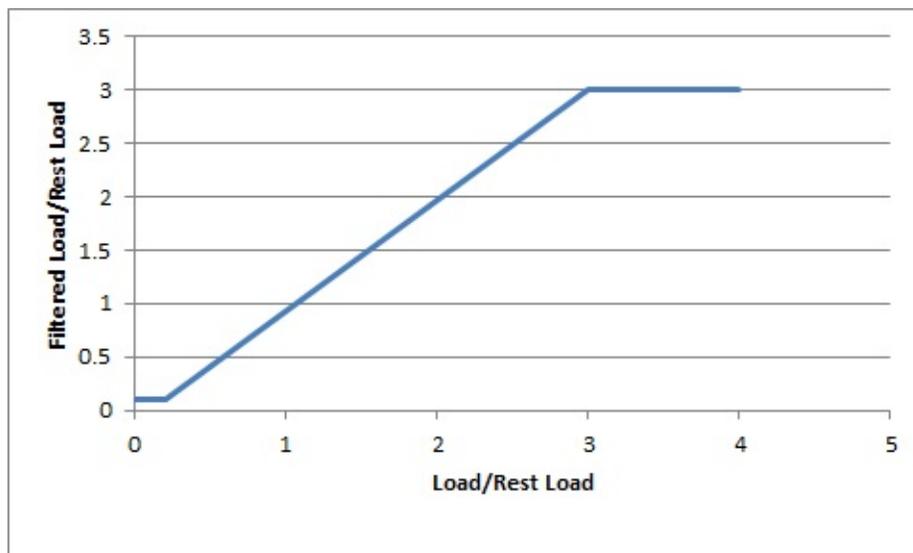
PxVehicleTireLoadFilterData

This is for very fine control of the handling, and corrects numerical issues in simulations at large timesteps.

At large simulation timesteps the amplitude of motion of the suspension is larger than it would be in real-life. This is unfortunately unavoidable. On a rough surface this could mean that the simulation lifts the car further from the ground than would really happen. This could be quickly followed by the suspension being compressed more than would be experienced with a real vehicle. A consequence of this is that the load on the tire is more variable than expected, and the tire forces have more variability than expected. This filter aims to solve this numerical problem by smoothing the tire load with the aim of making the simulation smoother and more predictable.

A key concept is that of normalized tire loads. A normalized tire load is the tire load divided by the load experienced when the vehicle is in its rest state. If a tire experiences more load than it does at rest then it has a normalized tire load greater than 1.0. Similarly, if a tire has less load than it does at rest then it has a normalized tire load less than 1.0. At rest, all tires obviously have a normalized tire load of exactly 1.0. The normalized tire load can never be less than zero.

The values here describe points on a 2d graph that generates filtered normalized tire loads. The x-axis of the graph is "normalized tire load", while the y-axis is "filtered normalized tire load". Normalized loads less than `mMinNormalisedLoad` produce a filtered normalized tire load of `mMinFilteredNormalisedLoad`. Normalized loads greater than `mMaxNormalisedLoad` produce a filtered normalized tire load of `mMaxFilteredNormalisedLoad`. Normalized loads in-between `mMinNormalisedLoad` and `mMaxNormalisedLoad` produce a filtered normalized tire load in-between `mMinFilteredNormalisedLoad` and `mMaxFilteredNormalisedLoad`, as computed by direct interpolation.



Choosing `mMaxNormalisedLoad` and `mMaxFilteredNormalisedLoad` is the choice of the maximum load that will ever be used in the simulation. On the other hand, choosing `mMinFilteredNormalisedLoad` > 0 and/or `mMinNormalisedLoad` > 0 allows the tire to potentially generate a non-zero tire force even when touching the ground at maximum droop.

The filtered load can be made identical to the computed tire load by

$mMinNormalisedLoad = mMaxFilteredNormalisedLoad = 0$
 $mMaxNormalisedLoad = mMaxFilteredNormalisedLoad = 1000.$

Note: Tires may only generate forces if the tire is touching the ground. If a tire cannot be placed on the ground then the tire force is always of zero magnitude. A tire touching the ground at maximum suspension droop, on the other hand, has zero measured load because the spring generates zero force at maximum droop. By editing PxVehicleTireLoadFilterData it is possible to generate tire forces even when there is very little load actually acting on the tire.

PxVehicleDifferential4WData

mType:

A number of differential types are supported: 4-wheel drive with open differential, front-wheel drive with limited slip, front-wheel drive with open differential, front-wheel drive with limited slip, rear-wheel drive with open differential, rear-wheel drive with limited slip.

mFrontRearSplit:

If a 4-wheel drive differential is chosen (open or limited slip) this option allows the drive torque to be split unevenly between the front and rear wheels. A value of 0.5 delivers an equal split of the torque between the front and rear wheels, the total torque delivered to the front wheels is equal to the total torque delivered to the rear wheels. Choosing a value greater than 0.5 delivers more torque to the front wheels, while choosing a value less than 0.5 delivers more torque to the rear wheels. This value is ignored for front-wheel drive and rear-wheel drive differentials.

mFrontLeftRightSplit:

This is similar to the Front Rear Split but instead splits the torque that is delivered to the front wheels between the front-left and front-right wheels. A value

delivers more torque to the front-left wheel, while a value less than 0.5 delivers more torque to the front-right wheel. This parameter can be used to prevent torque from being delivered to a damaged or disabled wheel. This value is ignored for rear-wheel drive.

mRearLeftRightSplit:

This is similar to **mFrontLeftRightSplit** except that it applies to the rear wheels instead of the front wheels. This value is ignored for front-wheel drive.

mFrontBias:

Limited slip differentials work by only allowing a certain difference in wheel rotation speed to accumulate. This prevents the situation where one wheel ends up taking all the available power. Further, by allowing a small difference in rotation speed to accumulate it is possible for the vehicle to corner more effectively by permitting the outside wheel to rotate quicker than the inside wheel.

This parameter describes the maximum difference in wheel rotation speed allowed to accumulate. The front bias is the maximum of the two front wheel rotation speeds divided by the minimum of the two front-wheel rotation speeds. If this ratio exceeds the value of the front bias the differential diverts torque from the faster wheel to the slower wheel in an attempt to preserve the maximum rotation speed ratio.

This value is ignored except for front-wheel drive or four wheel drive vehicles.

A good starting value is around 1.3.

mRearBias:

This is similar to **mFrontBias** except that it refers to the rear wheels.

This value is ignored except for rear-wheel drive or four wheel drive vehicles.

A good starting value is around 1.3.

mCentreBias:

This value is similar to the mFrontBias and mRearBias, except the sum of the front wheel rotation speeds and the sum of the rear wheel

This value is ignored except for four wheel drive with limited slip.

A good starting value is around 1.3.

PxRigidDynamic

Moment of Inertia:

The moment of inertia of the rigid body is an extremely important editing vehicles because it affects the turning and rolling of the vehicle

A good starting point for the moment of inertia of the rigid body is moment of inertia of the cuboid that bounds the chassis geometry. cuboid is W wide, H high, and L long then the moment of inertia for a M is:

$$((L*L+H*H)*M/12, (W*W+L*L)*M/12, (H*H+W*W)*M/12)$$

However, this is only a rough guide. Tweaking each value will move around the corresponding axis, with higher values making it harder rotational speed from tire and suspension forces.

Providing unphysical values for the moment of inertia will result in weird behavior or extremely twitchy and perhaps even unstable behavior. inertia must at least approximately reflect the length scales of the suspension force application points.

This parameter should be viewed as one of the first go-to editable values

Center of mass:

Along with the moment of inertia, the center of mass is one of the first

values and, as such, has a profound effect on handling.

To discuss the center of mass it is useful to consider a typical 4-wheel chassis mesh whose origin is at the center of the four wheels but requirement on the origin being at the center of the four wheels but following discussion a little simpler. It might be expected that the center of mass is somewhere near this origin because vehicles are designed in a way that the load is almost evenly between the four wheels. More specifically, it means that the center of mass needs to be a little above the base of the chassis at the height of the wheels. After all, vehicles have higher mass at the bottom of the chassis due to density of the engine and other mechanical components. As a consequence, it is expected that the center of mass is nearer the bottom of the chassis than the top, but definitely above the bottom. Without a parallel analysis of the chassis density distribution the exact location along the vertical axis is really a little arbitrary and subjective. Along the forward direction it means that the center of mass is a little nearer the front wheels than the rear wheels because of the mass of the front-located engine. Thinking about these factors of mass to be tweaked along the vertical and forward directions.

Tweaking the center of mass is really all about making incremental changes to the handling towards a desired goal. Moving the center of mass forward for cornering because more load is distributed to the front tires. However, at the expense of reduced load on the rear tires, meaning that the car is more likely to spin out because the rear tires lose grip more quickly. Extensive testing followed by tests on the handling are required.

When setting the center of mass it is important to bear in mind that sprung mass values might require simultaneous updating. If the center of mass moves nearer the front this means that more mass is supported by the front suspensions and less by the rear suspensions. This change needs to be made in a consistent way. It is possible to mathematically describe the relationship between the center of mass and the mass split between the suspensions. How the possibilities afforded by breaking this rigid link should allow more tweaking.

Mass:

A typical car might have a mass of around 1500kg.

Troubleshooting

This Section introduces common solutions to common problems with ve

Jittery Vehicles

1. Have `PxInitVehicleSDK` and `PxVehicleSetBasisVectors` been called before execution of `PxVehicleUpdates`? Check the error stream for warnings.
2. Does the length scale of `PxTolerancesScale` match the length scale of the simulation (e.g. 100 if centimeters are used)? Update `PxTolerancesScale::length` as appropriate.
3. Is the natural frequency of the spring too high/timestep of simulation too small for a reliable simulation? See Section [PxVehicleSuspensionData](#) for how to update the natural frequency or timestep accordingly. Remember that `PxVehicleWheelsSimData::setSub` can be updated per vehicle with `PxVehicleWheelsSimData::setSub`.
4. Are the maximum suspension droop and compression set to values that are too small for the suspension motion?

The Engine Rotation Refuses To Spin Quickly

1. Are the tires resisting the engine motion through excessive friction? Reduce the friction coefficient on the car very high above the ground and accelerate the engine to see if the wheels start to spin round.
2. Do the engine's moment of inertia, peak torque and damping rate scale appropriately? Note the documented SI units of each variable and recommend appropriate values.
3. Is the moment of inertia too large? A value of 1 or its equivalent in simulation scale is a good estimate for testing purposes.
4. Is the peak torque too small to drive the engine? Scale the default torque with the mass of the vehicle with the knowledge that the default torque for a standard car of around 1500kg.

5. Does the torque curve contain sensible values? Try a flat curve with having a y-value of 1.0.
6. Is the maximum engine angular speed a realistic value? Consult manufacturer data for typical values or revert to the default value for
7. Are any of the damping rates too high? Reduce the damping rates

The Engine Spins But the Wheels Refuse To Spin

1. Is the vehicle in neutral gear? Connect the engine to the wheels by to first gear and disabling the autobox.
2. Does the differential deliver drive torque to the wheels (for PxV only)? Make sure that the differential is properly configured.
3. Is the brake or handbrake engaged? Ensure that the brake and h zero.
4. Do the wheels' moment of inertia and damping rates reflect the length documented SI units of each variable and recompute the values as
5. Are the wheels' moments of inertia too high? Recompute the wheel inertia.
6. Are the wheels' damping rates too high? Reduce the wheels' damping
7. Are the tires resisting the engine motion through excessive friction? Lift the car very high above the ground and accelerate the engine to see if the wheels start to spin round.

The Wheels Are Spinning But The Vehicle Does Not Move

1. Is the filtering configured so that the vehicle is supported only by its wheels? Check the filtering configuration for shapes attached to the vehicle's actor and search for contacts involving shapes attached to the vehicle's actor
2. Is sufficient friction being delivered to the tire contact patch? Query the friction experienced by the tires during the execution of PxVeh

PxVehicleWheelsDynData::getTireFriction.

3. Do the suspension forces (and the loads on the tires) reflect the body actor? Query the suspension PxVehicleWheelsDynData::getSuspensionForce. A 4-wheeled vehicle should generate suspension forces of approximately $\text{actorMass} * \text{gravity} / 4$. Adjust the masses of the vehicle suspensions to ensure that the driven wheels have a significant tire load.
4. Do the tires generate significant longitudinal tire slip? Query PxVehicleWheelsDynData::getTireLongSlip to check that the longitudinal slip is non-zero and approaches 1.0 when the wheels are spinning during forward motion. Ensure that PxVehicleSetBasisVectors has been set to the correct forward vector if the longitudinal slip is vanishingly small. Query PxVehicleWheelsDynData::getTireLongitudinalDir to check that the forward vector has been set correctly.
5. Is the tire longitudinal stiffness too small? Adjust the longitudinal stiffness to a default value and test.
6. Is the mass of the vehicle's rigid body actor too large to be driven by the engine torque? Test that the mass of the actor is a sensible value and set it to a reasonable value.
7. Is the rigid body actor in a PhysX scene and is the scene being updated? Query the actor is not asleep and participates in the scene update.

The Vehicle Does Not Steer/Turn

1. Is the moment of inertia of the vehicle too large so that it resists steering? Check that the moment of inertia of the vehicle's rigid body actor is reasonable. Use the moment of inertia of a box with width/height/length of the vehicle as a guess for the moment of inertia of the actor.
2. Are the steer wheels receiving a steer angle? Check the PxVehicleWheelsDynData::getSteer. If the steer angle is zero when a steering input is expected check that a steer angle is being passed to the vehicle.

maximum steer angles of the steer wheels are sensible values.

3. Do the steer wheels have a sensible lateral slip? Use `PxVehicleWheelsDynData::getLatSlip` to query the slip angle. If `PxVehicleSetBasisVectors` has been called with the correct forward direction, the lateral slips are vanishingly small. Further test that the basis vectors are correct by using `PxVehicleWheelsDynData::getTireLateralDir`.
4. Is the lateral stiffness of the tire configured properly? Reset the default values and retest.

The Acceleration Feels Sluggish

1. Are the damping rates of the engine and wheels too large? First increase the engine damping rate, then the wheel damping rates and retest each time.
2. Is the vehicle stuck in the same gear all the time? Disable the automatic gear shifting to test if the autobox is failing to switch gear. Check the autobox settings to make sure that it will automatically increase the gear rotation speeds.
3. Is the engine powerful enough to quickly accelerate the car? Increase the engine power and retest.
4. Do the wheels have high moments of inertia that prevent significant acceleration? Reduce the moments of inertia of the wheels.

The Vehicle Does Not Slow Down When Not Accelerating

1. Are the wheel and engine damping rates too small? First increase the engine damping rate, then the wheel damping rates and retest each time.
2. Does the vehicle's rigid body actor have a velocity damping value that is not appropriate?

The Vehicle Turns Too Quickly/Too Slowly

1. Does the moment of inertia of the rigid body actor need two component of the moment of inertia that corresponds to motion a Increasing the moment of inertia will slow the turn rate, decrease inertia will increase the turn rate.

The Wheels Spin Too Much Under Acceleration

1. Is the accelerator pedal value increasing too rapidly from 0 to 1? If so, reduce the rate of increase of the accelerator pedal value by filtering the controller. Remember that aggressively pressing the accelerator pedal on a low inertia vehicle is more likely to lead to wheel spin.
2. Are the wheel moments of inertia too low? Increase the wheel moment of inertia.

The Wheels Spin Too Much When Cornering

1. Does the vehicle have a limited slip differential? If applicable set it to limited slip and adjust the differential biases accordingly.

The Vehicle Never Goes Beyond First Gear

1. Does the vehicle cycle between first gear and neutral? If the autobox is the problem is probably that the latency of the autobox is shorter than the time required to perform a gear change. The autobox latency controls the minimum time between automated gear changes. After an automated gear change the autobox will not make another gear change decision until the vehicle has passed the target gear. During a gear change the vehicle enters neutral gear and the accelerator pedal is uncoupled from the engine, meaning that the engine will not be able to perform the gear change. When the vehicle enters the target gear at the end of the gear change the autobox might decide immediately that the engine is still in the target gear and immediately initiate a downwards gear change. This will put the car back in neutral, meaning that the car spends a very long time in neutral.

and never reaches its target gear. This will not happen if the latency (PxVehicleAutoBoxData::setLatency) is set significantly larger than the time (PxVehicleGearsData::mSwitchTime).

The Vehicle Under-steers Then Over-steers

1. Is the vehicle on a bumpy surface? Edit the values in PxVehicleTireData that the filtered normalized tire load has a flatter response to vertical compression.

The Vehicle Slows Down Unnaturally

1. Does the vehicle not slow down smoothly to rest? Take a look at the values to see if they are oscillating between positive and negative. If there is oscillation then two options are available that can be used separately with each other. The first option is to set PxVehicleWheelsSimData::setSubStepCount to force more vehicle simulation steps as the forward speed of the vehicle approaches zero. The second option is to set PxVehicleWheelsSimData::setMinLongSlipDenominator to ensure the denominator of the longitudinal slip never falls below a specified value.

The Vehicle Climbs Too Steep Slopes

1. Are the front wheels slipping badly? Modify PxVehicleTireData::mFriction to reduce the available friction for slipping wheels.

References

Anti-roll Bar

http://www.youtube.com/watch?v=_liGnV3PTiQ http://en.wikipedia.org/roll_bar

Tire Modeling

The default tire model employed by PhysX vehicles is discussed in CarSimEd documentation:

<http://www.eggert.highpeakpress.com/ME485/Docs/CarSimEd.pdf>

PhysX vehicles allow any tire model to be simulated by specifying discussed in *Tire Shaders*. A tire model commonly used in engineering is the Pacejka tire model:

<http://phors.locost7.info/phors21.htm>

<http://phors.locost7.info/phors22.htm>

Engine Torque Curves

http://www.mitsubishi-fuso.com/en/technology/qanda/05_01.html?a5

Differentials

<http://www.howstuffworks.com/differential.htm>

Clutch

<http://auto.howstuffworks.com/clutch.htm>

Ackermann Steer Correction

http://en.wikipedia.org/wiki/Ackermann_steering_geometry

Tanks

<http://www.wikihow.com/Drive-an-Abbot-SPG-%28Tank%29>

General

<http://phors.locost7.info/intro.htm>

<http://www.millikenresearch.com/rcvd.html>

<http://www.sciencedirect.com/science/book/9780750651127>

[NVIDIA PhysX SDK 3.4.1 Documentation »](#)

[previous | next](#)

[User's Guide »](#)

Introduction

The character controller (CCT) SDK is an external component built on the SDK, in a manner similar to PhysXExtensions.

CCTs can be implemented in a number of ways: the PhysX implementation module is only one of them.

By nature, CCTs are often very game-specific, and they can have different features in each game. For example the character's bounding volume can be a sphere in one game, and an inverted pyramid in another. The CCT SDK does not provide a one-size-fits-all solution that would work out-of-the-box for all possible games. It provides the basic features common to all CCTs: character controller, collision, and physics interactions. It is a default starting point for users, a strong base that can be later modified or customized if needed.

Kinematic Character Controller

The PhysX CCT is a kinematic controller. Traditionally, character controllers are either kinematic or dynamic. A kinematic controller directly works with input displacements (1st order control). A dynamic controller works with input velocities (2nd order control) or forces (3rd order control).

In the past, games did not use a 'real' physics engine like the PhysX. Instead, they used a character controller to move a player in a level. These games, even Doom, had a dedicated, customized piece of code to implement movement and response, which was often the only piece of physics in the whole game. There was little physics, but a lot of carefully tweaked values to provide a specific feel for controlling the player. The particular behavior it implemented is often referred to as a 'walk and slide' algorithm, and it has been 'tweaked' for more than a decade. The PhysX module is an implementation of such an algorithm, providing a robust and predictable behavior for character control.

The main advantage of kinematic controllers is that they do not suffer from the issues, which are typical for dynamic controllers:

- (lack of) continuous collision detection: typical physics engines use discrete collision checks, leading to the notorious 'tunneling effect' that has plagued both commercial & non-commercial physics packages for years. This leads to several problems:
 - the tunneling effect itself : if the character goes too fast, it can pass through a wall
 - as a consequence, the character's maximum velocity becomes limited (limiting the game play possibilities)
 - even if it does not tunnel, the character might jitter when pushing a corner for example, because the physics engine keeps moving the character forth to slightly different positions.

- No direct control: a rigid body is typically controlled with impulses usually not possible to move it directly to its final position: instead the delta position vector to impulses/forces, apply them, and hope will end up at the desired position. This does not always work well the physics engine uses an imperfect linear solver.
- Trouble with friction: when the character is standing on a ramp, it s infinite friction is needed here. When the character is moving for ramp, it should not slow down. One does not need any friction he the character is sliding against a wall, it should not slow down eithe friction is usually either 0 or infinite. Unfortunately the friction n engine might not be perfect, and it is easy to end up with either friction (the character slows down a tiny bit) or a very-large-but-not character slides very slowly on that ramp no matter how artificial parameters are). The conflicting requirements for ramps also mean is simply no way to perfectly model desired behavior.
- Trouble with restitution: typically, restitution should be avoided fo character moves fast and collides with a wall, it should not bou When the character falls from a height and lands on the ground, fl bounce should be prevented. But once again, even when the re zero, a physics engine can nonetheless make the CCTs bounce a related to the imperfect nature of the linear solver, it also has to penetration-depth-based engines recover from overlap situations, s excessive forces that separate the objects too much.
- Undesired jumps: characters must often stick to the ground, n physical behavior should be. For example characters in action ga fast, at unrealistic speeds. When they reach the top of a ramp, often makes them jump a bit, in the same way a fast car would ju San Francisco. But that is often not the desired behavior: instead th often stick to the ground regardless of its current velocity. 7 implemented using fixed joints, but this is an unnecessarily con problem that is easily prevented with kinematic controllers.

- Undesired rotations: a typical character is always standing up & However physics engines often have poor support for that sort of great deal of effort is often put into preventing a capsule around falling (it should always stands up on its tip). This is often impleme joints, and the resulting system is neither very robust nor very fast.

To summarize, a lot of effort can be spent on tweaking and disabling th features simply to emulate what is otherwise a much less complex pie It is natural to instead keep using that simple piece of custom code.

Creating a character controller

First, create a controller manager somewhere in your application. This manager keeps track of all created controllers and allows characters from the same manager to interact with each other. Create the manager using the *PxCreateControllerManager* function:

```
PxScene* scene;    // Previously created scene
PxControllerManager* manager = PxCreateControllerManager(*scene);
```

Then, create one controller for each character in the game. At the time of writing, the *PxBBoxController* and *PxCapsuleController* are supported. A controller for a capsule controller for example, is created this way:

```
PxCapsuleControllerDesc desc;
...
<fill the descriptor here>
...
PxController* c = manager->createController(desc);
```

The manager class will keep track of all created controllers. They can be accessed at any time using the following functions:

```
PxU32      PxControllerManager::getNbControllers() const =
PxController* PxControllerManager::getController(PxU32 index) =
```

To release a character controller, simply call its release function:

```
void PxController::release() = 0;
```

To release all created character controllers at once, either release them individually, or use the following function if you intend to keep using the manager:

```
void PxControllerManager::purgeControllers() = 0;
```

The creation of a controller manager and its subsequent controller classes are covered in the *SampleBridges*.

Overlap Recovery Module

Ideally, character should not be created in an initial overlap state, i created in a position where they do not overlap the surrounding geo PScene overlap functions can be used to check the desired volume prior to creating the character. By default the CCT module does not itself, and creating a character that initially overlaps the world's static (undesired and undefined behavior - like the character going throu example.

However, the overlap recovery module can be used to automa character's initial position. As long as the amount of overlap is reaso module should be able to relocate the character to a proper, collision-fre

The overlap recovery module can be useful in several other situation main cases:

- when the CCT is directly spawned or teleported in another object
- when the CCT algorithm fails due to limited FPU accuracy
- when the "up vector" is modified, making the rotated CCT shape c objects

When activated, the CCT module will automatically try to resolve th move the CCT to a safe place where it does not overlap other objects concerns static objects, dynamic objects are ignored by this module.

Enable or disable the overlap recovery module with this function:

```
void PxControllerManager::setOverlapRecoveryModule(bool flag);
```

By default the character controllers use precise sweep tests, whose enough to avoid all penetration - provided the contact offset is not too s cases the overlap recovery module is not needed. When it is used thou can be switched to less accurate but potentially faster versions, u

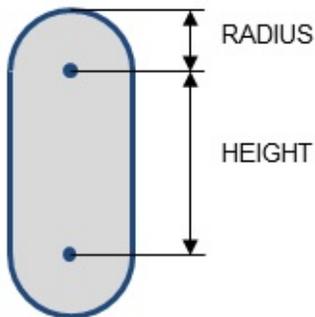
function:

```
void PxControllerManager::setPreciseSweeps(bool flag);
```

Character Volume

The character uses a bounding volume that is independent from area in the SDK. We currently support two different shapes around the character

- An AABB, defined by a center position and an extents vector. It does not rotate. It always has a fixed rotation even when the player is (view) avoids getting stuck in places too tight to let the AABB rotate.
- A capsule, defined by a center position, a vertical height and a radius. The distance between the two sphere centers at the end of the capsule has a better behavior when climbing stairs, for example. It is the default choice.



Note: versions prior to 2.3 also supported a sphere. This has been replaced by a capsule (PxCapsuleController is more robust and provides the same functionality as PxCapsuleController).

A small skin is maintained around the character's volume, to avoid numerical errors that would otherwise happen when the character touches other shapes. The skin is user-defined. When rendering the character's volume for debug purposes, you should expand the volume by the size of this skin to get accurate debug visualizations. This is defined in `PxControllerDesc::contactOffset` and later available in `PxController::getContactOffset()` function.

Volume Update

Sometimes it is useful to change the size of the character's volume. For example, if the character can crouch, it might be required to reduce its bounding volume so that it can then move to places he could not reach.

For the box controller, the related functions are:

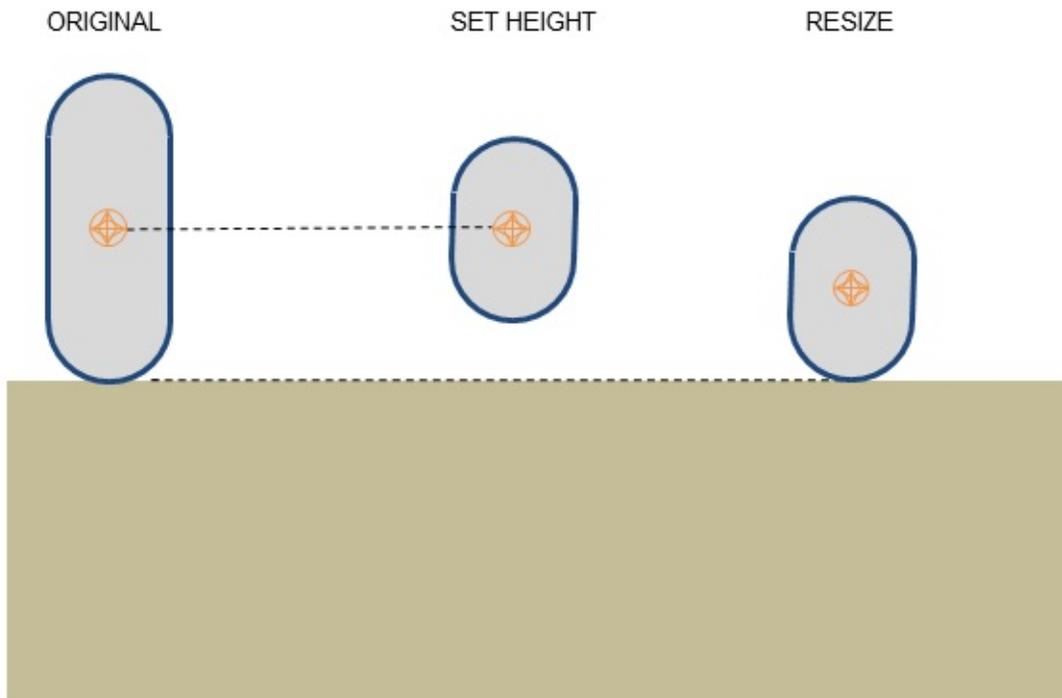
```
bool PxBoundingBoxController::setHalfHeight(PxF32 halfHeight)
bool PxBoundingBoxController::setHalfSideExtent(PxF32 halfSideExtent)
bool PxBoundingBoxController::setHalfForwardExtent(PxF32 halfForwardExtent)
```

And for the capsule controller:

```
bool PxCapsuleController::setRadius(PxF32 radius) = 0;
bool PxCapsuleController::setHeight(PxF32 height) = 0;
```

Changing the size of a controller using the above functions does not change its position. So if the character is standing on the ground (touching it) and its bounding volume is suddenly reduced without updating its position, the character will end up penetrating the ground for a few frames until gravity makes it fall and touch the ground again. This happens because the controller's position is located at the center of the bounding volume, not at the bottom. Thus, to modify a controller's height and preserve its position, you must change both the height and position of a controller. The following function does that automatically:

```
void PxController::resize(PxF32 height) = 0;
```



It is important to note that volumes are directly modified without any external function. However, it might happen that the resulting volume overlaps some geometry near the ground. When resizing the character to leave a crouch pose, i.e. when the size is *increased*, it is important to first check that the character can indeed 'stand up'. The space above the character must be empty (collision free). It is recommended to use various PxScene overlap queries for this purpose:

```
bool PxScene::overlap(...) = 0;
```

Updating the character's volume at runtime to implement a 'crouch' motion. Using overlap queries to leave the crouch pose. See `SampleNorthPole::tryStandup()` function.

Moving a Character Controller

The heart of the CCT algorithm is the function that actually moves characters.

```
PxControllerCollisionFlags collisionFlags =  
    PxController::move(const PxVec3& disp, PxU32 minDist, PxU32 e  
    const PxControllerFilters& filters, const PxObstacleContext*
```

disp is the displacement vector for current frame. It is typically a combination of motion due to gravity and lateral motion when your character is moving. Gravity and lateral motion are responsible for applying gravity to characters here.

minDist is a minimal length used to stop the recursive displacement algorithm. If the remaining distance to travel goes below this limit.

elapsedTime is the amount of time that passed since the last call to the function.

filters are filtering parameters similar to the ones used in the SDK. They define what the character should collide with.

obstacles are optional additional obstacle objects with which the character can collide. Those objects are fully controlled by users and do not need to have a physics component. Note that touched obstacles are cached, meaning that they are not invalidated if the collection of obstacles changes.

collisionFlags is a bit mask returned to users to define collision events that occur during the move. This is a combination of `PxControllerCollisionFlag` flags used to trigger various character animations. For example your character might be playing a falling idle animation, and you might start the land animation when `PxControllerCollisionFlag::eCOLLISION_DOWN` is returned.

It is important to understand the difference between `PxController::setPosition` and `PxController::move`. The `PxController::move` function is the core of the CCT algorithm. This is where the aforementioned 'collide-and-slide' algorithm takes place. The move function will start from the CCT's current position, and use sweep tests to attempt to move the character to the target position.

required direction. If obstacles are found, it may make the CCT slide them. Or the CCT can get blocked against a wall: the result of the motion is determined by the surrounding geometry. On the contrary, *PxController::setPosition* is a function that will move the CCT to desired position no matter what, regardless of where the CCT starts from, regardless of surrounding geometry, and even if it is in the middle of another object.

Both *PxController::move* and *PxController::setPosition* are available in `SampleBridges`.

Graphics Update

Each frame, after *PxController::move* calls, graphics object must be kept new CCT positions. Controllers' positions can be accessed using:

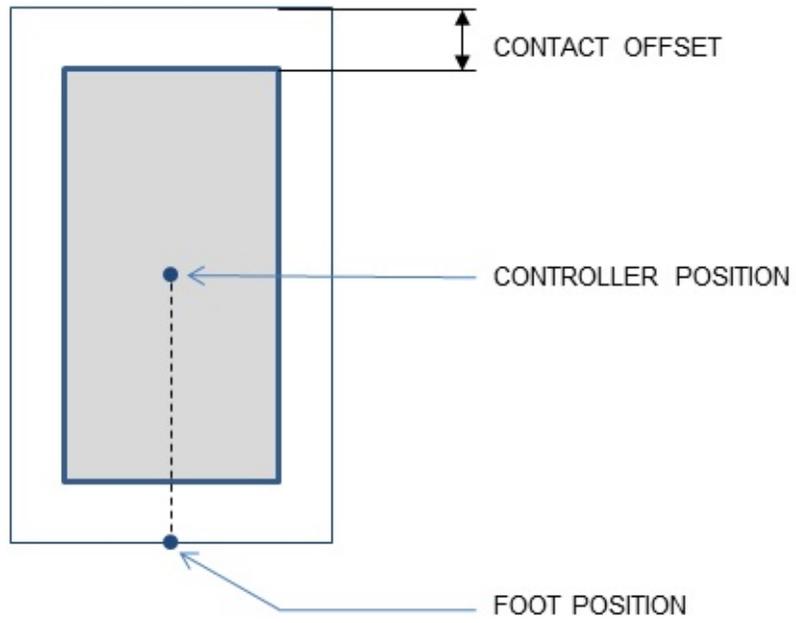
```
const PxExtendedVec3& PxController::getPosition() const;
```

This function returns the position from the center of the collision shape is used internally both within the PhysX SDK and by usual graphics API position and passing it to the renderer is illustrated in SampleBridge position uses double-precision, to make the CCT module work well with note that a controller never rotates so you can only access its position.

Alternative helper functions are provided to work using the character a.k.a. the foot position:

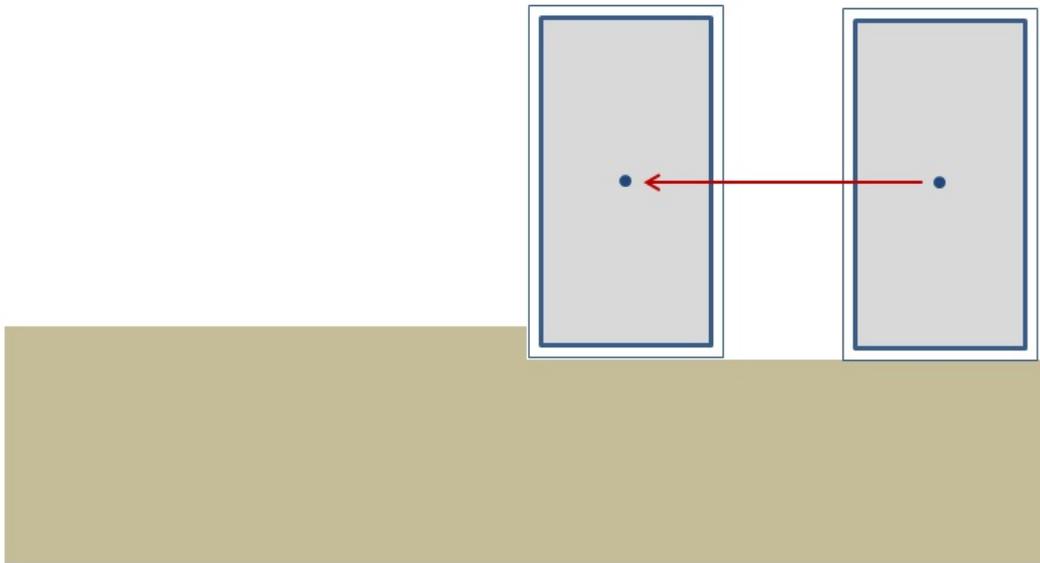
```
const PxExtendedVec3& PxController::getFootPosition() const;  
bool PxController::setFootPosition(const PxExtendedVec3& pos);
```

Note that the foot position takes the contact offset into account.

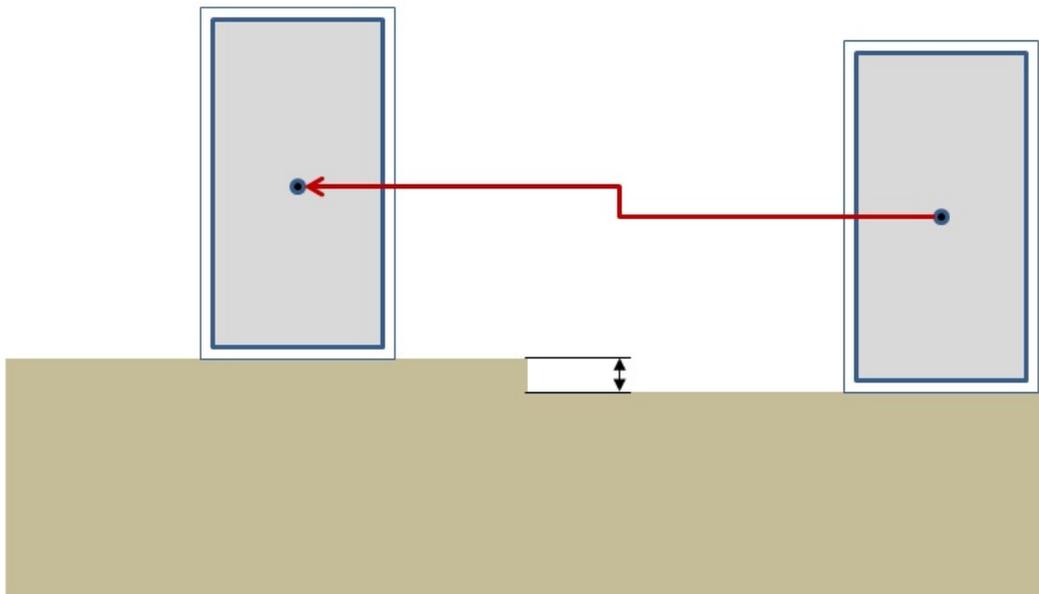


Auto Stepping

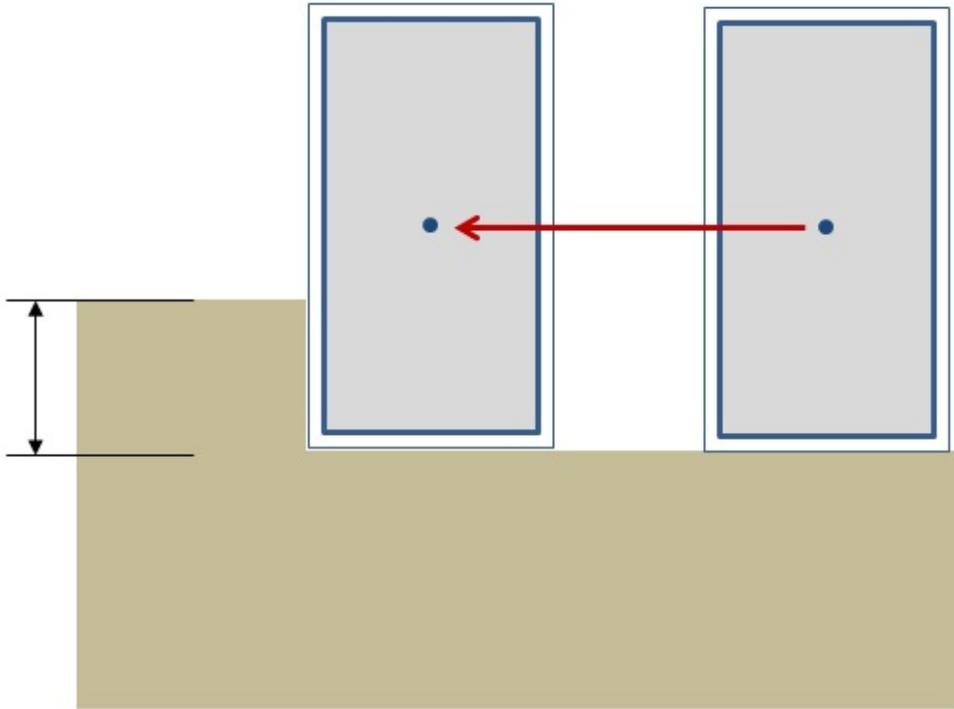
Without auto-stepping it is easy for a box-controlled character to get stuck on small elevations of the ground mesh. In the following picture the small step character completely. It feels unnatural because in the real world a character would cross this small obstacle without thinking about it.



This is what auto-stepping enables us to do. Without any intervention from the user (or without them thinking about it) the box correctly steps above the minor elevation.



However, if the obstacle is too big, i.e. its height is greater than the *ste* the controller cannot climb automatically, and the character gets stuck (



'Climbing' (over this bigger obstacle, for example) may also be implemented as an extension of auto-stepping. The step offset *PxControllerDesc::stepOffset* and later available through the *PxControl* function.

Generally speaking, the step offset should be kept as small as possible.

Climbing Mode

The auto-stepping feature was originally intended for box controllers blocked by small obstacles on the ground. Capsule controllers, thank nature, do not necessarily need the feature.

Even with a step offset of 0.0, capsules are able to go over small obstacles. A rounded bottom produces an upward motion after colliding with a small

Capsules with a non-zero step-offset can go over obstacles higher than their radius because of the combined effect of the auto-stepping feature and their velocity. In this case the largest altitude a capsule can climb over is difficult to predict. It depends on the auto-step value, the capsule's radius, and even the magnitude of the velocity vector.

This is why there are two different climbing modes for capsules:

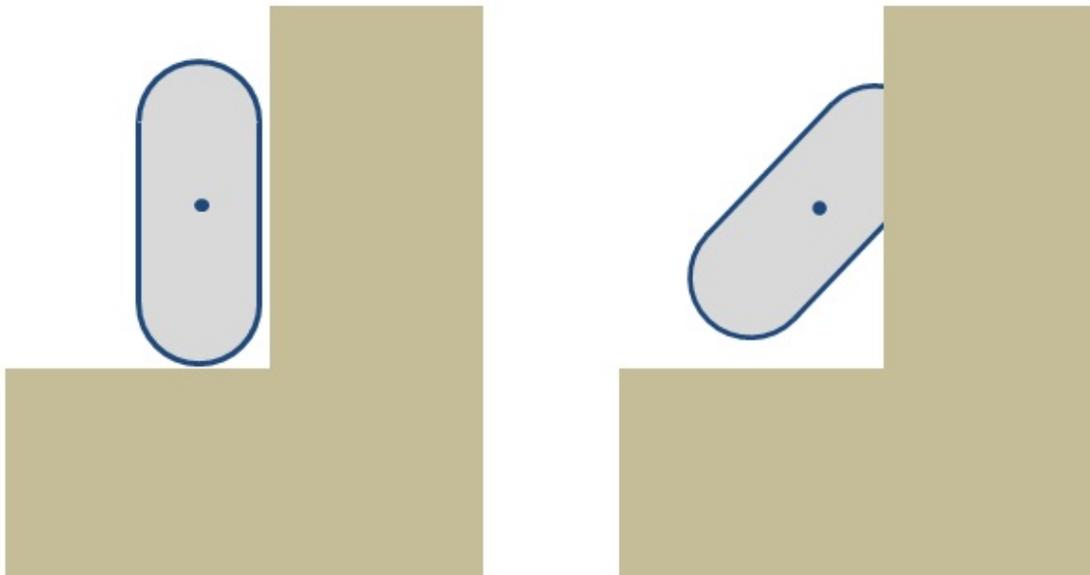
- *PxCapsuleClimbingMode::eEASY*: in this mode, capsules are not limited by the step offset value. They can potentially climb over obstacles higher than their radius.
- *PxCapsuleClimbingMode::eCONSTRAINED*: in this mode, an auto-step value is used to make sure the capsule can not climb over obstacles higher than the auto-step value.

Up Vector

In order to implement the auto-stepping feature, the SDK needs to know the up vector. The up vector is defined in *PxControllerDesc::upDirection* and accessed through the *PxController::getUpDirection()* function.

The up vector does not need to be axis-aligned. It can be arbitrary, modified using the *PxController::setUpDirection()* function, allowing the character to walk on spherical worlds. This is demonstrated in *SampleCustomGravity*.

Modifying the up vector changes the way the CCT library sees character geometry. For example a capsule is defined by a *PxCapsuleControllerDesc::height*, which is the 'height' along the up vector. Thus, changing the up vector effectively changes the height of the capsule from the point of view of the library. The modification happens immediately after the call. It is the responsibility of the user to validate that the character does not overlap nearby geometry. It is the responsibility of the user to ensure that the character is not penetrating some geometry right after the call. Using the *SampleCustomGravity* module is recommended to solve these issues.



In the above picture the capsule on the left uses a vertical up vector a with the surrounding geometry. On the right the up vector has been set the capsule now penetrates the wall nearby. For most applications th constant, and the same for all characters. These issues will only app navigating in spherical worlds (e.g. planetoids, etc).

Walkable Parts & Invisible Walls

By default the characters can move everywhere. This may not always be particular, it is often desired to prevent walking on polygons whose slope is higher than a user-defined slope limit. The `PxController` SDK can do this automatically thanks to a user-defined slope limit. If a polygon's slope is higher than the limit slope will be marked as non walk-able, and characters are not allowed to go there.

Two modes are available to define what happens when touching a non-walkable surface. The desired mode is selected with the `PxControllerDesc::nonWalkableMode` parameter.

- `PxControllerNonWalkableMode::ePREVENT_CLIMBING` prevents moving up a slope, but does not move the character otherwise. This allows the character to be able to walk laterally on these polygons, and to move down their slopes.
- `PxControllerNonWalkableMode::ePREVENT_CLIMBING_AND_FALL` not only prevents the character from moving up non walk-able slopes, but also makes it to slide down those slopes.

The slope limit is defined in `PxControllerDesc::slopeLimit` and later available in `PxController::getSlopeLimit()` function. The limit is expressed as the cosine of the angle. For example this uses a slope limit of 45 degrees:

```
slopeLimit = cosf(PxMath::degToRad(45.0f));
```

Using `slopeLimit = 0.0f` automatically disables the feature (i.e. characters can move everywhere).

This feature is not always needed. A common strategy is to disable it to create invisible walls in the level, to restrict player's movements. The character module can be used to create those walls for you, if `PxControllerDesc::invisibleWallHeight` is non-zero. The `PxController` library creates those extra triangles on the fly, and that parameter controls their height (extruded in the user-defined up direction). A common problem is that triangles are only created when non-walkable triangles are found. It is possible to create them in advance.

character to go over them, if its bounding volume is too small and does not detect non-walkable triangles below him. The *PxControllerDesc::maxJump* parameter addresses this issue, by extending the size of the bounding volume used for collision. This ensures that all potentially non-walkable triangles are properly returned by the collision system and that invisible walls are properly created - preventing the character from jumping over them.

A known limitation is that the slope limit mechanism is currently only enabled against static objects. It is not enabled against dynamic objects, and in particular against dynamic spheres or dynamic capsules. It is also not supported for static spheres or static capsules.

Obstacle Objects

Sometimes it is convenient to create additional obstacles for the C without creating an actual SDK object. This is useful in a number example:

- the obstacles might only exist for a couple of frames, in which deleting SDK objects is not always efficient.
- the obstacles might only exist for stopping the characters, not t objects. This would be for example invisible walls around geom characters should collide with. In this case it may not be very eff invisible walls as SDK objects, since their interactions would then out for everything except the characters. It is probably more effici additional invisible walls as external obstacles, that only characters
- the obstacles might be dynamic and updated with a variable times uses a fixed timestep. This could be for example a moving plat characters can stand.

At the time of writing the character controller supports box and c objects, namely *PxBoxObstacle* and *PxCapsuleObstacle*. To create th *PxObstacleContext* object using the following function:

```
PxObstacleContext* PxControllerManager::createObstacleContext()
```

Then manage obstacles with:

```
ObstacleHandle PxObstacleContext::addObstacle(const PxObstacle& o  
bool PxObstacleContext::removeObstacle(ObstacleHandle handle) = 0  
bool PxObstacleContext::updateObstacle(ObstacleHandle handle, con
```

Typically *updateObstacle* is called right before the controllers' *move* call

Using obstacles for moving platforms is illustrated in Sam

PLATFORMS_AS_OBSTACLES is defined in SampleBridgesSettings.h

Hit Callback

The *PxUserControllerHitReport* object is used to retrieve some controller's evolution. In particular, it is called when a character hits character, or a user-defined obstacle object.

When the character hits a shape, the *PxUserControllerHitReport::onShapeHit* callback is invoked - for both static and dynamic shapes. Various impact parameters are passed to the callback, and they can then be used to do various things like playing trails, applying forces, and so on. The use of *PxUserControllerHitReport* is illustrated in SampleBridges. Note that this callback will only be called if a character is moving against a shape. It will *not* be called if a character is moving against an otherwise non-moving character. In other words, this will only be called after a *PxController::move* call.

When the character hits another character, i.e. another object controller, the *PxUserControllerHitReport::onControllerHit* callback happens when the player collides with an NPC, for example.

Finally, when the character hits a user-defined obstacle, the *PxUserControllerHitReport::onObstacleHit* callback is invoked.

Behavior Callback

The *PxControllerBehaviorCallback* object is used to customize the cl after touching a *PxShape*, a *PxController*, or a *PxObstacle*. This following functions:

```
PxControllerBehaviorFlags PxControllerBehaviorCallback::getBehavi
    (const PxShape& shape, const PxActor& actor) = 0;
PxControllerBehaviorFlags PxControllerBehaviorCallback::getBehavi
    (const PxController& controller) = 0;
PxControllerBehaviorFlags PxControllerBehaviorCallback::getBehavi
    (const PxObstacle& obstacle) = 0;
```

At the time of writing the following returned flags are supported:

PxControllerBehaviorFlag::eCCT_CAN_RIDE_ON_OBJECT defines if effectively travel with the object it is standing on. For example a character dynamic bridge should follow the motion of the *PxShape* it is standing on (e.g. *SampleBridges*). But it should not be the case if the character stands on a bottle rolling on the ground (e.g. the snowballs in *SampleNorthPole*). This flag only controls the horizontal displacement communicated from an object. The vertical motion is something slightly different, as many factors are taken into account: the *step offset* used to automatically walk over small steps, the motion of underlying dynamic actors like e.g. the bridges in *SampleBridges* probably always been taken into account, etc.

PxControllerBehaviorFlag::eCCT_SLIDE defines if the character should slide when standing on the object. This can be used as an alternative to the *pr* slope limit feature, to define non walk-able objects rather than non-walk-able. It can also be used to make a capsule character fall off a platform's edge automatically when the center of the capsule crosses the platform's edge.

PxControllerBehaviorFlag::eCCT_USER_DEFINED_RIDE simply disables the feature related to controllers riding on objects. This can be useful to get the leg of a character which can sometimes be necessary when porting to PhysX 3.x a platform

around the PhysX 2.x character controller. The flag simply skips the r
lets users deal with this particular problem in their own application, c
library.

The behavior callback is demonstrated in SampleBridges.

Character Interactions: CCT-vs-dynamic actors

It is tempting to let the physics engine push dynamic objects by applying forces at contact points. However it is often not a very convincing solution.

The bounding volumes around characters are artificial (boxes, capsules) so the forces computed by the physics engine between a bounding volume and surrounding objects will not be realistic anyway. They will not provide a realistic interaction between an actual character and these objects. If the bounding volume is compared to the visible character, maybe to make sure that its limbs rest on the static geometry around, the dynamic objects will start moving (pushing the bounding volume) before the actual character touches them - making it look like the character is surrounded by some kind of force field.

Additionally, the pushing effect should not change when switching from a capsule controller. It should ideally be independent from the bounding volume.

Pushing effects are usually dictated by gameplay, and sometimes require inverse kinematic solvers, which are outside of the scope of the CCT. For simple use cases, it is for example difficult to push a dynamic box forward with a capsule controller: since the capsule never hits the box exactly in the middle, it will also rotate the box - even if gameplay dictates that it should move in a straight line.

Thus, this is an area where the CCT module should best be coupled to the game code, to implement a specific solution for a specific game. This can be done in many different ways. For simple use cases it is enough to use the `PxUserControllerHitReport::onShapeHit` callback to apply artificial forces to dynamic objects. Such an approach is illustrated in `SampleBridges`.

Note that the character controller does use overlap queries to determine nearby shapes. Thus, SDK shapes that should interact with the characters (either the character should push or be pushed) must have the `PxShapeFlag::eSCENE_QUERY_SHAPE` set to true, otherwise the CCT will not detect them and characters will not interact with these shapes.

Character Interactions: CCT-vs-CCT

The interactions between CCTs (i.e. between two `PxController` objects) in this case both objects are effectively kinematic objects. In other words, they should be fully controlled by users, and neither the PhysX SDK nor the user should be allowed to move them.

The `PxControllerFilterCallback` object is used to define basic interaction rules for characters. Its `PxControllerFilterCallback::filter` function can be used to define whether two `PxController` objects should collide at all with each other:

```
bool PxControllerFilterCallback::filter(const PxController& a, const PxController& b)
```

To make CCTs always collide-and-slide against each other, simply return `true`.

To make CCTs always move freely through each other, simply return `false`.

Otherwise, customized and maybe gameplay-driven filtering rules can be implemented in this callback. Sometimes the filtering changes at runtime, and two characters are allowed to go through each other only for a limited amount of time. When this time expires, the characters may be left in an overlapping state until they are separated again towards each other. To automatically separate overlapping characters, the `PxControllerManager::computeInteractions` function can be used:

```
void PxControllerManager::computeInteractions(PxF32 elapsedTime, PxControllerFilterCallback* cctFilterCb=NULL) = 0;
```

This function is an optional helper to properly resolve overlaps between characters. It should be called once per frame, before the `PxController::move` calls. To move the characters directly, but it will compute overlap information for each pair of characters which will be used in the next `PxController::move` call.

Hidden Kinematic Actors

The CCT library creates a kinematic actor under the hood, for each controller. When invoking the *PxController::move* function, the underlying hidden actor is also updated to reflect the CCT position in the physics scene.

Users should be aware of these hidden entities, since the total number of actors in the scene will be higher than the number they created themselves. Additionally, it is important to filter back these potentially confusing unknown actors from scene-level collision reports.

One possible strategy is to retrieve the controllers' kinematic actors using the *PxController::getActor* function:

```
PxRigidBody* PxController::getActor() const;
```

Then mark these actors with a special tag, using the *PxRigidBody::setTag* function. This way the CCT actors can easily be identified (and possibly ignored) in contact reports.

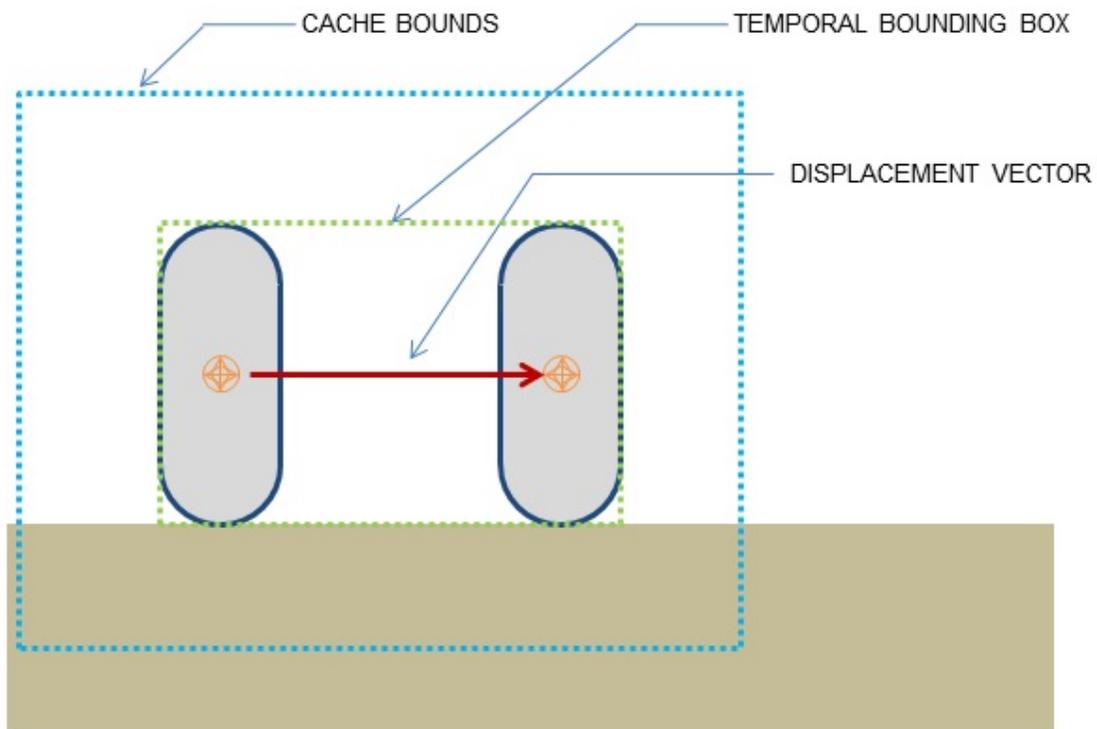
Time Stepping

Actors used internally by the CCT library follow the same rules as objects. In particular, they are updated using fixed or variable times. This is troublesome because the PxController objects are otherwise often updated at fixed time steps (typically using the elapsed time between two rendering frames).

Thus the PxController objects (using variable time steps) may not always sync with their kinematic actors (using fixed time steps). This phenomenon is addressed by the SampleBridges.

Invalidating Internal Geometry Caches

The CCT library caches the geometry around each character, in collision queries. The temporal bounding box for a character is an approximation of the character's motion (it contains the character's volume at both its start and end positions). The cached volume of space is determined by the size of the character's temporal bounding box, multiplied by a constant factor. This constant factor is determined for each character by *PxControllerDesc::volumeGrowth*. Each time a character's temporal bounding box is tested against the cached volume of space. If the motion is detected within that volume of space, the contents of the cache are reused instead of performing through PxScene-level queries.



In PhysX 3.3 and above, those caches should be automatically invalidated when an object gets updated or removed. However it is also possible to manually invalidate caches using the following function:

```
void PxController::invalidateCache();
```

Prior to deciding if a character will travel with the motion of an object it touches, a number of tests are automatically performed to decide if the touched object remains valid. These automatic validity tests mean that in the future it is not strictly necessary to invalidate the cache:

- If the shapes actor is released
- If the shape is released
- If the shape is removed from an actor
- If an actor is removed from scene or moved to another one
- If the shapes scene query flag changed
- If the filtering parameters of the shape or the scene have changed.

If a cached touched object is no longer actually touching the character, then the character no longer travels with the motion of that cached object. It is not strictly necessary to invalidate the cache. This holds true if the pair has the consequence of an updated global pose or modified geometry.

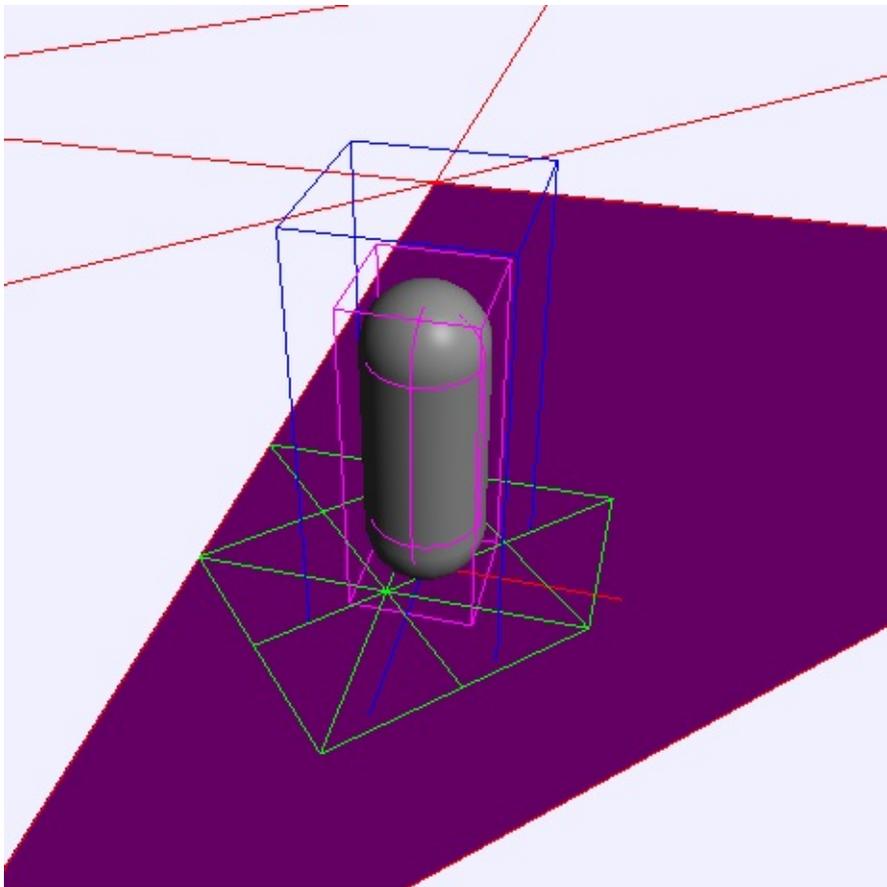
Runtime Tessellation

The CCT library is quite robust, but sometimes suffers from FPU accuracy issues where a character collides against large triangles. This can lead to characters reacting incorrectly against those triangles, or even penetrating them. One way to solve these problems is to tessellate the large triangles at runtime, replacing them with a collection of smaller triangles. The library supports a built-in tessellation feature with this function:

```
void PxControllerManager::setTessellation(bool flag, float maxEdgeLength)
```

The first parameter enables or disables the feature. The second parameter is the maximum allowed edge length for a triangle, before it gets tessellated. A smaller edge length leads to more triangles being created at runtime. The more triangles get generated, the slower it is to collide against them.

It is thus recommended to disable the feature at first, and only enable it when you encounter collision problems. When enabling the feature, it is recommended to use a reasonable *maxEdgeLength* that does fix encountered problems.



In the screenshot, the large magenta triangle on which the character is replaced with the smaller green triangles by the tessellation module geometry cache is represented by the blue bounding box. Note that triangles touching this volume of space are kept. Thus, the exact number produced by the tessellation code depends on both the *maxEdgeLen* and the *PxControllerDesc::volumeGrowth* parameter.

Troubleshooting

This section introduces common solutions to common problems with the

Character goes through walls in rare cases

1. Try increasing *PxControllerDesc::contactOffset*.
2. Try enabling runtime tessellation with *PxControllerManager::setTessellation* a small *maxEdgeLength* first, to see if it solves the problem. Then increase it as much as possible.
3. Try enabling overlap recovery with *PxControllerManager::setOverlapRecoveryModule*.

Tessellation performance issue

1. Try fine-tuning the *maxEdgeLength* parameter. Use the largest possible value to prevent tunneling issues.
2. Try reducing *PxControllerDesc::volumeGrowth*.

The capsule controller manages to climb over obstacles h step offset value

1. Try using *PxCapsuleClimbingMode::eCONSTRAINED*.

Introduction

The PhysX particle feature has been deprecated in PhysX standalone library PhysX FleX is an alternative with a richer feature set.

PhysX 3 offers two particle system types - a generic particle system and a fluid particle system. The generic particle system provides basic particle motion and collision with rigid actors. It can be used for objects that require collisions again: but for which inter-particle interactions are not needed. Examples include sparks or leaves. The SPH fluid particle system can be used for fluid simulation with approximate incompressibility and flowing behavior, such as liquids filling up a volume.

PhysX 3 takes care of collision detection and particle dynamics, while features such as emitters, lifetime maintenance etc. need to be provided by the user.

Creating Particle Systems

Both particle system classes *PxParticleSystem* and *PxParticleFluid* inherit from *PxParticleBase*, which is the common interface providing particle collision functionality. Particle systems inherit from *PxActor* and can be

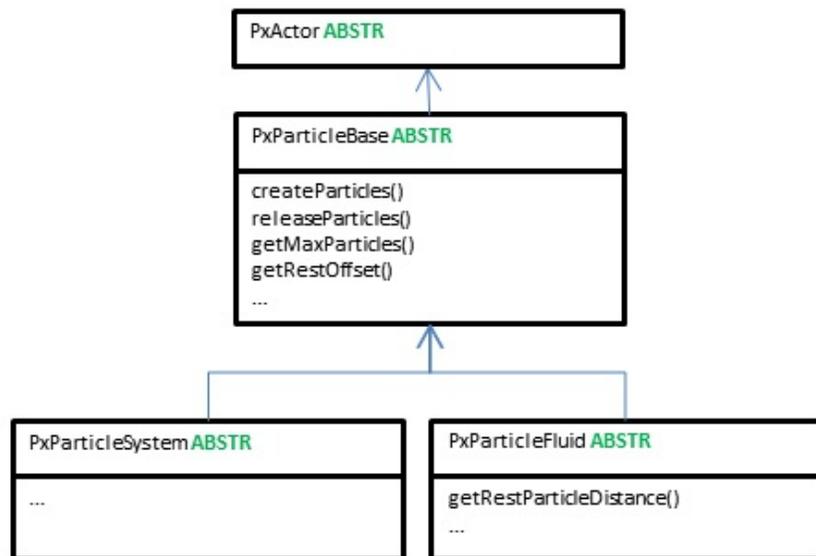


Figure 1: PxParticleSystem inherits all properties from PxParticleBase
PxParticleFluid adds fluid specific properties

The following section shows how a particle system is created and added to a scene.

```
// set immutable properties.
PxU32 maxParticles = 100;
bool perParticleRestOffset = false;

// create particle system in PhysX SDK
PxParticleSystem* ps = mPhysics->createParticleSystem(maxParticleCount, perParticleRestOffset);

// add particle system to scene, in case creation was successful
if (ps)
    mScene->addActor(*ps);
```

Note: The particle module has to be registered with *PxRegisterParticle* with static linking (non windows) before creating particle systems. *PxC* registers all modules by default as opposed to *PxCreateBasePhysics*.

Particle Management

Particle systems reserve memory for a fixed number `PxParticleBase::getMaxParticles`. Each of these particles can be addressed throughout its lifetime. The given range of `PxParticleBase::getMaxParticles`. In order to support a dynamic array of particles, particles are marked as being valid or invalid. This is achieved by two data structures: a valid particle range and a valid flags array. The valid particle range indicates the range within which particles may be valid. Initially, all particles are defined as being invalid. Within that range, valid particles are marked with the flag `PxParticleFlag::eVALID`. Alternatively, PhysX provides a bitmap corresponding to a valid particle within the valid particle range. The bitmap is an array of 32-bit unsigned integers with enough elements to cover the valid particle range.

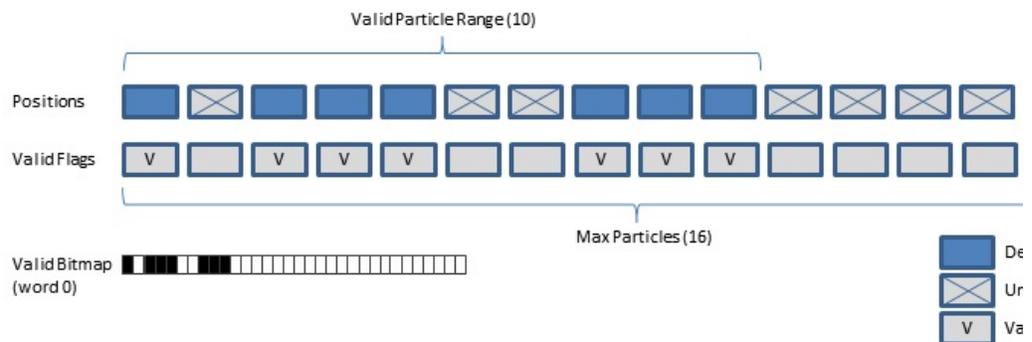


Figure 2: Scheme showing how valid particles are tracked

Creating Particles

The application specifies an index for each new particle at particle creation. The application maintains its own representation of particles, and already tracked indices may be re-used by PhysX. If the application does not track indices at its disposal, it can use an index pool provided by the PhysX `PxParticleExt::IndexPool` as explained here: [Index Pool Extension](#).

PhysX 3 itself has no built-in emitters. Instead, it simply provides an interface for creating particles with initial properties. When creating particles, specifying indices

mandatory, while velocities and rest offsets may be specified optionally.

The PhysX particle API uses the `PxStrideIterator` template class to pass data between the SDK and the application. This allows the particle data layout to be more flexible by supporting interleaved arrays or padded data without the need for reformatting. The stride iterator is configured by setting the type and specifying the pointer to the first element.

Example for creating a few particles:

```
// declare particle descriptor for creating new particles
// based on numNewAppParticles count and newAppParticleIndices,
// newAppParticlePositions arrays and newAppParticleVelocity
PxParticleCreationData particleCreationData;
particleCreationData.numParticles = numNewAppParticles;
particleCreationData.indexBuffer = PxStrideIterator<const PxU32>(
particleCreationData.positionBuffer = PxStrideIterator<const PxVec3>(
particleCreationData.velocityBuffer = PxStrideIterator<const PxVec3>(

// create particles in *PxParticleSystem* ps
bool success = ps->createParticles(particleCreationData);
```

The indices specified for particle creation need to be unique and less than `PxParticleBase::getMaxParticles()`.

In this example the stride iterator is used to set the same velocity for all particles. This is achieved by setting the stride to zero.

Note: For fluid particles it is necessary to spawn particles at distances greater than `PxParticleFluid::getRestParticleDistance()` in order to achieve a regular distribution. Otherwise particles will spread immediately in all directions.

Note: In PhysX 3 all particle access such as creating, releasing, updating particles can only be carried out while the simulation of the scene is not sleeping.

Releasing Particles

Particles can be released by providing indices to the particle system. All versions of the PhysX SDK, particles get immediately released.

Example for releasing a few particles:

```
// declare strided iterator for providing array of indices corresponding to
// particles that should be removed
PxStrideIterator<const PxU32> indexBuffer(appParticleIndices);

// release particles in *PxParticleSystem* ps
ps->releaseParticles(numAppParticleIndices, indexBuffer);
```

It is a requirement that the indices passed to the release method correspond to existing particles.

All particles can be released at once by calling:

```
ps->releaseParticles();
```

Since only a limited number of particle slots (*PxParticleBase::getMaxParticleCount*) are available it might be appropriate to replace old particles with new ones. This can be achieved for instance by maintaining an application-side particle lifetime and releasing particles for the following reasons:

- Drains can be useful to remove particles that go to locations where they are no longer needed anymore. See [Particle Drains](#).
- The spatial data structure used for particles may overflow. Particles that are no longer covered are marked and should be released. See [Particle Grid](#).

Index Pool Extension

Example for allocating particle indices using the PhysX extensions library:

```
// create an index pool for a particle system with maximum particle count
PxParticleExt::IndexPool* indexPool = PxParticleExt::createIndexPool(maxParticleCount);

// use the indexPool for allocating numNewAppParticles indices throughout the simulation
```

```

// for particle creation throughout the particle system lifetime.
// is smaller than numNewAppParticles, the maxParticles limit was
PxU32 numAllocated = indexPool->allocateIndices(numNewAppParticle
                                                PxStrideIterator<

// in order to reuse particle slots, the indices should be handed
// indexPool after the particles have been released
indexPool->freeIndices(numAppParticleIndices, PxStrideIterator<Px

// if no further index management is needed, the pool should be r
indexPool->release();

```

Updating Particles

The following per-particle updates are carried out immediately:

- Position updates: Teleporting particles from one location to another
- Velocity updates: Directly altering the velocities of particles.
- Rest offset updates: Changes particle rest offsets (only *PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET*).

Particle updates that are carried out during the next scene simulation step:

- Force updates: Results in a velocity change update according to the force mode specified by *PxForceMode*.

Example for force update:

```

// specify strided iterator to provide update forces
PxStrideIterator<const PxVec3> forceBuffer(appParticleForces);

// specify strided iterator to provide indices of particles that
PxStrideIterator<const PxU32> indexBuffer(appParticleForceIndices

// specify force update on PxParticleSystem ps choosing the "force
ps->addForces(numAppParticleForces, indexBuffer, forceBuffer, PxF

```

Reading Particles

The PhysX SDK does not provide to the user all simulated per-particle data by default. The application can specify the data it needs by setting the `PxParticleBase::particleReadDataFlags`:

- `PxParticleReadDataFlag::ePOSITION_BUFFER`: On by default.
- `PxParticleReadDataFlag::eFLAGS_BUFFER`: On by default.
- `PxParticleReadDataFlag::eVELOCITY_BUFFER`: Off by default.
- `PxParticleReadDataFlag::eREST_OFFSET_BUFFER`: Off by default, but can be enabled if the particle system was created with per particle rest offsets (see [Creating Particle Systems](#)).
- `PxParticleReadDataFlag::eCOLLISION_NORMAL_BUFFER`: Off by default.
- `PxParticleReadDataFlag::eDENSITY_BUFFER`: Only available for fluid particles, off by default.

Particle flags provide more information on individual particles:

- `PxParticleFlag::eVALID`: If set, the particle was created before the simulation started. If not set, the particle slot does not contain a valid particle. All properties are invalid in this case and should be ignored.
- `PxParticleFlag::eCOLLISION_WITH_STATIC`: Shows whether a particle collided with a rigid static during the last simulation step.
- `PxParticleFlag::eCOLLISION_WITH_DYNAMIC`: Shows whether a particle collided with a dynamic rigid body during the last simulation step.
- `PxParticleFlag::eCOLLISION_WITH_DRAIN`: Shows whether a particle collided with a rigid actor shape that was marked as a drain ([Particle Drains](#)).
- `PxParticleFlag::eSPATIAL_DATA_STRUCTURE_OVERFLOW`: Shows whether a particle had to be omitted when building the SDK internal spatial structure ([Particle Grid](#)).

Particle collision normals represent contact normals between particles.

surfaces. A non-colliding particle has a zero collision normal. Collision e.g. for orienting the particle visualization according to their contact with

Particle densities provided by particle fluids can be used for rendering has a value of zero for a particle that is completely isolated. It has a particle that has a particle neighborhood with a mean spacing *PxParticleFluid::getRestParticleDistance()*.

Particle data can only be read while the scene simulation is not executed. To get access to the SDK buffers a *PxParticleReadData* instance needs to be created in the SDK. It has the following properties:

- *numValidParticles*: Total number of valid particles for the current system.
- *validParticleRange*: The index range of valid particles in the particle system.
- *validParticleBitmap*: Bitmap of valid particle locations.
- *positionBuffer*, *positionBuffer*, *velocityBuffer*, *restOffsetBuffer*, *collisionNormalBuffer*: Strided iterators for particle properties.

Additionally particle fluids provide *PxParticleFluidReadData* with

- *densityBuffer*: Strided iterator for particle densities.

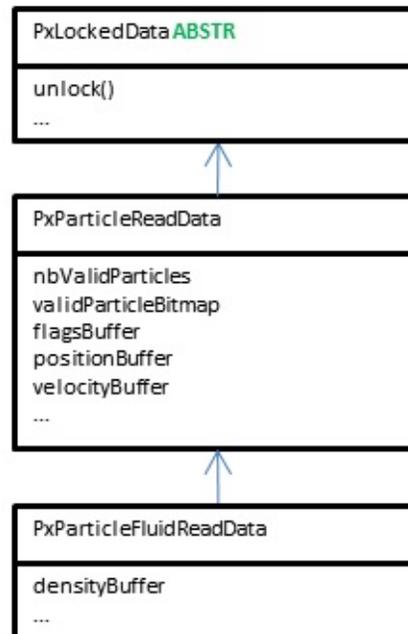


Figure 3: PxParticleReadData and PxParticleFluidReadData

Example of how to access particle data:

```

// lock SDK buffers of *PxParticleSystem* ps for reading
PxParticleReadData* rd = ps->lockParticleReadData();

// access particle data from PxParticleReadData
if (rd)
{
    PxStrideIterator<const PxParticleFlags> flagsIt(rd->flagsBuff
    PxStrideIterator<const PxVec3> positionIt(rd->positionBuffer)

    for (unsigned i = 0; i < rd->validParticleRange; ++i, ++flags
    {
        if (*flagsIt & PxParticleFlag::eVALID)
        {
            // access particle position
            const PxVec3& position = *positionIt;
        }
    }

    // return ownership of the buffers back to the SDK
    rd->unlock();
}
  
```

```
}
```

Example of how to use the valid particle bitmap to access particle data (the locking and unlocking):

```
if (rd->validParticleRange > 0)
{
    // iterate over valid particle bitmap
    for (PxU32 w = 0; w <= (rd->validParticleRange-1) >> 5; w++)
    {
        for (PxU32 b = rd->validParticleBitmap[w]; b; b &= b-1)
        {
            PxU32 index = (w << 5 | Ps::lowestSetBit(b));

            // access particle position
            const PxVec3& position = rd->positionBuffer[index];
        }
    }
}
```

Parameter Guide

There are three types of particle system parameter. Some need to be set when the particle system is created and cannot be changed afterwards. Some are set when the particle system is not part of a scene and others can be changed afterwards. The following description covers parameters that either cannot be set afterwards or induce a performance overhead when changed.

maxParticles:

The maximum number of particles that can be added to a particle system. The smaller the value, the smaller the memory footprint of the particle system. Can only be set on particle system creation.

PxParticleReadDataFlags:

Specifies a subset of simulation properties which are returned to the user during a simulation. See [Reading Particles](#). As few read data flags should be set in order to save memory and improve performance by avoiding unnecessary data copying. Parameter can only be changed while particle system is not part of a scene.

gridSize:

A hint for the PhysX SDK to choose the particle grouping granularity for collision tests and parallelization. See [Particle Grid](#). Parameter can only be changed when particle system is not part of a scene.

PxParticleBaseFlag::eENABLED:

Enables/disables particle simulation.

PxParticleBaseFlag::eGPU:

Enable/disable GPU acceleration. Changing this parameter while the particle system is part of a scene induces a large performance overhead.

PxParticleBaseFlag::eCOLLISION_WITH_DYNAMIC_ACTORS:

Enable/disable collision with dynamic rigids. Changing this parameter while the particle system is part of a scene induces a performance overhead.

PxParticleBaseFlag::eCOLLISION_TWOWAY:

Enable/disable twoway interaction between particles and rigid bodies parameter while the particle system is part of a scene induces a performance overhead.

PxParticleBaseFlag::ePER_PARTICLE_COLLISION_CACHE_HINT:

Enable/disable internal collision caches. Changing this parameter while the particle system is part of a scene induces a performance overhead.

Particle Dynamics

externalAcceleration:

Acceleration applied to each particle at each time step. The scene gravity is added to the external acceleration by default can be disabled using *PxActorFlag::eDISABLE_GRAVITY*.

maxMotionDistance:

The maximum distance a particle can travel during one simulation step. High values may hurt performance, while low values may restrict the particle velocity. In order to improve performance it's advisable to set this to a low value and then increase it until particles can move fast enough to achieve the target velocity. Parameter can only be changed while particle system is not part of a scene.

damping:

Velocity damping constant, which is globally applied to each particle. It is particularly useful when using particles for smoke to prevent ballistic motion of individual particles which can look odd.

particleMass:

Mass used for two way interaction with rigid bodies (*PxParticleBaseFlag::eCOLLISION_TWOWAY*) and different force multiplier in the context of *PxParticleBase::addForces*. This mass property doesn't have an effect on the fluid dynamics simulation.

PxParticleBaseFlag::ePROJECT_TO_PLANE, *projectionPlaneDistance:*

Parameter to configure the projection mode which confines particles to a plane. If projection is enabled particles can only move in a plane. This can be used in the context of a 2D-Game.

Collision with Rigid Actors

restOffset:

Defines the minimum distance between particles and the surface of rigid actors maintained by the collision system. Parameter can only be changed while particle system is not part of a scene.

PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET:

Enables/disables per-particle rest offsets. Memory can be saved by turning per-particle rest offsets off. Per-particle rest offsets should only be enabled for objects of significantly varying size, for example in the context of particle effects. See [Per-particle Rest Offsets](#). Can only be set on particle system.

contactOffset:

Defines the distance at which contacts between particles and rigid actors are created. The contacts are internally used to avoid jitter and sticking. It needs to be greater than *restOffset*. A good value to start with is about twice the size of the *restOffset*. Parameter can only be changed while particle system is not part of a scene.

restitution:

Restitution used for particle collision. This parameter defines how strongly particles bounce off rigid actors.

dynamicFriction:

Dynamic friction used for particle collision. This parameter defines how much particles slide over rigid actor surfaces. The lower the value is to 0, the more particles slide. One is the maximal value supported.

staticFriction:

Static friction used for particle collision. This parameter is similar to dynamic friction but defines how easily particles start to slide over a surface. Values from 0 to 1 are supported.

simulationFilterData:

Filter data used to filter collisions between particles and rigid bodies. See [Filtering](#).

PxParticleBaseFlag::eCOLLISION_TWOWAY:

The collision two-way flag allows enabling/disabling two-way interactions between rigid bodies and particles. The particle mass parameter defines the strength of the interaction. The flag can only be changed while the particle system is not in a scene.

Fluid (PxParticleFluid)

The SPH simulation can be tricky to tweak for good results. As this simulation (see [References](#)) uses an explicit integration scheme it only provides access to a certain parameter sub-space. A good set of parameter values depends on the size of the simulation and the external forces applied (such as gravity). Starting points for parameter values below assume a time step size of a gravity around 10 [m/s²]. Using a *damping* value larger than zero provides a useful parameter sub-space, for example useful when implementing a smoke effect.

restParticleDistance:

Defines the resolution of the particle fluid. It defines the approximate distance between neighboring particles will adopt within a fluid volume at rest. For the particle fluid simulation mentioned above, the particle rest distance should be smaller than 0.05 [m]. Parameter can only be changed while particle fluid is not part of a scene.

stiffness:

The stiffness (or gas constant) influences the calculation of the pressure. Low values of stiffness make the fluid more compressible (i.e., springy), while high values make it less compressible. The stiffness value has a significant impact on the numerical stability of the simulation; setting very high values will result in instability. Reasonable values are usually between 1 and 200.

viscosity:

Viscosity controls a fluid's thickness. For example, a fluid with a high viscosity will behave like treacle, while a fluid with low viscosity will be more like water. The viscosity value scales the force to reduce the relative velocity of particles in the fluid. Both, too high and too low values will typically result in instability. Reasonable values are usually between 5 and 300.

Collision Handling

By default, particles will collide with any shapes inside the PxScene. They will attempt to maintain a fixed distance from these shapes using `PxParticleBase::setRestOffset()`.

Collision Filtering

Filtering particle versus rigid body collisions can be useful to reduce performance overhead or simply to avoid undesired collisions.

For the following examples filtering is useful:

- Avoid particles colliding with trigger shapes (this is already handled by `PxDefaultSimulationFilterShader`)
- Configure a drain shape to exclusively collide with particles
- Have particles collide with a proxy shape as opposed to the shape collisions

Filter information for particles can be specified using `PxParticleBase::setSimulationFilterData()`. Instructions for how to setup filtering can be found here: [Collision Filtering](#).

Per-particle Rest Offsets

It is also possible to set a rest offset per-particle, using `PxParticleBase::setRestOffset()`. In order to provide per-particle rest offsets, the flag `PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET` needs to be set. Per-particle rest offsets must be smaller than the per-system rest offset, which can be retrieved using `PxParticleBase.getRestOffset()`.

Particle Drains

Using drains is a good method for keeping the particle count and speed low. Placing drains around the area of interest in which a particle system maintains good performance of the particle simulation. The area of interest, also be moved with the player.

Example of how to flag a *PxShape* *rbShape* as a drain:

```
rbShape->setFlag(PxShapeFlag::ePARTICLE_DRAIN, true);
```

Particles that collide with a drain are flagged with *PxParticleFlag::eCOLLISION_WITH_DRAIN* and may be released.

Best Practices / Troubleshooting

Particle Grid and Spatial Data Structure Overflow

The PhysX SDK uses a grid to subdivide the particles of a particle groups. This is done to accelerate proximity queries and for parallelization. The grid size parameter needs to be experimentally determined. *PxParticleBase::setGridSize()* for best performance. When doing this, visualize the grid using *PxVisualizationParameter::ePARTICLE_SYS*. Small grid size values might result in spatial data structure overflow, since the number of cells is limited to about 1000. Large grid size values on the other hand result in lower performance due to ineffective spatial queries or lack of parallelization.

In case of overflow, some particles will stop colliding with rigid actors in the scene. These particles are marked with *PxParticleFlag::eSPATIAL_DATA_STRUCTURE_OVERFLOW* and should be released.

GPU/CUDA Acceleration

PhysX 3 supports GPU acceleration. This allows for larger and more effects while retaining good performance levels. To achieve this goal, we create a `physx::PxGpuDispatcher` for the scene we want to add the particle system.

```
#if PX_WINDOWS
    // create cuda context manager
    PxFoundation& foundation = ...
    physx::PxCudaContextManagerDesc cudaContextManagerDesc;
    physx::PxCudaContextManager* cudaContextManager =
        PxCreateCudaContextManager(foundation, cudaContextManagerDesc);
#endif

    PxSceneDesc sceneDesc(mPhysics->getTolerancesScale());
    //...
#if PX_WINDOWS
    if (cudaContextManager)
        sceneDesc.gpuDispatcher = cudaContextManager->getGpuDispatcher();
#endif
    //...
    physicsSdk->createScene(sceneDesc);
```

A particle system can be configured for GPU simulation by setting the `PxParticleBaseFlag::eGPU`. Toggling GPU acceleration while the particle system is active on a scene might have a bad impact on performance since its state needs to be transferred from the GPU device memory. It is therefore better to set `PxParticleBase::setParticleBaseFlag()` before adding the particle system.

Particle data can be read directly from the GPU by using `PxParticleBase::lockParticleReadData(PxDataAccessFlag::eDEVICE)` or `PxParticleFluid::lockParticleFluidReadData(PxDataAccessFlag::eDEVICE)`. This is useful when used to render particles directly with CUDA Graphics Interop.

Convex, Triangle and Height field meshes are automatically mirrored in memory when the corresponding shapes are within the proximity of a GPU accelerated particle system. This may cause some undesired performance hiccups which can be avoided by mirroring the meshes explicitly, as shown in this example:

```
#if PX_WINDOWS
// mirror PxTriangleMesh triangleMesh providing the correspon
// the desired scene.
PxParticleGpu::createTriangleMeshMirror(triangleMesh, *cudaCo

// later release the obsolete mirror
PxParticleGpu::releaseTriangleMeshMirror(triangleMesh, *cudaC
#endif
```

On Kepler and above GPUs, the triangle meshes can be cached performance. The amount of memory to be allocated for caching can be

```
PxParticleGpu::setTriangleMeshCacheSizeHint(const class PxScene&
```

The triangle mesh cache will be shared among all the particle system in the scene. The optimal size depends on the scene (i.e. triangle mesh distribution). The cache usage statistics can be queried and analyzed using the cache size hint.

Sample Discussion

The `SampleParticles` shows both particle system types being used: `F` used for small debris and smoke, while `PxParticleFluid` is used for a wave. It provides example implementations of various aspects described in this

- `SampleParticles::createParticleSystem`, `SampleParticles::createFluid` create particle systems.
- `ParticleSystem::createParticles` creates particles within a particle system.
- `ParticleSystem::update` shows how to read, update, release particles with spatial data structure overflows.
- `SampleParticlesFilterShader` is an example for setting up collision filtering.
- `SampleParticles::createDrain` shows how to setup a rigid body sink drain.
- `SampleBase::onInit` illustrates how to setup GPU/CUDA acceleration.

The sample makes use of various helper classes:

- `ParticleSystem`: Encapsulates a `PxParticleSystem` or `PxParticleEmitter` and manages application side data such as particle lifetimes and orientations. It facilitates creating and releasing particles and double buffers for asynchronous rendering.
- `RenderParticleSystemActor`: Owns a `ParticleSystem` and provides rendering functionality.
- `ParticleEmitterRate`: Emits particles at a specified rate (#particles per second).
- `ParticleEmitterPressure`: Emits particles maintaining a certain distance from a surface.
- `SampleParticles::Emitter`: Connects an emitter as described in `RenderParticleSystemActor`.
- `SampleParticles::Raygun`: Provides functionality for the ray force field, particle debris and smoke emission.

In the sample, the smoke effect is achieved by using a *PxParticleSystem*. Each particle is rendered as a point sprite with a smoke texture. The smoke particles are spawned when the particles get close to the end of their lifespan. The smoke particles are visible throughout the scene, which can be seen when roaming the smoke with the camera. The smoke effect is generated for the craters, as well as for the ray-gun impacts. The realism of the smoke effect could be increased by using a particle fluid in order to get the smoke to expand. This is typically useful for indoor scenes or ground fog like effects. The smoke particles get into pooling situations.

Two kinds of debris are shown in the sample. Larger chunks of debris are rendered using convex-shaped rigid bodies. Smaller but more abundant chunks of debris are rendered as particles, which helps performance. The particle based debris is rendered as simple meshes. It is spawned in the craters and at the ray-gun impact location.

In order to give the chunks the appearance of a tumbling motion a simple approach is used:

1. Assign an initial random rotation matrix to each particle.
2. Change this rotation matrix proportional to the linear velocity of the particle.

The implementation of this approach can be found in *ParticleSystem::initializeParticleOrientations* and *ParticleSystem::modifyParticleOrientations*.

References

Particle-Based Fluid Simulation for Interactive Applications

Matthias Muller, David Charypar and Markus Gross, Eurographics/Siggraph
Breen, M. Lin Editors

<http://www.matthiasmueller.info/publications/sca03.pdf>

Fast GPU Fluid Simulation in PhysX

Simon Schirm and Mark Harris, NVIDIA Corporation

Chapter 7.3 of Game Programming Gems 8, Adam Lake

NVIDIA PhysX SDK 3.4.1 Documentation »

previous | next

User's Guide »

Introduction

The PhysX clothing feature has been deprecated in PhysX version 3.0 and APEX clothing features are replaced by the standalone NvClot.

Realistic movement of character clothing greatly improves the player experience. PhysX 3 cloth feature is a complete and high-performance solution to character clothing. It provides local space simulation for high accuracy and stability to reduce stretching, collision against a variety of shapes, as well as self-collision and inter-collision to avoid the cloth penetrating itself or other cloth. The simulation can be offloaded to CUDA capable GPUs for better performance on high resolution assets at higher resolutions than the CPU is able to handle.

PhysX 3 cloth is a rewrite of the PhysX 2 deformables, tailored for character cloth. Softbodies, tearing, and two-way interaction have been added. Behavior and performance for cloth simulation have been improved.

Simulation Algorithm

For one PhysX simulation frame, the cloth solver runs for multiple iterations. The number of iterations is determined by the solver frequency parameter and the time. Each iteration integrates particle positions, solves constraints for each character and self-collision. Cloth inter-collision is performed once per iteration. Cloth instances in the scene have been stepped forward. Local frame, motion and collision shapes are interpolated per iteration from the per-frame values. The user can control the solver frequency.

Solver Frequency

The size of the iteration time step is inversely proportional to the number of iterations:

```
cloth.setSolverFrequency(240.0f);
```

The solver frequency is specified as iterations per second. A solver frequency of 240 corresponds to 4 iterations per frame at 60 frames per second. In general, the simulation becomes more accurate if higher solver frequency value is used. However, the performance grows roughly linearly with solver frequency. Typically this value is between 120 and 240.

The number of iterations for each frame is derived using the simulation time-step. PhysX tries to handle variable time-steps by taking into account variations of the time-step during position integration and by adjusting damping parameters like constraint stiffness. While this reduces the artifacts due to varying time step sizes, use of variable time step sizes is not recommended.

Particle Integration

The first step in a cloth iteration predicts the new particle position based on the current position, velocity and external acceleration. While a particle state consists of position and the position before the last iteration, the particle velocity in

computed by dividing the position delta by the delta time of the previous

Local Space Simulation and Inertia Scale

Each PxCloth actor has a transformation that transforms particles from world space positions. For example:

```
cloth.setGlobalPose(PxTransform(PxVec3(1.0f, 0.0f, 0.0f));
```

will change the cloth's world space position to (1,0,0). Now compare the

```
cloth.setTargetPose(PxTransform(PxVec3(1.0f, 0.0f, 0.0f));
```

, which also changes the cloth's position to the same place. So what's d

PxCloth::setGlobalPose() only moves the cloth, but *PxCloth::setTargetPose()* generates acceleration (inertia) due to the position change. The amount of frame acceleration affects the cloth particles can be controlled using *PxCloth::setInertiaScale()*. For example to impart half the local frame acceleration to the particles use:

```
cloth.setInertiaScale(0.5f);
```

Scaling inertia effects individually per translation and rotation axis is also part of the family of *PxCloth::set*InertiaScale()* methods. Limiting the amount of frame accelerations affect particles can be especially useful for fast moving ch

Note: Using *setGlobalPose()* is equivalent to using *setTargetPose()* with *setInertiaScale(0)*. In this case, the cloth does not receive any acceleration due to frame

Constraints

After the particle positions have been integrated, a set of different constraints are used to simulate stretch, shear and bending forces, as well as to confine the cloth within a certain region.

Distance Constraints

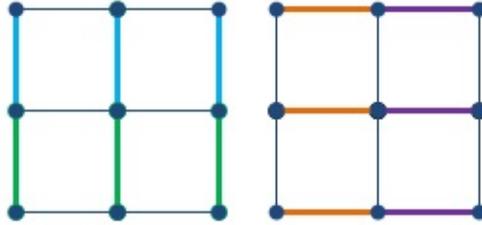


Figure 1. Typical configuration for vertical (left), and horizontal (right) stretching constraints.

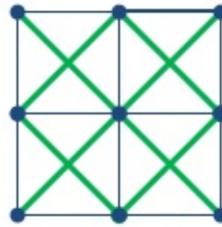


Figure 2. Typical configuration for shearing distance constraints.

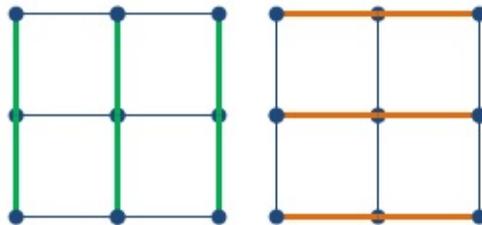


Figure 3. Typical configuration for vertical (left), and horizontal (right) bending constraints. Note bending constraints typically span more than one edge in a grid.

One of the most important roles for the cloth solver is to maintain the shape of particles so that the cloth does not stretch. This is achieved by constraints between pairs of particles. The way particles are connected allows the cloth to stretch, compress, shear, rotate, and bend. PhysX constraints are divided into 4 types (see *PxClothFabricPhaseType*), each of which has different stiffness parameters.

Below is an example of stiffness settings for each constraint type:

```
cloth.setStretchConfig(PxClothFabricPhaseType::eVERTICAL, PxCloth  
cloth.setStretchConfig(PxClothFabricPhaseType::eHORIZONTAL, PxClo  
cloth.setStretchConfig(PxClothFabricPhaseType::eSHEARING, PxCloth  
cloth.setStretchConfig(PxClothFabricPhaseType::eBENDING, PxClothS
```

Sometimes it is desirable that distance constraints are not enforced with a stiffness parameter allows only correcting a portion of the edge length in each iteration, for example to reduce the strength of bending constraints. A stiffness of 0.5 can be used for edges that are only moderately stretched or compressed. For example, a dress can be made to stretch when the character is taking large turns to behave correctly during pirouettes.

The following code sets up the vertical constraints such that when edges are stretched more than 60% or stretched more than 120%, a stiffness of 0.8 will be used. A stiffness of $0.4 = 0.8 * 0.5$ will be used:

```
PxClothStretchConfig stretchConfig;  
stretchConfig.stiffness = 0.8f;  
stretchConfig.stiffnessMultiplier = 0.5f;  
stretchConfig.compressionLimit = 0.6f;  
stretchConfig.stretchLimit = 1.2f;  
cloth.setStretchConfig(PxClothFabricPhaseType::eVERTICAL, stretchConfig);
```

Note: Stretch settings for horizontal and vertical directions are specific to the phase type. This can be used to handle stretching along the gravity (vertical) direction.

Tether Constraints

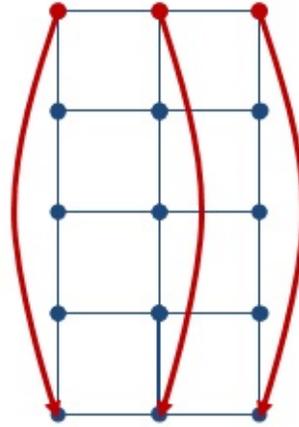


Figure 4. Example tether constraint configuration

The distance constraints are solved only once per iteration without con
The most visible artifact of this approximation is that the cloth
Increasing solver frequency reduces the stretching, but results in in
time.

PhysX 3.3 introduces tether constraints as a solution to avoid stretching
fast motion. Tether constraints prevent stretching by limiting the dista
move away from their anchor particles. This constraint adds very
computation to the solver, so it is more effective than increasing the nur

The tether constraints are automatically generated by the co
PxClothMeshDesc::invMasses values are set to zero, telling the
corresponding particles are non-simulated anchor particles whose pos
solely from users. Changing inverse masses after the fabric has been
affect which anchor particles are used for the tether constraints.

Motion Constraints

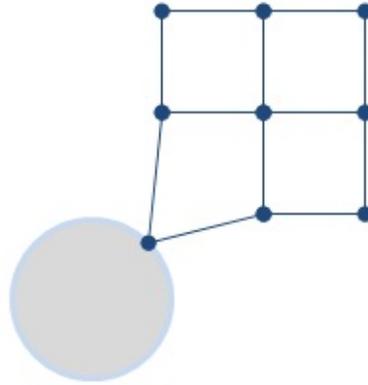


Figure 5. Example motion constraint

One can fully constrain a point to user specified position with zero inverse mass. It is sometimes desirable to confine a point within a small region around a specified position. This allows small details to be generated by suppressing any excessive deviation from the desired position.

Motion constraints lock the movement of each particle inside a sphere. In an animation system, one can sketch the overall movement of a cloth while the details are handled by the cloth simulation.

PxClothParticleMotionConstraint structure holds the position and radius of each particle, and motion constraints can be specified as follows:

```
PxClothParticleMotionConstraints motionConstraints[] = {
    PxClothParticleMotionConstraints(PxVec3(0.0f, 0.0f, 0.0f), 0.0f),
    PxClothParticleMotionConstraints(PxVec3(0.0f, 1.0f, 0.0f), 1.0f),
    PxClothParticleMotionConstraints(PxVec3(1.0f, 0.0f, 0.0f), 1.0f),
    PxClothParticleMotionConstraints(PxVec3(1.0f, 1.0f, 0.0f), 1.0f),
};

cloth.setMotionConstraints(motionConstraints);
```

If the sphere radius becomes zero or negative, the corresponding particle is fully constrained to the sphere center and the inverse particle mass is set to zero. For the example above, the first particle will fully lock to the constraint position and the third particle will remain within the sphere radius. The last particle is not constrained.

The motion constraint sphere radius can be globally scaled and bias transition between simulated and animated states. See *PxClothMotionC* details.

Separation Constraints

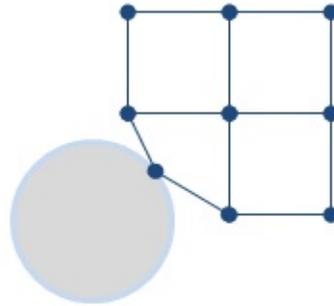


Figure 6. Example separation constraint

Separation constraints work exactly the opposite way to the motion constraint: a particle to stay outside of a sphere. When particle movement is moderate (e.g. sleeves around an arm), separation constraints represent the character's collision shape more accurately than using collision constraints. For example, separation constraints can be placed slightly inside the character's collision shape. The constraint radius to be the distance from the sphere center to the surface of the character's collision shape.

See *PxClothParticleSeparationConstraint* and *PxCloth::setSeparationConstraint*.

Collision Detection

Each cloth object supports collision with spheres, capsules, planes, and collision planes) and triangles. By default these shapes are all treated separately in the PhysX rigid body scene, however collision against other PxScene actors using the *PxClothFlag::eSCENE_COLLISION* flag.

Collision shapes are specified in local coordinates for the next frame before the scene. An independent and complete collision stage is performed as a single iteration, using shape positions interpolated from the values at the beginning of the frame.

of the frame. Sphere and capsule collision supports continuous collision and an acceleration structure to cull far-away particles early in the collision and capsules are therefore the preferred choice to model the character. Convexes and triangles should only be used sparingly.

Spheres are defined as center and radius. Note that the radius is specified in the local frame and will change from frame to frame. The total number of spheres is limited to 32.

Capsules are defined by a pair of indices into the spheres array and have a different radius thus forming a tapered capsule. Spheres can be used to form multiple capsules, which can be useful for modeling characters (upper and lower leg spheres can share the sphere at the knee). Sharing of spheres in a collision simulation is more efficient and robust, so is highly encouraged.

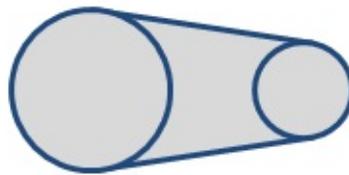


Figure 7. A tapered capsule collision shape formed by two connected spheres.

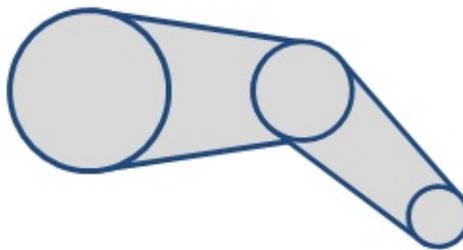


Figure 8. A leg shape formed by using two tapered capsules, each sharing a sphere at the middle.

Planes are defined by their normal and distance to origin. They will not collide unless they are referenced by a convex shape. Convexes reference a mask, where each bit corresponds to an entry in the array of planes.

32 planes per cloth.

Triangle colliders are defined as vertex triplets in counter-clockwise order. Triangles should form a closed patch near the cloth for consistent collision. A particle collides against its closest triangle expanded to an infinite plane.

The order of planes and triangles should remain unchanged (apart from reordering through the *PxCloth::removeCollisionPlane/Triangle()* method) as they are interpolated between simulation frames.

Continuous Collision Detection

Besides discrete collision which resolves particles inside shapes at each iteration, continuous collision detection is supported and can be enabled.

```
// Enable continuous collision detection
cloth.setClothFlag(PxClothFlag::eSWEPT_CONTACT, true);
```

Continuous collision is around 2x more computationally expensive than discrete collision but it is necessary to detect collision between fast moving objects. Continuous collision analyzes the trajectory of particles and capsules to determine when a collision occurs. At the first time of contact, the particle is moved with the shape until the end of the frame.

Note: The SIMD collision path handles sets of 4 particles in parallel. It is advantageous to spatially group cloth particles so that they are likely to be processed by the same set of shapes.

Virtual Particle Collision

Virtual particles provide a way of improving cloth collision without increasing resolution. They are called 'virtual' particles because they only exist during the collision processing stage and do not have their position, velocity or mass. Unlike regular particles, they can be thought of as providing additional sampling points on the cloth surface.

During collision processing each virtual particle is created from three using barycentric interpolation. It is then tested for discrete collision likelihood and the collision impulse is redistributed back to the original particle interpolation.

Section [Adding Virtual Particles](#) explains the necessary steps to use this

Friction and Mass Scaling

Coulomb friction can be enabled and will be applied for particle-particle collisions by setting a friction coefficient between 0 and 1:

```
cloth.setFrictionCoefficient(0.5f);
```

Additionally, there is an option to artificially increase the mass of collision temporary increase in mass can help reduce stretching along edges that are pulled over a collision shape. The effect is determined by the relative mass of the particle and collision shape and a user defined coefficient. A value of 1 is a starting point but users are encouraged to experiment with this value:

```
cloth.setCollisionMassScale(20.0f);
```

Self-Collision of a Single Cloth Actor

The particles of a cloth actor can collide among themselves. To enable self-collision behavior, one should set both self-collision distance and self-collision stiffness values:

```
cloth.setSelfCollisionDistance(0.1f);  
cloth.setSelfCollisionStiffness(1.0f);
```

Self-collision distance defines the diameter of a sphere around each particle. The solver ensures that these spheres do not overlap during simulation. Self-collision stiffness defines how strong the separating impulse should be.

Self-collision distance should be smaller than the smallest distance between particles in the rest configuration. If the distance is larger, self-collision may violate constraints and result in jittering.

When such a configuration cannot be avoided (e.g. due to irregular initial positions) one can assign additional rest positions:

```
cloth.setRestPositions(restPositions);
```

Collision between two particles is ignored if their rest-positions are closer than the collision distance. However, a large collision distance and use of additional rest positions significantly degrade performance of self-collision, so should be used sparingly.

Self-collision performance for high-resolution cloth instances can be improved by using self-collision to a subset of all particles (see *PxCloth::setSelfCollisionIndices*).

Inter-Collision between Multiple Cloth Actors

Different cloth actors can be made to interact with each other when inter-collision is enabled. The parameters for inter-collision are set for all cloth instances in the scene.

```
scene.setInterCollisionDistance(0.5f);  
scene.setInterCollisionStiffness(1.0f);
```

The definition of distance and stiffness values are the same as for self-collision. Instances that specify a particle subset for self-collision use the same collision parameters.

Best Practices / Troubleshooting

Performance Tips

The runtime of the cloth simulation scales approximately linearly with the number of particles and the solver frequency: Simulating a higher resolution mesh and increasing stretch stiffness and collision handling fidelity will increase the time it takes to simulate one frame. Additionally, performance drops somewhere below 3000 particles for the GPU solver in the next section. As a rough guideline, a dozen cloth instances with 2000 particles and a solver frequency of 300Hz can be simulated in real-time as part of a game.

Convex collision and triangle collision do not use any mid phase acceleration and are therefore slower than sphere and capsule collision.

Self-collision and inter-collision can take a significant amount of the time. Consider keeping the collision distance small and using self-collision to reduce the number of particles that collide with each other.

Using GPU Cloth

Cloth can be simulated on a CUDA or DirectCompute enabled GPU, by setting the corresponding flags:

```
cloth.setClothFlag(PxClothFlag::eCUDA, true);  
cloth.setClothFlag(PxClothFlag::eDIRECT_COMPUTE, true);
```

The entire cloth solver pipeline is run on the GPU, with the exception of collision detection. When no supported GPU is available PhysX will issue a warning and the simulations will be run on CPU.

When the cloth is simulated using CUDA, the GPU simulation results can be rendered using the graphics API by requesting CUDA device pointers to the particle data:

```
cloth.lockParticleData(PxDataAccessFlag::eDEVICE);
```

To take full advantage of the GPU hardware there should be at least 100 instances as streaming multiprocessors (SMs). This means it is better to simulate clothing as multiple instances (e.g. shirts and skirt) rather than a single instance.

GPU PhysX performance is better when the particle data of a cloth fits into shared memory. The number of particles that fit into shared memory depends on collision shapes, whether continuous collision or self-collisions are enabled in the GPU version. For GPUs supporting SM 2.0 and above, about 250 particles fit into shared memory. If particles don't fit into shared memory they are automatically streamed through global memory, which incurs a performance cost.

Furthermore, the limited size of shared memory requires the number of particles to be clamped to 500 when GPU simulation is enabled.

Fast-Moving Characters

Consistent collision handling for fast-moving characters can be difficult. Translations and rotations are best handled by tying the cloth local simulation frame to the character's transformation. The inertia effects of the local frame transformations can be mitigated by using the inertia scale settings.

If the cloth tunnels collision shapes during fast character animations, increase the solver frequency or enabling swept contacts (see *PxClothFlag::eSWEEP*).

Avoiding Stretching

Due to the iterative nature of the distance constraint solver, high stretch is undesirable under strong gravity even if the stretch stiffness is high. Increasing the solver frequency mitigates the stretching, but a higher frequency solver is better suited to eliminate stretching efficiently.

Avoiding Jitter

Under certain configurations, different constraint types can violate each other and over-constrain the particle positions. For example, a motion constraint can allow a particle to move further from the anchor particle than the tether constraint permits, or a particle can become pinched between two overlapping collision shapes. Over-constraining can result in jitter and should be avoided. In some situations jitter can be reduced by increasing the solver frequency or by reducing the corresponding constraints.

PVD Support

Cloth particle positions, distance constraints, and collision shapes are rendered as points, lines, and wireframes respectively in PVD. The SDK does not have a built-in fabric, and this mesh can't be displayed in PVD either. However, you can display individual sets of distance constraints instead of all at once by switching *Mode* to *Single Phase* in the *Preferences* dialog and use the *Cloth Phase* dropdown to display. The *Particle Scale* slider in the same dialog affects the size of both ordinary and virtual cloth particles as well. All properties of a selected object can be viewed in the *Inspector* panel of PVD.

Snippet Discussion

The following paragraph describes code of the cloth snippet provided w

The cloth constraint connectivity and rest values are stored in (*PxClothFabric*), separate from the cloth actor (*PxCloth*). The separatio from particles allows the same fabric data to be reused for multip reducing cooking time and storage requirements. *PxClothFabricCreate* library, creates a fabric from a triangle or quad mesh (see *PxClothMe* actor itself is created through the physics instance (*PxPhysics*) and ne a scene (*PxScene*) in order to be simulated. Once the cloth actor is assign simulation settings such as collision data, constraint stiffness, sc self-collision. The *createCloth* function in the cloth snippet performs the:

The *stepPhysics* function advances the simulation by one frame. It fir local frame, which rotates around the y-axis. The collision shapes are n coordinates, but their positions are specified in cloth local coordinates, to be updated every frame. The following sections detail some of the av and show how to configure them.

Note: The cloth module has to be registered with *PxRegisterCloth* on static linking (non windows) before creating cloth objects. *PxCreatePhy* modules by default as opposed to *PxCreateBasePhysics*.

Filling in PxClothMeshDesc

The first task to create a cloth is to fill in the *PxClothMeshDesc* stru programmatically creates a regular grid of cloth particles connected Below is a simpler example on how to create a cloth from a simple m single quad.

```
PxClothParticle vertices[] = {
    PxClothParticle(PxVec3(0.0f, 0.0f, 0.0f), 0.0f),
    PxClothParticle(PxVec3(0.0f, 1.0f, 0.0f), 1.0f),
```

```

    PxClothParticle(PxVec3(1.0f, 0.0f, 0.0f), 1.0f),
    PxClothParticle(PxVec3(1.0f, 1.0f, 0.0f), 1.0f)
};

PxU32 primitives[] = { 0, 1, 3, 2 };

PxClothMeshDesc meshDesc;
meshDesc.points.data = vertices;
meshDesc.points.count = 4;
meshDesc.points.stride = sizeof(PxClothParticle);

meshDesc.invMasses.data = &vertices->invWeight;
meshDesc.invMasses.count = 4;
meshDesc.invMasses.stride = sizeof(PxClothParticle);

meshDesc.quads.data = primitives;
meshDesc.quads.count = 1;
meshDesc.quads.stride = sizeof(PxU32) * 4;

```

Each particle is defined by its position in local coordinates and its inverse mass to zero indicates that the particle is not simulated. Inverse mass fixed in local space or kinematically constrained to user specified position. The mass of simulated particles can normally be set to any fixed positive value.

The *PxClothMeshDesc* structure allows positions and inverse masses arrays or interleaved like in the code above. The mesh can consist of triangles or quads or both. The cooker prefers quad meshes over triangle meshes when classifying constraint types. The extensions library therefore provides the *PxClothMeshQuadifier* helper class to extract quads from a triangle mesh.

Creating Fabric

Given the mesh descriptor, a call to *PxClothFabricCreate* in the extension library performs the generation of constraints and the creation of the *PxClothFabric* structure.

```

PxClothFabric* fabric = PxClothFabricCreate(physics, meshDesc, PxVec3(0, 0, 1));

```

The third parameter indicates the direction of gravity, which is used as the direction of 'horizontal' or 'vertical' constraints.

The *PxClothFabric* class describes internal solver data for a cloth. For constraints consisting of two particle indices and a rest-length are created and stored in the fabric data. Multiple cloth instances of the same mesh fabric instance.

Creating Cloth

A *PxCloth* object is created using a fabric instance and the initial pose. Like all actors, the cloth instance is simulated as part of a scene:

```
PxTransform pose = PxTransform(PxIdentity);
PxCloth* cloth = physics.createCloth(pose, fabric, vertices, PxCl
scene.addActor(cloth);
```

The first parameter specifies the initial pose. The second input is the cloth mesh created by the cooker. The third input provides initial particle positions and masses. Typically this array is the same as the one referenced by the fabric used to create the fabric. Note that the rest configuration (such as the rest distance constraint) is computed from *PxClothMeshDesc*, so the initial configuration do not affect rest configuration. The last parameter is a set of flags to enable simulation and continuous collision detection to be enabled. The default options.

Specifying Collision Shapes

The following code illustrates how to add two spheres of different radii and a tapered capsule between them:

```
// Two spheres located on the x-axis
PxClothCollisionSphere spheres[2] =
{
    PxClothCollisionSphere( PxVec3(-1.0f, 0.0f, 0.0f), 0.5f ),
    PxClothCollisionSphere( PxVec3( 1.0f, 0.0f, 0.0f), 0.25f )
};

cloth.setCollisionSpheres(spheres, 2);
cloth.addCollisionCapsule(0, 1);
```

Planes can be added through `PxCloth::addCollisionPlane()` method considered for collision unless they are referenced by a convex shape following code shows how to setup a typical upward facing ground origin:

```
cloth.addCollisionPlane(PxClothCollisionPlane(PxVec3(0.0f, 1.0f, 0.0f),
cloth.addCollisionConvex(1 << 0); // Convex references the first
```

Planes may be efficiently updated after constructing `PxCloth::setCollisionPlanes()` function.

Finally, triangles are added using the `PxCloth::setCollisionTriangles()` example, the following code adds a tetrahedron made of four triangles:

```
PxClothCollisionTriangle triangles[4] = {
    PxClothCollisionTriangle(PxVec3(0.0f, 0.0f, 0.0f),
                             PxVec3(1.0f, 0.0f, 0.0f),
                             PxVec3(0.0f, 1.0f, 0.0f)),
    PxClothCollisionTriangle(PxVec3(1.0f, 0.0f, 0.0f),
                             PxVec3(0.0f, 0.0f, 1.0f),
                             PxVec3(0.0f, 1.0f, 0.0f)),
    PxClothCollisionTriangle(PxVec3(0.0f, 0.0f, 1.0f),
                             PxVec3(0.0f, 0.0f, 0.0f),
                             PxVec3(0.0f, 1.0f, 0.0f)),
    PxClothCollisionTriangle(PxVec3(0.0f, 0.0f, 0.0f),
                             PxVec3(0.0f, 0.0f, 1.0f),
                             PxVec3(1.0f, 0.0f, 0.0f)),
};
```

Note: The snippet adds collision convex and capsule once in the `create` function and then updates collision spheres, planes and triangles every frame in the `update` function.

Adding Virtual Particles

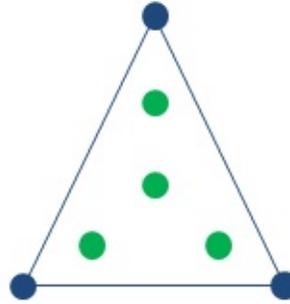


Figure 9. Four virtual particles (green) expressed as the weighted combination of the triangle's particles, virtual particles provide a better sampling of the cloth and improves collision detection.

A virtual particle is defined by 3 particle indices and an index into a weights table defines the barycentric coordinates used to create a virtual particle from a linear combination of the referenced particles. The following is a table that can be used to create a distribution of 4 virtual particles on a triangle.

```
static PxVec3 weights[] =
{
    PxVec3(1.0f / 3, 1.0f / 3, 1.0f / 3), // center point
    PxVec3(4.0f / 6, 1.0f / 6, 1.0f / 6), // off-center point
};
```

The code below shows an example of how to set up the virtual particles in a *PxClothMeshDesc*:

```
PxU32 numFaces = meshDesc.triangles.count;
assert(meshDesc.flags & PxMeshFlag::e16_BIT_INDICES);
PxU8* triangles = (PxU8*)meshDesc.triangles.data;

PxU32 indices[] = new PxU32[4*4*numFaces];
for (PxU32 i = 0, *it = indices; i < numFaces; i++)
{
    PxU16* triangle = (PxU16*)triangles;
    PxU32 v0 = triangle[0];
    PxU32 v1 = triangle[1];
    PxU32 v2 = triangle[2];

    // center
    *it++ = v0; *it++ = v1; *it++ = v2; *it++ = 0;
```

```

// off centers
*it++ = v0; *it++ = v1; *it++ = v2; *it++ = 1;
*it++ = v1; *it++ = v2; *it++ = v0; *it++ = 1;
*it++ = v2; *it++ = v0; *it++ = v1; *it++ = 1;

triangles += meshDesc.triangles.stride;
}

cloth.setVirtualParticles(numFaces*4, indices, 2, weights);
delete[] indices;

```

Accessing Particle Data

The cloth snippet doesn't render the result of the simulation, and then back any particle data. The *lockParticleData()* provides read and option the particle positions of the current and previous iteration. As an example code applies some external acceleration to each particle using *setParticleAccelerations()*:

```

Px ClothParticleData* data = cloth.lockParticleData(PxDataAccessFlags::eRead);
float dt = cloth.getPreviousTimeStep();
for(PxU32 i = 0, n = cloth.getNbParticles(); i < n; ++i)
{
    data->previousParticles[i].pos -= particleAccelerations[i] * dt;
}
data->unlock();

```

References

[1] Mueller, Matthias and Heidelberger, Bruno and Hennix, Marcus
Position based dynamics. Academic Press, Inc.. p.
<http://dx.doi.org/10.1016/j.jvcir.2007.01.005>

[2] Kim, Tae-Yong and Chentanez, Nuttapong and Mueller-Fischer, Ma
attachments - a method to simulate inextensible clothing in
Eurographics Association. p. 305--310 2012 [http://dl.ac
id=2422356.2422399](http://dl.ac.id=2422356.2422399)

Introduction

With the PhysX Visual Debugger (see [PhysX Visual Debugger \(PVD\)](#), tool to record information about simulated PhysX scenes and visualize a remote viewer application. However, sometimes it is preferable to integrate information directly into the application's view. For that purpose, F interface to extract visual debug information as a set of basic rendering primitives, lines, triangles and text. These primitives can then be rendered by the application render objects.

Usage

To enable debug visualization, the global visualization scale has to be set to a non-zero value first:

```
PxScene* scene = ...
scene->setVisualizationParameter(PxVisualizationParameter::eSCALE, 1.0f);
```

Then the individual properties that should be visualized can be enabled by setting a non-zero positive value:

```
scene->setVisualizationParameter(PxVisualizationParameter::eACTOR, 1.0f);
```

In the example, the actor world axes will be visualized. The scale used for the visualization will be the product of the global scale (1.0 in this example) and the property scale (1.0 in this example). Please note that for some properties the scale factor does not apply. For example, geometry, for example, will not be scaled since the size is defined by the object's geometry. Furthermore, for some objects, visualization has to be enabled on the object's corresponding object instances too (see *PxActorFlag::eVISUALIZATION*, ...).

After a simulation step, the visualization primitives can then be extracted from the scene:

```
const PxRenderBuffer& rb = scene->getRenderBuffer();
for(PxU32 i=0; i < rb.getNbLines(); i++)
{
    const PxDebugLine& line = rb.getLines()[i];
    // render the line
}
```

Note: Do not extract render primitives while the simulation is running.

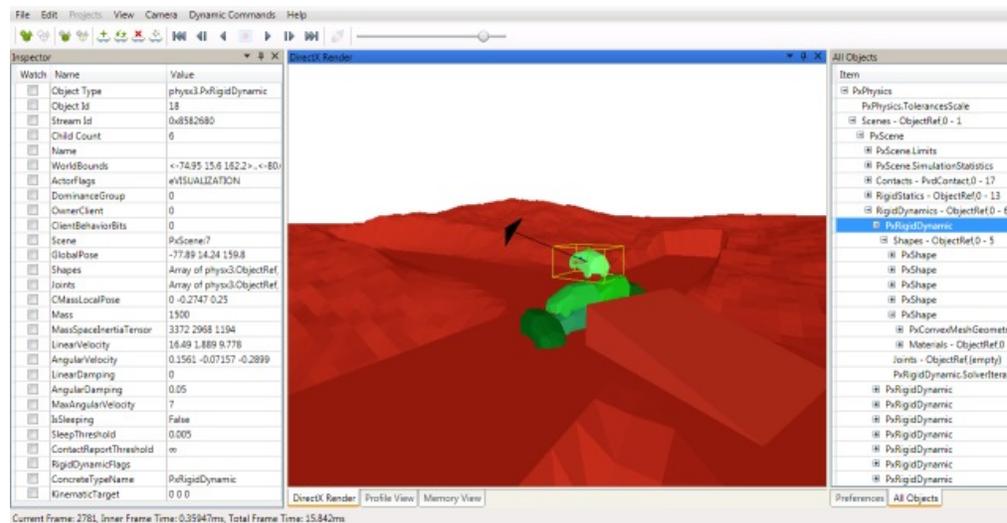
The amount of debug visualization data might be too vast to create scenes. In cases where only a localized area is of interest, there is a bounding box culling for debug visualization via *PxScene::setVisualizationCullingBox*.

Note that simply enabling debug visualization (PxVisualizationParam) have a significant performance impact, even when all the other indi flags are disabled. Thus, make sure debug visualization is disabled in builds.

[NVIDIA PhysX SDK 3.4.1 Documentation »](#)
[User's Guide »](#)

[previous](#) | [next](#)

PVD



The PhysX Visual Debugger (PVD) provides a graphical view of the includes various tools to inspect and visualize variables of eve Additionally it can also record and visualize memory and timing data.

PVD can be downloaded from: <http://supportcenteronline.com/ics/deptID=1949>

Questions regarding the usage of the GUI should all be answered by help.

Basic Setup (SDK Side)

PVD integration is enabled in the debug, checked and profiling configuration. In order to reduce memory footprint and code size, it is not enabled in the release configuration.

The SDK outputs the PVD debugging data in form of a stream. PVD stream can be either from a TCP/IP network socket or from a file.

Network Setup

Streaming to TCP/IP is supported on almost all platforms, and is a convenient way to collect PVD data. In this mode the stream can be written to a file or a network socket, depending on network speed and scene complexity. In network mode, a TCP/IP server must be launched before the SDK tries to connect. The default listening port is 5425:

```
use namespace physx;

PxPvd* pvd = PxCreatePvd(*foundation);
PxPvdTransport* transport = PxDefaultPvdSocketTransportCreate(PVD_STREAM_IP_ADDRESS);
pvd->connect(*transport, PxPvdInstrumentationFlag::eALL);

PxPhysics* physics = PxCreatePhysics(PX_PHYSICS_VERSION, *gFoundation, PxTolerancesScale());

//After releasing PxPhysics, release the PVD
physics->release();
pvd->release();
transport->release();
```

File Setup

Streaming to file is an alternative to network streams. This is the recommended way in case your platform or system setup does not support a network connection. File streams are often faster than network sockets and therefore a good choice when performance is more important than real-time viewing. Streams stored to file can be viewed later.

loaded by drag&drop or over the File->Load menu in PVD:

```
use namespace physx;  
  
PxPvd* pvd = PxCreatePvd(*foundation);  
PxPvdTransport* transport = PxDefaultPvdFileTransportCreate(filen  
pvd->connect(*transport,PxPvdInstrumentationFlag::eALL);  
  
PxPhysics* physics = PxCreatePhysics(PX_PHYSICS_VERSION, *gFounda  
  
//After releasing PxPhysics, release the PVD  
physics->release();  
pvd->release();  
transport->release();
```



Advanced Setup

Connection Flags

To optimize the stream size we provide flags to enable specific features and influence on PVD's and the SDK's performance:

- **PxPvdInstrumentationFlag::eDEBUG**: Transfer all debug data to inspect objects. This flag has usually the biggest impact on the stream size.
- **PxPvdInstrumentationFlag::ePROFILE**: Transfer timing information for profiling zones in our SDK.
- **PxPvdInstrumentationFlag::eMEMORY**: Transfer memory usage information.

Setup to transfer only profiling data over network:

```
pvd->connect(*transport, PxPvdInstrumentationFlag::ePROFILE);
```

Visualizing Externals and Extended Data

Joints are implemented as an extension to the SDK constraints and the handling to get transmitted to PVD. Both joint and contact data can increase stream size significantly. Visualizing it in PVD is therefore disabled by default. To enable it, use the following API calls:

```
mScene->getScenePvdClient()->setScenePvdFlags(PxPvdSceneFlag::eTR
```

or set the flags separately:

```
mScene->getScenePvdClient()->setScenePvdFlag(PxPvdSceneFlag::eTRA
```

Visualizing SceneQuery

Visualizing SceneQuery in PVD is disabled by default since queries increase the stream size significantly. To enable it use following API call

```
mScene->getScenePvdClient()->setScenePvdFlag(PxPvdSceneFlag::eTRA
```

Custom PvdClient

Implement the PvdClient interface if your application needs to react to disconnection from PVD, or if you plan to send custom PVD events from It is recommended to toggle the contact and constraint vis onPvdConnected/onPvdDisconnected callbacks to avoid potential memory overhead in the SDK:

```
// derive from PvdClient
struct MyPvdClient : public physx::pvd sdk::PvdClient
{
    virtual void onPvdConnected()
    {
        // 1. create a PvdDataStream
        // 2. send your custom PVD class descriptions from here
        // this then allows PVD to correctly identify and represent
        // custom data that is sent from your application to a Px
        // example in JointConnectionHandler
        // 3. do something when successfully connected
        // e.g. enable contact and constraint visualization
    }
    virtual void onPvdDisconnected()
    {
        // handle disconnection, release PvdDataStream
        // e.g. disable contact and constraint visualization
    }
    //implement other methods
    ...
};

// register custom handler
MyPvdClient myPvdClient;
pvd->addClient(myPvdClient);
```

PVD Error Stream

PhysX SDK sends all its own error messages to PVD if PVD is connected. You can call `Px::Foundation::error()` or `Px::Foundation::getErrorHandler` to report your error message. These functions will send error messages automatically.

The messages will be listed in ErrorStream view of PVD.

Custom profiling

When using `PxPvdInstrumentationFlag::ePROFILE`, PVD calls `PxSetProfilerCallback()` to set itself up as the current profiler. This happens after the `PxPvd::connect()` call, and it overrides the potentially already existing profiler. That is, if users call `PxSetProfilerCallback()` with their own user profiler before initializing PVD with `PxPvdInstrumentationFlag::ePROFILE`, then the user's profiler is lost. Similarly, initializing PVD first then calling `PxSetProfilerCallback()` with a custom profiler makes the PVD profiling results vanish.

In case both PVD's internal profiling and a user's custom profiling are needed at the same time, it is recommended to initialize PVD first, then call `PxSetProfilerCallback()` with your own profiler. In your implementation, call the PVD profiling functions `PxPvd::zoneStart()` and `PxPvd::zoneEnd()` after performing your own profiling operations:

```
struct UserProfilerCallback : public PxProfilerCallback
{
    PxPvd* mPvd;

    virtual void* zoneStart(const char* eventName, bool detached)
    {
        // Do custom profiling here

        // Then re-route to PVD implementation
        return mPvd->zoneStart(eventName, detached, context);
    }
    virtual void zoneEnd(void* profilerData, const char* eventName)
    {
        // Do custom profiling here
    }
}
```

```
        // Then re-route to PVD implementation
        mPvd->zoneEnd(profilerData, eventName, detached,
    }
};
```

This is illustrated in SnippetCustomProfiler.

[NVIDIA PhysX SDK 3.4.1 Documentation »](#)

[previous](#) | [next](#)

[User's Guide »](#)

Interface

In this chapter we will have a quick look at the statistics information that is available at every simulation step. Usually, this information can be explored in the PhysX Debugger but we do offer a PhysX API method as well to allow applications to retrieve statistics data directly. After a simulation step and a call to `PxScene::fetchResults()`, statistics for the processed step can be retrieved using the `PxScene::getSimulationStatistics()` interface. The method copies the provided `PxSimulationStatistics` structure. For details about the individual fields, refer to the API documentation.

Note: Do not fetch the simulation statistics while the simulation is running.

Usage

The provided simulation statistics is mainly meant to help investigate p
It provides a quantitative summary of the work done, i.e., the nur
combination of objects which have been processed in the current si
example, if you encounter performance spikes in certain frames, tl
statistics might give some insight into possible causes. For instance:

- Has a large amount of volumes been added or removed from the single step? You could try to distribute the addition/removal of object simulation steps or maybe there is a particle system in the scene very small.
- Are there suddenly many more collision pairs processed than expected be caused by a badly configured collision pair filter or maybe some been accidentally raised.
- etc.

Please keep in mind that the simulation statistics are currently less what the scene contains but rather what got processed. So it is only detect whether objects have been configured and arranged properly.

Introduction

PhysX 3 features two approaches to serialization:

- API-level serialization to RepX (an XML format)
- Binary serialization

API-level serialization uses a human readable XML format - RepX corresponds to the PhysX API. It is therefore suitable for manual modification for debugging purposes. It offers platform independence and the ability to load data that was serialized with a previous PhysX SDK version. API-level serialization is not expected to be used in performance critical situations.

The binary serialization approach on the other hand supports instantiating objects directly from memory without copying data. This in-place deserialization is well suited for performance critical real time situations. However, this approach is not as flexible as the binary format is specific to a given platform and PhysX SDK provides functionality to convert binary serialized data from authoring platforms to ease the asset management.

Note: *cooking* also generates a binary output stream. The primary purpose, however, is to translate from a user format to a format suitable for the engine so it is not considered a serialization mechanism. Loading a cooked mesh involves allocation and endian conversion. As a consequence, it is much slower than PhysX' binary serialization mechanism. See [Shapes](#) for more details.

The following documentation will discuss how to use both serialization mechanisms and show how to build collections of PhysX objects and how these collections are serialized and deserialized. Further it will show how dependencies to other application side objects can be re-established when deserializing.

PhysX also supports extending serialization to custom types, such as [User Defined Types](#). This is described in more detail in Section [Extending Serialization](#).

First Code

The following code creates and serializes a rigid dynamic using both formats:

```
// Create a material, a shape and a rigid dynamic
PxSphereGeometry geometry(1.0f);
PxMaterial* material = PxGetPhysics().createMaterial(0.0f, 0.0f,
PxShape* shape = PxGetPhysics().createShape(geometry, *material);
PxTransform t = PxTransform(PxIdentity);
PxRigidDynamic* dynamic = PxCreateDynamic(PxGetPhysics(), t, geom

PxSerializationRegistry* registry = PxSerialization::createSerial

// Create a collection and all objects for serialization
PxCollection* collection = PxCreateCollection();
collection->add(*dynamic);
PxSerialization::complete(*collection, *registry);

// Serialize either to binary or RepX
PxDefaultFileOutputStream outputStream("serialized.dat");

// Binary
    PxSerialization::serializeCollectionToBinary(outStream, *coll
//~Binary

// RepX
    PxSerialization::serializeCollectionToXml(outStream, *collect
//~RepX
```

Most operations related to serialization require an instance of *PxSerializationRegistry* which provides information on how to serialize PhysX types. In order to serialize an object, it needs to be added to a *PxCollection*. If an object has dependencies on other PhysX objects, they need to be serialized as well. *PxSerialization::complete* adds the required objects to the collection.

The following code deserializes the rigid dynamic and adds it to a scene

```
PxSerializationRegistry* registry = PxSerialization::createSerial
```

```

// Binary
// Open file and get file size
FILE* fp = fopen("serialized.dat", "rb");
fseek(fp, 0, SEEK_END);
unsigned fileSize = ftell(fp);
fseek(fp, 0, SEEK_SET);

// Allocate aligned memory, load data and deserialize
void* memory = malloc(fileSize+PX_SERIAL_FILE_ALIGN);
void* memory128 = (void*)((size_t(memory) + PX_SERIAL_FILE_AL
fread(memory128, 1, fileSize, fp);
fclose(fp);
PxCollection* collection = PxSerialization::createCollectionF
//~Binary

// RepX
// Load file and deserialize collection - needs cooking libra
PxDefaultFileInputData inputData("serialized.dat");
PxCollection* collection = PxSerialization::createCollectionF

//~RepX

scene->addCollection(*collection);

```

When deserializing a binary serialized collection, the data first needs memory block that is aligned to 128 bytes. The memory block may be released before the objects have been released: it needs to persist for the entire duration of the deserialization process. This does not apply to RepX deserialization, as the memory for PhysX objects is allocated within PhysX. Finally the objects of the resulting collection are added to the scene with *PxScene::addCollection*.

In-depth Discussion

Collections

The serialization system makes use of a class *PxCollection*, which manages objects deriving from *PxBase*. Each collection represents a set of objects and maintain a mapping between IDs of type *PxSerialObjectId* and object IDs. Object IDs may be defined by the application. One caveat here is that the IDs within a collection, but do not have to be unique across different collections. If required by the application, it is the application's responsibility to ensure

Here is an example of how to iterate over a collection, for instance objects intended for serialization have all been added to the collection. PhysX' dynamic typing mechanism can be used to classify the objects:

```
PxCollection* collection;
PxU32 size = collection->getNbObjects();
for(PxU32 i=0; i<size; i++)
{
    PxBase* object = collection->getObject(i);
    if(!object->is<PxActor>())
        continue;

    switch((PxConcreteType)object->getConcreteType())
    {
    case PxConcreteType::eRIGID_DYNAMIC:
        ...
    }
}
```

Note: In order to simplify releasing object within a collection, PhysXExt contains a function to remove and release all objects from a collection: *PxCollectionExt::releaseObjects*.

Note: Releasing an object within a collection invalidates the mapping objects.

A collection is said to be *complete* if no contained objects depend on a the collection. For example, an actor, a shape with a box geometry, and shape would together form a complete collection. The same collection v would be incomplete.

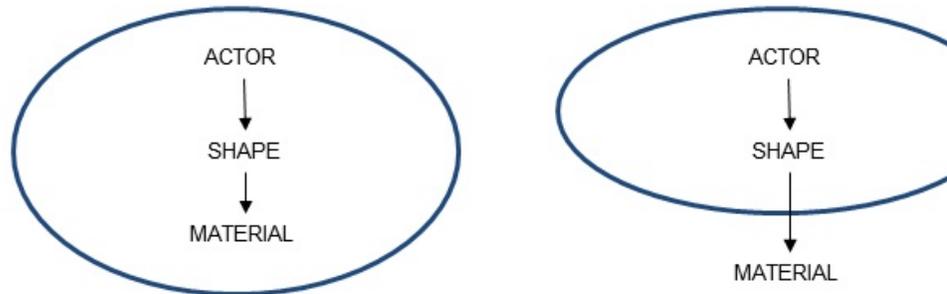


Figure 1: Left: Complete Collection, Right: Incomplete Coll

For a formal definition please refer to [Complete](#).

Both complete and incomplete collections can be serialized, but when an incomplete collection is serialized, references to objects which were not serialized are not resolved. The following two sections describe how PhysX collections can be serialized using the binary format or RepX. The first section shows how to serialize complete collections, and the second section shows how to deserialize incomplete collections.

Serializing Complete Collections

This code snippet shows how to prepare a collection of PhysX objects (e.g. an actor, its shapes, and the materials and meshes they reference).

```
PxPhysics* physics; // Th
PxRigidDynamic* dynamic = PxCreateDynamic(...); // Cr

//Cre
PxSerializationRegistry* registry = PxSerialization::createSerial

PxCollection* collection = PxCreateCollection(); // Cr
collection->add(*dynamic); // Ad
```

```
PxSerialization::complete(*collection, *registry);           // Ad
                                                           // re
                                                           // de
```

Instead of using *PxSerialization::complete* it is possible to manually add objects to the collection. All objects the *PxRigidBody* references added and then all objects referenced by the newly added objects would be added as well and so forth. See definitions: *Requires*, *Complete*.

By default *PxSerialization::complete* follows references from joints to actors and from actors to their joints. The *followJoint* parameter can be used to control the order of *PxSerialization::complete* to add the joints attached to each actor. This way actor-joint chains can be added to the collection.

When all the necessary objects have been added to a collection, create an instance of the *PxOutputStream* interface, then serialize the collection:

```
PxCollection* collection;                               // Comp
PxSerializationRegistry* registry;                     // Regi
PxOutputStream& outStream = ...;                       // Impl

// Serialize

// Binary
    PxSerialization::serializeCollectionToBinary(outStream, *collection);
//~Binary

// RepX
    PxSerialization::serializeCollectionToXml(outStream, *collection);
//~RepX

// Collection and registry can be released if they are no longer
// Note that releasing the collection will not release the contained
collection->release();
registry->release();
```

Note: Serialization of objects in a scene that is simultaneously being simulated is not supported and leads to undefined behavior.

The following code shows how to deserialize a collection from a memory buffer.

```
PxSerializationRegistry* registry;           // Registry
PxCooking* cooking;                         // Cooking
                                             // Instance

// Deserialize

// Binary
void* memory128 = ...;                      // A 128-byte
                                             // load

PxCollection* collection = PxSerialization::createCollectionFromMemory(
//~Binary

// RepX
PxInputData& inputData = ...;              // Input
PxCollection* collection = PxSerialization::createCollectionFromMemory(
//~RepX
```

To add all the objects to the scene and release the collection and registry:

```
PxScene* scene;                             // The scene
scene->addCollection(*collection);
collection->release();
registry->release();
```

See [Serializable](#) for the exact set of conditions a collection must satisfy to be serialized. These conditions can be checked with `PxSerialization::isSerializable`.

Serializing Incomplete Collections

Another common use case is where a collection of actors and joints - or meshes - must be deserialized multiple times, with each instance sharing the same meshes. To achieve this, serialize two collections:

- a collection A of the materials and meshes that will be deserialized
- a collection B of actors and joints which will be copied and deserialized

Collection B is *incomplete*, since it contains references to objects in A. In the serialized format will remember each reference to an object in A as (if it doesn't have an ID, then serialization will fail.) As long as an object with a matching ID is supplied when deserializing collection B, the references are resolved. Although collection B is incomplete, it is also said to be *complete* relative to collection A. For a formal definition of complete please refer to [Complete](#)

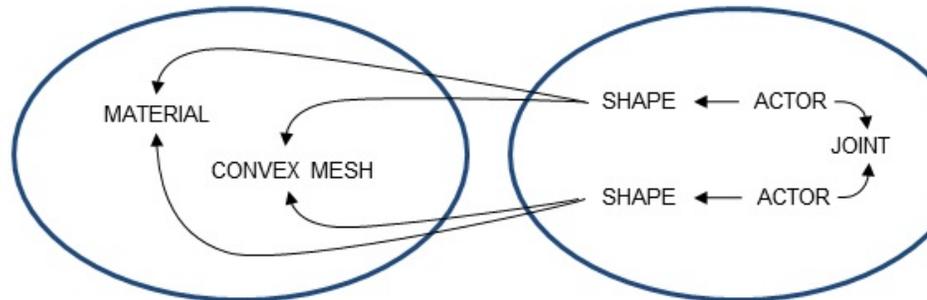


Figure 2: Left: Collection A with Sharable Objects, Right: Collection B

Concretely, to serialize and deserialize an incomplete collection:

- At serialization time, provide IDs for all objects in collection A that are referenced by objects in collection B.
- When deserializing, provide a collection with matching IDs for all the objects that were referenced by objects in B.

Here are examples of how the application can provide identities (IDs) to express requirements of one collection to another. This can be done by adding the object with:

```
PxCollection* collection;
PxTriangleMesh* triMesh;
PxSerialObjectId triMeshId = 1;

collection->add(*triMesh, triMeshId);
```

Or set the ID after adding the object:

```
collection->add(*triMesh);  
collection->addId(*triMesh, triMeshId);
```

There is a helper function to generate IDs for all objects in a collection. There are no IDs yet:

```
PxSerialObjectId baseId = 1;  
  
PxSerialization::createSerialObjectIds(*collection, baseId);
```

Already used ID values will be skipped by *createSerialObjectIds*, as values that already have IDs.

After providing correct IDs, all required objects have been added to the collection and serialized, but without adding the objects that are intended to be referenced. The function in *PxSerialization* supports completing a collection relative to another collection.

```
PxSerializationRegistry* registry;  
  
PxCollection* collectionB;  
PxCollection* collectionA;  
  
PxSerialization::complete(*collectionB, *registry, collectionA);
```

Serialization example:

```
PxConvexMesh** convexes;           // An array of mNbConvexes c  
PxRigidDynamic** actors;          // An array of mNbConvexes a  
  
PxSerializationRegistry* registry; // Registry for serializable  
PxOutputStream& convexStream;      // Output stream for the con  
PxOutputStream& actorStream;       // Output stream for the act  
  
PxCollection* convexCollection = PxCreateCollection();  
PxCollection* actorCollection = PxCreateCollection();
```

```

// Add convexes to collection
for(PxU32 i=0;i<mNbConvexes;i++)
    convexCollection->add(*convexes[i]);

// Create IDs for the convexes, starting with 1
PxSerialization::createSerialObjectIds(*convexCollection, PxSeriali

// Serialize the convexes along with their IDs

// Binary
    PxSerialization::serializeCollectionToBinary(convexStream, *co
//~Binary

// RepX
    PxSerialization::serializeCollectionToXml(convexStream, *conve
//~RepX

// Add actors to other collection
for(PxU32 i=0;i<mNbActors;i++)
    actorCollection->add(*actors[i]);

// Add all required objects except the convexes
PxSerialization::complete(*actorCollection, *registry, convexColl

// Serialize the actors with references to convexCollection

// Binary
    PxSerialization::serializeCollectionToBinary(actorStream, *ac
                                convexCollection
//~Binary

// RepX
    PxSerialization::serializeCollectionToXml(actorStream, *actor
                                convexCollection);
//~RepX

// Release collections and registry
convexCollection->release();
actorCollection->release();
registry->release();

```

Deserialization example:

```

PxPhysics* physics; // The physics SDK ob
PxSerializationRegistry* registry // Registry for seria

```

```

PxCooking* cooking; // Cooking lib needed
PxScene* scene; // The scene into whi

// Deserialize convexes along with their IDs (no external depende

// Binary
void* convexMemory128; // Aligned memory con
PxCollection* convexCollection =
    PxSerialization::createCollectionFromBinary(convexMemory1
//~Binary

// RepX
PxInputData& convexInputData = ...; // Implemented by the
PxCollection* convexCollection =
    PxSerialization::createCollectionFromXml(convexInputData,
//~RepX

// Deserialize actors referencing the convexCollection

// Binary
void* actorMemory128; // Aligned memory con
PxCollection* actorCollection =
    PxSerialization::createCollectionFromBinary(actorMemory12
//~Binary

// RepX
PxInputData& actorInputData = ...; // Implemented by the
PxCollection* actorCollection =
    PxSerialization::createCollectionFromXml(actorInputData,
convexCollection
//~RepX

// Release convex collection
convexCollection->release();

// Add actors to scene and release collection and registry
scene->addCollection(*actorCollection);
actorCollection->release();
registry->release();

```

The next example shows how to deal with situations where the seriali: objects that are not serialized and deserialized but created by other me:

```

PxSerializationRegistry* registry; // Registry for serializable
PxMaterial** materials; // Created procedurally by ap

```

```

PxRigidDynamic** actors;           // An array of mNbConvexes ac
PxOutputStream& actorStream;       // Output stream for the acto

// Add materials with IDs to collection
PxCollection* materialCollection = PxCreateCollection();

for(PxU32 i=0;i<mNbMaterials;i++)
    materialCollection->add(*materials[i], PxSerialObjectId(i+1))

// Create actor collection, complete and serialize
PxCollection* actorCollection = PxCreateCollection();

for(PxU32 i=0;i<mNbActors;i++)
    actorCollection->add(*actors[i]);

PxSerialization::complete(*actorCollection, *registry, materialCo

// Binary
    PxSerialization::serializeCollectionToBinary(actorStream, *ac
        materialCollecti

//~Binary

// RepX
    PxSerialization::serializeCollectionToXml(actorStream, *actor
        materialCollection)

//~RepX

actorCollection->release();
materialCollection->release();           // Note that materialColl
registry->release();

```

Deserialization:

```

PxScene* scene;                     // The scene into which t
PxSerializationRegistry* registry;  // Registry for serializa
PxCooking* cooking;                 // Cooking library needed
PxMaterial** materials;              // Created procedurally b

// recreate material collection with consistent IDs, no deseriali
PxCollection* materialCollection = PxCreateCollection();

for(PxU32 i=0;i<mNbMaterials;i++)
    materialCollection->add(*materials[i], PxSerialObjectId(i+1))

// Deserialize actors with reference material collection

```

```

// Binary
    void* actorMemory128;           // aligned memory containi
    PxCollection* actorCollection =
        PxSerialization::createCollectionFromBinary(actorMemory12
            materialCollection);
//~Binary

// RepX
    PxInputData& actorInputData = ...; // Implemented by the ap
    PxCollection* actorCollection =
        PxSerialization::createCollectionFromXml(actorInputData,
            materialCollecti

//~RepX

materialCollection->release();
scene->addCollection(*actorCollection);
actorCollection->release();
registry->release();

```

Reference Counting of Deserialized Objects

This section assumes the background in [Reference Counting](#).

Objects that are created by deserialization are always created with a reference. The application needs to give up by explicitly calling *release()*. The information that the application gave up a reference to an object is **not** preserved on serialization.

See [Shapes](#) for a discussion of the method *PxRigidActorExt::createShape* which automatically releases the initial reference to the shape, leaving no reference. Again, the information that this reference has been released is not preserved on serialization.

Example for shapes:

```

PxOutputStream& outStream;           // Output stream for the coll
PxSerializationRegistry* registry; // Registry for serializable
PxRigidActor* actor;                 // Any actor

// Creating shapes in different ways implies different rules for

```

```

// Shape is automatically released when actor gets released
PxShape* shapeA = PxRigidActorExt::createExclusiveShape(*actor, .

// Shape is either created as "shared" or "exclusive" and needs t
// the application
PxShape* shapeB = PxGetPhysics().createShape(...);
actor->attachShape(*shapeB);

// Create collection with actor and shapes and serialize
PxCollection* collection = PxCreateCollection();
collection->add(*actor);
collection->add(*shapeA);
collection->add(*shapeB);
PxSerialization::serializeCollectionToBinary(outStream, *collecti
collection->release();

// Releasing actors and shapes
actor->release();    // Releases actor and shapeA (automatically)
shapeB->release();  // Releases shapeB (necessary since shapeB w

// Deserialize collection
...
void* memory128 = ...; // Aligned memory for serialized data
collection = PxSerialization::createCollectionFromBinary(memory12

// Release actors and release ALL shapes (necessary since shape c
// not preserved across serialization
for(PxU32 i = 0; i < collection->getNbObjects(); i++)
{
    switch ( collection->getObject(i).getConcreteType() )
    {
        case PxConcreteType::eRIGID_DYNAMIC:
        case PxConcreteType::eRIGID_STATIC:
            static_cast<PxActor*>(collection->getObject(i)).relea
            break;
        case PxConcreteType::eSHAPE:
            static_cast<PxShape*>(collection->getObject(i)).relea
            break;
    }
}
}

```

Note: There is a PhysXExtensions function to release all objects with *PxCollectionExt::releaseObjects*.

Reconnecting PhysX and Game-Objects

Here is an example of how to fix up references with gameplay objects in a collection:

```
PxPhysics* physics;           // The physics SDK object
PxCooking* cooking;          // Cooking library needed
PxSerializationRegistry* registry; // Registry for serializa

// Deserialize objects along with IDs

// Binary
void* memory128;              // Aligned memory contain
PxCollection* collection =
    PxSerialization::createCollectionFromBinary(memory128, *r
//~Binary

// RepX
PxInputData& inputData = ...; // Implemented by the ap
PxCollection* collection =
    PxSerialization::createCollectionFromXml(actorInputData,
                                             materialCollecti
//~RepX

// Receive a list of all deserialized IDs
#define MAX_IDS 100
PxSerialObjectId idBuffer[MAX_IDS];
PxU32 numIds = collection->getIds(idBuffer, MAX_IDS);

// iterate over the list to patch up gameplay objects
for (PxU32 i = 0; i < numIds; i++)
{
    PxActor* actor = collection->find(idBuffer[i])->is<PxActor>()
    if (actor)
    {
        // this assumes that findGamePlayObjectFromId is able to
        // the corresponding game play object from a PxSerialObj
        actor->userData = findGamePlayObjectFromId(idBuffer[i]);
    }
}
}
```

Alternatively *PxCollection::getObjects(...)* and *PxCollection::getId(PxBa* used to achieve the same.

Serializing Everything

PhysX provides two utility functions for serializing the entirety of the scene:
`PxCollectionExt::createCollection(PxPhysics& physics, PxScene* scene, PxSerializationRegistry* registry, PxOutputStream& outStream, bool sdk)`
`PxCollectionExt::createCollection(PxScene& scene, PxSerializationRegistry* registry, PxOutputStream& outStream, bool sdk)`

```
PxPhysics* physics;           // The physics SDK object
PxScene* scene;              // The physics scene
PxSerializationRegistry* registry; // Registry for serializable
PxOutputStream& outStream;    // The user stream doing the

// 1) Create a collection from the set of all objects in the physics
//     multiple scenes.
PxCollection* everythingCollection = PxCollectionExt::createCollection(
    physics, scene, registry, outStream, true);

// 2) Create a collection from all objects in the scene and add it
//     to everythingCollection.
PxCollection* collectionScene = PxCollectionExt::createCollection(
    scene, registry, outStream, true);
everythingCollection->add(collectionScene);
collectionScene->release();

// 3) Complete collection
PxSerialization::complete(*everythingCollection, *registry);

// 4) serialize collection and release it

// Binary
PxSerialization::serializeCollectionToBinary(outStream, *everythingCollection);
//~Binary

// RepX
PxSerialization::serializeCollectionToXml(outStream, *everythingCollection);
//~RepX

everythingCollection->release();
registry->release();
```

Deserialization is as previously:

```
PxScene* scene;           // The physics scene
PxCooking* cooking;      // Cooking library needed for
PxSerializationRegistry* registry; // Registry for serializable
```

```

// Binary
void* memory128 = ...;           // a 128-byte aligned buffer
                                // by the user
PxCollection* everythingCollection =
    PxSerialization::createCollectionFromBinary(memory128, *r
//~Binary

// RepX
PxInputData& inputData = ...;    // Implemented by the applic
PxCollection* everythingCollection =
    PxSerialization::createCollectionFromXml(inputData, *cook
//~RepX

scene->addCollection(*everythingCollection);
everythingCollection->release();
registry->release();

```

Serializability

This section contains various definitions to describe serializability of a collection. A collection can be successfully serialized and deserialized, optionally references collection, can be queried by calling *PxSerialization::isSerial*

Requires

An object **A** requires another object **B** if **A** maintains a reference to **B** that is established for successfully deserializing **A**. This implies that **B** needs to be deserialized before **A**.

Here is the table of the relationship **requires** of all PhysX objects:

joints	require their actors and constraint
rigid actors	require their shapes
shapes	require their materials and mesh (triangle mesh, convex field), if any
articulations	require their links and joints
aggregates	require their actors
cloth actors	require their cloth fabric

Subordinate

Subordinates are objects that cannot be instantiated without being objects. An articulation link, for example, can only be instantiated as part of an object.

The following three types are **subordinates**:

articulation links
articulation joint
constraints

Complete

Definition of a complete set:

A set of objects **C** is **complete** if every object **required** by **C** is in **C**.

Definition of a set that is complete relative to another set:

A set of objects **C** is **complete** relative to a set **D** if every object **required** by **C** is in **D**. This means that **C** can be deserialized given **D**.

Serializable

Here is the complete set of requirements on a collection **C** with dependencies such that **C** can be serialized:

- **C** is complete relative to **D**. ("no dangling references")
- Every object in **D** required by an object in **C** has a valid reference to an object in **C**. ("no dangling references")
- Every subordinate object in **C** is required by another object in **C**. ("no dangling references")

Binary Serialization Specifics

The following sections describe specific properties of the binary serialization process.

Memory Management

Management of memory blocks containing deserialized objects is left user's responsibility to:

- allocate the memory block. Note that it must be proper `PX_SERIAL_FILE_ALIGN` (128) bytes boundary.
- fill the block with serialized data, typically by loading it from disk.
- deallocate the memory block when the objects within have been released.

Although the user owns the memory block, the PhysX runtime owns objects it contains. Concretely, calling `release()` on an object that was deserialized will cause its destructor to run, but will not deallocate the memory block. The memory block is deallocated before the destructors have run for all the objects. If the PhysX runtime will likely crash. For more information about how deserialized objects should be released see [Reference Counting of Deserialized Objects](#).

Versioning

The binary serialized data is typically specific to the version of the SDK with which it was created. However, a SDK version can load the data of older SDK versions if the data format didn't change. This is usually the case with bugfix releases. The compatibility rules are listed in the code documentation of `PX_BINARY_SERIALIZED` in `PxSerialization.h`.

Retargeting to other Platforms

Binary serialized data is platform-specific, and when serialized it is specific to the platform on which it was created. The binary converter in the extension can convert the data from one platform to another. Typically assets are serialized on an x86 Windows (Windows, Mac OS X and Linux). The serialized data can then be retargeted to a console or any other runtime platform.

The converter requires meta-data for the source and target platform

information about the binary layout of objects for that platform. To obtain the function provided in the extensions library for each platform:

```
void PxSerialization::dumpBinaryMetaData(PxOutputStream& stream,
```

On each target platform, run it once and keep generated data around. Some of pre-built binary metadata is included with the PhysX SDK at [path SDK]/Tools/BinaryMetaData.

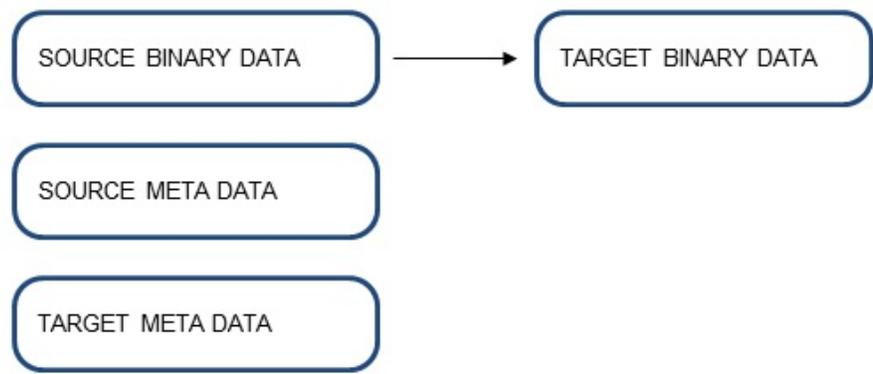


Figure 3: Schema of Retargeting

Assuming that the extensions library has been initialized, conversion follows:

```
PxSerializationRegistry* registry; // Registry for serializa  
PxInputStream& srcMetadata; // metadata for the 'from  
 // (e.g. PxDefaultFileInp  
PxInputStream& dstMetadata; // metadata for the 'to'  
  
PxInputStream& srcAsset; // stream containing sour  
PxU32 srcAssetSize; // size of the source ass  
PxOutputStream& dstAsset; // output stream for reta  
  
PxBinaryConverter* converter = PxSerialization::createBinaryConve  
converter->setMetaData(srcMetadata, dstMetadata);  
converter->convert(srcAsset, srcAssetSize, dstAsset);
```

The Convert Tool

The convert tool is at [path to installed PhysX SDK]/Snippets/SnippetC how to convert PhysX 3 serialized binary files from one platform compiles and runs on authoring platforms (Windows, MacOs and Linux)

SnippetConvert is a simple command-line tool supporting the following (

```
--srcMetadata=<filename>      Defines source metadata file
--dstMetadata=<filename>      Defines target metadata file
--srcBinFile=<filename>       Source binary file to convert
--dstBinFile=<filename>       Outputs target binary file
--generateExampleFile=<filename> Generates an example file
--verbose                      Enables verbose mode
```

Object Names

Some SDK objects, such as shapes and actors, can be given *PxShape::setName()* and *PxActor::setName()* functions. By default the serialized. The 'exportNames' parameter *PxSerialization::serializeCollectionToBinary()* can be set to true in or names along with the objects.

API-level Serialization (RepX) Specifics

RepX stands for Representation X and is the ASCII-XML serialization f As opposed to binary serialization, the RepX XML serialization is not in performance critical or memory constrained situations. The following specifics of the RepX XML serialization system.

Upgrading RepX Data

Upgrading RepX data from an older PhysX version to a newer one i implicitly when deserializing old RepX data with a newer PhysX SDK ar resulting PxCollection.

Example for upgrading a RepX stream:

```
PxPhysics* physics; // The
// Phxs
PxCooking* cooking; // Cook
// inst
PxSerializationRegistry* registry; // Regi

PxDefaultFileInputData inputData(pathTo30RepXFile); //load
PxCollection* collection =
    PxSerialization::createCollectionFromXml(inputData, *cooking,

PxDefaultFileOutputStream outputStream(pathToNewRepXFile);
PxSerialization::serializeCollectionToXml(outputStream, *collection,
```

Object Names

As opposed to binary serialization, the object names that can be *PxShape::setName()* and *PxActor::setName()* functions, are always serialized format. On deserialization with *PxSerialization::createCollect* names can be recovered by setting the *PxStringTable* parameter.

If *PxStringTable* parameter is set, the names will live within the memory by the string table. The string table must not be released unless it can the names will not be accessed any more.

Caching Cooked Geometry Data

In order to facilitate faster instantiation of XML data, it is possible to serialization to store the cooked triangle and convex mesh data along The cooked data caching can be enabled by passing a *PxCoo*. *PxSerialization::serializeCollectionToXml(...)*. The cached cooked data format is incompatible with the current SDK version.

Common Use Cases

API-level RepX serialization should be used whenever compatibility and performance are important. The PhysX plug-ins for the DCC tools 3ds Max and Maya export PhysX objects. The resulting RepX files can then be deserialized at the PhysX runtime. This is useful for rapid prototyping or for generating assets if performance is not of a big concern. For quick loading of assets, convert RepX data into binary serialized data. RepX is also useful in situations with unwanted behavior without the need to provide the whole scene. For this, the application may be connected to the PhysX Visual Debugger. The debugger records the scene of interest. A representative frame can then be saved from within PVD (see [PVD](#)).

Binary serialization should be used in performance and memory constrained environments. The main target use-case is streaming in chunks of a large game world. Loading the world into memory at once. Creating and loading save games is another use-case that could be optimized by using binary serialization. PhysX objects in binary format can be sent over the network to enable efficient game state synchronization.

Figure 4: SnippetSerialization

SnippetConvert

SnippetConvert illustrates how binary serialized data can be re-targeted platform to a runtime platform such as a console. The snippet is a simple tool that can load a binary serialized data file along with meta data file and destination platforms and then output a converted binary data file. [SnippetConvert's source documentation](#) for more details on usage.

SnippetLoadCollection

SnippetLoadCollection shows how to deserialize serialized collections or XML format. The snippet is a command line tool that can connect to a Debugger application and display the content of serialized collections. [SnippetLoadCollection's source documentation](#) for more details.

Best practices / Troubleshooting

- Concurrent simulation and serialization is not supported and I behavior.
- If releasing PhysX objects leads to crashes or errors it is possible is releasing some objects twice. The following two reasons should A potential source of error is to release PhysX objects without u referencing these objects. 2.) Shapes that where created through application reference automatically released on creation. If such a and deserialized the creation history will be lost. It might be con extension function *PxCollectionExt::releaseObjects* because it dea cases as required. See [Reference Counting of Deserialized Object](#).
- If accessing binary deserialized PhysX objects, including accesses causes crashes it might be due to the premature release of the holds the deserialized objects.
- If binary files are too large and/or too slow to load it might be that s been serialized multiple times. An example of a shared asset might referenced by multiple shapes. The solution is to separate shared a separate collection. See [Serializing Incomplete Collections](#).
- If loading PhysX objects from RepX files is too slow two things sho 1.) Could binary serialization be used instead? Even for debug sense to convert RepX files into binary serialized data by re-serial binary approach. 2.) Meshes tend to load very slowly from serialization offers an option to cache cooked mesh data by in-lini the RepX file. If such a cache is present and valid, the loa significantly faster. See [Caching Cooked Geometry Data](#).

PVD

The PhysX Remote Debugger provides the functionality to export single scenes as RepX files. The resulting files can be used to playback a scene state. In many cases this is sufficient to isolate an issue. The option is in the File menu of PVD: [Menu > File > Export Current Frame To RepX]

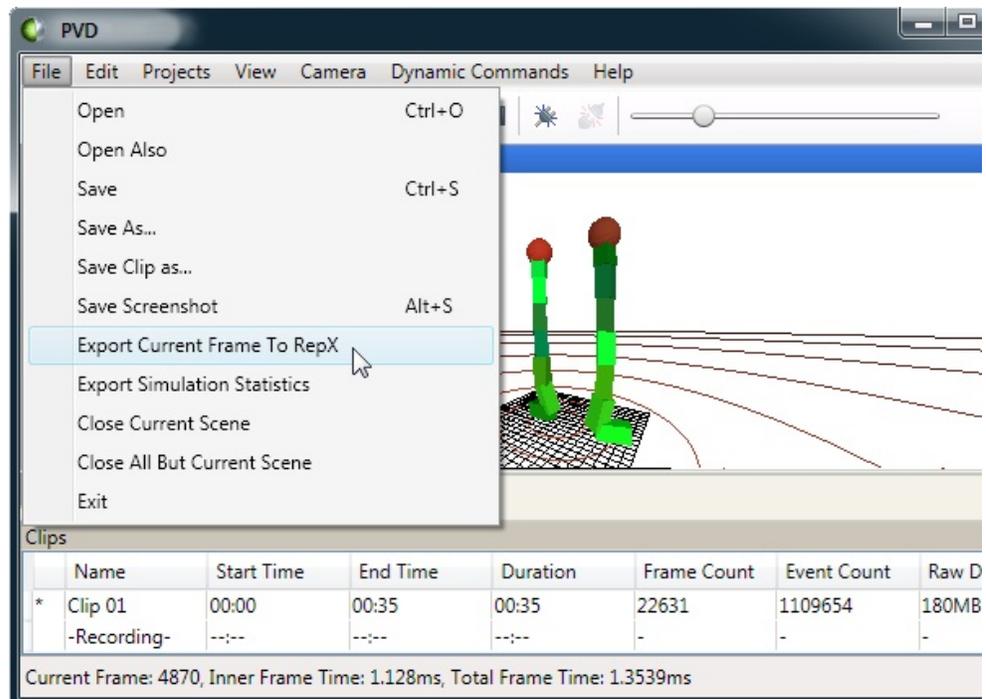


Figure 5: RepX Functionality in PVD

Introduction

The PhysX serialization system (*Serialization*) is extendable to custom applications. If an application were to require a new joint type, for example, the serialization system would be extended to add support for serialization of that new joint type.

The following document contains some recipes and example code that show how the serialization system may be extended to custom types. It doesn't cover all the details of the extension mechanisms. It is therefore advisable to look into the following example for more details:

- PhysXVehicle library (PhysXVehicle/src)

Overview

Both binary and RepX serialization can be extended for custom types. For a custom type to be added to a *PxCollection*, which is a pre-requisite for core serialization functionality, it must first inherit from *PxBase*. This allows the custom type to be added to a *PxCollection*, which is a pre-requisite for core serialization functionality. The core serialization functionality needs to be provided by implementing the *PxSerializer* interface. The template *PxSerializerDefaultAdapter* provides a default implementation that can be specialized for the custom type as required. In order to support RepX, an additional *PxRepXSerializer* interface needs to be implemented. This implementation relies on automatic code generation using clang. Scripts to run the code generation examples can be found in (Tools/PhysXMetaDataGenerator).

Binary Serialization of Custom Classes

Serialization and deserialization of a custom class can be achieved through the following steps:

1. Define a *PxConcreteType* and type info for the custom class. Make the type info unique.
2. The custom class needs to inherit from *PxBase* and implement its *Serialize* and *Deserialize* methods.
3. Instance *PxSerializerDefaultAdapter<T>* and implement specialization if necessary.
4. If retargeting to other platforms is needed, implement *getBinaryMetadata*.
5. Register the adapter and metadata, see *PX_NEW_SERIALIZE* and *PX_NEW_DESERIALIZE*.
PxSerializationRegistry::registerSerializer
PxSerializationRegistry::registerBinaryMetadataCallback. Note that the adapter needs to be unregistered before *PxSerializationRegistry::release*. The application is responsible for custom type serializer allocation and deallocation.

For pointer members the following needs to be done (Note that reference counting is currently not supported):

6. Implement *PxSerializer::requires*. It should enumerate *PxBase* objects that the current object depends on for deserialization. See *Requires*.
7. For a member pointer to another *PxBase* object, register the implementation of *PxSerializer::registerReferences*. The implementation of *PxSerializer::requires* may be used to help with this.
8. Resolve references in the implementation of *PxSerializer::convertToPxBinary* and *PxDeserializationContext::resolveReference*, *translatePxBinary*.
9. Make sure that *PxSerializer::isSubordinate* returns whether the object is serialized along with an owner object. See *Subordinate*.
10. Export non *PxBase* data by implementing *PxSerializer::exportNonPxBinary*.

PxSerializationContext::writeData, alignData.

11. Import non *PxBase* data in the implementation of *PxSerializer::PxDeserializationContext::readExtraData, alignExtraData.*

Note: In checked builds (PX_CHECKED defined as 1) metadata definitions are checked against serialized data. If metadata definitions are missing warnings are written to the error stream during re-targeting (*PxBinaryConverter::convert*). To avoid all unused memory in custom serialized class instances should be marked with the *markSerializedMem* pattern. This can be done with *Cm::markSerializedMem* from *CmUtils.h*.

Note: The memory of a deserialized class instance should not be deallocated. If the memory is embedded in the memory buffer containing the serialized data, the *PxBaseFlag::eOWNS_MEMORY* can be used to decide whether the object should be deallocated or not.

Example for a custom class:

```
#include "extensions/PxSerialization.h"
#include "common/PxTypeInfo.h"
#include "common/PxMetaData.h"
#include "common/PxSerializer.h"
#include "common/PxSerialFramework.h

using namespace physx;

const PxType customClassType = PxConcreteType::eFIRST_USER_EXTENS
PX_DEFINE_TYPEINFO(CustomClass, customClassType);

class CustomClass : public PxBase
{
    friend class PxSerializerDefaultAdapter<CustomClass>;
public:

    // constructor setting up PxBase object
    CustomClass()
    : PxBase(customClassType, PxBaseFlag::eOWNS_MEMORY | PxBaseFl
    {}

    // constructor called on deserialization
    CustomClass(PxBaseFlags baseFlags) : PxBase(baseFlags) {}
```

```

virtual ~CustomClass() {}

//PxBase
virtual const char* getConcreteTypeName() const { return "Cus

virtual bool isKindOf(const char* name) const
{
    return !strcmp("CustomClass", name) || PxBase::isKindOf(n
}
//~PxBase

//PxSerializationRegistry::registerBinaryMetaDataCallback
static void getBinaryMetaData(PxOutputStream& stream)
{
    PX_DEF_BIN_METADATA_VCLASS(stream, CustomClass)
    PX_DEF_BIN_METADATA_BASE_CLASS(stream, CustomClass, PxBas

    PX_DEF_BIN_METADATA_ITEM(stream, CustomClass, PxRigidDyna
        PxMetaDataFlag::ePTR)
    PX_DEF_BIN_METADATA_ITEM(stream, CustomClass, char, mBuf,
    PX_DEF_BIN_METADATA_ITEM(stream, CustomClass, PxU32, mSiz

    PX_DEF_BIN_METADATA_EXTRA_ITEMS(stream, CustomClass, char
}
//~PxSerializationRegistry::registerBinaryMetaDataCallback

private:
    PxRigidDynamic* mActor;    //add in requires
    char* mBuf;                //extra data
    PxU32 mSize;                //size of mBuf
};

//PxSerializerDefaultAdapter
template<>
void PxSerializerDefaultAdapter<CustomClass>::requires(PxBase& ob
    PxProcessP
{
    CustomClass* custom = obj.is<CustomClass>();
    PX_ASSERT(custom);
    c.process(*custom->mActor);
}

template<>
void PxSerializerDefaultAdapter<CustomClass>::registerReferences(

```

```

{
    CustomClass* custom = obj.is<CustomClass>();
    PX_ASSERT(custom);

    s.registerReference(obj, PX_SERIAL_REF_KIND_PXBASE, size_t(&
s.registerReference(*custom->mActor, PX_SERIAL_REF_KIND_PXBAS
}

template<>
void PxBinarySerializerDefaultAdapter<CustomClass>::exportExtraData(PxB
                                                                    PXS
{
    CustomClass* custom = obj.is<CustomClass>();
    PX_ASSERT(custom);
    s.alignData(PX_SERIAL_ALIGN);
    s.writeData(custom->mBuf, custom->mSize);
}

template<>
PxBinary* PxBinarySerializerDefaultAdapter<CustomClass>::createObject(PxU
                                                                    PxD
                                                                    con
{
    CustomClass* custom = new (address) CustomClass(PxBinaryFlag::e
address += sizeof(CustomClass);

    // resolve references
    context.translatePtr(custom->mActor);

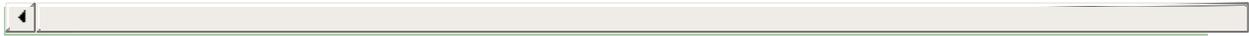
    // import extra data
    custom->mBuf = context.readExtraData<char*, PX_SERIAL_ALIGN>(

    // return deserialized object
    return custom;
}
//~PxBinarySerializerDefaultAdapter

void registerCustomClassBinarySerializer(PxBinarySerializationRegistry&
{
    registry.registerSerializer(customClassType, PX_NEW_SERIALIZE
registry.registerBinaryMetadataCallback(CustomClass::getBinar
}

void unregisterCustomClassBinarySerializer(PxBinarySerializationRegistr
{
    PX_DELETE_SERIALIZER_ADAPTER(registry.unregisterSerializer(cu
}

```



RepX Serialization of Custom Classes

Serialization and deserialization of a custom class can be achieved through the following steps:

1. Perform the first three steps from *Binary Serialization of Custom Classes*. The *PxSerializer* and *PxSerializerDefaultAdapter<T>* required for custom serialization may be left empty.
2. Create a custom RepX serializer that implements the *PxRepXSerializer*. The *PxRepXSerializer* is used to create an object from the xml file and the *RepXSerializerImpl* can be used for the implementations of some methods.
3. Register the general serializer adapter and the RepX serializer. No other serializers also need to be unregistered and deallocated.
4. RepX supports automatic reading and writing of class properties. *PhSystem* has to be used to generate corresponding metadata: *PhSystem*.

Example for a custom class:

```
#include "SnRepXSerializerImpl.h"

const PxType customClassType = PxConcreteType::eFIRST_USER_EXTENS
PX_DEFINE_TYPEINFO(CustomClass, customClassType);

struct CustomClassRepXSerializer : public RepXSerializerImpl<CustomClass>
{
    CustomClassRepXSerializer(PxAllocatorCallback& inCallback)
        : RepXSerializerImpl<CustomClass>(inCallback)
    {}

    virtual PxRepXObject fileToObject(XmlReader& inReader, XmlMem
        PxRepXInstantiationArgs& inArgs, PxCollection* inCollecti
    {
        // factory for CustomClass instance provided by applicati
        CustomClass* object = createCustomClass();
    }
};
```

```

        // when using the PhysX API metadata system readAllProperties
        // all properties automatically
        readAllProperties(inArgs, inReader, object, inAllocator,

        return PxCreateRepXObject(object);
    }

    virtual void objectToFileImpl(const CustomClass* obj, PxCollector
        XmlWriter& inWriter, MemoryBuffer
        PxRepXInstantiationArgs&)
    {
        // when using the PhysX API metadata system writeAllProperties
        // all properties automatically
        writeAllProperties(obj, inWriter, inTempBuffer, *inCollector
    }

    // this can return NULL if fileToObject(...) is overwritten with
    virtual CustomClass* allocateObject(PxRepXInstantiationArgs&)
};

void registerCustomClassRepXSerializer(PxSerializationRegistry& r
{
    registry.registerSerializer(customClassType,
        PX_NEW_SERIALIZER_ADAPTER(CustomClass,
        registry.registerRepXSerializer(customClassType,
        PX_NEW_REPX_SERIALIZER<CustomClass>);
}

void unregisterCustomClassRepXSerializer(PxSerializationRegistry&
{
    PX_DELETE_SERIALIZER_ADAPTER(registry.unregisterSerializer(customClassType);
    PX_DELETE_REPX_SERIALIZER(registry.unregisterRepXSerializer(customClassType);
}

```

Note: Implementing a PxRepXSerializer is currently not practical with internal PhysXExtension header "SnRepXSerializerImpl.h".

PhysX API Metadata System

This system produces a set of objects that are analogues of the

descriptors in the PhysX system, all based on the public interface. The generator heuristically finds functions that start with get/set and, through a series of transformations, combines those into several types of properties.

Currently the generator supports the following property types:

- Basic property
 - `{ptype} get{pname}() const;`
 - `void set{pname}(const ptype& prop);` //plus variations
 - read-only, write-only variants of above.
- Range property
 - `void get{pname}({ptype}& lowEnd, {ptype}& highEnd);`
 - `void set{pname}({ptype} lowEnd, {ptype} highEnd);`
- Indexed property
 - `{ptype} get{pname}(enumType idx);`
 - `void set{pname}(enumType idx, const {ptype}& prop);`
- Dual indexed property (like above, but with two enumeration indexes)
- Collection
 - `PxU32 getNb() const;`
 - `PxU32 get({ptype}* buffer, PxU32 count);`
 - `void set({ptype}* buffer, PxU32 count);`

In order to make use of the generator the following files need to be generated according to the following recipe:

- CustomTypeExtensionAPI.h
 - Add all the types that should be exported to `gUserPhysXTypes`
 - Add the unnecessary types to `gAvoidedPhysXTypes`. It contains metadata information for these types.
 - Be sure to append the included files for these types.
- `runClang_[windows|osx|linux].[bat|sh]` (e.g. `runClang_windows.bat`)

- Set definition folder for these autogenerated files and set the s
- Specify the filename of autogenerated files. Then it will gen files:

```
include/CustomTypeAutoGeneratedMetaDataObjectNames.h
include/CustomTypeAutoGeneratedMetaDataObjects.h
src/CustomTypeAutoGeneratedMetaDataObjects.cpp
```

- CustomTypeMetaDataObjects.h
 - CustomTypePropertyInfoName has to be CustomTypeAutoGeneratedMetaDataObjects.h has to be in The file will then export the properties of the custom class ar for implementing the custom RepX serializer.
- CustomTypeMetaDataObjects.cpp
 - This file is optional. It is only required when custom properties

PxVehicle serialization is a useful example. With Source/PhysXVehicle the structure of the files is as follows:

```
src/PhysXMetaData/include/PxVehicleMetaDataObjects.h
src/PhysXMetaData/src/PxVehicleMetaDataObjects.cpp
../../Tools/PhysXMetaDataGenerator/PxVehicleExtensionAPI.h
../../Tools/PhysXMetaDataGenerator/generateMetaData.py
```

Running the script will auto-generate the following files:

```
src/PhysXMetaData/include/PxVehicleAutoGeneratedMetaDataObjectNam
src/PhysXMetaData/include/PxVehicleAutoGeneratedMetaDataObjects.h
src/PhysXMetaData/src/PxVehicleAutoGeneratedMetaDataObjects.cpp
```

1. PxVehicleExtensionAPI.h: The type DisabledPropertyEntry is used which do not require export. CustomProperty is for properties customized and gUserPhysXTypes is for general properties that ne

2. `runClang_[windows|osx|linux].[bat|sh]`: The target director `src/PhysXMetaData`, and the target name is `PxVehicle`.
3. `PxVehicleMetaDataObjects.h`: It defines the custom property `PxVehicleAutoGeneratedMetaDataObjects.h`
4. `PxVehicleMetaDataObjects.cpp`: It implements the custom properties

Note: The properties defined in `PxVehicleAutoGeneratedMetaDataObjects.h` are written to the RepX file automatically if `PxVehicleMetaDataObjects.h` is a custom RepX serializer.

[NVIDIA PhysX SDK 3.4.1 Documentation »](#)
[User's Guide »](#)

[previous](#) | [next](#)

Introduction

This chapter covers a number of best practices for the PhysX SDK to avoid and fixing frequently encountered issues.

Debugging

The PhysX SDK contains a few debugging helpers. They can be used if your scenes are properly set up.

Use checked builds and the error stream

The PhysX SDK has different build configurations: Debug, Checked, and Release. For Debug and Checked builds, make sure that the scene is properly set up without warnings or errors. For Release builds, monitor the error callback. Please refer to the [Reporting](#) chapter for details. Note that some checks can be expensive and are not performed in Release or Profile builds. If the SDK silently fails in a Release build, please switch to Debug or Checked builds to ensure that you catch an uncaught error.

Visualizing physics data

Use the PhysX Visual Debugger (PVD) to see what PhysX is seeing. Verify that the physics data is what you expect it to be. Please refer to the [PhysX Visual Debugger](#) chapter for details. Note that this is only available in Debug, Checked, and Release builds.

Visualizing physics data (2)

An alternative to PVD is the built-in debug visualization system. Please refer to the [Visualization](#) chapter for details. This option is available with all build configurations.

Limiting coordinates

Bugs in applications, or issues in content creation, can sometimes occur at unexpected coordinates. We recommend using `PxSceneDesc::sanityBounds`, to generate reports when objects are in positions beyond what your application expects, or when application code moves objects to unexpected positions. Note that these bounds only apply to application objects.

coordinates, not updates by the simulation engine.

Performance Issues

The PhysX SDK has been optimized a lot in the past dot releases. However, there still exist various performance pitfalls that the user should be aware of.

Use profile builds to identify performance bottlenecks

The PhysX SDK has different build configurations: Debug, Checked, Release, and Profile. To identify performance bottlenecks, please use Profile builds and the `PxPvdInstrumentationFlag::ePROFILE` only, since enabling the other flags might negatively affect performance. Please refer to the [PhysX Visualization](#) chapter for details.

Use release builds for final performance tests

The PhysX SDK has different build configurations: Debug, Checked, Release, and Profile. Release builds are the most optimal. If you encounter a performance issue with other builds, please switch to Release builds and check if the problem is resolved.

Disable debug visualization in final/release builds

Debug visualization is great for debugging but it can have a significant performance impact. Make sure it is disabled in your final/release builds. Please refer to the [Visualization](#) chapter for details.

Debug visualization is very slow

Debug visualization can be very slow, because both the code gathering and the code rendering it is usually not optimal. Use a culling box to reduce the data the SDK gathers and sends to the renderer. Please refer to the [Culling](#) chapter for details.

Consider using tight bounds for convex meshes

By default PhysX computes approximate (loose) bounds around convex objects. `PxConvexMeshGeometryFlag::eTIGHT_BOUNDS` enables smaller/tighter bounds, which are more expensive to compute but can result in improved simulation performance. A lot of convex objects are interacting with each other. Please refer to the [Tight Bounds](#) chapter for details.

Use scratch buffers

The `PxScene::simulate` function accepts optional scratch buffers that can reduce temporary allocations and improve simulation performance. For more details, see the [Simulation](#) chapter for details.

Use the proper mid-phase algorithm

`PxCookingParams::midphaseDesc` can be used to select the desired mid-phase algorithm. It is a good idea to try the different options and see which one works best for your scene. Generally speaking the new `PxMeshMidPhase::eBVH34` introduced in PhysX 3.4 provides better performance for scene queries against large triangle meshes. For more details, see the [Geometry](#) chapter for details.

Use the proper narrow-phase algorithm

`PxSceneFlag::eENABLE_PCM` enables an incremental "persistent" narrow-phase algorithm, which is often faster than the previous implementation. It is the default algorithm since PhysX 3.4, but you can also try to enable it in PhysX 3.3 like 3.3.

Use the proper broad-phase algorithm

PhysX also supports two different broad-phase implementations: `PxSceneDesc::broadPhaseType`. The different implementations have various characteristics, and it is a good idea to experiment with both and find the one that is best for you. Please refer to the [Rigid Body Collision](#) chapter for details on broad-phases.

Use the scene-query and simulation flags

If a shape is only used for scene-queries (raycasts, etc), disable its simulation flag. If a shape is only used for simulation (e.g. it will never be raycasted), enable its scene-query flag. This is good for both memory usage and performance. Please refer to the [Rigid Body Collision](#) chapter for details.

Tweak the dynamic tree rebuild rate

If the `PxScene::fetchResults` call takes a significant amount of time in a scene with a lot of dynamic objects, try to increase the `PxSceneDesc::dynamicTreeRebuildRate` parameter. Please refer to the [Scene Queries](#) chapter for details.

Use the insertion callback when cooking at runtime

Use `PxPhysicsInsertionCallback` for objects that are cooked at runtime. It allows you to first write the data to a file or a memory buffer, and then pass the data to the physics engine.

The "Well of Despair"

One common use-case for a physics engine is to simulate fixed time-steps, independent of the frame rate that the application is rendered at. If the application is capable of being rendered at a higher frequency than the simulation frequency, it has the option to render the same simulation state, interpolate between frames, or sometimes it is not possible to render the scene at a frequency higher than the simulation frequency. At this point, the options are to either run the physics engine at a larger time-step or to simulate multiple, smaller sub-steps. The latter is the preferable solution because changing the size of time-steps in a physics engine can significantly change perceived behavior. However, when using a sub-step, one must always be aware of the potential that this has to damage performance.

As an example, let's imagine a game that is running using v-sync at 60FPS, but is simulating a large number of physics bodies and, as a result, the physics engine is expensive. In order to meet the 60FPS requirement, the entire frame is rendered using the simulation state from the previous frame.

within ~16ms. As already mentioned, the physics is reasonably expensive scenario, takes 9ms to simulate 1/60th of a second. If the game was e.g. as a result of some OS activity, saving a check-point or loading a level, we may miss the deadline for 60FPS. If this happens, we must sub-step in the physics to catch up the missed time in the next frame. If the previous frame took 50ms instead of 16ms, we must now simulate 3 sub-steps to simulate all the elapsed time. However, each sub-step takes ~9ms, so we will take ~27ms to simulate 50ms. As a result, this frame also misses the deadline for 60FPS, meaning that the frame including v-sync took 33ms. In the next frame, we must now simulate 2 sub-steps in the next frame, which takes ~18ms a 16ms deadline. As a result, we never manage to recover back to 60FPS. Our decision to sub-step as a result of a spike has resulted in our application experiencing a performance trough indefinitely. The application is capable of simulating at 60FPS but becomes stuck in the so-called "physics well of despair" due to substepping.

Problems like this can be alleviated in several ways:

- Decouple the physics simulation from the game's update/render loop. If the physics simulation becomes a scheduled event that occurs at a fixed interval, it can make player interaction in the scene more difficult and may introduce jitter. This must be well-thought through. However, using multiple scenes (or "important" objects, one asynchronous for "unimportant" objects) can help.
- Permit the game to "drop" time when faced with a short-term spike. This can introduce visible motion artifacts if spikes occur frequently.
- Introduce slight variations in time-step (e.g. instead of simulating a fixed time-step, simulating a range between 1/50th and 1/60th). This can introduce jitter into the simulation so should be used with caution. If this is done, the cost of the simulation must be amortized over several frames by using slightly larger time-steps.
- Consider simplifying the physics scene, e.g. reducing object count, reducing iteration counts etc. Provided physics simulation is a simple

total frame time, the application should find it easier to recover from

Pruner Performance for Streamed Environments

PhysX provides multiple types of pruners, each of which aimed at specific use cases. These are:

- Static AABB tree
- Dynamic AABB tree

By default, the static AABB tree is used for the static objects in the environment and the dynamic AABB tree is used for the dynamic objects in the environment. This approach works well but it must be noted that creating the static AABB tree is expensive. As a result, adding, removing or moving any static objects will result in the static AABB tree being fully recomputed, which can incur a significant performance cost. As a result, we recommend the use of dynamic AABB trees for static and dynamic pruners in games which stream in the static environment. The scene query performance against newly added objects can be improved by using PXPPruningStructure, which can precompute the AABB structure of the scene offline.

Performance Implications for Multi-Threading

The PhysX engine is designed from the ground-up to take advantage of multi-core architectures to accelerate physics simulation. However, this does not mean that more threads are always better. When simulating extremely simple scenes, adding additional worker threads can detrimentally affect performance. This is because, at the core, PhysX operates around a task queue. When a frame's simulation is dispatched, it dispatches a chain of tasks that encapsulate that frame of physics simulation. The various stages of the physics pipeline, work can be performed in parallel across multiple threads. However, if there is insufficient work, there will be little or no parallelism. In this case, the use of additional worker threads may detrimentally affect performance because the various phases of the pipeline may be run by different worker threads, which may incur some additional overhead depending on the CPU architecture.

running on just a single worker thread. As a result, developers sacrifice performance of the engine with their expected physics loads with multiple threads to maximize their performance and make sure that they are using all the available processing resources for their game.

Memory allocation

Minimizing dynamic allocation is an important aspect of performance and PhysX provides several mechanisms to control memory usage.

Reduce allocation used for tracking objects by pre-sizing the capacity of the scene structures, using either `PxSceneDesc::limits` before creating the scene or `PxScene::setLimits()`. When resizing, the new capacities will be at least as large as required to deal with the objects currently in the scene. These values are pre-allocation and do not represent hard limits, so if you add more objects than the capacity limits you have set, PhysX will allocate more space.

Much of the memory PhysX uses for simulation is held in a pool of blocks. You can control the current and maximum size of this pool with the `nbContactDataBlocks` and `maxNbContactDataBlocks` members of `PxSceneDesc`. PhysX will never allocate more than the maximum number of blocks specified in `maxNbContactDataBlocks`. If insufficient memory is available, it will instead simply drop contacts or joint constraints. You can find out how many blocks are currently in use with the `getNbContactDataBlocksUsed()` method and find out the maximum number that have ever been used with the `getMaxNbContactDataBlocksUsed()` method.

Use `PxScene::flushSimulation()` to reclaim unused blocks, and to shrink the size of data structures to the size presently required.

To reduce temporary allocation performed during simulation, provide a pre-allocated block in the `simulate()` call. The block may be reused by the `fetchResults()` call which marks the end of simulation. The size of the block must be a multiple of 16K, and it must be 16-byte aligned.

Character Controller Systems using Scene Queries and Penetration Computation

Implementing a Character Controller (CCT) is a common use case for a Scene Query (SQ) system. A popular approach is to use sweeps to implement collision detection and to improve robustness by using Geometry Queries (GQ) to compute penetration depths. However, this approach does not account for penetrations that occur due to object movement that does not account for the controller, or due to numerical precision issues.

Basic Algorithm:

1. Call a SQ-Sweep from the current position of the CCT shape to its target position.
2. If no initial overlap is detected, move the CCT shape to the target position. If an initial overlap is detected, adjust the trajectory of the CCT by removing the motion relative to the hit.
3. Repeat Steps 1 and 2 until the goal is reached, or until an SQ-Sweep detects an initial overlap.
4. If an SQ-Sweep in Step 1 detects an initial overlap, use the GQ-Penetration Depth computation function to generate a direction for depenetration. Move the CCT shape out of penetration and begin again with Step 1.

Limitations and Problems

Step 4 of the algorithm above can sometimes run into trouble due to differences in SQ-Sweep, SQ-Overlap and GQ-Penetration Depth computation. In certain initial conditions it is possible that the SQ system will detect an overlap while the GQ-Penetration Depth computation detects no penetration (or vice-versa). Penetration depth calculations involve shrinking the convex hull and performing distance calculations between the shape and the shrunken convex hull. To understand the conditions under which this occurs and how to resolve the artefacts, please refer to the diagrams below. Each diagram represents the initial conditions of two shapes (the Character Controller shape (red boxes), a convex obstacle (black boxes), at the

the algorithm above is executed. In the diagrams, the outermost rectangle represents the convex hull as seen by the SQ algorithms; the inner black box with rounded corners represents the shrunken convex shape and the black box with rounded corners represents the shrunken convex shape inflated by the amount by which we shrank the original shape. These boxes are used by the GQ-Penetration Depth computation. Although this issue occurs with convex hull obstacles, the issue is not exclusive to the convex hull shape and is similar for other shape types as well.

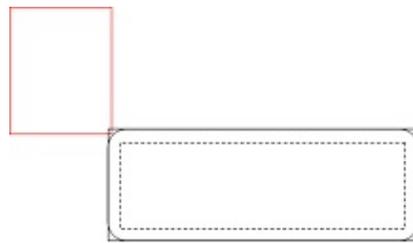


Diagram 1: CCT Shape Barely Touches an Obstacle

In **Diagram 1**, the red box of the CCT is barely touching the outermost convex obstacle. In this situation the SQ-Sweep will report an initial overlap and the Penetration Depth function will report no hit, because the red box is not a box with rounded corners.

To resolve this, inflate the CCT shape for the GQ-Penetration Depth computation so that it detects an overlap and returns a valid normal. Note that after inflation, the shape is larger than they actually are, so take this additional penetration into account when depenetrating in Step 4. This may result in some clipping around the corners of convex objects but the CCT's motion should be acceptable. As the corners become more acute, the amount of clipping will increase.

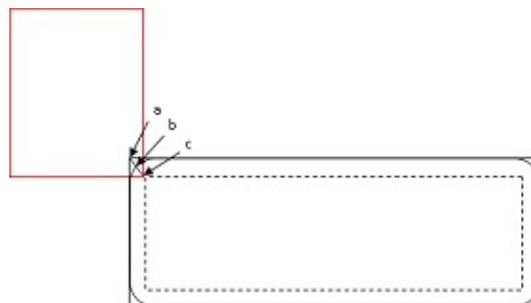


Diagram 2: CCT Overlaps an Obstacle Slightly

Diagram 2 shows a case where the CCT initially overlaps the outer boundary of the SQ system, but does not overlap the shrunken shape seen by the GQ calculator. The GQ-Penetration Depth system will return the penetration point *b* but not from point *c* to point *a*. Therefore the CCT may clip through the convex hull after de-penetration. This can be corrected in Step 4.

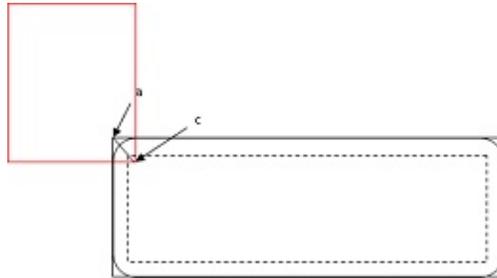


Diagram 3: CCT Overlaps an Obstacle Significantly

As can be seen from **Diagram 3**, if the CCT penetrates sufficiently into the shrunken shape seen by GQ, the GQ-Penetration Depth calculation will return penetration from point *c* to point *a*. In this case, the GQ-Penetration Depth system can be used without modification in Step 4. However, as this condition will be categorized without additional computational cost, it is best to inflate the obstacle and then subtract this inflation from the resulting penetration depth.

Unified MTD Sweep

A recent addition to the scene query sweeps is the flag `PxHitFlag::eMTD` used in conjunction with default sweeps to generate the MTD (Minimum Direction) when an initial overlap is detected by a sweep. This flag will generate an appropriate normal under all circumstances, including cases where the sweep may detect an initial overlap but calling a stand-alone MTD function may still suffer from accuracy issues with penetration depths but, in cases where the sweep is above around corners/edges, it will report a distance of 0 and the correct normal. This can be used to remove components of the sweep moving into the obstacle and then re-sweeping when attempting to implement a CCT. The

compound MTDs for meshes/heightfields, which means that it reports penetrates the shape from the entire mesh rather than just an individual MTD exists.

Quantizing HeightField Samples

Heightfield samples are encoded using signed 16-bit integers for the y-l converted to a float and multiplied by PxHeightFieldGeometry::heightS space scaled coordinates. Shape transform is then applied on top to location. The transformation is performed as follows (in pseudo-code):

```
localScaledVertex = PxVec3(row * desc.rowScale, PxF32(heightSampl
    col * desc.columnScale)
worldVertex = shapeTransform( localScaledVertex )
```

The following code snippet shows one possible way to build quanti: space heightfield coordinates from world space grid heights stored in te

```
const PxU32 ts = ...; // user heightfield dimensions (ts = terrai
// create the actor for heightfield
PxRigidStatic* actor = physics.createRigidStatic(PxTransform(PxId

// iterate over source data points and find minimum and maximum h
PxReal minHeight = PX_MAX_F32;
PxReal maxHeight = -PX_MAX_F32;
for(PxU32 s=0; s < ts * ts; s++)
{
    minHeight = PxMin(minHeight, terrainData.verts[s].y);
    maxHeight = PxMax(maxHeight, terrainData.verts[s].y);
}

// compute maximum height difference
PxReal deltaHeight = maxHeight - minHeight;

// maximum positive value that can be represented with signed 16
PxReal quantization = (PxReal)0x7fff;

// compute heightScale such that the forward transform will gener
// to the source
// clamp to at least PX_MIN_HEIGHTFIELD_Y_SCALE to respect the Ph
PxReal heightScale = PxMax(deltaHeight / quantization, PX_MIN_HEI

PxU32* hfSamples = new PxU32[ts * ts];

PxU32 index = 0;
```

```

for(PxU32 col=0; col < ts; col++)
{
    for(PxU32 row=0; row < ts; row++)
    {
        Pxi16 height;
        height = Pxi16(quantization * ((terrainData.verts[(col*ts
            deltaHeight]));

        PxHeightFieldSample& smp = (PxHeightFieldSample&)(hfSampl
        smp.height = height;
        smp.materialIndex0 = userValue0;
        smp.materialIndex1 = userValue1;
        if (userFlipEdge)
            smp.setTessFlag());
    }
}

// Build PxHeightFieldDesc from samples
PxHeightFieldDesc terrainDesc;
terrainDesc.format          = PxHeightFieldFormat::eS16_TM;
terrainDesc.nbColumns      = ts;
terrainDesc.nbRows        = ts;
terrainDesc.samples.data   = hfSamples;
terrainDesc.samples.stride = sizeof(PxU32); // 2x 8-bit material
terrainDesc.thickness      = -10.0f; // user-specified heightfie
terrainDesc.flags          = PxHeightFieldFlags();

PxHeightFieldGeometry hfGeom;
hfGeom.columnScale = terrainWidth / (ts-1); // compute column and
// height grid
hfGeom.rowScale    = terrainWidth / (ts-1);
hfGeom.heightScale = deltaHeight!=0.0f ? heightScale : 1.0f;
hfGeom.heightField = cooking.createHeightField(terrainDesc, physi

delete [] hfSamples;

PxTransform localPose;
localPose.p = PxVec3(-(terrainWidth * 0.5f), // make it so tha
    minHeight, -(terrainWidth * 0.5f)); // heightfield is
localPose.q = PxQuat(PxIdentity);
PxShape* shape = PxRigidActorExt::createExclusiveShape(*actor, hf
shape->setLocalPose(localPose);

```

Reducing memory usage

The following strategies can be used to reduce PhysX's memory usage.

Consider using tight bounds for convex meshes

See the above chapter about Performance Issues for details. Using convex meshes is mainly useful for performance, but it can also reduce pairs coming out of the broad-phase, which decreases the amount of memory to manage these pairs.

Use scratch buffers

See the above chapter about Performance Issues for details. Scratch buffers shared between multiple sub-systems (e.g. physics and rendering), can improve memory usage. PhysX will not use less memory per-se, but it can reuse it.

Flush simulation buffers

Call the `PxScene::flushSimulation` function to free internal buffers used for computations. But be aware that these buffers are usually allocated for subsequent frames, so releasing the memory might trigger new re-allocations on the next `simulate` call, which can decrease performance. Please refer to the [memory](#) chapter for details.

Use preallocation

Use `PxSceneDesc::limits` to preallocate various internal arrays. Preallocating the necessary size for internal buffers may use less memory overall than the resizing strategy of dynamic arrays. Please refer to the [Simulation n](#) chapter for details.

Tweak cooking parameters

Some cooking parameters have a direct impact on memory usage. You can use `PxMeshPreprocessingFlag::eDISABLE_ACTIVE_EDGES_PRECOMPUTE`, `PxCookingParams::suppressTriangleMeshRemapTable`, `PxBVH33MidphaseDesc::meshCookingHint`, `PxBVH33MidphaseDesc::meshSizePerformanceTradeOff`, `PxBVH34MidphaseDesc::numTrisPerLeaf`, `PxCookingParams::gaussMapLimit` and `PxCookingParams::buildTriangleMesh` to be modified to choose between runtime performance, cooking performance, and memory usage.

Share shape and mesh data

Share the same `PxConvexMesh` and `PxTriangleMesh` objects between instances if possible. Use shared shapes if possible. Please refer to the [Collision](#) chapter for details about shape sharing.

Use the scene-query and simulation flags

If a shape is only used for scene-queries (raycasts, etc), disable its simulation flag. If a shape is only used for simulation (e.g. it will never be raycasted), disable its scene-query flag. This is good for both memory usage and performance. Please refer to the [Rigid Body Collision](#) chapter for details.

Behavior issues

Objects do not spin realistically

For historical reasons the default maximum angular velocity is set to `PxRigidBodyDynamic::eMAX_ANGULAR_VELOCITY`. This can artificially prevent the objects from spinning quickly, which may be right and wrong in some cases. Please use `PxRigidBodyDynamic::setMaxAngularVelocity` to increase the maximum allowed angular velocity.

Overlapping objects explode

Rigid bodies created in an initially overlapping state may explode, because the solver has to resolve the penetrations in a single time-step, which can lead to large velocities. Use `PxRigidBody::setMaxDe penetrationVelocity` to limit the de-penetration velocity to a reasonable value (e.g. 3.0).

Rigid bodies are jittering on the ground

Visualize the contacts with the visual debugger. If the jittering is caused by contacts appearing and disappearing from one frame to another, try to increase the contact offset (`PxShape::setContactOffset`).

Piles or stacks of objects are not going to sleep

`PxSceneFlag::eENABLE_STABILIZATION` might help here. This is not supported for jointed objects though, so use `PxRigidBody::setStabilization` to enable/disable this feature on a per-object basis. It should be safe to use for objects like debris.

Jointed objects are unstable

There are multiple things to try here:

- Increase the solver iteration counts, in particular the number of iterations. Please refer to the *Rigid Body Dynamics* chapter for details.
- Consider creating the same constraints multiple times. This is similar to having a larger number of solver iterations, but the performance impact is local to the object rather than the simulation island it is a part of. So it can vary overall. Note that the order in which constraints are created is important. If you have 4 constraints named A, B, C, D, and you want to create them multiple times, creating them in the AAAABBBBCCCCDDDD order will not improve performance, while creating them in the ABCDABCDABCDABCD order will.
- Consider using joint projection. This might help for simple cases where objects are connected. Please refer to the *Joints* chapter for details.
- Use smaller time steps. This can be an effective way to improve stability, although it can be an expensive solution. Instead of running 1 simulation time-step dt and N solver iterations, consider trying N simulation time-steps of dt/N and 1 solver iteration.
- Consider tweaking inertia tensors. In particular, for ropes or chains, the `PxJoint::setInvMassScale` and `PxJoint::setInvInertiaScale` functions can be effective. An alternative is to compute the inertia tensor for each link (`PxRigidBodyExt::setMassAndUpdateInertia`) with an artificially increased mass, then set the proper mass directly afterwards (using `PxRigidBody::setMass`).
- Consider adding extra distance constraints. For example in a rope, you can add a distance constraint between the two ends of the rope to prevent stretching. Alternatively, one can create distance constraints between adjacent links $N+2$ in the chain.
- Use spheres instead of capsules. A rope made of spheres will be more stable than a rope made of capsules. The positions of pivots can also affect stability. Placing pivots at the spheres' centers is more stable than placing them on the surfaces.
- Use articulations. Perhaps not surprisingly, articulations are much more stable than articulated objects. They can be used to model better ropes, or

ragdolls out-of-the-box, without the need for the above workaroun
the *Articulations* chapter for details. They are more expensive
though.

GPU Rigid Bodies

Collision detection with `PxSceneFlag::eENABLE_GPU_DYNAMICS` w GPU for all convex-convex, convex-box, box-box, convex-mesh, box and box-HF pairs. However, such pairs will not be processed if either the convex hull exceeds 64 vertices (convex `PxConvexFlag::eGPU_COMPATIBLE` can be used to create compatible requests contact modification, the triangle mesh was not cooked with G (`PxCookingParams::buildGrbData`) or if the triangle mesh makes use of materials.

Aggregates are used to lighten the load on broad phases. When running on the CPU, aggregates frequently improve performance by reducing the cost of the broad phase algorithm. However, there is some cost when using aggregates as these overlaps must be processed by a separate module. When using aggregates on the GPU, the use of aggregates generally result in performance regressions because of aggregate overlaps occurs on the CPU and, while using aggregates on the GPU broad phase, the amount by which they improve performance is frequently smaller than the cost of processing the aggregate overlaps.

Determinism

The PhysX SDK can be described as offering limited determinism. between platforms due to differences in hardware maths precision and the compiler reorders instructions during optimization. This means that different between different platforms, different compilers operating on the between optimized and unoptimized builds using the same compiler on However, on a given platform, given the exact same sequence of even exact scene using a consistent time-stepping scheme, PhysX is ex deterministic results. In order to achieve this determinism, the applica the scene in the exact same order each time and insert the actors in PxScene. There are several other factors that can affect determinism s (e.g. variable) time-stepping scheme is used or if the application do same sequence of API calls on the same frames, the PhysX simulation

In addition, the PhysX simulation can produce divergent behavior if ar simulation has varied. Even the addition of a single actor that is not existing set of actors in the scene can produce divergent results.

PhysX provides a mechanism to overcome the issue of divergent b configurations as a result of additional actors being added or actors b the scene that do not interact with the other actors in the scene. This enabled by raising `PxSceneFlag::eENABLE_ENHANCED_DE PxSceneDesc::flags` prior to creating the scene. Enabling this m performance concessions to be able to offer an improved level of application must still follow all the requirements to achieve dete described previously in order for this mechanism to produce consistent

This guide describes how to upgrade applications that have an integrat using PhysX 3.x. As the changes are numerous and significant, the lev in upgrading to PhysX 3 should be carefully assessed before sta application's integration code.

Removed Features

This section lists features of PhysX 2 that do not have a PhysX 3 equivalent. Features that rely on these features may need fundamental changes, or should be removed from PhysX 2.

Compartments

PhysX 2 scenes supported scene compartments. A separate compartment was assigned to simulating rigid bodies, deformables or fluids. The compartments were simulated in parallel and the scene code contained some extra logic to handle interaction between compartments. Compartments were added as an afterthought. PhysX 2 was not originally designed to support interaction between different simulation technologies. This design deficiency was addressed from the ground up in PhysX 3. In PhysX 3, compartments were no longer needed.

One missing detail is separate time steps are no longer directly supported. A workaround is to create multiple PxScenes and step them at different time steps. In a future scenario the force exchange implementation would be entirely up to the user. A possible approach is to simulate the entire scene using the minimum time step required for any of the compartments.

Deformables

PhysX 2 supported a wide range of deformable mesh simulation including environmental cloth, soft bodies, inflatable balloons and plastic deformation. For performance and code quality reasons, 3.3 temporarily stopped supporting 2.8 deformable features in favor of a much simpler and higher performance simulation engine. In PhysX 3 dot releases, we will be incrementally adding back features such as environmental simulation. For the time being there is no support for applications of PhysX 2 deformables.

NxUtilLib

The assorted utility functions that were in this library was either removed or deleted. Sweep, overlap and ray tests are available in `PxGeometryQuery`. Diagonalization is in `PxDiagonalize()`. Density computation from mass and point unit manipulation routines are gone. Geometrical helpers are in `ge`

Anisotropic Friction

Friction on a surface in PhysX 2 could be configured to be stronger in one direction than another. This is no longer supported in PhysX 3, and there is no known workaround that will give comparable behavior.

Basics

SDK Header

In PhysX 2, the symbols of the SDK could be included in the user's re through the following header:

```
#include "NxPhysics.h"
```

In PhysX 3, this should be replaced with:

```
#include "PxPhysicsAPI.h"
```

SDK Redistribution

Unlike versions of PhysX prior to 2.8.4, PhysX 3 no longer needs a installation on Windows.

API Conventions

The Nx prefix of API classes has changed to a Px prefix. Descriptors were removed and replaced with creation parameters inline in the creat

For example, a capsule was created with PhysX 2 like this:

```
NxCapsuleShapeDesc capsuleDesc;  
capsuleDesc.height = height;  
capsuleDesc.radius = radius;  
capsuleDesc.materialIndex= myMaterial->getMaterialIndex();  
NxShape* aCapsuleShape = aCapsuleActor->createShape(capsuleDesc);
```

In PhysX 3 it is created more succinctly like this:

```
PxShape* aCapsuleShape = PxRigidActorExt::createExclusiveShape(*a  
    PxCapsuleGeometry(radius, halfHeight), myMaterial);
```

Callback Classes

PhysX 2 callback classes are listed below, followed by the corresponding PhysX 3 class. There is one:

NxUserAllocator	PxAllocatorCallback
NxUserOutputStream	PxErrorCallback
NxUserContactReport	PxSimulationEventCallback
NxUserNotify	PxSimulationEventCallback
NxUserTriggerReport	PxSimulationEventCallback

The following PhysX 2 callback classes have no PhysX 3 direct equivalent:

NxUserRaycastReport	Ray casting Results. Results are now passed to PxHitBuffer object.
NxUserEntityReport	Sweep and Overlap results. Results are now passed to PxHitBuffer object.
NxStream	Data serialization. Serialized data is now written to binary buffers.

Below is a list of new callback classes that offer functionality that did not exist in PhysX 2:

PxBroadPhaseCallback	Broad-phase related events.
PxSimulationFilterCallback	Contact filtering.
PxUserControllerHitReport	Reports character controller events.
PxControllerBehaviorCallback	Customizes behavior of character controller.
PxContactModifyCallback	Modification of contact constraints.
PxCCDContactModifyCallback	Modification of CCD contact constraint.
PxConstraintConnector	Custom constraints.
PxProcessPxBaseCallback	Serialization.
PxQueryFilterCallback	Scene query filtering.
PxSpatialLocationCallback	Scene Queries against PxSpatialIndex.
PxSpatialOverlapCallback	Scene Queries against PxSpatialIndex.

Memory Management

NxUserAllocator is renamed to PxAllocatorCallback. An important change is that the SDK now requires that the memory that is returned be 16-byte aligned. On Windows platforms malloc() returns memory that is 16-byte aligned, but on other platforms the function _aligned_malloc() provides this capability.

Debug Rendering

Debug visualization formerly provided by NxScene::getDebugRenderable() is now provided by PxScene::getRenderBuffer() and related functions.

Error Reporting

NxUserOutputStream is now called PxErrorCallback, but works the same way. A separate reportAssertViolation() function. Asserts are only contained in the source code which only ships with the source release and go directly to platform hooks.

Type Casting

PhysX 2 style downcasting:

```
NxSphereShape * sphere = shape->isSphere();
```

is replaced by the following template syntax:

```
const PxRigidDynamic* myActor = actor->is<PxRigidDynamic>();
```

Multithreading

Compared to PhysX 2, there are now more situations where it is legal to run multiple threads. See the section on Multithreading for details.

While PhysX 2 simulation threads were managed internally by the SDK,

could simply specify the number to use, PhysX 3 allows the application the simulation's thread scheduling. It is also possible for the application tasks and submit them to the SDK's default scheduler. See TaskManagement for details.

Startup and Shutdown

PxCreatePhysicsSDK() has been renamed PxCreatePhysics(), and the slightly changed. A foundation instance must first be created PxCreateFoundation().

Extensions

A lot of non-essential utility code has been moved to the extensions like NxActor::addForceAtPos() is now exposed as PxRigidBodyExt::add former function appears to be missing, look there. It is available PxInitExtensions().

Heightfields

Heightfields now need to be pre-cooked like convexes and meshes. P can be set to use the same internal collision logic as meshes so behavior.

Cooking

The PhysX 2 cooking library was created by calling:

```
NxCookingInterface *gCooking = NxGetCookingLib(NX_PHYSICS_SDK_VERSION);  
gCooking->NxInitCooking();
```

It can now be accessed through a single PxCreateCooking() call. Cooking are slightly changed, e.g. NxCookTriangleMesh() is now cooking.cookTriangleMesh().

Serialization

PhysX 3 has two serialization systems: 'RepX' based on XML, and a fast binary data. Neither approach is similar to PhysX 2's save-to-desc ; based serialization code, though the PhysX 3 'RepX' serialization is s NxUStream.

API Design Changes

Changed Actor Hierarchy

PhysX 2 only had a single actor class, and it was possible to call a instance of this class even if it wasn't applicable to the kind of actor obj example, `isSleeping()` could be called on static actors which did not ha In PhysX 3, we decoupled actor into a hierarchy of specialized sub-cla `PxCloth` and `PxParticleSystem` are now subclasses of `PxActor`.

Actor Creation

In PhysX 2, the objects inside each scene were created by the scene cl 3, objects are created by `PxPhysics`, and need to be added to a sc subsequent step by calling:

```
mScene->addActor(actor);
```

Material Indexes

PhysX 2 uses so-called material indexes for stored materials. Material indexes are only supported in PhysX 3 to specify per-triangle materials in meshes. In other cases the material object is referenced directly.

Continuous Collision Detection

PhysX 2 uses CCD skeleton meshes for CCD. PhysX 3 no longer needs skeleton related code can simply be removed.

Pose Description

In PhysX 2 pose is specified using a matrix. In PhysX 3, pose is PxTransform type that consists of a PxVec3 for translation and a P Constructors are provided to convert 4x4 matrices to PxTransform matrices from quaternions, as well as conversely.

Shape Description

PhysX 2 has multiple subclasses of `NxShape`, one for each type corresponding `NxShapeDesc` classes. PhysX 3 has only a single `PxShape` and a `PxGeometry` object is passed on creation. To determine the geometry type, call `PxShape::getGeometryType()`. To extract a `PxGeometry` object of an unknown type, use `PxShape::getGeometry()`.

Skin Width

PhysX 2's `NX_SKIN_WIDTH` and `NxShapeDesc::skinWidth` were replaced by `PxShape::setContactOffset()` and `setRestOffset()`. See [Tuning Shape Collision](#).

Joins

The D6 driveType in PhysX 2 no longer exists in PhysX 3. Now drive is spring-like: if you want position drive you set the 'spring' value non-zero, and if you set both spring and damping field non-zero, you get a velocity drive. Some specialized joints like NxJointDriveDesc, NxJointLimitsDesc (and their names) now were moved to Extensions (see the extensions folder inside the PhysX directory).

If you have used the deleted NxSpringAndDamperEffector, you should now use NxSpringAndDamper with a spring property.

All special axes for a joint (rotation axis for revolute, translation axis for D6) now use the x-axis.

Joint limits now require a contact offset, which determines the distance from the joint to the contact point at which it becomes active. It functions similarly to the contactOffset parameter in the contact detection.

Time Stepping

PhysX 2 had two different time stepping modes: `NX_TIMESTEP_FIXED` (user specified steps into fixed steps) and `NX_TIMESTEP_VARIABLE` (user specified steps to the `setTiming()` function. This controlled SDK-internal substepping and the proper size of the next time step, and called an internal `simulate` function after the elapsed time.

PhysX 3 discards with the substepping code altogether, and exposes `simulate` function directly:

```
mScene->simulate(mStepSize);
```

In PhysX 2 it was legal to call `simulate` with a timestep of zero to force various side-effects of simulation. PhysX 3 neither requires nor supports

The `fetchResults` function stayed the same, however there is no more flush simulation to fetch, as there is now only a single simulation.

Simulation Parameters

The global speeds below which objects go to sleep (`NX_DEFAULT_SLEEP_LIN_VEL_SQUARED` and `NX_DEFAULT_SLEEP_ANG_VEL_SQUARED`) are gone. PhysX 3 introduces a new function `PxRigidDynamic::setSleepThreshold()` which is an energy more similar to the PhysX 2 `NX_DEFAULT_SLEEP_ENERGY`.

The global `NX_BOUNCE_THRESHOLD` is now `PxSceneDesc::bounceThresholdVelocity`.

The `NX_DYN_FRICT_SCALING`, `NX_STATIC_FRICT_SCALING` scaling factors are removed. These values should now be pre-baked into friction coefficients.

The `NX_MAX_ANGULAR_VELOCITY` value has been removed.

NX_ADAPTIVE_FORCE has been renamed PxScenFlag.ADAPTIVE_F

Collision Filtering

PhysX 2 supported multiple fixed function mechanisms for filtering collisions such as collision groups. In PhysX 2 multiple group tags specified as collidable with each other and assigned to shapes.

PhysX 3, supports user callbacks for collision filtering with a restriction that GPU memory cannot be accessed by filtering code so that it can be executed on GPUs with optimal performance. If performance is not a priority, this can be achieved via conventional callbacks (PxSimulationFilterCallback).

When migrating PhysX 2 code, note that we provide PxDefaultSimulationFilterShader in PhysX 3, which emulates a portion of the old behavior. Start by checking if this class is sufficient. As this is an example source code is available and may be extended or customized.

To migrate your fixed function PhysX 2 filtering code on your own, you replicate its exact behavior and implement it as a callback or shader. Let us look at the old mechanisms and make some recommendations for porting:

```
virtual void NxScene::setShapePairFlags(NxShape& shapeA,
    NxShape& shapeB,
    NxU32  nxContactPairFlag    //0 or NX_IGNORE_PAIR
)

virtual void NxScene::setActorPairFlags(NxActor& actorA,
    NxActor& actorB,
    NxU32  nxContactPairFlag
)
```

The first function stored explicit shape pairs in a hash, and a lookup table indicating to filter or not. The second did the same for actor pairs. Because of the size of the pair hash, implementing this mechanism as a shader will be difficult in practice, but implementing as a callback should be trivial using something such as the STL hash_map where Key is a struct holding the two pointers and a bit flag.

Another scheme provided by PhysX 2 were collision groups:

```
virtual void NxShape::setGroup(NxCollisionGroup collisionGroup)
virtual void NxScene::setGroupCollisionFlag(NxCollisionGroup group1,
      NxCollisionGroup group2,
      bool enable
)
)
```

This approach let the user assign shapes to one of 32 collision groups pair of groups be assigned a boolean pair flag. This approach lend shader based implementation. To do this, you should reserve a word filterData (say word0) to hold the group index, and assign this as bef matrix to hold the group pair bits, and a function to set it:

```
NxU32 groupCollisionFlags[32];

//init all group pairs to true:
for (unsigned i = 0; i < 32; i++)
    groupCollisionFlags[i] = 0xffffffff;

void setU32CollisionFlag(NxU32 groups1, NxU32 groups2, bool enable)
{
    NX_ASSERT(groups1 < 32 && groups2 < 32);
    if (enable)
    {
        //be symmetric:
        groupCollisionFlags[groups1] |= (1 << groups2);
        groupCollisionFlags[groups2] |= (1 << groups1);
    }
    else
    {
        groupCollisionFlags[groups1] &= ~(1 << groups2);
        groupCollisionFlags[groups2] &= ~(1 << groups1);
    }
}
```

Unfortunately it is not possible to change this state after the scene because if the matrix could change during simulation, it would force an existing contact pairs to be refiltered. In a large simulation, this could be a large amount of computation. Therefore the matrix must be initialized to its final state.

scene is created, like this:

```
PxSceneDesc desc;
...
desc.filterShaderData = groupCollisionFlags;
desc.filterShaderDataSize = 32 * sizeof(PxU32);
scene = sdk.createScene(desc);
```

Finally, you need to code the filter shader to access this data:

```
PxFilterFlags FilterShader(
    PxFilterObjectAttributes attributes0, PxFilterData filterData
    PxFilterObjectAttributes attributes1, PxFilterData filterData
    PxPairFlags& pairFlags, const void* constantBlock, PxU32 cons
{
    // let triggers through, and do any other prefiltering you ne
    if(PxFilterObjectIsTrigger(attributes0) || PxFilterObjectIsTr
    {
        pairFlags = PxPairFlag::eTRIGGER_DEFAULT;
        return PxFilterFlag::eDEFAULT;
    }
    // generate contacts for all that were not filtered above
    pairFlags = PxPairFlag::eCONTACT_DEFAULT;

    PxU32 ShapeGroup0 = filterData0.word0 & 31;
    PxU32 ShapeGroup1 = filterData1.word0 & 31;
    PxU32* groupCollisionFlags = (PxU32*)constantBlock;

    if ((groupCollisionFlags[ShapeGroup0] & (1 << ShapeGroup1)) =
        return PxFilterFlag::eSUPPRESS;
    else
        return PxFilterFlag::eDEFAULT;
}
```

Scene Queries

The API for scene query functions that return multiple intersections (`PxScene::raycast(...)`) has changed. In PhysX 3, `raycast/overlap/sweep` take a pre-allocated buffer or a callback class as a parameter in order to collect multiple intersections. If you do not know the maximum number of intersections you can inherit from `PxHitCallback` and override `processTouches` virtual function to handle an arbitrary number of intersections via multiple callbacks using only a single callback. Please refer to the Scene Query section of the guide for more details and examples.

Raycasts

The interface for making raycasts was changed in PhysX 3. Now you provide an origin (`PxVec3`) and a direction (`PxVec3`) instead of a `NxRay` that combined position and direction. Please refer to PhysX 2.

Overlaps

Routines like `overlapSphereShapes`, `overlapAABBShapes`, `overlapCapsuleShapes` are now all covered with `PxScene::overlap`. You can use `PxSphereGeometry`, `PxBoxGeometry` or `PxCapsuleGeometry` as a first argument.

Sweep Tests

PhysX 2 provides a `linearCapsuleSweep` that takes two points to define the capsule's spherical ends. In PhysX 3 we have a general `sweep()` routine that takes a capsule and an initial `PxTransform` position. Capsules were defined in PhysX 2 as `PxCapsuleGeometry` and should be converted to an initial transformation (`PxTransform`) that contains position and `PxQuat` for rotation. `PxCapsuleGeometry`'s length is along the capsule's local space.

-
- This guide highlights all significant parts of the API that have changed in the latest release. An application with a working integration of the older version should be able to easily migrate to the newer version by following these pointers.

Math Classes

The static `createIdentity()` and `createZero()` methods are now deprecated in a future release. The preferred method is to use `PxMat33(PxIdentity)`, `PxMat44(PxIdentity)`, `PxQuat(PxIdentity)`, `PxTrs` for identity transforms, and `PxMat33(PxZero)` and `PxMat44(PxZero)` for

Scene Query API

- The Scene Query API underwent significant changes. The highlights are:
 - Former raycastAny, raycastMultiple, raycastSingle API calls were consolidated into a single PxScene::raycast call
 - Same for overlaps and sweeps
 - Same for PxBatchQuery and PxVolumeCache
 - For PxScene queries a deprecated backwards compatibility mode was added to aid the transition
 - This mapping will be removed in the next development cycle
 - There are now dedicated callback and buffer classes for handling query results, replacing PxRaycastHit array and count parameters:
 - Same for sweeps and overlaps
 - See PxRaycastHitBuffer, PXSweepBuffer, PxRaycastHitCallback, PXSweepCallback, PxOverlapCallback
 - The way results are returned is now more robust and transparently handle unbounded number of results allocations.
 - Header PxSceneQueryFiltering.h was renamed to PxSceneQueryReport.h to PxQueryReport.h
 - PxHitFlag::eIMPACT changed to PxHitFlag::ePOSITION
 - PxRaycastHit.impact renamed to PxRaycastHit.position (PxSweepHit.impact)
 - PxQueryFlag::eNO_BLOCK and PxQueryFlag::eANY_HIT

- The following classes were renamed
 - PxSceneQueryHit -> PxQueryHit
 - PxSceneQueryFlags -> PxHitFlags
 - PxSceneQueryHitType -> PxQueryHitType
 - PxSceneQueryFilterData -> PxQueryFilterData
 - PxSceneQueryFilterCallback -> PxQueryFilterCallb
 - PxSceneQueryFilterFlags -> PxQueryFlags
 - PxSceneQueryCache -> PxQueryCache
 - PxCCTNonWalkableMode -> PxControllerNonWalk
 - PxControllerFlags -> PxControllerCollisionFlags
 - PxCCTHit -> PxControllerHit
 - PxConstraintDominance -> PxDominanceGroupPa
 - PxActorTypeSelectionFlags -> PxActorTypeFlags
 - PxFindOverlapTriangleMeshUtil -> PxMeshOverlap
 - Old versions are #defined to new versions to simpli
#defines are deprecated and will be phased out.

- queryClient parameter was removed from raycast/sweep,
list and added to PxQueryFilterData
 - The fix is to simply pass the sa
PxQueryFilterData::clientId

- PxBatchQueryDesc now requires 3 parameters at constru
maxRaycastsPerExecute, PxU32 maxSweepsPerE
maxOverlapsPerExecute
 - Each of these numbers is an upper bound c
PxBatchQuery::raycast(), sweep() and overlap() ca
execute()
 - Previously there was no way to check for results
batch query code since sizes of these buffers were

- The fix is to specify the batch query result (different sizes at construction).
- PxBatchQueryDesc no longer directly holds pointers to memory. In 3.3 these are moved to PxBatchQueryMemory.
 - It is now possible to set a new batch query memory each execute
 - userRaycastHitBuffer has been renamed to userRaycastHitBuffer
 - raycastHitBufferSize has been renamed to raycastHitBufferSize
 - same for overlaps and sweeps (userOverlapHitBuffer, overlapHitBufferSize, userSweepHitBuffer, sweepHitBufferSize)
 - A code snippet below illustrates the migration for the userRaycastHitBuffer
- PxQueryFilterData constructors are now explicit. This means it was possible to write
 - scene->raycast(..., PxQueryFlag::eDYNAMIC, PxQueryFlag::eSTATIC, ...), causing PxQueryFilterData to be implicitly constructed by the compiler
 - now it is required to explicitly construct PxQueryFilterData: scene->raycast(..., PxQueryFilterData(PxQueryFlag::eDYNAMIC, PxQueryFlag::eSTATIC), ...)
 - This change was made to improve type safety and readability while reading the code employing implicit construction
- PxRaycastBufferN, PxOverlapBufferN and PxSweepBufferN for convenience
 - A buffer object with space for 10 touching hits can now be conveniently declared as PxRaycastBufferN
- PxRaycastHit and PxSweepHit now inherit from PxLocation (PxSceneQueryImpactHit)

- `bool PxLocationHit::hadInitialOverlap()` function was added. A swept shape was overlapping at sweep distance=0 or if a hit occurred at distance=0.
 - Functionality of `PxSceneQueryFlag::eINITIAL_CONTACT` and `PxSceneQueryFlag::eINITIAL_OVERLAP_KEEP` was replaced by `PxHitFlag::eASSUME_NO_INITIAL_OVERLAP` and `PxLocationHit::hadInitialOverlap()`.
 - Overlap scene queries with `preFilter` or `postFilter` returning hits would previously return multiple results as touching hits
 - `eBLOCK` should not be returned from user filters for touching hits so will result in undefined behavior, and a warning will be issued.
 - If the `PxQueryFlag::eNO_BLOCK` flag is set, the `eBLOCK` will be automatically converted to an `eTOUCH` and the warning will be suppressed.
 - Sweeps in 3.3 execute using a new faster code path, in exchange for reduced precision. If you encounter precision issues experienced in earlier versions of PhysX, use `ePRECISE` to enable the backwards compatible more accurate sweep code path.
- Snippets demonstrating API migration:

Former `raycastSingle` call:

```
PxRaycastHit hit;
bool hadHit = scene->raycastSingle(..., hit, ...);
if (hadHit) doStuff(hit);
```

Is now:

```
PxRaycastBuffer buf;
bool hadHit = scene->raycast(..., buf, ...);
if (hadHit) doStuff(buf.block);
```

Former raycastAny call:

```
PxSceneQueryHit hit;
bool hadHit = scene->raycastAny(hit);
if (hadHit) doStuff(hit);
```

Is now:

```
PxRaycastBuffer buf; // declare a hit buffer with room for a sing
PxFilterData fdAny; fdAny.flags |= PxQueryFlag::eANY_HIT;
bool hadHit = scene->raycast(buf, PxHitFlags(), fdAny);
if (hadHit) doStuff(buf.block);
```

Former Multiple call:

```
PxRaycastHit buffer[N];
bool hasBlock;
PxI32 result = Scene->raycastMultiple(buffer, N, hasBlock);
if (result == -1)
    handleOverflow();
else
{
    if (hasBlock)
    {
        doBlocking(buffer[result-1]);
        doTouches(buffer, result-1);
    }
    else
    {
        doTouches(buffer, result);
    }
}
```

Is now:

```
PxRaycastBufferN<N> buf;
scene->raycast(buf);
if (buf.hasBlock)
    doBlocking(buf.block);
doTouches(buf.touches, buf.nbTouches);
```

or:

```

for (PxU32 i = 0; i < buf.getNbAnyHits(); i++) // "any" in this c
                                                // touching hits
    doAnyHit(buf.getAnyHit(i));

```

Former batch query memory setup code in 3.2:

```

const PxU32 maxRaycastHits = 16, maxRaycastQueries = 8;
PxRaycastQueryResult* resultBuffer = new PxRaycastQueryResult[max
PxRaycastHitBuffer* hitBuffer = new PxRaycastHit[maxRaycastHits];
PxBatchQueryDesc desc; // required no arguments, there was no saf
                        // of queries per batch (not hits per quer
desc.userRaycastResultBuffer = resultBuffer;
desc.userRaycastHitBuffer = hitBuffer;
desc.raycastHitBufferSize = maxRaycastHits;
PxBatchQuery* bq = PxCreateBatchQuery(desc);
for (PxU32 iQuery = 0; iQuery < maxRaycastQueries; iQuery++)
    bq->raycastSingle(...); // up to 8 raycast queries are allowe
                            // call but there was no overflow che
bq->execute();

for (PxU32 iResult = 0; iResult < nQueries; iResult++)
{
    for (PxU32 iHit = 0; iHit < resultBuffer[i].nbHits; iHit++)
    {
        bool isBlocking = (iHit == resultBuffer[i].nbHits &&
            (resultBuffer[iResult].hits[iHit].flags & PxSceneQuer
            processHit(resultBuffer[iResult].hits[iHit], isBlocking);
    }
}

```

Batch query setup code in 3.3:

```

const PxU32 maxRaycastHits = 16, maxRaycastQueries = 8;
PxBatchQueryDesc desc(maxQueries, 0, 0); // note the new required
                                          // (this is different fr
PxBatchQuery* bq = scene->createBatchQuery(desc);

PxRaycastQueryResult* resultBuffer = new PxRaycastQueryResult[max
PxRaycastHitBuffer hitBuffer = new PxRaycastHit[maxRaycastHits];
PxBatchQueryMemory mem(maxQueries, 0, 0); // maximum number of qu
                                          // (raycasts, overlaps,
mem.userRaycastResultBuffer = resultBuffer;
mem.userRaycastTouchBuffer = hitBuffer;

```

```
mem.raycastTouchBufferSize = maxHits;

PxBatchQuery* bq = PxCreateBatchQuery(desc);
bq->setUserMemory(mem);

for (PxU32 iQuery = 0; iQuery < maxRaycastQueries; iQuery++)
    bq->raycastSingle(...); // up to 8 raycast queries are allowed
                             // with query count overflow check as
bq->execute();

for (PxU32 iResult = 0; iResult < nQueries; iResult++)
{
    // note that the blocking hit is now reported in resultBuffer
    // resultBuffer[i].touches
    for (PxU32 iHit = 0; iHit < resultBuffer[i].nbTouches; iHit++)
        processTouchingHit(resultBuffer[iResult].touches[iHit]);

    processBlockingHit(resultBuffer[iResult].block);
}
```

SPU batch queries

In 3.2 the number of SPU's to be used per batch query was controlled via `setSceneParamInt` call:

```
PxPS3Config::setSceneParamInt(getScene(), PxPS3ConfigParam::eSPU_
```

In 3.3 `PxBatchQuery` no longer automatically executes on multiple SPU's in a separate PPU thread, this design allows higher flexibility, such as executing multiple SPU and PPU threads simultaneously, better control of parameters, allows the user to fine tune thread load balancing. Here's one possible implementation of batch queries on multiple SPU's in 3.3:

```
struct BQThread : shdfnd::Thread
{
    Ps::Sync mBatchReady;
    Ps::Sync mBatchCompleted;
    PxBatchQuery* mBatch;

    PX_FORCE_INLINE BQThread() { mBatch = NULL; }
    PX_FORCE_INLINE void submitBatch(PxBatchQuery* batch) { mBatch = batch; }

    virtual void execute()
    {
        // execute submitted batches until quit is signalled
        for(;;)
        {
            mBatchReady.wait();
            mBatchReady.reset();

            if (quitIsSignalled())
                break;

            mBatch->execute();

            mBatch = NULL;
            mBatchCompleted.set();
        } // for (;;)

        quit(); // shutdown thread
    }
}
```

```

};

// main thread code:
// pre-create and launch batch execute threads
for (PxU32 iThread = 0; iThread < nThreads; iThread++)
{
    BQThread* t = PX_NEW(BQThread);
    t->start();
    mThreads.pushBack(t);
}

// submit batches
for (PxU32 iThread = 0; iThread < nThreads; iThread++)
{
    // create batches
    PxBatchQuery* threadBatch = createBatch(...);
    threadBatch->setRunOnSpu(true);

    mThreads[iThread]->submitBatch(threadBatch);
    mThreads[iThread]->mBatchReady.set();
}

// execute another batch on PPU in the meantime.
PxBatchQuery* threadBatch = createBatch(...);
threadBatch->setRunOnSpu(false);
threadBatch->execute();

// do other PPU work...

// wait for SPU batches to complete:
for (PxU32 i=0; i<mThreads.size(); ++i)
{
    mThreads[i]->mBatchCompleted.wait();
    mThreads[i]->mBatchCompleted.reset();
    releaseBatch(mThreads[i]->mBatch);
}

// terminate batch threads
for (PxU32 i=0; i<mThreads.size(); ++i)
{
    mThreads[i]->signalQuit();
    mThreads[i]->mBatchReady.set();
    mThreads[i]->waitForQuit();
    PX_DELETE(mThreads[i]);
}

```

Whether the batch is executed on SPU or PPU is determined by `PxBatchQueryDesc::runOnSpu` or `PxBatchQuery::setRunOnSpu(bool)`. If `runOnSpu` is true, the query is executed on SPU:

```
PxBatchQueryDesc desc;  
...  
desc.runOnSpu = true;  
...
```

Core PhysX

- The following methods require that the corresponding objects have been added to the scene. Calling these methods for objects which are not in a scene results in undefined behavior. In the CHECKED build configuration an error message will be sent.
 - `addForce/addTorque/clearForce/clearTorque()` on a `PxRigidBody`
 - `isSleeping/wakeUp/putToSleep()` on a `PxRigidBody`, `PxArticulation`, or `PxCloth`
 - `PxScene::resetFiltering()` and the deprecated counterparts `PxScene::resetFilteringForRigidBody()` and `PxScene::resetFilteringForParticleBase()` on `PxScene` and `PxParticleBase`
- The sleep behavior of dynamic rigid bodies has changed significantly. The following changes are:
 - The `wakeUp()` method of `PxRigidBody` and `PxArticulation` now takes a `wakeCounter` parameter. Use the newly introduced `setWakeCounter()` instead to set a specific value.
 - Putting a dynamic rigid actor to sleep will clear any pending wakeups.
 - Switching a dynamic actor to kinematic will put the actor to sleep immediately.
 - Switching a kinematic actor back to dynamic will not affect its sleep state (previously the actor was woken up).
 - Calling `wakeUp/putToSleep()` on a kinematically controlled actor is not valid any longer. The sleep state of a kinematic actor is now determined based on whether a target pose has been set (see API documentation for `isSleeping()` for details).
 - A call to `PxRigidBody::setCMassLocalPose()` does not wake up the actor anymore. Add a call to `PxRigidBody::wakeUp()` to get the actor awake.

Note: this also affects related methods in PhysX `PxRigidBodyExt::updateMassAndInertia()` etc.

- Adding or removing a `PxConstraint` to/from the scene does not wake up connected actors up automatically anymore (note: this applies to PhysX Extensions as well).
 - If a non-zero velocity or force is set using `PxRigidBody::setLinearVelocity()`, `::setAngularVelocity()`, `::addTorque()`, the actor will get woken up automatically even if the `wakeUp` parameter is false.
 - `PxRigidBody::clearForce()` and `::clearTorque()` do not have a `wakeUp` parameter, to optionally wake the actor up, anymore. These methods no longer change the sleep state any longer. Call `::wakeUp()` subsequently to restore old default behavior.
- Shapes may now be shared between actors. This change has several implications:
 - `PxShape::getActor()` now returns a pointer rather than a `PxActor`. If a shape is shareable, the pointer is non-NULL.
 - The following methods of `PxShape` have been removed: `raycast()`, `sweep()`, `overlap()`, `getWorldBounds()`. Replacements are found in `PxShapeExt`.
 - `PxShape` now has the same reference counting semantics as `PxMaterial`, so that `release()` releases the user reference, and when a reference is released, the shape is destroyed.
 - Shapes created through `PxRigidActor::createShape()` are automatically destroyed when the actor is released. However, after deserializing such a shape, the regular reference counting semantics apply.
 - `raycast()` and `sweep()` return results from scene queries which previously specifically specify an actor also.
 - Shape local transforms cannot be specified on shape creation anymore.

the local transform after creation with `PxShape::setLocalPose()`.

- The `PxObserver/PxObservable` system has been replaced by the API. The supported object types have been extended from `PxActo` inheriting from `PxBase`. Furthermore, two kinds of deletion distinguished: user release and memory release. The following pseudocode for the transition from the previous to the new API:

old API:

```
class MyObserver : public PxObserver
{
public:
    virtual void onRelease(const PxObservable& observable);
}

MyObserver myObs;
PxRigidDynamic* d = create...;
d->registerObserver(myObs);
```

new API:

```
class MyDelListener : public PxDeletionListener
{
public:
    virtual void onRelease(const PxBase* observable, void* userData,
        PxDeletionEventFlag::Enum deletionEvent);
}

MyDelListener myDelListener;
PxPhysics* physics = create...;
PxRigidDynamic* d = create...;
physics->registerDeletionListener(myDelListener, PxDeletionEventF
PxBase* b = d;
physics->registerDeletionListenerObjects(myDelListener, &b, 1);
```

- The `contactStream` in `PxContactPair` is now stored in a variable `contactStream`. This is used to save memory. As such, you can **no** it to a `PxContactPoint*` and access the data. Instead, you `PxContactPair::extractContacts` or use a `PxContactStreamIterator` to

Please see the callbacks section of the user guide for further information.

- The friction API and behavior for dynamic rigid bodies has changed
 - Friction mode flags `eENABLE_ONE_DIRECTIONAL_FRICTION` and `eENABLE_TWO_DIRECTIONAL_FRICTION` have been replaced by `PxFrictionType::Enum` and `PxSceneDesc::frictionType`.
 - `PxSceneDesc::contactCorrelationDistance` has been deprecated. It no longer has an influence on how many friction anchors are created in a frame, only on when they are removed in later frames. This is a very minor change in friction behavior.
- `PxShape::resetFiltering()` and `PxParticleBase::resetFiltering()` have been deprecated. Please use one of the new overloaded methods `PxSceneDesc::resetFiltering()` and `PxParticleBase::resetFiltering()`.
- `PxClientBehaviorBit` and `PxActorClientBehaviorBit` have been replaced by `PxClientBehaviorFlag` and `PxActorClientBehaviorFlag` respectively.
- `PxActorTypeSelectionFlag` and `PxActorTypeSelectionFlags` have been replaced by `PxActorTypeFlag` and `PxActorTypeFlags` respectively.
- `PxConstraintDominance` has been renamed to `PxDominanceGroup`.
- The parameter 'spring' on articulation joints has been renamed 'stiffness'.
- The parameter 'tangentialSpring' on articulation joints has been renamed 'tangentialStiffness'.
- `PxConstraintFlag::Type` has been renamed to `PxConstraintFlag::Error`.
- Discrete contact reports are no longer produced for pairs with the flag `PxPairFlag::eDETECT_DISCRETE_CONTACT` raised in the filter. Discrete contact generation would always have been performed in the presence of the flag `PxPairFlag::eDETECT_DISCRETE_CONTACT`.

potentially improves performance when using specific shapes for which would have previously generated discrete contacts and the the solver.

- Trigger reports are no longer produced for PxFPairFlag::eDETECT_DISCRETE_CONTACT raised in the PxFPairFlag::eTRIGGER_DEFAULT has been modified in PxFPairFlag::eDETECT_DISCRETE_CONTACT flag.

PhysX Extensions

- Joint limits have been more carefully separated into `PxJointAngularLimitPair`, `PxJointLinearLimitPair`.
- `PxJoint::getType()` is deprecated. Joints now inherit from `PxJoint`. `getConcreteType()` replaces `getType()`. Alternatively, to dynamically check for a particular joint type, use e.g. `joint->is<PxD6Joint>()` which will return `true` if the type matches, otherwise `NULL`.
- The parameter 'spring' in joint limits and drives has been renamed 'springiness'.
- Dominance settings no longer apply to joints. To achieve dominance, use `setInvMassScale`. For example if actor0 in the joint is to affect actor1, use `setInvMassScale0(0.0f)`, `setInverseInertiaScale0(0.0f)` on actor0.

PhysX Character Controller

- When creating a PxControllerManager, a reference to a PxScene is required. As a consequence, creating a controller from a PxControllerManager requires the controller descriptor as an argument.
- On PxControllerManager::release(), all associated PxObstacleController objects get deleted automatically. Make sure to not access PxObstacleController objects after the corresponding manager has been released.

PhysX Vehicles

- A new struct has been introduced to hold the enumerated list `PxVehicleDrive4W::eFRONT_LEFT_WHEEL`. The changes are
 - `PxVehicleDrive4W::eFRONT_LEFT_WHEEL`
`PxVehicleDrive4WWheelOrder::eFRONT_LEFT`
 - `PxVehicleDrive4W::eFRONT_RIGHT_WHEEL`
`PxVehicleDrive4WWheelOrder::eFRONT_RIGHT`
 - `PxVehicleDrive4W::eREAR_LEFT_WHEEL`
`PxVehicleDrive4WWheelOrder::eREAR_LEFT`
 - `PxVehicleDrive4W::eREAR_RIGHT_WHEEL`
`PxVehicleDrive4WWheelOrder::eREAR_RIGHT`
- A new struct has been introduced to hold the enumerated list `PxVehicleDrive4WControl::eANALOG_INPUT_ACCEL`. The changes are
 - `PxVehicleDrive4W::eANALOG_INPUT_ACCEL`
`PxVehicleDrive4WControl::eANALOG_INPUT_ACCEL`
 - `PxVehicleDrive4W::eANALOG_INPUT_BRAKE`
`PxVehicleDrive4WControl::eANALOG_INPUT_BRAKE`
 - `PxVehicleDrive4W::eANALOG_INPUT_HANDBRAKE`
`PxVehicleDrive4WControl::eANALOG_INPUT_HANDBRAKE`
 - `PxVehicleDrive4W::eANALOG_INPUT_STEER_LEFT`
`PxVehicleDrive4WControl::eANALOG_INPUT_STEER_LEFT`
 - `PxVehicleDrive4W::eANALOG_INPUT_STEER_RIGHT`
`PxVehicleDrive4WControl::eANALOG_INPUT_STEER_RIGHT`
 - `PxVehicleDrive4W::eMAX_NUM_DRIVE4W_ANALOG_INPUTS`
`PxVehicleDrive4WControl::eMAX_NB_DRIVE4W_ANALOG_INPUTS`

- A new struct has been introduced to hold the enumerated list PxVehicleDrive4W::eFRONT_LEFT_WHEEL. The changes are
 - PxVehicleDriveTank::eTANK_WHEEL_FRONT_LEFT
PxVehicleDriveTankWheelOrder::eFRONT_LEFT
 - PxVehicleDriveTank::eTANK_WHEEL_FRONT_RIGHT
PxVehicleDriveTankWheelOrder::eFRONT_RIGHT,
 - PxVehicleDriveTank::eTANK_WHEEL_1ST_FROM_FRONT
PxVehicleDriveTankWheelOrder::e1ST_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_1ST_FROM_FRONT
PxVehicleDriveTankWheelOrder::e1ST_FROM_FRONT_RI
 - PxVehicleDriveTank::eTANK_WHEEL_2ND_FROM_FRONT
PxVehicleDriveTankWheelOrder::e2ND_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_2ND_FROM_FRONT
PxVehicleDriveTankWheelOrder::e2ND_FROM_FRONT_RI
 - PxVehicleDriveTank::eTANK_WHEEL_3RD_FROM_FRONT
PxVehicleDriveTankWheelOrder::e3RD_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_3RD_FROM_FRONT
PxVehicleDriveTankWheelOrder:: e3RD_FROM_FRONT_RI
 - PxVehicleDriveTank::eTANK_WHEEL_4TH_FROM_FRONT
PxVehicleDriveTankWheelOrder::e4TH_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_4TH_FROM_FRONT
PxVehicleDriveTankWheelOrder::e4TH_FROM_FRONT_RI
 - PxVehicleDriveTank::eTANK_WHEEL_5TH_FROM_FRONT
PxVehicleDriveTankWheelOrder::e5TH_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_5TH_FROM_FRONT
PxVehicleDriveTankWheelOrder::e5TH_FROM_FRONT_RI
 - PxVehicleDriveTank::eTANK_WHEEL_6TH_FROM_FRONT
PxVehicleDriveTankWheelOrder::e6TH_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_6TH_FROM_FRONT

- PxVehicleDriveTankWheelOrder::e6TH_FROM_FRONT_RIGHT
 - PxVehicleDriveTank::eTANK_WHEEL_7TH_FROM_FRONT
 - PxVehicleDriveTankWheelOrder::e7TH_FROM_FRONT_LEFT
 - PxVehicleDriveTank::eTANK_WHEEL_7TH_FROM_FRONT
 - PxVehicleDriveTankWheelOrder::e7TH_FROM_FRONT_RIGHT
 - PxVehicleDriveTank::eTANK_WHEEL_8TH_FROM_FRONT
 - PxVehicleDriveTankWheelOrder::e8TH_FROM_FRONT_LEFT
 - PxVehicleDriveTank::eTANK_WHEEL_8TH_FROM_FRONT
 - PxVehicleDriveTankWheelOrder::e8TH_FROM_FRONT_RIGHT
 - PxVehicleDriveTank::eTANK_WHEEL_9TH_FROM_FRONT
 - PxVehicleDriveTankWheelOrder::e9TH_FROM_FRONT_LEFT
 - PxVehicleDriveTank::eTANK_WHEEL_9TH_FROM_FRONT
 - PxVehicleDriveTankWheelOrder::e9TH_FROM_FRONT_RIGHT
 - A new struct has been introduced to hold the enumerated list PxVehicleDriveTank::eANALOG_INPUT_ACCEL. The changes are
 - PxVehicleDriveTank::eANALOG_INPUT_ACCEL
 - PxVehicleDriveTankControl::eANALOG_INPUT_ACCEL
 - PxVehicleDriveTank::eANALOG_INPUT_BRAKE_LEFT
 - PxVehicleDriveTankControl::eANALOG_INPUT_BRAKE_LEFT
 - PxVehicleDriveTank::eANALOG_INPUT_BRAKE_RIGHT
 - PxVehicleDriveTankControl::eANALOG_INPUT_BRAKE_RIGHT
 - PxVehicleDriveTank::eANALOG_INPUT_THRUST_LEFT
 - PxVehicleDriveTankControl::eANALOG_INPUT_THRUST_LEFT
 - PxVehicleDriveTank::eANALOG_INPUT_THRUST_RIGHT
 - PxVehicleDriveTankControl::eANALOG_INPUT_THRUST_RIGHT
 - PxVehicleDriveTank::eMAX_NUM_DRIVETANK_ANALOG_

PxVehicleDriveTankControl::eMAX_NB_DRIVETANK_ANAL

- A new struct has been introduced to hold the enumerated list PxVehicleDriveTank::eDRIVE_MODEL_STANDARD. The changes are
 - PxVehicleDriveTank::eDRIVE_MODEL_STANDARD
PxVehicleDriveTankControlModel::eSTANDARD
 - PxVehicleDriveTank::eDRIVE_MODEL_SPECIAL
PxVehicleDriveTankControlModel::eSPECIAL
- A new struct has been introduced to hold the enumerated list eVEHICLE_TYPE_DRIVE4W. The changes are
 - eVEHICLE_TYPE_DRIVE4W -> PxVehicleTypes::eDRIVE4W
 - eVEHICLE_TYPE_DRIVETANK -> PxVehicleTypes::eDRIVETANK
 - eVEHICLE_TYPE_NODRIVE -> PxVehicleTypes::eNODRIVE
 - eMAX_NUM_VEHICLE_TYPES
PxVehicleTypes::eMAX_NB_VEHICLE_TYPES
- A new struct has been introduced to hold the enumerated list PxVehicleGraph::eCHANNEL_JOUNCE. The changes are
 - PxVehicleGraph::eCHANNEL_JOUNCE
PxVehicleWheelGraphChannel::eJOUNCE
 - PxVehicleGraph::eCHANNEL_SUSPFORCE
PxVehicleWheelGraphChannel::eSUSPFORCE
 - PxVehicleGraph::eCHANNEL_TIRELOAD
PxVehicleWheelGraphChannel::eTIRELOAD
 - PxVehicleGraph::eCHANNEL_NORMALIZED_TIRELOAD
PxVehicleWheelGraphChannel::eNORMALIZED_TIRELOAD
 - PxVehicleGraph::eCHANNEL_WHEEL_OMEGA
PxVehicleWheelGraphChannel::eWHEEL_OMEGA

- PxVehicleGraph::eCHANNEL_TIRE_FRICTION
PxVehicleWheelGraphChannel::eTIRE_FRICTION
 - PxVehicleGraph::eCHANNEL_TIRE_LONG_SLIP
PxVehicleWheelGraphChannel::eTIRE_LONG_SLIP
 - PxVehicleGraph::eCHANNEL_NORM_TIRE_LONG_FORCE
PxVehicleWheelGraphChannel::eNORM_TIRE_LONG_FOF
 - PxVehicleGraph::eCHANNEL_TIRE_LAT_SLIP
PxVehicleWheelGraphChannel::eTIRE_LAT_SLIP
 - PxVehicleGraph::eCHANNEL_NORM_TIRE_LAT_FORCE
PxVehicleWheelGraphChannel::eNORM_TIRE_LAT_FORC
 - PxVehicleGraph::eCHANNEL_NORM_TIRE_ALIGNING_MC
PxVehicleWheelGraphChannel::eNORM_TIRE_ALIGNING_
 - PxVehicleGraph::eMAX_NUM_WHEEL_CHANNELS
PxVehicleWheelGraphChannel::eMAX_NB_WHEEL_CHAN
- A new struct has been introduced to hold the enumerated list PxVehicleGraph::eCHANNEL_ENGINE_REVS. The changes are
 - PxVehicleGraph::eCHANNEL_ENGINE_REVS
PxVehicleDriveGraphChannel::eENGINE_REVS
 - PxVehicleGraph::eCHANNEL_ENGINE_DRIVE_TORQUE
PxVehicleDriveGraphChannel::eENGINE_DRIVE_TORQUE
 - PxVehicleGraph::eCHANNEL_CLUTCH_SLIP
PxVehicleDriveGraphChannel::eCLUTCH_SLIP
 - PxVehicleGraph::eCHANNEL_ACCEL_CONTROL
PxVehicleDriveGraphChannel::eACCEL_CONTROL
 - PxVehicleGraph::eCHANNEL_BRAKE_CONTROL
PxVehicleDriveGraphChannel::eBRAKE_CONTROL
 - PxVehicleGraph::eCHANNEL_HANDBRAKE_CONTROL
PxVehicleDriveGraphChannel::eHANDBRAKE_CONTROL

- PxVehicleGraph::eCHANNEL_STEER_LEFT_CONTROL
PxVehicleDriveGraphChannel::eSTEER_LEFT_CONTROL
 - PxVehicleGraph::eCHANNEL_STEER_RIGHT_CONTROL
PxVehicleDriveGraphChannel::eSTEER_RIGHT_CONTROL
 - PxVehicleGraph::eCHANNEL_GEAR_RATIO
PxVehicleDriveGraphChannel::eGEAR_RATIO
 - PxVehicleGraph::eMAX_NUM_ENGINE_CHANNELS
PxVehicleDriveGraphChannel::eMAX_NB_DRIVE_CHANNELS
- A new struct has been introduced to hold the enumerated list PxVehicleGraph::eGRAPH_TYPE_WHEEL. The changes are
 - PxVehicleGraph::eGRAPH_TYPE_WHEEL
PxVehicleGraphType::eWHEEL
 - PxVehicleGraph::eGRAPH_TYPE_ENGINE
PxVehicleGraphType::eDRIVE
- Non-persistent data is no longer stored in the vehicle. Instead of each vehicle it is stored in an array and passed to PxVehicleUpdate function argument. A simple example of how to construct, use, and array is given below. This example code updates an array of vehicles and the air. If the vehicles are not in the air then the actor under each vehicle is stored in an array:

```

void updateVehicles(const PxF32 timestep, const PxVec3& gravi
const PxVehicleDrivableSurfaceToTireFrictionPairs& fricPa
PxVehicleWheels** vehicles, PxU32 numVehicles, std::vecto
{
    //Count the total number of wheels.
    unsigned int numWheels = 0;
    for(unsigned int i = 0; i < numVehicles; i++)
    {
        numWheels += vehicles[i]->mWheelsSimData.getNbWheels(
    }

    //Allocate buffers to store results for each vehicle and

```

```

PxVehicleWheelQueryResult* vehicleWheelQueryResults =
    new PxVehicleWheelQueryResult[numVehicles];
PxWheelQueryResult* wheelQueryResults = new PxWheelQueryR
PxU32 wheelCount = 0;
for(PxU32 i = 0; i < numVehicles; i++)
{
    vehicleWheelQueryResults[i].nbWheelQueryResults =
        vehicles[i]->mWheelsSimData.getNbWheels();
    vehicleWheelQueryResults[i].wheelQueryResults = &wheel
    wheelCount += vehicles[i]->mWheelsSimData.getNbWheel
}

//Update the array of vehicles.
PxVehicleUpdates(timestep, gravity, fricPairs, numVehicle
    vehicleWheelQueryResults);

//Test if each vehicle is in the air.
for(PxU32 i = 0; i < numVehicles; i++)
{
    if(!PxVehicleIsInAir(vehicleWheelQueryResults[i]))
    {
        for(PxU32 j = 0; j < vehicleWheelQueryResults[i].
        {
            if(vehicleWheelQueryResults[i].wheelQueryResu
            {
                hitActors.push_back
                    (vehicleWheelQueryResults[i].wheelQue
            }
        }
    }
}

delete[] vehicleWheelQueryResults;
delete[] wheelQueryResults;
}

```

- The following accessors to non-persistent data associated with each vehicle were replaced as follows
 - PxVehicleWheelsDynData::getSuspLineStart
PxWheelQueryResult::suspLineStart
 - PxVehicleWheelsDynData::getSuspLineDir
PxWheelQueryResult::suspLineDir

- PxVehicleWheels::getSuspRaycast -> PxWheelQueryResu
PxWheelQueryResult::suspLineDir, PxWheelQueryResult::s
 - PxVehicleWheelsDynData::getTireDrivableSurfaceShape
PxWheelQueryResult::tireContactShape
 - PxVehicleWheelsDynData::getTireDrivableSurfaceMaterial
PxWheelQueryResult::tireSurfaceMaterial
 - PxVehicleWheelsDynData::getTireDrivableSurfaceType
PxWheelQueryResult::tireSurfaceType
 - PxVehicleWheelsDynData::getTireDrivableSurfaceContactP
PxWheelQueryResult::tireContactPoint
 - PxVehicleWheelsDynData::getTireDrivableSurfaceContactN
PxWheelQueryResult::tireContactNormal
 - PxVehicleWheelsDynData::getTireFriction
PxWheelQueryResult::tireFriction
 - PxVehicleWheelsDynData::getSuspJounce
PxWheelQueryResult::suspJounce
 - PxVehicleWheelsDynData::getSuspensionForce
PxWheelQueryResult::suspSpringForce
 - PxVehicleWheelsDynData::getTireLongitudinalDir
PxWheelQueryResult::tireLongitudinalDir
 - PxVehicleWheelsDynData::getTireLateralDir
PxWheelQueryResult::tireLateralDir
 - PxVehicleWheelsDynData::getTireLongSlip
PxWheelQueryResult::longitudinalSlip
 - PxVehicleWheelsDynData::getTireLatSlip ->PxWheelQueryF
 - PxVehicleWheelsDynData::getSteer -> PxWheelQueryResu
 - PxVehicleWheels::isInAir -> PxWheelQueryResult::isInAir
- PxVehicleWheels::setWheelShapeMapping
PxVehicleWheels::getWheelShapeMapping have been

PxVehicleWheelsSimData::setWheelShapeMapping
PxVehicleWheelsSimData::getWheelShapeMapping

- PxVehicleWheels::setSceneQueryFilterData
PxVehicleWheels::getSceneQueryFilterData have been
PxVehicleWheelsSimData::setSceneQueryFilterData
PxVehicleWheelsSimData::getSceneQueryFilterData
- PxVehicle4WEnable3WTadpoleMode and PxVehicle4WEnable3
take an extra function argument: a non-const r
PxVehicleWheelsDynData
- PxVehicleWheels::isInAir() has been replaced with Px\
PxVehicleWheelQueryResult& vehWheelQueryResults)
- PxVehicleDrive4WSmoothAnalogRawInputsAndSetAnalogInputs n
function argument "const bool isVehicleInAir". This can be ca
function PxVehicleIsInAir
- To improve api consistency PxVehicleTelemetryData::getNumWf
PxVehicleTelemetryData::getNbWheelGraphs
- To improve api consistency PX_MAX_NUM_WHEI
PX_MAX_NB_WHEELS
- To improve api consistency PxVehicleGraph::eMAX_NUM_TITL
PxVehicleGraph::eMAX_NB_TITLE_CHARS
- PxVehicleTireData::mCamberStiffness has been
PxVehicleTireData::mCamberStiffnessPerUnitGravity.
PxVehicleTireData::mCamberStiffnessPerUnitGravity should be
equivalent to the old value of PxVehicleTireData::mCamberStiffn
magnitude of gravitational acceleration (PxScene::getGravity()).
advantage of using PxVehicleTireData::mCamberStiffnessPerUn
independent of length scale.

- PxVehicleComputeTireForceDefault has been removed from the Custom tire shaders that call PxVehicleComputeTireForce implemented by taking a copy of PxVehicleComputeTireForceDefault copy instead.

CCD

- The mechanism to activate CCD per shape has changed. `PxShapeFlag::eUSE_SWEPT_BOUNDS` that was used in 3.2 to activate CCD per shape has been removed. In its place is `PxRigidBodyFlag::eENABLE_CCD` that is set per rigid actor. Setting `eENABLE_CCD` on a rigid actor in 3.3 has approximately the same effect as setting `PxShapeFlag::eUSE_SWEPT_BOUNDS` on all the actor's shapes in 3.2.
- `PxPairFlag::eSWEPT_INTEGRATION_LINEAR` has been replaced by `PxPairFlag::eCCD_LINEAR` in PhysX 3.3.
- `PxSceneFlag::eENABLE_SWEPT_INTEGRATION` flag in 3.2 has been replaced by `PxSceneFlag::eENABLE_CCD` in PhysX 3.3.
- A simple example of how to enable CCD on a specific shape is shown here. It demonstrates creating a body consisting of a large box and a small sphere. The box is only used in discrete collision detection and the sphere is used for CCD. The simulation filter shader shown here requires that the shapes be flagged with `eCCD_RESPONSE` to generate a collision response (`PxPairFlag::eCCD_LINEAR`). Likewise, the filter shader shown here requires that the filter data of both shapes need to be flagged with `eDISCRETE_RESPONSE` in order to generate a collision response (`PxPairFlag::eRESOLVE_CONTACT`). A final remark is that the following shader requires that shapes of static actors be flagged with flags `eDISCRETE_RESPONSE | eCCD_RESPONSE` in order to generate collision response from pairs that involve a static actor and a dynamic actor:

```
struct CCDFilterTest
{
    enum FilterFlags
    {
        eDISCRETE_RESPONSE    = 1 << 0
        eCCD_RESPONSE         = 1 << 1
    };
};
```

```

static PxFilterFlags filterShader(
    PxFilterObjectAttributes attributes0,
    PxFilterData filterData0,
    PxFilterObjectAttributes attributes1,
    PxFilterData filterData1,
    PxPairFlags& pairFlags,
    const void* constantBlock,
    PxU32 constantBlockSize)
{
    pairFlags = PxPairFlags(0);

    PxU32 combo = filterData0.word0 & filterData1.word0;
    if(combo & eDISCRETE_RESPONSE)
    {
        pairFlags |= PxPairFlag::eRESOLVE_CONTACTS;
    }
    if(combo & eCCD_RESPONSE)
    {
        pairFlags |= PxPairFlag::eCCD_LINEAR;
    }
    return PxFilterFlags();
}
};

```

....

```

PxRigidDynamic* dyn = getPhysics().createRigidDynamic(PxTrans
PxBoxGeometry box;
box.halfExtents = PxVec3(1.f, 1.f, 1.f);
PxSphereGeometry sphere;
sphere.radius = 0.75f;
PxShape* boxShape = dyn->createShape(box, getDefaultMaterial(
PxShape* sphereShape = dyn->createShape(sphere, getDefaultMat

```

```

PxFilterData data = boxShape->getSimulationFilterData();
data.word0 |= CCDFilterTest::eDISCRETE_RESPONSE;
boxShape->setSimulationFilterData(data);

```

```

data = sphereShape->getSimulationFilterData();
data.word0 |= CCDFilterTest::eCCD_RESPONSE;
sphereShape->setSimulationFilterData(data);

```

```

dyn->setRigidBodyFlag(PxRigidBodyFlag::eENABLE_CCD, true);
getActiveScene().addActor(*dyn);

```

PhysX Visual Debugger

- A new flag has been introduced to configure the visualizing of constraints
`PxVisualDebuggerFlag::eTRANSMIT_CONSTRAINTS;`
- A new function has been introduced to configure PxVisualDebugger
`PxVisualDebugger::setVisualDebuggerFlags(PxVisualDebugger`
- A new function has been introduced to send error stream to PVD:
`PxVisualDebugger::sendErrorMessage((PxErrorCode::Enum c
message, const char* file, PxU32 line);`
- The following functions were renamed:
`PxVisualDebugger::getPvdConnectionFactory()
PxVisualDebugger::getPvdConnection(); PxVisualDebugger::ge
> PxVisualDebugger::getPvdDataStream();`
- The PVD connect function changed to the same method as previous
`PxVisualDebuggerExt::connect -> PxVisualDebuggerExt::create`
- The constraint, contacts and scene queries visualizing can all
PxVisualDebuggerFlag in 3.3. Here is an example for how to enable
the contacts :

```
mPhysics->getVisualDebugger()-  
>setVisualDebuggerFlags(PxVisualDebuggerFlag::eTRANSMIT_  
PxVisualDebuggerFlag::eTRANSMIT_CONSTRAINTS);
```

PhysX Cloth

There have been substantial changes to the PhysX 3.3 cloth solver performance and behavior. This has resulted in a reorganization of how data is stored and processed in the cloth fabric. Prior to PhysX 3.3 the cloth solver organized edge constraints into independent groups. In PhysX 3.3 it is now possible to decompose constraints into fibers, instead edge constraints now exist and are solved in larger, independent sets. Interface changes are detailed below.

- Previously there were multiple solver types to choose from for edge constraints such as eFAST, eSTIFF, eBENDING, PxClothPhaseSolverConfig::SolverType). There is now one type of edge constraints, this is a flexible distance constraint with controls to address certain ranges of compression and stretch (see PxClothStretch). Bending and stretching such as bending are now achieved by the way distance constraints are defined geometrically, rather than through a specialized bending solver.
- To reduce stretching a new constraint type has been added called 'Tether'. These constraints do not act along edges of the mesh, but rather act on attachments between particles that enforce a maximum distance between them. See PxClothFabric::getTetherAnchors().
- Cloth cooking which was previously part of the PxCooking library is now in the extension library, see PxClothFabricCooker:

```
// PhysX 3.2.x
cooking->cookClothFabric(meshDesc, gravity, outputStream);

// PhysX 3.3
PxClothFabricCooker cooker(meshDesc, gravity, useGeodesicTether);
cooker.save(outputStream, false);
```

- The PxClothCollisionData parameter has been removed from PxPhysics. The collision shapes can now be added after cloth creation using PxCloth::addCollisionSphere and PxCloth::addCollisionCapsule.

- `PxCloth::wakeUp()` does not have a parameter anymore. Use the method `setWakeCounter()` instead to set a specific value.
- `PxCloth::setDampingCoefficient` now takes a `PxVec3` instead of a the damping per axis.
- `PxCloth::setPhaseSolverConfig()` has been renamed to `PxCloth::se`
- `PxCloth::lockClothReadData()` has been renamed to `PxCloth::lockF`
- `PxClothFabricTypes.h` has been removed, this header has b
`PxClothFabric.h`

RepX Serialization

Substantial changes were made to the PhysX 3.3 serialization into collections and references between collections have been unified for serialization.

- The RepX and RepXUpgrader libraries have been removed. RepX is now provided through PhysXExtensions.
- RepXCollection has been replaced with PxCollection, which is the class for both RepX and binary serialization in 3.3. Collections are now serialized and deserialized with PxSerialization::createCollectionFromXml(). Entities can now be created with PxCreateCollection(). Serialization into RepX format is achieved through PxSerialization::serializeCollectionToXml().
- TRepXId has been replaced with PxSerialObjectId.
- RepXIdToRepXObjectMap and RepXObject have been replaced with repX functionality in PxCollection, which now maps between serializable objects and PxSerialObjectId values.
- RepXExtension was removed. Serialization and deserialization of scenes is now achieved through the PxRepXSerializer interface.
- RepXUtility and PxToolkit functionality has been replaced with various methods in PxSerialization, PxCollection and PxScene.
 - A PxCollection with all PxPhysics-level objects such as spheres, planes, materials (formally referred to as buffers) can be created with PxCollectionExt::createCollection(PxPhysics&).
 - Similarly PxCollectionExt::createCollection(PxScene&) can be used to create a collection of PxScene-level objects.
 - Dependencies between objects and collections can be managed with PxCollectionExt::addReference(PxSerialObjectId, PxSerialObjectId).

PxSerialization::complete().

- The objects of a collection can be added to PxScene::addCollection().
- Operations on files are generally handled with a PxOutputStream and PxInputData. Default implementation: PxDefaultFileOutputStream and PxDefaultFileInputData.
- RepXUpgrader::upgradeCollection was removed. RepX data can be upgraded to newer PhysX versions by deserializing and re-serializing with PxSerialization::createCollectionFromXml(), PxSerialization::serializeCollectionToXml().
- Serialization functionality requires a PxSerializationRegistry instance created with PxSerialization::createSerializationRegistry().
- XML serialization can be configured to store the cooked triangle data along with the plain data for faster loading.
- PhysXVehicles supports RepX serialization. PxSerializationRegistry is provided to PxInitVehicleSDK() for vehicle serialization, PxCloseVehicleSDK() for cleanup.
- Custom class RepX serialization is supported in 3.3, more information in [Serialization](#).

Binary Serialization

The binary serialization interface has been refactored and unified into a single serialization interface.

- Most serialization functionality requires an instance of `PxSerializationRegistry`. It is application managed and can be created with `PxSerialization::createSerializationRegistry()` and released with `PxSerializationRegistry::release()`.
- The base class for serializable types has been renamed from `PxBase`. Most of the serialization functionality moved to a separate `PxSerializer` instance per serializable type is registered with `PxSerializationRegistry`. All `PhysX` and `PhysXExtension` serializable types are supported by default.
- `PxCollection` has been reworked.
 - `PxCollection::serialize()` and `PxCollection::deserialize()` were replaced with `PxSerialization::createCollectionFromBinary()` and `PxSerialization::serializeCollectionToBinary()` in `PhysXExtension`.
 - `PxSerializable::collectForExport()` has been replaced with `PxCollection::collectForExport()`. `PxSerialization::complete()` helps to add required object dependencies. `PxSerializable::isSerializable()` should be used to check if a collection can be successfully serialized.
- `PxUserReferences` was removed: `PxCollection` instances can now resolve dependencies between collections on their own. `PxSerialization::complete()` supports creating collections with external dependencies to other collections.
- `PxSerialObjectRef` has been replaced with `PxSerialObjectId`.
- `PxCollectForExportSDK()` and `PxCollectForExportScene()` functions were removed.

with `PxCollectionExt::createCollection(PxPhysics& physics)` and `PxCollectionExt::createCollection(PxScene& scene)`.

- `PxDumpMetaData()` was replaced with `PxSerialization::dumpBinary`
- The `PxBinaryConverter` moved from `PhysXCooking` to `PxCooking::createBinaryConverter()` was replaced by `PxSerialization::createBinaryConverter()`.
- `PhysXVehicles` supports binary serialization. `PxSerializationReg` provided to `PxInitVehicleSDK()` for vehicle serialization, `PxClose` cleanup.
- Custom class binary serialization is supported in 3.3, more information in [Serialization](#).

PhysX TaskManager

- The pxtask namespace has been removed and all it's types are r physx namespace with a Px prefix, for example pxtask::LightCpuTask physx::PxLightCpuTask

[NVIDIA PhysX SDK 3.4.1 Documentation »](#)
[User's Guide »](#)

[previous](#) | [next](#)

-
- This guide highlights all significant parts of the API that have changed in this release. An application with a working integration of the older version should be able to easily migrate to the newer version by following these pointers.
 - Functionality shared with the APEX SDK was moved into a separate directory outside of the "PhysX" directory. Since the PxFoundation and PxShared libraries are versioned separately, PxCreatFoundation() now takes PX_FOUNDATION_VERSION as an argument.

Deprecated APIs

PxRigidActor::createShape

PxRigidActor::createShape() is deprecated, and will be removed. PxRigidActorExt::createExclusiveShape() replaces this method.

PxSceneFlag::eDEPRECATED_TRIGGER_TRIGGER_REPC

PxSceneFlag::eDEPRECATED_TRIGGER_TRIGGER_REPORTS is deprecated and will be removed in PhysX 3.5. More details are mentioned under [Core PhysX](#)

PhysX particles

The PhysX particle feature has been deprecated in PhysX version 3.4. The library PhysX FleX is an alternative with a richer feature set.

PhysX cloth

The PhysX clothing feature has been deprecated in PhysX version 3.4. APEX clothing features are replaced by the standalone NvCloth library.

Core PhysX

- `PxCreatePhysics` now requires a `PxFoundation` object to be passed. It now receives a pointer to a `PxPvd` object, used for connecting PhysX debugger.
- `PxActor::isRigidStatic`, `isRigidDynamic`, `isParticleSystem`, `isArticulationLink`, `isCloth`, `isRigidActor`, `isRigidBody`, `isParticle` removed. Use corresponding `PxBase::is()` with class template pointer casting.
- `PxContactPairFlag::eINTERNAL_HAS_FACE_INDICES` is obsolete and removed.
- Trigger shapes will no longer send notification events for intersection shapes. For PhysX 3.4 there is the option to re-enable the `PxSceneFlag::eDEPRECATED_TRIGGER_TRIGGER_REPORTS` no longer be available in PhysX 3.5. It is recommended to use `eDEPRECATED_TRIGGER_TRIGGER_REPORTS` and instead use a non-trigger shape, both with the same geometry and local pose, to receive notifications for overlaps between trigger shapes.
- Implementations of `PxSimulationEventCallback` will have to provide an implementation of the newly added method `onAdvance()` to avoid crashes.
- The deprecated method `PxPhysics::createHeightField(const PxHeightFieldDesc&, PxPhysicsInsertionCallback&)` has been removed. Please use `PxCooking::createHeightField(const PxHeightFieldDesc&, PxPhysicsInsertionCallback&)` instead. The `PxCooking` object can be obtained through `PxPhysics::getPhysicsInsertionCallback()`.
- The deprecated flag `PxActorTypeSelectionFlag/PxActorTypeSelectionFlags` removed. Please use `PxActorTypeFlag/PxActorTypeFlags` instead.

- The deprecated class `PxFindOverlapTriangleMeshUtil` has been replaced by `PxMeshOverlapUtil` instead.
- The deprecated flag `PxConstraintFlag::eREPORTING` has been removed. Reports are now always generated.
- The deprecated flag `PxConstraintFlag::eDEPRECATED_32_CC` has been removed.
- `PxRegisterHeightFields()` now registers unified heightfields. To register legacy heightfields, call `PxRegisterLegacyHeightFields()`. Legacy heightfields are deprecated and will be removed in a future PhysX release.
- The following deprecated simulation event flags have been removed:
 - `PxContactPairHeaderFlag::eDELETED_ACTOR_0, ::eDELETED_ACTOR_1`
(use `PxContactPairHeaderFlag::eREMOVED_ACTOR_0, ::eREMOVED_ACTOR_1` instead)
 - `PxContactPairFlag::eDELETED_SHAPE_0, ::eDELETED_SHAPE_1`
(use `PxContactPairFlag::eREMOVED_SHAPE_0, ::eREMOVED_SHAPE_1` instead)
 - `PxTriggerPairFlag::eDELETED_SHAPE_TRIGGER, ::eDELETED_SHAPE_OTHER`
(use `PxTriggerPairFlag::eREMOVED_SHAPE_TRIGGER, ::eREMOVED_SHAPE_OTHER` instead)
- `PxContactPair` now reports separate pointers for `contactPatches` and `contactImpulses` rather than reporting a single `pc` pointer. The interface for `PxContactStreamIterator` has been modified accordingly. See the `PxContactPair::extractContacts` method for details.

further guidance on how to iterate over this contact data if required.

Contact Generation

- PCM contact generation is now used by default. Legacy S generation can be re-enabled by clearing the `PxSceneFlag::eE PxSceneDesc::flags`.
- Unified heightfields are now the default heightfield collision approach mirrors the way in which mesh contact gen functions so per heightfields to be used interchangeably with negligible behavior legacy heightfield collision approach can be used `PxRegisterLegacyHeightFields()`.
- When unified heightfields are in use, the bounds of heightfield extruded by "thickness". If legacy heightfield collision is used, the extruded by thickness.

PhysX Cooking

- The `PxMeshPreprocessingFlag::eREMOVE_UNREFERENCED_TRIANGLES` and `PxMeshPreprocessingFlag::eREMOVE_DUPLICATED_TRIANGLES` have been removed. Meshes are now cooked up by default unless `PxMeshPreprocessingFlag::eDISABLE_CLEANUP` is set.
- `PxCookingParams::meshSizePerformanceTradeOff` and `PxCookingParams::meshCookingHint` have been moved to `PxBVH33` since they only affect the BVH33.
- The `PxGaussMapLimit.h` file has been removed. The `PxGetGaussMapVertexLimitForPlatform` function has been moved to `PxBVH33` but the function is now deprecated, along with the `PxPlatform` enum. Instead, use now an explicit `PxCookingParams::gaussMapLimit` parameter. As of PhysX 3.4 is concerned there is nothing to do other than remove `PxGaussMapLimit.h`, and perhaps including `PxCooking.h` instead if needed.
- Legacy `PxConvexMeshCookingType::eINFLATION_INCREMENTAL_HULL` is deprecated. (The `PxConvexMeshCookingType::eQUICKHULL` algorithm must be used in all cases. To cook a convex mesh without inflation, the `PxConvexMeshCookingType::eQUICKHULL` algorithm must be used as `PxConvexMeshCookingType::eINFLATION_INCREMENTAL_HULL` does not support inflation.)

Reference Counting

- In previous releases, `isReleasable()` for shareable objects (shape, convex meshes, cloth fabrics, materials and heightfields) would `release()` had been called on the object, which was only allowed if reference counts can be manually incremented with `acquire()` and decremented with `release()`, and so the fact that `release()` has been called is a reliable indicator of whether it can be called again.
- As a consequence of the above, applications must ensure they have a counted reference to each shareable object in a collection before calling `PxCollectionExt::releaseObjects`. The main case in which this might be the case is when using `PxRigidActor::createShape()`, since in that case or when using `releaseExclusiveShapes` to `PxCollectionExt::releaseObjects` may be the case.
- Since there is no unique user release for shareable objects, the `USER_RELEASE` events when `release()` is called.

PhysX Visual Debugger

- PxVisualDebugger is deprecated, and new PxDpvd has been introduced. These are mentioned in *PhysX Visual Debugger (PVD)*.

Scene queries

- PxPruningStructure enum has been renamed to PxPruningStructur
- Deprecated type PxSceneQueryHit has been removed. Please use PxSceneQueryHitData instead.
- Deprecated type PxSceneQueryFilterData has been removed. Please use PxQueryFilterData instead.
- Deprecated type PxSceneQueryFilterCallback has been removed. Please use PxQueryFilterCallback instead.
- Deprecated type PxSceneQueryCache has been removed. Please use PxQueryCache instead.
- Deprecated types PxSceneQueryFlag(s) has been removed. Please use PxSceneQueryFlags instead.
- Deprecated scene query functions have been removed (e.g. PxScene::raycastAny() etc). To make the transition easier they are still available in PxScene part of PhysXExtensions. A previous PxScene::raycastAny(...) call can use PxSceneQueryExt::raycastAny(PxScene, ...), or PxScene::raycastAny(PxScene, ..., PxSceneQueryExt) instead.
- PxHitFlag::eFACE_INDEX was introduced. In order to receive sweep results against convex geometry, the flag needs to be set.
- PxHitFlag::eDISTANCE has been deprecated, since the distance and its computation cannot be skipped. Please simply avoid using it. The flag has no effect and it will be removed in the next version.
- The "anyHit" parameter of the PxGeometryQuery and PxShapeExt::raycast() functions has been removed. Please use PxHitFlag::eMESH_ANY instead.
- PxMeshQuery::sweep() now respects PxHitFlag::eMESH_BOTH_SIDES. If you previously used that flag when calling that function, it was ignored, and in 3.4 might start generating different results compared to 3.3. If keeping the current behaviour is important, please disable PxHitFlag::eMESH_BOTH_SIDES.

PxMeshQuery::sweep() calls.

- Batched scene queries are marked as deprecated and will be removed from the system in future releases.
- Volume cache feature is marked as deprecated, it will be removed in a future release.
- Spatial index feature is marked as deprecated, it will be removed in a future release.

PxExtensions

- The signatures for the PxComputeMeshPe and PxComputeHeightFieldPenetration functions have changed. The old signatures are still available but they are now deprecated. It is recommended to transition to the new functions (with the same names but a different signature).



Built February 12, 2018

Contents:

- Welcome to PhysX
 - About this User Guide
 - Physics vs. PhysX
 - World and Objects
- Snippets
 - What are PhysX Snippets?
 - HelloWorld: PhysX Basics
 - Using PhysX Visual Debugger with SnippetHelloWorld
- Building with PhysX
 - Build Settings
 - Build Configurations
 - Header Files
 - Libraries
 - Redistribution
- The PhysX API
 - Introduction
 - Memory Management
 - Error Reporting
 - Math Classes
 - Connecting PhysX Objects with User Application Objects

- Type Casting
- Reference Counting
- Using Different Units
- Assertions
- Determinism
- Startup and Shutdown
 - Introduction
 - Foundation and Physics
 - Cooking
 - Extensions
 - Optional SDK Components
 - Delay-Loading DLLs
 - PhysXCommon DLL and PsFoundation DLL load
 - PxDelayLoadHook
 - PxGpuLoadHook
 - PhysXCommon Secure Load
 - Shutting Down
- Threading
 - Introduction
 - Data Access from Multiple Threads
 - Scene Locking
 - Locking Semantics
 - Locking Best Practices
 - Asynchronous Simulation
 - Double Buffering
 - Events involving removed objects
 - Support
 - Memory Considerations
 - Multithreaded Simulation
 - TaskManager

- CpuDispatcher
- CpuDispatcher Implementation Guidelines
- BaseTask
- Task
- LightCpuTask
- Geometry
 - Introduction
 - Geometry Types
 - Spheres
 - Capsules
 - Boxes
 - Planes
 - Convex Meshes
 - Convex Mesh cooking
 - Triangle Meshes
 - Height Fields
 - Mesh Scaling
 - PxGeometryHolder
 - Vertex and Face Data
 - Convex Meshes
 - Triangle Meshes
 - Height Fields
 - Heightfield Modification
- Rigid Body Overview
 - Introduction
 - Rigid Body Object Model
 - The Simulation Loop
- Rigid Body Collision
 - Introduction
 - Shapes

- Simulation Shapes and Scene Query Shapes
- Kinematic Triangle Meshes (Planes, Heighfields)
- Broad-phase Algorithms
- Regions of Interest
- Broad-phase Callback
- Collision Filtering
- Aggregates
- Creating an Aggregate
- Populating an Aggregate
- Releasing an Aggregate
- Amortizing Insertion
- Trigger Shapes
- Interactions
- Rigid Body Dynamics
 - Velocity
 - Mass Properties
 - Applying Forces and Torques
 - Gravity
 - Friction and Restitution
 - Sleeping
 - Sleep state change events
 - Kinematic Actors
 - Active Transforms
 - Active Actors
 - Dominance
 - Solver Iterations
 - Immediate Mode
 - Enhanced Determinism
 - Axis locking
- Simulation

- Callback Sequence
- Simulation memory
 - Scene Limits
 - 16K Data Blocks
 - Scratch Buffer
 - In-place Serialization
 - PVD Integration
- Completion Tasks
- Synchronizing with Other Threads
- Substepping
- Split sim
- Split fetchResults
- Advanced Collision Detection
 - Tuning Shape Collision Behavior
 - Contact Modification
 - Contact reporting
 - Contact Reports and CCD
 - Extracting Contact information
 - Extra Contact Data
 - Continuous Collision Detection
 - Contact Notification and Modification
 - Triggers
 - Tuning CCD
 - Performance Implications
 - Limitations
 - Raycast CCD
 - Speculative CCD
 - Persistent Contact Manifold (PCM)
- Joints
 - Joint Basics

- Visualization
 - Force Reporting
 - Breakage
 - Projection
 - Limits
 - Actuation
 - Mass Scaling
- Fixed Joint
- Spherical Joint
- Revolute Joint
- Prismatic Joint
- Distance Joint
- D6 Joint
 - Locking and Unlocking Axes
 - Limits
 - Drives
 - Configuring Joints for Best Behavior
- Custom Constraints
 - Defining Constraint Behavior
 - The Constraint Shader Table
 - Data Management
- Articulations
 - Creating an Articulation
 - Articulation Joints
 - Driving an Articulation
 - Articulation Projection
 - Articulations and Sleeping
- Scene Origin
 - Shifting The Scene Origin
- GPU Rigid Bodies

- Introduction
- Using GPU Rigid Bodies
- What is GPU accelerated?
- Tuning
- Performance Considerations
- Geometry Queries
 - Introduction
 - Raycasts
 - Raycasts against Spheres, Capsules, Boxes and Convex
 - Raycasts against Planes
 - Raycasts against Triangle Meshes
 - Raycasts against Heightfields
 - Overlaps
 - Penetration Depth
 - Sweeps
 - Initial Overlaps
 - Precise Sweeps
 - Sweeps against Height Fields
 - Pitfalls
 - Additional PxGeometryQuery functions
 - PxMeshQuery
 - Mesh Overlaps
 - Sweeps against Triangles
- Spatial Queries
 - PxSpatialIndex
- Scene Queries
 - Introduction
 - Basic queries
 - Raycasts
 - Sweeps

- Overlaps
 - Touching and blocking hits
 - Query modes
 - Closest hit
 - Any hit
 - Multiple hits
 - Multiple hits with blocking hit
 - Filtering
 - PxQueryFlag::eSTATIC, PxQueryFlag::eDYNAMIC
 - PxQueryFlag::ePREFILTER, PxQueryFlag::ePOSTFILTER
 - PxQueryFlag::eANY_HIT
 - PxQueryFlag::eNO_BLOCK
 - PxFilterData fixed function filtering
 - User defined hit callbacks for unbounded results
 - Batched queries
 - Volume Caching
 - Filling the Cache
 - Querying the Cache
 - Single Object Caching
 - PxPruningStructureType
 - Generalities
 - PxPruningStructureType::eNONE
 - PxPruningStructureType::eSTATIC_AABB_TREE
 - PxPruningStructureType::eDYNAMIC_AABB_TREE
 - PxPruningStructure
 - Merge process
- Vehicles
 - Introduction
 - Algorithm
 - First Code

- Vehicle SDK Initialization
- Introduction To Vehicle Creation
- Introduction To Vehicle Update
- Snippets
 - SnippetVehicle4W
 - SnippetVehicleTank
 - SnippetVehicleNoDrive
 - SnippetVehicleScale
 - SnippetVehicleMultiThreading
- Advanced Concepts
 - Vehicle Creation
 - Filtering
 - Tire Friction on Drivable Surfaces
 - Vehicle Controls
 - Vehicle Update
- More Advanced Concepts
 - Vehicle Telemetry
 - Vehicle Update Multi-Threaded
 - Tire Shaders
 - Vehicle Types
 - SI Units
 - Level of Detail
 - Wheel Contact Beyond Raycasts
- Tuning Guide
 - PxVehicleWheelData
 - PxVehicleWheelsSimData
 - PxVehicleSuspensionData
 - PxVehicleAntiRollBar
 - PxVehicleTireData
 - PxVehicleEngineData

- PxVehicleGearsData
- PxVehicleAutoBoxData
- PxVehicleClutchData
- PxVehicleAckermannGeometryData
- PxVehicleTireLoadFilterData
- PxVehicleDifferential4WData
- PxRigidDynamic
- Troubleshooting
 - Jittery Vehicles
 - The Engine Rotation Refuses To Spin Quickly
 - The Engine Spins But the Wheels Refuse To Spin
 - The Wheels Are Spinning But The Vehicle Does Not Move
 - The Vehicle Does Not Steer/Turn
 - The Acceleration Feels Sluggish
 - The Vehicle Does Not Slow Down When Not Accelerating
 - The Vehicle Turns Too Quickly/Too Slowly
 - The Wheels Spin Too Much Under Acceleration
 - The Wheels Spin Too Much When Cornering
 - The Vehicle Never Goes Beyond First Gear
 - The Vehicle Under-steers Then Over-steers
 - The Vehicle Slows Down Unnaturally
 - The Vehicle Climbs Too Steep Slopes
- References
 - Anti-roll Bar
 - Tire Modeling
 - Engine Torque Curves
 - Differentials
 - Clutch
 - Ackermann Steer Correction
 - Tanks

- General
- Character Controllers
 - Introduction
 - Kinematic Character Controller
 - Creating a character controller
 - Overlap Recovery Module
 - Character Volume
 - Volume Update
 - Moving a Character Controller
 - Graphics Update
 - Auto Stepping
 - Climbing Mode
 - Up Vector
 - Walkable Parts & Invisible Walls
 - Obstacle Objects
 - Hit Callback
 - Behavior Callback
 - Character Interactions: CCT-vs-dynamic actors
 - Character Interactions: CCT-vs-CCT
 - Hidden Kinematic Actors
 - Time Stepping
 - Invalidating Internal Geometry Caches
 - Runtime Tessellation
 - Troubleshooting
 - Character goes through walls in rare cases
 - Tessellation performance issue
 - The capsule controller manages to climb over obstacles h offset value
- Particles (deprecated)
 - Introduction

- Creating Particle Systems
- Particle Management
 - Creating Particles
 - Releasing Particles
 - Index Pool Extension
 - Updating Particles
 - Reading Particles
- Parameter Guide
 - Particle Dynamics
 - Collision with Rigid Actors
 - Fluid (PxParticleFluid)
- Collision Handling
 - Collision Filtering
 - Per-particle Rest Offsets
 - Particle Drains
- Best Practices / Troubleshooting
 - Particle Grid and Spatial Data Structure Overflow
- GPU/CUDA Acceleration
- Sample Discussion
- References
- Cloth
 - Introduction
 - Simulation Algorithm
 - Solver Frequency
 - Particle Integration
 - Local Space Simulation and Inertia Scale
 - Constraints
 - Distance Constraints
 - Tether Constraints
 - Motion Constraints

- Separation Constraints
- Collision Detection
- Continuous Collision Detection
- Virtual Particle Collision
- Friction and Mass Scaling
- Self-Collision of a Single Cloth Actor
- Inter-Collision between Multiple Cloth Actors
- Best Practices / Troubleshooting
 - Performance Tips
 - Using GPU Cloth
 - Fast-Moving Characters
 - Avoiding Stretching
 - Avoiding Jitter
 - PVD Support
- Snippet Discussion
 - Filling in PxClothMeshDesc
 - Creating Fabric
 - Creating Cloth
 - Specifying Collision Shapes
 - Adding Virtual Particles
 - Accessing Particle Data
- References
- Debug Visualization
 - Introduction
 - Usage
- PhysX Visual Debugger (PVD)
 - PVD
 - Basic Setup (SDK Side)
 - Network Setup
 - File Setup

- Advanced Setup
 - Connection Flags
 - Visualizing Externals and Extended Data
 - Visualizing SceneQuery
 - Custom PvdClient
 - PVD Error Stream
 - Custom profiling
- Simulation Statistics
 - Interface
 - Usage
- Serialization
 - Introduction
 - First Code
 - In-depth Discussion
 - Collections
 - Serializing Complete Collections
 - Serializing Incomplete Collections
 - Reference Counting of Deserialized Objects
 - Reconnecting PhysX and Game-Objects
 - Serializing Everything
 - Serializability
 - Binary Serialization Specifics
 - API-level Serialization (RepX) Specifics
 - Common Use Cases
 - Snippet Discussion
 - SnippetSerialization
 - SnippetConvert
 - SnippetLoadCollection
 - Best practices / Troubleshooting
 - PVD

- Extending Serialization
 - Introduction
 - Overview
 - Binary Serialization of Custom Classes
 - RepX Serialization of Custom Classes
 - PhysX API Metadata System
- Best Practices Guide
 - Introduction
 - Debugging
 - Use checked builds and the error stream
 - Visualizing physics data
 - Visualizing physics data (2)
 - Limiting coordinates
 - Performance Issues
 - Use profile builds to identify performance bottlenecks
 - Use release builds for final performance tests
 - Disable debug visualization in final/release builds
 - Debug visualization is very slow
 - Consider using tight bounds for convex meshes
 - Use scratch buffers
 - Use the proper mid-phase algorithm
 - Use the proper narrow-phase algorithm
 - Use the proper broad-phase algorithm
 - Use the scene-query and simulation flags
 - Tweak the dynamic tree rebuild rate
 - Use the insertion callback when cooking at runtime
 - The "Well of Despair"
 - Pruner Performance for Streamed Environments
 - Performance Implications for Multi-Threading
 - Memory allocation

- Character Controller Systems using Scene Queries and Computation
- Quantizing HeightField Samples
- Reducing memory usage
 - Consider using tight bounds for convex meshes
 - Use scratch buffers
 - Flush simulation buffers
 - Use preallocation
 - Tweak cooking parameters
 - Share shape and mesh data
 - Use the scene-query and simulation flags
- Behavior issues
 - Objects do not spin realistically
 - Overlapping objects explode
 - Rigid bodies are jittering on the ground
 - Piles or stacks of objects are not going to sleep
 - Jointed objects are unstable
- GPU Rigid Bodies
- Determinism
- Migrating From PhysX SDK 2.x to 3.x
 - Removed Features
 - Compartments
 - Deformables
 - NxUtilLib
 - Anisotropic Friction
 - Basics
 - SDK Header
 - SDK Redistribution
 - API Conventions
 - Callback Classes

- Memory Management
- Debug Rendering
- Error Reporting
- Type Casting
- Multithreading
- Startup and Shutdown
- Extensions
- Heightfields
- Cooking
- Serialization
- API Design Changes
 - Changed Actor Hierarchy
 - Actor Creation
- Material Indexes
- Continuous Collision Detection
- Pose Description
- Shape Description
 - Skin Width
- Joints
- Time Stepping
 - Simulation Parameters
- Collision Filtering
- Scene Queries
 - Raycasts
 - Overlaps
 - Sweep Tests
- Migrating From PhysX SDK 3.2 to 3.3
 - Math Classes
 - Scene Query API
 - SPU batch queries

- Core PhysX
- PhysX Extensions
- PhysX Character Controller
- PhysX Vehicles
- CCD
- PhysX Visual Debugger
- PhysX Cloth
- RepX Serialization
- Binary Serialization
- PhysX TaskManager
- Migrating From PhysX SDK 3.3 to 3.4
 - Deprecated APIs
 - PxRigidActor::createShape
 - PxSceneFlag::eDEPRECATED_TRIGGER_TRIGGER_RI
 - PhysX particles
 - PhysX cloth
 - Core PhysX
 - Contact Generation
 - PhysX Cooking
 - Reference Counting
 - PhysX Visual Debugger
 - Scene queries
 - PxExtensions