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, 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
    - 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 major rewrite of the SDK, we are excited to bring you a great number of enhancements, including numerous API improvements. Because so much has changed in the API, we recommend even experienced PhysX users to read through this guide to familiarize themselves with the new programming interface.
About this User Guide

This Guide will help the reader to understand the PhysX-3 SDK and its applications. The Guide presents an overview of the features and implementation of the PhysX SDK, and its 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 interested reader should refer to the PhysX API Reference Documentation. (See PhysX Documentation directory under the main directory where the PhysX SDK was unpacked.) Users migrating from PhysX-2 will find the *Migrating From 3.x* chapter of particular interest.
Physics vs. PhysX

Physics is a rich and broad scientific pursuit, an attempt to explain with mathematics the behavior of all matter, everything in the entire universe, using concepts such as space, time, energy, inertia, momentum and force. In physics, space is assumed to be infinitely in three dimensions, and can be divided into infinitely small units with arbitrarily fine precision. In other words, positions in physics space are described by vectors of single-precision floating point numbers.

In contrast, positions in PhysX simulation space are vectors of single-precision floating point numbers. Like the dimensions of space, time in physics is described by a real number, duration divisible into arbitrarily small intervals. Physics promises that if the forces imposed on a system are known throughout some period of time, and if the state of the system is known precisely at some instant of time in that period, then the state of 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 simulated system is known only at specific instants in time, usually referred to as 'steps', and the simulation may only step forwards in time, never backwards. The state of a 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 where 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 a given time step, say 1/60 second, will take less than that amount of time on an observer's clock if the performance of the hardware platform is sufficient for the complexity of the simulation. That the PhysX SDK is more widely used in computer and video game applications is both a cause and an effect of these design choices. Consequently this Guide usually refers to PhysX in the context of 'the game world', 'rigid body game objects', 'the character', etc.
The basic concepts of the world within a PhysX simulation are easy to visualize:

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

Games are a very visual medium and audible and games usually place special requirements on their graphics and sound. Production quality graphics are outside the scope of PhysX, but it is enormously valuable to be able to visualize this otherwise hidden world. Some of our example programs come with rudimentary built-in visualization, and we also provide a stand-alone debugging tool called PhysX Visual Debugger (PVD). PVD provides a graphical view of the PhysX scene together with tools to inspect and visualize variables of every PhysX object. Additionally it can also record and visualize memory and timing data. See *PhysX Visual Debugger* details.
What are PhysX Snippets?

In the context of the PhysX SDK, a 'Snippet' is a simple, minimalistic code sample. SDK version 3.3.0 offers a collection of Snippets to illustrate usage of the PhysX API in a concise format, free from the complexity of a sample framework or game engine. 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 file, e.g. `Snippets/compiler/vc14win64/Snippets.sln`

Although a few of the Snippets support rendering, (Win32, Win64, OSX) 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 viewing the code 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 snippets comprise a single source file, but SnippetHelloWorld, among others, supports rendering through a second source file. SnippetHelloWorld creates stacks on a plane, and if rendering is enabled, allows the user to create a ball from the camera position.

The primary code for SnippetHelloWorld is found in `{SDK Root}/Snippets/SnippetHelloWorld/SnippetHelloWorld.cpp`. 
Using PhysX Visual Debugger with SnippetHelloWorld

As is the case with any Snippet built against PROFILE, CHECKED or DEBUG configurations of the PhysX runtime, HelloWorld will automatically connect to the PhysX Visual Debugger if that application is already running when the Snippet executable is launched. For Snippets without rendering, PVD provides an easy way to visualize the contents of the PhysX scene presented in the Snippet. In the screenshot below, PhysX Visual Debugger appears on the right hand side, while Visual Studio and SnippetHello World are on the left.
On Windows, PhysX requires either Visual Studio 2010 or Visual Studio
Build Settings

The PhysX headers should compile cleanly at the highest typical warning levels: /W4 for Visual Studio, -Wall -Wextra -pedantic for gcc- and clang-based compilers. Stricter warning settings may result in a small number of benign informational warnings.

The PhysX source projects and snippets will compile cleanly using the project files or makefiles supplied.
Build Configurations

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

- the **debug** build can be useful for error analysis, but contains assertions used for SDK 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 and other incorrect uses of the API which might otherwise cause runtime 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. It omits most checks and instrumentation.

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

**Note:** We strongly recommend that you use the checked build as the primary 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 PhysXVisualDebuggerSDK) should never be mixed in an application because this will result 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 include and library directories respectively. There is a combined include header available as:

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

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

```c
#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 platforms:

- 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 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 simply be recompiled using different CRT settings.
Redistribution

On the Windows platform, you need to redistribute some of our DLLs to end users as part of 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 cooks geometry data on the fly.
- PhysX3GPU_*.dll - is only needed if your application runs some simulation on the GPU.
- PhysX3CharacterKinematic_*.dll - is only needed if your application uses the 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 depending on whether your application is built in 64 bit mode.
Introduction

This chapter covers the basic patterns common to the PhysX application programming 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 Pvd prefix.

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 required to build the PhysX libraries which are distributed as source, and are also used in the samples. They can be recognized because they do not have the Px prefix. Even though in principle accessible to users, they are largely undocumented and we do not maintain compatibility of this code between PhysX versions. For that reason we recommend strongly against their use in applications.
Memory Management

PhysX performs all allocations via the `PxAllocatorCallback` interface. You must implement this interface in order to initialize PhysX:

```cpp
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 that is 16-byte aligned, and on Windows the system function `_aligned_malloc()` provides this capability. The other parameters to `allocate()` are a string which identifies the type of the allocated memory, and the `__FILE__` and `__LINE__` location inside PhysX code where the allocation was made. 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 memory allocator. If you are instrumenting system memory allocations, you may observe unusual behavior. To prevent PhysX requesting type names, disable allocation names using the method `PxFoundation::setReportAllocationNames()`.

You can place PhysX objects in memory owned by the application using the binary deserialization mechanism. See `Serialization` for details.

As an alternative to instrumenting the allocator, you can obtain detailed memory allocation information in the PhysX Visual Debugger (see: `PhysX Visual Debugger` (PVD)).
**Error Reporting**

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

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

There is only a single function to implement, `reportError`. This function should log the passed message, or print it on the application's output console. For the more serious error codes `eABORT`, `eINVALID_PARAMETER`, `eINVALID_OPERATION`, `eINTERNAL_ERROR` and `eOUT_OF_MEMORY`, breaking into the debugger may be a more appropriate choice. Whatever you do, do not just ignore the messages.

A simple implementation of the error callback class can be found in the PhysX Extensions 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 their respective header files, e.g. (SDKRoot)/Include/foundation/PxVec3.h. The types support standard operator overloads and typical math operations. Zero and identity objects 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 quaternion 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.x + d = 0`.

PxMat33 and PxMat44 matrices represent transformations with basis vectors in the 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 OpenGL and Direct3D. For example, to set the model transformation for a rigid body in OpenGL:

```cpp
// 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 used directly instead of 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 with a PhysX object is to use the `userData` member provided by the most important classes (`PxActor::userData`, `PxShape::userData`, `PxMaterial::userData`, ...). The member is a `void*` pointer which is reserved for application use. Each `userData` field, so to manage multiple associations another mechanism
Type Casting

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

```cpp
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 concrete 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 a scene graph. For example, a PxConvexMesh may be referenced by multiple objects, each sharing the same geometry but associated with different actors. The specific types are PxTriangleMesh, PxHeightField, PxConvexMesh, PxMaterial, PxClothFabric, 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 1, plus the number of counted references that exist to the object.

The initial reference count of 1 ensures the object is not destroyed until the application allows it by calling release() - thereafter it will be destroyed when no 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. After calling the shape's release() method, it has a reference count of 1. When the actor is destroyed, the shape is detached from the actor, the reference count is decremented to 0, and the shape is destroyed.
The `acquireReference()` method increments the reference count of an object. For example, when a spatial query returns a reference to a mesh shape, and you want to pass that result to another thread for deferred processing, incrementing the reference count will ensure that even if the shape referencing the mesh is released, the mesh continues to exist.

**Note:** Subtypes of `PxGeometry` do not have counted references to the meshes they point, e.g. when `PxConvexMeshGeometry` points to a `PxConvexMesh`. A counted 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 such actors (see *Shapes* and *Reference Counting of Deserialized Objects*).
Using Different Units

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

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

```cpp
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, scale);
```

This will result in the defaults for values like PxShape::contactDistance being scaled 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 of PxPhysics, and also when creating PxSceneDesc objects.
Assertions

PhysX uses the PX_DEBUG macro to enable or disable assertions. The macro is not set in the PhysXCore and PhysXCommon libraries, and so by default the trigger assertions, however you may configure the libraries provided a them. When an assert is triggered, PhysX calls an assert handler. By handler will trigger a debug breakpoint. However, you may PxSetAssertHandler() to customize the assert handler.
Determinism

PhysX is deterministic in the sense it will produce identical simulation results from the same sequence of API calls applied from the point where a scene is created (and the same responses from simulation callbacks which modify data). Note that removing all the objects from a scene is not in general sufficient to reinitialize it for this purpose.

PhysX simulation behavior is not sensitive to the number of CPU worker threads used.

An important caveat to determinism is the state of the x87 FPU on 32-bit platforms. Some compilers produce x87 floating point instructions even when configured to prefer SSE instructions, and the results of those operations may depend on the x87 control word. Since it is too expensive to modify the x87 FPU state at every PhysX entry point, this is delegated to the application if necessary. PhysX operations do not result in changes to the x87 control word, but certain other libraries (including DirectX) may modify it.

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

The first step in using the PhysX SDK in a program is the initialization of objects. These objects can be released when PhysX is no longer needed to free resources. This chapter describes how to do this.
First, in some startup code, create a *PxFoundation* object:

```cpp
static PxDefaultErrorCallback gDefaultErrorCallback;
static PxDefaultAllocator gDefaultAllocatorCallback;

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

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

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

Now create the top-level *PxPhysics* object:

```cpp
bool recordMemoryAllocations = true;

mPvd = PxCreatePvd(*gFoundation);
PxPvdTransport* transport = PxDefaultPvdSocketTransportCreate(PVD_HOST,
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 parameter...
easier to author content at different scales and still have PhysX work at
get started simply pass a default object of this type. The record
parameter specifies whether to perform memory profiling. The optional
enables the debugging and profiling with the PhysX Visual Debugger.
The PhysX cooking library provides utilities for creating, converting, and serializing bulk data. Depending on your application, you may wish to link to the cooking library to process such data at runtime. Alternatively, you may be able to process all such data in advance and just load it into memory as required. Initialize the cooking library as follows:

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

The PxCookingParams struct configures the cooking library to target different platforms, use non-default tolerances or produce optional outputs. It is important to use consistent PxTolerancesScale values everywhere in your application (see Using Different Units 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 meshes can be directly inserted into PxPhysics without serialization using the PxPhysicsInsertionCallback. The default callback must be used and can be obtained using 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 either for code size reasons or to avoid use of certain subsystems, such as those pertaining to networking.

Initializing the extensions library requires the PxPhysics object:

```cpp
if (!PxInitExtensions(*mPhysics, mPvd))
    fatalError("PxInitExtensions failed!");
```
Optional SDK Components

When linking PhysX as a static library on memory constrained platforms, it is possible to avoid linking the code of some PhysX features that are not always used in order to save memory. Currently the optional features are:

- Articulations
- Height Fields
- Cloth
- Particles

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

```cpp
physx::PxPhysics* customCreatePhysics(physx::PxU32 version,
                                     physx::PxFoundation& foundation,
                                     const physx::PxTolerancesScale& scale,
                                     bool trackOutstandingAllocations
                                     physx::PxPvd* pvd)
{
    physx::PxPhysics* physics = PxCreateBasePhysics(version, foundation,
                                                    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, as we rely on 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 dll is possible to have delay-loaded PxFoundation, PxPvdSDK, PhysXCo 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 Phys Foundation and PxPvdSDK is made as follows:

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

 PxDelayLoadHook

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

```cpp
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, PhysXGpu, PnPvdSDK:

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

PxGpuLoadHook

The PxGpuLoadHook class supports loading of different versions of PhysX 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

Now the hook must be set for PhysX:

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

**PhysXCommon Secure Load**

All PhysX DLLs distributed by NVIDIA are signed. The PhysXCommon checked, when it is loaded by PhysX or PhysXCooking. If signa 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 and refer to the reference guide for details. To shut down the extension function `PxCloseExtensions()`. To shut down physics, call release() on the `PxPhysics` object, and this will clean up all of the physics objects:

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

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

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

This chapter explains how to use PhysX in multithreaded applications. The main aspects to using PhysX with multiple threads are:

- how to make read and write calls into the PhysX API from multiple threads without causing race conditions.
- how to use multiple threads to accelerate simulation processing.
- how to perform asynchronous simulation, and read and write while simulation is being processed.
Data Access from Multiple Threads

For efficiency reasons, PhysX does not internally lock access to its data structures, 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 particularly that it is not legal to perform a write operation on 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 threads to objects within a scene, to try and detect problems at the point when the illegal API call is made.

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 an APEX and a game engine's physics code. The scene lock provides a way for these systems to coordinate 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 simpler and more efficient application-specific mechanism that guarantees your application's correctness.
conditions. However, using the scene lock has two potential benefits:

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

There are four methods for acquiring / releasing the lock:

```cpp
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 \texttt{PxSceneLock.h}.

**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 threads are reading.
- If a thread holds a write lock then it may call both read and write API methods.
- Re-entrant read locks are supported, meaning a \texttt{lockRead()} on a thread that has already acquired a read lock is permitted. Each \texttt{lockRead()} must have a \texttt{unlockRead()}.
- Re-entrant write locks are supported, meaning a \texttt{lockWrite()} on a thread that has already acquired a write lock is permitted. Each \texttt{lockWrite()} must have a \texttt{unlockWrite()}.
- Calling \texttt{lockRead()} by a thread that has already acquired the write lock is permitted and the thread will continue to have read and write access. Each lock*() must have 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 an error, in release builds it will lead to deadlock.

• Writers are favored - if a thread attempts a `lockWrite()` while the read lock is acquired 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 access the scene. This prevents writers being starved in the presence of multiple readers.

• If multiple writers are queued then the first writer will receive priority, 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 to perform multiple operations. This minimizes the overhead of the lock, and in addition can prevent cases such as a sweep test in one thread seeing a rag doll that has been partially inserted by another thread.

Clustering writes can also help reduce contention for the lock, as acquiring a write will stall any other thread trying to perform a read access.
PhysX simulation is asynchronous by default. Start simulation by calling

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

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

To wait until simulation completes, call:

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

The boolean parameter to fetchResults denotes whether the call should wait for simulation to complete, or return immediately with the current completion status. See the API 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 `PxScene::simulate()` call (see figure below, blue area "1").
2. After the call to `PxScene::simulate()` has returned and before `PxScene::fetchResults()` call (see figure below, green area "2").

```
... 1 simulate() 2 fetchResults() 1 ...
```

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

In the second time slot the simulation is running and in the process, read...
the state of objects. Concurrent access from the user might corrupt the or lead to data races or inconsistent views in the simulation code. Hence the code'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, 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 scene (with some exceptions, see further below). This includes adding/removing them to/from a scene.

From the user perspective, API changes are reflected immediately. For example, if an object is created while the simulation is running, it can be accessed by other object. However, these changes are buffered so that the simulation sees the object state as it was when PxScene::simulate() was called. For instance, changes to the filter data of an object while the simulation is running are ignored for collision generation of the running step, and will only affect for the next simulation step.

When PxScene::fetchResults() is called, any buffered changes are flushed by the simulation are reflected in API view of the objects, and API changes are visible to the simulation code for the next step. User changes take precedence: 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 account the latest changes.

**Events involving removed objects**

Deleting objects or removing them from the scene while the simulation
affect the simulation events sent out at `PxScene::fetchResults()`. The behavior is as follows:

- `PxSimulationEventCallback::onWake()`, `::onSleep()` events will not get fired if an object is involved which got deleted/removed during the running simulation.
- `PxSimulationEventCallback::onContact()`, `::onTrigger()` events will get fired if an object is involved which got deleted/removed during the running simulation. The deleted/removed object will be marked as `PxContactPairHeaderFlag::eREMOVED_ACTOR_0`, `PxContactPairFlag::eREMOVED_SHAPE_0`, `PxTriggerPairFlag::eREMOVED_SHAPE_TRIGGER`). Furthermore, if `PxPairFlag::eNOTIFY_TOUCH_LOST`, `::eNOTIFY_THRESHOLD_FORCE_LOST` events were requested for the pair containing the deleted/removed object, these events will be created.

**Support**

Not all PhysX objects have full buffering support. Operations which cannot run while the simulation is in process are mentioned in the API documentation and the SDK aborts such operations and reports an error. The most important exceptions are as follows:

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

**Memory Considerations**

The buffers to store the object changes while the simulation is running on demand. If memory usage concerns outweigh the advantage of reading/writing objects in 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 resources. Tasks are created with dependencies so that they are resolved in a given order, when ready they are submitted to a user-implemented dispatcher for execution.

Middleware products typically do not want to create CPU threads for their own use, especially true on consoles where execution threads can have significant overhead. In the task model, the computational work is broken into jobs that are submitted to the 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 ready tasks to their respective dispatcher. There is a dispatcher for CPU tasks and GPU tasks assigned to the TaskManager.

TaskManagers are owned and created by the SDK. Each PxScene will have its own 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 the thread pool. Typically, there will be one single CpuDispatcher for the entire 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 being used.

PhysX includes a default CpuDispatcher implementation, but we prefer applications to implement this class themselves so PhysX and APEX can efficiently share CPU resources 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 must be thread-safe.

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

```cpp
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 SampleBase to show how the default dispatchers are created. `mNbThreads which PxDefaultCpuDispatcherCreate` defines how many worker threads the CPU dispatcher will have:

```cpp
PxSceneDesc sceneDesc(mPhysics->getTolerancesScale());
[...]
// create CPU dispatcher which mNbThreads worker threads
mCpuDispatcher = PxDefaultCpuDispatcherCreate(mNbThreads);
if(!mCpuDispatcher)
    fatalError("PxDefaultCpuDispatcherCreate failed!");
sceneDesc.cpuDispatcher = mCpuDispatcher;
#if 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. Creating more worker threads than hardware threads will often lead to worse performance. For platforms with a single execution core, the CPU dispatcher can be created with zero worker threads (PxDefaultCpuDispatcherCreate(0)). In this case all work will be executed on the thread that calls PxScene::simulate(), which can be more efficient.
Note: CudaContextManagerDesc support appGUID now. It only works on release build. If your application employs PhysX modules that use CUDA you need to use GUID so that patches for new architectures can be released for your game. You can obtain a GUID for your application from NVIDIA. The application should log any failure 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 submitted it to the appropriate dispatcher it is up to the dispatcher implementation to decide how and when the task will be run.

Often in game scenarios the rigid body simulation is time critical and the goal is to reduce the latency from simulate() to the completion of fetchResults(). The lowest latency will be achieved when the PhysX tasks have exclusive access to CPU resources during the update. In reality, PhysX will have to share compute resources with other game tasks.

Below are some guidelines to help ensure a balance between throughput and latency when mixing the PhysX update with other work.

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

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

**BaseTask**

BaseTask is the abstract base class for all task types. All task run() functions will be executed on application threads, so they need to be careful with their stack usage, use as 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 each simulation step for them to be executed. Tasks may be named at submission time, this allows them to be discoverable. Tasks will be given a reference count of 1 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. Once 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 programmer to avoid race hazards. You cannot add dependencies to tasks that have already been dispatched, and newly submitted Tasks must have their reference count decremented before that Task will be allowed to execute.

Synchronization points can also be defined using Task names. The TaskManager will assign the name a TaskID with no Task implementation. When all of the named TaskID's 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 resources. The ApexScene defines a number of named Tasks that the modules use to schedule their work: start after LOD calculations are complete, finish before the PhysX scene is stepped).
**LightCpuTask**

LightCpuTask is another subclass of BaseTask that is explicitly scheduled by the programmer. LightCpuTasks have a reference count of 1 when they are initialized, so their reference count must be decremented before they are dispatched. LightCpuTasks increment their continuation task reference count when they are initialized, and decrement the reference count when they are released (after completing their run()).

PhysX 3.x uses LightCpuTasks almost exclusively to manage CPU resources. For example, each stage of the simulation update may consist of multiple parallel tasks, each of these tasks has finished execution it will decrement the reference count on 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 resources, the TaskManager startSimulation() and stopSimulation() calls must be made each step to keep the GpuDispatcher synchronized.

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

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

```cpp
class Crab: public ClassType, public physx::PxLightCpuTask, public RenderMaterial
{
public:
    Crab(SampleSubmarine& sample, const PxVec3& crabPos, RenderMaterial& render);
    ~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, the application calls removeReference() on each Crab task, this in turn causes it to be submitted to the CpuDispatcher for update. Note that it is also possible to submit tasks directly (without manipulating reference counts) as follows:

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

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

```cpp
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 Threading information.

An example for explicit reference count modification and task dependency setup:

```cpp
// 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 refcount
// 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' scene system. PhysX also provides standalone functions for testing intersection between geometries, raycasting against them, and sweeping one geometry against another.

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

Geometries fall into two classes:

- **primitives** (PxBoxGeometry, PxSphereGeometry, PxPlaneGeometry) where the geometry object contains all of the data directly.
- **meshes or height fields** (PxConvexMeshGeometry, PxTriangleMeshGeometry, PxHeightFieldGeometry), where the geometry object contains a reference to a much larger object (PxConvexMesh, PxTriangleMesh, PxHeightField respectively). You can use these objects with different scales in each PxGeometry type that references them. The larger objects must be created using a cooking process, as described below.

When passed into and out of the SDK for use as simulation geometry, they are copied into and out of a PxShape class. It can be awkward in this context to retrieve the geometry without knowing its type, so PhysX provides a union-like class (PxGeometryHolder) that can be used to pass any geometry type by value. Each mesh (or height field) 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. The shape and actor is otherwise the same as for the sphere:
The function `PxTransformFromSegment()` converts from a line segment defining the capsule axis to a transform and halfheight.

**Boxes**

![Box Diagram]

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

```cpp
PxShape* aBoxShape = PxRigidActorExt::createExclusiveShape(*aBoxActor, 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" will collide with it.

The Plane lies on the YZ plane with "above" pointing towards positive X plane equation to an equivalent transform, use PxTransformFromPlaneEquation(). PxPlaneEquationFromTransform() performs the reverse conversion.

A PxPlaneGeometry has no attributes, since the shape's pose entirely defines the 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 contains the line between them. A PxConvexMesh is a convex polyhedron represented as a set of vertices and polygonal faces. The number of vertices and faces of a convex mesh in PhysX 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 steps explain how to create a simple square pyramid.

First, define the vertices of the convex object:

```cpp
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:

```cpp
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:

```cpp
PxDATAOutputStream 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 instances the mesh:

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

Alternatively the PxConvexMesh can be cooked and directly inserted into PxPhysics without stream serialization. This is useful if real-time cooking is required but recommended to use offline cooking and streams. Here is an example to 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 | PxConvexFlag::eFAST_INERTIA_COMPUTATION

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

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

Please note that mesh validation is required for debug and checked builds; meshes from unvalidated input descriptors may result in undefined behavior. Providing PxConvexFlag::eFAST_INERTIA_COMPUTATION flag the volume integration will use SIMD code path which does faster computation but with lesser precision.

The user can optionally provide a per-instance PxMeshScale in PxConvexMeshGeometry. The scale defaults to identity. Negative scale is not supported for convex meshes.

PxConvexMeshGeometry also contains flags to tweak some aspects of
By default the system computes approximate (loose) bounds around convex objects. Using PxConvexMeshGeometryFlag::eTIGHT_BOUNDS enables smaller bounds, which are more expensive to compute but could result in improved simulation performance when a lot of convex objects are interacting with each other.

PxConvexMeshGeometry also contains a variable called maxMargin. By default, it is set to 3.4e38f. If the maxMargin is smaller than the margin amount calculated by the contact gen, it will choose the smallest margin for the shrunk shape to perform incremental update using GJK algorithm. In this case, application might notice some visibility artefacts around the vertex collision. If the maxMargin is set to be a small value, this will improve the visibility of these artefacts. If maxMargin is set to zero, PCM will use the original shape for the GJK algorithm. This will result in no artefacts for this approach. However, there is a trade-off between performance and accuracy.

**Convex Mesh cooking**

Convex Mesh cooking transforms the mesh data into a form which allows the SDK to perform efficient collision detection. The input to cooking is defined using PxConvexMeshDesc.

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

**If Only Vertex Points are Provided**

When providing only vertices, set the PxConvexFlag::eCOMPUTE_CONVEX flag to compute the mesh:

```cpp
PxConvexMeshDesc convexDesc;
convexDesc.points.count = 20;
convexDesc.points.stride = sizeof(PxVec3);
convexDesc.points.data = convexVerts;
convexDesc.flags = PxConvexFlag::eCOMPUTE_CONVEX;
convexDesc.maxVerts = 10;
PxDefaultMemoryOutputStream buf;
```
if(!cooking.cookConvexMesh(convexDesc, buf))
    return NULL;

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

This routine can sometimes fail when the source data is geometrically challenging, for example if it contains a lot of vertices close to each other. If cooking fails, an error is reported to the error stream and the routine returns false.

If PxConvexFlag::eCHECK_ZERO_AREA_TRIANGLES is used, the algorithm will not include triangles with an area less than PxCookingParams::areaTestEpsilon. If the algorithm cannot find 4 initial vertices without a small triangle, PxConvexMeshCookingResult::eZERO_AREA_TEST_FAILED is returned, indicating that the provided vertices were in a very small area and the cooker could not produce a valid hull. The toolkit helper function PxToolkit::createConvexMeshSafe illustrates the most robust strategy for convex mesh cooking. First it tries to create the hull without inflation. If 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 transformation in PxShape, otherwise addional PxConvexFlag::eSHIFT_VERTICES flag for the mesh computation.

If huge amount of input vertices are provided, it might be useful to quantize the 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 whose vertices are a subset of the original vertices, and the number of vertices is guaranteed to be no more 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 set.
- If the vertexLimit is reached, expand the limited hull around the vertices to ensure we encapsulate all the input vertices.
- Compute a vertex map table. (Requires at least 3 neighbor polygons for each vertex.)
- 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 further than PxCookingParams::planeTolerance from the hull, if not that vertex is dropped.

Inflation Based Incremental Algorithm

This algorithm always uses the PxConvexFlag::eINFLATE_CONVEX flag and inflates the 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 set.
- 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 input vertices. The algorithm is significantly slower than the quickhull and produces significantly less stable results. It is recommended to use the quickhull algorithm.
Vertex Limit Algorithms

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

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

When plane shifting is enabled (PxConvexFlag::ePLANE_SHIFTING), the hull computation stops when vertex limit is reached. The hull planes are then shifted to contain all input vertices, and the new plane intersection points are then used to compute the final 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 contained within the resulting hull. However, it can be used with a vertex limit as low as 4, making it 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 polygons (hullPolygons):

```cpp
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 the PxConvexMesh directly. This is the fastest way to create a convex mesh. The SDK requires at least 3 neighbor polygons for each vertex. Otherwise, acceleration...
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 each vertex.
- 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 collection of vertices and the triangle indices. Triangle mesh creation requires use of the cooking library.

Shutdown):
Alternatively `PxTriangleMesh` can be cooked and directly inserted into stream serialization. This is useful if real-time cooking is required to use offline cooking and streams. Example how to speed if needed:

```cpp
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;

// should be validated before cooked without the mesh cl
bool res = theCooking->validateTriangleMesh(meshDesc);
PX_ASSERT(res);
```
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_TRIANGLE* to the user if the mesh contains large triangles, which should be tessellated to ensure better simulation and CCT stability.

Like height fields, triangle meshes support per-triangle material indices. To use per-triangle materials for a mesh, provide per-triangle indices to the cooking library in the mesh descriptor. Later, when creating the PxShape, supply a table mapping the index 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 invalid triangles (containing out-of-range vertex references), duplicate triangles, and zero-area triangles. When this happens, PhysX optionally outputs a mesh remapping table that links each internal triangle to its source triangle in the user's data.

There are multiple parameters to control mesh creation.

In *PxTriangleMeshDesc*: 

```cpp
PxTriangleMesh* aTriangleMesh = theCooking->createTriangleMesh(me thePhysics->getPhysicsInsertionCallback());
```
- *materialIndices* defines per triangle materials. When a triangle collides with another object, a material is required at the collision point. If *materialIndices* is NULL, then the material of the PxShape instance is used.

In *PxCookingParams*:

- *scale* defines Tolerance scale is used to check if cooked triangles are too huge. This check will help with simulation stability.

- *suppressTriangleMeshRemapTable* specifies whether the face remap table is created. If not, this saves a significant amount of memory, but the SDK will not be able to provide information about which original mesh triangle is hit in collisions, sweeps or raycasts.

- *buildTriangleAdjacencies* specifies if the triangle adjacency information is created. The adjacent triangles can be retrieved for a given triangle using the `getTriangle` function.

- *meshPreprocessParams* specifies mesh pre-processing parameters:
  - *PxMeshPreprocessingFlag::eWELD_VERTICES* enables vertex welding during triangle mesh cooking.
  - *PxMeshPreprocessingFlag::eDISABLE_CLEAN_MESH* disables mesh clean process. Vertices duplicities are not searched, huge triangles test is not done. Does speed up the cooking.
  - *PxMeshPreprocessingFlag::eDISABLE_ACTIVE_EDGES_PRECOMPUTE* disables vertex edge precomputation. Makes cooking faster but slows down contact generation.

- *meshWeldTolerance* - If mesh welding is enabled, this controls the distance at which vertices are welded. If mesh welding is not enabled, this value defines the acceptance distance for mesh validation. Provided no two vertices are within this distance, the mesh is considered to be clean. If not, a warning will be emitted. Having a clean mesh 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 up to PhysX 3.3. It has great performance and is supported on all platforms.
  - **PxBVH34MidphaseDesc** - `PxMeshMidPhase::eBVH34` implementation introduced in PhysX 3.4. It can be significantly faster both in terms of cooking performance and runtime performance, but it is currently only available on platforms supporting the SSE2 instruction set.

**PxBVH33MidphaseDesc params:**

- **meshCookingHint** specifies mesh hierarchy construction preferences. It enables better cooking performance over collision performance, for applications where cooking performance is more important than best quality mesh creation.
- **meshSizePerformanceTradeOff** specifies the trade-off between mesh size and runtime performance.

**PxBVH34MidphaseDesc params:**

- **numTrisPerLeaf** specifies the number of triangles per leaf. Less 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 values on a regular, rectangular sampling grid:

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

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

The flag and materials refer to the cell below and to the right of the sample point, indicating along which diagonal to split it into triangles, and the materials of those triangles. A special predefined material `PxHeightFieldMaterial::eHOLE` specifies a hole in the height field. See the reference documentation for `PxHeightFieldSample` for more details.
Examples:

<table>
<thead>
<tr>
<th>Tessellation flags</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0,0</td>
<td><img src="image1" alt="Result 1" /></td>
</tr>
<tr>
<td>0,0,0</td>
<td><img src="image2" alt="Result 2" /></td>
</tr>
<tr>
<td>0,0,0</td>
<td><img src="image3" alt="Result 3" /></td>
</tr>
</tbody>
</table>

- **Sample**: The sample point is located at the origin.
- **Triangle 0**: The first triangle is defined by the points (0,0), (1,0), and (0,1).
- **Triangle 1**: The second triangle is defined by the points (0,1), (1,1), and (1,0).
To tell the system the number of sampled heights in each direction, instantiate a PxHeightField object:

```cpp
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:

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

The row and column scales tell the system how far apart the sampled points lie in the associated direction. The height scale scales the integer height values to a floating point range.

The variant of `createExclusiveShape()` used here specifies an array of materials for the height field, which will be indexed by the material indices of each cell to resolve collisions with that cell. The single-material variant may be used instead, but the
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 createHeightField, does precompute and store the edge information. This allows much for heightfield, since the edges are already precomputed. It is very useful to create heightfields in the runtime, since it does improve the speed of 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 heightfield and utilizes the same low-level contact generation code that 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 if CCD is not enabled.

The legacy heightfield collision code, which was default in previous

works differently from triangle mesh contact generation. In addition to generating contacts with shapes touching the surface of the heightfield, it generates contacts with shapes that are beneath the surface. The heightfield's "thickness" is used to control how far beneath the 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 any shape below the surface whose bounds intersect the heightfields extruded bounds.

Unified heightfield contact generation is enabled by calling:

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

Legacy heightfield contact generation is enabled by calling:

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

These calls must be made before and scenes have been created, otherwise warnings will be issued. The heightfield collision setting is a global setting, and it applies to all scenes.

If PxCreatePhysics(...) is called, this will automatically call PxRegisterHeightFields to register the default, unified heightfield collision approach. If PxCreateBasePhysics(...) is called, no heightfield contact generation is registered by default. If heightfields are used, 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 defines 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 mesh triangles when scale.x*scale.y*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:

```cpp
// 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(*myActor, myTriMeshShape);
```

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

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

Height fields can also be scaled, using scale factors stored in PxHeight
this case the scale is assumed to be along the axes of the rows, columns, and height directions of the height field. The scaling of is demonstrated in SampleNorthPoleBuilder.cpp:

```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 creation with `PxShape::setGeometry()`, the geometry is copied into the SDK's internal structures. If you know the type of a shape's geometry you may retrieve it directly:

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

The status return code is set to false if the shape's geometry is not of the expected 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` reference as an argument.

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

```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.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:

```cpp
bool testForOverlap(const PxShape& s0, const PxShape& s1)
{...}
```
return PxGeometryQuery::overlap(s0.getGeometry().any(), PxSha
s1.getGeometry().any(), PxSha
Vertex and Face Data

Convex meshes, triangle meshes, and height fields can all be queried for vertex and face 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 height field using PxMeshQuery::getTriangle function. You can also retrieve adjacent triangles to the given triangle (triangle triangleNeighbour[i] shares the edge vertex[i]-v[i] triangle indexed as 'triangleIndex', where vertex is in the range from 0 to 2). To enable this 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 index buffer which 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], 0);
    }
    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.mIndexBase], and the count of vertices is given by face.mNbVerts.

Triangle Meshes

Triangle meshes contain arrays of vertices and index triplets which define the triangles by indexing into the vertex buffer. The arrays can be accessed directly from the shared 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, specified when the mesh was originally cooked. To determine the storage format at runtime, use the
\texttt{const bool has16bitIndices = tm->has16BitTriangleIndices();}

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

\begin{verbatim}
const PxU16* triIndices = (const PxU16*)tris;
const PxU16 index = triIndices[3*i + j];
\end{verbatim}

The corresponding vertex is:

\begin{verbatim}
const PxVec3& vertex = verts[index];
\end{verbatim}

**Height Fields**

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

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

\begin{verbatim}
const PxHeightFieldGeometry& geometry = geom.heightField();
\end{verbatim}

The height field has three scaling parameters:

\begin{verbatim}
const PxReal rs = geometry.rowScale;
const PxReal hs = geometry.heightScale;
const PxReal cs = geometry.columnScale;
\end{verbatim}

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

\begin{verbatim}
PxHeightField* hf = geometry.heightField;
const PxU32 nbCols = hf->getNbColumns();
const PxU32 nbRows = hf->getNbRows();
\end{verbatim}

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

\begin{verbatim}
const PxU32 nbVerts = nbRows * nbCols;
\end{verbatim}
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 ith row and i*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)).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 it's 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(heightField);

// create modified HF samples, this 10-sample strip will be used.
// Source samples that are out of range of target heightfield will be clipped with no error.
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 with new sample data

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

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

Please also note that PxShape::setGeometry() does not guarantee correct/continuous 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.
Rigid Body Overview
Introduction

This chapter will introduce the fundamentals of simulating rigid body dynamics using the NVIDIA PhysX engine.
Rigid Body Object Model

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

Class | Extends | Functionality
--- | --- | ---
PxBase | N/A | Reflection/querying object types.
PxActor | PxBase | Actor name, actor flags, dominance, aggregates, query world bounds.
PxRigidActor | PxActor | Shapes and transforms.
PxRigidBody | PxRigidActor | Mass, inertia, velocities, body flags.
PxRigidStatic | PxRigidActor | Interface for static body in the scene body has implicit infinite mass/inertia.
PxRigidDynamic | PxRigidBody | Interface for dynamic rigid body in th Introduces support for kinematic targ
sleeping.

<table>
<thead>
<tr>
<th>PxArticulationLink</th>
<th>PxRigidBody</th>
<th>Interface for a dynamic rigid body link PxArticulation. Introduces support for articulation and adjacent links.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PxArticulation</td>
<td>PxBase</td>
<td>Defines interface for a PxArticulation contained referencing multiple PxArt bodies.</td>
</tr>
</tbody>
</table>

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

Now use the method PxScene::simulate() to advance the world forward.

Here is simplified code from the samples' fixed stepper class:

```cpp
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 is greater than 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 deal with time when stepping the simulation forward.

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

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

True indicates that the simulation should block until it is finished, so results are guaranteed to be available. When fetchResults completes, 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. The advantage of this to perform rendering work in parallel with physics. However, until 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-simulate
chapter *Threading* for more details about reading and writing while
running.

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

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

See *Simulation memory* for details of how memory is used in simulation
Rigid Body Collision
Introduction

This section will introduce the fundamentals of rigid body collision.
Shapes

Shapes describe the spatial extent and collision properties of actors. They are used for three purposes within PhysX: intersection tests that determine the contacting features of rigid objects, scene query tests such as raycasts, and defining trigger volumes 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 be specified upon creation. The following code creates a shape with a sphere geometry and a specific material:

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

The method PxRigidActorExt::createExclusiveShape() is equivalent 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 with other actors. You can use shape sharing to reduce the memory when you have many actors with identical geometry, but shape with very strong restriction: you cannot update the attributes of a shared shape 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 copied into the shape. 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 shapes that are attached to dynamic actors, unless the dynamic actor is configured 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:

```c
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 scene and contact tests. By default, a shape will participate in both.

The following pseudo-code configures a PxShape instance so that it no longer participates in shape pair intersection tests:

```cpp
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:

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

To disable a PxShape instance from scene query tests:

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

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

```cpp
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 by the 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, heighfield) shape. If this shape has a simulation shape flag, the kinematic. If you change the flag to not simulated, you can switch even

To setup kinematic triangle mesh see following code:

```cpp
PxRigidDynamic* meshActor = getPhysics().createRigidDynamic(PxTransform());
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:

```cpp
PxRigidDynamic* meshActor = getPhysics().createRigidDynamic(PxTransform());
PxShape* meshShape;
if (meshActor) {
    meshActor->setRigidDynamicFlag(PxRigidDynamicFlag::eKINEMATIC,
        PxTriangleMeshGeometry triGeom;
        triGeom.triangleMesh = triangleMesh;
        meshShape = PxRigidActorExt::createExclusiveShape(*meshActor,
            defaultMaterial);
        getScene().addActor(*meshActor);
        PxConvexMeshGeometry convexGeom = PxConvexMeshGeometry(convexBox);
        convexShape = PxRigidActorExt::createExclusiveShape(*meshActor,
            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 good generic choice with great performance when many objects are sleeping. Performance can degrade significantly though, when all objects are moving, or when large numbers of objects are added to or removed from the broad-phase. This algorithm does not need 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 issues as `eSAP` when all objects are moving or when inserting large numbers of objects. However, its generic performance when many objects are sleeping might be inferior to `eSAP`, and it requires users to define world bounds in order to work.

The desired broad-phase algorithm is controlled by the `PxBroadPhase` and the `PxSceneDesc` structure.
Regions of Interest

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

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

Regions can be defined by the `PxBroadPhaseRegion` structure, along with a user-data assigned to them. They can be defined at scene creation time or at runtime using the `PxScene::addBroadPhaseRegion` function. The SDK returns handles assigned to the newly created regions, that can be used later to remove regions using the `PxScene::removeBroadPhaseRegion` function.

A newly added region may overlap already existing objects. The SDK will add those objects to the new region, if the `populateRegion` parameter from 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. The region would then be created empty, and it would only be populated either with objects added to the scene after the region has been created, or with previously existing objects when they are updated (i.e. when they move).

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

Runtime information about current regions can be retrieved using \textit{PxScene::getNbBroadPhaseRegions} and \textit{PxScene::getBroadPhaseReg}

The maximum number of regions is currently limited to 256.
Broad-phase Callback

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

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

- 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, we need to be notified when the submarine touched a mine, or the chain of a mine, so that they blow up. The crab's AI also needs to know when crabs touch the heightfield.

Before we can understand what the sample does to achieve this, we need to understand the possibilities of the SDK filtering system. Because filtering potentially interacting pairs happens in the deepest parts of the simulation engine, and needs to be applied to all pairs of objects that come near each other, it is particularly performance sensitive. A way to implement it would be to always call a callback function for every potentially interacting pair, where the application, based on the two object pointers, using some custom logic -- like consulting its game data base -- whether the pair should interact. Unfortunately this quickly becomes too slow if done for a very large game, especially if the collision detection processing happens on a remote GPU or an other kind of vector processor with local memory, which would have to suspend its parallel computations, interrupt the main processor that runs game code, and have it execute the callback before it can continue. Even if it were to be executed on a CPU, it would likely be done so simultaneously on multiple cores or hyperthreads, code would have to be put in place to make sure that concurrent access is safe. Far better is to use some kind of fixed function logic that can execute on the remote processor. This is what we did in PhysX 2.x -- unfortunately the simple group based filtering rules we provided were not flexible enough to cover all applications. We introduce both a shader system, which lets the developer implement an arbitrary system of rules using code that runs on the vector processor (and is therefore not able to access any current game data base in main memory), which is more flexible than fixed function 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 shader
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 memory other than arguments of the function and its own local stack variables -- because the function may be compiled and executed on a remote processor. SampleSubmarineFilterShader() will be called for all pairs of shapes that have an axis aligned bounding box and that are found to intersect for the first time. All behavior beyond that is determined by what SampleSubmarineFilterShader() returns.

The arguments of SampleSubmarineFilterShader() include PxFilterObjectAttributes and PxFilterData for the two objects, and a constant block of memory. Note that the two objects are NOT passed, because those pointers refer to the computer's main memory, and that may, as we said, not be available to the shader, so it is not be very useful, as dereferencing them would likely crash the execution of the shader.
PxFilterObjectAttributes and PxFilterData are intended to contain all the information that one could quickly glean from the pointers. PxFilterObjectAttributes are intended to contain all the information that one could quickly glean from the pointers. PxFilterObjectAttributes encode the type of object: For example PxFilterObjectType::eRIGID_DYNAMIC, or even ::ePARTICLE_SYSTEM. Additionally, it lets you find out if the object is kinematic, or a trigger.

Each PxShape and PxParticleBase object in PhysX has a member variable 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 the application can give to the shader to operate on. You will want to use this to encode rules about what to filter and what not.

Finally, SampleSubmarineFilterShader() also has a PxPairFlags parameter input. It tells the SDK if it should ignore the pair for good (eKILL), ignore the pair while it is overlapping, but ask again, when filtering related data changes for one of the objects (eSUPPRESS), or call the low performance but more flexible CPU callback if the shader 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 notify the user when the pair really starts to touch, not just potentially.

Let us look at what the above shader does:

```cpp
// 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 behavior (notify the application about start and end of touch), and otherwise perform...
detection between them.

```cpp
// 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. In addition, there is a rule based on the filterDatas that determines particular pairs where touch notifications. To understand what this means, we need to know the special meaning that the sample gives to the filterDatas.

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

```cpp
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 PxFilterData::word0 to this FilterGroup type. Then, it puts a bit mask that specifies each type of generate a report when touched by an object of type word0 into word1. In the samples whenever a shape is created, but because shape is encapsulated in SampleBase, it is done after the fact, using this function.
```c
void setupFiltering(PxRigidActor* actor, PxU32 filterGroup, PxU32 filterGroup::eSUBMARINE, FilterGroup::eMINE_LINK); setupFiltering(link, FilterGroup::eMINE_LINK, FilterGroup::eSUBMARINE, FilterGroup::eMINE_LINK); setupFiltering(mineHead, FilterGroup::eMINE_HEAD, FilterGroup::eSUBMARINE, FilterGroup::eMINE_LINK); setupFiltering(heightField, FilterGroup::eHEIGHTFIELD, FilterGroup::eSUBMARINE, FilterGroup::eHEIGHTFIELD); setupFiltering(mCrabBody, FilterGroup::eCRAB, FilterGroup::eHEIGHTFIELD);
```
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 use case is a ragdoll, made of multiple different actors. Without aggregates, this gives rise to as many broad-phase entries as there are shapes in the ragdoll. It is typically more efficient to represent the ragdoll in the broad-phase as a single entity, and perform 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:

```cpp
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 should be set 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 mechanism, as it bypasses all internal filtering logic. A typical use case would be an aggregate of static or kinematic actors.

Note that both the maximum number of actors and the self-collisions attribute are immutable.
Populating an Aggregate

Adds an actor to an aggregate as follows:

```cpp
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 aggregate to the scene and then the actors to the aggregate.

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

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

Similarly, to remove the aggregate from the scene:

```cpp
scene->removeAggregate(*aggregate);
```
Releasing an Aggregate

To release an aggregate:

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

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

Adding many objects to a scene in one frame can be a costly operation. One 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 the cost of object insertion into the broad-phase structure over several frames, spawn the debris in a PxAggregate, then remove each actor from the aggregate and re-insert it into the scene over those frames.
Trigger Shapes

Trigger shapes play no part in the simulation of the scene (though they can participate in scene queries). Instead, their role is to report that there has been an overlap with another shape. Contacts are not generated for the intersection, and as a result contact reports are not available for trigger shapes. Further, because they play no part in the simulation, the SDK will not allow the the eSIMULATION_SHAPE and eTRIGGER_SHAPE flags to be raised simultaneously; that is, if one attempt to raise the other will be rejected, and an error will be passed to the error stream.

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

```cpp
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 through the implementation of PxSimulationEventCallback::onTrigger in the PSub class, a sub-class of PxSimulationEventCallback:

```cpp
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 are shapes. If it is found that the treasure has been touched by the submarine, then the flag `gTreasureFound` is set true.
Interactions

The SDK internally creates an interaction object for each overlapping pair reported by 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 assume that objects are created regardless of the involved objects' types (rigid body, trigger, cloth, etc.) 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 message and the extra interactions are ignored.
In this chapter we cover a number of topics that are also important to ur are comfortable with setting up a basic rigid body simulation world.
Velocity

A rigid body's motion is separated into linear and angular velocity components. During 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 collisions or joints.

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

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

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

```cpp
void PxRigidBody::setLinearVelocity(const PxVec3& linVel, bool autowake);
void PxRigidBody::setAngularVelocity(const PxVec3& angVel, bool autowake);
```
Mass Properties

A dynamic actor needs mass properties: the mass, moment of inertia, and the mass frame which specifies the position of the actor's center of mass. The easiest way to calculate mass properties is to use the PxRigidBodyExt::updateMassAndInertia() helper function, which will set all three properties based on the actor's shapes and a uniform density value. Variants of this function allow combinations of per-shape densities and manual specification of some 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 an empty shell with the bottom filled with some heavy material. The low centers of mass cause them to move back to an upright position after they have been tilted. They come in different flavors, depending on how the mass properties are set:

The first is basically massless. There is just a little sphere with a relatively high 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 bigger inertia. Later on, the center of mass is moved to the bottom of the actor. This approximation is by no means physically correct, but the resulting snowman feels a bit more filled.

The third and fourth snowman use shapes to calculate the mass. The first one 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 about the low center of mass that we pass to the calculation routine. Note how much slower the wobbling is for the second case although both have the same mass. The head accounts for much more in the moment of inertia (the distance squared).

The last snowman's mass properties are set up manually. The sample uses different values for the 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 to rotation around the X-axis and high resistance around Y and Z. As a consequence, the snowman will 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 positive values for the mass and each principal axis of inertia. However, it is legal to provide 0 values. When provided with a 0 mass or inertia value, PhysX interprets this to mean 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 of what 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 assume that we are creating a common structure - a windmill. The code to construct the parts of the windmill are provided below:

```cpp
PxRigidDynamic* dyn = physics.createRigidDynamic(PxTransform(PxVec3(0.f, 0.f, 0.f));
PxRigidActorExt::createExclusiveShape(*dyn, PxBoxGeometry(2.f, 0.f, 0.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(*t, PxBoxGeometry(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 cross representing 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 at a constant angular velocity indefinitely. However, if another object collided with the turbine, 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 remain in position when other bodies interact with it. One such approach might be to make the turbine have infinite mass and inertia. In this case, any interactions with other bodies would not affect the turbine at all:

```cpp
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 velocities cannot be affected 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 kinematic body.

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

```cpp
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 has infinite mass so its linear velocity cannot be changed by any constraints. However, in this example, the body's inertia is configured to permit the body's angular velocity to be affected by constraints around one principal axis or 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 by using forces. Because of the law:

\[ f = m \times a \text{ (force = mass \times acceleration)} \]

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

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

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

```c++
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& pos, PxForceMode::Enum mode, bool wakeup);
void PxRigidBodyExt::addForceAtLocalPos(PxRigidBody& body, const PxVec3& pos, PxForceMode::Enum mode, bool wakeup);
void PxRigidBodyExt::addLocalForceAtPos(PxRigidBody& body, const PxVec3& pos, PxForceMode::Enum mode, bool wakeup);
void PxRigidBodyExt::addLocalForceAtLocalPos(PxRigidBody& body, const PxVec3& pos, PxForceMode::Enum mode, bool wakeup);
```
The PxForceMode member defaults to PxForceMode::eFORCE to apply simple forces. There are other possibilities. For example PxForceMode::eIMPULSe will apply an impulsive force. PxForceMode::eVELOCITY_CHANGE will do the same, but ignore the mass of the body, effectively leading to an instantaneous velocity change.

**Note:** The methods in PxRigidBodyExt support only the force modes eFORCE and eIMPULSe.

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

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

A use case for this function might be to predict an updated velocity for a game object so that asset loading may be initiated in advance of the simulation frame if the body is likely to exceed a threshold velocity at the end of the frame. The impulsive force or impulsive torque are simply the force and torque that are to be applied to the body multiplied by the timestep of the simulation frame. Neglecting the effect of constraint and contact forces, the change in linear and angular velocity that are expected to arise in the next simulation frame are returned in deltaLinearVelocity and deltaAngularVelocity. The predicted linear velocity can then be computed with body.getLinearVelocity() + deltaLinearVelocity, while the predicted angular velocity can be computed with body.getAngularVelocity() + deltaAngularVelocity.

If required, it is possible to immediately update the velocity of the body using 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 simple to apply. For a scene-wide gravity effect, or any other uniform force field, set the PxScene class’ gravity vector using PxScene::setGravity().

The parameter is the acceleration due to gravity. In meters and seconds, this will have a magnitude of about 9.8 on earth, and should point downwards. The force that will be applied at the center of mass of each body in the scene is this acceleration vector 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 the simulation. For performance reasons the change will not wake up sleeping actors automatically. 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 vector for the whole scene, then apply your own gravity force to rigid bodies, each frame. This can be used 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():

```cpp
PxMaterial* mMaterial;
mMaterial = mPhysics->createMaterial(0.5f, 0.5f, 0.1f); // static friction, dynamic 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 can be combined in various ways. See the reference documentation for PxMaterial for more details.

PhysX objects whose collision geometry is a triangle mesh or a heightfield 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 (sometimes called kinetic friction). Friction resists relative lateral motion of two surfaces in contact. These two coefficients define a relationship between the normal forces 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 that is applied 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 < 1 is said to be inelastic.
**Sleeping**

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

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

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

```cpp
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 certain time. Basically, every dynamic rigid actor has a wake counter which gets decremented by 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 step, the wake counter gets reset to a minimum default value and the whole process starts anew. Once the wake counter reaches zero, it does not get decremented any further and the actor is ready to go to sleep. Please note that a zero wake counter does not mean that the actor has to be asleep, it only indicates that it is ready to go to sleep. There are other factors that might keep an actor awake for a while longer.

The energy threshold as well as the minimum amount of time an actor will stay awake can be manipulated using the following methods:

```cpp
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 asleep 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 consequence, it is not allowed to use setWakeCounter() for kinematic actors. The wake counter 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 all the 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 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 larger than zero.
- PxRigidBody::setLinearVelocity(), ::setAngularVelocity(), if the velocity is non-zero.
- PxRigidBody::addForce(), ::addTorque(), if the torque is non-zero.

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

- PxRigidDynamic::setKinematicTarget() in the case of a kinematic actor (it 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` with `PxActorFlag::eDISABLE_SIMULATION`.
- `PxScene::resetFiltering()`.
- `PxShape::setSimulationFilterData()`, if the subsequent re-filtering causes 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 it 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 an actor 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:

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

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

The API reference documents exactly which methods cause an actor to

### Sleep state change events

As mentioned above, PhysX provides an event system that reports changes
state of dynamic rigid bodies during `PxScene::fetchResults()`:

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

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

- A body added since the previous `fetchResults()` or `flushSimulation()` generates 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.

Sometimes it is desirable to detect transitions between awake and asleep states, for example, keeping track of the number of awake bodies. Suppose a sleeping body becomes awake again, and during the next simulation step, `onWake()` is called. Even though B's sleep state did not change during simulation, since the previous `fetchResults()`, and so an `onWake()` event will be generated. If the counter is incremented again in response to this event, its value will be incorrect.

To use sleep state events to detect transitions, a record of the sleep state for objects of interest has to be kept, for example in a hash. When processing an event, this record 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 scheme is provided by kinematic actors.

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

To create a kinematic actor, simply create a regular dynamic actor then set its kinematic flag:

```cpp
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, this mass will not actually be used for anything while the actor is in kinematic mode.

Caveats:

- It is important to understand the difference between PxRigidDynamic::setKinematicTarget() and PxRigidActor::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, with setGlobalPose() 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 pushes it back. As a result, a kinematic can easily squish a dynamic actor against a static actor or against another kinematic actor. As a result, the squished dynamic object can deeply 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 these cases if PxSceneFlag::eENABLE_KINEMATIC_PAIRS or ::eENABLE_KINEMATIC_STATIC_PAIRS gets set.
Active Transforms

**Note:** the active transforms are currently deprecated. See next paragraph about Active Actors for its replacement.

The active transforms API provides an efficient way to reflect actor transform changes 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 list, this approach is potentially much more efficient than, for example, analyzing each actor in the scene individually.

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

```cpp
// 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*>(activeTransforms[i].actor2World);
    renderObject->setTransform(activeTransforms[i].actor2World);
}
```

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

**Note:** Since the target transform for kinematic rigid bodies is set by the user, kinematics 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 transform changes 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 active PxActor objects is generated. 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 individually.

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

```cpp
// 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*>(activeActors[i]);
    renderObject->setTransform(activeActors[i]->getGlobalPose());
}
```

**Note:** PxSceneFlag::eENABLE_ACTIVE_ACTORS must be set on the scene for the active actors array to be generated.

**Note:** Since the target transform for kinematic rigid bodies is set by the user, kinematics 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 imbues the dominant body in a pair with infinite mass. This mass modification within the constraint solver and, as such, can override the bodies in a pair. Similar effects can be achieved through local mass contact modification but dominance has the advantage of being handled automatically within the SDK so does not incur the additional memory and performance overhead 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 bodies are placed in dominance group 0. An actor can be assigned to a dominance group using the following method on PxActor:

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

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

```cpp
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:

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

The user can define 3 different states for a given PxDominanceGroupPair to indicate 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 groups and configures the dominance group to make actorA dominate actorB:

```cpp
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 can be performed using the following methods on PxJoint:

```cpp
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 0 or 1 and dominance values are applied uniformly to both the inverse mass and inverse inertia scales, which accept any values within the range [0, PX_MAX_REAL], 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 dominance body C
- Body C dominates body A

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

When the motion of a rigid body is constrained either by contacts or joints, the constraint solver comes into play. The solver satisfies the constraints on the bodies 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 include position iterations and 1 velocity iteration. Those counts may be set in a body using the following function:

```cpp
void PxRigidDynamic::setSolverIterationCounts(PxU32 minPositionIters, PxU32 maxPositionIters, PxU32 maxVelocityIters);
```

Typically it is only necessary to significantly increase these values for objects with lots of joints and a small tolerance for joint error. If you find a need to use a value higher than 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. However, the solver permits a maximum of 32 friction patches per contact manager (pair of shapes). If more than 32 friction patches are produced, which may be due to very complex collision geometry or very large contact offsets, the solver will ignore the remaining patches. 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 mode called "immediate mode". This provides an API to access the low-level contact and constraint solver. This approach currently supports only CPU rigid bodies and does not support articulations, clothing or particles.

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

The API provides a function to perform contact generation:

```cpp
PX_C_EXPORT PX_PHYSX_CORE_API bool PxGenerateContacts(const PxGeometry& mesh, const PxReal contactDistance, const PxReal meshContactMargin);
```

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

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

```cpp
PX_C_EXPORT PX_PHYSX_CORE_API void PxConstructSolverBodies(const PxRigidBodyData& bodyDesc);
PX_C_EXPORT PX_PHYSX_CORE_API void PxConstructStaticSolverBody(const PxRigidBodyData& bodyDesc);
```

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

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

```cpp
PX_C_EXPORT PX_PHYSX_CORE_API PxU32 PxBatchConstraints(PxSolverConstraintDesc* outOrderedConstraintDescs);
```
Batching constraints reorders the provided constraints and produces batchHeaders, which can be used by the solver to accelerate constraint solving by grouping together independent constraints and solving them in parallel using multiple lanes in SIMD registers. This process is entirely optional and can be bypassed if not desired. Note that this will change the order in which constraints are processed, which can change the outcome of the solver.

The following methods is provided to create contact constraints:

```c
PX_C_EXPORT PX_PHYSX_CORE_API bool PxCreateContactConstraints(PxConstraintBatchHeader allocator, PxReal invDt, PxReal bounceThreshold)
```

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

The following methods are provided to create joint constraints:

```c
PX_C_EXPORT PX_PHYSX_CORE_API bool PxCreateJointConstraints(PxConstraintBatchHeader)
PX_C_EXPORT PX_PHYSX_CORE_API bool PxCreateJointConstraintsWithShaders(PxConstraintBatchHeader)
```

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

The following method solves the constraints:

```c
PX_C_EXPORT PX_PHYSX_CORE_API void PxSolveConstraints(PxConstraintBatchHeader linearMotionVelocity, PxVec3* angularMotionVelocity)
```

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

The following method is provided to integrate the bodies' final poses:
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 results will be identical between runs if simulating the exact same scene (same actors in the same order) using the same time-stepping scheme and same PhysX release on the same platform. The simulation behavior is not influenced by the number of worker threads that are used.

However, the results of the simulation can change if actors are inserted in a different order. 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 present in the scene or not, irrespective of whether these actors actually interact with the collection of actors. This behavioral property is usually tolerable but there are circumstances in which it is not acceptable.

To overcome this issue, PhysX provides PxSceneFlag::eENABLE_ENHANCED_DETERMINISM, which provides additional levels of determinism. Specifically, provided the application inserts the actors in a deterministic order, with this flag raised, the simulation of an island will be identical regardless of any 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 PxRigidDynamicLockFlag. For example, the below code snippet demonstrates how to restrict a PxRigidDynamic to two-dimensional simulation. In this case, we permit the PxRigidDynamic to rotate only around the Z-axis and to translate only along the X- and Y-axes:

```cpp
PxRigidDynamic* dyn = physics.createRigidDynamic(PxTransform(PxVec3(...

// Lock the motion
dyn->setRigidDynamicLockFlags(PxRigidDynamicLockFlag::eLOCK_LINEAR_Z);
```

It is legal to restrict movement or rotation around any combination of degrees of freedom.
Callback Sequence

The simplest type of simulation callbacks are the events. Using callbacks, an application can simply listen for events and react as required, provided the callbacks obey the rule that SDK state changes are forbidden. This restriction may be a bit surprising given that the SDK permits writes to an inactive back-buffer while the simulation is running. Callbacks, however, are not called from within the simulation thread, but from fetchResults(). The key point here is that fetchResults() processes the buffered writes, meaning that writing to the SDK from an event callback can be a particularly fragile affair. 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. More specifically, this means that properties of each object's internal simulation state are copied to the API-visible state. 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. will still be in the state they were in immediately before the simulation started. This is preferable, because these events were detected early on during the simulation, before objects were integrated (moved) forward. For example, a pair of shapes that get an onContact() callback are in contact will still be in contact when the call is made, even though they may have 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 implemented 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. To avoid the expense of reporting all sleep and wake events, actors identified as worthy of notification require the flag PxActorFlag::eSEND_SLEEP_NOTIFIES to be raised. To receive onContact and onTrigger events it is necessary to set a flag in the filter shader 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:

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

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

**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 that is 16-byte aligned, but on Windows the system function _aligned_malloc() provides this capability.

**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 memory allocator. If you are instrumenting system memory allocations, you may observe this behavior. To prevent PhysX requesting type names, disable allocation names using the method PxFoundation::setReportAllocationNames().

Minimizing dynamic allocation is an important aspect of performance optimization and provides several mechanisms to control and analyze memory usage as discussed in turn.

**Scene Limits**
The number of allocations for tracking objects can be minimized by resizing the capacities of scene data structures, using either PxSceneDesc::limits before creating the scene or the function PxScene::setLimits(). It is useful to note that these limits do not represent hard limits, meaning that PhysX will automatically perform further 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, each 16K in size. The initial number of blocks allocated to the pool can be controlled by setting 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 in the pool, it will start dropping contacts and joint constraints. When this happens, warnings are passed to the error stream in the PX_CHECKED configuration.

To help tune nbContactDataBlocks and maxNbContactDataBlocks it can be useful 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 this function is called any allocated blocks not required by the current scene state will be deleted so that they may be reused by the application. Additionally, a number of other memory resources are freed by shrinking them to the minimum size required by the scene configuration.

**Scratch Buffer**

A scratch memory block may be passed as a function argument to PxScene::simulate. As far as possible, PhysX will internally allocate temporary buffers from the scratch memory block, thereby reducing the need to perform temporary allocations from PxAllocatorCallback. The block may be reused by the application.
PxScene::fetchResults() call, which marks the end of simulation. One restriction on the scratch memory block is that it must be a multiple of 16K, and it must be 16-byte aligned.

**In-place Serialization**

PhysX objects can be stored in memory owned by the application using PhysX’s binary deserialization mechanism. See *Serialization* for details.

**PVD Integration**

Detailed information about memory allocation can be recorded and displayed in the PhysX Visual Debugger. This memory profiling feature can be configured by setting the trackOutstandingAllocations flag when calling PxCreatePhysics(), and raising the flag PxVisualDebuggerConnectionFlag::eMEMORY when connecting to the debugger using PxVisualDebuggerExt::createConnection().
Completion Tasks

A completion task is a task that executes immediately after PxScene::simulate has exited. If PhysX has been configured to use worker threads then PxScene::simulate will start simulation tasks on the worker threads and will likely exit before the worker threads have completed the work necessary to complete the scene update. As a consequence, a typical completion task would first need to call PxScene::fetchResults(true) to ensure that fetchResults blocks until all worker threads started during simulate() have completed 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.
```

The completion task is specified as a function argument in PxScene::simulate. More details can be found in the PhysAPI documentation.
Synchronizing with Other Threads

An important consideration for substepping is that simulate() and fetchResults() are classed as write calls on the scene, and it is therefore illegal to read from the scene while those functions are running. For the simulate() function it is important to make the distinction between running and ongoing. In this context, it is illegal to read from a scene before simulate() exits. It is perfectly legal, however, to read from a scene after simulate() has exited but before the worker threads that started during the simulate() 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, unless they are all read calls.
**Substepping**

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

```cpp
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 of the `simulate()` function. To illustrate this, consider the situation where the scene is simulated until the graphics component signals that it has completed updating the render state. Here, the completion task will naturally run after `simulate()` has exited. It will block with `fetchResults(true)` to ensure that it waits until both `simulate()` and `fetchResults()` have completed their sequential work. When the completion task is allowed to proceed its next work item will be to query the graphics component to check if another simulate() step is required or if it can exit. In the case that another `simulate()` step is required it will need to pass a completion task to `simulate()`. A tricky point here is that a completion task cannot submit itself as the next completion task because it would cause an illegal recursion. A solution to this problem might be to have two completion tasks where each stores a reference to the other. Each completion task can then pass its partner to `simulate()`:  

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

As an alternative to simulate(), you can split the simulation into two
*collide()* and *advance()*. For some properties, called write-through
properties, modifications during the *collide()* phase will be seen immediately by
*advance()* phase. This allows *collide()* to begin before the data required
by *advance()* is available and to run in parallel with game logic that generates inputs to
*advance()* particularly useful for animation logic generating kinematic targets, or
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.
The users can interleave the physics-dependent game logic between *collide()* and
*fetchCollision()*:

```
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
operations performed on the objects in the executing scene will be buffered and will
not be reflected 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 format. The split format offers some advantages over the standard fetchResult() method because it permits the user to parallelize processing of contact reports, which can be expensive when simulating complex scenes.

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

```cpp
// Setup a shared index
int gSharedIndex = 0;
gScene->simulate(1.0f / 60.0f);

// Call fetchResultsStart. Get the set of pair headers
const PxContactPairHeader* pairHeader;
PxF32 nbContactPairs;
gScene->fetchResultsStart(pairHeader, nbContactPairs, true);

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

// process the callbacks
const bool bFinished = gScene->processCallbacks(&callbackFinishTask);
callbackFinishTask.removeReference();
callbackFinishTask.wait();
gScene->fetchResultsFinish();
```

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

This feature is demonstrated in SnippetSplitFetchResults. In order to
approach, contact notification callbacks must be thread-safe. For this approach to be beneficial, contact notification callbacks need to be doing a significant amount of work to benefit from multi-threading them.
Tuning Shape Collision Behavior

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

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

The generation of contact points does not however mean that a
immediately be applied at these locations to separate the shapes, or
further motion in the direction of penetration. This would make the sim
the simulation time step is selected to be tiny, which is not desir
performance. Instead, we want the force at the contact to smoothly
penetration increases until it reaches a value sufficiently high to
penetrating motion. The distance at which this maximum force is
restDistance, because at this distance two shapes stacked on each oth
equilibrium and come to rest. When the shapes are for some reason p
much that they have a distance below restDistance, an even greater force is applied to push them apart until they are at restDistance again. The variation of force as the distance changes is not necessarily linear, but it is smooth and continuous which results in a pleasing simulation even at large time steps.

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

Once the restOffset is determined, the contactOffset should be chosen to be slightly larger. The rule of thumb is to make the difference between the two as small 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 this difference too large is that contacts will be generated sooner as two shapes approach, which can increase the number of contacts that the simulation has to worry about. This can affect performance. Also, the simulation code often makes the assumption that 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 contact behavior to implement sticky contacts, give objects the appearance of swimming inside each other, or making objects go through apparent holes in walls. A 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 contact solver. Because both of these steps occur within the scene simulate() function, a callback must be used.

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

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

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

And then implement the function onContactModify of PxContactModifyCallback:

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

Every pair of shapes comes with an array of contact points, that have a number of properties that can be modified, such as position, contact normal, and separation. For the time being, restitution and friction properties of the contacts cannot be modified. See 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 body

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 contact normal.

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

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

Uniformly scaling a body's inverse mass and inverse inertia by the same scale factor results in the body behaving like a body that is either heavier or lighter depending on the values used. Providing inverse mass/inverse inertia scales < 1 results in the body appearing heavier; providing scales > 1 result in the body appearing lighter. For example, inverse mass/inertia scales of 0.5 result in the body appearing to have double the mass of its original mass. Providing inverse mass/inertia scale of 0 results in the body behaving as if it has infinite mass.

However, it is also possible to non-uniform scale a body's inverse mass and inertia by providing different values to a body's inverse mass and inverse inertia scale. For example, it is possible to reduce or increase the amount of angular velocity change as a result of contacts by adjusting just the inverse inertia scale. The use-cases for this kind of modification are extremely game-dependent but may involve, for example,
interactions between a player's vehicle and traffic vehicles in an arcade-style game, where the player's car is expected to be bumped by traffic vehicles but would be extremely frustrating to the player if the car was to spin-out as a result of the collision. This could also be achieved by making the traffic vehicles much lighter than the player's vehicle but this may make the traffic vehicles appear “too light” and damage the player's immersion.

When performing local mass modification, the impulse reported in PxSimulationEventCallback::onContact() will be relative to the locally scaled masses of the bodies involved in that contact. Therefore, this reported impulse may not accurately reflect the change in momentum caused by a given contact. To address this issue, we have provided the following methods in the rigid body extensions to extract the linear and angular impulse and velocity change caused by a contact using local mass modification:

```cpp
static void computeLinearAngularImpulse(const PxRigidBody& body, const PxVec3& point, const PxVec3& impulse, const PxReal invMassScale, const PxReal invInertiaScale, PxVec3& linearImpulse, PxVec3& angularImpulse);

static void computeVelocityDeltaFromImpulse(const PxRigidBody& body, const PxTransform& globalPose, const PxVec3& point, const PxVec3& impulse, const PxReal invMassScale, const PxReal invInertiaScale, PxVec3& deltaAngularVelocity);
```

These methods return separate linear and angular impulse and velocity change values to reflect the fact that the mass and inertia may have been non-uniformly scaled. When local mass modification has been used, it may be necessary to extract separate linear and angular impulses for each contact point, for each body in the pair. Please note that these helper functions are provided to provide users with accurate impulse values and are not mandatory. For simple use-cases, e.g. triggering effects or damage based on impulse thresholds, the single impulse value reported by the contact report should be perfectly acceptable even when local mass modification has been used and the impulse values are being used for more complex behaviors, e.g. balance control for a ragdoll, then these helper functions will most likely be required to achieve correct behavior. Please note that for articulations, computeLinearAngularImpulse will return the correct impulse.
respective articulation link. However, computeVelocityDeltaFromImpulse will not return the 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 mass modification:

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

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

The contact modification callback can be set using the contactModifyCallback member of PxSceneDesc or the setContactModifyCallback() method of PxScene.
Contact reporting

Here is an example for a contact event function from SampleSubmarine:

```cpp
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()) == mMinesToExplode.end())
                    mMinesToExplode.push_back(mine);
                break;
            }
        }
    }
}
```

SampleSubmarine is a subclass of PxSimulationEventCallback. `onContact` receives the pair for which the requested contact events have been triggered. The function is only interested in eNOTIFY_TOUCH_FOUND events, which are raised whenever two shapes start to touch. In fact it is only interested in touch events of the submarine -- which is checked in the second if-statement. It then goes on to assume that the second actor is a mine (which works in this example because the sample is configured such that no other contact reports will get sent when a submarine actor is involved). After that, it adds the mine to a set of mines that should explode during the next update.
**Note:** By default collisions between kinematic rigid bodies and kinetic 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 impact is larger than a certain threshold. This allows to reduce the amount of reported pairs which need to be processed. To take advantage of this option the following additional configurations 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 through PxRigidDynamic::setContactReportThreshold(). If the body collides with another object and the contact force is above the threshold, a report will be sent (if enabled according to the PxPairFlag setting of the pair). If two or more dynamic bodies both have a force threshold specified then the lower threshold will be used.

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

*eNOTIFY_TOUCH_CCD* has to be raised for the collision pair. *eNOTIFY_TOUCH_CCD* events for the non primary impacts. For performance reasons, the system can not always tell whether the contact pair lost touch in a CCD passes and thus can also not always tell whether the contact persisted. *eNOTIFY_TOUCH_CCD* just reports when the two collision objects detected as being in contact during a CCD pass.
Extracting Contact information

The onContact simulation event permits read-only access to all contact PxContactPair. In previous releases, these were available as a PxContactPoint objects. However, PhysX 3.3 introduces a new format, compressed contact stream. The contact information is now compressed into an appropriate format for a given PxContactPair depending on certain properties, e.g., depending on the shapes involved, the properties of the contacts, materials and whether the contacts are modifiable.

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

```cpp
void MySimulationCallback::onContact(const PxContactPairHeader& pairHeader, 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 = cp.extractContacts(contacts, bufferSize);
        for(PxU32 j=0; j < nbContacts; j++)
        {
            PxVec3 point = contacts[j].position;
            PxVec3 impulse = contacts[j].impulse;
            PxU32 internalFaceIndex0 = contacts[j].internalFaceIndex0;
            PxU32 internalFaceIndex1 = contacts[j].internalFaceIndex1;
            //...
        }
    }
}
```

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 without copying their own copy:
```cpp
void MySimulationCallback::onContact(const PxContactPairHeader& pairHeader, const PxContactPair* pairs, PxU32 nbPairs)
{
    for(PxU32 i=0; i < nbPairs; i++)
    {
        const PxContactPair& cp = pairs[i];

        PxContactStreamIterator iter(cp.contactPatchIndex);

        const PxReal* impulses = cp.contactImpulses;

        PxU32 flippedContacts = (cp.flags & PxContactPairFlag::eINTERNAL_CONTACTS_ARE_FLIPPED);
        PxU32 hasImpulses = (cp.flags & PxContactPairFlag::eINTERNAL_HAS_IMPULSES);
        PxU32 nbContacts = 0;

        while(iter.hasNextPatch())
        {
            iter.nextPatch();
            while(iter.hasNextContact())
            {
                iter.nextContact();
                PxVec3 point = iter.getContactPoint();
                PxVec3 impulse = hasImpulses ? dst.normal * impulses : 0;

                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 not only iterate over all of the data but also consider conditions like whether the pair has been flipped or whether impulses have been reported with the pair. However, this approach of iterating over the data in-place may be more efficient because it doesn't require copying data.

## Extra Contact Data
Since pointers to the actors of a contact pair are provided in contact properties can be read directly within the callback. However, the pose and the velocity of an actor usually refer to the time of impact. If for some reasons the velocity response is of interest, then the actor can not provide that information possibly get the actor velocity or the pose at impact if those properties were modified by the user while the simulation was running (in such a case the newly set property values will be returned). Last but not least, if CCD with multiple passes is enabled, 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 pointers in the callback. For these scenarios, the PhysX SDK provides an additional contact stream that can hold all sorts of extra information related to the contact pair. This stream can be requested per pair through the pair flags `PxPairFlags` (see the API documentation for more details). If requested, the extra data stream is available as a member of the `PxContactPairHeader` structure. The stream can be parsed by using the predefined iterator `PxContactPairExtraDataIterator` and a custom parsing code (see the implementation of `PxContactPairExtraDataIterator` for details about the format of the stream).

Example code:

```cpp
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 -...
            PxVec3 linearVelocityActor1 = iter.postSolverVelocity -...
        }
    }
}
```
Continuous Collision Detection

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

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

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

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

   ```cpp
   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:

   ```cpp
   PxRigidBody* body;
   ...
   ```
body->setRigidBodyFlag(PxRigidBodyFlag::eENABLE_CCD, true);

Once enabled, CCD only activates between shapes whose relative speeds sum of their respective CCD velocity thresholds. These velocities are automatically calculated based on the shape's properties and support non-uniform scales.

Contact Notification and Modification

CCD supports the full set of contact notification events that are supported by collision detection. For details on contact notification, see the documentation for Callbacks.

CCD supports contact modification. To listen to these modify callbacks, derive from the class PxCCDContactModifyCallback:

```cpp
class MyCCDContactModification : public PxCCDContactModifyCallback {
    ...
    void onCCDContactModify(PxContactModifyPair* const pairs, PxU32 count);
};
```

And then implement the function onContactModify of PxContactModifyCallback:

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

This onContactModify callback operates using the same semantic 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 included in CCD. However, it is possible to get trigger events from CCD by not flagging as PxShapeFlag::eTRIGGER_SHAPE and instead configuring the filter shader to return the following state for pairs involving trigger shapes:

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

It should be noted that not flagging shapes as PxShapeFlag::eTRIGGER_SHAPE can result in the triggers being more expensive. Therefore, this workaround should be reserved for use only in situations where important trigger events will be missed without CCD.

**Tuning CCD**

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

1. **PxSceneDesc.ccdMaxPasses**: This variable controls the number of CCD passes to perform. This is defaulted to 1, meaning that all objects are attempted to be updated 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::setMinCCDAccurateAdvanceCoefficient(PxReal advanceCoefficient)**: This method allows you to adjust the amount by which the CCD advances objects in a given pass. By default, this value is 0.15, meaning that CCD will advance an object by the 0.15 * ccdThreshold, where ccdThreshold is a value computed per-shape that acts as a lower-bound of the maximum amount of time that could pass before there is a chance that the object could have tunneled. This is how the CCD ensures that it can catch any potential tunnelling or penetration, even if the time step is large.
0.15 improves the fluidity of motion without risking missed collisions. Reducing this value can negatively impact fluidity but will reduce the likelihood of missed collisions at the end of a frame. Increasing this value may increase the likelihood of objects clipping at object. This value should only be set in the range [0,1].

3. Enabling the flag `PxSceneFlag::eDISABLE_CCD_RESWEEP` in the `PxSceneDesc.flags`:
   - Enabling this flag disables CCD resweeps. This can result in missed collisions as the result of ricochets but has the potential to reduce the overhead of the CCD. In general, enabling this advancement mode will ensure that objects will not pass through the static environment but no longer guarantees that dynamic objects with CCD enabled will not pass through each other.

4. `PxRigidBody::setRigidBodyFlag(PxRigidBodyFlag::eENABLE_CCD_FRICTION, true)`: Enabling this flag enables the application of friction forces in the CCD. This is disabled by default. As the CCD operates using only linear motion, enabling friction inside CCD can cause visual artefacts.

**Performance Implications**

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

**Limitations**

The CCD system is a best-effort conservative advancement scheme with a finite 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 noticeable. 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 cause a proportion of simulation time being dropped for that object and any other objects in the island as it (any objects whose bounds overlap the bounds of that object) become noticeably out-of-sync with the rest of the simulation. It is therefore recommended that paper-thin/tiny objects should be avoided if possible.

It is also recommended that you filter away CCD interactions between objects 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 lead to time being dropped unnecessarily. CCD interactions are automatically disabled between links in 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 demonstrated in a snippet ("SnippetRaycastCCD"). Contrary to the built-in CCD algorithm implemented within the PhysX SDK, this cheaper and simpler alternative version is fully implemented outside of the SDK itself.

After the traditional simulate/fetchResults calls, the system performs raycasts from 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 an overlap position. Then next frame, the SDK's contact generation takes over and generates a convincing motion. There are some subtle details not described here, but this is how it works in a nutshell.

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

However the implementation should be able to prevent important objects from leaving the game world, provided the world is watertight. The code is very small, easy to follow or modify, and its performance is often better overall than for the built-in CCD. So it can be 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-robust 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 contact offsets based on object motion and depending on the constraint solver to ensure that objects do not tunnel through each other.

This approach generally works well and, unlike the sweep-based CCD, speculative CCD on kinematic actors. However, there are cases where objects do not pass through each other. As an example, if the constraint solver 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:

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

Unlike the sweep-based CCD, this form of CCD does not require settings to be raised on either the scene or on the pair in the filter shader.

Note that this approach works best with PCM collision detection. It may not work as well if the legacy SAT-based collision detection approach is used.

This feature can work in conjunction with the sweep-based CCD, e.g., if a kinematic 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 using speculative contacts and the CCD-enabled dynamic actors do not also enable speculative 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 manifolds for the potential contacts in one frame, so it lends itself better to stable stacking. This approach is stable for small contact offsets and rest offsets but may not generate the correct contact points when large offsets are used because it approximates the 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 contacts in the subsequent frame if the shapes move relative to each other more than a threshold 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, full manifold 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 with insufficient solver iterations. As this approach is distance-based, it will generate the correct contact points for arbitrary contact offsets/rest offsets.

To enable PCM, set the flag in the PxSceneDescflags:

```cpp
PxSceneDesc sceneDesc;
sceneDesc.flags |= PxSceneFlag::eENABLE_PCM;
```
Joint Basics

A joint constrains the way two actors move relative to one another. A typical use would be to model a door hinge or the shoulder of a character. Joints are implemented in the PhysX extensions library and cover many common scenarios, but if you have 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 snippet.

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

```cpp
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 constraint frame.

One of the actors must be movable, either a `PxRigidDynamic` or a `PxArticulationLink`, the other may be of one of those types, or a `PxRigidStatic`. Use a NULL pointer there 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 constraint frames that will be enforced by the PhysX constraint solver. In this example, the revolute joint constrains the origin points of the two frames to be coincident, and their x-axes to coincide, but allows the two actors to rotate freely relative to one another around this common axis.

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
- a **revolute** joint constrains positions and rotations of the two frames
- an **angle** joint constrains the angle between the x-axes of the two frames
- a **twist** joint constrains the angle between the origins of the two frames
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, but 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 freedom either to move freely or be locked together. It can be used for a wide variety of mechanical and anatomical joints, but is somewhat configurable than the other joint types. This joint is covered in detail below.

All joints are implemented as plugins to the SDK through the PxConstraint class. A number of the properties for each joint are configured using the PxConstraintFlag enumeration.

**Note:** As in the rest of the PhysX API, all joint angles for limits and drive targets 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 and the appropriate scene-level visualization parameters:

```cpp
scene->setVisualizationParameter(PxVisualizationParameter::eJOINT_FRAMES
scene->setVisualizationParameter(PxVisualizationParameter::eJOINT_LIMITS

... 

joint->setConstraintFlag(PxConstraintFlag::eVISUALIZATION)
```

### Force Reporting
The force applied at a joint may be retrieved after simulation with a call to:

```cpp
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 the joint exceeds this threshold, the joint will break. Breaking a joint generates a simulation event (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:

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

A constraint flag records whether a joint is currently broken:

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

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

```cpp
class MySimulationEventCallback
{
    void onConstraintBreak(PxConstraintInfo* constraints, PxU32 count) { 
        for(PxU32 i=0; i<count; i++) 
        {
        
```
Projection

Under stressful conditions, PhysX’ dynamics solver may not be able to enforce the constraints specified by the joint. PhysX provides kinematic projection to bring violated constraints back into alignment even when the solver fails in a physical process and does not preserve momentum or respect collision geometry. 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:

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

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

A constraint with projection enabled can be part of a graph of rigid body constraints. If this graph is acyclic, the algorithm will choose a root connected rigid bodies, 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 for each. Please note that having more than one constraint attached to a fixed anchor (world or static/kinematic rigid body) in a graph does count as a cycle (for example, bodies connected with constraints and both ends attached to world anchors). Constraints fight over the same body or conflicting projection directions. The projection direction will be chosen based on the following priorities (highest to lowest):

- 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 projecting towards this dynamic and at least one one-way projecting constraint)
- world attachment or a rigid static actor without any projecting constraints
- kinematic actor without any projecting constraints
- dynamic actor with or without two-way projecting constraints (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, so requires a *contactDistance* tolerance specifying how far from the configuration may be before the solver tries to enforce it. Note that enforcement always starts before the limit is violated, so the role played by contactDistance is analogous to the role a positive contactDistance value plays in collision detection. A larger contact makes the limit less likely to be violated even at high relative velocity, but 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 enabled:

```c
revolute->setLimit(PxJointAngularLimitPair(-PxPi/4, PxPi/4, 0.1f))
revolute->setRevoluteJointFlag(PxRevoluteJointFlag::eLIMIT_ENABLE)
```

```c
```
Limits may be either *hard* or *soft*. When a hard limit is reached, relative motion will stop dead if the limit is configured with zero restitution, or bounce if the restitution is non-zero. When a soft limit is violated, the solver will pull the joint back towards the limit using a spring specified by the limit's spring and damping parameters. By default, limits are hard and without restitution, so when the joint reaches a limit motion will simply stop. To specify softness for a limit, declare the limit structure and set the spring and damping parameters directly:

```cpp
PxJointAngularLimitPair limitPair(-PxPi/4, PxPi/4, 0.1f));
limitPair.spring = 100.0f;
limitPair.damping = 20.0f;
revolute->setRevoluteJointLimit(limitPair);
revolute->setRevoluteJointFlag(PxRevoluteJointFlag::eLIMIT_ENABLED);
```

**Note:** Limits are not projected.

When using spring limits, the eACCELERATION flag is strongly recommended as it will automatically scale the strength of the spring according to the masses and inertias of objects that the limit is acting upon, and can substantially reduce the amount of tuning required for good, stable behavior.

### Actuation

Some PhysX joints may be actuated by a motor or a spring implicitly integrated by the PhysX solver. While driving simulations with actuated joints is more expensive than simply 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 reported by the solver, nor does it contribute towards exceeding the joint's breakage force.

**Note:** Changing the drive parameters for a joint, or activating or deactivating the drive, 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 eACCELERATION flag is strongly recommended. This flag will automatically scale the strength of the spring according to the masses and inertias of objects that the limit is acting upon, and 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 scale 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 should adjust the inertia scales in accordance with the bodies' inertias as well. Applying mass scales such 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 sets of jointed 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 will not be conserved, the energy of the system may increase, the force reported for the joint may be incorrect, and non-physical tuning of breakage thresholds and force limits may be required.
The fixed joint constrains two objects so that the positions and orientations of their constraint frames are the same.

**Note:** All joints are enforced by the dynamics solver, so although under ideal conditions the objects will maintain their spatial relationship, there may be some drift. A common alternative, which is cheaper to simulate and does not suffer from drift, is to construct a single actor with multiple shapes. However fixed joints are useful, for example, when a joint must be breakable or report its constraint force.
A spherical joint constrains the origins of the actor's constraint frames to be coincident.

The spherical joint supports a cone limit, which constrains the angle between the X-axes of the two constraint frames. Actor1's X-axis is constrained by a limit on the x-axis of actor0's constraint frame. The allowed limit values are the maximum rotation around the y- and z-axes of that frame. Different values for the y- and z-axes may be specified, in which case the limit takes the form of an elliptical angular cone:

```cpp
  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.
A revolute joint removes all but a single rotational degree of freedom from two objects. The axis along which the two bodies may rotate is specified by the common 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 bodies are closest.

The joint supports a rotational limit with upper and lower extents. The angle between the y- and z-axes of the joint frames are coincident, and increases moving from the y-axis towards the z-axis:

```cpp
joint->setLimit(PxJointLimitPair(-PxPi/4, PxPi/4, 0.01f));
joint->setRevoluteJointFlag(PxRevoluteJointFlag::eLIMIT_ENABLED, true);
```

The joint also supports a motor which drives the relative angular velocity of the two objects towards a user-specified target velocity. The magnitude of the force applied by the motor may be limited to a specified maximum:

```cpp
joint->setDriveVelocity(10.0f);
joint->setRevoluteJointFlag(PxRevoluteJointFlag::eDRIVE_ENABLED, true);
```
By default, when the angular velocity at the joint exceeds the target velocity, the motor acts as a brake; a freespin flag disables this braking behavior.

The drive force limit for a revolute joint may be interpreted either as a force or an impulse, depending on the value of PxConstraintFlag::eDRIVE_LIMITS_ARE_FORCES.
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 between 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, enabled 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 an implicit spring, for which spring and damping parameters may be specified.
**D6 Joint**

The D6 joint is by far the most complex of the standard PhysX joints. In its default state, it behaves like a fixed joint—that is, it rigidly fixes the constraint frames of its two actors. However, individual degrees of freedom may be unlocked to permit a combination of 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:

```cpp
d6joint->setMotion(PxD6Axis::eX, PxD6Motion::eFREE);
```

Unlocking translational degrees of freedom allows the origin point of one actor's constraint frame to move along a subset of the axes defined by the other actor's frame. 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) and `swing` (around the y- and z-axes). Different effects are achieved by unlocking various combinations of twist and swing.

- If just a single degree of angular freedom is unlocked, the result is equivalent to a revolute joint. It is recommended that if just one angular freedom is unlocked, it should be the twist degree, because the joint has various configuration and optimization options that are designed for this case.

- If both swing degrees of freedom are unlocked but the twist degree remains locked, the result is a zero-twist joint. The x-axis of actor1 swings freely away from the x-axis of actor0 but twists to minimize the rotation required to align the two frames. This creates a kind of isotropic universal joint which avoids the problems of the usual 'engineering style' universal joint (see below) that is sometimes used as a twist 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 kept locked, a zero-swing joint results (often also called a uni example the SWING1 (y-axis rotation) is unlocked, the x-axis of actor to remain orthogonal to the z-axis of actor0. In character applications, used to model an elbow swing joint incorporating the twist freedom or a knee swing joint incorporating the twist freedom of the low applications, these joints can be used as 'steered wheel' joints in which is the wheel, free to rotate about its twist axis, while the free swing axis acts as the steering axis. Care must be taken with this combination anisotropic behavior and singularities (beware the dreaded gimbal π/2 radians (90 degrees), making the zero-twist joint a better behaved most use cases.

- if all three angular degrees are unlocked, the result is equivalent to

Three of the joints from PhysX 2 that have been removed from implemented as follows:

- The cylindrical joint (with axis along the common x-axis of the two is given by the combination:

```plaintext
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:

```plaintext
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 given by the combination:

```cpp
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 or three degrees of freedom are locked.

### Limits

Instead of specifying that an axis is free or locked, it may also be specified as 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 origins of the frames when projected onto these axes. For example, the combination:

```cpp
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 actor0's constraint frame.

The twist degree of freedom is limited by a pair limit with upper and lower bounds identical to the limit of the revolute joint.

If both swing degrees of freedom are limited, a limit cone is generated, identical to the limit of the spherical joint. As with the spherical joint, very small or highly elliptical limit cones 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 \( \pi \) 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:

\[
\text{force} = \text{spring} \times (\text{targetPosition} - \text{position}) + \text{damping} \times (\text{targetVelocity} - \text{velocity})
\]

The drive model may also be configured to generate a proportional acceleration instead of a force, factoring in the masses of the actors to which the joint is attached. 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 this can be an impulse, depending on PxConstraintFlag::eDRIVE_LIMITS_ARE_FORCES)
- acceleration drive flag

The drive attempts to follow the desired position input with the configured stiffness and damping properties. A physical lag due to the inertia of the driven body acting through the drive spring will occur; therefore, sudden step changes will result over multiple time steps. Physical lag can be reduced by stiffening the spring or supplying a velocity target.

With a fixed position input and a zero target velocity, a position drive will spring about the drive position with the specified springing/damping characteristics:

```c
// 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 joint provides two angular drive models - twist and swing and SLERP (Spherical Linear Interpolation).

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 quaternion is used directly. In a twist and swing drive, it is decomposed into separate twist and swing components and each component is interpolated separately. Twist and swing is intuitive in many situations; however, there is a singularity when driven to 180 degrees swing. In addition, the drive will not follow the shortest arc between two orientations. On the other hand, SLERP drive will follow the shortest arc between a pair of angular configurations, 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 frame
- 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
- damping - applied to the drive spring (used to smooth out oscillations about the drive target).
- forceLimit - the maximum torque the drive can apply (note that this can be an impulsive torque, depending on PxConstraintFlag::eDRIVE_LIMITS_ARE_FORCES)
- acceleration drive flag. If this flag is set the acceleration (rather than the force) applied by the drive is proportional to the angle from the target.

Best results will be achieved when the drive target inputs are consistent with joint freedom and limit constraints.

**Note:** if any angular degrees of freedom are locked, the SLERP drive parameters are ignored. If all angular degrees of freedom are unlocked, and parameters are set for 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 ability of the iterative solver to converge. Better convergence can be achieved simply by increasing the attributes of the PxRigidDynamic which controls the solver iteration count. However, joints can also be configured to produce better convergence.

- the solver can have difficulty converging well when a light object is constrained between two heavy objects. Mass ratios of higher than 10 are best avoided in such scenarios.
- when one body is significantly heavier than the other, make 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 obtained when the solver has access to the velocity of motion as well as the change in position.
• if you want a very stiff controller that moves the object to specific positions, consider jointing the object to a kinematic actor and use the `setKinematicTarget` function to move the actor.

• if you want a more springy controller, use a D6 joint with a drive target to set the desired position and orientation, and control the spring parameters to increase stiffness and damping. In general, acceleration drive is much easier to tune than force drive.

When using mass scaling or when constraining bodies with infinite inertia along certain axes, the reduction in degrees of freedom of the rigid bodies combined with inaccuracies in floating point calculation can produce arbitrarily stiff constraint responses trying to correct unnoticeably small errors. This can appear, for example, when attempting to perform 2D-simulation using infinite inertia to suppress velocity out of the plane of simulation. In these cases, set the flag `PxConstraintFlag::eDISABLE_PREPROCESSING` and set the `minResponseThreshold` on the constraint to a small value, e.g., 1e-8. This will result in such stiff constraint rows being ignored when encountered, and 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 SnippetCustomJoint 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 achieve behavior in simulation. This is an advanced topic, and assumes familiarity with the mathematics underlying rigid body simulation. The presentation here assumes that the joint constrains two bodies; the case for a static body is equivalent to an infinite mass whose transform is the identity.

The functions which implement dynamic behavior of joints are PhysX nature to the PxFilterShader (see Collision Filtering). In particular, these functions may execute in parallel and asynchronously, and should not access any state except that passed in as parameters.

To create a custom joint class, define the following:

- the functions which implement the behavior of the constraint. The 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
- 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 process which generates inputs to PhysX' velocity-based constraint solver, and a 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 updates a copy of the joint's constant block (so that the joint's copy may be modified 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 joint projection function.
- the solver updates body velocities and positions.
- if the constraint's ePROJECTION flag is set, the simulation calls the solver preparation function.

The Solver Preparation Function

The solver preparation function for a joint has the following signature:

```c
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 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 array, but when reporting forces it is necessary to choose a point at which 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
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 the actor is static, or a NULL pointer was supplied in constraint creation.
- **bB2w** is the transform of the second body. It is the identity transform if the actor is static, or a NULL pointer was supplied in constraint creation.

The role of the solver preparation function is to populate the buffer or provide the point of application for force reporting, and provide mass scaling properties. The return value is the number of Px1DConstraints generated in the output buffer.

Notice that although the joint parameters (relative pose etc) are typically specified relative to an actor, the solver preparation function works with the transforms of the rigid bodies. The constraint infrastructure (see *Data Management*) maintains consistency when, for example, the application modifies the center of mass of an actor.

Each Px1D constraint constrains one degree of freedom between the bodies. The structure looks like this:

```cpp
struct Px1DConstraint
{
  PxVec3 linear0;
  PxReal geometricError;
  PxVec3 angular0;
  PxReal velocityTarget;

  PxVec3 linear1;
  PxReal minImpulse;
  PxVec3 angular1;
  PxReal maxImpulse;

  union
```
Each Px1DConstraint is either a hard constraint (for example, one axis of a fixed joint) or a soft constraint (for example, a spring). A joint may have a mixture of constraint rows - for example, the actuated joint at a rag doll shoulder often has:

- 3 hard 1D-constraints which prevent the shoulder from separating.
- 3 hard 1D-constraints constraining the angular degrees of freedom within some limits.
- 3 soft constraints simulating resistance to angular motion from muscles.

The constraint is treated as hard unless the Px1DConstraintFlag::eSPRING flag is set.

For both soft and hard constraints, the solver velocity for each row is the quantity:

\[
v = \text{body0vel}.\dot{(\text{lin0}, \text{ang0})} - \text{body1vel}.\dot{(\text{lin1}, \text{ang1})}
\]

### Hard Constraints

For a hard constraint, the solver attempts to generate:

- a set of motion solver velocities vMotion for objects which, when integrated, respect the constraint errors, represented by the equation:
vMotion + (geometricError / timestep) = velocityTarget

- a set of post-simulation solver velocities vNext 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 used limits: restitution and velocity biasing. They are set by the constraint flags and eKEEPBIAS, are mutually exclusive, and restitution takes priority (if restitution is set, biasing is ignored).

Restitution simulates bouncing (off a limit, for example). If the impact vCurrent at the start of simulation exceeds the restitution velocity threshold, the velocity of the constraint will be set to:

\[ \text{restitution} \ast -v_{\text{Current}} \]

and the input velocityTarget field will be ignored. To use Px1DConstraintFlag::eRESTITUTION.

Velocity biasing generates post-simulation velocities to satisfy the same 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 has not yet 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 as an implicit spring. In this case, the motion velocity $v_{Motion}$ and post-simulation velocity $v_{Next}$ are the same. The solver solves the equation:

$$F = \text{stiffness} \times -\text{geometricError} + \text{damping} \times (\text{velocityTarget} - v)$$

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 acceleration springs (PxConstraintFlag::eACCELERATION). With this option, the magnitude of the force in accordance with the response of the bodies is scaled; effectively it implicitly solves the equation:

$$\text{acceleration} = \text{stiffness} \times -\text{geometricError} + \text{damping} \times (\text{velocityTarget} - v)$$

### Force Limits and Reporting

All constraints support limits on the minimum or maximum impulse applied to the simulation. There is a special flag for force limits: eHAS_DRIVE_FORCE_LIMIT. If this flag is set, the force limits will be scaled by the timestep unless PxConstraintFlag::eLIMITS_ARE_FORCES is set for the constraint.

The flag eOUTPUT_FORCE flag on a 1D constraint determines whether the force applied for this row should be included in the constraint force output. The reporting force is also used internally to determine joint breakage. For example, if creating a spherical joint with angular drive that breaks when the stress on the linear part exceeds a threshold, 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 convergence. The solveHint value controls preprocessing for each row:

- if the constraint is a hard equality constraint with unbounded impulses
impulse limits are \(-\text{PX\_MAX\_FLT} \text{ and } \text{PX\_MAX\_FLT}\)
PxConstraintSolveHint::eEQUALITY.

- If one of the force limits is zero and the other unbounded,
PxConstraintSolveHint::eINEQUALITY.
- for all soft constraints, and hard constraints with impulse limits other than the above,
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 behav

The Projection Function

The other behavior that joints may specify for simulation is projection.
positional correction designed to act when the velocity-based solver fails.
function has the following signature:

```c
typedef void (*PxConstraintProject)(
   const void* constantBlock,
   PxTransform& bodyAToWorld,
   PxTransform& bodyBToWorld,
   bool projectToA);
```

It receives the constant block and the two body transforms. It updates
bodyBToWorld transform if the projectToA flag is set, and otherwise
transform. See the implementations in the extensions library for examples
projection functions.

The Constraint Shader Table

After coding the behavior functions, define a structure of type PxConstraintShaderTable,
which holds the pointers to the constraint functions. This structure will
argument to PxPhysics::createConstraint, and is shared by all instances

```c
struct PxConstraintShaderTable
{
   PxConstraintSolverPrep solverPrep;
   PxConstraintProject project;
};
```
The constraint visualizer allows the joint to generate visualization information via the PxConstraintVisualizer interface. The functionality of this interface is somewhat biased towards 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 should inherit from 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:

```cpp
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 joints in the extensions library:

- The prepareData() function requests a pointer to the joint constant block, allowing the joint to update any state caches etc. When the function returns, an internal copy of this data is made, so that the joint may be modified during simulation without race conditions. The function is called at the start of the simulation step after the joint is inserted into the scene, and on a subsequent simulation step if PhysX is informed that the joint’s state has changed. To inform PhysX that the joint’s state has changed, call PxConstraint::markDirty().
- onConstraintRelease() is associated with joint deletion. To delete a joint,

```cpp
};
```

```cpp
};
```
PxConstraint::release() on the constraint. When it is safe to destroy no internal references are being held by currently executing simulation constraint code will call PxConstraint::onConstraintRelease(). This run the destructor and release the joint's memory etc.

- **onComShift()** is called when the application calls setCMassLocalPose() on actors connected by the joint. This is provided because the solver projection functions are defined using the frame of the underlying rigid body joint configuration is typically defined in terms of the actors.

- **onOriginShift()** is called when the application shifts the origin of the joint configuration. This is necessary because some joints may have a NULL actor, signifying that they are attached to the world frame.

- **getExternalReference()** is used by PhysX to report simulation events involving constraints, particularly breakage. The returned pointer is passed to the application in the event callback, along with the typeID which the application can use in order to cast the pointer to the appropriate type. The typeID should be distinct for each custom joint type, and different from any of the values in PxJointConcreteType. If the joint also implements the PxBase interface, use the concrete type value from 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 has a tree-like structure - so there can be no loops or breaks. Their primary use is modelling physically actuated characters. They support higher mass ratios, more accurate drive models, better dynamic stability and a more robust recovery from joint separation compared to standard PhysX joints. However, they are considerably more expensive to simulate.

Although articulations do not directly build on joints, they use very similar configuration mechanisms. In this section we assume familiarity with PhysX joints.
Creating an Articulation

To create an articulation, first create the articulation actor without links:

```cpp
PxArticulation* articulation = physics.createArticulation();
```

Then add links one by one, each time specifying a parent link (NULL for the initial link), and the pose of the new link:

```cpp
PxArticulationLink* link = articulation->createLink(parent, linkPose);
PxrRigidActorExt::createExclusiveShape(*link, linkGeometry, material);
PxrRigidBodyExt::updateMassAndInertia(*link, 1.0f);
```

Articulation links have a restricted subset of the functionality of rigid bodies. They cannot be kinematic, and they do not support damping, velocity clamping, or contact force thresholds. Sleep state and solver iteration counts are properties of the entire articulation rather than the individual links.

Each time a link is created beyond the first, a `PxArticulationJoint` is created between it and its parent. Specify the joint frames for each joint, in exactly the same way as for a `PxJoint`:

```cpp
PxArticulationJoint* joint = link->getInboundJoint();
joint->setParentPose(parentAttachment);
joint->setChildPose(childAttachment);
```

Finally, add the articulation to the scene:

```cpp
scene.addArticulation(articulation);
```
Articulation Joints

The only form of articulation joint currently supported is an anatomical joint, whose properties are similar to D6 joint configured for a typical rag doll. Specifically, the joint is a spherical joint, with angular drive, a twist limit around the child joint frame's x-axis, and an elliptical swing cone limit around the parent joint frame's x-axis. The configuration of these properties is very similar to a D6 or spherical joint, but the options provided are slightly different.

The swing limit is a hard elliptical cone limit which does not support spring or restitution from movement perpendicular to the limit surface. You can set the limit as follows:

```cpp
joint->setSwingLimit(yAngle, zAngle);
```

for the limit angles around y and z. Unlike the PxJoint cone limit the limit provides a tangential spring to limit movement of the axis along the limit surface. Once configured, enable the swing limit:

```cpp
joint->setSwingLimitEnabled(true);
```

The twist limit allows configuration of upper and lower angles:

```cpp
joint->setTwistLimit(lower, upper);
```

and again you must explicitly enable it:

```cpp
joint->setTwistLimitEnabled(true);
```

As usual with joint limits, it is good practice to use a sufficient limit coefficient so that the solver will start to enforce the limit before the limit threshold is exceeded.

Articulation joints are not breakable, and it is not possible to retrieve the constraint force applied at the joint.
Driving an Articulation

Articulations are driven through joint acceleration springs. You can set a target, an angular velocity target, and spring and damping parameters to drive the joint towards the target. You can also set compliance values to indicate how strongly a joint resists acceleration. A compliance near zero indicates very strong resistance, and a compliance of 1 indicates no resistance.

Articulations are driven in two phases. First the joint spring forces are applied (we use the term *internal* forces for these) and then any *external* forces such as gravity and contact forces. You may supply different compliance values at each joint for each phase.

Note that with joint acceleration springs, the required strength of the spring is estimated using just the mass of the two bodies connected by the joint. By contrast, articulation drive springs account for the masses of all the bodies in the articulation, and any actuation at other joints. This estimation is an iterative process, controlled using the `externalDriveIterations` and `internalDriveIterations` properties of the PxArticulation class.

Instead of setting the target quaternion for the joint drive, it is possible to set the orientation error term directly as a rotation vector. The value is set as the imaginary part of the target quaternion, with the real part set to 0.

```cpp
ejoint->setDriveType(PxArticulationJointDriveType::eERROR);
ejoint->setTargetOrientation(PxQuat(error.x, error.y, error.z, 0));
```

This allows the spring to be driven with a larger positional error than can be generated by the difference between 2 quaternions. Obtain the same behavior with quaternions by computing the error from the target quaternion, link frames as follows:

```cpp
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 threshold, the articulation is projected back together automatically. Projection is an iterative process, and the PxArticulation functions `PxArticulation::setSeparationTolerance()` and `PxArticulation::setMaxProjectionIterations()` control when projection occurs 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 articu
to sleep if each individual articulation link fulfills the sleep criteria.
The further away objects move from the origin, the larger the chance to point precision issues. This can cause troubles especially in scenar worlds. To avoid these problems, a straightforward solution seems to towards the origin in certain intervals. However, this is not only cumber be pretty expensive due to the invalidation of cached data and persister 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 vector:

```cpp
PxScene::shiftOrigin(const PxVec3& shift)
```

The positions of all objects in the scene and the corresponding data will be adjusted to reflect the new origin location (basically, the shift vector will get subtracted from all object positions). The intended use pattern for this API is to shift the origin such that object positions move closer towards zero. Please note that it is the user's responsibility to keep track of the summed total origin shift and adjust input/output to/from PhysX accordingly. Even though this method preserves some cached data, it is still an expensive operation and we recommend to use it only in the case where distance related precision issues may arise in areas far from the origin. If extensions modules of PhysX are used like the character controller or vehicle library, it will be necessary to propagate the scene shift to those modules as well. Please refer to the documentation of these modules for details.
Introduction

GPU Rigid Bodies is a new feature introduced in PhysX 3.4. It supports the entire rigid body pipeline feature-set but currently does not support articulations. GPU accelerated rigid bodies can be modified and queried using the exact same API as used to modify and query CPU rigid bodies. GPU rigid bodies can interact with clothing and particles in the same way that CPU rigid bodies can and can easily be used in conjunction with character controllers (CCTs) and vehicles.
Using GPU Rigid Bodies

GPU rigid bodies are no more difficult to use than CPU rigid bodies. GPUs use the exact same API and same classes as CPU rigid bodies. GPU rigid body acceleration 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 Kepler or later compatible GPU. If no compatible device is found, simulation will fall back onto the CPU and corresponding error messages will be provided.

This feature is split into two components: rigid body dynamics and broad phase. Rigid body dynamics 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 CUDA context manager and set the GPU dispatcher on the PxSceneDesc. This is a requirement to make use of GPU-accelerated particles or clothing. A snippet demonstrating how to enable GPU rigid body simulation is provided in the code below:

```cpp
 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 contact generation, 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 broad phase.

Each can be enabled independently so, for example, you may enable with CPU rigid body dynamics, CPU broad phase (either SAP or MBP) 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 of:

- 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, triangle meshes, and heightfields. Any spheres, capsules or planes will have contact generation involving those shapes processed on the CPU, rather than GPU.
- Convex hulls require PxCookingParam::buildGRBData to be set to true to support contact generation on the GPU. If a hull with more than 64 vertices or more than 32 vertices per-face is used, it will be processed on the CPU. If the PxConvexFlag::eGPU_COMPATIBLE flag is used when the convex hull is created, the limits are applied to ensure the resulting hull can be used on GPU.
- Triangle meshes require PxCookingParam::buildGRBData to be set to true to support contact generation on the GPU. If this flag is not set during cooking, the GPU data for the mesh will be absent and any contact generation involving this mesh will be processed on the 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 on the GPU. If eENABLE_PCM is not raised, contact generation will be processed on CPU for all pairs using the non distance-based legacy contact generation.
Irrespective of whether contact generation for a given pair is processed on CPU or GPU, the GPU solver will process all pairs with contacts that request collision in their filter shader.

As mentioned above, GPU rigid bodies currently do not support articulations. If `eENABLE_GPU_DYNAMICS` is enabled on the scene, any attempts to add an articulation to the scene will result in an error message being displayed and the articulation will not be added to the scene.

The GPU rigid body solver provides full support for joints and contacts. Best performance is achieved using D6 joints because D6 joints are natively supported on the GPU, i.e. the full solver pipeline from prep to solve is implemented on the GPU. Other joint types are supported by the GPU solver but their joint shaders are run on the CPU. This will incur some additional host-side performance overhead compared to
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. If insufficient memory is available, the system will discard contacts/constraints/pairs, which means that behavior may be adversely affected.

The following buffers are adjustable in PxSceneDesc::gpuDynamicsConfig:

```cpp
struct PxgDynamicsMemoryConfig
{
    PxU32 constraintBufferCapacity; ///< Capacity of constraint buffer allocated in GPU global memory
    PxU32 contactBufferCapacity;    ///< Capacity of contact buffer allocated in GPU global memory
    PxU32 tempBufferCapacity;       ///< Capacity of temp buffer
    PxU32 contactStreamCapacity;    ///< Capacity of contact stream buffer allocated in pinned host memory.
    PxU32 patchStreamCapacity;      ///< Capacity of the contact patch stream buffer allocated in pinned host memory.
    PxU32 forceStreamCapacity;      ///< Capacity of force buffer
    PxU32 heapCapacity;             ///< Initial capacity of heap capacity
    PxU32 foundLostPairsCapacity;   ///< Capacity of found and lost buffers allocated in GPU global memory.
};

PxgDynamicsMemoryConfig() :
    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 approximately 10,000 rigid bodies.

- `constraintBufferCapacity` defines the total amount of memory that can be occupied by constraints in the solver. If more memory is required, a warning is issued and no 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 will be 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 if insufficient memory is allocated, a warning will be issued and contacts will be dropped.
- **patchStreamCapacity** defines the size of a buffer used to store contact patches in the contact stream. This data is allocated in pinned host memory and if insufficient memory is allocated, a warning will be issued and contacts will be dropped.
- **forceStreamCapacity** defines the size of a buffer used to report applied contact 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. Additional memory will be allocated if more memory is required. The cost of physically 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 the 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 emitted and pairs will be dropped by the broad phase.
Performance Considerations

GPU rigid bodies can provide extremely large performance advantages over CPU rigid bodies in scenes with several thousand active rigid bodies. However, there are some performance considerations to be taken into account.

GPU rigid bodies currently only accelerate contact generation involving convex hulls and boxes (against convex hulls, boxes, triangle meshes and heighfields). If you use other shapes, e.g. capsules or spheres, contact generation involving these shapes will only be processed on CPU.

D6 joints will provide best performance when used with GPU rigid bodies. Other joint types 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 per-face will have their contacts processed by the CPU rather than the GPU, so, if possible, keep vertex counts within these limits. Vertex limits can be defined in cooking to ensure that cooked convex hulls do not exceed these limits.

If your application makes heavy use of contact modification, this may limit the number of pairs that have contact generation performed on the GPU.

Modifying the state of actors forces data to be re-synced to the GPU, e.g. 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 still processed on the CPU.
Introduction

This chapter describes how to use PhysX' collision functionality with individual geometry 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 to find the first point of intersection with another geometry object.
- overlaps ("overlap queries") determine whether two geometry objects intersect.
- penetration depth computations ("minimal translational distance queries", abbreviated here to "MTD") test two overlapping geometry objects to find the direction along which they can be separated by the minimum distance.

In addition, PhysX provides helpers to compute the AABB of a geometry object, to compute the distance between a point and a geometry object.

In all of the following functions, a geometry object is defined by its shape (a structure) and its pose (a PxTransform structure). All transforms are interpreted as being in the same space, and the results are also returned...
Raycasts

A raycast query traces a point along a line segment until it hits a geometry object. PhysX supports raycasts for all geometry types.

The following code illustrates how to use a raycast query:

```cpp
PxRaycastHit hitInfo;
PxU32 maxHits = 1;
PxHitFlags hitFlags = PxHitFlag::ePOSITION|PxHitFlag::eNORMAL|PxHitFlag::eNORMAL
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 must be in the \([0, \infty)\) range. If the maximum distance is 0, a hit will only be returned if the ray starts inside a 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 queue processing the query.
• **maxHits** is the maximum number of hits to return.
• **hitInfo** specifies the **PxRaycastHit** structure(s) into which the raycast results will be stored.
• The **anyHit** parameter is **deprecated**. It is equivalent to **PxHitFlag::eMESH_ANY** 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:

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PxRigidActor</strong></td>
<td>actor</td>
</tr>
<tr>
<td><strong>PxShape</strong></td>
<td>shape</td>
</tr>
<tr>
<td><strong>PxVec3</strong></td>
<td>position</td>
</tr>
<tr>
<td><strong>PxVec3</strong></td>
<td>normal</td>
</tr>
<tr>
<td><strong>PxF32</strong></td>
<td>distance</td>
</tr>
<tr>
<td><strong>PxHitFlags</strong></td>
<td>flags</td>
</tr>
<tr>
<td><strong>PxU32</strong></td>
<td>faceIndex</td>
</tr>
<tr>
<td><strong>PxF32</strong></td>
<td>u, v</td>
</tr>
</tbody>
</table>

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** flag in the output. If the input flag is not set for a specific member, the result structure may or may not contain valid data for that member. Omitting the corresponding flags in the input can sometimes result in faster queries.

For a raycast which is not initially intersecting the geometry object, the fields are as follows (optional fields are listed together with the flag that controls their inclusion):

• **actor** and **shape** are not filled (these fields are used only in scene-level raycasts).
- **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 at which the intersection was found.
- **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 polygon index. For other shapes it is always set to 0xffffffff.
- **u** and **v** (*PxHitFlag::eUV*) are the barycentric coordinates of the intersection. 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, where **v0**, **v1** and **v2** are the vertices from the hit triangle:

\[
\text{position} = (1 - u - v) \times v0 + u \times v1 + v \times v2;
\]

This mapping is implemented in *PxTriangle::pointFromUV()*.

See **Geometry** for details of how to retrieve face and vertex data from triangle meshes and height fields using face and vertex indices.

Exceptions to the above behavior may apply if a ray starts inside an object. In these cases, the field will remain unmodified and the corresponding flag will not be set. Specific 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 origin is inside a solid object:

- the reported hit distance is set to zero, and the *PxHitFlag::eDISTANCE* flag is not set.
the output.

- the hit normal is set to be the opposite of the ray's direction, and the \texttt{PxHitFlag::eNORMAL} flag is set in the output.
- the hit impact position is set to the ray's origin and the \texttt{PxHitFlag::ePOSITION} set in the output.

If the start or end point of a ray is very close to the surface of the object, it may be treated as being on either side of the surface.

### Raycasts against Planes

For raycasts, a plane is treated as an infinite single-sided quad that includes its boundary (note that this is not the same as for overlaps). At most one result is returned, and 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 treated as being on either side of the plane.

### Raycasts against Triangle Meshes

Triangle meshes are treated as thin triangle surfaces rather than solid objects. They can be configured to return either an arbitrary hit, the closest hit, or multiple hits.

- if \texttt{maxHits} is 1 and \texttt{PxHitFlag::eMESH\_ANY} is not set, the query will return the closest intersection.
- if \texttt{maxHits} is 1 and \texttt{PxHitFlag::eMESH\_ANY} is set, the query will return an arbitrary intersection. Use this when it is sufficient to know whether or not the ray intersects the mesh, e.g. for line-of-sight queries or shadow rays.
- if \texttt{maxHits} is greater than 1, the query will return multiple intersections. If more than \texttt{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 triangles, or where special filtering is required. Note that \texttt{PxHitFlag::eMESH\_MULTIPLE} is
be used in this case.

In general "any hit" queries are faster than "closest hit" queries, and "
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
will have a negative dot product with the ray direction. This behavior
the mesh instance's `PxMeshGeometryFlag::eDOUBLE_SIDED` flag
`PxHitFlag::eMESH_BOTH_SIDES` flag:

- if either `PxMeshGeometryFlag::eDOUBLE_SIDED`
  `PxHitFlag::eMESH_BOTH_SIDES` is set, culling is disabled.
- if `PxMeshGeometryFlag::eDOUBLE_SIDED` is set, the reported normal
  for a back face hit.

For example a transparent glass window could be modeled as a double
that a ray would hit either side with the reported normal facing the
direction. A raycast tracing the path of a bullet that may penetrate the
and emerge from the back could use `eMESH_BOTH_SIDES` to find both
facing triangles even when the mesh is single-sided.

The following diagram shows what happens with different flags, fo
intersecting a mesh in several places.
To use \texttt{PxHitFlag::eMESH_BOTH_SIDES} for selected meshes rather than all, set the flag inside the \texttt{PxQueryFilterCallback}.

If the start or end point of a ray is very close to the surface of a triangle, it may be 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, it may be treated as being on the either side of the triangle.

\section*{Raycasts against Heightfields}

- Heightfields are treated the same way as triangle meshes with normals pointing (in object space) in \(+y\) direction when thickness is \(\leq 0\) and \(-y\) direction when thickness is \(> 0\).
- Double-sided heightfields are treated the same way as double sided meshes.
Overlaps

Overlap queries simply check whether two geometry objects overlap. One of the geometries must be a box, sphere, capsule or convex, and the other may be of any type.

The following code illustrates how to use an overlap query:

```cpp
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 behind it is considered part of the volume.
- Triangle meshes are treated as thin triangle surfaces rather than solid objects.
- Heightfields are treated as triangle surface extruded by their thickness. Overlap 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 sometimes referred to as MTD, for *minimum translational distance*, as it is the vector by which translation will separate the shapes). One geometry object must be a box, sphere, capsule or convex mesh, and the other may be of any type.

The following code illustrates how to use a penetration depth query:

```cpp
bool isPenetrating = PxGeometryQuery::computePenetration(direction geom0, pose0, geom1, pose1);
```

The arguments are interpreted as follows:

- *direction* is set to the direction in which the first object should be translated to depenetrate from the second.
- *distance* is set to the distance by which the first object should be translated to depenetrate 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 it sets the direction and depth fields. Translating the first object by the depenetration vector will separate the two objects. If the function returns true, the depth will 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`:

```cpp
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 overlap is detected, `nb` is set to zero. The code will attempt at most `maxIter` iterations but may exit earlier if a depenetration vector is found. Usually `maxIter = 4` gives good results.

These functions only compute an approximate depenetration vector, and the amount of overlap between the geometry object and the mesh/heightfield is small. In particular, an intersection with a triangle will be ignored when the object's center is behind the triangle, and if this holds for all intersecting triangles then no overlap is detected, and 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 impact 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:

```cpp
PxSweepHit hitInfo;
PxFHitFlags 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, and is clamped by SDK code to at most `PX_MAX_SWE` sweep of length 0 is equivalent to an overlap check.
- **geomToSweep** is the geometry to sweep. Supported geometries capsule or convex mesh.
- **poseToSweep** is the initial pose of the geometry to sweep.
- **geomSweptAgainst** is the geometry to sweep against (any geometry can be 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 is returned if an impact is found.
- **inflation** inflates the first geometry with a shell extending outward from the object surface, making any corners rounded. It can be used to ensure a minimum margin of space is kept around the geometry when using sweeps to test whether movement is possible.

As with raycasts, fields will be filled in the output structure if the corresponding flags were set in the input hitFlags. The fields of PxSweepHit are as follows:

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PxRigidActor*</td>
<td>actor;</td>
</tr>
<tr>
<td>PxShape*</td>
<td>shape;</td>
</tr>
<tr>
<td>PxVec3</td>
<td>position;</td>
</tr>
<tr>
<td>PxVec3</td>
<td>normal;</td>
</tr>
<tr>
<td>PxF32</td>
<td>distance;</td>
</tr>
<tr>
<td>PxHitFlags</td>
<td>flags;</td>
</tr>
<tr>
<td>PxU32</td>
<td>faceIndex;</td>
</tr>
</tbody>
</table>
```

- **actor** and **shape** are not filled (these fields are used only in scene queries).
- **position** (**PxHitFlag::ePOSITION**) is the position of the intersecting multiple impact points, such as two boxes meeting face-to-face, Ph
point arbitrarily. More detailed information for meshes or height fields may be obtained using the functions in *PxMeshQuery*.

- **normal** (*PxHitFlag::eNORMAL*) is the surface normal at the point of impact, pointing outwards from the hit object and backwards along the impact normal (in the sense that the dot product between the sweep direction and the impact normal is negative).
- **distance** (*PxHitFlag::eDISTANCE*) is the distance along the sweep direction at which 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 from the hit object, not from the swept object. For triangle mesh and height field intersections, it is a triangle index. For convex mesh intersections, it is a polygon index. For other shapes, it is always set to 0xffffffff. For convex meshes, the face index computation is expensive. The face index computation can be disabled by not providing the scene query hit flag *PxHitFlag::eFACE_INDEX*. If needed, the face index can be computed externally using the function *PxFindFaceIndex* which is part of the PhysX extensions 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, with the notable difference that *eMESH_MULTIPLE* is not supported.

### Initial Overlaps

Similarly to a raycast starting inside an object, a sweep may start with the two geometries 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 checks, and thus no initial overlap is reported if a triangle is culled.

Depending on the value of PxHitFlag::eMTD, PhysX may also calc PxHitFlag::eMTD is not set:

- the distance is set to zero, and the PxHitFlag::eDISTANCE PxSweepHit result structure.
- the normal is set to be the opposite of the sweep d PxHitFlag::eNORMAL flag is set in the PxSweepHit result structure
- the position is undefined, and the PxHitFlag::ePOSITION flag PxSweepHit result structure.
- the faceIndex is a face from the second geometry object. For a he mesh, it is the index of the first overlapping triangle found. For oth 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::eDI in the PxSweepHit result structure.
- the normal is set to the depenetration direction, and the PxHitFlag set in the PxSweepHit result structure.
- the position is a point on the sweep geometry object (i.e. the first g and the PxHitFlag::ePOSITION flag is set in the PxSweepHit result
- the faceIndex is a face from the second geometry object:
  - For triangle meshes and heightfields it is the last penetrated triar 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 an addition, the following restrictions apply:
- `PxHitFlag::eMTD` is incompatible with `PxHitFlag::ePRECISE_SWEEP` and `PxHitFlag::eASSUME_NO_INITIAL_OVERLAP` (see below). Using in conjunction with either of these flags will result in a warning being issued and the flag(s) that are incompatible with `PxHitFlag::eMTD` being ignored.

Testing for initial overlaps sometimes uses a specialized code path which incurs a performance penalty. If it is possible to guarantee that geometry objects are not initially overlapping, the check for overlaps can be suppressed with `PxHitFlag::eASSUME_NO_INITIAL_OVERLAP`. There are some restrictions on the use of this flag (also, see *Pitfalls*).

- Using `PxHitFlag::eASSUME_NO_INITIAL_OVERLAP` flag when the geometries initially overlap produces undefined behavior.
- `PxHitFlag::eASSUME_NO_INITIAL_OVERLAP` in combination with zero sweep 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 whether there is an overlap. If an overlap is detected, they are further culled by whether the centroid is behind the triangle, and if no triangles are found, the direction will be set opposite 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 penetration algorithm is used by default for triangle meshes and height fields, and the MTD result may not provide complete penetration from the mesh in extreme cases. In this case the query should 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 convex mesh is undefined when the eMTD flag is not set.

**Precise Sweeps**
PxHitFlag::ePRECISE_SWEEP enables more accurate sweep code (by default a potentially faster but less accurate solution is used). The ePRECISE_ compatible with the inflation parameter, or with the flag PxHitFlag::eMTI

Sweeps against Height Fields

- Height fields are treated as thin triangle surfaces rather than solid objects.
- Thickness magnitude has no effect on initial overlap detection or point of impact.
- For single-sided height fields the normal of the hit will face in +y local if thickness is < 0 and -y when thickness is > 0.
- Height fields are treated as double sided if either one of eDOUBLE_SIDED 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 have very large size disparities.
- Due to algorithmic differences, a sweep query may detect a different overlapping shapes than an overlap query. In particular, it is not sufficient to perform an overlap check in order to determine the PxHitFlag::eIGNORE_INITIAL_OVERLAP flag. Applications that need consistent overlap/sweep/penetration depth information should use sweep
overlap testing and the \textit{PxHitFlag::eMTD} flag.
The following function computes the distance between a point and a geometry object. Only solid objects (box, sphere, capsule, convex) are supported:

\[
\text{PxReal } \text{dist} = \text{PxGeometryQuery}::\text{pointDistance}(\text{point}, \text{geom}, \text{pose}, \text{closestPoint})
\]

*closestPoint* is an optional output argument which returns the closest point.

The following function computes the axis-aligned bounding box (AABB) of a geometry object, given its pose:
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. Only boxes, spheres and capsules may be tested against meshes or heightfields using these functions.

Mesh Overlaps

The following code illustrates how to process the mesh triangles for a spherical volume:

```cpp
PXU32 triangleIndexBuffer[bufferSize];
PXU32 startIndex = 0;
bool bufferOverflowOccured = false;
PXU32 nbTriangles = PxMeshQuery::findOverlapTriangleMesh(sphereGeom, meshGeom, triangleIndexBuffer, startIndex for (PXU32 i=0; i < nbTriangles; i++)
{
    PXTriangle tri;
PXU32 vertexIndices[3];
PxMeshQuery::getTriangle(meshGeom, meshPose, triangleIndexBuffer...
    // process triangle info
}
```

The `findOverlapTriangleMesh` method is used to extract the indices of the triangles:

- `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 its size.
- `startIndex` is used to restart the query if the buffer size is exceeded.
- `bufferOverflowOccured` is set if more triangles would be returned for
```
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 with the following signature:

```cpp
bool sweep(const PxVec3& unitDir,
            const PxReal distance,
            const PxGeometry& geom,
            const PxTransform& pose,
            PxU32 triangleCount,
            const PxTriangle* triangles,
            PxSweepHit& sweepHit,
            PxHitFlags hitFlags = PxHitFlag::eDEFAULT,
            const PxU32* cachedIndex = NULL,
            const PxReal inflation = 0.0f,
            bool doubleSided = false);
```

The arguments are interpreted as follows:

- `unitDir`, `distance`, `geom` and `pose` function identically to the first four parameters of `PxGeometryQuery::sweep`. `distance` is clamped to `PX_MAX_SWEEP_DISTANCE`.
- `triangleCount` is the number of triangles contained in the buffer against which to 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 or not. This is equivalent to the `PxMeshGeometryFlag::eDOUBLE_SIDED` flag, which suppresses backface culling, and for any hit the returned normal faces opposite to the
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. Convex geometry is not supported.
- the function returns a single hit. Multiple hits (ar PxHitFlag::eMESH_MULTIPLE) are not supported.
- The function always returns the closest hit.
- The only supported flags are PxHitFlag::eDEFAULT, PxHitFlag::eASSUME_NO_INITIAL_OVERLAP, PxHitFlag::ePRECISE_SWEEP, PxHitFlag::eMESH_BOTH_SIDES and PxHitFlag::eMESH_ANY.

The function tests each input triangle in the order they are given. By default, the function will test all triangles and return the closest sweep hit (if a hit has been found). If PxHitFlag::eMESH_ANY is used, the function will return as soon as a hit is found (skipping the remaining untested triangles). This flag can also be used to emulate PxHitFlag::eMESH_MULTIPLE, by calling the function PxHitFlag::eMESH_ANY, using as a starting point the previously returned hit (whose index, between 0 and 'triangleCount', is available in sweepHit.faceIndex).
Applications commonly need to efficiently query volumes in space or trace 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 performed without the need to instantiate a PxScene. It supports insertion, removal and updating of objects defining a bounding box, and raycasts, sweeps, and overlap queries against those bounds.

Spatial index has been marked as deprecated in 3.4 and will be removed in future releases.

SnippetSpatialIndex shows an example of how to use this class.

PxSpatialIndex has no internal locking, and there are special considerations when using it from multiple threads. Query operations (marked const in the interface) must not be issued in parallel with update (non-const) operations, or update operations in other. When issuing query operations in parallel, it is important to call the flush() method to force the updates to be processed immediately. Between a call to flushUpdates() and any subsequent queries, queries may be safely issued in parallel.

A query against a PxSpatialIndex structure will result in a callback for each hit, allowing filtering or precise intersection as desired. The PxGeometryQuery class can be used to perform these intersection tests. Results will typically be in approximately sorted order, and when looking for the closest object in a raycast or sweep query against PxSpatialIndex, a useful optimization is to clip the length of the query inside the callback. For example, in SnippetSpatialIndex:

```
PxAgain onHit(PxSpatialIndexItem& item, PxReal distance, PxReal& shrunkDistance) {
    PX_UNUSED(distance);
    Sphere& s = static_cast<Sphere&>(item);
    PxRaycastHit hitData;
```
// the ray hit the sphere's AABB, now we do a ray-sphere intersection
// 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 shapes are within a numerical tolerance of intersection or impact. To obtain results that are identical when using PxSpatialIndex and when not using a culling hierarchy, the bounding boxes must be slightly padded. PxGeometryQuery::getWorldBounds adds this padding by default.

PxSpatialIndex has the same performance characteristics as the scene query when using the PxPruningStructureType::eDYNAMIC_AABB_TREE option. If the AABBs 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 entirely using the function rebuildFull(). Alternatively, a second tree may be built incrementally in the background over many small steps, using the function rebuildStep() in 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 against actors and attached shapes in the scene. There are three types of queries: raycasts, sweeps, and overlaps, and each can return either a single result, or multiple results. Each query traverses a culling structure containing the scene objects, performs a precise test using the GeometryQuery functions (see *Geometry Queries*), and accumulates the results. Filtering may occur before or after precise testing.

The scene uses two different query structures, one for *PxRigidStatic* actors and the other for *PxRigidBody* actors (*PxRigidDynamic* and *PxArticulationLink*). These structures can be configured to use different culling implementations depending on the 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 along a given ray as follows:

```cpp
PxScene* scene;
PxVec3 origin = ...; // [in] Ray origin
PxVec3 unitDir = ...; // [in] Normalized ray direction
PxReal maxDistance = ...; // [in] Raycast max distance
PxRaycastBuffer hit; // [out] Raycast results

// Raycast against all static & dynamic objects (no filtering)
// The main result from this call is the closest hit, stored in the hit object
bool status = scene->raycast(origin, unitDir, maxDistance, hit);
if (status)
    applyDamage(hit.block.position, hit.block.normal);
```

In this code snippet a PxRaycastBuffer object is used to receive results from the raycast query. A call to raycast() returns true if there was a hit. hit.hadBlock is also set if there was a hit. The distance for raycasts has to be in the [0, inf) 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 in some cases they are not set.

Sweeps

A `PxScene::sweep()` query is geometrically similar to a raycast(): a PxGeometry shape is swept from a specified initial pose in a direction unitDir with specified maximum length, to find the points of impacts of the geometry with scene objects. The maximum distance for sweeps has to be in the [0, inf) 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:

```cpp
PxSweepBuffer hit;  // [out] Sweep results
PxGeometry sweepShape = ...;  // [in] swept shape
PxTransform initialPose = ...;  // [in] initial shape pose (at distance=0)
PxVec3 sweepDirection = ...;  // [in] normalized sweep direction
bool status = scene->sweep(sweepShape, initialPose, sweepDirection);
```

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 specified overlapping objects in the scene. The region is specified as a transformed box, capsule or convex geometry.

A PxOverlapBuffer object is used to receive results from overlap() queries:

```cpp
PxOverlapBuffer hit;  // [out] Overlap results
PxGeometry overlapShape = ...;  // [in] shape to test for overlaps
PxTransform shapePose = ...;  // [in] initial shape pose (at distance=0)

bool status = scene->overlap(overlapShape, shapePose, hit);
```

Overlaps results only include actor/shape and faceIndex since there is no single point of intersection.
**Touching and blocking hits**

For queries with multiple results we distinguish between *touching* and blocking. The choice of whether a hit is touching or blocking is made by the user-implemented logic. Intuitively a blocking hit prevents further progress of a raycast or sweep along its path, and a touching hit is recorded but allows the ray or sweep to continue. So a multiple-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". All blocking hits, picks the one with the minimum distance and PxHitBuffer::block member.

- For overlap() queries an arbitrary blocking hit is chosen as the reported hit (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 hint to the 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 block queries. Performance improvement may be a factor of 3 or more, depending on scenario.

To activate this mode use PxQueryFlag::eANY_HIT filter data flag PxQueryFilterData object, for instance:

```cpp
def flags |= PxQueryFlag::eANY_HIT; // note the OR with the default
bool status = scene->raycast(origin, unitDir, maxDistance, hit,
    PxHitFlags(PxHitFlag::eDEFAULT), fdAny
```

Multiple hits

All three query types (raycast, overlap, sweep) can also report multiple hits with objects in the scene.

- To activate this mode for raycasts use the PxRaycastBuffer constructor with user provided buffer for touching hits.
- In this mode all hits default to 'touching' type and are
PxRaycastBuffer::touches array.

For instance:

```cpp
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, st
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 PxOverlaps (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 co 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

```cpp
// 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 provided. The filter callback should return PxQueryHitType::eBLOCK to denote that a hit is blocking. See *Filtering* for details.

For overlap() queries all touching hits will be recorded even if a blocking hit was encountered and PxQueryFlag::eNO_BLOCK flag is set.
Filtering

Filtering controls how shapes are excluded from scene query results and how results are reported. All three query types support the following filtering parameters:

- a `PxQueryFilterData` structure, containing both `PxQueryFlags` and `PxFilterData`
- an optional `PxQueryFilterCallback`

**PxQueryFlag::eSTATIC, PxQueryFlag::eDYNAMIC**

`PxQueryFlag::eSTATIC` and `PxQueryFlag::eDYNAMIC` flags control whether the query should include shapes from the static and/or dynamic query structures. This is the most efficient way to filter out all static/dynamic shapes. For example, an explosion effect that applies forces to all dynamics in a region could use a spherical overlap query with the `PxQueryFlag::eDYNAMIC` flag to exclude all statics since forces cannot be applied to static objects. By default both statics and dynamics are included in query results.

For instance:

```cpp
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 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 structures to find candidates for mid and narrow phases.
- Mid phase traverses the triangle mesh and heightfield internal culling structures, 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() queries, and exact sweep shape tests or overlap tests for sweep() and overlap() queries).

To implement custom filtering in queries, set the `PxQueryFlag::ePREFILTER` or `PxQueryFlag::ePOSTFILTER` flags and subclass `PxQueryFilterCallback` for filtering logic.

- Pre-filtering happens before midphase and narrow phase and allows shapes to be efficiently discarded before the potentially expensive exact collision tests. These tests are more expensive for triangle meshes, heightfields, convexes, and most sweeps than raycast and overlap tests involving only simple shapes (such as spheres, capsules and boxes.)
- Post-filtering happens after the narrow phase test and can therefore use the test results (such as `PxRaycastHit.position`) to determine whether a hit should be discarded or not. These results can be accessed via the `hit` input argument to the post-filtering callback (`PxQueryFilterCallback::postFilter`).
  
  ```cpp
  static_cast<PxRaycastHit&>(hit),
  ```

  access data specific to a raycast query, 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 with non-zero `PxHitCallback::nbTouches` and `PxHitCallback::touches` parameters, all but those that are no further (touchDistance <= blockDistance) than the closest
will be reported. For example, to record all hits from a raycast query:

**Note:** Returning eTOUCH from a filter callback requires the hit buffer to have a non-zero 
::touches array, otherwise PhysX will generate an exception in checked builds and discard any touching hits.

**Note:** eBLOCK should not be returned from user filters for overlap(). Doing so 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 may not be the closest) as a blocking hit. Performance may be more than three times faster, depending on the scenario. Best gains can be expected for long raycasts/sweeps with a nearby 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 to eTOUCH or in cases when no blocking hits are expected (in this case this flag serves as a performance hint.) All hits will then be reported as touching regardless of the filter callback 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 overflowed.

**Note:** this flag overrides the return value from pre and post-filter functions, so hits that were previously returned as blocking will instead be returned as touching.
A fast, fixed-function filter is provided by \texttt{PxFilterData}, a 4*32-bit bitmask in filtering equation. Each shape has a bitmask (set via \texttt{PxShape::setQueryFilterData}) and the query also has a bitmask.

The query data is used differently by batched and unbatched queries. For unbatched queries, the following rules are applied:

- If the query's bitmask is all zeroes, custom filtering and intersection testing proceed as normal.
- Otherwise, if the bitwise-AND value of the query's bitmask and the shape's bitmask is zero, the shape is skipped.

Or in other words:

\begin{verbatim}
PxU32 keep = (query.word0 & object.word0) 
  | (query.word1 & object.word1) 
  | (query.word2 & object.word2) 
  | (query.word3 & object.word3);
\end{verbatim}

This hardcoded equation can provide simple filtering while avoiding the overhead of the filtering callback. For example, to emulate the behavior of active groups, define the groups as follows:

\begin{verbatim}
enum ActiveGroup
{
  GROUP1    = (1<<0),
  GROUP2    = (1<<1),
  GROUP3    = (1<<2),
  GROUP4    = (1<<3),
  ...
};
\end{verbatim}

When shapes are created, they can be assigned to a group, for example:

\begin{verbatim}
PxShape* shape; // Previously created shape
\end{verbatim}
Or to multiple groups, for example GROUP1 and GROUP3:

```cpp
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:

```cpp
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::

// [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 PxRaycastCallback, PxSweepCallback and PxOverlapCallback provide efficient callback based solutions for such scenarios. For instance a raycast query with a PxRaycastCallback will return all touch hits via multiple virtual PxHitCallback::processTouches() callbacks:

```cpp
struct UserCallback : PxRaycastCallback
{
    UserData data;
    virtual PxAgain processTouches(const PxRaycastHit* buffer, PxU32 nbHits)
    // This callback can be issued multiple times and can be used to process an unbounded number of touching hits.
    // Each reported touching hit in buffer is guaranteed to be closer than 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 UserCallback
```

In this code snippet the raycast() query will potentially invoke processTouches multiple times, with all touching hits already clipped to the globally nearest block hit.

- 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. This may simplify multi-threaded implementations.

The batched query feature has been deprecated in PhysX version 3.4.

- `PxBatchQuery` interface facilitates batching and execution of queries together. `PxBatchQuery` buffers raycast, overlap and sweep queries until `PxBatchQuery::execute()` is called.
- Use `PxScene::createBatchQuery(const PxBatchQueryDesc& desc)` to create a `PxBatchQuery` object.
- The hardcoded filtering equation is not used for batched queries. Instead, it is replaced with two filter shaders, respectively running before (`PxBatchQueryPreFilterShader`) and after (`PxBatchQueryPostFilterShader`) the exact per-shape collision test.
- `PxBatchQueryDesc::preFilterShader` and `PxBatchQueryDesc::postFilterShader` are specified separately.
- `BatchQueryFilterData::filterShaderData` will be copied and passed via the `constantBlock` parameter.
- Results are written to user-defined buffers `PxBatchQueryMemory`. These buffers can be changed before each batch query execute call. The SDK 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 by implementing caching for objects within a specified volume and providing `PxScene` for executing raycasts, overlaps, and sweeps. `PxVolumeCache` can provide a performance boost when objects within the same localized region of 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 localized cloud.
- Multiple short range character controller raycasts within the same character.
- Caching query results across multiple frames, the cache can be filled on a larger 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 objects in `PxScene::createVolumeCache()`.

For purposes of multithreaded access, any operation on the cache counts as a read call on the scene.

**Filling the Cache**

To fill the cache, call `PxVolumeCache::fill()`. This will query the scene for overlapping with the volume defined by the geometry and transform and store the results in an internal buffer up to the maximum sizes for static and dynamic objects using `PxBoxGeometry`, `PxSphereGeometry` and `PxCapsuleGeometry` in `cacheVolume`. 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, forEach) will refill the cache automatically using the same volume if the scene query subsystem has been updated since the last fill. The update status is tracked independently for statics and dynamics, so a query might only refill the cache for dynamics while reusing valid cached results for statics. If any attempt to fill or refill fails, the cache is invalid, and any 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 Object Caching is supported for PxVolumeCache queries. Query results are returned via PxVolumeCache::Iterator::shapes() callback, and the query may invoke the callback multiple times to deliver multiple batches of results.

- Raycasts, overlaps and sweeps against a valid cache will return only results that overlap the cache volume, but is guaranteed to return all such volumes.
- Raycasts, overlaps and sweeps against an invalid cache will fall back to scene queries. In this case results may be returned which do not overlap the cache.

Since the cache refills automatically on any query where the scene has changed, these 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. If the query does not lie entirely within the cache volume (and the cache is valid) only those shapes which overlap the cache volume will be returned. If a query is issued against a cache on which fill() has never been called, an error is reported.

The cache also provides a low-level forEach() mechanism that iterates over the cached objects. If forEach() is executed on a cache for which fill() has never been called, it will return without reporting an error. If the cache is invalid, forEach() will return the shapes that overlap the cached volume directly from the scene. This process
allocation of a temporary buffer, and if the allocation fails, \textit{forEach()} message and return.

This code snippet shows how to use \texttt{PxVolumeCache}:

```c++
PxScene* scene;
PxVec3 poi = ...;       // point of interest
PxVec3 origin = ...;    // [in] Ray origin
PxVec3 unitDir = ...;   // [in] Normalized ray direction
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, maxDynamicShapes);

// 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* actorShapePairs)
    {
        for (PxU32 i = 0; i < count; i++)
            doSomething(actorShapePairs[i].actor, actorShapePairs[i].shape);
    }
    iter;

    // invoke UserIterator::shapes() callback for all actor/shape pairs
    cache->forEach(iter);
};
```
Another special case mechanism for accelerating scene queries is single-object caching, using *PxQueryCache*.

- This cache can provide additional speedups and memory savings for *sweep* queries in any operation mode.
- The cache object defines which shape should be tested first. For high temporal coherence, this can provide significant performance gains. To capture that coherence is simply to fill the cache object of a given *eBLOCK* result (last blocking shape) from the previous frame.
- Note that it is likely incorrect to use a past touching hit (recorded with *eTOUCH* flag) for caching since it will be interpreted as blocking and override any

For example, there is a good chance that an AI visibility query will return the same line-of-sight blocking shape for several frames. Using a *raycast* query with a *PxQueryCache* object will allow PhysX to test a single shape - before traversing the internal spatial partitioning structures, and in case of a "cache hit" the traversal can be bypassed entirely. For instance:

```
PxScene* scene;
PxVec3 origin = ...; // [in] Ray origin
PxVec3 unitDir = ...; // [in] Normalized ray direction
PxReal maxDistance = ...; // [in] Raycast max distance
PxRaycastBuffer hit; // [out] Raycast results

// Per-raycast persistent cache, valid from one frame to the next
static PxQueryCache persistentCache;

// Define cache for current frame:
// - if there was a hit in the previous frame, use the cache.
// - otherwise do not (PhysX requires given cache has a valid shape)
const PxQueryCache* cache = persistentCache.shape ? &persistentCache : NULL;

// Perform a raycast query using the cache
const bool status = scene->raycast(origin, unitDir, maxDistance, hit, PxHitFlags(PxHitFlag::eDEFAULT, NULL, cache);
```
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 or 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 collision tests and early out from the traversal.

**Note:** PhysX does not detect stale pointers, so the application is responsible for cached object validity when shapes are deleted.

**Note:** Overlaps do not support single hit blocking caches.
PhysX SDK offers different pruning structures which are used to accelerate scene queries. This paragraph describes the differences between them.

**Generalities**

The Scene Query system uses two different acceleration structures, a hierarchical grid, and an AABB tree.

The grid builds quickly, in $O(n)$ time, with queries executing in between $O(1)$ and $O(N)$ time depending on how uniformly the objects are distributed in space, with pathological 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 run in $O(\log(n))$ time. Queries returning multiple results will traverse more of the tree, thus 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 object positions change, and in pathological cases query performance may degrade to $O(n)$ time.

Acceleration structures must be continually modified in accordance with objects being added or removed, or object AABB updates due to changes in position. To minimize the cost, modifications are deferred for as long as possible. Adding or removing objects or updating AABBs occurs in amortized constant time. Modifications deferred until the changes 'commit'. This happens on the next query or the next `fetchResults()` call. To force an immediate commit, call the `PxScene::flushQueryUpdates()` function.

The exact details of the commit process depend on the values of the `staticStructure` and `dynamicStructure` specified in `PxSceneDesc`.

To avoid automatic resizing triggered by insertions into internal data structures, reserve the space in advance. See `PxSceneDesc::maxNbStaticShapes` 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 objects in this structure.

**PxPruningStructureType::eSTATIC_AABB_TREE**

The acceleration structure is a tree. Committing changes requires a full rebuild. It is generally recommended, but can be a good choice for staticStructure if your scene are created on initialization, and not modified thereafter. If you frequently add or remove static geometry, the default eDYNAMIC_AABB_TREE setting is usually 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 is initially built by the first commit. Once a tree is built, committing changes proceeds as follows:

- **The tree is refitted in accordance with updates and removals of objects it contains.**
- **Added objects are inserted into the grid.**
- **Objects currently in the grid, or changes to AABBs of objects in the grid, cause it to be rebuilt.**

In addition, a new tree is incrementally built during fetchResults(), frames controlled by PxScene’s dynamicTreeRebuildRateHint attribute. When it starts, it includes all of the objects in the current tree and grid. When frames later, 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 can be 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 refitting degrade the quality of the current tree
PxPruningStructure

Provides access to precomputed pruning structure used to accelerate scene queries against newly added actors.

A pruning structure can be provided to PxScene::addActors. The actors shapes will then be directly merged into the scenes AABB tree, without the need for AABB tree recompute:

```cpp
// Create pruning structure from given actors.
PxPruningStructure* ps = PxPhysics::createPruningStructure(&actors);
// 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 with its actors.

For usage of PxPruningStructure please refer to the snippet SnippetPrunerSerialization.

A typical use case for PxPruningStructure is a large world scenario where blocks of closely 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 AABB tree. This might unbalance the tree and it is recommended to recompute the static tree at some point. * eDYNAMIC_AABB_TREE - the pruning structure is merged into a temporary pruning structures until the scene's new optimized AABB tree is computed.
Vehicles
Introduction

PhysX support for vehicles has been significantly reworked in 3.x. NxWheelShape class of 2.8.x, a more optimal integration of the core vehicle simulation code has been developed. More specifically, the vehicle now sits outside the core SDK in a manner similar to PhysXExtensions, allowing vehicles to be updated in a single pass as well as promoting an approach to modeling vehicle data. Vehicles support has been extended from the suspension/wheel/tire modeling of 2.8.x to a more complete model that couples modular vehicle components including engine, clutch, gears, autobox, differential, wheels, tires, suspensions, and chassis. A quick glance at the data structures in PxVehicleComponents.h will provide a flavor of the behaviors supported in vehicles.
Algorithm

The PhysX Vehicle SDK models vehicles as collections of sprung masses, where each sprung mass represents a suspension line with associated wheel and tire data. Collections of sprung masses have a complementary representation as a rigid body actor whose mass, center of mass, and moment of inertia matches exactly the masses and coordinates of the sprung masses. This is illustrated below.

![Vehicle representation as a rigid body actor with shapes for wheels. Note that the wheel rest offsets are specified relative to the center of mass.](image)

Figure 1a: Vehicle representation as a rigid body actor with shapes for wheels. Note that the wheel rest offsets are specified relative to the center of mass.
The relationship between the sprung mass and rigid body vehicle representation can be mathematically formalized with the rigid body center of mass equations:

\[ M = M_1 + M_2 \]

\[ X_{cm} = \frac{(M_1 \times X_1 + M_2 \times X_2)}{(M_1 + M_2)} \]

where \( M_1 \) and \( M_2 \) are the sprung masses; \( X_1 \) and \( X_2 \) are the sprung mass coordinates in actor space; \( M \) is the rigid body mass; and \( X_{cm} \) is the rigid body center of mass offset.

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 and angular velocity. Interaction of the rigid body actor with other scene objects and global pose update is then managed by the PhysX SDK.

The update of each vehicle begins with a raycast for each suspension line. The raycast starts just above the top of the tire at maximum compression and casts downwards 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.
The suspension force from each elongated or compressed spring is computed and added to the aggregate force to be applied to the rigid body. Additionally, the suspension force is used to compute the load that is bearing down on the tire. This load is then used to compute the tire forces that will be generated in the contact plane and then added to the aggregate force to be applied to the rigid body. The tire force computation actually depends on a number of factors including steer angle, camber angle, friction, wheel rotation speed, and rigid body momentum. The aggregated force of all tire and suspension forces is then applied to the rigid body actor associated with the vehicle so that the transform may be 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 at both sides 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 motion is purely rotational and 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 effective 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

Figure 2: Suspension limits and suspension raycasts
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 in the Guide.
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 calling the following function:

```c
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 needs to be specified. A PxSerializationRegistry instance can be created with 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:

```c
void PxVehicleSetBasisVectors(const PxVec3& up, const PxVec3& forward);
```

This function can be called at any time prior to the first execution of PxVehicleUpdates.

The rigid body actors associated with vehicles can be updated either with velocity modifications or updated with an acceleration that is applied in the next PhysX SDK simulate call. The following function can be used to select the required update mode:

```c
void PxVehicleSetUpdateMode(PxVehicleUpdateMode::Enum vehicleUpdateMode);
```

As expected, the vehicle SDK also has a shutdown process which needs:

```c
PX_C_EXPORT void PX_CALL_CONV PxCloseVehicleSDK(
  PxSerializationRegistry* serializationRegistry = NULL);
```

This needs to be called before the PxPhysics instance and PxFoundation are destroyed.
released; that is, the order of shutdown is the reverse of the initialization order. If serialization is required the PxSerializationRegistry specified for PxInitVehicleSDK needs to be passed to PxCloseVehicleSDK. If vehicle serialization is used called before closing the PhysXExtensions.

As an illustration of the usage of these functions, SnippetVehicle4W initialization code:

```cpp
PxInitVehicleSDK(*gPhysics);
PxVehicleSetBasisVectors(PxVec3(0,1,0), PxVec3(0,0,1));
PxVehicleSetUpdateMode(PxVehicleUpdateMode::eVELOCITY_CHANGE);
```

The shutdown code in SnippetVehicle4W is as follows:

```cpp
PxCloseVehicleSDK();
```

## Introduction To Vehicle Creation

The following pseudo-code illustrates the basic process of setting up a instance:

```cpp
const PxU32 numWheels = 4;
PxVehicleWheelsSimData* wheelsSimData = PxVehicleWheelsSimData::a
setupWheelsSimulationData(wheelsSimData);
PxVehicleDriveSimData4W driveSimData;
setupDriveSimData(driveSimData);
PxRigidDynamic* vehActor = myPhysics.createRigidDynamic(startPose
setupVehicleActor(vehActor);
myScene.addActor(*vehActor);
PxVehicleDrive4W* vehDrive4W = PxVehicleDrive4W::allocate(numWheels
vehDrive4W->setup(physics, veh4WActor, *wheelsSimData, driveSimDa
wheelsSimData->free();
```

The code above first instantiates a PxVehicleWheelsSimData 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 includes fields such as engine peak torque, clutch strength, gearing ratios, and Ackermann correction. Following this, a PxRigidDynamicActor is instantiated and geometry for the wheels and chassis as well as dynamic properties such as mass, moment of inertia, and center of mass. The final step is to instantiate an 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 geometry for each tire. A more detailed discussion of PhysX batched scene queries can be found in Section Batched queries.

The following pseudo-code initializes a batched scene query with buffers large enough for a single vehicle with four wheels:

```cpp
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 phase and then reused each frame. It is possible to instantiate a PxBatchQuery instance for each vehicle or to instantiate a single PxBatchQuery instance with buffers large enough for the wheels of a batched array of vehicles. The only restriction is that all batched vehicle arrays and associated buffers configured at the start of a vehicle simulation frame must persist until the end of the vehicle simulation frame.
PhysX vehicles make use of scene query filter shaders to eliminate interacting with any geometry that is not to be considered as a drivable surface. More details for how to set up "myFilterShader" about Section Filtering.

For a batch containing just a single 4-wheeled vehicle the suspension raycasts can be performed with the following pseudo-code:

```cpp
PxVehicleWheels* vehicles[1] = {myVehicle};
PxVehicleSuspensionRaycasts(batchQuery, 1, vehicles, 4, sqResults);
```

The function PxVehicleSuspensionRaycasts performs suspension raycasts for all vehicles in the batched array of vehicles. Each element in the sqResults array stores a raycast report for a single suspension. Pointers to contiguous blocks will be stored by each vehicle in turn as the function iterates through the vehicles array. Memory blocks are stored by each vehicle so that they may easily query the raycast results in PxVehicleUpdates. As a consequence, the sqResults array must persist until at least the end of PxVehicleUpdates and must have length at least the total number of wheels in the vehicles array.

The vehicles are updated with the following function call:

```cpp
PxVehicleUpdates(timestep, gravity, frictionPairs, 1, vehicles, NULL);
```

The function PxVehicleUpdates updates the internal dynamics of each wheel shapes of the vehicle's actor and applies either velocity or acceleration to the actor, depending on the update mode chosen with PxVehicleSetUpdateMode. More details can be found in Section Wheel Pose and Section Vehicle Update. FrictionPairs is basically a lookup table that associates unique friction values with combinations of tire type and PxMaterial. The idea here is to allow tuning for each surface type. This shall be discussed in more depth in Section Drivable Surfaces.
Snippets

Four snippets are currently implemented to illustrate the operation of the PhysX Vehicles SDK. These are:

1. SnippetVehicle4W
2. SnippetVehicleTank
3. SnippetNoDrive
4. SnippetVehicleScale
5. SnippetVehicleMultiThreading

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 it to perform a number of choreographed maneuvers such as accelerate, reverse, brake, handbrake, and turn.

SnippetVehicleTank

SnippetVehicleTank demonstrates how to instantiate and update PxVehicleDriveTank. It creates a tank on a plane and then controls it to perform a number of choreographed maneuvers such as accelerate, soft turns, and hard turns.

SnippetVehicleNoDrive

SnippetVehicleNoDrive demonstrates how to instantiate and update PxVehicleNoDrive. It creates a vehicle on a plane and then controls it to perform a number of choreographed manoeuvres such as accelerate, reverse, soft turns, 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.
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 configuration. Each of these phases is discussed in turn.

setupWheelsSimulationData

The following code, taken from SnippetVehicle4W, instantiates a PxVehicleWheelsSimData:

```cpp
void setupWheelsSimulationData(const PxF32 wheelMass, const PxF32 wheelRadius, const PxF32 wheelWidth, const PxU32 numWheels
const PxVec3* wheelCenterActorOffsets, const PxVec3& chassisCMOffset, const PxF32 chassisMass, PxVehicleWheelsSimData* wheelsSimData
{
    //Set up the wheels.
    PxVehicleWheelData wheels[2];
    {
        //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[PxVehicleDrive4WWheelOrder::eREAR_LEFT].mMaxHandBrake =
        wheels[PxVehicleDrive4WWheelOrder::eREAR_RIGHT].mMaxHandBrake;

        //Enable steering for the front wheels only.
        wheels[PxVehicleDrive4WWheelOrder::eFRONT_LEFT].mMaxSteer =
        wheels[PxVehicleDrive4WWheelOrder::eFRONT_RIGHT].mMaxSteer;
    }
}```
//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 = -camberAngleAtRest;
        suspensions[i + 0].mCamberAtMaxDroop = camberAngleAtMaxDroop;
        suspensions[i + 1].mCamberAtMaxDroop = -camberAngleAtMaxDroop;
        suspensions[i + 0].mCamberAtMaxCompression = camberAngleAtMaxCompression;
        suspensions[i + 1].mCamberAtMaxCompression = -camberAngleAtMaxCompression;
    }
}

//Set up the wheel geometry.
PxVec3 suspTravelDirections[PX_MAX_NB_WHEELS];
PxVec3 wheelCentreCMOffsets[PX_MAX_NB_WHEELS];
PxVec3 suspForceAppCMOffsets[PIX_MAX_NB_WHEELS];
PxVec3 tireForceAppCMOffsets[PIX_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].z);

        // Tire force application point 0.3 metres below rigid body center of mass.
        tireForceAppCMOffsets[i] =
            PxVec3(wheelCentreCMOffsets[i].x, -0.3f, wheelCentreCMOffsets[i].z);
    }
}

// Set up the filter data of the raycast that will be issued by the query.
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 PxVehicleComputeSprungMasses in the frame of the actor because the rigid body center of mass is always specified in the actor's frame. The vehicle suspension system, on the other hand, is specified in the frame of mass frame. As a consequence, the functions setSuspForceAppPointOffset and setTireForceAppPointOffset all describe offsets from the rigid body center of mass. The directness of this approach can make changes to the rigid body center of mass a bit more involved than might be expected. To solve this problem the function PxVehicleUpdateCMassLocalPose has been introduced, though not used in the code above. This function recomputes and sets all suspension offsets, 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 Guide. The function setupNonDrivableSurface, which sets up scene query filter data for each suspension raycast, shall be discussed in more detail in Section the link between TIRE_TYPE_NORMAL and tire friction shall be made clear in Section Tire Friction on Drivable Surfaces. Finally, the use of the function setWheelShapeMapping shall be clarified in Section Wheel Pose.

```
setupDriveSimData

The following code, taken from SnippetVehicle4W, 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);
```
Details of many of the parameters and functions above can be found in the Guide.

Configuring PxVehicleDriveSimDataNW and PxVehicleDriveSimDataTank instances follows a very similar procedure, albeit with slightly different components. More details can be found, for example, in SnippetVehicleTank.

setupVehicleActor

The following code, common to all vehicle snippets, sets up a rigid dynamic actor with geometry, filter and dynamics data:

```cpp
PxRigidDynamic* createVehicleActor
(const PxVehicleChassisData& chassisData,
 PxMaterial** wheelMaterials, PxConvexMesh** wheelConvexMeshes,
 PxMaterial** chassisMaterials, PxConvexMesh** chassisConvexMeshes,
```
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::createExclusive
        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::createExclusive
        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, wheel
chassisQryFilterData shall be discussed in Section Filtering. Further, the
ordering of the wheel shapes in the above code as PxVehicleWheelsSimData::setWheelShapeMapping is clarified in Section

Filtering

In this Section the concepts behind vehicle query and vehicle simulation filtering shall be 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 wheel-shape 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 issuing

Ensuring that wheel shapes don't hit drivable surfaces can be achieved with filtering. This is discussed in more detail in Section Collision Filtering. The vehicle snippets use the following simulation filter shader:

```cpp
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.word1 & filterData1.word0)
        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:

```cpp
PxFilterData wheelSimFilterData;
wheelSimFilterData.word0 = COLLISION_FLAG_WHEEL;
```
Finally, the following simulation filter data is applied to drivable surfaces:

```cpp
PxFilterData simFilterData;
simFilterData.word0 = COLLISION_FLAG_GROUND;
simFilterData.word1 = COLLISION_FLAG_GROUND_AGAINST;
...
shapes[0]->setSimulationFilterData(simFilterData);
```

The combination of collision flags (COLLISION_FLAG_WHEEL_AGAINST etc) and filter shader ensures don't collide with drivable surfaces.

A remarkably similar process may be employed to configure the conquest query filters. This is accomplished in the vehicle snippets with the following:

```cpp
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(filterData1);
    PX_UNUSED(queryFlags);
    return PX_QUERY_HIT_DRIVER;
}
```
Each vehicle wheel is given filter data configured with setupNonDrivableSurface passed to the vehicle with:

```cpp
wheelsSimData->setSceneQueryFilterData(i, qryFilterData);
```

The parameter filterData0 in WheelRaycastPreFilter corresponds to the vehicle snipped shape of the drivable ground plane has scene query filter data configured with the function setupDrivableSurface. This satisfies the requirement that suspension raycasts can hit drivable surfaces. Vehicle shapes, on the other hand, corresponds to the query filter data of a shape potentially hit by the vehicle. The parameter filterData1, corresponds to the query filter data of a shape potentially hit by the raycast. The vehicle snippets the shape of the drivable ground plane has scene query filter data configured with the function setupDrivableSurface. This satisfies the requirement that suspension raycasts can hit drivable surfaces. Vehicle shapes, on the other hand, are configured with setupNonDrivableSurface. This satisfies the restriction that suspension raycasts must not hit the vehicle issuing the raycasts but also prevents vehicles from driving on any other vehicles that might be added to the scene. This extra restriction could readily be avoided by employing a more complex filter shader that perhaps exploits unique IDs encoded in both the shape filter data and the filter data applied to the query itself. Care must be taken, however, to configure the filters to ensure that suspension raycasts only interact with the shapes of other vehicles.

**Note:** It is vital that WheelRaycastPreFilter returns PxQueryHitType::eBLOCK if a raycast hit is allowed for the filter data pair. Using PxQueryHitType::eBLOCK guarantees that each raycast returns either no hits or just the hit closest to the start point of the raycast. This is important because PxVehicleSuspensionRaycasts and PxVehicleUpdates expect a one-to-one correspondence between each wheel element in the PxRaycastQueryResult and PxRaycastHit arrays passed to the batch query.

**Tire Friction on Drivable Surfaces**
In this Section setting up tire types, drivable surface types, or 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 setupWheelsSii type was assigned to each tire:

```cpp
//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 index of the PxMaterial array element that matches the material hit by the raycast hit returns the PxMaterial array element that matches the material hit by the raycast. And the table of friction values is managed by the class PxVehicleDrivableSurfaceToTireFrictionPairs. To make the feature more general each 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 value:

```cpp
PxVehicleDrivableSurfaceToTireFrictionPairs* createFrictionPairs( const PxMaterial* defaultMaterial)
{
  PxVehicleDrivableSurfaceType surfaceTypes[1];
  surfaceTypes[0].mType = SURFACE_TYPE_TARMAC;

  PxMaterial* surfaceMaterials[1];
  surfaceMaterials[0] = defaultMaterial;
```
PxVehicleDrivableSurfaceToTireFrictionPairs* surfaceTirePairs = new PxVehicleDrivableSurfaceToTireFrictionPairs;
PxVehicleDrivableSurfaceToTireFrictionPairs::allocate(MAX_NUM_SURFACE_TYPES, MAX_NUM_TIRE_TYPES);
surfaceTirePairs->setup(MAX_NUM_TIRE_TYPES, MAX_NUM_SURFACE_TYPES, 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, gTireFrictionMultipliers);
    }
}
return surfaceTirePairs;

Note: It is not necessary to provide an exhaustive array of all materials. PxVehicleDrivableSurfaceToTireFrictionPairs has no knowledge of the hit material it assumes a value of zero for the surface type.

There is no upper bound on the friction values used in the PhysX vehicle SDK, the maximum value of friction that obeys the laws of physics is 1.0, the PhysX vehicles SDK purposefully does not enforce this rule. One reason for this is that the model is far from a complete description of a real vehicle, meaning that some liberties need to be taken with friction values to generate the desired behavior. A more complete model could certainly provide greater accuracy given a specific set of vehicle parameters, but it would provide a greater range of editable and controllable behaviors.

A reason that friction is not clamped at 1.0 is that games typically simulate the physics update at 60Hz or KHz update frequencies. One source of numerical accuracy is the amplitude of oscillation of the suspension, which is governed in turn by the distance that the vehicle falls under gravity between each update. At KHz update frequencies this simulation artifact is acceptably small, but not at 60Hz. The last reason is simply no need to impose the strict rules of friction on the vehicles SDKs interesting behaviors to be generated that would perhaps be impossible to achieve under such constraints.
by the laws of rigid body and tire dynamics. Having said all this, however, the model simulated at 60Hz ought to have enough integrity that only small tweaks above 1.0 should be necessary. If very large friction values are required, say greater than 2.0, then it 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 Px 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:

```cpp
void PxVehicleDriveDynData::setAnalogInput(const PxReal analogVal
```

One of the difficulties with vehicle dynamics in games is knowing how to filter the raw controller data in a way that results in pleasing handling. Players, for example, accelerate by pressing very quickly on the accelerator trigger in a way that would never happen in a real car. This rapid acceleration can have a counter-productive effect because the resulting wheel spin reduces the lateral and longitudinal forces that can be generated by the tire. To help overcome some of these problems some optional code has been provided to filter the control data from keyboard and gamepad.

A solution to the problem of filtering controller input data is to assign a rise and fall rate to each button or pad. For analog values under digital control it is possible to increase or decrease the analog value at a specified rate depending on whether the digital input is 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 complication to this simple model is that the difficulty of achieving a large steer angle at large speed must also be modeled. One technique to achieve this would be to model the forces from the aligning moments and apply these to a steering linkage model. This sounds 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 speed.

Rise and fall rates for digital and analog control have been implemented in the helper classes and functions:

```cpp
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
```
A look-up table has also been specified to describe the maximum steer speed:

```c
PXF32 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(gSteerVsForwa
```

Using a PxVehicleDrive4WRawInputData instance it is straightforward inputs in the event a keyboard is used:

```c
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:

```c
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 SDK 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 is controlled by digital inputs then the following function is used:

```cpp
PxVehicleDrive4WSmoothDigitalRawInputsAndSetAnalogInputs(gKeySmoothingData, gSteerVsForwardSpeedTable, carRawInputs, timestep, isInAir, (PxVehicleDrive4W)
```

while gamepad controllers employ the following code:

```cpp
PxVehicleDrive4WSmoothAnalogRawInputsAndSetAnalogInputs(gCarPadSmoothingData, gSteerVsForwardSpeedTable, carRawInputs, timestep, (PxVehicleDrive4W)
```

The code above smoothes the controller inputs and applies them to a PxVehicleDrive4W instance. For other vehicle types the process is remarkably similar with 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 collision with other scene bodies.

In Section *Introduction To Vehicle Update* the functions used to perform vehicle updates were introduced. In this Section these separate update phases will be discussed in more detail.

### Raycast and Update Ordering

Prior to the first time that a vehicle is updated in PxVehicleUpdates, it performed suspension line raycasts at least once with PxVehicleSuspensionRaycasts. Subsequent updates it is not strictly necessary to issue fresh raycasts.
vehicle caches raycast hit planes that can be re-used. It is recommended that there is a 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 a low level of detail is discussed in Section *Level of Detail*.

There is some freedom in the order in which raycasts can be issued relative to the vehicle dynamics update. In a real-world situation it might be that raycasts can be issued 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.

Wheel Pose

PxVehicleUpdates poses the wheels shapes of the vehicle's actor to take account of steer, camber, and rotation angles. The computed pose also attempts to place the wheel geometry exactly on the contact plane identified by the raycast that was issued along the suspension line. To perform this function the PhysX Vehicles SDK needs to know which shapes of the actor correspond to each wheel of the vehicle. This is achieved with the function PxVehicleWheelsSimData::setWheelShapeMapping.

**Note:** The vehicle SDK has a default mapping for each wheel that is equivalent to PxVehicleWheelsSimData::setWheelShapeMapping(i,i). This needs to be corrected if the layout of the shapes is different from the default pattern.

**Note:** PxVehicleWheelsSimData::setWheelShapeMapping(i,-1) can be called to disable setting the local wheel pose. This is particularly useful if a wheel has no corresponding actor geometry.

The wheel pose is always within the limits of PxVehicleSuspensionData::mMaxDroop and PxVehicleSuspensionData::mMaxCompression. If the suspension requires the wheel to be placed beyond the compression limit the wheel will be placed at...
the compression limit and a rigid body constraint will handle the difference in the next SDK simulate() call.

Vehicle State Queries

Each vehicle stores persistent simulation data that is updated each time PxVehicleUpdates is called. Examples of persistent data include wheel rotation angle, and wheel rotation speed. Additionally, a large amount of non-persistent data is computed during each update. This non-persistent data is stored in the vehicle's own data structures. Instead, a data buffer is passed to PxVehicleUpdates and queried after PxVehicleUpdates completes. Examples of non-persistent data include suspension jounce, tire force and raycast hit actor. The combination of these data types allows an almost complete snapshot of the state of the vehicle and can trigger secondary effects such as skid marks, engine and clutch audio, and smoke particles.

Persistent wheel data is stored in PxVehicleWheelsDynData, while persistent drive model data is stored in PxVehicleDriveDynData. The most useful functions are:

```cpp
PX_FORCE_INLINE PxReal PxVehicleDriveDynData::getEngineRotationSpeed()
PxReal PxVehicleWheelsDynData::getWheelRotationSpeed(const PxU32 wheelIdx)
PxReal PxVehicleWheelsDynData::getWheelRotationAngle(const PxU32 wheelIdx)
```

To record non-persistent simulation data so that it may be later be queried, an extra function argument must be passed to PxVehicleUpdates. The following pseudo-code records non-persistent data for a single 4-wheeled car:

```cpp
PxWheelQueryResult wheelQueryResults[4];
PxVehicleWheelQueryResult vehicleWheelQueryResults[1] = {{wheelQueryResults}};
PxVehicleUpdates(timestep, gravity, frictionPairs, 1, vehicles, vehicleWheelQueryResults);
```

Here, a PxVehicleWheelQueryResult array, whose length equals at least the number of vehicles in the batched vehicles array, is passed to PxVehicleUpdates. Each PxVehicleWheelQueryResult instance has a pointer to a PxWheelQueryResult buffer, 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-persistent data. This feature allows memory budgets to be targeted at the vehicles of highest interest.
More Advanced Concepts

Vehicle Telemetry

The purpose of telemetry data is to expose the inner dynamics of the car 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:

```cpp
void PxVehicleUpdateSingleVehicleAndStoreTelemetryData (const PxReal timestep, const PxVec3& gravity, const PxVehicleDrivableSurfaceToTireFrictionPairs& vehicleDrivableSurfaceToTireFrictionPairs, PxVehicleWheels* focusVehicle, PxVehicleWheelQueryResult* ve WheelQueryResult, PxVehicleTelemetryData& telemetryData);
```

The function above is identical to PxVehicleUpdates with the exception that it can only update a single vehicle at a time and takes an extra function argument telemetryData.

Setting up the telemetry data is relatively straightforward. In addition to storing the telemetry data streams, the PxVehicleTelemetryData structure also stores the size, position, and color scheme of the graph. The following pseudo-code initializes and configures telemetry data for a 4-wheeled vehicle:

```cpp
PxVehicleTelemetryData* myTelemetryData = PxVehicleTelemetryData::
const PxF32  graphSizeX=0.25f;
const PxF32  graphSizeY=0.25f;
const PxF32  engineGraphPosX=0.5f;
const PxF32  engineGraphPosY=0.5f;
const PxF32  wheelGraphPosX[4]={0.75f,0.25f,0.75f,0.25f};
const PxF32  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 color coding used to visualize the telemetry data.

In the above example, the coordinates have been configured to render an engine-related graph in the center of the screen under the assumption that (1,1) is the top left-hand side of the screen and (0,0) the bottom right-hand side of the screen. Screen coordinates 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:

```c
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, normalized tire load, wheel rotation speed, tire friction, tire longitudinal slip, tire longitudinal force, tire lateral slip, tire lateral force, and tire aligning moment. Data is also collected for engine revs, engine drive torque, clutch slip, applied acceleration/brake/steer, and gear ratio. For each graph all associated data is collected in separate channels that can be accessed after the update is complete.

Prior to rendering the graph of a particular wheel and channel the following pseudo-code is required:

```cpp
PXFX32 xy[2*PxVehicleGraph::eMAX_NB_SAMPLES];
PxVec3 color[PxVehicleGraph::eMAX_NB_SAMPLES];
char title[PxVehicleGraph::eMAX_NB_TITLE_CHARS];
myTelemetryData->getWheelGraph(wheel).computeGraphChannel(PxVehicleWheelGraphChannel xy, color, title);
```

This code computes a sequence of screen coordinates [x0,y0,x1,y1,x2,y2,....xn,yn] that represent the points of the specified graph engine's graph data. It also stores a color for each sample by choosing between lineColorHigh and lineColorLow depending on the value of the sample. Each channel stores the last 256 samples so that a history of each parameter can be displayed on the screen.

The PhysX Vehicles SDK does not render the graphs. This is an exercise left to the application 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 steps proceed almost exactly as described in Section *Vehicle Update* but with an extra sequential call to the function PxVehiclePostUpdates after all concurrent calls to PxVehicleSuspensionRaycasts and PxVehicleUpdates are complete. PxVehiclePostUpdates performs write operations normally executed in PxVehicleUpdates but which are not possible to call when concurrency is employed.
PxVehicleSuspensionRaycasts is a thread-safe function and can be called concurrently without any modifications to the calling code with the exception, of course, of managing the tasks and threads that will execute the raycasts concurrently. On the other hand, PxVehicleUpdates as used in Section Vehicle Update is not thread-safe and requires an extra PxVehicleConcurrentUpdateData array to be specified for it to be concurrently executed. When this extra data is specified PxVehicleUpdates defers a number of writes to PhysX actors that are involved in the vehicle updates: these writes are stored in the PxVehicleConcurrentUpdateData array during calls to PxVehicleUpdates and then executed sequentially in PxVehiclePostUpdates.

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 shader data that must be set per-wheel:

```cpp
void PxVehicleWheelsDynData::setTireForceShaderFunction(PxVehicleComputeTireForce tireForceShaderFn)
void PxVehicleWheelsDynData::setTireForceShaderData(const PxU32 tireId, const void* tireForceShaderData)
```

The shader function must implement this function prototype:

```cpp
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 recipWheelRadius,
                                         const PxF32 restTireLoad, const PxF32 normalisedTireLoad, const PxF32 gravity,
                                         const PxF32 gravity, const PxF32 recipGravity,
                                         PxF32& wheelTorque, PxF32& tireLongForceMag, PxF32& tireLatForceMag)
```

The vehicle update code will call the shader function for each wheel with shader data 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 and racing cars. PxVehicleDriveNW is very similar to PxVehicleDrive4W except that it allows all wheels to be coupled to the differential. This general differential models of PxVehicleDriveNW cannot match the range or dynamics of PxVehicleDrive4W. PxVehicleDriveTank implements a simple but efficient tank model by constraining the left and right wheel speeds to mimic the effect of tank tracks. 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 skateboards and hovercraft to be implemented using PhysX vehicles.

PxVehicleDrive4W

The class PxVehicleDrive4W has already been discussed in some detail, but the discussion so far has focused on 4-wheeled vehicles. In the following, PxVehicleDrive4W shall be discussed with special reference to instances with less than and more than 4 wheels.

3-Wheeled Cars

Utility functions have been provided to quickly configure 3-wheeled cars, starting with a 4-wheeled car and then disable one of the wheels:

```cpp
void PxVehicle4WEnable3WTadpoleMode(PxVehicleWheelsSimData& wheelsSimData, PxVehicleWheelsDynData& wheelsDynData, PxVehicleDriveSimData4W driveSimData);
void PxVehicle4WEnable3WDeltaMode(PxVehicleWheelsSimData& wheelsSimData, PxVehicleWheelsDynData& wheelsDynData, PxVehicleDriveSimData4W driveSimData);
```

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 disable Ackermann correction, re-position the opposite remaining wheel to the center of the axle, and adjust the suspension of the opposite remaining wheel to...
missing suspension of the disabled wheel. Further wheels could in theo
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 attempting to set the local pose of a PxShape that may no longer be valid. PxVehicleWheelsSimData::setWheelShapeMapping can be used to satisfy this requirement. The second requirement is that any wheel that has been disabled must have zero wheel rotation speed. This can be satisfied through PxVehicleWheelsDynData::setWheelRotationSpeed for the relevant wheel. The final requirement is that disabled wheels must receive no drive torque. This requirement can actually be ignored because it is automatically enforced by code called by the PxVehicleUpdates function. For vehicles of type PxVehicleNoDrive, the requirement on drive torque is fulfilled by ensuring that PxVehicleNoDrive::setDriveTorque 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 the differential is disconnected from the disabled wheel. This is achieved using PxVehicleDifferentialNWData::setDrivenWheel.

Configuring the differential of a PxVehicle4W to ensure that no drive torque is delivered 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 driven. On the other hand, if the wheel has index eFRONT_LEFT or eFRONT_RIGHT then the differential does need to be modified to enforce the requirement. One way to do this is to set up the differential to deliver torque to only the rear(front) wheels if a front(rear) wheel has been disabled. This is achieved using PxVehicleDifferential4WData::setDrivenWheel.

```
PxVehicleDifferential4WData diff = myVehicle.getDiffData();
if(PxVehicleDrive4WWheelOrder::eFRONT_LEFT == wheelToDisable || PxVehicleDrive4WWheelOrder::eREAR_LEFT == wheelToDisable ||
```
In some situations limiting the drove torque to just the front or rear wheels might not be acceptable. If only a single wheel has been disabled then it is possible to engage a mode where 3 wheels are driven. This can be achieved by modifying a differential that delivers torque to all four wheels (eDIFF_TYPE_LS_4WD or eDIFF_TYPE_OPEN_4WD) so that torque is only delivered to 3 wheels:

```cpp
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_REARWD;
        //could also be PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_REARWD;
    }
}
myVehicle.setDiffData(diff);
```
In some situations it will make sense to disable Ackermann steer correction if the disabled wheel was able to steer. In particular, if the remaining wheel of the front or rear axle is repositioned so that it is at the center of the axle then it would almost certainly follow that Ackermann correction would be disabled. 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 construct a PxVehicleDrive4W with more than 4 wheels but with the caveat that only 4 wheels may be driven. As a consequence of this caveat the functionality of the extra wheels is slightly 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 can 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 not preclude the ability to call PxVehicle4WEnable3WTadpoleMode or PxVehicle4WEnable3WDeltaMode. 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 6-wheeled PxVehicleDrive4W vehicle:

```cpp
PxVehicleWheelsSimData* wheelsSimData = PxVehicleWheelsSimData::allocate();
PxVehicleDriveSimData4W* driveSimData = PxVehicleDriveSimData4W::allocate();
setupSimData(wheelsSimData, driveSimData);
PxVehicleDrive4W* car = PxVehicleDrive4W::allocate(6);
PxRigidDynamic* vehActor = createVehicleActor4W();
setup(vehActor, *wheelsSimData, driveSimData);
```

**PxVehicleDriveNW**

While the PxVehicleDrive4W allows cars with any number of wheels simulated it only allows 4 of those wheels to be driven by engine torque. The vehicle type PxVehicleDriveNW has been introduced to address this limitation. This vehicle class makes use of the differential type PxVehicleDifferentialNW, a 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 PxVehicleNW means that only a simple equal-split differential model can be provided.

The following pseudo-code illustrates the key steps in the creation of a 6-wheeled PxVehicleDriveNW vehicle:

```cpp
PxVehicleWheelsSimData* wheelsSimData = PxVehicleWheelsSimData::allocate();
PxVehicleDriveSimDataNW* driveSimData = PxVehicleDriveSimDataNW::allocate();
setupSimData(wheelsSimData, driveSimData);
PxVehicleDriveNW* car = PxVehicleDriveNW::allocate(6);
PxRigidDynamic* vehActor = createVehicleActorNW();
setup(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 all driven through the differential in a way that ensures that all the wheels on the left-hand side have the same speed, and all the wheels on the right-hand have the same speed. A constraint on wheel speed mimics the effect of the caterpillar track structure but avoids the expense of simulating the jointed track structure. Adding the geometry of the caterpillar tracks is as easy as adding an actor shape down each side and setting up collision and query filters as appropriate for the tracks. The motion of the caterpillar tracks could be 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 sets up a 12-wheeled tank:

```cpp
PxVehicleWheelsSimData* wheelsSimData = PxVehicleWheelsSimData::allocate();
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 that orients the front wheels relative to the forward motion of the vehicle. This requires a different 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 directions, while eSTANDARD does not. These two modes result in quite different turning actions.

model eSTANDARD simulates the usual turning action of a tank: pushing forward on the left(right) stick drives the left(right) wheels forward, while pulling back on the right(left) stick applies 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 left(right) wheels backwards. This can result in a turning circle focused at the center of 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 closer backwards compatibility with the interface to the 2.8.x NxWheelShape class. Essentially a rigid body with N suspension/wheel/tire units attached, identical to that of a PxVehicleDrive4W which is permanently in neutral gear so that the engine has no influence on the wheels and the wheels are coupled through the motion of the rigid body. This comes, of course, without the storage overhead of Ackermann steering correction data, engine torque curve data etc. The idea is that users can develop their own drive model on top of already existing vehicle code, suspension raycasts, tire and suspension force computation, and PhysX.

The key functions are the application of per wheel drive and brake torques and steer angles:

```cpp
/**
 \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. Furthermore, 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 braking torque. Inspection of the constructor for PxVehicleWheelData reveals a value of `mMaxBrakeTorque`. This number actually represents a value of 1500 "Kilogram Metres-Squared Per Second-Squared" (an alternative way of expressing this is "1500 Newton Metres"). An important question is how to set up a vehicle with meaningful values if SI units are not adopted. The purpose of this Section is to illustrate the steps required. In particular, the case where distance is measured in centimeters rather than metres will be used as an example. This particular deviation from the adoption of SI Units is probably the most common one in game development, arising from the units of distance in the chosen 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 more complex units involving combinations of other properties such as mass or time or even powers of distance. The former category includes data fields such as wheel radius or maximum suspension droop, while the latter category includes data fields such as maximum braking 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::mAxleSeparation`
- `PxVehicleWheelData::mRadius`
It is useful to note that all the above parameters have default values of zero; they must always be set with measured values in the corresponding length scale if a legal vehicle is to be successfully instantiated.

Setting parameters that involve more complex combinations of length scale require slightly more thought than those featured in the list above. A simple rule of thumb is that any 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 the square of distance 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 \text{ kilograms centimeters-squared per second-squared}. Consequently, when centimeters is used as the length scale a good initial guess for wheel braking torque is 15000000 \text{ [kilograms centimeters-squared per second-squared]}. If inches are used as the length scale then a good initial guess for the wheel braking torque would be 1500\times 39.37 \times 39.37 \approx 2324995.35 \text{ [kilograms inches-squared per second-squared]}. Each non-dimensionless parameter has been described with the corresponding SI Units in PxVehicleComponents.h. The following is an exhaustive list of vehicle parameters that are indirect expressions of distance scale:

\begin{itemize}
  \item \texttt{PxVehicleWheelData::mWidth}
  \item \texttt{PxVehicleSuspensionData::mMaxCompression}
  \item \texttt{PxVehicleSuspensionData::mMaxDroop}
  \item \texttt{PxVehicleWheelsSimData::setSuspForceAppPointOffset}
  \item \texttt{PxVehicleWheelsSimData::setTireForceAppPointOffset}
  \item \texttt{PxVehicleWheelsSimData::setWheelCentreOffset}
  \item \texttt{PxVehicleEngineData::mMOI \text{ (kg m}^2\text{)}}
  \item \texttt{PxVehicleEngineData::mPeakTorque \text{ (kg m}^2 \text{s}^{-2}\text{)}}
  \item \texttt{PxVehicleEngineData::mDampingRateFullThrottle \text{ (kg m}^2 \text{s}^{-1}\text{)}}
\end{itemize}
All but the last three of the above parameters have non-zero initial values in their associated constructors. This means that a good guess for their initial value, found 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 are equivalent to 1 meter.

It is important to note that the wheel handbrake torque has a default value of zero because 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 1.0 and 2.0 to ensure that the handbrake is stronger than the brake.

The wheel moment of inertia and chassis moment of inertia are typically computed from the wheel radius and chassis dimensions so naturally reflect the length scale used 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 remainder of the vehicle data fields or to perform the appropriate unit conversion.

A number of functions also have parameters that are functions of length scale, following is an exhaustive list of such functions:

```cpp
PxVehicleWheelsSimData::setSubStepCount
PxVehicleWheelsSimData::setMinLongSlipDenominator
```
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
centimeters the chosen length scale a value of 500 [centimeters per
passed to achieve the equivalent behavior, or with inches as the cho
value of 5*39.37 (= 196.85) [inches per second] is required. The same
be applied to PxVehicleWheelsSimData::setMinLongSlipDenominator.

4.0 metres per second. If centimeters is the adopted scale then the equ
[centimeters per second], while 4*39.37 (=157.48) [inches per second] i
is the chosen scale. PxVehicleSetMaxHitActorAcceleration takes a
linearly with the length scale. If the desired maximum acceleration
second per second then that would be scaled to 10*100 centimetres
second in centimetres scale. With inches as the length scale the equi
be 10*39.37 inches per second per second.

The PhysX Vehicle SDK supports any system of units with the ca
supplied must conform to the same unit system. Further, the default da
strictly expressed in the SI unit system, can be used as a guide to
values in any unit system for almost any conceivable vehicle. A quick w
be to decide if, say, a truck would have a stronger handbrake than t
family car. Now, the default data approximates that of a standard family
a good estimate to start with the truck having a handbrake that is per
that is, 5000 kilograms metres-squared per second-squared. If centimet
length scale then a quick conversion can be performed by noting that
100 centimeters, leading to the brake torque being set as 5000*:
centimeters-squared per second-squared. If the natural unit of mass
noting that 1 kilogram is 1000 grams leads to an equivalent value of
metres-squared per second-squared. This rule can be repeated for all
by simply noting the default value and the SI units in the relevant cla
then performing the conversion to the chosen unit system.

The PhysX Vehicle SDK depends on a number of threshold values th
length scale. These are set with the function PxInitVehicleSD
PxBTolerancesScale values that have already been already configured for the PhysX SDK.

If PxInitVehicleSDK is not called prior to the first call to PxVehicleUpdates a warning will 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 it is motion is exactly in step with the world geometry. The PhysX vehicle number of options for reducing the computational load for vehicles the levels of detail.

**Extrapolation**

The most obvious strategy for a vehicle that requires only a low level of detail is simply to 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 each and it might be acceptable to avoid these steps completely and let the PhysX SC body with the legacy momentum of the rigid body. After several frames wheels will likely either be hovering above the ground or intersecting the needs to be a strategy to decide how many PhysX SDK updates can a vehicle is once more updated properly by including it in the vehicle array passed to PxVehicleSuspensionRaycasts/PxVehicleUpdates. The details of any such strategy are left to users of the vehicles SDK because it depends on a number of factors such as distance from the camera; the spatial coherence of the world geometry; the speed of the vehicle; and whether the audio or graphics fx for the important role.

**Disable Wheels**

If there exist vehicles with large wheel counts it might also be possible to reduce the number of wheels that participate in the simulation by calling PxVehicleWheelsSimData::disableWheel. An example might be a truck...
Now, such a truck will clearly need to perform 18 raycasts, 18 tire force, 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 computational work is required. It is important to note that when wheels are disabled they no longer participate in supporting the mass of the vehicle’s rigid body. In the extreme case of a 18-wheeled truck reduced to just 4 active wheels this will mean that the remaining enabled suspension springs are only configured to support approximately 4/18 of the mass of the vehicle’s rigid body. To remedy this problem the mass of the rigid body will need to be re-distributed among the enabled wheels and suspensions using 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 wheel counts. The two vehicles can be easily swapped in the vehicles array passed to PxVehicleSuspensionRaycasts/PxVehicleUpdates as the required level of detail increases and decreases. It is worth considering how this might work in the case of the 18-wheeled truck mentioned earlier. The simplest strategy would be to first construct the required rigid body and attach a PxShape instance for each of the 18 wheels of the 18-wheeled version of the truck with PxVehicleNW::create or 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. Many choices are available but the most obvious choice would be front-left/front-right/rear-left/rear-right wheels of the 18-wheeled truck. The 4-wheeled version can then be instantiated using the same rigid body as for the 18-wheeled version of the PxShape instances to the rest pose of the 4-wheeled truck. If the 4-wheeled version have been set up correctly the rest poses ought to be identical to 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 local 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:

```cpp
void PxVehicleComputeSprungMasses(const PxU32 nbSprungMasses, const PxVec3* sprungMassCoordinates, const PxVec3& centreOfMass, const PxU32 gravityDirection, PxReal* sprungMasses);
void PxVehicleWheelsSimData::copy(const PxVehicleWheelsSimData& src, const PxU32 trgWheels);
void PxVehicleSuspensionData::setMassAndPreserveNaturalFrequency(const PxVehicleCopyDynamicsData& wheelMap, const PxVehicleWheels& src, PxVehicleWheels* trg);
```

The following pseudo-code hopefully makes clear how to apply these functions: first construct the lower LOD vehicle and then swap between the different versions:

```cpp
PxVehicleDriveNW* instantiate4WVersion(const PxVehicleDriveNW& vehicle18W) {
    //Compute the sprung masses of the 4-wheeled version.
    PxReal sprungMasses[4];
    {
        const PxReal rigidBodyMass = vehicle18W.getRigidDynamicActor().getMass();
        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 = PxVehicleWheelsSimData;
    for(PxU32 i = 0; i < 4; i++)
    {
        wheelsSimData4W->copy(vehicle18W.mWheelsSimData, i, i);
        PxVehicleSuspensionData suspData = wheelsSimData4W->getSu
```
suspData.setMassAndPreserveNaturalFrequency(sprungMasses[i]);
wheelsSimData4W->setSuspensionData(i, suspData);
}
wheelsSimData4W->setTireLoadFilterData(vehicle18W.mWheelsSimData);

//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.mDriveSimData;
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.getRigidDynamicActor);
PxVehicleDriveNW* vehicle4W =
    PxVehicleDriveNW::create(&physics, rigidDynamic, *wheelsSimData4W);

//Delete the wheels simulation data now that we have copied the data to the instantiated vehicle.
wheelsSimData4W->free();

//Finished.
return vehicle4W;

void swapToLowLodVersion(const PxVehicleDriveNW& vehicle18W, 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;
void swapToHighLowVersion(const PxVehicleDriveNW& vehicle4W, 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 function argument in `PxVehicleSuspensionRaycasts`. An alternative to disabling raycasts would be to alter the array of vehicles passed to `PxVehicleSuspensionRaycasts` so that some vehicles scheduled for update in `PxVehicleUpdates` do not participate in the batched raycast prior to the update. It is anticipated that using the boolean array will allow the same vehicle array to be passed to both the raycast and update functions, thereby allowing simpler vehicle management.

Vehicles that participate in the batched raycast automatically store hit planes 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.
especially if the vehicle is moving slowly or the vehicle is far from the camera or the vehicle remains on the same plane for several updates in a row. As updates preceded by a raycast decreases, the accuracy of the cached hit planes decreases, meaning that the likelihood of visibly poor wheel placement increases. A lack of accuracy in the cached hit planes means that some wheels might end up hovering or intersecting the ground if raycasts are not performed prior to each update. It is left to users of the SDK to develop their own strategy to decide whether a fresh raycast or not.

If a raycast is not performed prior to an update then the vehicle will only report partial description of its interaction with the scene. For example, as a deletion the actor or shape or material hit by the last suspension raycast may exist in the scene several updates later. For this reason, the vehicle reports NULL pointers for the shapes/actors/materials if a cached plane is used instead of the hit planes of a fresh raycast. The documentation for PxWheelQueryResult describes this.

The first update of any vehicle requires that a raycast is performed prior to the 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 setToRestState the vehicle also needs to perform a raycast prior to the next update. This is because setToRestState clears the cached hit planes, meaning that they need to be recomputed 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. 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 fast analytical solution. As a consequence, the vehicle SDK ignores the clutch 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 gained by switching to eESTIMATE because at most only 4 wheels can ever couple to the clutch. The real performance gains from the estimated solution are
PxVehicleNW instances with high wheel count.

If eESTIMATE is chosen the quality of the estimate can be tuned with 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 without any precision. On the other hand, particularly low values such as 1 might result in 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 aggressively applied. In such situations large angular velocity differences at the clutch result in large torques 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 expected smooth transitions. The magnitude of accuracy loss and its subsequent effect on vehicle behavior are very difficult to quantify and really need tested for each vehicle and its use case.

It is recommended that eBEST_POSSIBLE is chosen for vehicles that require high levels of detail and that eESTIMATE is only chosen for vehicles that require lower levels of 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 the lowest possible value of 1 will turn out to provide perfectly acceptable. Smooth and physically 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 and contact modification. Sample code can be found in SnippetVehicleContactMod.

Section *Algorithm* described how scene query raycasts are used to compute vehicle suspension forces. Expanding on this theme, Section *Filtering* described how scene query and simulation filtering to categorise scene shapes as either drivable or non-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 system of raycasts and filtering. One problem is that it may be impractical to author every shape in the scene as either drivable or non-drivable: it is easy to imagine a landscape modelled with a mesh that is partially drivable and partially non-drivable. Another problem is that raycasts 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 the angled ground plane.
Figure 2b: The wheel rolls towards a wall in Frame 1 and is immediately pushed up to the elevated surface in Frame 2.

The problem illustrated in Figure 2a can be solved by replacing raycasts with sweeps. Instead of performing a raycast along the suspension direction through the wheel, the shape representing the wheel is swept from its transform at maximum compression to its transform at maximum elongation. Sweeping a volume through the scene means that all possible contact planes are considered. This is illustrated in Figure 3.
In Figure 3, it is easy to see that there are multiple contact points under the wheel with a different normal. A decision needs to be made about which of these contacts to accept as the driving surface and which to ignore. In some scenarios it is sufficient just to take the first contact encountered by the sweep and ignore all others. For such cases it is recommended to issue a blocking sweep. PhysX supports two types of scene query: blocking and non-blocking. A detailed description of blocking and non-blocking queries can be found in Section Filtering. In summary, however, a blocking sweep will return the first contact encountered by the swept volume, while non-blocking sweeps will return all contacts encountered by the sweep. The scenario in Figure 3 suggests that a blocking sweep will 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 scenarios, such as those depicted in Figure 2b, are more complex and require a non-blocking sweep.

Figure 4 shows a wheel rolling along a horizontal plane towards a vertical plane. The expected behavior is that the wheel continues to drive on the horizontal plane and is blocked by the vertical plane. It turns out that this can be readily achieved by judicious choice of sweep contacts and rigid body contacts. The first thing to note is that the sweep...
will return the three contact planes labelled A, B and C in Figure 4. If 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 strategy 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 continues to drive on the lower horizontal plane. The strategy adopted by PhysX is 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 introducing 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 contact points in the light blue zone are accepted as driving planes, while rigid body contacts in the pink zone are accepted as rigid body contact planes.
Figure 6: The angle between contact normal and the suspension direction is used to categorise contact planes as either rigid body contacts or sweep contacts. Normals close to the suspension direction are accepted as driving planes, while 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 numerical test for drivable and non-drivable contact points and normals allows a relaxation of the filtering rules described in Section Filtering. The idea now is to set up simulation data so that wheel shapes sweep against and collide with pretty much everything in the scene. The two threshold angles will filter and categorise sweep and rigid body contacts to generate the desired behavior.

The threshold angles shown in Figure 5 and Figure 6 are configured with the following function call:

```cpp
void PxVehicleSetSweepHitRejectionAngles(const PxF32 pointRejectAngle, const PxF32 faceRejectAngle)
```

The code snippet SnippetVehicleContactMod demonstrates how to configure blocking and non-blocking sweeps. This snippet can be configured to run with either type of sweep by modifying the BLOCKING_SWEEPS define. Running the
BLOCKING_SWEEPS demonstrates that the situation depicted in Figure 4 requires non-blocking sweeps to ensure that the elevated horizontal plane is not chosen as the driving surface.

Suspension sweeps are issued with the following code:

```cpp
//Suspension sweeps (instead of raycasts).
//Sweeps provide more information about the geometry under the wheel.
 PxVehicleWheels* vehicles[NUM_VEHICLES] = {gVehicle4W[0], gVehicle4W[1]};
 PxSweepQueryResult* sweepResults = gVehicleSceneQueryData->getSweepQueryResultBuffer;
 const PxU32 sweepResultsSize = gVehicleSceneQueryData->getQueryResultBufferSize;
 PxVehicleSuspensionSweeps(gBatchQuery, NUM_VEHICLES, vehicles, sweepResults, sweepResultsSize);
```

In the event that non-blocking sweeps are implemented, the function `PxVehicleSetSweepHitRejectionAngles` is used to reject and accept sweep hits using the threshold angles. When blocking sweeps are implemented, only a single sweep contact is recorded. As a consequence, `PxVehicleUpdateSweeps` uses the blocking threshold angles and automatically works with the blocking sweep hit. Whether to use blocking or non-blocking sweeps is left to the developer, as it 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 computationally cheaper option of blocking sweeps, while other applications may expect the vehicle to drive on complex geometry and are prepared to accept the extra cost of non-blocking sweeps.

Categorisation of rigid body contacts is implemented using contact modification callbacks. Contact modification is described in Section Contact Modification. The PhysX Vehicles SDK provides the function `PxVehicleModifyWheelContacts` to accept or reject contact points using the defined threshold angles. This function should be called from the contact modification callback, which is owned by the application. Configuring contact modification callbacks involves a combination of simulation filter data and shader. The implementation details, therefore, are left to the application developer. SnippetVehicleContactMod illustrates one way to implement a contact modification callback using simulation filter data and the userdata pointers of `PxShape` and `PxRigidDynamic`. Other techniques are available using local knowledge in the application.

In addition to adding sweeps and contact modification, the snippet also applies continuous collision detection (CCD) to the wheel shapes. CCD is introduced in Section...
Collision Detection.
**Tuning Guide**

This Sections 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 data relating to the wheel/tire/suspension. Without this parameter it would be difficult to compute coordinates for render points and lines that ensure their visibility. Ideally, this 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 rolling axis. High 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 of looking at it is that a high MOI will result in less wheel spin when stamping on the accelerator. Conversely, lower values will result in more wheel spin when stamping on the accelerator.
If the wheel is approximately cylindrical then a simple formula can be used to compute MOI:

\[ \text{MOI} = 0.5 \times \text{Mass} \times \text{Radius} \times \text{Radius} \]

There is no reason, however, to rely on equations to compute this value. A strategy for tuning this number might be to start with the equation and 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 rest. The damping rate describes the rate at which a freely spinning wheel loses speed. Here, a freely spinning wheel is one that experiences no force except damping forces arising from the wheel's internal bearings. Higher damping rates will result in the wheel coming to rest in shorter times, while lower damping rates will result in the wheel maintaining speed for longer. Values in range (0.25, 2) seem sensible. Experimentation is always a good idea, even outside this range. Exercise some caution with very small damping rates. In particular, a damping rate of exactly 0 should be avoided.

**mMaxBrakeTorque:**

This is the value of the torque applied to the wheel when the brakes are maximally applied. Higher torques will lock the wheel quicker when braking, while lower torques will take longer to lock the wheel. This value is strongly related to the MOI because the MOI determines how quickly the wheel will react to applied torques.

A value of around 1500 is a good starting point for a vanilla wheel but a web search will reveal typical braking torques. One difficulty is that these are often expressed by manufacturers as braking horsepower or in "pounds inches". The values required here are in "Newton metres".

**mMaxHandBrakeTorque:**

This is the same as the max brake torque except for the handbrake rather than the 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 point, while a value of 0 is necessary for the front wheels to make sur react to the handbrake.

mMaxSteer:

This is the value of the steer angle of the wheel (in radians) when the is at full lock. Typically, for a 4-wheeled car, only the front wheels respecs to somewhere between 30 degrees and 90 degrees seems like a good but it really depends on the vehicle being simulated. Larger values of max steer will result in tighter turns, while smaller values will result in wider turns. Be aware, though, that large steer angles at large speeds are likely to result in the car losing control, just as would happen with a real car. A good way 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 turns applied by the steering wheel).

mToeAngle:

This is the angle of the wheel (in radians) that occurs with no steer angle can be used to help the car straighten up after coming out of a good number to experiment with but is best left at 0 unless detailed required.

To help the car straighten up apply a small negative angle to one of and a small positive angle to the other front wheel. By choosing which the positive angles, and which the negative, it is straightforward to need either "toe-in" or "toe-out". A "toe-in" configuration, the front wheels towards each other, should help the car straighten up after a turn but of making it a little harder to turn in the first place. A "toe-out" config the opposite effect. Toe angles greater than a few degrees are best a

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 a good point.

void setSuspForceAppPointOffset(const PxU32 id, const PxVec3& offset):

This is the application point of the suspension force, expressed as an offset vector from the center of mass of the vehicle's rigid body. Another way of expressing this is to start at the center of mass of the rigid body, then move along the offset vector. The point at the end of the offset vector is the point at which suspension forces will be applied.

In a real vehicle the suspension forces are mediated through the suspension strut. These are often incredibly complex mechanical systems that are computationally expensive to simulate. As a consequence, instead of modeling the actual suspension strut, it makes sense to assume that the suspension strut has an effective 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 sorts of possibilities, freed from the constraints of the real world.

Deciding on the suspension force application point requires some thought. The 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 travel direction. Somewhere along this line seems like an even better idea for the application point. At the same time, it opens up all sorts of possibilities, freed from the constraints of the real world.

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 rather than into the turn. This narrows down the application point to really quite a small section of a known line.

When editing the suspension force application point it is important to remember that 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 so much load 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 achieve a good 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 duplicate the suspension force application point. Only for really detailed editing is it advised to start tweaking the tire force app offset independently of the suspension force app offset.

void setWheelCentreOffset(const PxU32 id, const PxVec3& offset):

This is the center of the wheel at rest position, expressed as an offset vector 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 four wheels 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 is moved forward then it would be expected that the front wheels would need to support more mass than the rear wheels. Conversely, a center of mass nearer the rear wheels ought to result in the rear suspension springs supporting more mass.

**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 mass.
center of mass. More specifically, the rigid body mass and center of mass can be computed using the equations presented in Section Algorithm and then applied to the vehicle's PxRigidDynamic instance. The second approach starts with the rigid body mass and center of mass of the vehicle's PxRigidDynamic instance and works backwards to compute and set the sprung masses. It makes use of the function PxVehicleComputeSprungMasses that was introduced in Section setupWheelsSimulationData.

mMaxCompression:

mMaxDroop:

These values describe the maximum compression and elongation in metres that 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 consider 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 maximum droop is reached. The spring cannot be elongated beyond this point, so 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 is the maximum compression of the spring.

It is important to choose the maximum compression value so that the wheel is never placed where the visual mesh of the wheel intersects the visual mesh of the car's chassis. Ideally, these values will be exported from the 3D modeler.

mSpringStrength:

This value describes the strength of the suspension spring. The spring strength has a profound influence on handling by modulating the time it takes for the vehicle to respond to bumps in the road and on the amount of load experienced by the suspension.

Key to understanding the effect of spring strength is the concept of a spring's natural frequency. Consider a simple spring system, such as a pendulum swinging on a spring.
forth. The number of trips per second that the pendulum makes from right to full left and then back again is called the natural frequency of the pendulum. A powerful pendulum spring will result in the pendulum swinging faster, thereby increasing the natural frequency. Conversely, increasing the pendulum mass will result in a slower oscillation, thereby reducing the natural frequency.

In the context of a suspension spring supporting a fixed portion of vehicle mass, the strength of the spring will affect the natural frequency; that is, the stronger the spring can respond to changes in load distribution. Consider a car taking a corner. 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 applying forces to redistribute the load is controlled by the natural frequency. Very high natural frequencies, such as those on a racing car, will naturally produce a very quick response because the load on the tires, and therefore the forces they can generate, is varying very rapidly. Very low natural frequencies, on the other hand, will result in the car taking a long time to straighten up even after the turn is complete. This will produce sluggish and unresponsive handling.

Another effect of strength and natural frequency is the response of the vehicle to a bump in the road. High natural frequencies can result in the car responding strongly and quickly to the bump, with the wheel possibly even leaving the road for a short while. This not only creates a bumpy ride but also periods of time when the tire is generating no forces. Weaker springs will result in a smoother trip, with weaker but more constant tire forces. A balance must be found to accommodate 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 much 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 it takes to complete a single oscillation, and is mathematically equal to the reciprocal of the natural frequency. In order to achieve a stable simulation the spring must be sampled at several points during each oscillation. 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 that will occur during each spring oscillation. This ratio is simply the spring period divided by the timestep

\[
\alpha = \frac{\sqrt{m_{sprung mass}/m_{spring strength}}}{\text{timestep}}
\]

where \( \sqrt{m_{sprung mass}/m_{spring strength}} \) is the period of the spring. A value of 1.0 means that the chosen timestep and spring properties only allow a single sample of the spring during each oscillation. As described above, this is guaranteed to produce unstable behavior. In fact, the argument presented so far 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 difficult to predict and depends on many other parameters. As a guide, it is recommended that the timestep and spring properties are chosen to produce an \( \alpha \) value greater than 5.0; that is, a minimum of five simulation updates 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 not always 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 springs of a typical family car have natural frequency somewhere between 5 and 10; that is, such a car would make 5-10 oscillations per second if gently dropped to the ground. If the sprung mass is already known, then the spring strength can be calculated from the following equation

\[
m_{spring strength} = \text{natural frequency} \times \text{natural frequency} \times m_{sprung mass}
\]

**Note:** To achieve a spring that is theoretically correct, the values of \( m_{sprung mass}, m_{spring strength} \) and \( m_{max droop} \) should be chosen so that they obey the equation

\[
m_{spring strength} \times m_{max droop} = m_{sprung mass} \times g \]

where \( g \) is the gravitational acceleration. When this equation is satisfied, the spring is guaranteed to provide exactly zero force at maximum elongation and also to support the sprung mass at the rest pose (the rest pose is defined by...
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 this 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 game, 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 stops providing upward force before it hits maximum elongation or that it still 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 particular, tire load filtering, discussed in Section PxVehicleTireLoadFilterData, further disguises any imperfection.

mSpringDamperRate:

This describes the rate at which the spring dissipates the energy stored.

Key to the understanding of damper rate are the concepts of under-damping, and critical damping. An over-damped pendulum displaced from rest is unable to make a single back-and-forth trip before it dissipates all its energy, while 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 shifting the weight from the left suspension to the right suspension then back again. If this happens, the tire load, and the forces generated, would be extremely variable, resulting in twitchy and 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 dam spring. It is helpful to introduce a value known as the damping ratio mathematically describe the under-damping, critical damping an regimes.

\[
dampingRatio = \frac{mSpringDamperRate}{2 \cdot \sqrt{mSpringStrength \cdot mSprungMass}}
\]

A dampingRatio with value greater than 1.0 produces over-damping, 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 number to be subjectively applied to the damping ratio. From here it can be directly computed by rearranging the equation above

\[
mSpringDamperRate = dampingRatio \cdot 2 \cdot \sqrt{mSpringStrength \cdot mSprungMass}
\]

A typical family car is probably slightly over-damped, having damping perhaps just over 1.0. A guideline would be that values very far from critical 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 seems like 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 camber in such that is, the left and right wheels almost seem to form the edges of a V shape when viewed from the front or rear along the forward axis of the vehicle. Springs, on the other hand, typically camber outwards; that is, they form the outer edges of an A shape when viewed from the front or rear along the forward axis.
axis of the vehicle.

These three values allow the camber angle to be computed for any spring compression using simple linear interpolation. At rest, when the spring is neither elongated or compressed, the camber angle is equal to \( m_{\text{CamberAtRest}} \). When the spring is compressed the camber is computed as a linear interpolation between \( m_{\text{CamberAtRest}} \) and \( m_{\text{CamberAtMaxCompression}} \). When the spring is elongated the camber is computed as a linear interpolation between \( m_{\text{CamberAtRest}} \) and \( m_{\text{CamberAtMaxDroop}} \).

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 PxShape that geometrically represents the wheel.

**PxVehicleAntiRollBar**

When a vehicle takes a corner the turning force causes the car to roll. Suspension springs on the outside of the turn are compressed while suspension 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 will lose control of the vehicle. In such cases, there is even a danger that the vehicle will rotate onto its side. For less severe rolls there still remains a handling problem that arises from the distribution of load between the inside and outside tires. The issue there is 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 when cornering. They typically work as a torsion spring that applies a torque in order to minimise the difference in spring displacement for a pair of wheels. A standard family car might feature a front and rear anti-roll bar. The front bar applies a torque to reduce 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 magnitude is also 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 rear anti-roll bar. Increasing the stiffness of the front anti-roll bar typically reduces over-steer.

**mWheel0**: mWheel1:

The anti-roll bar connects two wheels described by the indices mW mWheel1.

**mStiffness**:

This parameter describes the stiffness of the anti-roll bar.

**PxVehicleTireData**

The tire force computation is performed in two conceptual stages. The computation independently computes the lateral and longitudinal components of the force using linear equations. These independent forces are computed by treating 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 experienced by the tire. The second stage of the computation applies the rule that the force is limited by the product of the tire load and friction. Just as with rigid bodies, tires are able to resist greater horizontal forces when they experience a large normal 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 friction value. The default PhysX Vehicle tire model employs a series of smoothing functions to implement the normalization of the combined tire forces.

In addition to the lateral and longitudinal components of force a camber thrust force, arising from the camber angle of the tire, is also computed. Typically this provides only a small correction to the effect of the lateral and longitudinal components, and the camber force participates in the normalization process.

The following tire parameters describe the computation of the independent lateral and longitudinal components; that is, the first conceptual stage of the force computation. Reference is made throughout to the handling consequences of the
normalization process.

mLongitudinalStiffnessPerUnitGravity:

The longitudinal stiffness describes the longitudinal forces that develop due to longitudinal slip (in radians). Here, a variable that represents the longitudinal stiffness per unit gravity has been introduced in order to make the variable robust against edits to the value of gravitational acceleration. The longitudinal stiffness approximately the product of the longitudinal stiffness per unit gravity and the longitudinal slip and the magnitude of gravitational acceleration:

\[
\text{longitudinalTireForce} = \text{mLongitudinalStiffnessPerUnitGravity} \times \text{longitudinalSlip} \times \text{gravity};
\]

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 load on the tire so be aware that increases in this value might have no effect or even come 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 load on the tire so be aware that increases in this value might have no effect or even come at the expense of reduced longitudinal force.

Lateral stiffness is a little more complicated than longitudinal stiffness typically provide poor response under heavy load. Typical for car tires, lateral force against load that has linear response close to zero load and saturates 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 is in a regime where applying more load will not result in more tire stiffness. In this latter regime it would be expected that the tire would start slipping.

The combination of two values mLatStiffX and mLatStiffY describe a graph of lateral stiffness per unit load as a function of normalized tire load. The tire force computation 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 stiffness per 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 the tire has 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 at rest. For example, a value of 2 for mLatStiffX means that when the tire has a load more than twice its rest load it can deliver no more lateral stiffness no matter how much extra load is applied to the tire. In the graph below mLatStiffX has value 3.

The parameter mLatStiffY describes the maximum stiffness per unit of lateral slip (in radians) per unit rest load. The maximum stiffness is delivered when the tire is in the 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 load on the tire, then computing the normalized load in order to compute the number of rest loads 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 parameterized by mLatStiffX and mLatStiffY is queried to provide the lateral stiffness per unit rest load. The final value for the lateral stiffness is then computed by multiplying the queried graph value by the rest load. This final value describes the lateral stiffness per unit 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, and attempts to approximate this relationship.

Typically, tires have a linear response at small slips. This means that when the tire is only slightly slipping, it is able to generate a response force as the slip increases. At greater values of slip, the force can actually decrease from the peak value that occurs at the optimum slip. At optimum slip, the tire eventually starts behaving less and less efficiently and hits a plateau of inefficiency.

The friction value for the combination of surface type and tire type has already been discussed in Section *Tire Friction on Drivable Surfaces*. 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 tire model then responds to the final friction value.
The first two values describe the friction at zero tire slip: \( \text{mFrictionVsSlipGraph}[0][0] = 0 \), and \( \text{mFrictionVsSlipGraph}[0][1] = \) friction at zero slip.

The next two values describe the optimum slip and the friction at optimum slip: \( \text{mFrictionVsSlipGraph}[1][0] = \) optimum slip, \( \text{mFrictionVsSlipGraph}[1][1] = \) 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: \( \text{mFrictionVsSlipGraph}[2][0] = \) slip at the start of the plateau of inefficiency, \( \text{mFrictionVsSlipGraph}[2][1] = \) the friction available at the plateau of inefficiency.

In the graph below the following values have been used:

\[ \begin{align*}
\text{mFrictionVsSlipGraph}[0][0] &= 0.0 \\
\text{mFrictionVsSlipGraph}[0][1] &= 0.4 \\
\text{mFrictionVsSlipGraph}[1][0] &= 0.5 \\
\text{mFrictionVsSlipGraph}[1][1] &= 1.0 \\
\text{mFrictionVsSlipGraph}[2][0] &= 0.75 \\
\text{mFrictionVsSlipGraph}[2][1] &= 0.60
\end{align*} \]
The friction values described here are used to scale the friction of the ground surface. This means they should be in range (0,1) but this is not a strict requirement. The friction from the graph would be close to 1.0 in order to provide a small correction to the ground surface friction.

A good starting point for this is a flat graph of friction vs slip with these values:

\[
\begin{align*}
m\text{FrictionVsSlipGraph}[0][0] &= 0.0 \\
m\text{FrictionVsSlipGraph}[0][1] &= 1.0 \\
m\text{FrictionVsSlipGraph}[1][0] &= 0.5 \\
m\text{FrictionVsSlipGraph}[1][1] &= 1.0 \\
m\text{FrictionVsSlipGraph}[2][0] &= 1.0 \\
m\text{FrictionVsSlipGraph}[2][1] &= 1.0
\end{align*}
\]

The camber stiffness is analogous to the longitudinal and lateral stiffness, except that it describes the camber thrust force arising per unit camber angle (in radians). To the longitudinal stiffness, a camber stiffness per unit gravity has been introduced:
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:

\[ \text{camber Tire Force} = \text{mCamber Stiffness Per Unit Gravity} \times \text{camber Angle} \times \text{gravity}; \]

\mType:

This parameter has been explained in Section *Tire Friction on Drivable Surfaces*.

**PxVehicleEngineData**

\mMOI:

This is the moment of inertia of the engine around the axis of rotation. Larger values make it harder to accelerate the engine, while lower 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. This is expressed 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, where the interpolation is governed by the acceleration control value.
gamepad or keyboard. At full throttle `mDampingRateFullThrottle` is applied, while `mDampingRateZeroThrottleClutchEngaged` is applied at zero throttle. The damping rate is an interpolation between `mDampingRateFullThrottle` and `mDampingRateZeroThrottleClutchDisengaged`.

The three values allow a range of effects to be generated: good acceleration that isn't hampered by strong damping forces, tunable damping forces when neutral gear during a gear change, and strong damping forces that will 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 at rates of 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 ranges of engine speed that produce poor torques. A skilled driver will make good use of the gears to ensure that the car remains in the "good" range where the engine is most responsive. 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 multiplier in range (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...
top speed in that gear. Typically, the higher the gear, the lower the gear ratio. Gear must always be given a value of 0, while reverse gear must have a negative gear ratio. Typical values might be 4 for first gear and 1.1 for fifth gear.

mFinalRatio:

The gear ratio used in the simulator is the gear ratio of the current gear multiplied by the final ratio. The final ratio is a quick and rough way of changing the gearing without having to edit each individual entry. Further, quoted gearing values from manufacturers typically mention ratios for each gear along with a final ratio; a value might be around 4.

mSwitchTime:

The switch time describes how long it takes (in seconds) for a gear change to be completed. It is impossible to change gear immediately in a real car; for example, require neutral to be engaged for a short time before engaging the desired target gear. While the gear change is being completed the car will be in neutral. A good trick might be to penalize players that use an automatic gear box by increasing the gear switch time.

If the autobox is enabled it is a good idea to set this value significantly lower than PxVehicleAutoBoxData::setLatency. If the autobox latency is smaller than the switch time then the autobox might decide to initiate a downward gear change immediately after an upward gear shift has been completed. This situation can leave the car cycling between neutral and first gear with very short interludes.

PxVehicleAutoBoxData

The autobox initiates gear changes up or down based on the rotation speed of the engine. If the engine is rotating faster than a threshold value stored in PxVehicleAutoBoxData then a gear increment will be initiated. On the contrary, if the engine is rotating slower than a threshold value then the autobox will initiate a gear decrement. The autobox only initiates gear changes upward or downward, never skipping a single gear at a time.
It is worth noting that if the autobox initiates a gear change then the accelerator pedal 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 limitation. This is in keeping with typical real-world autobox behavior. The idea behind this is to stop the engine wildly accelerating during the neutral phase of the gear change, thereby 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 or manual gear change is still active.

If the autobox is too simplistic for the application's requirements, PxVehicleGearsData can be readily disabled. The choices following this are either to revert to a manual gear model or to implement a custom autobox in the application. A transition to a specific gear can be initiated with PxVehicleDriveDynData::startGearChange, while single gear changes can be initiated with PxVehicleDriveDynData::mGearUpPressed / PxVehicleDriveDynData::mGearDownPressed.

The autobox can be enabled or disabled by toggling 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 stored in mUpRatios[PxVehicleDriveDynData::getCurrentGear()]

PxReal mDownRatios[PxVehicleGearsData::eGEARSRATIO_COUNT]:
The autobox will initiate a gear decrement if the ratio of the engine rotation speed to the maximum allowed engine rotation speed:

\[
\frac{\text{PxVehicleDriveDynData::getEngineRotationSpeed()}}{\text{PxVehicleEngineData::mMaxOmega}}
\]

is less than the value stored in \text{mUpRatios}[\text{PxVehicleDriveDynData::getCurrentGear()}].

\begin{verbatim}
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 value significantly higher than \text{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 situation can leave the car cycling between neutral and first gear with very short interludes in 2nd gear.

\textbf{PxVehicleClutchData}
\end{verbatim}

\textbf{mStrength}:

This describes how strongly the clutch couples the engine to the wheels and how quickly differences in speed are eliminated by distributing torque to the wheels.

Weaker values will result in more clutch slip, especially after changing gear or 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.

Value of 10 is a good starting point.

\textbf{PxVehicleAckermannGeometryData}
mAccuracy:

Ackermann correction allows better cornering by steering the left and 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 steering correction to be controlled. Choosing a value of 0 completely disables steer correction. A value of 1.0, on the other hand, achieves the impossible perfection.

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 bumpy 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 spring being more compressed than would be experienced with a real vehicle. A consequence of this oscillation 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 correct this numerical problem by smoothing the tire load with the aim of making the handling smoother and more predictable.
A key concept is that of normalized tire loads. A normalized tire load is the load divided by the load experienced when the vehicle is in its rest configuration. 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 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 $m_{\text{MinNormalisedLoad}}$ produce a filtered normalized load of $m_{\text{MinFilteredNormalisedLoad}}$. Normalized loads greater than $m_{\text{MaxNormalisedLoad}}$ produce a filtered normalized load of $m_{\text{MaxFilteredNormalisedLoad}}$. Normalized loads between $m_{\text{MinNormalisedLoad}}$ and $m_{\text{MaxNormalisedLoad}}$ produce a filtered normalized load in-between $m_{\text{MinFilteredNormalisedLoad}}$ and $m_{\text{MaxFilteredNormalisedLoad}}$, as computed by direct interpolation.

![Graph showing normalized vs. filtered normalized tire loads](image)

Choosing $m_{\text{MaxNormalisedLoad}}$ and $m_{\text{MaxFilteredNormalisedLoad}}$ limits the maximum load that will ever be used in the simulation. On the other hand, choosing $m_{\text{MinFilteredNormalisedLoad}}>0$ and/or $m_{\text{MinNormalisedLoad}}>0$ 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

\[
\begin{align*}
    m_{\text{MinNormalisedLoad}} &= m_{\text{MaxFilteredNormalisedLoad}} = 0 \\
    m_{\text{MaxNormalisedLoad}} &= m_{\text{MaxFilteredNormalisedLoad}} = 1000.
\end{align*}
\]

**Note:** Tires may only generate forces if the tire is touching the ground; if the 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, 4-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. Choosing 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 available for 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 power 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 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 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 easily corner by permitting the outside wheel to rotate quicker than the inside wheel.

This parameter describes the maximum difference in wheel rotation speeds 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 the front bias 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 allowed wheel rotation speed ratio.

This value is ignored except for front-wheel drive or four wheel drive with limited slip.
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 with limited slip.
A good starting value is around 1.3.
mCentreBias:

This value is similar to the mFrontBias and mRearBias, except that it refers to the sum of the front wheel rotation speeds and the sum of the rear wheel rotation speeds.

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 parameter when 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 the moment of inertia of the cuboid that bounds the chassis geometry. If the cuboid is \( W \) wide, \( H \) high, and \( L \) long then the moment of inertia for a vehicle of mass \( M \) is:

\[
\begin{align*}
((L^2+H^2)M/12, (W^2+L^2)M/12, (H^2+W^2)M/12)
\end{align*}
\]

However, this is only a rough guide. Tweaking each value will modify the motion around the corresponding axis, with higher values making it harder to induce rotational speed from tire and suspension forces.

Providing unphysical values for the moment of inertia will result in either 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 go-to editable values.
values and, as such, has a profound effect on handling.

To discuss the center of mass it is useful to consider a typical 4-wheeled vehicle. A chassis mesh whose origin is at the center of the four wheels but with the following discussion a little simpler. It might be expected that the center of mass lies somewhere near this origin because vehicles are designed in a way to load almost evenly between the four wheels. More specifically, it might be expected 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 density near the bottom of the chassis due to density of the engine and other mechanical systems.

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 particular 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 might be expected that the center of mass is a little nearer the front wheels than the rear wheels due to 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 should help cornering because more load is distributed to the front tires. However, this comes at the expense of reduced load on the rear tires, meaning that the car might turn more quickly only to spin out because the rear tires lose grip more quickly. Small changes 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 consistent. It is possible to mathematically describe the relationship between the center of mass and the mass split between the suspensions. However, the editing 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 vehicle tuning.

Jittery Vehicles

1. Have PxInitVehicleSDK and PxVehicleSetBasisVectors been called before the first execution of PxVehicleUpdates? Check the error stream for warnings.
2. Does the length scale of PxTolerancesScale match the length scale (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 reliable simulation? See Section PxVehicleSuspensionData for more details and update the natural frequency or timestep accordingly. Remember the timestep can be updated per vehicle with PxVehicleWheelsSimData::setSubStepCount.
4. Are the maximum suspension droop and compression set to values that allow some suspension motion?

The Engine Rotation Refuses To Spin Quickly

1. Are the tires resisting the engine motion through excessive friction? Place 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 rates reflect the length scale? Note the documented SI units of each variable and recompute the values as appropriate.
3. Is the moment of inertia too large? A value of 1 or its equivalent in the relevant length scale is a good estimate for testing purposes.
4. Is the peak torque too small to drive the engine? Scale the default with the mass of the vehicle with the knowledge that the default value is 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 testing purposes.
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 PxVehicleNW only)? Make sure that the differential is properly configured.
3. Is the brake or handbrake engaged? Ensure that the brake and handbrake are both 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 appropriate.
5. Are the wheels' moments of inertia too high? Recompute the wheels' moment of inertia.
6. Are the wheels' damping rates too high? Reduce the wheels' damping rates.
7. Are the tires resisting the engine motion through excessive friction forces? 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 suspensions? Check the filtering configuration for shapes attached to the vehicle's actor, search for contacts involving shapes attached to the vehicle's actor using PVD.
2. Is sufficient friction being delivered to the tire contact patch? Query experienced by the tires during the execution of PxVehicleUpdates.
PxVehicleWheelsDynData::getTireFriction.

3. Do the suspension forces (and the loads on the tires) reflect the body actor? Query the suspension forces using PxVehicleWheelsDynData::getSuspensionForce. A 4-wheeled vehicle should generate suspension forces of approximately actorMass*gravity/4. Ensure that the masses of the vehicle suspensions to ensure that the driven wheels experience significant tire load.

4. Do the tires generate significant longitudinal tire forces? Check that the longitudinal slip of the tire is non-zero and approaches 1.0 when the wheels are spinning quickly without forward motion. Ensure that PxVehicleSetBasisVectors has been called with the correct forward vector if the longitudinal slip is vanishingly small. Further test that the forward vector has been set correctly by using PxVehicleWheelsDynData::getTireLongitudinalDir.

5. Is the tire longitudinal stiffness too small? Adjust the longitudinal stiffness back to the default value and test.

6. Is the mass of the vehicle's rigid body actor too large to be driven by the engine peak torque? Test that the mass of the actor is a sensible value and set it accordingly.

7. Is the rigid body actor in a PhysX scene and is the scene being updated? Ensure that 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 turning motion? Check that the moment of inertia of the vehicle's rigid body actor is a sensible value. Use the moment of inertia of a box with width/height/length of the vehicle as a starting 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, 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 angle? Use PxVehicleWheelsDynData::getLatSlip to query the slip angles. PxVehicleSetBasisVectors has been called with the correct forward and up vectors if the lateral slips are vanishingly small. Further test that the basis vectors have been set correctly 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? Reduce 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 selection and change gears manually to test if the autobox is failing to switch gear. Adjust the gear settings to make sure that it will automatically increase the gear at sensible engine rotation speeds.

3. Is the engine powerful enough to quickly accelerate the car? Increase the peak torque of the engine.

4. Do the wheels have high moments of inertia that prevent significant longitudinal slips developing? 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 appropriate.

The Vehicle Turns Too Quickly/Too Slowly
1. Does the moment of inertia of the rigid body actor need tweaking? The component of the moment of inertia that corresponds to motion about the up vector.

Increasing the moment of inertia will slow the turn rate, decreasing 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? Decrease the rate of increase of the accelerator pedal value by filtering the controller. Remember that aggressively pressing the accelerator pedal on a powerful car ought to lead to wheel spin.

2. Are the wheel moments of inertia too low? Increase the wheel moments of inertia.

The Wheels Spin Too Much When Cornering

1. Does the vehicle have a limited slip differential? If applicable set the differential type 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 enabled then the problem is probably that the latency of the autobox is shorter than the time spent performing 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 latency time has passed. During a gear change the vehicle enters neutral gear and the accelerator pedal is uncoupled from the engine, meaning that the engine will slow down. When the vehicle enters the target gear at the end of the gear change the autobox might decide immediately that the engine is too slow for 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
and never reaches its target gear. This will not happen if the latency (PxVehicleAutoBoxData::setLatency) is set significantly larger than the gear switch time (PxVehicleGearsData::mSwitchTime).

**The Vehicle Under-steers Then Over-steers**

1. Is the vehicle on a bumpy surface? Edit the values in PxVehicleTireLoadFilterData so that the filtered normalized tire load has a flatter response to suspension compression.

**The Vehicle Slows Down Unnaturally**

1. Does the vehicle not slow down smoothly to rest? Take a look at the longitudinal slip values to see if they are oscillating between positive and negative. If oscillation occurs, then two options are available that can be used separately or in conjunction with each other. The first option is to use PxVehicleWheelsSimData::setSubStepCount to force more vehicle updates as the forward speed of the vehicle approaches zero. The second option is to use PxVehicleWheelsSimData::setMinLongSlipDenominator to ensure that 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::mF to reduce the available friction for slipping wheels.
References

Anti-roll Bar


Tire Modeling

The default tire model employed by PhysX vehicles is discussed in CarSimEd documentation:


PhysX vehicles allow any tire model to be simulated by specifying discussed in Tire Shaders. A tire model commonly used in engineering Pacejka tire model:


http://phors.locost7.info/phors22.htm

Engine Torque Curves


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


General

http://phors.locost7.info/intro.htm

http://www.millikenresearch.com/rcvd.html

Introduction

The character controller (CCT) SDK is an external component built on top of the PhysX 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 a features in each game. For example the character's bounding volume in one game, and an inverted pyramid in another. The CCT SDK does not a one-size-fits-all solution that would work out-of-the-box for all possible. It provides the basic features common to all CCTs: character control and interactions. It is a default starting point for users, a strong base that one can later modify or customize if needed.
Kinematic Character Controller

The PhysX CCT is a kinematic controller. Traditionally, character controllers can be either kinematic or dynamic. A kinematic controller directly works with input displacement vectors (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 SDK, but they still used a character controller to move a player in a level. These games, even Doom, had a dedicated, customized piece of code to implement collision detection and response, which was often the only piece of physics in the whole game. It actually had little physics, but a lot of carefully tweaked values to provide a good controlling the player. The particular behavior it implemented is often called the 'collide and slide' algorithm, and it has been 'tweaked for more than a decade'. The PhysX CCT module is an implementation of such an algorithm, providing a robust behavior for character control.

The main advantage of kinematic controllers is that they do not suffer from 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 various commercial & non-commercial physics packages for years. This leads to three main problems:
  - the tunneling effect itself: if the character goes too fast, it might tunnel through a wall
  - as a consequence, the character's maximum velocity be limited (thus also limiting the game play possibilities)
  - even if it does not tunnel, the character might jitter when pushed into a corner for example, because the physics engine keeps moving it back and 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 should not slide. So infinite friction is needed here. When the character is moving forward on the ramp, it should not slow down. One does not need any friction here. But when the character is sliding against a wall, it should not slow down either. Friction is usually either 0 or infinite. Unfortunately the friction model in the physics engine might not be perfect, and it is easy to end up with either too much friction (the character slows down a tiny bit) or a very-large-but-not-infinite friction (the character slides very slowly on that ramp no matter how artificially big the friction parameters are). The conflicting requirements for ramps also mean that simply no way to perfectly model desired behavior.

• Trouble with restitution: typically, restitution should be avoided for characters. When the character moves fast and collides with a wall, it should not bounce away. When the character falls from a height and lands on the ground, flexing its legs, any bounce should be prevented. But once again, even when the restitution is zero, a physics engine can nonetheless make the CCTs bounce a bit. This is not only related to the imperfect nature of the linear solver, it also has to do with how typical penetration-depth-based engines recover from overlap situations, sometimes applying excessive forces that separate the objects too much.

• Undesired jumps: characters must often stick to the ground, not slide. Physical behavior should be. For example characters in action games move fast, at unrealistic speeds. When they reach the top of a ramp, they often makes them jump a bit, in the same way a fast car would jump in San Francisco. But that is often not the desired behavior: instead the character should often stick to the ground regardless of its current velocity. This is implemented using fixed joints, but this is an unnecessarily complex 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 implemented 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 the features simply to emulate what is otherwise a much less complex piece. 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:

```cpp
PxScene* scene; // Previously created scene
PxControllerManager* manager = PxCreateControllerManager(*scene);
```

Then, create one controller for each character in the game. At the time of writing, boxes (`PxBoxController`) and capsules (`PxCapsuleController`) are supported. A capsule controller, for example, is created this way:

```cpp
PxCapsuleControllerDesc desc;
...<fill the descriptor here>
...
PxController* c = manager->createController(desc);
```

The manager class will keep track of all created controllers. They can be retrieved at any time using the following functions:

```cpp
PxU32 PxControllerManager::getNbControllers() const = 0;
PxController* PxControllerManager::getController(PxU32 index) = 0;
```

To release a character controller, simply call its release function:

```cpp
void PxController::release() = 0;
```

To release all created character controllers at once, either release the manager itself, or use the following function if you intend to keep using the manager:

```cpp
void PxControllerManager::purgeControllers() = 0;
```

The creation of a controller manager and its subsequent controller is illustrated in SampleBridges.
Overlap Recovery Module

Ideally, character should not be created in an initial overlap state, i.e. created in a position where they do not overlap the surrounding geometry. PxScene overlap functions can be used to check the desired volume of space is empty prior to creating the character. By default the CCT module does not check for overlaps itself, and creating a character that initially overlaps the world's static geometry can have undesired and undefined behavior - like the character going through the ground for example.

However, the overlap recovery module can be used to automatically correct the character's initial position. As long as the amount of overlap is reasonable, the module should be able to relocate the character to a proper, collision-free position.

The overlap recovery module can be useful in several other situations:

- 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 overlap surrounding objects

When activated, the CCT module will automatically try to resolve the overlap by moving the CCT to a safe place where it does not overlap other objects anymore. This only concerns static objects, dynamic objects are ignored by this module.

Enable or disable the overlap recovery module with this function:

```cpp
void PxControllerManager::setOverlapRecoveryModule(bool flag);
```

By default the character controllers use precise sweep tests, whose accuracy is enough to avoid all penetration - provided the contact offset is not too small. In most cases the overlap recovery module is not needed. When it is used though, the sweep tests can be switched to less accurate but potentially faster versions, thus
function:

```cpp
void PxControllerManager::setPreciseSweeps(bool flag);
```
Character Volume

The character uses a bounding volume that is independent from already in the SDK. We currently support two different shapes around the chara

- An AABB, defined by a center position and an extents vector. TI rotate. It always has a fixed rotation even when the player is (visually) 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 default choice.

Note: versions prior to 2.3 also supported a sphere. This has been | PxCapsuleController is more robust and provides the same functionality (zero length)

A small skin is maintained around the character's volume, to avoid nu would otherwise happen when the character touches other shapes. The user-defined. When rendering the character's volume for debug purpose, expand the volume by the size of this skin to get accurate debug visual. defined in PxControllerDesc::contactOffset and later available through PxController::getContactOffset() function.
Volume Update

Sometimes it is useful to change the size of the character's volume at runtime. For example, if the character can crouch, it might be required to reduce the bounding volume so that it can then move to places he could not reach otherwise.

For the box controller, the related functions are:

```cpp
bool PxBoxController::setHalfHeight(PxF32 halfHeight)
bool PxBoxController::setHalfSideExtent(PxF32 halfSideExtent)
bool PxBoxController::setHalfForwardExtent(PxF32 halfForwardExtent)
```

And for the capsule controller:

```cpp
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 height is suddenly reduced without updating its position, the character will end up levitating above the ground for a few frames until gravity makes it fall and touch the ground again. This happens because the controllers' positions are located at the center of their shapes, rather than the bottom. Thus, to modify a controller's height and preserve its bottom position, one must change both the height and position of a controller. The following helper function does that automatically:

```cpp
void PxController::resize(PxF32 height) = 0;
```
It is important to note that volumes are directly modified without any extra tests, and thus it might happen that the resulting volume overlaps some geometry nearby. For example, when resizing the character to leave a crouch pose, i.e. when the size of the character is increased, it is important to first check that the volume of space above the character must be empty (collision free). It is recommended to use the various PxScene overlap queries for this purpose:

```cpp
bool PxScene::overlap(...) = 0;
```

Updating the character's volume at runtime to implement a 'crouch' motion is illustrated in SampleNorthPole. Using overlap queries to leave the crouch pose is done in the `SampleNorthPole::tryStandup()` function.
Moving a Character Controller

The heart of the CCT algorithm is the function that actually moves characters around:

```cpp
PxControllerCollisionFlags collisionFlags = PxController::move(const PxVec3& disp, PxF32 minDist, PxF32 elapsedTime const PxControllerFilters& filters, const PxObstacleContext* obstacles);
```

disp is the displacement vector for current frame. It is typically a combination of vertical motion due to gravity and lateral motion when your character is moving. Users are responsible for applying gravity to characters here.

minDist is a minimal length used to stop the recursive displacement algorithm when the remaining distance to travel goes below this limit.

elapsedTime is the amount of time that passed since the last call to the move function.

filters are filtering parameters similar to the ones used in the SDK. Use these to control what the character should collide with.

obstacles are optional additional obstacle objects with which the character should collide. Those objects are fully controlled by users and do not need to have counterpart SDK objects. Note that touched obstacles are cached, meaning that the cache needs to be invalidated if the collection of obstacles changes.

collisionFlags is a bit mask returned to users to define collision events that happened during the move. This is a combination of PxControllerCollisionFlag flags to trigger various character animations. For example your character might be playing a falling idle animation, and you might start the land animation as soon as 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 algorithm where the aforementioned 'collide-and-slide' algorithm takes place. The function will start from the CCT's current position, and use sweep tests to attempt...
required direction. If obstacles are found, it may make the CCT slide against them. Or the CCT can get blocked against a wall: the result of the move depends on the surrounding geometry. On the contrary, *PxController::setPosition* is a simple 'teleport' 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 the required position is in the middle of another object.

Both *PxController::move* and *PxController::setPosition* are demonstrated in SampleBridges.
Graphics Update

Each frame, after `PxController::move` calls, graphics object must be kept in sync with the new CCT positions. Controllers' positions can be accessed using:

```cpp
const PxExtendedVec3& PxController::getPosition() const;
```

This function returns the position from the center of the collision shape, which is used internally both within the PhysX SDK and by usual graphics APIs. Retrieving this position and passing it to the renderer is illustrated in SampleBridges. Note that the position uses double-precision, to make the CCT module work well with large worlds. Also note that a controller never rotates so you can only access its position.

Alternative helper functions are provided to work using the character's bottom position, a.k.a. the foot position:

```cpp
const PxExtendedVec3& PxController::getFootPosition() const;
bool PxController::setFootPosition(const PxExtendedVec3&);
```

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 elevations of the ground mesh. In the following picture the small step would stop the character completely. It feels unnatural because in the real world a character would just cross this small obstacle without thinking about it.

This is what auto-stepping enables us to do. Without any intervention from the player (i.e. without them thinking about it) the box correctly steps above the minor obstacle.
However, if the obstacle is too big, i.e. its height is greater than the step offset, the controller cannot climb automatically, and the character gets stuck (i.e., it cannot progress).
'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, thanks to their rounded nature, do not necessarily need the feature.

Even with a step offset of 0.0, capsules are able to go over small obstacles. Their rounded bottom produces an upward motion after colliding with a small obstacle.

Capsules with a non-zero step-offset can go over obstacles higher than the step-offset value because of the combined effect of the auto-stepping feature and their rounded shape. In this case the largest altitude a capsule can climb over is difficult to predict, depending on the auto-step value, the capsule's radius, and even the magnitude of the displacement vector.

This is why there are two different climbing modes for capsules:

- *PxCapsuleClimbingMode::eEASY*: in this mode, capsules are not constrained by the step offset value. They can potentially climb over obstacles higher than the step-offset value.
- *PxCapsuleClimbingMode::eCONSTRAINED*: in this mode, an attempt is made to make sure the capsule can not climb over obstacles higher than the
**Up Vector**

In order to implement the auto-stepping feature, the SDK needs to know about the up vector. The up vector is defined in `PxControllerDesc::upDirection` and later available through the `PxController::getUpDirection()` function.

The up vector does not need to be axis-aligned. It can be arbitrary, modified each frame using the `PxController::setUpDirection()` function, allowing the character to navigate on spherical worlds. This is demonstrated in SampleCustomGravity.

Modifying the up vector changes the way the CCT library sees character volumes. For example, a capsule is defined by a `PxCapsuleControllerDesc::height`, which is the 'vertical height' along the up vector. Thus, changing the up vector effectively rotates the capsule from the point of view of the library. The modification happens immediately, without tests to validate that the character does not overlap nearby geometry. It is then possible for the character to be penetrating some geometry right after the call. Using the overlap recovery module is recommended to solve these issues.
In the above picture the capsule on the left uses a vertical up vector with the surrounding geometry. On the right the up vector has been set to 45 degrees, and the capsule now penetrates the wall nearby. For most applications the up vector will be constant, and the same for all characters. These issues will only appear for characters navigating in spherical worlds (e.g. planetoids, etc).
Walkable Parts & Invisible Walls

By default the characters can move everywhere. This may not always be a good thing. In particular, it is often desired to prevent walking on polygons whose slope is steep. The SDK can do this automatically thanks to a user-defined slope limit. All polygons whose slope is higher than the limit slope will be marked as non-walkable, and the SDK will not let characters go there.

Two modes are available to define what happens when touching a non-walkable part. The desired mode is selected with the `PxControllerDesc::nonWalkableMode`.

- `PxControllerNonWalkableMode::ePREVENT_CLIMBING` prevents the character from moving up a slope, but does not move the character otherwise. The character will still be able to walk laterally on these polygons, and to move down their slope.
- `PxControllerNonWalkableMode::ePREVENT_CLIMBING_AND_FORCE_SLIDING` not only prevents the character from moving up non-walkable slopes but also forces it to slide down those slopes.

The slope limit is defined in `PxControllerDesc::slopeLimit` and later available through the `PxController::getSlopeLimit()` function. The limit is expressed as the cosine of the desired limit angle. For example, this uses a slope limit of 45 degrees:

```cpp
slopeLimit = cosf(PxMath::degToRad(45.0f));
```

Using `slopeLimit = 0.0f` automatically disables the feature (i.e. characters can go everywhere).

This feature is not always needed. A common strategy is to disable it and place invisible walls in the level, to restrict player's movements. The character module can create those walls for you, if `PxControllerDesc::invisibleWallHeight` is non-zero. The 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 those invisible walls are only created when non-walkable triangles are found. It is possible...
character to go over them, if its bounding volume is too small and does non-walkable triangles below him. The \texttt{PxControllerDesc::maxJumpHeight} addresses this issue, by extending the size of the bounding volume downward. That way all potentially non-walkable triangles are properly returned by the collision queries, and invisible walls are properly created - preventing the character from jumping.

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 kinematic objects. It is also not supported for static spheres or static capsules.
Obstacle Objects

Sometimes it is convenient to create additional obstacles for the CUs without creating an actual SDK object. This is useful in a number of situations. For example:

- the obstacles might only exist for a couple of frames, in which case deleting SDK objects is not always efficient.
- the obstacles might only exist for stopping the characters, not the SDK objects. This would be for example invisible walls around geometry that characters should collide with. In this case it may not be very efficient to create invisible walls as SDK objects, since their interactions would then be filtered out for everything except the characters. It is probably more efficient to create additional invisible walls as external obstacles, that only characters can interact with.
- the obstacles might be dynamic and updated with a variable timestep, while the SDK uses a fixed timestep. This could be for example a moving platform that characters can stand.

At the time of writing the character controller supports box and capsule objects, namely `PxBoxObstacle` and `PxCapsuleObstacle`. To create the `PxObstacleContext` object using the following function:

```cpp
PxObstacleContext* PxControllerManager::createObstacleContext()
```

Then manage obstacles with:

```cpp
ObstacleHandle PxObstacleContext::addObstacle(const PxObstacle& obstacle)
bool PxObstacleContext::removeObstacle(ObstacleHandle handle) = 0
bool PxObstacleContext::updateObstacle(ObstacleHandle handle, const char* name, PxObstacleFrameType type, PxObstacleHandle handle)
```

Typically `updateObstacle` is called right before the controllers' `move` call.

Using obstacles for moving platforms is illustrated in SampleBridges.
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* invoked - for both static and dynamic shapes. Various impact parameter callback, and they can then be used to do various things like playing trails, applying forces, and so on. The use of * PxUserControllerHitReport::onShapeHit* illustrated in SampleBridges. Note that this callback will only be called if a character moving against a shape. It will * not* be called if a (dynamic) shape collides against an otherwise non-moving character. In other words, this will only be called during a * PxController::move* call.

When the character hits another character, i.e. another object control controller, the * PxUserControllerHitReport::onControllerHit* callback happens when the player collides with an NPC, for example.

Finally, when the character hits a user-defined * PxUserControllerHitReport::onObstacleHit* callback is invoked.
Behavior Callback

The `PxControllerBehaviorCallback` object is used to customize the character's behavior after touching a `PxShape`, a `PxController`, or a `PxObstacle`. This is done using the following functions:

```cpp
PxControllerBehaviorFlags PxControllerBehaviorCallback::getBehaviorFlags(
    const PxShape& shape, const PxActor& actor) = 0;
PxControllerBehaviorFlagsPxControllerBehaviorCallback::getBehaviorFlags(
    const PxController& controller) = 0;
PxControllerBehaviorFlags PxControllerBehaviorCallback::getBehaviorFlags(
    const PxObstacle& obstacle) = 0;
```

At the time of writing the following returned flags are supported:

`PxControllerBehaviorFlag::eCCT_CAN_RIDE_ON_OBJECT` defines if the character can effectively travel with the object it is standing on. For example a character standing on a dynamic bridge should follow the motion of the `PxShape` it is standing on (see 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 contribute to it: the `step_offset` used to automatically walk over small bumps, the vertical 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 or not when standing on the object. This can be used as an alternative to the previous slope limit feature, to define non walk-able objects rather than non-walkable parts. 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_USERDEFINED_RIDE` simply disables all built-in code related to controllers riding on objects. This can be useful to get the legacy behavior back, which can sometimes be necessary when porting to PhysX 3.x a piece of code.
around the PhysX 2.x character controller. The flag simply skips the
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, etc.), so the forces computed by the physics engine between a bounding volume and surrounding objects will not be realistic anyway. They will not properly model the interaction between an actual character and these objects. If the bounding volume is large compared to the visible character, maybe to make sure that its limbs never penetrate the static geometry around, the dynamic objects will start moving (pushed by a 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 box controller to 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, the applied force tends to 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 specific game code, to implement a specific solution for a specific game. This coupling 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 Sample Bridges.

Note that the character controller does use overlap queries to determine which shapes are nearby. Thus, SDK shapes that should interact with the characters (e.g., the character should push) must have the `PxShapeFlag::eSCENE_QUERY_SHAPE` flag set to true, otherwise the CCT will not detect them and characters will move right through 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 CCT module should be allowed to move them.

The \textit{PxControllerFilterCallback} object is used to define basic interactions between characters. Its \textit{PxControllerFilterCallback::filter} function can be used to determine if two PxController objects should collide at all with each other:

\begin{verbatim}
bool PxControllerFilterCallback::filter(const PxController& a, const PxController& b) {
// implementation...
}
\end{verbatim}

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 might be allowed to go through each other only for a limited amount of time. When that limited time expires, the characters may be left in an overlapping state until they start moving again towards each other. To automatically separate overlapping characters, the following function can be used:

\begin{verbatim}
void PxControllerManager::computeInteractions(PxF32 elapsedTime, PxControllerFilterCallback* cctFilterCb=NULL) = 0;
\end{verbatim}

This function is an optional helper to properly resolve overlaps between characters. It should be called once per frame, before the \textit{PxController::move} calls. The function does not move the characters directly, but it will compute overlap information for each object and will be used in the next \textit{PxController::move} call.
Hidden Kinematic Actors

The CCT library creates a kinematic actor under the hood, for each controlled character. When invoking the `PxController::move` function, the underlying hidden kinematic 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, users might get back these potentially confusing unknown actors from scene-level collision queries.

One possible strategy is to retrieve the controllers' kinematic actors using the following function:

```cpp
PxRigidDynamic* PxController::getActor() const;
```

Then mark these actors with a special tag, using the `PxRigidDynamic::userData` way the CCT actors can easily be identified (and possibly ignored) in collision queries or 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 troublesome because the PxController objects are otherwise often upd time steps (typically using the elapsed time between two rendering fram

Thus the PxController objects (using variable time steps) may not alw sync with their kinematic actors (using fixed time steps). This phenor SampleBridges.
Invalidating Internal Geometry Caches

The CCT library caches the geometry around each character, in order to speed up collision queries. The temporal bounding box for a character is an AABB around 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 defined for each character by `PxControllerDesc::volumeGrowth`. Each time a character moves, its temporal bounding box is tested against the cached volume of space. If the motion of the character is fully contained within that volume of space, the contents of the cache are reused instead of being regenerated through PxScene-level queries.

In PhysX 3.3 and above, those caches should be automatically invalidated when a cached object gets updated or removed. However, it is also possible to manually flush those caches using the following function:
void PxController::invalidateCache();

Prior to deciding if a character will travel with the motion of an object that is touching the character, a number of tests are automatically performed to decide if the cached touched object remains valid. These automatic validity tests mean that in the following cases 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, that the character no longer travels with the motion of that cached object, it is necessary to invalidate the cache. This holds true if the pair have separated as a consequence of an updated global pose or modified geometry.
Runtime Tessellation

The CCT library is quite robust, but sometimes suffers from FPU accuracy issues when a character collides against large triangles. This can lead to characters not sliding against those triangles, or even penetrating them. One way to effectively solve these problems is to tessellate the large triangles at runtime, replacing them on-the-fly with a collection of smaller triangles. The library supports a built-in tessellation feature, enabled with this function:

```cpp
void PxControllerManager::setTessellation(bool flag, float maxEdgeLength)
```

The first parameter enables or disables the feature. The second parameter defines the maximum allowed edge length for a triangle, before it gets tessellated. A smaller edge length leads to more triangles being created at runtime, and 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 if experiencing collision problems. When enabling the feature, it is recommended to use the largest possible `maxEdgeLength` that does fix encountered problems.
In the screenshot, the large magenta triangle on which the character is standing is replaced with the smaller green triangles by the tessellation module. The internal geometry cache is represented by the blue bounding box. Note that only the green triangles touching this volume of space are kept. Thus, the exact number of triangles 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 CCT library.

Character goes through walls in rare cases

1. Try increasing \texttt{PxControllerDesc::contactOffset}.
2. Try enabling runtime tessellation with \texttt{PxControllerManager::setTessellation} a small \textit{maxEdgeLength} first, to see if it solves the problem. Then as much as possible.
3. Try enabling overlap recovery module \texttt{PxControllerManager::setOverlapRecoveryModule}.

Tessellation performance issue

1. Try fine-tuning the \textit{maxEdgeLength} parameter. Use the largest possible value that still prevents tunneling issues.
2. Try reducing \texttt{PxControllerDesc::volumeGrowth}.

The capsule controller manages to climb over obstacles higher than the step offset value

1. Try using \texttt{PxCapsuleClimbingMode::eCONSTRAINED}.
Particles (deprecated)
Introduction

The PhysX particle feature has been deprecated in PhysX. The standalone library PhysX FleX is an alternative with a richer feature set.

PhysX 3 offers two particle system types - a generic particle system and an SPH fluid particle system. The generic particle system provides basic particle motion with rigid actors. It can be used for objects that require collisions against the environment, but for which inter-particle interactions are not needed. Examples include small debris, sparks or leaves. The SPH fluid particle system can be used for fluid effects that require approximate incompressibility and flowing behavior, such as liquids or smoke filling up a volume.

PhysX 3 takes care of collision detection and particle dynamics, while auxiliary facilities such as emitters, lifetime maintenance etc. need to be provided by the application.
Creating Particle Systems

Both particle system classes `PxParticleSystem` and `PxParticleFluid` derive from `PxParticleBase`, which is the common interface providing particle manipulation and collision functionality. Particle systems inherit from `PxActor` and can be added to a scene.

![Particle System Class Diagram]

Figure 1: PxParticleSystem inherits all properties from PxParticleBase and adds fluid specific properties.

The following section shows how a particle system is created and added:

```cpp
// set immutable properties.
PxU32 maxParticles = 100;
bool perParticleRestOffset = false;

// create particle system in PhysX SDK
PxParticleSystem* ps = mPhysics->createParticleSystem(maxParticles);

// 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. `PxCreatePhysics` registers all modules by default as opposed to `PxCreateBasePhysics`. 
Particle Management

Particle systems reserve memory for a fixed number of particles. Each of these particles can be addressed throughout its lifetime. The given range of particles is marked as being valid or invalid. This is achieved by two means: the particle range indicates the range within which particles may be valid, 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 consists of an array of 32-bit unsigned integers with enough elements to cover the valid particle range.

Creating Particles

The application specifies an index for each new particle at particle creation time. If the application maintains its own representation of particles, and already tracks these indices may be re-used by PhysX. If the application does not have 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 to create particles with initial properties. When creating particles, specifying indic
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 of the iterated data and specifying the pointer to the first element.

Example for creating a few particles:

```cpp
// 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>(newAppParticleIndices);
particleCreationData.positionBuffer = PxStrideIterator<const PxVec3>(newAppParticlePositions);
particleCreationData.velocityBuffer = PxStrideIterator<const PxVec3>(newAppParticleVelocities);

// create particles in *PxParticleSystem* ps
bool success = ps->createParticles(particleCreationData);
```

The indices specified for particle creation need to be unique and within the limit of `PxParticleBase::getMaxParticles()`.

In this example the stride iterator is used to set the same velocity for all new particles. This is achieved by setting the stride to zero.

**Note:** For fluid particles it is necessary to spawn particles at distances close to `PxParticleFluid::getRestParticleDistance()` in order to achieve a regular emission, otherwise particles will spread immediately in all directions.

**Note:** In PhysX 3 all particle access such as creating, releasing, updating and reading particles can only be carried out while the simulation of the scene is not running.

**Releasing Particles**
Particles can be released by providing indices to the particle system. As opposed to older versions of the PhysX SDK, particles get immediately released.

Example for releasing a few particles:

```cpp
// 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:

```cpp
ps->releaseParticles();
```

Since only a limited number of particle slots (`PxParticleBase::getMaxParticles()`) 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. Other reasons to release particles:

- 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 cannot be covered are marked and should be released. See *Particle Grid*.

**Index Pool Extension**

Example for allocating particle indices using the PhysX extensions library:

```cpp
// create an index pool for a particle system with maximum particle count
PxParticleExt::IndexPool* indexPool = PxParticleExt::createIndexPool(maxParticles);

// use the indexPool for allocating numNewAppParticles indices that...
```
// 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 specified by `PxForceMode`.

Example for force update:

```cpp
// 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 system by default. The application can specify the data it needs by configuring `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. May only be enabled if the particle system was created with per particle rest offset (`Creating Particle Systems`).  
- `PxParticleReadDataFlag::eCOLLISION_NORMAL_BUFFER`: Off by default.  
- `PxParticleReadDataFlag::eDENSITY_BUFFER`: Only available for particle fluids and off by default.

Particle flags provide more information on individual particles:

- `PxParticleFlag::eVALID`: If set, the particle was created before released. If not set, the particle slot does not contain a valid 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 data structure (Particle Grid).

Particle collision normals represent contact normals between parti...
surfaces. A non-colliding particle has a zero collision normal. Collision normals can be useful e.g. for orienting the particle visualization according to their contact with rigid actors.

Particle densities provided by particle fluids can be used for rendering: A particle that is completely isolated has a value of zero. A particle that has a particle neighborhood with a mean spacing $\text{PxParticleFluid::getRestParticleDistance()}$.

Particle data can only be read while the scene simulation is not executing. In order to get access to the SDK buffers, a `PxParticleReadData` instance needs to be acquired from the SDK. It has the following properties:

- `numValidParticles`: Total number of valid particles for the corresponding particle system.
- `validParticleRange`: The index range of valid particles in the particle buffers.
- `validParticleBitmap`: Bitmap of valid particle locations.
- `positionBuffer`, `velocityBuffer`, `restOffsetBuffer`, `flagsBuffer`, `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:

```cpp
// lock SDK buffers of *PxParticleSystem* ps for reading
PxParticleReadData* rd = ps->lockParticleReadData();

// access particle data from PxParticleReadData
if (rd)
{
    PxStrideIterator<const PxParticleFlags> flagsIt(rd->flagsBuffer)
    PxStrideIterator<const PxVec3> positionIt(rd->positionBuffer)

    for (unsigned i = 0; i < rd->validParticleRange; ++i, ++flagsIt)
    {
        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 without the locking and unlocking):

```cpp
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 specified when the particle system is created and cannot be changed afterwards. Some are mutable while the particle system is not part of a scene and others can be changed at any time. The following description covers parameter that either cannot be set at any time, or may 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 will be. Can only be set on particle system creation.

*PxParticleReadDataFlags:*

Specifies a subset of simulation properties which are returned to the application after simulation. See Reading Particles. As few read data flags should be set as possible in order to save memory and improve performance by avoiding unnecessary particle 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 proximity tests and parallelization. See Particle Grid. Parameter can only be changed while 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. Changing this 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 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 too much. 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 effect. 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. This is particularly useful when using particles for smoke to prevent ballistic behavior of individual particles which can look odd.
particleMass:

Mass used for two way interaction with rigid bodies (PxParticleBaseFlag::eCOLLISION_TWOWAY) and different force modes in the context of PxParticleBase::addForces. This mass property doesn't have any impact 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 useful 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 the 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 if particles represent objects of significantly varying size, for example in the context of debris effects. See Per-particle Rest Offsets. Can only be set on particle system creation.

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 larger 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 of rigid actors.

**dynamicFriction:**

Dynamic friction used for particle collision. This parameter defines how easily particles slide over rigid actor surfaces. The lower the value is to 0, the easier 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 larger than one 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 interaction 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 part of a scene.

**Fluid (PxParticleFluid)**

The SPH simulation can be tricky to tweak for good results. As this simulation technique (see References) uses an explicit integration scheme it only provides stable results within 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). The suggested starting points for parameter values below assume a time step size of about 1/60 [s] and a gravity around 10 [m/s^2]. Using a damping value larger than zero provides a larger parameter sub-space, for example useful when implementing a smoke effect.
restParticleDistance:

Defines the resolution of the particle fluid. It defines the approximate distance neighboring particles will adopt within a fluid volume at rest. For the particle tweaking assumption mentioned above, the particle rest distance should not be smaller than 0.05 [m]. Parameter can only be changed while particle system 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 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 within 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 belong to. They will attempt to maintain a fixed distance from these shapes as specified by `PxParticleBase::setRestOffset()`.

Collision Filtering

Filtering particle versus rigid body collisions can be useful to avoid unnecessary performance overhead or simply to avoid undesired collisions.

For the following examples filtering is useful:

- Avoid particles colliding with trigger shapes (this is already the behavior of `PxDefaultSimulationFilterShader`)
- Configure a drain shape to exclusively collide with particles
- Have particles collide with a proxy shape as opposed to the shape used for rigid body collisions

Filter information for particles can be specified by calling `PxParticleBase::setSimulationFilterData()`. Instructions for how to setup filter shaders can be found here: Collision Filtering.

Per-particle Rest Offsets

It is also possible to set a rest offset per-particle, using `PxParticleBase::setRestOffsets()`. In order to provide per-particle offsets, `PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET` needs to be set and the rest offsets must be smaller than the per-system value given by `PxParticleBase.getRestOffset()`.

Particle Drains
Using drains is a good method for keeping the particle count and spread under control. Placing drains around the area of interest in which a particle system is used helps to maintain good performance of the particle simulation. The area of interest could, for example, also be moved with the player.

Example of how to flag a `PxShape` `rbShape` as a drain:

```cpp
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 system into groups. This is done to accelerate proximity queries and for parallelization purposes. The grid size parameter needs to be experimentally adjusted with `PxParticleBase::setGridSize()` for best performance. When doing this, it is helpful to visualize the grid using `PxVisualizationParameter::ePARTICLE_SYSTEM_GRID`.

Grid size values might result in spatial data structure overflow, since the number of grid cells is limited to about 1000. Large grid size values on the other hand might result in poor performance due to ineffective spatial queries or lack of parallelization opportunities.

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 gain, `physx::PxGpuDispatcher` for the scene we want to add the particle system.

```cpp
#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 using `PxParticleBaseFlag::eGPU`. Toggling GPU acceleration while the particle system is part of a scene might have a bad impact on performance since its state needs to be copied to or from the GPU device memory. It is therefore better to set `PxParticleBase::setParticleBaseFlag()` before adding the particle system to the scene.

Particle data can be read directly from the GPU using `PxParticleBase::lockParticleReadData(PxDataAccessFlag::eDEVICE)` and `PxParticleFluid::lockParticleFluidReadData(PxDataAccessFlag::eDEVICE)` used to render particles directly with CUDA Graphics Interop.

Convex, Triangle and Height field meshes are automatically mirrored in the GPU 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 prevented by mirroring the meshes explicitly, as shown in this example:
On Kepler and above GPUs, the triangle meshes can be cached to achieve better performance. The amount of memory to be allocated for caching can be set using:

```cpp
PxParticleGpu::setTriangleMeshCacheSizeHint(const class PxScene& scene);
```

The triangle mesh cache will be shared among all the particle systems in the scene. The optimal size depends on the scene (i.e. triangle mesh density and particle distribution). The cache usage statistics can be queried and analyzed to fine-tune the cache size hint.
Sample Discussion

The SampleParticles shows both particle system types being used:\( \text{PxParticleSystem} \) used for small debris and smoke, while \( \text{PxParticleFluid} \) is used for a waterfall. The sample provides example implementations of various aspects described in this guide:

- \textit{SampleParticles::createParticleSystem}, \textit{SampleParticles::createFluid} create particle systems.
- \textit{ParticleSystem::createParticles} creates particles within a particle system.
- \textit{ParticleSystem::update} shows how to read, update, release particles and deal with spatial data structure overflows.
- \textit{SampleParticlesFilterShader} is an example for setting up collision filtering.
- \textit{SampleParticles::createDrain} shows how to setup a rigid body shape as a particle drain.
- \textit{SampleBase::onInit} illustrates how to setup GPU/CUDA acceleration.

The sample makes use of various helper classes:

- \textit{ParticleSystem}: Encapsulates a \textit{PxParticleSystem} or \textit{PxParticleFluid} manages application side data such as particle lifetimes and orientations. It facilitates creating and releasing particles and double buffers particle data for asynchronous rendering.
- \textit{RenderParticleSystemActor}: Owns a \textit{ParticleSystem} and provides rendering functionality.
- \textit{ParticleEmitterRate}: Emits particles at a specified rate (\#particles per second).
- \textit{ParticleEmitterPressure}: Emits particles maintaining a certain distance between them.
- \textit{SampleParticles::Emitter}: Connects an emitter as described in \textit{RenderParticleSystemActor}.
- \textit{SampleParticles::Raygun}: Provides functionality for the ray force field and 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 collide with the scene, which can be seen when roaming the smoke with the ray-gun. The realism of the smoke effect could be increased by using a particle fluid in order to get the smoke volume to expand. This is typically useful for indoor scenes or ground fog like effects where the particles get into pooling situations.

Two kinds of debris are shown in the sample. Larger chunks of debris are represented using convex-shaped rigid bodies. Smaller but more abundant chunks are represented by particles, which helps performance. The particle based debris is rendered using instanced 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 trick 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::initializeParticlesOrientations` and `ParticleSystem::modifyRotationMatrix`. 
References

*Particle-Based Fluid Simulation for Interactive Applications*
Matthias Muller, David Charypar and Markus Gross, Eurogrphics/Siggr Breeen, 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
Introduction

The PhysX clothing feature has been deprecated in PhysX version 3.4.1. The PhysX and APEX clothing features are replaced by the standalone NvCloth library.

Realistic movement of character clothing greatly improves the player experience. The PhysX 3 cloth feature is a complete and high-performance solution to simulating character clothing. It provides local space simulation for high accuracy and stability, new techniques to reduce stretching, collision against a variety of shapes, as well as particle self- and inter-collision to avoid the cloth penetrating itself or other cloth instances. The simulation can be offloaded to CUDA capable GPUs for better performance or to run assets at higher resolutions than the CPU is able to handle.

PhysX 3 cloth is a rewrite of the PhysX 2 deformables, tailored towards character cloth. Softbodies, tearing, and two-way interaction have been removed, while 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 simulation time. Each iteration integrates particle positions, solves constraints, performs character and self-collision. Cloth inter-collision is performed once per-frame after all cloth instances in the scene have been stepped forward. Local frame, motion constraints and collision shapes are interpolated per iteration from the per-frame values specified by the user.

**Solver Frequency**

The size of the iteration time step is inversely proportional to the number of iterations:

```java
cloth.setSolverFrequency(240.0f);
```

The solver frequency is specified as iterations per second. A solver frequency value of 240 corresponds to 4 iterations per frame at 60 frames per second. In general, simulation becomes more accurate if higher solver frequency value is used. However, the simulation time grows roughly linearly with solver frequency. Typically this value is between 120 and 300.

The number of iterations for each frame is derived using the simulation frequency and the simulation time-step. PhysX tries to handle variable time-steps carefully by taking variations of the time-step into account during position integration and when applying damping parameters like constraint stiffness. While this reduces the appearance of artifacts due to varying time step sizes, use of variable time step sizes is generally not recommended.

**Particle Integration**

The first step in a cloth iteration predicts the new particle position based on its current position, velocity and external acceleration. While a particle state consists of the current 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 iteration.

**Local Space Simulation and Inertia Scale**

Each PxCloth actor has a transformation that transforms particles from local space to world space positions. For example:

```cpp
cloth.setGlobalPose(PxTransform(PxVec3(1.0f, 0.0f, 0.0f)));```

will change the cloth's world space position to (1,0,0). Now compare that to:

```cpp
cloth.setTargetPose(PxTransform(PxVec3(1.0f, 0.0f, 0.0f)));```

which also changes the cloth's position to the same place. So what's different?

*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 an example to impart half the local frame acceleration to the particles use:

```cpp
cloth.setInertiaScale(0.5f);```

Scaling inertia effects individually per translation and rotation axis is also possible through the family of *PxCloth::set*InertiaScale() methods. Limiting the amount local frame accelerations affect particles can be especially useful for fast moving characters.

**Note:** Using setGlobalPose() is equivalent to using setTargetPos() when inertia scale is 0. In this case, the cloth does not receive any acceleration due to frame changes.

**Constraints**

After the particle positions have been integrated, a set of different constraints are solved to simulate stretch, shear and bending forces, as well as to confine the particles within a certain region.
One of the most important roles for the cloth solver is to maintain particles so that the cloth does not stretch. This is achieved by constraints between pairs of particles. The way particles are connected, cloth stretches, compresses, shears, rotates, and bends. PhysX classifies distance constraints into 4 types (see `PxClothFabricPhaseType`), each of which with different stiffness parameters.

Below is an example of stiffness settings for each constraint type:
Sometimes it is desirable that distance constraints are not enforced rigorously. The stiffness parameter allows only correcting a portion of the edge length per iteration, for example to reduce the strength of bending constraints. A separate, lower stiffness 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 steps, but still behave correctly during pirouettes.

The following code sets up the vertical constraints such that when edges are compressed more than 60% or stretched more than 120%, a stiffness of 0.8 will be used, otherwise a stiffness of 0.4 = 0.8 * 0.5 will be used:

```cpp
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 specified separately. This can be used to handle stretching along the gravity (vertical) direction.

---

**Tether Constraints**
The distance constraints are solved only once per iteration without convergence. The most visible artifact of this approximation is that the cloth becomes stretchy. Increasing solver frequency reduces the stretching, but results in increased simulation time.

PhysX 3.3 introduces tether constraints as a solution to avoid stretching under gravity or fast motion. Tether constraints prevent stretching by limiting the distance a particle can move away from their anchor particles. This constraint adds very small computation to the solver, so it is more effective than increasing the number of iterations.

The tether constraints are automatically generated by the cooker when some PxClothMeshDesc::invMasses values are set to zero, telling the cooker that the corresponding particles are non-simulated anchor particles whose positions are provided solely from users. Changing inverse masses after the fabric has been created does not 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; however, it is sometimes desirable to confine a point within small region around 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, and animation system can sketch the overall movement of a cloth while the fine scale 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:

```cpp
PxClothParticleMotionConstraints motionConstraints[] = {
    PxClothParticleMotionConstraint(PxVec3(0.0f, 0.0f, 0.0f), 0.0f),
    PxClothParticleMotionConstraint(PxVec3(0.0f, 1.0f, 0.0f), 1.0f),
    PxClothParticleMotionConstraint(PxVec3(1.0f, 0.0f, 0.0f), 1.0f),
    PxClothParticleMotionConstraint(PxVec3(1.0f, 1.0f, 0.0f), FLT_MAX)
};

cloth.setMotionConstraints(motionConstraints);
```

If the sphere radius becomes zero or negative, the corresponding particle is locked at the sphere center and the inverse particle mass is set to zero for the next iteration. In the above example, the first particle will fully lock to the constraint position, while the second and third particles will remain within the sphere radius. The last particle will not be constrained.
The motion constraint sphere radius can be globally scaled and biased, for example, to transition between simulated and animated states. See PxClothMotionConstraint details.

**Separation Constraints**

![Figure 6. Example separation constraint](image)

Separation constraints work exactly the opposite way to the motion constraints, forcing a particle to stay outside of a sphere. When particle movement is moderately constrained by motion constraints (e.g., sleeves around an arm), separation constraints can represent the character's collision shape more accurately than using capsules alone. For example, separation constraints can be placed slightly inside the character's collision shape more accurately than using capsules alone. The radius of the separation constraint can be set to be the distance from the sphere center to the surface of the character. See PxClothParticleSeparationConstraint and PxCloth::setSeparationConstraints.

**Collision Detection**

Each cloth object supports collision with spheres, capsules, planes, convexes (groups of planes) and triangles. By default, these shapes are all treated separately from the main PhysX rigid body scene, however, collision against other PxCloth::setSeparationConstraints::eSCENE_COLLISION flag.

Collision shapes are specified in local coordinates for the next frame before simulating the scene. An independent and complete collision stage is performed as part of each solver iteration, using shape positions interpolated from the values at the beginning and the end of each solver iteration.
of the frame. Sphere and capsule collision supports continuous collision
an acceleration structure to cull far-away particles early in the collision
and capsules are therefore the preferred choice to model the cha
convexes and triangles should only be used sparingly.

Spheres are defined as center and radius. Note that the radius is spe
change from frame to frame. The total number of spheres is limited to 3.

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
multiple capsules, which can be useful for modeling characters (upper a
up from capsules can share the sphere at the knee). Sharing of spher
simulation more efficient and robust, so is highly encouraged.

Figure 7. A tapered capsule collision shape formed by two conne

Figure 8. A leg shape formed by using two tapered capsules, each shar

Planes are defined by their normal and distance to origin. They will no
collision unless they are referenced by a convex shape. Convexes refe
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 winding order. The triangles should form a closed patch near the cloth for consistent collision handling. Each particle collides against its closest triangle expanded to an infinite plane.

The order of planes and triangles should remain unchanged (apart from removing them through the `PxCloth::removeCollisionPlane/Triangle()` method) as their positions are interpolated between simulation frames.

**Continuous Collision Detection**

Besides discrete collision which resolves particles inside shapes at the end of each iteration, continuous collision detection is supported and can be enabled.

```cpp
// 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 contact occurs. After the first time of contact, the particle is moved with the shape until the end of the iteration.

**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 collide with 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 explicitly stored like regular particles, they can be thought of as providing additional samples on the collision surface.
During collision processing each virtual particle is created from three normal particles using barycentric interpolation. It is then tested for discrete collision like a regular particle and the collision impulse is redistributed back to the original particles using reverse interpolation.

Section *Adding Virtual Particles* explains the necessary steps to use this feature.

**Friction and Mass Scaling**

Coulomb friction can be enabled and will be applied for particle and virtual particle collisions by setting a friction coefficient between 0 and 1:

```java
cloth.setFrictionCoefficient(0.5f);
```

Additionally, there is an option to artificially increase the mass of colliding particles. The temporary increase in mass can help reduce stretching along edges that are being pulled over a collision shape. The effect is determined by the relative velocities of the particle and collision shape and a user-defined coefficient. A value of 20 is a reasonable starting point but users are encouraged to experiment with this value:

```java
cloth.setCollisionMassScale(20.0f);
```

**Self-Collision of a Single Cloth Actor**

The particles of a cloth actor can collide among themselves. To enable this behavior, one should set both self-collision distance and self-collision stiffness to non-zero values:

```java
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 two particles in the rest configuration. If the distance is larger, self-collision may violate distance constraints and result in jittering.

When such a configuration cannot be avoided (e.g. due to irregular input meshes), 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 rest positions will significantly degrade performance of self-collision, so should be used sparingly.

Self-collision performance for high-resolution cloth instances can be improved by limiting 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 of a scene:

```
scene.setInterCollisionDistance(0.5f);
scene.setInterCollisionStiffness(1.0f);
```

The definition of distance and stiffness values are the same as self-collision. Cloth instances that specify a particle subset for self-collision use the same subset for inter-collision.
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, there is a performance drop somewhere below 3000 particles for the GPU solver as explained in the next section. As a rough guideline, a dozen cloth instances with 2000 particles each 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 structures 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 indices 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 that no supported GPU is available PhysX will issue a warning and subsequent simulations will be run on CPU.

When the cloth is simulated using CUDA, the GPU simulation results can be integrated into the graphics API by requesting CUDA device pointers to the particle data:
To take full advantage of the GPU hardware there should be at least as many cloth instances as streaming multiprocessors (SMs). This means it is generally 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 can fit in shared memory. The number of particles that fit into shared memory depends on the number of collision shapes, whether continuous collision or self-collisions are enabled, and the GPU version. For GPUs supporting SM 2.0 and above, about 2500-2900 particles can fit into shared memory. If particles don't fit into shared memory the cloth solver will automatically stream particles through global memory, which incurs some performance cost.

Furthermore, the limited size of shared memory requires the number of collision triangles to be clamped to 500 when GPU simulation is enabled.

**Fast-Moving Characters**

Consistent collision handling for fast-moving characters can be difficult and rotations are best handled by tying the cloth local simulation frame to the character transformation. The inertia effects of the local frame transformations can be fine-tuned using the inertia scale settings.

If the cloth tunnels collision shapes during fast character animations, increasing the solver frequency or enabling swept contacts (see `PxClothFlag::eSWEPT_CONTACT`) can help.

**Avoiding Stretching**

Due to the iterative nature of the distance constraint solver, high resolution cloth can stretch undesirably under strong gravity even if the stretch stiffness is set to one. Increasing the solver frequency mitigates the stretching, but tether constraints are usually better suited to eliminate stretching efficiently.
Avoiding Jitter

Under certain configurations, different constraint types can violate each other and constrain the particle positions. For example, a motion constraint can move the particle further from the anchor particle than the tether constraint permits, or particles can get pinched between two overlapping collision shapes. Over-constraining can result in jitter and should be avoided. In some situations jitter can be avoided by increasing the solver frequency or by reducing the corresponding constraint stiffness.

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 access to the mesh used to create the fabric, and this mesh can't be displayed in PVD either. You can display individual sets of distance constraints instead of all at once by setting the Mode to Single Phase in the Preferences dialog and using the Cloth Phase phase to display. The Particle Scale slider in the same dialog affects the rendering size of ordinary and virtual cloth particles as well. All properties of a selected cloth object can be viewed in the Inspector panel of PVD.
Snippet Discussion

The following paragraph describes code of the cloth snippet provided with the PhysX SDK.

The cloth constraint connectivity and rest values are stored in a fabric instance (PxClothFabric), separate from the cloth actor (PxCloth). The separation from particles allows the same fabric data to be reused for multiple cloth instances, reducing cooking time and storage requirements. PxClothFabricCreate library, creates a fabric from a triangle or quad mesh (see PxClothMeshDesc actor itself is created through the physics instance (PxPhysics) and needs to be added to a scene (PxScene) in order to be simulated. Once the cloth actor is assigned simulation settings such as collision data, constraint stiffness, solver frequency and self-collision. The createCloth function in the cloth snippet performs these steps.

The stepPhysics function advances the simulation by one frame. It first updates the cloth local frame, which rotates around the y-axis. The collision shapes are not moving in scene coordinates, but their positions are specified in cloth local coordinates, and therefore need to be updated every frame. The following sections detail some of the available parameters and show how to configure them.

**Note:** The cloth module has to be registered with PxRegisterCloth on platforms with static linking (non windows) before creating cloth objects. PxCreatePhysics modules by default as opposed to PxCreateBasePhysics.

Filling in PxClothMeshDesc

The first task to create a cloth is to fill in the PxClothMeshDesc structure programmatically creates a regular grid of cloth particles connected. Below is a simpler example on how to create a cloth from a simple mesh with a single quad.

```cpp
 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. Setting the inverse mass to zero indicates that the particle is not simulated. Instead, the particle is fixed in local space or kinematically constrained to user specified positions. The inverse mass of simulated particles can normally be set to any fixed positive value.

The `PxClothMeshDesc` structure allows positions and inverse masses to be stored in separate arrays or interleaved like in the code above. The mesh can consist of quads or triangles, or both. The cooker prefers quad meshes over triangle meshes when creating constraints and 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 extensions library wraps the generation of constraints and the creation of the `PxClothFabric` structure:

```
PxClothFabric* fabric = PxClothFabricCreate(physics, meshDesc, PxVec3(0.0f, 0.0f, -9.81f));
```

The third parameter indicates the direction of gravity, which is used as the direction of 'horizontal' or 'vertical' constraints.
The \textit{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 \textit{PxCloth} object is created using a fabric instance and the initial particle configuration. Like all actors, the cloth instance is simulated as part of a scene:

```cpp
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 fabric instance created by the cooker. The third input provides initial particle positions and inverse masses. Typically this array is the same as the one referenced by the mesh used to create the fabric. Note that the rest configuration (such as the distance constraint) is computed from \textit{PxClothMeshDesc}, so the initial positions do not affect rest configuration. The last parameter is a set of flags that allow GPU simulation and continuous collision detection to be enabled. The default is to turn both options off.

**Specifying Collision Shapes**

The following code illustrates how to add two spheres of different radii and a tapered capsule between them:

```cpp
// 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 but will not be considered for collision unless they are referenced by a convex shape. The following code shows how to setup a typical upward facing ground plane through the origin:

```cpp
cloth.addCollisionPlane(PxClothCollisionPlane(PxVec3(0.0f, 1.0f, 0.0f)), cloth.addCollisionConvex(1 << 0)); // Convex references the first plane
```

Planes may be efficiently updated after construction using the `PxCloth::setCollisionPlanes()` function.

Finally, triangles are added using the `PxCloth::setCollisionTriangles()` example, the following code adds a tetrahedron made of four triangles:

```cpp
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, 0.0f), PxVec3(0.0f, 1.0f, 0.0f)),
    PxClothCollisionTriangle(PxVec3(0.0f, 0.0f, 1.0f), PxVec3(0.0f, 1.0f, 0.0f), PxVec3(1.0f, 0.0f, 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 `createCloth` and then updates collision spheres, planes and triangles every frame in the `updateCloth` function.

**Adding Virtual Particles**
Figure 9. Four virtual particles (green) expressed as the weighted combination of a triangle's particles, virtual particles provide a better sampling of the cloth geometry that 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 weights 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 from 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 therefore doesn't read back any particle data. The lockParticleData() provides read and optionally write access to the particle positions of the current and previous iteration. As an example, the following code applies some external acceleration to each particle:

```cpp
PxClothParticleData* data = cloth.lockParticleData(PxDataAccessFlag::eREAD_WRITE);
float dt = cloth.getPreviousTimeStep();
for(PxU32 i = 0, n = cloth.getNbParticles(); i < n; ++i) {
    data->previousParticles[i].pos -= particleAccelations[i] * dt;
}
data->unlock();
```
References


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, PhysX provides an interface to extract visual debug information as a set of basic rendering primitives: points, lines, triangles, and text. These primitives can then be rendered over the application render objects.
Usage

To enable debug visualization, the global visualization scale has to be set to a positive value first:

```cpp
PxScene* scene = ...
scene->setVisualizationParameter(PxVisualizationParameter::eSCALE,...)
```

Then the individual properties that should be visualized can be enabled using a positive value:

```cpp
scene->setVisualizationParameter(PxVisualizationParameter::eACTOR_AXES,...)
```

In the example, the actor world axes will be visualized. The scale used will be the product of the global scale (1.0 in this example) and the property scale (2.0 in this example). Please note that for some properties the scale factor does not apply. For example, shape geometry, will not be scaled since the size is defined by the user application.

Furthermore, for some objects, visualization has to be enabled on the corresponding object instances too (see `PxActorFlag::eVISUALIZATION`, `PxShapeFlag::eVISUALIZATION`, ...).

After a simulation step, the visualization primitives can then be extracted:

```cpp
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 efficiently for large scenes. In cases where only a localized area is of interest, there is the option to use a culling box for debug visualization via `PxScene::setVisualizationCullingBox()`.
Note that simply enabling debug visualization (PxVisualizationParam::eSCALE) can have a significant performance impact, even when all the other individual visualization flags are disabled. Thus, make sure debug visualization is disabled in your final/release builds.
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/si 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 configurations of the SDK. 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 supports reading the stream 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 the convenient way to collect PVD data. In this mode the stream can be watched in real-time, depending on network speed and scene complexity. In network mode PVD acts as a TCP/IP server and must therefore be launched before the SDK tries to connect to it. The default listening port is 5425:

```cpp
use namespace physx;

PxPvd* pvd = PxCreatePvd(*foundation);
PxPvdTransport* transport = PxDefaultPvdSocketTransportCreate(PVD_HOST);
pvd->connect(*transport, PxPvdInstrumentationFlag::eALL);

PxPhysics* physics = PxCreatePhysics(PX_PHYSICS_VERSION, *gFoundation);

//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 fall-back in case your platform or system setup does not support a network connection. Streams are often faster than network sockets and therefore a good alternative if performance is more important than real-time viewing. Streams store
use namespace physx;

PxPvd* pvd = PxCreatePvd(*foundation);
PxPvdTransport* transport = PxDefaultPvdFileTransportCreate(filename);
pvd->connect(*transport,PxPvdInstrumentationFlag::eALL);

PxPhysics* physics = PxCreatePhysics(PX_PHYSICS_VERSION, *gFoundation);

//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. This has both influence on PVD's and the SDK's performance:

- **PxPvdInstrumentationFlag::eDEBUG**: Transfer all debug data to visualize and inspect objects. This flag has usually the biggest impact on the stream size.
- **PxPvdInstrumentationFlag::ePROFILE**: Transfer timing information of various profiling zones in our SDK.
- **PxPvdInstrumentationFlag::eMEMORY**: Transfer memory usage data of our SDK.

Setup to transfer only profiling data over network:

```cpp
pvd->connect(*transport, PxPvdInstrumentationFlag::ePROFILE);
```

Visualizing Externals and Extended Data

Joints are implemented as an extension to the SDK constraints and therefore need special handling to get transmitted to PVD. Both joint and contact data can increase the stream size significantly. Visualizing it in PVD is therefore disabled by default. To enable them use the following API calls:

```cpp
mScene->getScenePvdClient()->setScenePvdFlags(PxPvdSceneFlag::eTRANSMIT_CONSTRAINTS);
```

or set the flags separately:

```cpp
mScene->getScenePvdClient()->setScenePvdFlag(PxPvdSceneFlag::eTRANSMIT_CONSTRAINTS);
```

Visualizing SceneQuery
Visualizing SceneQuery in PVD is disabled by default since queries increase the stream size significantly. To enable it use following API call:

```cpp
mScene->getScenePvdClient()->setScenePvdFlag(PxPvdSceneFlag::eTRANSMIT_SCENEQUERIES);
```

**Custom PvdClient**

Implement the PvdClient interface if your application needs to react upon disconnection from PVD, or if you plan to send custom PVD events from your application. It is recommended to toggle the contact and constraint visualization in the onPvdConnected/onPvdDisconnected callbacks to avoid potential memory and compute overhead in the SDK:

```cpp
// derive from PvdClient
struct MyPvdClient : public physx::pvdsdk::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 `Ps::Foundation::error()` or `Ps::Foundation::getErrorHandler()::reportError()` 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 `PxSetProfilerCallback()` to set itself up as the current profiler. This happens during `PxPvd::connect()` call, and it overrides the potentially already existing profiler callback. That is, if users call `PxSetProfilerCallback()` with their own user profiler, initialize PVD with `PxPvdInstrumentationFlag::ePROFILE`, then the user profiler callback is lost. Similarly, initializing PVD first then calling `PxSetProfilerCallback()` will make PVD's 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 before or after performing your own profiling operations:

```cpp
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, contextId);
    }

    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.
Interface

In this chapter we will have a quick look at the statistics information that PhysX collects every simulation step. Usually, this information can be explored in the PhysX Visual Debugger but we do offer a PhysX API method as well to allow applications to access the data directly. After a simulation step and a call to `PxScene::fetchResults()` statistics for the processed step can be retrieved through the `PxScene::getSimulationStatistics()` interface. The method copies the data to a user-provided `PxSimulationStatistics` structure. For details about the individual members please 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 performance issues. It provides a quantitative summary of the work done, i.e., the number of objects or combination of objects which have been processed in the current simulation step. For example, if you encounter performance spikes in certain frames, the simulation 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 objects over several simulation steps or maybe there is a particle system in the scene with a very small grid size.
- Are there suddenly many more collision pairs processed than expected? This could be caused by a badly configured collision pair filter or maybe some threshold has been accidentally raised.
- etc.

Please keep in mind that the simulation statistics are currently less a measurement of what the scene contains but rather what got processed. So it is only partially helpful to 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 - that corresponds to the PhysX API. It is therefore suitable for manual modification for debugging purposes. It offers platform independence and allows loading 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 instancing objects directly from memory without copying data. This in-place deserialization is well suited for performance critical real time situations. However, this approach is less 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 run-time 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 SDK, so it is not considered a serialization mechanism. Loading a cooked mesh involves allocation and endian conversion. As a consequence, it is much less efficient than PhysX' binary serialization mechanism. See *Shapes* for more details.

The following documentation will discuss how to use both serialization approaches, show how to build collections of PhysX objects and how these collections are serialized and deserialized. Further it will show how dependencies to other PhysX objects or application side objects can be re-established when deserializing.

PhysX also supports extending serialization to custom types, such as...

This is described in more detail in Section *Extending Serialization*.
First Code

The following code creates and serializes a rigid dynamic using both formats:

```cpp
// Create a material, a shape and a rigid dynamic
PxSphereGeometry geometry(1.0f);
PxMaterial* material = PxGetPhysics().createMaterial(0.0f, 0.0f, 0.0f);
PxShape* shape = PxGetPhysics().createShape(geometry, *material);
PxTransform t = PxTransform(PxIdentity);
PxRigidDynamic* dynamic = PxCreateDynamic(PxGetPhysics(), t, geometry);

// 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 outStream("serialized.dat");

// Binary
PxSerialization::serializeCollectionToBinary(outStream, *collection);

// ~Binary

// RepX
PxSerialization::serializeCollectionToXml(outStream, *collection);

// ~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 required objects to the collection.

The following code deserializes the rigid dynamic and adds it to a scene:

```cpp
PxSerializationRegistry* registry = PxSerialization::createSerializationRegistry;
```
When deserializing a binary serialized collection, the data first needs memory block that is aligned to 128 bytes. The memory block may before the objects have been released: it needs to persist for the er objects. This does not apply to RepX deserialization, as the memory for PhysX objects is allocated within PhysX. Finally the objects of the result be 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 maintains a mapping between IDs of type *PxSerialObjectId* and objects in the collection. IDs may be defined by the application. One caveat here is that the IDs must be unique within a collection, but do not have to be unique across different collections. If the latter is required by the application, it is the application's responsibility to ensure uniqueness.

Here is an example of how to iterate over a collection, for instance to ensure that the objects intended for serialization have all been added to the collection:

```
PxCollection* collection;
PxB32 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 an object within a collection, *PhysXExtensions* contains a function to remove and release all objects from a collection: *PxCollectionExt::releaseObjects*.

**Note:** Releasing an object within a collection invalidates the mapping to indices of objects.
A collection is said to be \textit{complete} if no contained objects depend on an object outside the collection. For example, an actor, a shape with a box geometry, and the material of the shape would together form a complete collection. The same collection without the material would be incomplete.

![Diagram](image)

**Figure 1:** Left: Complete Collection, Right: Incomplete Collection

For a formal definition please refer to \textit{Complete}.

Both complete and incomplete collections can be serialized, but when deserializing an incomplete collection, references to objects which were not serialized need to be resolved. The following two sections describe how PhysX collections can be serialized and deserialized using the binary format or RepX. The first section shows how to deal with complete collections, and the second section shows how to deal with 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):

```cpp
PxPhysics* physics; // Th
PxRigidDynamic* dynamic = PxCreateDynamic(...); // Cr
//Create
PxSerializationRegistry* registry = PxSerialization:::createSerializationRegistry();
PxCollection* collection = PxCreateCollection(); // Cr
collection->add(*dynamic); // Ad
```
Instead of using `PxSerialization::complete` it is possible to manually add all objects required for serialization. All objects the `PxRigidDynamic` references would need to be added and then all objects referenced by the newly added objects would need to be added as well and so forth. See definitions: `Requires`, `Complete`.

By default `PxSerialization::complete` follows references from joints to their actors, but not from actors to their joints. The `followJoint` parameter can be used to change the behavior of `PxSerialization::complete` to add the joints attached to each actor. This will cause entire actor-joint chains to be added to the collection.

When all the necessary objects have been added to a collection, create an implementation of the `PxOutputStream` interface, then serialize the collection:

```cpp
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 needed
// Note that releasing the collection will not release the contained objects
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 block or XML:

```cpp
PxSerializationRegistry* registry; // Registry for serializable types
PxCooking* cooking; // Cooking library needed for instantiating objects by RepX

// Deserialize

// Binary
void* memory128 = ...; // A 128-byte aligned buffer previously loaded from disk by the user
PxCollection* collection = PxSerialization::createCollectionFromBinary...
// ~Binary

// RepX
PxInputData& inputData = ...; // Impl
PxCollection* collection = PxSerialization::createCollectionFromXml...
// ~RepX
```

To add all the objects to the scene and release the collection and registry:

```cpp
PxScene* scene; // The scene object
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 - say, a ragdoll - will be deserialized multiple times, with each instance sharing the same materials and 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. The serialized format will remember each reference to an object in A using that object’s ID (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 reference can be 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 objects in collection B refer to.
- When deserializing, provide a collection with matching IDs for all the objects in A that were referenced by objects in B.

Here are examples of how the application can provide identities (*PxSerialObjectId* express requirements of one collection to another. This can be done:

```cpp
PxCollection* collection;
PxTriangleMesh* triMesh;
PxSerialObjectId triMeshId = 1;

collection->add(*triMesh, triMeshId);
```

Or set the ID after adding the object:
There is a helper function to generate IDs for all objects in a collection that do not have IDs yet:

```cpp
 PxSerialObjectId baseId = 1;
 PxSerialization::createSerialObjectId(*collection, baseId);
```

Already used ID values will be skipped by `createSerialObjectId`, as we already have IDs.

After providing correct IDs, all required objects have been added to the collection to be serialized, but without adding the objects that are intended to be referenced. The function in `PxSerialization` supports completing a collection relative to another collection:

```cpp
 PxSerializationRegistry* registry;
 PxCollection* collectionB;
 PxCollection* collectionA;

 PxSerialization::complete(*collectionB, *registry, collectionA);
```

Serialization example:

```cpp
 PxConvexMesh** convexes; // An array of mNbConvexes convexes
 PxRigidDynamic** actors; // An array of mNbConvexes actors

 PxSerializationRegistry* registry; // Registry for serializable types
 PxOutputStream& convexStream; // Output stream for the convex collection
 PxOutputStream& actorStream; // Output stream for the actors collection

 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, PxSerialObjectId);

// Serialize the convexes along with their IDs
// Binary
PxSerialization::serializeCollectionToBinary(convexStream, *convexCollection);

// RepX
PxSerialization::serializeCollectionToXml(convexStream, *convexCollection);

// 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, convexCollection);

// Serialize the actors with references to convexCollection
// Binary
PxSerialization::serializeCollectionToBinary(actorStream, *actorCollection, convexCollection);

// RepX
PxSerialization::serializeCollectionToXml(actorStream, *actorCollection, convexCollection);

// Release collections and registry
convexCollection->release();
actorCollection->release();
registry->release();

Deserialization example:

PxPhysics* physics;  // The physics SDK object
PxSerializationRegistry* registry  // Registry for serialization
PxCooking* cooking; // Cooking lib needed
PxScene* scene; // The scene into which the objects will be inserted

// Deserialize convexes along with their IDs (no external dependencies)

// Binary
void* convexMemory128; // Aligned memory containing serialized convexes
PxCollection* convexCollection = PxSerialization::createCollectionFromBinary(convexMemory128);

// RepX
PxInputData& convexInputData = ...; // Implemented by the application
PxCollection* convexCollection = PxSerialization::createCollectionFromXml(convexInputData, convexCollection);

// Deserialize actors referencing the convexCollection

// Binary
void* actorMemory128; // Aligned memory containing serialized actors
PxCollection* actorCollection = PxSerialization::createCollectionFromBinary(actorMemory128);

// RepX
PxInputData& actorInputData = ...; // Implemented by the application
PxCollection* actorCollection = PxSerialization::createCollectionFromXml(actorInputData, convexCollection);

// 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 serialized objects require objects that are not serialized and deserialized but created by other means:

PxSerializationRegistry* registry; // Registry for serializable types
PxMaterial** materials; // Created procedurally by application
PxRigidDynamic** actors; // An array of mNbConvexes actors
PxOutputStream& actorStream; // Output stream for the actors

// 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, materialCollection);

// Binary
    PxSerialization::serializeCollectionToBinary(actorStream, *actorCollection, materialCollection);

//~Binary

// RepX
    PxSerialization::serializeCollectionToXml(actorStream, *actorCollection, materialCollection);

//~RepX

actorCollection->release();
materialCollection->release(); // Note that materialCollection was not serialized
registry->release();

Deserialization:

PxScene* scene; // The scene into which the objects will be inserted
PxSerializationRegistry* registry; // Registry for serializable types
PxCooking* cooking; // Cooking library needed for instantiating objects
PxMaterial** materials; // Created procedurally by application

// recreate material collection with consistent IDs, no deserialization
PxCollection* materialCollection = PxCreateCollection();

for(PxU32 i=0;i<mNbMaterials;i++)
    materialCollection->add(*materials[i], PxSerialObjectId(i+1));

// Deserialize actors with reference material collection
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 that the application needs to give up by explicitly calling \texttt{release()}. The information whether the application gave up a reference to an object is \texttt{not} preserved on serialization.

See \textit{Shapes} for a discussion of the method \texttt{PxRigidActorExt::createExclusiveShape()} which automatically releases the initial reference to the shape, leaving the reference. Again, the information that this reference has been released is not preserved by serialization.

Example for shapes:

```cpp
PxOutputStream& outStream; // Output stream for the collection
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 to be released by 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, *collection);
collection->release();

// Releasing actors and shapes
actor->release(); // Releases actor and shapeA (automatically)
shapeB->release(); // Releases shapeB (necessary since shapeB was created through PxPhysics)

// Deserialize collection
...
void* memory128 = ...; // Aligned memory for serialized data
collection = PxSerialization::createCollectionFromBinary(memory128);

// Release actors and release ALL shapes (necessary since shape creation history is 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)).release();
        break;
    case PxConcreteType::eSHAPE:
        static_cast<PxShape&>(collection->getObject(i)).release();
        break;
    }
}

Note: There is a PhysXExtensions function to release all objects within a collection:
PxCollectionExt::releaseObjects.
Reconnecting PhysX and Game-Objects

Here is an example of how to fix up references with gameplay objects by querying the IDs of a collection:

```cpp
#include "PxSerialization.h"

// Deserialize objects along with IDs

// Binary
void* memory128; // Aligned memory containing serialized objects
PxCollection* collection = PxSerialization::createCollectionFromBinary(memory128, *registry);

// RepX
PxInputData& inputData = ...; // Implemented by the application
PxCollection* collection = PxSerialization::createCollectionFromXml(actorInputData, materialCollection);

// Receive a list of all deserialized IDs
#define MAX_IDS 100
PxSerialObjectId idBuffer[MAX_IDS];
PxU32 numIds = collection->getId(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 locate the corresponding gameplay object from a PxSerialObjectId
        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 PhysX runtime:

\[\text{PxCollectionExt::createCollection(PxPhysics\& sdk)}\]
\[\text{PxCollectionExt::createCollection(PxScene\& scene)}\]

```cpp
PxPhysics* physics; // The physics SDK object
PxScene* scene; // The physics scene
PxSerializationRegistry* registry; // Registry for serializable
PxOutputStream& outStream; // The user stream doing the writing

// 1) Create a collection from the set of all objects in the physics SDK that are shareable across multiple scenes.
PxCollection* everythingCollection = PxCollectionExt::createCollection(sdk);

// 2) Create a collection from all objects in the scene and add it to everythingCollection.
PxCollection* collectionScene = PxCollectionExt::createCollection(scene);
everythingCollection->add(collectionScene);
collectionScene->release();

// 3) Complete collection
PxSerialization::complete(*everythingCollection, *registry);

// 4) serialize collection and release it

// Binary
PxSerialization::serializeCollectionToBinary(outStream, *everythingCollection);

// RepX
PxSerialization::serializeCollectionToXml(outStream, *everythingCollection);

everythingCollection->release();
registry->release();
```

Deserialization is as previously:

```cpp
PxScene* scene; // The physics scene
PxCooking* cooking; // Cooking library needed for instantiating objects by RepX
PxSerializationRegistry* registry; // Registry for serializable
```
Serializability

This section contains various definitions to describe serializability of a collection. A collection can be successfully serialized and deserialized, optionally given an external references collection, can be queried by calling `PxSerialization::isSerializable(...)`.

Requires

An object A requires another object B if A maintains a reference to B that needs to be re-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:

<table>
<thead>
<tr>
<th>object type</th>
<th>requires</th>
</tr>
</thead>
<tbody>
<tr>
<td>joints</td>
<td>their actors and constraint</td>
</tr>
<tr>
<td>rigid actors</td>
<td>their shapes</td>
</tr>
<tr>
<td>shapes</td>
<td>their materials and mesh (triangle mesh, convex field), if any</td>
</tr>
<tr>
<td>articulations</td>
<td>their links and joints</td>
</tr>
<tr>
<td>aggregates</td>
<td>their actors</td>
</tr>
<tr>
<td>cloth actors</td>
<td>their cloth fabric</td>
</tr>
</tbody>
</table>
Subordinate

Subordinates are objects that cannot be instantiated without being owned by other objects. An articulation link, for example, can only be instantiated as part of its articulation.

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 **require** **D**. This means that **C** can be deserialized given **D**.

Serializable

Here is the complete set of requirements on a collection **C** with deper 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 references")
- Every subordinate object in **C** is required by another object in **C**. ("r

Binary Serialization Specifics

The following sections describe specific properties of the binary serialization
Management of memory blocks containing deserialized objects is left to the user's responsibility:

- allocate the memory block. Note that it must be properly aligned to a 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 any deserialized objects it contains. Concretely, calling release() on an object that was deserialized will cause its destructor to run, but will not deallocate the block. If the block is deallocated before the destructors have run for all the objects within, the PhysX runtime will likely crash. For more information about how deserialized objects need to be released see Reference Counting of Deserialized Objects.

Versioning

The binary serialized data is typically specific to the version of the SDK with. However, a SDK version can load the data of older SDK versions if the binary format didn't change. This is usually the case with bugfix releases. The compatible SDK versions are listed in the code documentation of PX_BINARY_SERIAL_VERSION PxSerialization.h.

Retargeting to other Platforms

Binary serialized data is platform-specific, and when serialized it is created on the platform on which it was created. The binary converter in the extensions/library retargets data from one platform to another. Typically assets are serialized on an authoring platform (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:

```cpp
void PxSerialization::dumpBinaryMetaData(PxOutputStream& stream, 
                                          // metadata for the 'from' platform  
                                          PxSerializationRegistry* registry;  // Registry for serializing
                                          // metadata for the 'to'
                                          PxInputStream& dstMetadata;  // metadata for the 'to'
                                          PxInputStream& srcMetadata;  // metadata for the 'from'
                                          PxInputStream& srcAsset;  // stream containing source asset
                                          PxU32 srcAssetSize;  // size of the source asset
                                          PxOutputStream& dstAsset;  // output stream for retargeted asset

                                          PxBinaryConverter* converter = PxSerialization::createBinaryConverter;
                                          converter->setMetaData(srcMetadata, dstMetadata);
                                          converter->convert(srcAsset, srcAssetSize, dstAsset);
```

On each target platform, run it once and keep generated data around. Alternatively a set of pre-built binary metadata is included with the PhysX SDK at [path to installed PhysX SDK]/Tools/BinaryMetaData.

Figure 3: Schema of Retargeting

Assuming that the extensions library has been initialized, conversion follows:
The Convert Tool

The convert tool is at [path to installed PhysX SDK]/Snippets/SnippetConvert. It illustrates how to convert PhysX 3 serialized binary files from one platform to another. It only compiles and runs on authoring platforms (Windows, MacOs and Linux).

SnippetConvert is a simple command-line tool supporting the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--srcMetadata=&lt;filename&gt;</td>
<td>Defines source metadata file</td>
</tr>
<tr>
<td>--dstMetadata=&lt;filename&gt;</td>
<td>Defines target metadata file</td>
</tr>
<tr>
<td>--srcBinFile=&lt;filename&gt;</td>
<td>Source binary file to convert</td>
</tr>
<tr>
<td>--dstBinFile=&lt;filename&gt;</td>
<td>Outputs target binary file</td>
</tr>
<tr>
<td>--generateExampleFile=&lt;filename&gt;</td>
<td>Generates an example file</td>
</tr>
<tr>
<td>--verbose</td>
<td>Enables verbose mode</td>
</tr>
</tbody>
</table>

Object Names

Some SDK objects, such as shapes and actors, can be given names using the `PxShape::setName()` and `PxActor::setName()` functions. By default these names are not serialized. The 'exportNames' parameter of `PxSerialization::serializeCollectionToBinary()` can be set to true in order to serialize the names along with the objects.

API-level Serialization (RepX) Specifics

RepX stands for Representation X and is the ASCII-XML serialization format for PhysX 3. As opposed to binary serialization, the RepX XML serialization is not intended to be used in performance critical or memory constrained situations. The following sections describe specifics of the RepX XML serialization system.

Upgrading RepX Data

Upgrading RepX data from an older PhysX version to a newer one is implicit when deserializing old RepX data with a newer PhysX SDK and re-serializing the resulting PxCollection.
Example for upgrading a RepX stream:

```cpp
PxPhysics* physics; // The physics SDK object (e.g. PhxsX 3.3)
PxCooking* cooking; // Cooking library needed for instantiating objects
PxSerializationRegistry* registry; // Registry for serializable types

 PxDefaultFileInputData inputData(pathTo30RepXFile); //load
PxCollection* collection = PxSerialization::createCollectionFromXml(inputData, *cooking,
PxDefaultFileOutputStream outStream(pathToNewRepXFile);
PxSerialization::serializeCollectionToXml(outStream, *collection,
```

Object Names

As opposed to binary serialization, the object names that can be ` PxShape::setName() ` and ` PxActor::setName() ` functions, are always included in the serialized format. On deserialization with ` PxSerialization::createCollectionFromXml(...) ` 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 be guaranteed that 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 configure the XML serialization to store the cooked triangle and convex mesh data along with the plain data. The cooked data caching can be enabled by passing a ` PxCooking ` flag to ` PxSerialization::serializeCollectionToXml(...) `. The cached cooked data format is incompatible with the current SDK version.
API-level RepX serialization should be used whenever compatibility and human readability 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 into the PhysX runtime. This is useful for rapid prototyping or for generally loading assets if performance is not of a big concern. For quick loading of assets, it is better to convert RepX data into binary serialized data. RepX is also useful in situations with unwanted behavior without the need to provide the whole application. For this, the application may be connected to the PhysX Visual Debugger (PVD) 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 situations. The main target use-case is streaming in chunks of a large game level that can’t be loaded into memory at once. Creating and loading save games is another application that could be optimized by using binary serialization. PhysX objects in binary format can also be sent over the network to enable efficient game state synchronization.
Snippet Discussion

The following snippets illustrate common operations such as managing collections, serialization, deserialization and re-targeting of binary data.

SnippetSerialization

SnippetSerialization shows binary and XML serialization of a scene with jointed rigid bodies representing a chain. This is done in a way that allows the instantiation of multiple chains while sharing the shape and the material across all of them. It shows how to create and populate collections, specify IDs to resolve dependencies, serialize collections, deserialize collections and add actors to the scene for simulation.

The snippet also shows how to allocate a data block aligned to 128 bytes and demonstrates how to copy binary serialized data into it. It further demonstrates that the data blocks containing the binary deserialized collections must be maintained until the corresponding objects are not needed anymore and have been released.
Figure 4: SnippetSerialization

**SnippetConvert**

SnippetConvert illustrates how binary serialized data can be re-targeted from an authoring platform to a runtime platform such as a console. The snippet is a simple command line tool that can load a binary serialized data file along with meta data files for both source and destination platforms and then output a converted binary data file. See the 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 collect files. See the snippet's source documentation for more details.
Best practices / Troubleshooting

- Concurrent simulation and serialization is not supported and leads to undefined behavior.
- If releasing PhysX objects leads to crashes or errors it is possible that the application is releasing some objects twice. The following two reasons should be considered:
  1. A potential source of error is to release PhysX objects without updating collections referencing these objects.
  2. Shapes that were created through an actor have their application reference automatically released on creation. If such a shape is serialized and deserialized the creation history will be lost. It might be convenient to use the extension function `PxCollectionExt::releaseObjects` because it deals with different cases as required. See Reference Counting of Deserialized Object.
- If accessing binary deserialized PhysX objects, including accesses during simulation, causes crashes it might be due to the premature release of the memory block that holds the deserialized objects.
- If binary files are too large and/or too slow to load it might be that they have been serialized multiple times. An example of a shared asset might be a mesh that is referenced by multiple shapes. The solution is to separate shared assets into a separate collection. See Serializing Incomplete Collections.
- If loading PhysX objects from RepX files is too slow two things should be considered:
  1. Could binary serialization be used instead? Even for debugging it might make sense to convert RepX files into binary serialized data by re-serializing them with the binary approach.
  2. Meshes tend to load very slowly from text files. RepX serialization offers an option to cache cooked mesh data by in-lining the binary data into the RepX file. If such a cache is present and valid, the load will become significantly faster. See Caching Cooked Geometry Data.
The PhysX Remote Debugger provides the functionality to export single scenes as RepX files. The resulting files can be used to playback a snapshot of the PhysX state. In many cases this is sufficient to isolate an issue. The option can be found in the 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 types. If an application were to require a new joint type, for example, the serialization system could be extended to add support for serialization of that new joint type.

The following document contains some recipes and example code that show how serialization may be extended to custom types. It doesn't cover all 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. A custom type for serialization 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 needs to be provided by implementing an interface. The template `PxSerializerDefaultAdapter` provides a default implementation, which can be specialized for the custom type as required. In order to support RepX serialization an additional `PxRepXSerializer` interface needs to be implemented. RepX serialization relies on automatic code generation using clang. Scripts to run the code generation for the examples can be found in (Tools/PhysXMetaDataGenerator).
Binary Serialization of Custom Classes

Serialization and deserialization of a custom class can be achieved in steps:

1. Define a `PxConcreteType` and type info for the custom class. Make sure its type value is unique.
2. The custom class needs to inherit from `PxBase` and implement it's interface.
3. Instance `PxSerializerDefaultAdapter<T>` and implement specialized methods where necessary.
4. If retargeting to other platforms is needed, implement `getBinaryMetaData()`.
5. Register the adapter and metadata, see `PX_NEW_SERIALIZER
   PxSerializationRegistry::registerSerializer
   PxSerializationRegistry::registerBinaryMetaDataCallback`. Note that serializers also need to be unregistered before `PxSerializationRegistry::release`
   application is responsible for custom type serializer allocation and deallocation.

For pointer members the following needs to be done (Note that references currently not supported):

6. Implement `PxSerializer::requires`. It should enumerate `PxBase` objects the object depends 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::createObject` `PxDeserializationContext::resolveReference`, `translatePxBase`.
9. Make sure that `PxSerializer::isSubordinate` returns whether the object is serialized along with its owner object. See `Subordinate`.
10. Export non `PxBase` data by implementing `PxSerializer::exportExtraData`
PxSerializationContext::writeData, alignData.

11. Import non PxBase data in the implementation of PxSerializer::createObject
PxDeserializationContext::readExtraData, alignExtraData.

**Note:** In checked builds (PX_CHECKED defined as 1) metadata definitions are verified against serialized data. If metadata definitions are missing warnings are output to the error stream during re-targeting (PxBinaryConverter::convert). To avoid false warnings all unused memory in custom serialized class instances should be marked with a 0xcd pattern. This can be done with Cm::markSerializedMem from CmUtils.h.

**Note:** The memory of a deserialized class instance should not be deallocated. The memory is embedded in the memory buffer containing the serialized data. The flag PxBaseFlag::eOWNS_MEMORY can used to decide whether the object memory needs to be deallocated or not.

Example for a custom class:

```cpp
#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>;

   // constructor setting up PxBase object
   CustomClass()
      : PxBase(customClassType, PxBaseFlag::eOWNS_MEMORY | PxBaseFlag::eOWN_BASE)
   {}

   // constructor called on deserialization
   CustomClass(PxBaseFlags baseFlags) : PxBase(baseFlags) {}
```
virtual ~CustomClass() {} 

//PxBase
virtual const char* getConcreteTypeName() const { return "CustomClass"; }

virtual bool isKindOf(const char* name) const
{
    return !strcmp("CustomClass", name) || PxBase::isKindOf(name);
}

//~PxBase

//PxSerializationRegistry::registerBinaryMetaDataCallback
static void getBinaryMetaData(PxOutputStream& stream)
{
    PX_DEF_BIN_METADATA_VCLASS(stream, CustomClass)
    PX_DEF_BIN_METADATA_BASE_CLASS(stream, CustomClass, PxBase)
    PX_DEF_BIN_METADATA_ITEM(stream, CustomClass, PxRigidDynamic, PxMetaDataFlag::ePTR)
    PX_DEF_BIN_METADATA_ITEM(stream, CustomClass, char, mBuf, PxMetaDataFlag::ePTR)
    PX_DEF_BIN_METADATA_ITEM(stream, CustomClass, PxU32, mSize, PxMetaDataFlag::ePTR)
    PX_DEF_BIN_METADATA_EXTRA_ITEMS(stream, CustomClass, char)
}

private:
    PxRigidDynamic* mActor;       //add in requires
    char* mBuf;                    //extra data
    PxU32 mSize;                   //size of mBuf
};

//PxSerializerDefaultAdapter
template<>
void PxSerializerDefaultAdapter<CustomClass>::requires(PxBase& obj, PxProcessPxBaseCallback c)
{
    CustomClass* custom = obj.is<CustomClass>();
    PX_ASSERT(custom);
    c.process(*custom->mActor);
}

template<>
void PxSerializerDefaultAdapter<CustomClass>::registerReferences(PxBase& obj, PxProcessPxBaseCallback c)
{
```cpp
{  
    CustomClass* custom = obj.is<CustomClass>();  
    PX_ASSERT(custom);  

    s.registerReference(obj, PX_SERIAL_REF_KIND_PXBAS, size_t(&o);  
    s.registerReference(*custom->mActor, PX_SERIAL_REF_KIND_PXBAS;  
}

//template<>  
void PxSerializerDefaultAdapter<CustomClass>::exportExtraData(PxB
}

{  
    CustomClass* custom = obj.is<CustomClass>();  
    PX_ASSERT(custom);  
    s.alignData(PX_SERIAL_ALIGN);  
    s.writeData(custom->mBuf, custom->mSize);  
}

//template<>  
PxBase* PxSerializerDefaultAdapter<CustomClass>::createObject(PxU8
}

{  
    CustomClass* custom = new (address) CustomClass(PxBaseFlag::eIS_RELEASABLE;  
    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;  
}

//~PxSerializerDefaultAdapter

void registerCustomClassBinarySerializer(PxSerializationRegistry&  
{  
    registry.registerSerializer(customClassType, PX_NEW_SERIALIZE
    registry.registerBinaryMetaDataCallback(CustomClass::getBinaryMeta
}

void unregisterCustomClassBinarySerializer(PxSerializationRegistry  
{  
    PX_DELETE_SERIALIZER_ADAPTER(registry.unregisterSerializer(cu
```
RepX Serialization of Custom Classes

Serialization and deserialization of a custom class can be achieved steps:

1. Perform the first three steps from *Binary Serialization of Custom Classes* *PxSerializer* and *PxSerializerDefaultAdapter<T>* required exclusively for binary serialization may be left empty.

2. Create a custom RepX serializer that implements the *PxRepXSerializer*. *PxRepXSerializer* is used to create an object from the xml file and the xml file. The class *RepXSerializerImpl* can be used implementations of some methods.

3. Register the general serializer adapter and the RepX serializer. Note serializers also need to be unregistered and deallocated.

4. RepX supports automatic reading and writing of class properties. clang has to be used to generate corresponding metadata: *Physics System*.

Example for a custom class:

```cpp
#include "SnRepXSerializerImpl.h"

const PxType customClassType = PxConcreteType::eFIRST_USER_EXTENSION
    PX_DEFINE_TYPEINFO(CustomClass, customClassType);

struct CustomClassRepXSerializer : public RepXSerializerImpl<CustomClassRepXSerializer, PxAllocatorCallback&> {
    CustomClassRepXSerializer(PxAllocatorCallback& inCallback)
        : RepXSerializerImpl<CustomClass>(inCallback) {}

    virtual PxRepXObject fileToObject(XmlReader& inReader, XmlMemoryAllocator PxRepXInstantiationArgs& inArgs, PxCollection* inCollection) {
        // factory for CustomClass instance provided by application
        CustomClass* object = createCustomClass();
    }
};
```
// when using the PhysX API metadata system readAllProperties(...) can be used to read all properties automatically
readAllProperties(inArgs, inReader, object, inAllocator,
    return PxCreateRepXObject(object);
}

virtual void objectToFileImpl(const CustomClass* obj, PxCollec
XmlWriter& inWriter, MemoryBuff
PxColl
{
    // when using the PhysX API metadata system writeAllProperties(...) can be used to save all properties automatically
    writeAllProperties(obj, inWriter, inTempBuffer, *inColl
}

// this can return NULL if fileToObject(...) is overwritten with fileObject(...)
virtual CustomClass* allocateObject(PxRepXInstantiationArgs&)
};

void registerCustomClassRepXSerializer(PxSerializationRegistry& reg
{        
    registry.registerSerializer(customClassType,
        PX_NEW_SERIALIZER_ADAPTER(CustomC
    registry.registerRepXSerializer(customClassType,
        PX_NEW_REPX_SERIALIZER<Custom
    }

void unregisterCustomClassRepXSerializer(PxSerializationRegistry&
{        
    PX_DELETE_SERIALIZER_ADAPTER(registry.unregisterSerializer(cu
    PX_DELETE_REPX_SERIALIZER(registry.unregisterRepXSerializer(cu

**Note:** Implementing a PxRepXSerializer is currently not practical without including the 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 cascading rules, 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 created with the following recipe:

- **CustomTypeExtensionAPI.h**
  - Add all the types that should be exported to gUserPhysXTypes.
  - Add the unnecessary types to gAvoidedPhysXTypes. It will not generate 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 source file in there.
- Specify the filename of autogenerated files. Then it will generate the following files:

```
include/CustomTypeAutoGeneratedMetaDataObjectNames.h
include/CustomTypeAutoGeneratedMetaDataObjects.h
src/CustomTypeAutoGeneratedMetaDataObjects.cpp
```

- **CustomTypeMetaDataObjects.h**
  - `CustomTypePropertyInfoName` has to be defined and `CustomTypeAutoGeneratedMetaDataObjects.h` has to be included in this file.
  - The file will then export the properties of the custom class and can be included for implementing the custom RepX serializer.
- **CustomTypeMetaDataObjects.cpp**
  - This file is optional. It is only required when custom properties are needed.

PxVehicle serialization is a useful example. With `Source/PhysXVehicle` as the root folder, 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/PxVehicleAutoGeneratedMetaDataObjectNames.h
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 need to be exported.
2. runClang_[windows|osx|linux].[bat|sh]: The target directory is set to src/PhysXMetaData, and the target name is PxVehicle.
3. PxVehicleMetaDataObjects.h: It defines the custom properties.
   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 included for the custom RepX serializer.
Introduction

This chapter covers a number of best practices for the PhysX SDK to ε and fixing frequently encountered issues.
Debugging

The PhysX SDK contains a few debugging helpers. They can be used to make sure scenes are properly set up.

Use checked builds and the error stream

The PhysX SDK has different build configurations: Debug, Checked, Release, Profile. To make sure that the scene is properly set up without warnings or errors, use Debug or Checked builds, and monitor the error callback. Please refer to the Reporting chapter for details. Note that some checks can be expensive and thus not performed in Release or Profile builds. If the SDK silently fails or crashes in a Release build, please switch to Debug or Checked builds to ensure that an uncaught error is not the cause.

Visualizing physics data

Use the PhysX Visual Debugger (PVD) to see what PhysX is seeing and ensure 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 Profile 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 result in unexpected coordinates. We recommend using PxSceneDesc::sanityBounds, to generate reports when objects are inserted at positions beyond what your application expects, or when application code moves them to unexpected positions. Note that these bounds only apply to application updates.
coordinates, not updates by the simulation engine.
Performance Issues

The PhysX SDK has been optimized a lot in the past dot releases. However, there 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, Profile. To identify performance bottlenecks, please use Profile builds and `PxPvdInstrumentationFlag::ePROFILE` only, since enabling the other might negatively affect performance. Please refer to the PhysX Visual Debugger chapter for details.

Use release builds for final performance tests

The PhysX SDK has different build configurations: Debug, Checked, Release. Release builds are the most optimal. If you encounter a performance issue while using other builds, please switch to Release builds and check if the problem is still there.

Disable debug visualization in final/release builds

Debug visualization is great for debugging but it can have a significant 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 the debug data and the code rendering it is usually not optimal. Use a culling box to limit the amount of data the SDK gathers and sends to the renderer. Please refer to the Debug Visualization chapter for details.

Consider using tight bounds for convex meshes
By default PhysX computes approximate (loose) bounds around convex objects. Using PxConvexMeshGeometryFlag::eTIGHT_BOUNDS enables smaller/tighter bounds, which are more expensive to compute but can result in improved simulation performance when a lot of convex objects are interacting with each other. Please refer to the documentation for details.

**Use scratch buffers**

The PxScene::simulate function accepts optional scratch buffers that reduce temporary allocations and improve simulation performance. Please refer to 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. Generally speaking, the new PxMeshMidPhase::eBVH34 introduced in PhysX 3.4 has better performance for scene queries against large triangle meshes. Please refer to the Geometry chapter for details.

**Use the proper narrow-phase algorithm**

PxSceneFlag::eENABLE_PCM enables an incremental "persistent contact manifold" 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 previous versions like 3.3.

**Use the proper broad-phase algorithm**

PhysX also supports two different broad-phase implementations, selected with PxSceneDesc::broadPhaseType. The different implementations have various performance characteristics, and it is a good idea to experiment with both and find the one that works best for you. Please refer to the Rigid Body Collision chapter for details.
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 again), disable 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 scenes containing a lot of dynamic objects, try to increase the PxSceneDesc::dynamicTreeRebuildRateHint 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. This is faster than first writing the data to a file or a memory buffer, and then passing the data to PhysX.

The "Well of Despair"

One common use-case for a physics engine is to simulate fixed-size 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 frames 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 simulation with a larger time-step or to simulate multiple, smaller sub-steps. The latter is generally a preferable solution because changing the size of time-steps in a physics simulation can significantly change perceived behavior. However, when using a sub-stepping approach, 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 60 FPS. Simulating a large number of physics bodies and, as a result, the physics behavior is expensive. In order to meet the 60FPS requirement, the entire frame...
within ~16ms. As already mentioned, the physics is reasonably expensive, 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 add sub-steps 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 16ms deadline for 60FPS, meaning that the frame including v-sync took 33ms.

If this happens, we must now simulate 2 sub-steps in the next frame, which takes ~18ms and also misses our 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 stuck in a performance trough indefinitely. The application is capable of simulating at 60FPS but becomes stuck in the so-called "physics well of despair" as a result of substepping.

Problems like this can be alleviated in several ways:

- Decouple the physics simulation from the game's update/render loop. The physics simulation becomes a scheduled event that occurs at a fixed frequency. This can make player interaction in the scene more difficult and may introduce latency, so it must be well-thought through. However, using multiple scenes (one synchronous for "important" objects, one asynchronous for "unimportant" objects) can help.
- Permit the game to "drop" time when faced with a short-term spike. This may introduce visible motion artifacts if spikes occur frequently.
- Introduce slight variations in time-step (e.g. instead of simulating at 1/60th, consider simulating a range between 1/50th and 1/60th). This can introduce non-determinism into the simulation so should be used with caution. If this is done, the additional time that must be simulated can potentially be amortized over several frames by simulating slightly larger time-steps.
- Consider simplifying the physics scene, e.g. reducing object count, adjusting iteration counts etc. Provided physics simulation is a small portion of the
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 applications. 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 dynamics 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 can be very expensive. As a result, adding, removing or moving any static objects will result in the static AABB tree being fully recomputed, which can introduce significant performance cost. As a result, we recommend the use of dynamics AABB trees for both static and dynamic pruners in games which stream in the static environment. Scene query performance against newly added objects can be improved by using PxPruningStructure, which can precompute the AABB structure of inserted objects 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, additional worker threads can detrimentally affect performance. This is because, at its core, PhysX operates around a task queue. When a frame's simulation is started, it dispatches a chain of tasks that encapsulate that frame of physics simulation. At various stages of the physics pipeline, work can be performed in parallel on multiple worker threads. However, if there is insufficient work, there will be little or no parallelization. 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 and may incur some additional overhead depending on the CPU architecture.
running on just a single worker thread. As a result, developers should measure the performance of the engine with their expected physics loads with different numbers of threads to maximize their performance and make sure that they are making the most of the available processing resources for their game.

**Memory allocation**

Minimizing dynamic allocation is an important aspect of performance tuning, and PhysX provides several mechanisms to control memory usage.

Reduce allocation used for tracking objects by presizing the capacities of scene data 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 for preallocation 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, each 16K in size. You can control the current and maximum size of the pool using the nbContactDataBlocks and maxNbContactDataBlocks members of PxSceneDesc. PhysX will never allocate more than the maximum number of blocks specified, and if there is insufficient memory it will instead simply drop contacts or joint constraints. You can find out how many blocks are currently in use with the getNbContactBlocksUsed() method, and find out the maximum number that have ever been used with getMaxNbContactDataBlocksUsed() method.

Use PxScene::flushSimulation() to reclaim unused blocks, and to shrink the size of scene data structures to the size presently required.

To reduce temporary allocation performed during simulation, provide a memory 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 Depth Computation

Implementing a Character Controller (CCT) is a common use case for the Scene Query (SQ) system. A popular approach is to use sweeps to implement movement logic, and to improve robustness by using Geometry Queries (GQ) to compute 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 goal position.
2. If no initial overlap is detected, move the CCT shape to the position of the first hit, and adjust the trajectory of the CCT by removing the motion relative to the contact normal of 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. In certain initial conditions it is possible that the SQ system will determine that a pair of objects is initially overlapping while the GQ-Penetration Depth computes them as disjoint (or vice-versa). Penetration depth calculations involve operating by shrinking the convex hull and performing distance calculations between a shape and the shrunken convex hull. To understand the conditions that occur and how to resolve the artefacts, please refer to the diagrams below. Each diagram represents the initial conditions of two shapes: a Character Controller shape (red boxes), a convex obstacle (black boxes), at the
the algorithm above is executed. In the diagrams, the outermost rectangular black box represents the convex hull as seen by the SQ algorithms; the inner black box with a dashed line represents the shrunken convex shape and the black box with rounded corners is the shrunken convex shape inflated by the amount by which we shrunk. These boxes are used by the GQ-Penetration Depth computation. Although the example refers to convex hull obstacles, the issue is not exclusive to the convex hull shape; it 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 black box representing the convex obstacle. In this situation the SQ-Sweep will report an initial overlap but the GQ-Penetration Depth function will report no hit, because the red box is not touching the black box with rounded corners.

To resolve this, inflate the CCT shape for the GQ-Penetration Depth calculation so that it detects an overlap and returns a valid normal. Note that after inflating the shape, the GQ-Penetration Depth function will report that the shapes are penetrated more deeply than they actually are, so take this additional penetration into account when depenetrating in Step 4. This may result in some clipping around the 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 black box seen by the SQ system, but does not overlap the shrunken shape seen by the GQ calculator. The GQ-Penetration Depth system will return the penetration from point b but not from point c to point a. Therefore the CCT may clip through the convex hull after depenetration. 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 that it overlaps with the shrunken shape seen by GQ, the GQ-Penetration Depth calculator will return the penetration from point c to point a. In this case, the GQ-Penetration Depth value can be used without modification in Step 4. However, as this condition would be difficult to categorize without additional computational cost, it is best to inflate the shape as recommended in Step 4 and then subtract this inflation from the returned 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 Translation Direction) when an initial overlap is detected by a sweep. This flag is guaranteed to generate an appropriate normal under all circumstances, including cases where it may detect an initial overlap but calling a stand-alone MTD function may not. It still may suffer from accuracy issues with penetration depths but, in cases around corners/edges, it will report a distance of 0 and the correct contact normal. This can be used to remove components of the sweep moving into the normal and then re-sweeping when attempting to implement a CCT.
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-height that are then converted to a float and multiplied by PxHeightFieldGeometry::heightScale to obtain local space scaled coordinates. Shape transform is then applied on top to obtain world space location. The transformation is performed as follows (in pseudo-code):

\[
\text{localScaledVertex} = \text{PxVec3}(\text{row} \times \text{desc.rowScale}, \text{PxF32(\text{heightSample})} \times \text{col} \times \text{desc.columnScale})
\]

\[
\text{worldVertex} = \text{shapeTransform(\text{localScaledVertex})}
\]

The following code snippet shows one possible way to build quantized local space heightfield coordinates from world space grid heights stored in terrainData.verts:

```cpp
const PxU32 ts = ...; // user heightfield dimensions (ts = terrain samples)
// create the actor for heightfield
PxRigidStatic* actor = physics.createRigidStatic(PxTransform(PxIdentity)
// iterate over source data points and find minimum and maximum heights
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 generate
// to the source
// clamp to at least PX_MIN_HEIGHTFIELD_Y_SCALE to respect the PhysX API
PxReal heightScale = PxMax(deltaHeight / quantization, PX_MIN_HEIGHTFIELD_Y_SCALE);
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&)(hfSamples
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 heightfield
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 needed to manage these pairs.

Use scratch buffers

See the above chapter about Performance Issues for details. Scratch buffers can be shared between multiple sub-systems (e.g. physics and rendering), which can globally improve memory usage. PhysX will not use less memory per-se, but it will allocate less of it.

Flush simulation buffers

Call the PxScene::flushSimulation function to free internal buffers used for temporary computations. But be aware that these buffers are usually allocated once and reused in subsequent frames, so releasing the memory might trigger new re-allocations during 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 memory chapter for details.
Tweak cooking parameters

Some cooking parameters have a direct impact on memory usage. PxMeshPreprocessingFlag::eDISABLE_ACTIVE_EDGES_PRECOMPUTE, PxCookingParams::suppressTriangleMeshRemapTable, PxBVH33MidphaseDesc::meshCookingHint, PxBVH33MidphaseDesc::meshSizePerformanceTradeOff, PxBVH34MidphaseDesc::numTrisPerLeaf, PxCookingParams::midphaseDesc, PxCookingParams::gaussMapLimit and PxCookingParams::buildTriangleAdjacencies can be modified to choose between runtime performance, cooking performance or 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 scene-query flag. If a shape is only used for simulation (e.g. it will never be raycasted again), disable its simulation 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 a low value (7.0). This can artificially prevent the objects from spinning quickly, which might look unrealistic and wrong in some cases. Please use PxRigidDynamic::setMaxAngularVelocity to increase the maximum allowed angular velocity.

Overlapping objects explode

Rigid bodies created in an initially overlapping state may explode, because the SDK tries to resolve the penetrations in a single time-step, which can lead to large velocities. Please use PxRigidBody::setMaxDepenetrationVelocity 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 that appear and disappear 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 recommended for jointed objects though, so use PxRigidDynamic::setStabilizationThreshold 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
Please refer to the *Rigid Body Dynamics* chapter for details.
- Consider creating the same constraints multiple times. This is similar to increasing the number of solver iterations, but the performance impact is local object rather than the simulation island it is a part of. So it can be better overall. Note that the order in which constraints are created is important. Say you have 4 constraints named A, B, C, D, and you want to create them in the AAAABBBBBCCCDDDD order will not improve the behavior, but 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 performance, although it can be an expensive solution. Instead of running 1 simulation call with time-step dt and N solver iterations, consider trying N simulation calls with 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 quite effective. An alternative is to compute the inertia tensor (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, it can be effective to create an extra distance constraint between the two ends of the rope to limit its stretching. Alternatively, one can create distance constraints between elements N and 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 the pivots at the spheres' centers is more stable than placing them on the spheres' surfaces.
- Use articulations. Perhaps not surprisingly, articulations are much better at simulating articulated objects. They can be used to model better ropes, br...
ragdolls out-of-the-box, without the need for the above workaround. Please refer to the Articulations chapter for details. They are more expensive though.
GPU Rigid Bodies

Collision detection with PxSceneFlag::eENABLE_GPU_DYNAMICS will be executed on GPU for all convex-convex, convex-box, box-box, convex-mesh, box-mesh, convex-HF, 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 hulls), the pair requests contact modification, the triangle mesh was not cooked with GPU (PxCookingParams::buildGrbData) or if the triangle mesh makes use of per-triangle materials.

Aggregates are used to lighten the load on broad phases. When running on the CPU, aggregates frequently improve performance by reducing the load on the broad phase algorithm. However, there is some cost when aggregate overlaps must be processed by a separate module. When using aggregate overlaps occurs on the CPU and, while using aggregates can reduce load 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, as results can vary between platforms due to differences in hardware maths precision and the compiler readers instructions during optimization. This means that different between different platforms, different compilers operating on the same compiler on between optimized and unoptimized builds using the same compiler on. However, on a given platform, given the exact same sequence of events and exact scene using a consistent time-stepping scheme, PhysX is expected to produce deterministic results. In order to achieve this determinism, the application must recreate the scene in the exact same order each time and insert the actors into a new PxScene. There are several other factors that can affect determinism such as (e.g. variable) time-stepping scheme is used or if the application does not perform the same sequence of API calls on the same frames, the PhysX simulation can diverge.

In addition, the PhysX simulation can produce divergent behavior if any conditions in the simulation has varied. Even the addition of a single actor that is not interacting with the existing set of actors in the scene can produce divergent results.

PhysX provides a mechanism to overcome the issue of divergent behavior in existing configurations as a result of additional actors being added or actors that do not interact with the other actors in the scene. This is enabled by raising PxSceneFlag::eENABLE_ENHANCED_DETERMINISM on PxSceneDesc::flags prior to creating the scene. Enabling this mode makes some performance concessions to be able to offer an improved level of determinism. The application must still follow all the requirements to achieve deterministic behavior described previously in order for this mechanism to produce consistent results.
This guide describes how to upgrade applications that have an integration using PhysX 3.x. As the changes are numerous and significant, the level in upgrading to PhysX 3 should be carefully assessed before starting the application's integration code.
Removed Features

This section lists features of PhysX 2 that do not have a PhysX 3 equivalent that rely on these features may need fundamental changes, or should stay with using PhysX 2.

Compartments

PhysX 2 scenes supported scene compartments. A separate compartment could be assigned to simulating rigid bodies, deformables or fluids. The compartments were simulated in parallel and the scene code contained some extra logic for interaction between compartments. Compartments were added as an afterthought, as the SDK was not originally designed to support interaction between multiple simulation technologies. This design deficiency was addressed from the ground up in PhysX 3 and compartments were no longer needed.

One missing detail is separate time steps are no longer directly supported, a possible workaround is to create multiple PxScenes and step them at different rates. In this scenario the force exchange implementation would be entirely up to the user. Another possible approach is to simulate the entire scene using the minimum required for any of the compartments.

Deformables

PhysX 2 supported a wide range of deformable mesh simulation features such as environmental cloth, soft bodies, inflatable balloons and plastic deformation of rigid metal. For performance and code quality reasons, 3.3 temporarily stopped supporting some 2.8 deformable features in favor of a much simpler and higher performance cloth 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 substitute for many applications of PhysX 2 deformables.

NxUtilLib
The assorted utility functions that were in this library was either moved or deleted. Sweep, overlap and ray tests are available in PxGeometryQuery. Diagonalization is in PxDiagonalize(). Density computation from mass point unit manipulation routines are gone. Geometrical helpers are in general gone.

**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 relevant source files 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 'system software' 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 creation function.

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(*aPxCapsuleGeometry(radius, halfHeight), myMaterial);
```
### Callback Classes

PhysX 2 callback classes are listed below, followed by the corresponding PhysX 3 class, if there is one:

<table>
<thead>
<tr>
<th>NxUserAllocator</th>
<th>PxAllocatorCallback</th>
</tr>
</thead>
<tbody>
<tr>
<td>NxUserOutputStream</td>
<td>PxErrorCallback</td>
</tr>
<tr>
<td>NxUserContactReport</td>
<td>PxSimulationEventCallback</td>
</tr>
<tr>
<td>NxUserNotify</td>
<td>PxSimulationEventCallback</td>
</tr>
<tr>
<td>NxUserTriggerReport</td>
<td>PxSimulationEventCallback</td>
</tr>
</tbody>
</table>

The following PhysX 2 callback classes have no PhysX 3 direct equivalent:

<table>
<thead>
<tr>
<th>NxUserRaycastReport</th>
<th>Ray casting Results. Results are now passed to PxHitBuffer object.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NxUserEntityReport</td>
<td>Sweep and Overlap results. Results are now passed to the user using a PxHitBuffer object.</td>
</tr>
<tr>
<td>NxStream</td>
<td>Data serialization. Serialized data is now written to binary buffers.</td>
</tr>
</tbody>
</table>

Below is a list of new callback classes that offer functionality that did not exist in PhysX 2 yet:

<table>
<thead>
<tr>
<th>PxBroadPhaseCallback</th>
<th>Broad-phase related events.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PxSimulationFilterCallback</td>
<td>Contact filtering.</td>
</tr>
<tr>
<td>PxUserControllerHitReport</td>
<td>Reports character controller events.</td>
</tr>
<tr>
<td>PxControllerBehaviorCallback</td>
<td>Customizes behavior of character controller collisions.</td>
</tr>
<tr>
<td>PxContactModifyCallback</td>
<td>Modification of contact constraints.</td>
</tr>
<tr>
<td>PxCCDContactModifyCallback</td>
<td>Modification of CCD contact constraints.</td>
</tr>
<tr>
<td>PxConstraintConnector</td>
<td>Custom constraints.</td>
</tr>
<tr>
<td>PxProcessPxBaseCallback</td>
<td>Serialization.</td>
</tr>
<tr>
<td>PxQueryFilterCallback</td>
<td>Scene query filtering.</td>
</tr>
<tr>
<td>PxSpatialLocationCallback</td>
<td>Scene Queries against PxSpatialIndex</td>
</tr>
<tr>
<td>PxSpatialOverlapCallback</td>
<td>Scene Queries against PxSpatialIndex</td>
</tr>
</tbody>
</table>


Memory Management

NxUserAllocator is renamed to PxAllocatorCallback. An important change since PhysX 2: The SDK now requires that the memory that is returned be 16-byte aligned. On many platforms malloc() returns memory that is 16-byte aligned, but on Windows the system function _aligned_malloc() provides this capability.

Debug Rendering

Debug visualization formerly provided by NxScene::getDebugRenderable is now handled by PxScene::getRenderBuffer() and related functions.

Error Reporting

NxUserOutputStream is now called PxErrorCallback, but works the same way. A separate reportAssertViolation() function is introduced, which only ships with the source release and go directly to platform hooks.

Type Casting

PhysX 2 style downcasting:

```cpp
NxSphereShape * sphere = shape->isSphere();
```

is replaced by the following template syntax:

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

Multithreading

Compared to PhysX 2, there are now more situations where it is legal to call the SDK from 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 to take over all of the simulation's thread scheduling. It is also possible for the application to define its own tasks and submit them to the SDK's default scheduler. See TaskManagement for details.

**Startup and Shutdown**

PxCreatePhysicsSDK() has been renamed PxCreatePhysics(), and the parameters have slightly changed. A foundation instance must first be created using PxCreateFoundation().

**Extensions**

A lot of non-essential utility code has been moved to the extensions library. NxActor::addForceAtPos() is now exposed as PxRigidBodyExt::addForceAtPos(). The former function appears to be missing, look there. It is available after calling PxInitExtensions().

**Heightfields**

Heightfields now need to be pre-cooked like convexes and meshes. PhysX 3 heightfields can be set to use the same internal collision logic as meshes so they have uniform behavior.

**Cooking**

The PhysX 2 cooking library was created by calling:

```c
NxCookingInterface *gCooking = NxGetCookingLib(NX_PHYSICS_SDK_VERSION);
gCooking->NxInitCooking();
```

It can now be accessed through a single PxCreateCooking() call. Cooking function names are slightly changed, e.g. NxCookTriangleMesh() is now cooking.cookTriangleMesh().
Serialization

PhysX 3 has two serialization systems: 'RepX' based on XML, and a second for fast binary data. Neither approach is similar to PhysX 2's save-to-desc and load-from-desc based serialization code, though the PhysX 3 'RepX' serialization is similar to 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 object, for example, isSleeping() could be called on static actors which did not have any sleep logic. In PhysX 3, we decoupled actor into a hierarchy of specialized sub-classes. PxCloth and PxParticleSystem are now subclasses of PxActor.

Actor Creation

In PhysX 2, the objects inside each scene were created by the scene class itself. In PhysX 3, objects are created by PxPhysics, and need to be added to a scene in a subsequent step by calling:

```
mScene->addActor(actor);
```
**Material Indexes**

PhysX 2 uses so-called material indexes for stored materials. Material indexes are supported in PhysX 3 only 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 PxQuat for rotation. 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 class, to which a PxGeometry object is passed on creation. To determine the geometrical type, call PxShape::getGeometryType(). To extract a PxGeometry object of unknown type, use PxShape::getGeometry().

**Skin Width**

PhysX 2’s NX_SKIN_WIDTH and NxShapeDesc::skinWidth were replaced with PxShape::setContactOffset() and setRestOffset(). See *Tuning Shape Collision Behavior*. 
Joints

The D6 driveType in PhysX 2 no longer exists in PhysX 3. Now drive spring-like: if you want position drive you set the 'spring' value non-zero, velocity drive you set the damping field non-zero, and if you set both you get a spring. Some specialized joints like NxJointDriveDesc, NxJointLimitS names) now were moved to Extensions (see the extensions folder inside directory).

If you have used the deleted NxSpringAndDamperEffecttor, you should with a spring property.

All special axes for a joint (rotation axis for revolute, translation axis axis for D6) now use the x-axis.

Joint limits now require a contact offset, which determines the distance which it becomes active. It functions similarly to the contactOffset parameter for collision detection.
**Time Stepping**

PhysX 2 had two different time stepping modes: NX_TIMESTEP_FIXED (division into fixes steps) and NX_TIMESTEP_VARIABLE (user specified steps) to the setTiming() function. This controlled SDK-internal substepping calculation of the proper size of the next time step, and called an internal simulate function with this elapsed time.

PhysX 3 discards with the substepping code altogether, and exposes the simulate function directly:

```cpp
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 this.

The fetchResults function stayed the same, however there is no more flag to specify which 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 introduced a per-body function PxRigidDynamic::setSleepThreshold() which is an energy-based setting, more similar to the PhysX 2 NX_DEFAULT_SLEEP_ENERGY.

The global NX_BOUNCE_THRESHOLD is replaced by PxSceneDesc::bounceThresholdVelocity.

The NX_DYN_FRICT_SCALING, NX_STA_FRICT_SCALING scaling factors have been 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 arbitrary memory cannot be accessed by filtering code so that it can be executed on GPUs with optimal performance. If performance is not a priority, similar functionality 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 behavior. Start by checking if this class is sufficient. As this is an extension class, the source code is available and may be extended or customized.

To migrate your fixed function PhysX 2 filtering code on your own, you need to implement its exact behavior and implement it as a callback or shader. Let us look at the precise mechanisms and make some recommendations for porting:

```cpp
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 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 would be difficult in practice, but implementing as a callback should be trivial using a data structure such as the STL hash_map where Key is a struct holding the two pointers and Data is the bit flag.
Another scheme provided by PhysX 2 were collision groups:

```cpp
class NxShape {
public:
  virtual void setGroup(NxCollisionGroup collisionGroup) {
  }
private:
  NxCollisionGroup m_collisionGroup;
};

class NxScene {
public:
  virtual void setGroupCollisionFlag(NxCollisionGroup group1,
                                      NxCollisionGroup group2,
                                      bool enable) {
  }
private:
  NxCollisionGroup m_group1;  // collision group 1
  NxCollisionGroup m_group2;  // collision group 2
};
```

This approach let the user assign shapes to one of 32 collision groups. Each pair of groups can be assigned a boolean pair flag. This approach lends itself better to a shader based implementation. To do this, you should reserve a word of each shape's filterData (say word0) to hold the group index, and assign this as before. Next, define a matrix to hold the group pair bits, and a function to set it:

```cpp
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 is created. This is because if the matrix could change during simulation, it would force an arbitrary amount of existing contact pairs to be refiltered. In a large simulation, this could be an unacceptable amount of computation. Therefore the matrix must be initialized to its final state before the

scene is created, like this:

```cpp
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:

```cpp
PxFilterFlags FilterShader(
    PxFilterObjectAttributes attributes0, PxFilterData filterData0,
    PxFilterObjectAttributes attributes1, PxFilterData filterData1,
    PxPairFlags& pairFlags, const void* constantBlock, PxU32 cons
{
    // let triggers through, and do any other prefiltering you need
    if(PxFilterObjectIsTrigger(attributes0) || PxFilterObjectIsTrigger(attributes1))
    {
        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 (e.g., PxScene::raycast(...)) has changed. In PhysX 3, raycast/overlap/sweep functions now require a pre-allocated buffer or a callback class as a parameter in order to receive multiple intersections. If you do not know the maximum number of intersections in advance, you can inherit from PxHitCallback and override processTouches virtual function to receive an arbitrary number of intersections via multiple callbacks using only a fixed-size buffer. 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 should pass an origin (PxVec3) and a direction (PxVec3) instead of a NxRay that combined these fields in PhysX 2.

Overlaps

Routines like overlapSphereShapes, overlapAABBShapes, or overlapCapsuleShapes are now all covered with PxScene::overlap(passing in a PxSphereGeometry, PxBoxGeometry or PxCapsuleGeometry as a first parameter).

Sweep Tests

PhysX 2 provides a linearCapsuleSweep that takes two points to define a capsule with spherical ends. In PhysX 3 we have a general sweep() routine that takes a PxGeometry and an initial PxTransform position. Capsules were defined in PhysX 2 as two points that should be converted to an initial transformation (PxTransform) that contains PxVec3 for position and PxQuat for rotation. PxCapsuleGeometry's length is along the x-axis in local space.
• This guide highlights all significant parts of the API that have changed in the last 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 and will be removed in a future release. The preferred method is to use PxMat33(PxIdentity), PxMat44(PxIdentity), PxQuat(PxIdentity), PxTransform(PxIdentity) 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 are now folded into a single `PxScene::raycast` call
  - Same for overlaps and sweeps
  - Same for `PxBatchQuery` and `PxVolumeCache`
  - For `PxScene` queries a deprecated backwards compatible mapping was added to aid the transition
    - This mapping will be removed in the next dot release
  - There are now dedicated callback and buffer classes for results, replacing `PxRaycastHit` array and count parameters:
    - Same for sweeps and overlaps
    - See `PxRaycastBuffer`, `PxSweepBuffer`, `PxRaycastCallback`, `PxSweepCallback`, `PxOverlapCallback`
  - The way results are returned is now more robust and it is possible to transparently handle unbounded number of results without dynamic allocations.
  - Header `PxSceneQueryFiltering.h` was renamed to `PxQueryFiltering.h`
  - `PxSceneQueryReport.h` to `PxQueryReport.h`
  - `PxHitFlag::eIMPACT` changed to `PxHitFlag::ePOSITION`
  - `PxRaycastHit.impact` renamed to `PxRaycastHit.position` (same for `PxSweepHit.impact`)
  - `PxQueryFlag::eNO_BLOCK` and `PxQueryFlag::eANY_HIT` flags were added
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 clientId

- PxBatchQueryDesc now requires 3 parameters at constr
  maxRaycastsPerExecute, PxU32 maxSweepsPerE
  maxOverlapsPerExecute
  - Each of these numbers is an upper bound c
    PxBatchQuery::raycast(), sweep() and overlap() c
    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 before each execute.
  - userRaycastHitBuffer has been renamed to userRaycastHitBuffer.
  - raycastHitBufferSize has been renamed to raycastHitBufferSize.
  - same for overlaps and sweeps (userSweepHitBuffer, sweepHitBufferSize, userOverlapHitBuffer, overlapHitBufferSize).
  - A code snippet below illustrates the migration for these changes.

- 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 write:
    
  - >raycast(...,PxQueryFilterData(PxQueryFlag::eDYNAMIC | PxQueryFlag::eSTATIC), ...)
  - This change was made to improve type safety and reduce confusion while reading the code employing implicit constructors.

- PxRaycastBufferN, PxOverlapBufferN and PxSweepBufferN for convenience.
  - A buffer object with space for 10 touching hits and one blocking hit can now be conveniently declared as PxRaycastBufferN<10>.

- PxRaycastHit and PxSweepHit now inherit from PxLocationHit (PxSceneQueryImpactHit).
• bool PxLocationHit::hadInitialOverlap() function was added to determine if a swept shape was overlapping at sweep distance=0 or if a raycast hit a shape at distance=0.

• Functionality of PxSceneQueryFlag::eINITIAL_OVERLAP and PxSceneQueryFlag::eINITIAL_OVERLAP_KEEP was replaced with 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 overlap(); 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 suppressed.

• Sweeps in 3.3 execute using a new faster code path, in some cases with reduced precision. If you encounter precision issues not previously experienced in earlier versions of PhysX, use ePRECISE_SWEEP to enable the backwards compatible more accurate sweep code.

Snippets demonstrating API migration:

Former raycastSingle call:

```cpp
PxRaycastHit hit;
bool hadHit = scene->raycastSingle(..., hit, ...);
if (hadHit) doStuff(hit);
```

Is now:

```cpp
PxRaycastBuffer buf;
Bool hadHit = scene->raycast(..., buf, ...);
if (hadHit) doStuff(buf.block);
```
Former raycastAny call:

```cpp
PxSceneQueryHit hit;
bool hadHit = scene->raycastAny(hit);
if (hadHit) doStuff(hit);
```

Is now:

```cpp
PxRaycastBuffer buf; // declare a hit buffer with room for a single hit
PxFilterData fdAny; fdAny.flags |= PxQueryFlag::eANY_HIT;
bool hadHit = scene->raycast(buf, PxHitFlags(), fdAny);
if (hadHit) doStuff(buf.block);
```

Former Multiple call:

```cpp
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:

```cpp
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 context refers to blocking or touching hits
    doAnyHit(buf.getAnyHit(i));

Former batch query memory setup code in 3.2:

```cpp
const PxU32 maxRaycastHits = 16, maxRaycastQueries = 8;
PxRaycastQueryResult* resultBuffer = new PxRaycastQueryResult[maxRaycastQueries];
PxRaycastHitBuffer* hitBuffer = new PxRaycastHit[maxRaycastHits];
PxBatchQueryDesc desc;  // required no arguments, there was no safety check for maximum number of queries per batch (not hits per query)
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 allowed but there was no overflow check
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 & PxSceneQueryFlag::processHit(resultBuffer[iResult].hits[iHit], isBlocking));
    }
}
```

Batch query setup code in 3.3:

```cpp
const PxU32 maxRaycastHits = 16, maxRaycastQueries = 8;
PxBatchQueryDesc desc(maxQueries, 0, 0);  // note the new required maximum number of queries for each type (raycasts, overlaps, sweeps)
PxBatchQuery* bq = scene->createBatchQuery(desc);

PxRaycastQueryResult* resultBuffer = new PxRaycastQueryResult[maxRaycastQueries];
PxBatchQueryMemory mem(maxQueries, 0, 0);  // maximum number of queries per type
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 SPUs 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 SPUs. Using a separate PPU thread, this design allows higher flexibility, such as executing batches on multiple SPU and PPU threads simultaneously, better control of parallel execution, and allows the user to fine tune thread load balancing. Here's one possible way to run batch queries on multiple SPUs 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 either `PxBatchQueryDesc::runOnSpu` or `PxBatchQuery::setRunOnSpu(bool)` function. If the batch query is executed on SPU:

```cpp
PxBatchQueryDesc desc;
...
desc.runOnSpu = true;
...```

By default, the batch query is executed on SPU.
Core PhysX

- The following methods require that the corresponding objects have been added to a scene. Calling these methods for objects which are not in a scene will result 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 `PxRigidDynamic`, `PxArticulation`, `PxCloth`
  - `PxScene::resetFiltering()` and the deprecated counterparts on `PxShape` and `PxParticleBase`

- The sleep behavior of dynamic rigid bodies has changed significantly. Among the changes are:
  - The `wakeUp()` method of `PxRigidDynamic` and `PxArticulation` has lost the wake counter parameter. Use the newly introduced `setWakeCounter()` instead to set a specific value.
  - Putting a dynamic rigid actor to sleep will clear any pending force updates.
  - Switching a dynamic actor to kinematic will put the actor to sleep immediately.
  - Switching a kinematic actor back to dynamic will not affect the sleep state (previously the actor was woken up).
  - Calling `wakeUp/putToSleep()` on a kinematically controlled object is not valid any longer. The sleep state of a kinematic actor is solely defined based on whether a target pose has been set (see API documentation of `isSleeping()` for details).
  - A call to `PxRigidBody::setCMassLocalPose()` does not wake the actor anymore. Add a call to `PxRigidBody::wakeUp()` to get the old behavior back.
Note: this also affects related methods in PhysX
PxRigidBodyExt::updateMassAndInertia() etc.

- Adding or removing a PxConstraint to/from the scene does
  connected actors up automatically anymore (note: this applies
  to PhysX Extensions as well).
- If a non-zero velocity or force is
  PxRigidBody::setLinearVelocity(), ::setAngularVelocity(),
  ::addTorque(), the actor will get woken up automatically even
  parameter is false.
- PxRigidBody::clearForce() and ::clearTorque() do not have
  parameter, to optionally wake the actor up, anymore. These
  change the sleep state any longer. Call ::wakeUp() subseq
  old default behavior.

- Shapes may now be shared between actors. This change has sev

  - PxShape::getActor() now returns a pointer rather than a
    shape is shareable, the pointer is NULL.
  - The following methods of PxShape have been removed:
    raycast(), sweep(), overlap(), getWorldBounds(). Replace
    found in PxShapeExt.
  - PxShape now has the same reference counting semantics
    materials, so that release() releases the user reference, an
    reference is released, the shape is destroyed.
  - Shapes created through PxRigidActor::createShape() are
    automatically when the actor is released. However, after
    deserializing such a shape, the regular reference counting s
    return results from scene queries which previously specify an actor also.

- Shape local transforms cannot be specified on shape creation an
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 PxActor 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:

```cpp
class MyObserver : public PxObserver
{
public:
    virtual void onRelease(const PxObservable& observable);
}

MyObserver myObs;
PxRigidDynamic* d = create...;
d->registerObserver(myObs);
```

new API:

```cpp
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, PxDeletionEventFlag::kRelease);
PxBase* b = d;
physics->registerDeletionListenerObjects(myDelListener, &b, 1);
```

- The contactStream in PxContactPair is now stored in a variable-size contact stream. This is used to save memory. As such, you cannot treat it to a PxContactPoint* and access the data. Instead, you must use 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 PxSceneDesc::frictionType`.
  - `PxSceneDesc::contactCorrelationDistance` has been deprecated, and it no longer has an influence on how many friction anchors are created in a single frame, only on when they are removed in later frames. This may cause a very minor change in friction behavior.

- `PxShape::resetFiltering()` and `PxParticleBase::resetFiltering()` have been deprecated. Please use one of the new overloaded methods `PxScene::resetFiltering()` instead.

- `PxClientBehaviorBit` and `PxActorClientBehaviorBit` have been renamed to `PxClientBehaviorFlag` and `PxActorClientBehaviorFlag` respectively.

- `PxActorTypeSelectionFlag` and `PxActorTypeSelectionFlags` have been renamed to `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::Enum`

- Discrete contact reports are no longer produced for `PxPairFlag::eDETECT_DISCRETE_CONTACT` raised in the filter shader. Previously, discrete contact generation would always have been performed regardless of the presence of the `PxPairFlag::eDETECT_DISCRETE_CONTACT` flag.
potentially improves performance when using specific shapes for collision which would have previously generated discrete contacts and then ignored them in the solver.

- Trigger reports are no longer produced for pairs without \texttt{PxPairFlag::eDETECT_DISCRETE_CONTACT} raised in the filter shader. \texttt{PxPairFlag::eTRIGGER_DEFAULT} has been modified to include \texttt{PxPairFlag::eDETECT_DISCRETE_CONTACT} flag.
PhysX Extensions

- Joint limits have been more carefully separated into PxJointAngularLimitPair, PxJointLinearLimitPair.
- PxJoint::getType() is deprecated. Joints now inherit from PxBase and getConcreteType() replaces getType(). Alternatively, to dynamically cast to a particular joint type, use e.g. joint->is<PxD6Joint>() which will return joint if the type matches, otherwise NULL.
- The parameter 'spring' in joint limits and drives has been renamed 'stiffness'.
- Dominance settings no longer apply to joints. To achieve this effect, use setInvMassScale. For example if actor0 in the joint is to affect actor1, use setInvMassScale0(0.0f), setInverseInertiaScale0(0.0f).
PhysX Character Controller

- When creating a PxControllerManager, a reference to a PxScene has to be provided. As a consequence, creating a controller from a PxControllerManager now only requires the controller descriptor as an argument.
- On PxControllerManager::release(), all associated PxObstacleContext instances get deleted automatically. Make sure to not access PxObstacleContext instances after the corresponding manager has been released.
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_INPUT
  PxVehicleDrive4WControl::eMAX_NB_DRIVE4W_ANALOG_INPUT
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_LEFT
  - PxVehicleDriveTankWheelOrder::e1ST_FROM_FRONT_LEFT
- PxVehicleDriveTank::eTANK_WHEEL_1ST_FROM_FRONT_RIGHT
  - PxVehicleDriveTankWheelOrder::e1ST_FROM_FRONT_RIGHT
- PxVehicleDriveTank::eTANK_WHEEL_2ND_FROM_FRONT_LEFT
  - PxVehicleDriveTankWheelOrder::e2ND_FROM_FRONT_LEFT
- PxVehicleDriveTank::eTANK_WHEEL_2ND_FROM_FRONT_RIGHT
  - PxVehicleDriveTankWheelOrder::e2ND_FROM_FRONT_RIGHT
- PxVehicleDriveTank::eTANK_WHEEL_3RD_FROM_FRONT_LEFT
  - PxVehicleDriveTankWheelOrder::e3RD_FROM_FRONT_LEFT
- PxVehicleDriveTank::eTANK_WHEEL_3RD_FROM_FRONT_RIGHT
  - PxVehicleDriveTankWheelOrder::e3RD_FROM_FRONT_RIGHT
- PxVehicleDriveTank::eTANK_WHEEL_4TH_FROM_FRONT_LEFT
  - PxVehicleDriveTankWheelOrder::e4TH_FROM_FRONT_LEFT
- PxVehicleDriveTank::eTANK_WHEEL_4TH_FROM_FRONT_RIGHT
  - PxVehicleDriveTankWheelOrder::e4TH_FROM_FRONT_RIGHT
- PxVehicleDriveTank::eTANK_WHEEL_5TH_FROM_FRONT_LEFT
  - PxVehicleDriveTankWheelOrder::e5TH_FROM_FRONT_LEFT
- PxVehicleDriveTank::eTANK_WHEEL_5TH_FROM_FRONT_RIGHT
  - PxVehicleDriveTankWheelOrder::e5TH_FROM_FRONT_RIGHT
- PxVehicleDriveTank::eTANK_WHEEL_6TH_FROM_FRONT_LEFT
  - PxVehicleDriveTankWheelOrder::e6TH_FROM_FRONT_LEFT
- PxVehicleDriveTank::eTANK_WHEEL_6TH_FROM_FRONT_RIGHT
A new struct has been introduced to hold the enumerated list that began with `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_INPUTS` ->
PxVehicleDriveTankControl::eMAX_NB_DRVETANK_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::eDRIVE4
  - eVEHICLE_TYPE_DRVETANK -> 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_FORCE
- PxVehicleGraph::eCHANNEL_TIRE_LAT_SLIP
  PxVehicleWheelGraphChannel::eTIRE_LAT_SLIP
- PxVehicleGraph::eCHANNEL_NORM_TIRE_LAT_FORCE
  PxVehicleWheelGraphChannel::eNORM_TIRE_LAT_FORCE
- PxVehicleGraph::eCHANNEL_NORM_TIRE_ALIGNING_MOMENT
  PxVehicleWheelGraphChannel::eNORM_TIRE_ALIGNING_MOMENT
- PxVehicleGraph::eMAX_NUM_WHEEL_CHANNELS
  PxVehicleWheelGraphChannel::eMAX_NB_WHEEL_CHANNELS

- A new struct has been introduced to hold the enumerated list
  PxVehicleGraph::eCHANNEL_ENGINE_REVS. The changes are

  - PxVehicleGraph::eCHANNELngine_REVS
    PxVehicleDriveGraphChannel::eENGINE_REVS
  - PxVehicleGraph::eCHANNELngine_DRIVE_TORQUE
    PxVehicleDriveGraphChannel::eENGINE_DRIVE_TORQUE
  - PxVehicleGraph::eCHANNELngine_CLUTCH_SLIP
    PxVehicleDriveGraphChannel::eCLUTCH_SLIP
  - PxVehicleGraph::eCHANNELngine_ACCEL_CONTROL
    PxVehicleDriveGraphChannel::eACCEL_CONTROL
  - PxVehicleGraph::eCHANNELngine_BRAKE_CONTROL
    PxVehicleDriveGraphChannel::eBRAKE_CONTROL
  - PxVehicleGraph::eCHANNELngine_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 PxVehicleUpdates function argument. A simple example of how to construct, use, and 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 and stored in an array:

```cpp
void updateVehicles(const PxF32 timestep, const PxVec3& gravity,
                     const PxVehicleDrivableSurfaceToTireFrictionPairs& fricPairs,
                     PxVehicleWheels** vehicles, PxU32 numVehicles, std::vector
{  
    //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 PxWheelQueryResult;
PxU32 wheelCount = 0;
for(PxU32 i = 0; i < numVehicles; i++)
{
    vehicleWheelQueryResults[i].nbWheelQueryResults = vehicles[i]->mWheelsSimData.getNbWheels();
    vehicleWheelQueryResults[i].wheelQueryResults = &wheelQueryResults;
    wheelCount += vehicles[i]->mWheelsSimData.getNbWheel;
}

//Update the array of vehicles.
PxVehicleUpdates(timestep, gravity, fricPairs, numVehicles, 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].wheelCount +=
            vehicles[i]->mWheelsSimData.getNbWheels());
        hitActors.push_back (vehicleWheelQueryResults[i].wheelQue

        delete[] vehicleWheelQueryResults;
        delete[] wheelQueryResults;
    }

    The following accessor to non-persistent data associated with each wheel replaced as follows

    ○ PxVehicleWheelsDynData::getSuspLineStart
    ○ PxWheelQueryResult::suspLineStart
    ○ PxVehicleWheelsDynData::getSuspLineDir
    ○ PxWheelQueryResult::suspLineDir
- PxVehicleWheels::getSuspRaycast -> PxWheelQueryResult::suspLineStart, PxWheelQueryResult::suspLineDir, PxWheelQueryResult::suspLineLength
- PxVehicleWheelsDynData::getTireDrivableSurfaceShape -> PxWheelQueryResult::tireContactShape
- PxVehicleWheelsDynData::getTireDrivableSurfaceMaterial -> PxWheelQueryResult::tireSurfaceMaterial
- PxVehicleWheelsDynData::getTireDrivableSurfaceType -> PxWheelQueryResult::tireSurfaceType
- PxVehicleWheelsDynData::getTireDrivableSurfaceContactPoint -> PxWheelQueryResult::tireContactPoint
- PxVehicleWheelsDynData::getTireDrivableSurfaceContactNormal -> 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 -> PxWheelQueryResult::lateralSlip
- PxVehicleWheels::getSteer -> PxWheelQueryResult::steerAngle
- PxVehicleWheels::isInAir -> PxWheelQueryResult::isInAir

- PxVehicleWheels::setWheelShapeMapping and PxVehicleWheels::getWheelShapeMapping have been
PxVehicleWheelsSimData::setWheelShapeMapping
PxVehicleWheelsSimData::getWheelShapeMapping

- PxVehicleWheels::setSceneQueryFilterData
  PxVehicleWheels::getSceneQueryFilterData have been moved
  PxVehicleWheelsSimData::setSceneQueryFilterData
  PxVehicleWheelsSimData::getSceneQueryFilterData

- PxVehicle4WEnable3WTadpoleMode and PxVehicle4WEnable3'WTadpoleMode now take an extra function argument: a non-const reference to a PxVehicleWheelsDynData

- PxVehicleWheels::isInAir() has been replaced with PxVehicleWheelsQueryResult& vehWheelQueryResults

- PxVehicleDrive4WSmoothAnalogRawInputsAndSetAnalogInputs now takes an extra function argument "const bool isVehicleInAir". This can be calculated using the function PxVehicleIsInAir

- To improve api consistency PxVehicleTelemetryData::getNumWheelGraphs has been replaced with PxVehicleTelemetryData::getNbWheelGraphs

- To improve api consistency
  PX_MAX_NUM_WHEELS
  PX_MAX_NB_WHEELS

- To improve api consistency
  PxVehicleGraph::eMAX_NUM_TITLE_CHARS
  PxVehicleGraph::eMAX_NB_TITLE_CHARS

- PxVehicleTireData::mCamberStiffness has been replaced with PxVehicleTireData::mCamberStiffnessPerUnitGravity. PxVehicleTireData::mCamberStiffnessPerUnitGravity should be set so that it is equivalent to the old value of PxVehicleTireData::mCamberStiffness divided by the magnitude of gravitational acceleration (PxScene::getGravity().magnitude()). The advantage of using PxVehicleTireData::mCamberStiffnessPerUnitGravity is that it is independent of length scale.
- PxVehicleComputeTireForceDefault has been removed from the public vehicle api. Custom tire shaders that call PxVehicleComputeTireForceDefault are best implemented by taking a copy of PxVehicleComputeTireForceDefault and calling the copy instead.
The mechanism to activate CCD per shape has changed. PxShapeFlag::eUSE_SWEPT_BOUNDS that was used in 3.2 to activate swept bounds per shape has been removed. In its place is PxRigidBodyFlag::eENABLE_CCD that is set per rigid actor. Setting this flag for an actor in 3.3 has approximately the same effect as PxShapeFlag::eUSE_SWEPT_BOUNDS on all the actor's shapes in 3.2.

PxPairFlag::eSWEPT_INTEGRATION_LINEAR has been replaced with PxPairFlag::eCCD_LINEAR in PhysX 3.3.

PxSceneFlag::eENABLE_SWEPT_INTEGRATION flag in 3.2 has been replaced with PxSceneFlag::eENABLE_CCD in PhysX 3.3.

A simple example of how to enable CCD on a specific shape is given below. This 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 only used in CCD. The simulation filter shader shown here requires that the shapes be flagged with eCCD_RESPONSE to generate a CCD 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_CONTACTS). A final remark is that the following shader requires that shapes of static actors and collision response from pairs that involve a static actor are enabled with eDISCRETE_RESPONSE | eCCD_RESPONSE:

```cpp
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(PxTransform
    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(PxVisualDebuggerFlag);

- A new function has been introduced to send error stream to PVD:
  PxVisualDebugger::sendErrorMessage(( PxErrorCode::Enum code, const char* message, const char* file, PxU32 line);)

- The following functions were renamed:
  PxVisualDebugger::getPvdConnectionFactory() -> PxVisualDebugger::getPvdConnection();
  PxVisualDebugger::getPvdConnection() -> PxVisualDebugger::getPvdDataStream();

- The PVD connect function changed to the same method as previous:
  PxVisualDebuggerExt::connect -> PxVisualDebuggerExt::createConnection;

- The constraint, contacts and scene queries visualizing can all be configured with PxVisualDebuggerFlag in 3.3. Here is an example for how to enable the contacts:

  mPhysics->getVisualDebugger() ->
  >setVisualDebuggerFlags(PxVisualDebuggerFlag::eTRANSMIT_CONSTRAINTS);
There have been substantial changes to the PhysX 3.3 cloth solver that improve performance and behavior. This has resulted in a reorganization of how edge constraints are stored and processed in the cloth fabric. Prior to PhysX 3.3 the cloth solver used fibers to organize edge constraints into independent groups. In PhysX 3.3 it is no longer necessary to decompose constraints into fibers, instead edge constraints now exist individually and are solved in larger, independent sets. Interface changes are detailed below:

- Previously there were multiple solver types to choose from for different types of constraints such as eFAST, eSTIFF, eBENDING, PxClothPhaseSolverConfig::SolverType). There is now one type of solver for all edge constraints, this is a flexible distance constraint with controls to adjust stiffness within certain ranges of compression and stretch (see PxClothStretchConfig). Behaviors such as bending are now achieved by the way distance constraints are arranged geometrically, rather than through a specialized bending solver.

- To reduce stretching a new constraint type has been added called 'tether' constraints. These constraints do not act along edges of the mesh, but as long range attachments between particles that enforce a maximum distance between them. See PxClothFabric::getTetherAnchors().

- Cloth cooking which was previously part of the PxCooking library has been moved to the extension library, see PxClothFabricCooker:

```cpp
// PhysX 3.2.x
cooking-&gt;cookClothFabric(meshDesc, gravity, outputStream);

// PhysX 3.3
PxClothFabricCooker cooker(meshDesc, gravity, useGeodesicTethers); cooker.save(outputStream, false);
```

- The PxClothCollisionData parameter has been removed from PxPhysx::createCloth(). 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::setStretchConfig()

- PxCloth::lockClothReadData() has been renamed to PxCloth::lockParticleData()

- PxClothFabricTypes.h has been removed, this header has been merged with PxClothFabric.h
**RepX Serialization**

Substantial changes were made to the PhysX 3.3 serialization interface collections and references between collections have been unified for serialization.

- The RepX and RepXUpgrader libraries have been removed. RepX functionality 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 instantiated with PxSerialization::createCollectionFromXml(). Empty collections can 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 new functionality in PxCollection, which now maps between serializable objects and PxSerialObjectId values.

- RepXExtension was removed. Serialization and deserialization of serializable types is achieved through the PxRepXSerializer interface.

- RepXUtility and PxToolkit functionality has been replaced with various functions in PxSerialization, PxCollection and PxScene.
  - A PxCollection with all PxPhysics-level objects such as shapes, meshes, or materials (formerly referred to as buffers) can be created with PxCollectionExt::createCollection(PxPhysics&).
  - Similarly PxCollectionExt::createCollection(PxScene&) can create a collection of PxScene-level objects.
  - Dependencies between objects and collections can
The objects of a collection can be added to PxScene::addCollection().

Operations on files are generally handled with abstract interfaces: PxOutputStream and PxInputData. Default implementations are PxDefaultFileOutputStream and PxDefaultFileInputData.

- RepXUpgrader::upgradeCollection was removed. RepX data can be converted to newer PhysX versions by deserializing and re-serializing:
  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. PxSerializationReg provided to PxInitVehicleSDK() for vehicle serialization, PxCloseVehicleSDK() for cleanup.

- Custom class RepX serialization is supported in 3.3, more information can be found in the Serialization documentation.
**Binary Serialization**

The binary serialization interface has been refactored and unified with the 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 class. A `PxSerializer` instance per serializable type is registered in the `PxSerializationRegistry`. All PhysX and PhysXExtension serializable types are registered by default.

- `PxCollection` has been reworked.
  - `PxCollection::serialize()` and `PxCollection::deserialize()` were replaced with `PxSerialization::createCollectionFromBinary()` and `PxSerialization::serializeCollectionToBinary()` in PhysXExtensions.
  - `PxSerializable::collectForExport()` has been replaced with `PxSerialization::complete()` helps to add required objects for a collection.
  - `PxSerializable::isSerializable()` should be used to check whether a collection can be successfully serialized.

- `PxUserReferences` was removed: `PxCollection` instances can now resolve dependencies between collections on deserialization. `PxSerialization::complete()` supports creating collections with external dependencies.

- `PxSerialObjectRef` has been replaced with `PxSerialObjectId`.

- `PxCollectForExportSDK()` and `PxCollectForExportScene()` functions were replaced with...
with `PxCollectionExt::createCollection(PxPhysics& physics)` and `PxCollectionExt::createCollection(PxScene& scene)`.

- `PxDumpMetaData()` was replaced with `PxSerialization::dumpBinaryMetadata()`.

- The `PxBinaryConverter` moved from `PhysXCooking` to `PxCooking::createBinaryConverter()` was replaced with `PxSerialization::createBinaryConverter()`.

- `PhysXVehicles` supports binary serialization. `PxSerializationRegistry` needs to be provided to `PxInitVehicleSDK()` for vehicle serialization, `PxCloseVehicleSDK()` for cleanup.

- Custom class binary serialization is supported in 3.3, more information please read `Serialization`. 
PhysX TaskManager

- The pxtask namespace has been removed and all its types are now included in the physx namespace with a Px prefix, for example pxtask::LightCpuTask has become physx::PxLightCpuTask
- This guide highlights all significant parts of the API that have changed.
- 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 PxShared library, it is versioned separately. PxCreateFound 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_REPORTS**

PxSceneFlag::eDEPRECATED_TRIGGER_TRIGGER_REPORTS is deprecated, and will be removed in PhysX 3.5. More details are mentioned under Core Phys

**PhysX particles**

The PhysX particle feature has been deprecated in PhysX version 3. The standalone 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 and receives a pointer to a PxPvd object, used for connecting PhysX with the visual debugger.

- PxActor::isRigidStatic, isRigidDynamic, isParticleSystem, isArticulationLink, isCloth, isRigidActor, isRigidBody, isParticle removed. Use corresponding PxBase::is() with class template parameter casting.

- PxContactPairFlag::eINTERNAL_HAS_FACE_INDICES is obsolete and removed.

- Trigger shapes will no longer send notification events for interactions with other trigger shapes. For PhysX 3.4 there is the option to re-enable the PxSceneFlag::eDEPRECATED_TRIGGER_TRIGGER_REPORTS flag; this option will no longer be available in PhysX 3.5. It is recommended to use PxSceneFlag::eDEPRECATED_TRIGGER_TRIGGER_REPORTS and instead use a non-trigger shape, both with the same geometry and local pose, notifications for overlaps between trigger shapes.

- Implementations of PxSimulationEventCallback will have to provide an implementation of the newly added method onAdvance() to avoid compilation errors.

- The deprecated method PxPhysics::createHeightField(const PxHeightFieldDesc&) has been removed. Please use PxCooking::createHeightField(const PxHeightFieldDesc&, PxPhysicsInsertionCallback&) instead. The insertion callback can be obtained through PxPhysics::getPhysicsInsertionCallback().

- The deprecated flag PxActorTypeSelectionFlag/PxActorTypeSelectionFlags has been removed. Please use PxActorTypeFlag/PxActorTypeFlags instead.
• The deprecated class PxFindOverlapTriangleMeshUtil has been removed. Please use PxMeshOverlapUtil instead.

• The deprecated flag PxConstraintFlag::eREPORTING has been removed. Force 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 heightfield collision is 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, contactPoints and contactImpulses rather than reporting a single pointer that the PxContactStreamIterator parses. The interface for PxContactStreamIterator has been modified accordingly. See the PxContactPair::extractContacts implementation
further guidance on how to iterate over this contact data if required.
Contact Generation

- PCM contact generation is now used by default. Legacy SAT-based contact generation can be re-enabled by clearing the PxSceneFlag::eENABLE_PCM from PxSceneDesc::flags.
- Unified heightfields are now the default heightfield collision approach, mirroring the way in which mesh contact gen functions so permits meshes and heightfields to be used interchangeably with negligible behavioral difference. The legacy heightfield collision approach can be used by calling PxRegisterLegacyHeightFields().
- When unified heightfields are in use, the bounds of heightfield shapes are extruded by "thickness". If legacy heightfield collision is used, the bounds are extruded by thickness.
PhysX Cooking

- The deprecated PxMeshPreprocessingFlag::eREMOVE_UNREFERENCED_VERT ::eREMOVE_DUPLICATED_TRIANGLES have been removed. Meshes get cleaned up by default unless PxMeshPreprocessingFlag::eDISABLE_CLEA
- PxCookingParams::meshSizePerformanceTradeOff and PxCookingParams::meshCookingHint have been moved to PxBVH since they only affect the BVH33.
- The PxGaussMapLimit.h file has been removed. The PxGetGaussMapVertexLimitForPlatform function has been moved but the function is now deprecated, along with the PxPlatform enum. Now an explicit PxCookingParams::gaussMapLimit parameter. As far as transition to PhysX 3.4 is concerned there is nothing to do other than removing includes of PxGaussMapLimit.h, and perhaps including PxCooking.h instead if necessary.
- Legacy convex hull (PxConvexMeshCookingType::eINFLATION_INCREMENTAL_HULL) uses inflation in all cases. To cook a convex mesh without inflation, PxConvexMeshCookingType::eQUICKHULL algorithm must be used. It does not support inflation.
Reference Counting

- In previous releases, isReleasable() for shareable objects (shapes, convex meshes, cloth fabrics, materials and heightfields) would return false once release() had been called on the object, which was only allowed once. In 3.4, reference counts can be manually incremented with acquireReference() and decremented with release(), and so the fact that release() has called once is not a reliable indicator of whether it can be called again.
- As a consequence of the above, applications must ensure they own at least one counted reference to each shareable object in a collection before calling PxCollectionExt::releaseObjects. The main case in which this might be different in 3.4 is when using PxRigidActor::createShape(), since in that case only the actor has a counted reference to the shape. In this specific case, the releaseExclusiveShapes to PxCollectionExt::releaseObjects may be useful.
- Since there is no unique user release for shareable objects, they do not generate USER_RELEASE events when release() is called.
**PhysX Visual Debugger**

- PxVisualDebugger is deprecated, and new PxPvd has been introduced. More details are mentioned in *PhysX Visual Debugger (PVD)*.
Scene queries

- PxPruningStructure enum has been renamed to PxPruningStructureType.
- Deprecated type PxSceneQueryHit has been removed. Please use PxQueryHit 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 PxHitFlag(s) instead.
- Deprecated scene query functions have been removed (e.g. PxScene::raycastAny(), etc). To make the transition easier they are still available in PxSceneQueryExt.h as part of PhysXExtensions. A previous call to PxScene::raycastAny(...) should now use PxSceneQueryExt::raycastAny(PxScene, ...), or PxScene::raycast(...).
- PxHitFlag::eFACE_INDEX was introduced. In order to receive the face index in sweeps against convex geometry, the flag needs to be set.
- PxHitFlag::eDISTANCE has been deprecated, since the distance is always needed and its computation cannot be skipped. Please simply avoid using this flag. The flag has no effect and it will be removed in the next version.
- The "anyHit" parameter of the PxGeometryQuery::raycast() and PxShapeExt::raycast() functions has been removed. Please use PxHitFlag::eMESH_ANY instead.
- PxMeshQuery::sweep() now respectsPxHitFlag::eMESH_BOTH. Previously, using that flag when calling that function, it was ignored. Starting with version 3.4, different results may be generated compared to previous versions. If keeping the previous behaviour is important, please disable PxHitFlag::eMESH_BOTH.
PxMeshQuery::sweep() calls.

- Batched scene queries are marked as deprecated and will be system in future releases.
- Volume cache feature is marked as deprecated, it will be removed in future releases.
- Spatial index feature is marked as deprecated, it will be removed in future releases.
PxExtensions

- The signatures for the PxComputeMeshPenetration and PxComputeHeightFieldPenetration functions have changed. The old functions 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).
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
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 higher than the step 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_REPORTS
    • PhysX particles
    • PhysX cloth
• Core PhysX
• Contact Generation
• PhysX Cooking
• Reference Counting
• PhysX Visual Debugger
• Scene queries
• PxExtensions