

Built December 15, 2017

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

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

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

About this User Guide

This Guide will help the reader to understand the PhysX-3 SDK and it Guide presents an overview of the features and implementation of the F 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 c

The Guide does not attempt to explain the details of the API, and the should refer to the PhysX API Reference Documentation. (See Phy Documentation directory under the main directory where the Phys2 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 will behavior of all matter, everything in the entire universe, using conceptime, energy, inertia, momentum and force. In physics, space is a infinitely in three dimensions, and can be divided into infinitely small usine precision. In other words, positions in physics space are described numbers in a 3-dimensional Cartesian coordinate system. In contrast, simulation space are vectors of single-precision floating point numbers.

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

Because of such approximations a PhysX simulation is subject to limit seen in ordinary physics, and later sections in this Guide will highligh wherever they are likely to concern the user. PhysX is best suited interactive 3D applications where performance and scalability are multiple precision. Here "quasi-real time" means that advancing a PhysX simple time step, say 1/60 second, will take less than that amount of time on a first the performance of the hardware platform is sufficient for the simulation. That the PhysX SDK is more widely used in computer and vescientific or engineering applications is both a cause and an effect choices. Consequently this Guide usually refers to PhysX in the content the game world', 'rigid body game objects', 'the character', etc.

World and Objects

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

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

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

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

What are PhysX Snippets?

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

The folder {SDK Root}/Snippets/compiler/{platform} contains the Snippet Snippets/compiler/vc14win64/Snippets.sln

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

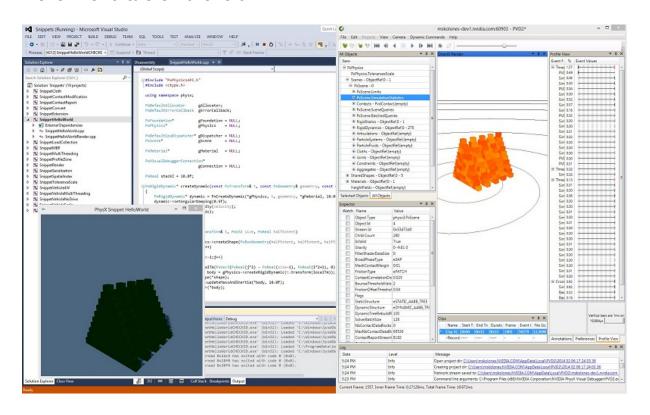
HelloWorld: PhysX Basics

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

The primary code for SnippetHelloWorld is fou Root}/Snippets/SnippetHelloWorld/SnippetHelloWorld.cpp.

Using PhysX Visual Debugger with SnippetHelloWorld

As is the case with any Snippet built against PROFILE, CHEC configurations of the PhysX runtime, HelloWorld will automatically cor Visual Debugger if that application is already running when the Sni launched. For Snippets without rendering, PVD provides an easy w contents of the PhysX scene presented in the Snippet. In the screer PhysX Visual Debugger appears on the right hand side, while Visual 1 Hello World are on the left.



NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »



Build Settings

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

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

Build Configurations

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

- the *debug* build can be useful for error analysis, but contains ass development which some customers may find too intrusiv Optimizations are turned off for this configuration.
- the *checked* build contains code to detect invalid parameters, Al and other incorrect uses of the API which might otherwise cause r or failures in simulation.
- the profile build omits the checks, but still has PVD and memory ins
- the *release* build is built for minimal footprint and maximum spechecks and instrumentation.

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

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

Note: PhysX libraries of different build configurations (e.g. the DEBUC PhysXVehicle and the CHECKED version of PhysXVisualDebuggerSD 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 par your project makefile or IDE.

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

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

This will include the entire PhysX API including core, extensions, veh possible to include subsets of the SDK if preferred, for example:

#include "vehicle/PxVehicleSDK.h"

Libraries

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

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

Note: The static libraries we provide with the Windows binary distribu against the Multi-Threaded static C Run-Time (CRT) libraries. This mean 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 recompiled using different CRT settings.

Redistribution

On the Windows platform, you need to redistribute some of our DLLs to 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 data on the fly.
- PhysX3GPU_*.dll is only needed if your application runs some GPU.
- PhysX3CharacterKinematic_*.dll is only needed if your approximation 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.

NVIDIA PhysX SDK 3.4.1 Documentation » previous | next User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

Introduction

This chapter covers the basic patterns common to the PhysX applic interface (API.) We are committed to keeping this API stable and bac from one minor release to the next, to protect the investment you make 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 prefix: the PhysX Visual Debugger connection library which has the pre

The PhysX libraries also expose some classes and functions that are nearly the API. These are primarily containers and platform abstractions that are nearly PhysX libraries which are distributed as source, and are also used in can be recognized because they do not have the Px prefix. Even 1 principle accessible to users, they are largely undocumented and we compatibility of this code between PhysX versions. For that reasc strongly against their use in applications.

Memory Management

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

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

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

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

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

You can place PhysX objects in memory owned by the application u deserialization mechanism. See *Serialization* for details.

As an alternative to instrumenting the allocator, you can obtain detailed memory allocation in the PhysX Visual Debugger (see: PhysX Visual De

Error Reporting

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

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

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

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

Math Classes

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

Some points to note are:

- PxTransform is a representation of a rigid body transform as a rotal a position vector, and PhysX functions which take transforms all us
- PxPlane is a homogeneous plane equation: that is, the construrepresents the equation n.x + d = 0.

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

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

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

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

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

// draw model
glPopMatrix()
```

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

Connecting PhysX Objects with User Application Objects

Often an application needs to associate PhysX objects with applicatio logic or rendering purposes. An easy way to connect a single user application as PhysX object is to use the *userData* member provided by the most classes (*PxActor::userData*, *PxShape::userData*, *PxMaterial::userData*, 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 calle provides mechanisms for type-safe down-casting between interface type cast from a PxActor to a PxRigidDynamic, use the following idiom:

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

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

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

- getConcreteType() provides an integer value which corresponds to 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 multipl scene graph. For example, a PxConvexMesh may be referenced by objects, each sharing the same geometry but associated with different a types are PxTriangleMesh, PxHeightField, PxConvexMesh, PxMater and PxShape. Each object of these types has a reference count. The counting are as follows:

- when an object is created from PxPhysics, it has a reference count
- when an object's reference count reaches 0, the object is destroyed
- when a new counted reference is created, the reference cou
 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() m reference count is decremented.
- when an object is created through deserialization, its reference c number of counted references that exist to the object.

The initial reference count of 1 ensures the object is not destroyed ι 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() actor with PxRigidActor::attachShape(), it has a reference count of 2. shape's release() method, it has a reference count of 1. When the act the shape is detached from the actor, the reference count is decremer now 0, the shape is destroyed.

The acquireReference() method increments the reference count example, when a spatial query returns a reference to a mesh shape, an that result to another thread for deferred processing, incrementing the ensure that even if the shape referencing the mesh is released, the exist.

Note: subtypes of PxGeometry do not have counted references to the they point, e.g. when PxConvexMeshGeometry points to a PxConvexN 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 des actors (see *Shapes* and *Reference Counting of Deserialized Objects*)

Using Different Units

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

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

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

You can also set the typical object mass in PxTolerancesScale.

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

Assertions

PhysX uses the PX_DEBUG macro to enable or disable assertions. 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 simulatic same sequence of API calls applied from the point where a scene is (and the same responses from simulation callbacks which modify removing all the objects from a scene is not in general sufficient to repurpose.

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

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

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

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Introduction

The first step in using the PhysX SDK in a program is the initializati objects. These objects can be released when PhysX is no long resources. This chapter describes how to do this.

Foundation and Physics

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

```
static PxDefaultErrorCallback gDefaultErrorCallback;
static PxDefaultAllocator gDefaultAllocatorCallback;

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

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

Usually, the allocator callback and error callback are specific to the approvides default implementations that make it easy to get standanagement and *Error Reporting* for more details of these callbacks. Code supports an advanced memory allocator that tracks allocations ins but we have omitted that detail here.)

Now create the top-level *PxPhysics* object:

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

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

Cooking

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

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

The PxCookingParams struct configures the cooking library to target use non-default tolerances or produce optional outputs. It is importan PxTolerancesScale values everywhere in your application (see *Using* more details).

The cooking library generates data through a streaming interface implementations of streams are provided in the PxToolkit library to refiles and memory buffers. Heightfield or Trianglemesh cooked mesh inserted into PxPhysics without serialization using the PxPhysicsInsedefault callback must be used and can be obtain 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 reasons or to avoid use of certain subsystems, such as those pertain Initializing the extensions library requires the PxPhysics object:

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

Optional SDK Components

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

- Articulations
- Height Fields
- Cloth
- Particles

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

Note that this will only save memory when linking PhysX as a static lik 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 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 a DLLs. In the PhysX and PhysXCooking DLLs, the choice of PxFoundation and PxPvdSDK is made as follows:

- If delay load hook is specified the PhysXCommon name, I 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 PxDelay following example:

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

```
virtual const char* getPhysXCommonDllName() const
        { return "PhysX3Common_x64_Test.dll"; }
    virtual const char* getPxFoundationDEBUGDllName() const
        { return "PxFoundationDEBUG_x64_Test.dll"; }
    virtual const char* getPxFoundationCHECKEDDllName() const
        { return "PxFoundationCHECKED_x64_Test.dll"; }
    virtual const char* getPxFoundationPROFILEDllName() const
        { return "PxFoundationPROFILE x64 Test.dll"; }
    virtual const char* getPxFoundationDllName() const
        { return "PxFoundation_x64_Test.dll"; }
    virtual const char* getPxPvdSDKDEBUGDllName() const
        { return "PxPvdSDKDEBUG x64 Test.dll"; }
    virtual const char* getPxPvdSDKCHECKEDDllName() const
        { return "PxPvdSDKCHECKED_x64_Test.dll"; }
    virtual const char* getPxPvdSDKPROFILEDllName() const
        { return "PxPvdSDKPROFILE x64 Test.dll"; }
    virtual const char* getPxPvdSDKDllName() const
        { return "PxPvdSDK_x64_Test.dll"; }
} qDelayLoadHook;
```

Now the hook must be set for PhysX, PhysXCooking, PhysXGpt PxPvdSDK:

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

PxGpuLoadHook

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

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

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

Now the hook must be set for PhysX:

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

PhysXCommon Secure Load

All PhysX DLLs distributed by NVIDIA are signed. The PhysXCommo 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 design all contained objects. The precise behavior depends on the object type refer to the reference guide for details. To shut down the extension function *PxCloseExtensions()*. To shut down physics, call release() object, and this will clean up all of the physics objects:

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

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

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

NVIDIA PhysX SDK 3.4.1 Documentation » previous | next User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

 -		

Introduction

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

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

Data Access from Multiple Threads

For efficiency reasons, PhysX does not internally lock access to its dat application, so be careful when calling the API from multiple application are as follows:

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

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

Scene Locking

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

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

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

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

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

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

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

Additionally there is an RAII helper class to manage these locks, see P>

Locking Semantics

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

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

- Lock upgrading is not supported a lockWrite() by a thread that he
 a read lock is not permitted. Attempting this in checked builds will I
 release builds it will lead to deadlock.
- Writers are favored if a thread attempts a lockWrite() while the re
 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 cha
 scene. This prevents writers being starved in the presence of multip
- If multiple writers are queued then the first writer will receive p writers will be granted access according to OS scheduling.

Note: *PxScene::release()* automatically attempts to acquire the wr necessary to acquire it manually before calling release().

Locking Best Practices

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

Clustering writes can also help reduce contention for the lock, as according to the lock, as a condition to the lock, as a condi

Asynchronous Simulation

PhysX simulation is asynchronous by default. Start simulation by calling

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

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

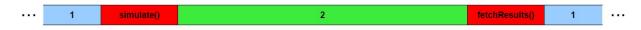
To wait until simulation completes, call:

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

The boolean parameter to fetchResults denotes whether the call should to complete, or return immediately with the current completion standocumentation 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").



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

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

the state of objects. Concurrent access from the user might corrupt the or lead to data races or inconsistent views in the simulation code. He code's view of the objects is protected from API writes, and any attrib updates are buffered to allow API reads. The consequences will be disthe 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 acces scene (with some exceptions, see further below). This includes add to/from a scene.

From the user perspective, API changes are reflected immediately. For velocity of a rigid body is set and then queried, the new velocity will be if an object is created while the simulation is running, it can be access other object. However, these changes are buffered so that the simulation object state as it was when *PxScene::simulate()* was called. For instanfilter data of an object while the simulation is running are ignored generation of the running step, and will only affect for the next simulation

When *PxScene::fetchResults()* is called, any buffered changes are flush by the simulation are reflected in API view of the objects, and API visible to the simulation code for the next step. User changes tak example, a user change to the position of an object while the simulation 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 simulatio

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

- PxSimulationEventCallback::onWake(), ::onSleep() events will n
 object is involved which got deleted/removed during the running sin
- PxSimulationEventCallback::onContact(), ::onTrigger() events will ç
 is involved which got deleted/removed during the running
 deleted/removed object will be marked as
 PxContactPairHeaderFlag::eREMOVED_ACTOR_0,

PxContactPairFlag::eREMOVED SHAPE 0,

PxTriggerPairFlag::eREMOVED_SHAPE_TRIGGER). Furt PxPairFlag::eNOTIFY_TOUCH_LOST, ::eNOTIFY_THRESHO events were requested for the pair containing the deleted/removec events will be created.

Support

Not all PhysX objects have full buffering support. Operations which ca simulation is in process are mentioned in the API documentation and th operations and reports an error. The most important exceptions are as f

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

Memory Considerations

The buffers to store the object changes while the simulation is runni demand. If memory usage concerns outweigh the advantage of readin parallel with simulation, do not write to objects while the simulation is ru

Multithreaded Simulation

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

Middleware products typically do not want to create CPU threads for the especially true on consoles where execution threads can have significa task model, the computational work is broken into jobs that are subm 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 rerespective dispatcher. There is a dispatcher for CPU tasks and GPU taskManager.

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

CpuDispatcher

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

PhysX includes a default CpuDispatcher implementation, but we pre implement this class themselves so PhysX and APEX can efficiently shouth the application.

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

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

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

The PxExtensions library has default implementations for all disp following code snippets are taken from SampleParticles and SampleB the default dispatchers are created. *mNbThreads* which *PxDefaultCpuDispatcherCreate* defines how many worker threads the (have.:

```
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(cudaContextM sceneDesc.gpuDispatcher = mCudaContextManager->getGpuDispatch
#endif
[...]
mScene = mPhysics->createScene(sceneDesc);
```

Note: Best performance is usually achieved if the number of threads i equal to the available hardware threads of the platform you are running more worker threads than hardware threads will often lead to worse pe platforms with a single execution core, the CPU dispatcher can be creat worker threads (PxDefaultCpuDispatcherCreate(0)). In this case all wo executed on the thread that calls PxScene::simulate(), which can be m

using multiple threads.

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

CpuDispatcher Implementation Guidelines

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

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

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

required.

For more details see the default CpuDispatcher implementation that cc PxExtensions package. It uses worker threads that each have their ox steal tasks from the back of other worker's queues (LIFO order) to distribution.

BaseTask

BaseTask is the abstract base class for all task types. All task run(executed on application threads, so they need to be careful with their 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 a this allows them to be discoverable. Tasks will be given a reference co are submitted, and the TaskManager::startSimulation() function decremed count of all tasks and dispatches all Tasks whose reference count reataskManager::startSimulation() is called, Tasks can set dependencies control the order in which they are dispatched. Once simulation ha possible to submit new tasks and add dependencies, but it is up to avoid race hazards. You cannot add dependencies to tasks that I dispatched, and newly submitted Tasks must have their reference of before that Task will be allowed to execute.

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

APEX uses the Task class almost exclusively to manage CPU resource defines a number of named Tasks that the modules use to schedule the start after LOD calculations are complete, finish before the PhysX scene

LightCpuTask

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

PhysX 3.x uses LightCpuTasks almost exclusively to manage CF example, each stage of the simulation update may consist of multiple peach of these tasks has finished execution it will decrement the reference task in the update chain. This will then be automatically dispatched by the stage of the simulation update may consist of multiple peach of these tasks has finished execution it will decrement the reference task in the update chain. This will then be automatically dispatched by the simulation update may consist of multiple peach of these tasks has finished execution it will decrement the reference tasks in the update chain. This will then be automatically dispatched by the simulation update may consist of multiple peach of these tasks has finished execution it will decrement the reference tasks in the update chain.

Note: Even when using LightCpuTasks exclusively to manage CPU re TaskManager startSimulation() and stopSimulation() calls must be mad step to keep the GpuDispatcher synchronized.

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

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

```
}
[◀]
```

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

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

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

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

It is safe to perform API read calls, such as scene queries, from mu simulate() is running. However, care must be taken not to overlap API r from multiple threads. In this case the SDK will issue an error, see information.

An example for explicit reference count modification and task dependen

```
// 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 parall
//
// a0-a1-a2
// \
// b0-b1-b2
// c0-c1-c2
// setup the 3 chains
```

```
for(PxU32 i = 0; i < 2; i++)
{
    a[i].setContinuation(&a[i+1]);
    b[i].setContinuation(&b[i+1]);
    c[i].setContinuation(&c[i+1]);
}
// b0 shall start after a1
b[0].startAfter(a[1].getTaskID());
// setup is done, now start all task by decrementing their refcou
// tasks with refcount == 0 will be submitted to the dispatcher (
for(PxU32 i = 0; i < 3; i++)
{
    a[i].removeReference();
    b[i].removeReference();
    c[i].removeReference();
}
4
```

NVIDIA PhysX SDK 3.4.1 Documentation » previous | next User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Introduction

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

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

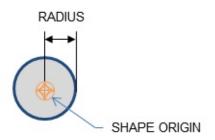
Geometries fall into two classes:

- primitives (PxBoxGeometry, PxSphereGeometry, PxPlaneGeometry) where the geometry object contains all of the day
- meshes or height fields (PxConvexMeshGeometry, PxTrian PxHeightFieldGeometry), where the geometry object contains a larger object (PxConvexMesh, PxTriangleMesh, PxHeightField res use these objects with different scales in each PxGeometry type them. The larger objects must be created using a *cooking* process, type below.

When passed into and out of the SDK for use as simulation geometropied into and out of a PxShape class. It can be awkward in this consider geometry without knowing its type, so PhysX provides a union-leading (PxGeometryHolder) that can be used to pass any geometry type by va height field) has a reference count that tracks the number of PxShapes reference the mesh.

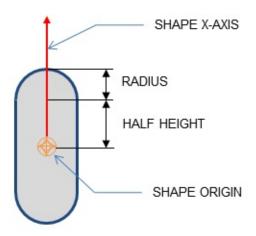
Geometry Types

Spheres



A PxSphereGeometry is specified by one attribute, its radius, and is cer

Capsules

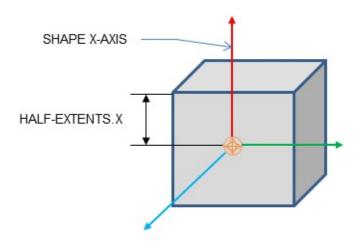


A PxCapsuleGeometry is centered at the origin. It is specified by a radic value by which its axis extends along the positive and negative X-axis.

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 shape and actor is otherwise the same as for the sphere:

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

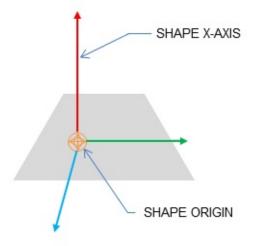
Boxes



A PxBoxGeometry has three attributes, the three extents halved:

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

Planes



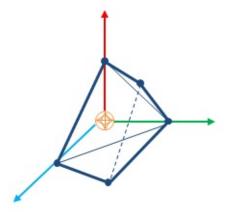
Planes divide space into "above" and "below" them. Everything "belowlide with it.

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

A PxPlaneGeometry has no attributes, since the shape's pose entirely 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 shap between them. A PxConvexMesh is a convex polyhedron represented and polygonal faces. The number of vertices and faces of a convex limited to 255.

Creating a PxConvexMesh requires cooking. It is assumed here that has already been initialized (see *Startup and Shutdown*.) The following to create a simple square pyramid.

First, define the vertices of the convex object:

Then construct a description of the convex data layout:

Now use the cooking library to construct a PxConvexMesh:

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

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

Finally, create a shape using a PxConvexMeshGeometry which instance

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

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

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

PxConvexMeshGeometry also contains flags to tweak some aspects of

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

PxConvexMeshGeometry also contains a variable called maxMargin. By be 3.4e38f. If the maxMargin is smaller than the margin amount calcu contact gen, it will choose the smallest margin for the shrunk shape to pupdate using GJK algorithm. In this case, application might notice son the vertex collision. If the maxMargin is set to be a small value, the visibility of these artefacts. If maxMargin is set to zero, PCM will use the GJK algorithm. This will result in no artefacts for this approach. It trade off between performance and accuracy.

Convex Mesh cooking

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

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

If Only Vertex Points are Provided

When providing only vertices, set the PxConvexFlag::eCOMPUTE compute the mesh:

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

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

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

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

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

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

Convex cooking supports two different algorithms:

Quickhull Algorithm

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

The Quickhull algorithm performs these steps:

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

When the hull is constructed each new vertex added must PxCookingParams::planeTolerance from the hull, if not that vertex is dro

Inflation Based Incremental Algorithm

This algorithm always uses the PxConvexFlag::eINFLATE_CONVEX fl 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
- 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 polygon:
- 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 algorithm is significantly slower than the quickhull and produces signiresults. It is recommended to use the quickhull algorithm.

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

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

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

Vertex Points, Indices and Polygons are Provided

To create a PxConvexMesh given a set of input vertices (convexVe (hullPolygons):

```
PxConvexMeshDesc convexDesc;
convexDesc.points.count
                                    = 12;
                                    = sizeof(PxVec3);
convexDesc.points.stride
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 mes PxConvexmesh directly. This is the fastest way to create a convex r SDK requires at least 3 neighbor polygons for each vertex. Othe

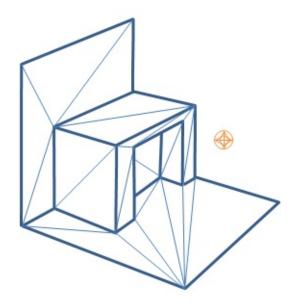
structure for PCM is not created and it does result in performance enabled.

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

Internal steps during convex cooking:

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

Triangle Meshes



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

```
meshDesc.points.data = verts;

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

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

PxDefaultMemoryInputData readBuffer(writeBuffer.getData(), writeBreturn physics.createTriangleMesh(readBuffer);
```

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

```
PxTolerancesScale scale;
PxCookingParams params(scale);
// disable mesh cleaning - perform mesh validation on development
params.meshPreprocessParams |= PxMeshPreprocessingFlag::eDISABLE_
// disable edge precompute, edges are set for each triangle, slow
params.meshPreprocessParams |= PxMeshPreprocessingFlag::eDISABLE_
// lower hierarchy for internal mesh
params.meshCookingHint = PxMeshCookingHint::eCOOKING_PERFORMANCE;
theCooking->setParams(params);
PxTriangleMeshDesc meshDesc;
meshDesc.points.count
                             = nbVerts;
meshDesc.points.stride
                            = sizeof(PxVec3);
meshDesc.points.data
                             = verts;
#ifdef _DEBUG
   // mesh should be validated before cooked without the mesh cl
   bool res = theCooking->validateTriangleMesh(meshDesc);
   PX_ASSERT(res);
#endif
```

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

Returned result enum *PxTriangleMeshCookingResult::eLARGE_TRI/* the user if the mesh contains large triangles, which should be tessellat simulation and CCT stability.

Like height fields, triangle meshes support per-triangle material inc triangle materials for a mesh, provide per-triangle indices to the coc mesh descriptor. Later, when creating the PxShape, supply a table 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 t subset different from the original input set. Mesh cleaning remove (containing out-of-range vertex references), duplicate triangles, and z When this happens, PhysX optionally outputs a mesh remapping tal internal triangle to its source triangle in the user's data.

There are multiple parameters to control mesh creation.

In PxTriangleMeshDesc:

• materialIndices defines per triangle materials. When a triangle another object, a material is required at the collision point. If mater then the material of the PxShape instance is used.

In PxCookingParams:

- scale defines Tolerance scale is used to check if cooked triangles. This check will help with simulation stability.
- suppressTriangleMeshRemapTable specifies whether the face rem
 If not, this saves a significant amount of memory, but the SDK
 provide information about which original mesh triangle is hit in co
 raycasts hits.
- buildTriangleAdjacencies specifies if the triangle adjacency infor The adjacent triangles can be retrieved for a given triangle using th
- meshPreprocessParams specifies mesh pre-processing parameter
 - PxMeshPreprocessingFlag::eWELD_VERTICES enables during triangle mesh cooking.
 - PxMeshPreprocessingFlag::eDISABLE_CLEAN_MESH disprocess. Vertices duplicities are not searched, huge triangled Vertices welding is not done. Does speed up the cooking.
 - PxMeshPreprocessingFlag::eDISABLE_ACTIVE_EDGES_ disables vertex edge precomputation. Makes cooking f contact generation.
- meshWeldTolerance If mesh welding is enabled, this controls the vertices are welded. If mesh welding is not enabled, this value defind distance for mesh validation. Provided no two vertices are within mesh is considered to be clean. If not, a warning will be emitted. He is required to achieve the best possible performance.

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

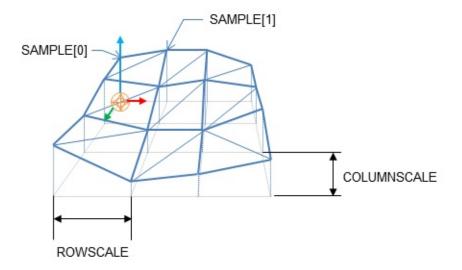
PxBVH33MidphaseDesc params:

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

PxBVH34MidphaseDesc params:

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

Height Fields



Local space axes for the height fields are:

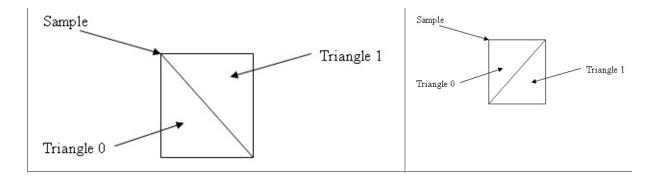
- Row X axis
- Column Z axis
- Height Y axis

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

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

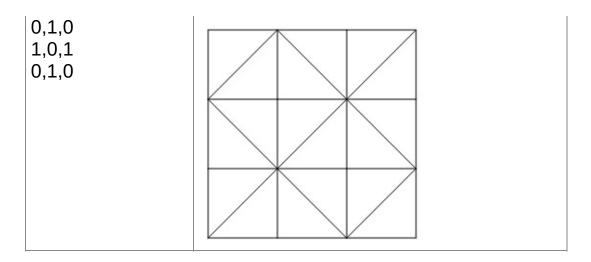
The flag and materials refer to the cell below and to the right of the indicate along which diagonal to split it into triangles, and the materials A special predefined material PxHeightFieldMaterial::eHOLE specifies a field. See the reference documentation for PxHeightFieldSample for mo

Tesselation flag set	Tesselation flag not set



Examples:

Tesselation flags	Result
0,0,0 0,0,0 0,0,0	
1,1,1 1,1,1 1,1,1	



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

Now create a PxHeightFieldGeometry and a shape:

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

The row and column scales tell the system how far apart the sample associated direction. The height scale scales the integer height values range.

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

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 createHeight does precompute and store the edge information. This allows much f heightfield, since the edges are already precomputed. It is very use 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 interchangeathis 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 ξ with shapes touching the surface of the heightfield, it generates contac are beneath the surface. The heightfield's "thickness" is used to contr the surface contacts are generated. This works by extruding the AABB the broad phase by the "thickness" along the vertical axis. Contacts are shape below the surface whose bounds intersects the heightfields extru

Unified heightfield contact generation is enabled by calling:

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

Legacy heightfield contact generation is enabled by calling:

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

These calls must be made before and scenes have been created, othe be issued. The heightfield collision setting is a global setting, and it appl

If PxCreatePhysics(...) is called, this will automatically call PxRegister register the default, unified heightfield collision approach. If PxCreate called, no heightfield contact generation is registered by default. If hei the application must call the appropriate heightfield registration function

Mesh Scaling

A shared PxTriangleMesh or PxConvexMesh may be stretched or corr instanced by a geometry. This allows multiple instancing of the same scale factors applied. Scaling is specified with the PxMeshScale clascale factors to be applied along 3 orthogonal axes. A factor greater stretching, while a factor less than 1.0 results in compression. The direct are governed by a quaternion, and specified in the local frame of the sh

Negative mesh scale is supported, with negative values producing a re corresponding axis. In addition PhysX will flip the normals for me 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:

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

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

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

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

Height fields can also be scaled, using scale factors stored in PxHeigh

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

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

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

PxGeometryHolder

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

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

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

However, it is often convenient to retrieve a geometry object from a knowing its type - for example, to call a function which takes a PxGeo an argument.

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

```
PxGeometryHolder geom = shape->getGeometry();

switch(geom.getType())
{
    case PxGeometryType::eSPHERE:
        shapeRenderActor = SAMPLE_NEW(RenderSphereActor)(renderer, ge break;
    case PxGeometryType::eCAPSULE:
        shapeRenderActor = SAMPLE_NEW(RenderCapsuleActor)(renderer, g geom.capsule().halfHeight);
        break;
...
}
```

The function *PxGeometryHolder::any()* returns a reference to a PxGe example, to compare two shapes in a scene for overlap:

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

Vertex and Face Data

Convex meshes, triangle meshes, and height fields can all be queried data. This is particularly useful, for example, when rendering the m shape. 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 of PxMeshQuery::getTriangle function. You can also retrieve adjacent triangle triangle (triangle triangleNeighbour[i] shares the edge vertex[i]-vertriangle indexed as 'triangleIndex', where vertex is in the range from 0 to feature the triangle mesh is cooked with buildTriangleAdjacencies parar

Convex Meshes

A convex mesh contains an array of vertices, an array of faces, and an concatenates the vertex indices for each face. To unpack a convex me 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++)</pre>
{
    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++)</pre>
        vertices[offset+j] = convexVerts[faceIndices[j]];
        normals[offset+j] = PxVec3(face.mPlane[0], face.mPlane[1]
    }
    for(PxU32 j=2;j<face.mNbVerts;j++)</pre>
        *triangles++ = PxU16(offset);
        *triangles++ = PxU16(offset+j);
        *triangles++ = PxU16(offset+j-1);
    }
    offset += face.mNbVerts;
}
```

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

Triangle Meshes

Triangle meshes contain arrays of vertices and index triplets which defindexing into the vertex buffer. The arrays can be accessed directly 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, specific was originally cooked. To determine the storage format at runtime, use

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

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

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

The corresponding vertex is:

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

Height Fields

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

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

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

The height field has three scaling parameters:

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

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

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

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

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

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

The samples are stored in row-major order; that is, row0 is stored first then row2, and so on. Thus the sample corresponding to the ith row ar i*nbCols + i.

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(is) * rs, PxReal(is)
```

Then tessellate the field from the samples as required.

Heightfield Modification

Heightfield samples can be modified at runtime in rectangular blocks. Ir 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);</pre>
```

```
PxPhysics* physics = getPhysics();
PxHeightField* pHeightField = cooking->createHeightField(heightFi
// create modified HF samples, this 10-sample strip will be used
// Source samples that are out of range of target heightfield wil
PxHeightFieldSample samplesM[10];
for (PxU32 i = 0; i < 10; i ++)
    samplesM[i].height = 1000;
    samplesM[i].materialIndex0 = 1;
    samplesM[i].materialIndex1 = 127;
}
PxHeightFieldDesc desc10Rows;
desc10Rows.nbColumns = 1;
desc10Rows.nbRows = 10;
desc10Rows.samples.data = samplesM;
desc10Rows.samples.stride = sizeof(PxHeightFieldSample);
pHeightField->modifySamples(1, 0, desc10Rows); // modify row 1 wi
```

PhysX does not keep a mapping from the heightfield to heightfield shait. Call PxShape::setGeometry on each shape which references the heightfield to reflect the new geometry:

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

The method PxHeightField::getTimestamp() returns the number of time been modified.

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

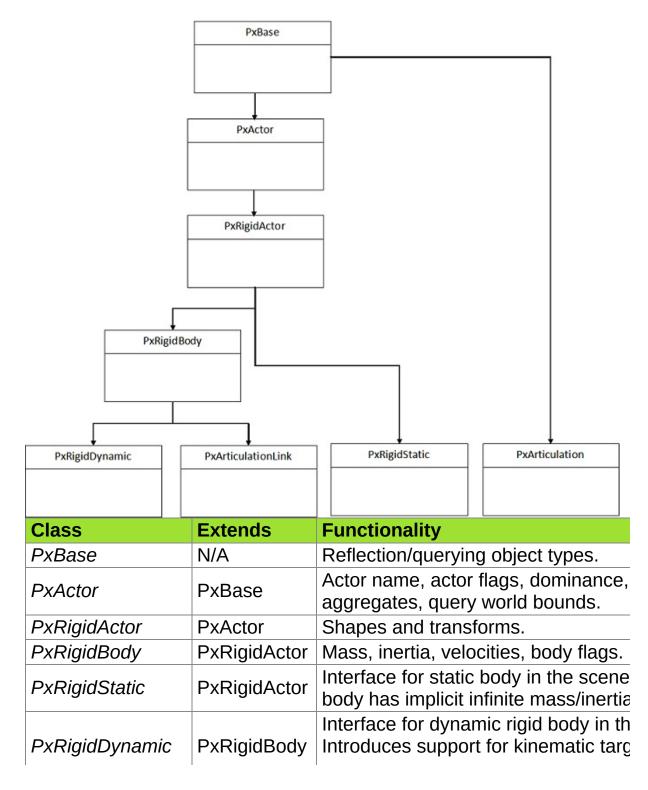
previous | next

Introduction

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

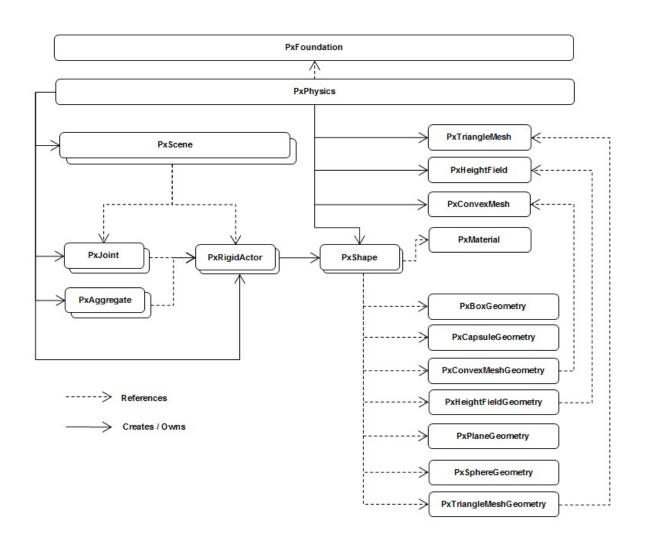
Rigid Body Object Model

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



		sleeping.
PxArticulationLink	PxRigidBody	Interface for a dynamic rigid body lin PxArticulation. Introduces support fo articulation and adjacent links.
PxArticulation	PxBase	Defines interface for a PxArticulation contained referencing multiple PxArt bodies.

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



The Simulation Loop

Now use the method PxScene::simulate() to advance the world forwasimplified code from the samples' fixed stepper class:

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

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

    mAccumulator -= mStepSize;

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

This is called from the sample framework whenever the app is dor events and is starting to idle. It accumulates elapsed real time until i sixtieth of a second, and then calls simulate(), which moves all ob forward by that interval. This is probably the simplest of very many diff with time when stepping the simulation forward.

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

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

True indicates that the simulation should block until it is finished, so results are guaranteed to be available. When fetchResults complet 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. advantage of this to perform rendering work in parallel with physics. returns, the results of the current simulation step are not available. So results of the current simulation step are not available.

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 le frames per second, with each frame corresponding to a physics ti smooth, realistic simulation of more complex physical scenes, use at le second.

Note: If you are making a real-time interactive simulation, you may be 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 the constant-sized time steps: The simulation code is sensitive to both very time steps, and also to too much variation between time steps. In these likely produce jittery simulation.

See *Simulation memory* for details of how memory is used in simulation

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Introduction

This section will introduce the fundamentals of rigid body collision.

Shapes

Shapes describe the spatial extent and collision properties of actors. three purposes within PhysX: intersection tests that determine the corrigid objects, scene query tests such as raycasts, and defining trigenerate 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 PxM both be specified upon creation. The following code creates a shapeometry and a specific material:

```
PxShape* shape = physics.createShape(PxSphereGeometry(1.0f), myMa
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 men simulation when you have many actors with identical geometry, but shavery strong restriction: you cannot update the attributes of a shared attached to an actor.

Optionally you may configure a shape by specifying shape flags of type default a shape is configured as

- a simulation shape (enabled for contact generation during simulatic
- 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 objec shape. There are some restrictions on which geometries may be spe depending on the shape flags and the type of the parent actors.

- TriangleMesh, HeightField and Plane geometries are not supposhapes that are attached to dynamic actors, unless the dynamic actors to be kinematic.
- TriangleMesh and HeightField geometries are not supported for tric

See the following sections for more details.

Detach the shape from the actor as follows:

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

Note: in previous versions of PhysX, release() was used to detach a sactor and destroy it. This use of release() is deprecated in PhysX 3.3 a supported in futures version of PhysX.

Simulation Shapes and Scene Query Shapes

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

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

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

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

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

To disable a PxShape instance from scene query tests:

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

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

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

Note: If the movement of the shape's actor does not need to be contr 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 hav (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:

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

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 Physiqueneric choice with great performance when many objects are sleeping degrade significantly though, when all objects are moving, or when objects are added to or removed from the broad-phase. This algorit world bounds to be defined in order to work.

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

The desired broad-phase algorithm is controlled by the *PxBroadPhase* the *PxSceneDesc* structure.

Regions of Interest

A region of interest is a world-space AABB around a volume of spac broad-phase. Objects contained inside those regions are properly han phase. Objects falling outside of those regions lose all collision deteregions should cover the whole game space, while limiting the amoun space.

Regions can overlap, although for maximum efficiency it is recomment amount of overlap between regions as much as possible. Note that AABBs just touch are not considered overlapping. Fo PxBroadPhaseExt::createRegionsFromWorldBounds helper function or non-overlapping region bounds by simply subdividing a given world A 2D grid.

Regions can be defined by the *PxBroadPhaseRegion* structure, alon assigned to them. They can be defined at scene creation time or at *PxScene::addBroadPhaseRegion* function. The SDK returns handle newly created regions, that can be used later to remove r *PxScene::removeBroadPhaseRegion* function.

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

Note that only *PxBroadPhaseType::eMBP* requires regions to *PxBroadPhaseType::eSAP* algorithm does not. This information is c *PxBroadPhaseCaps* structure, which lists information and capabilitic phase algorithm. This structure can be retrieved by the *PxScene::ge*

function.

Runtime information about current regions can be retrient PxScene::getNbBroadPhaseRegions and PxScene::getBroadPhaseRegions

The maximum number of regions is currently limited to 256.

Broad-phase Callback

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

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

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

Collision Filtering

In almost all applications beyond the trivial, the need arises to exer objects from interacting, or to configure the SDK collision detection beh way for an interacting pair. In the submarine sample, like indicated about notified when the submarine touched a mine, or the chain of a mine, so them blow up. The crab's Al also needs to know when crabs touch the h

Before we can understand what the sample does to achieve this, we r the possibilities of the SDK filtering system. Because filtering potentia happens in the deepest parts of the simulation engine, and needs to be of objects that come near each other, it is particularly performance sense way to implement it would be to always call a callback function t interacting pair, where the application, based on the two object pointer using some custom logic -- like consulting its game data base -- whet interact. Unfortunately this quickly becomes too slow if done for a very especially if the collision detection processing happens on a remote GPU or an other kind of vector processor with local memory, which wou its parallel computations, interrupt the main processor that runs game execute the callback before it can continue. Even if it were to be exewould likely be done so simultaneously on multiple cores or hyperthread code would have to be put in place to make sure that concurrent acces safe. Far better is to use some kind of fixed function logic that can exe processor. This is what we did in PhysX 2.x -- unfortunately the si filtering rules we provided were not flexible enough to cover all appli introduce both a shader system, which lets the developer implement an rules using code that runs on the vector processor (and is therefore not eventual game data base in main memory), which is more flexible that filtering, but just as efficient, and a totally flexible callback mechanis shader calls a CPU callback function that is able to access any appl cost of performance -- see PxSimulationFilterCallback for details. The application can decide on a per-pair basis to make this speed vs. flexibi

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

SampleSubmarine:

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

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

SampleSubmarineFilterShader() will be called for all pairs of shapes th other -- more precisely: for all pairs of shapes whose axis aligned boung space are found to intersect for the first time. All behavior beyond the what SampleSubmarineFilterShader() returns.

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

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

Each PxShape and PxParticleBase object in PhysX has a member PxFilterData. This is 128 bits of user defined data that can be used to 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 i application can give to the shader to operate on. You will want to use t about what to filter and what not.

Finally, SampleSubmarineFilterShader() also has a PxPairFlags para output, like the return value PxFilterFlags, though used slightly differentells the SDK if it should ignore the pair for good (eKILL), ignore 1 overlapping, but ask again, when filtering related data changes for (eSUPPRESS), or call the low performance but more flexible CPU cal cannot decide (eCALLBACK).

PxPairFlags specifies additional flags that stand for actions that the sim in the future for this pair. For example, eNOTIFY_TOUCH_FOUND me when the pair really starts to touch, not just potentially.

Let us look at what the above shader does:

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

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

detection between them.

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

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

return PxFilterFlag::eDEFAULT;
```

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

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

The sample identifies each shape's type by assigning its PxFilterE 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 encapsulated in SampleBase, it is done after the fact, using this function

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

```
setupFiltering(mSubmarineActor, FilterGroup::eSUBMARINE, FilterGr
FilterGroup::eMINE_LINK);
setupFiltering(link, FilterGroup::eMINE_LINK, FilterGroup::eSUBMA
setupFiltering(mineHead, FilterGroup::eMINE_HEAD, FilterGroup::eS
setupFiltering(heightField, FilterGroup::eHEIGHTFIELD, FilterGroup
setupFiltering(mCrabBody, FilterGroup::eCRAB, FilterGroup::eHEIGHTFIELD)
```

This scheme is probably too simplistic to use in a real game, but it show of the filter shader, and it will ensure that SampleSubmarine::onContainteresting pairs.

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

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 cluste in turn allows the SDK to optimize its spatial data operations. A typ ragdoll, made of multiple different actors. Without aggregates, this giv broad-phase entries as there are shapes in the ragdoll. It is typicall represent the ragdoll in the broad-phase as a single entity, and perfo tests in a second pass if necessary. Another potential use case is a large number of attached shapes.

Creating an Aggregate

Create an aggregate from the *PxPhysics* object:

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

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

PxAggregate* aggregate = physics->createAggregate(nbActors, selfC
```

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

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

Note that both the maximum number of actors and the self-coll immutable.

Populating an Aggregate

Adds an actor to an aggregate as follows:

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

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

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

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

Similarly, to remove the aggregate from the scene:

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

Releasing an Aggregate

To release an aggregate:

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

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

Amortizing Insertion

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

Trigger Shapes

Trigger shapes play no part in the simulation of the scene (though they to participate in scene queries). Instead, their role is to report that 1 overlap with another shape. Contacts are not generated for the inte result contact reports are not available for trigger shapes. Further, being part in the simulation, the SDK will not allow the the eSIN eTRIGGER_SHAPE flags to be raised simultaneously; that is, if one attempts to raise the other will be rejected, and an error will be passed to

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

```
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 SampleSubmimplementation of PxSimulationEventCallback::onTrigger in the Px class, a sub-class of PxSimulationEventCallback:

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

Interactions

The SDK internally creates an interaction object for each overlapping proad-phase. These objects are not only created for pairs of colliding righter pairs of overlapping triggers. Generally speaking users should objects are created regardless of the involved objects' types (rigid body and regardless of involved *PxFilterFlag* flags.

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

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

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 of simulation, PhysX will modify the velocity of an object in accordance applied forces and torques and as a result of various constraints, su joints.

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

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

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

```
void PxRigidBody::setLinearVelocity(const PxVec3& linVel, bool au
void PxRigidBody::setAngularVelocity(const PxVec3& angVel, bool a
```

Mass Properties

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

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

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

The second uses the mass of the bottom snowball only, resulting in a lon, the center of mass is moved to the bottom of the actor. This appropriate means physically correct, but the resulting snowman feels a bit more fill

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

The last snowman's mass properties are set up manually. The sample 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 the X-axis and high resistance around Y and Z. As a consequence, wobble back and forth only around the X axis.

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

When manually setting the mass/inertia tensor of bodies, PhysX requi for the mass and each principal axis of inertia. However, it is legal to partial values. When provided with a 0 mass or inertial value, PhysX interprincipal axis. This can be used to resist all linear motion or that resist all or some angular motion. Example that could be achieved using this approach are:

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

Some examples of what could be achieved are detailed below. First, le are creating a common structure - a windmill. The code to construct the be part of the windmill are provided below:

```
PxRigidDynamic* dyn = physics.createRigidDynamic(PxTransform(PxVe PxRigidActorExt::createExclusiveShape(*dyn, PxBoxGeometry(2.f, 0. PxRigidActorExt::createExclusiveShape(*dyn, PxBoxGeometry(0.2f, 2 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 PxRigidActorExt::createExclusiveShape(*t, PxBoxGeometry(0.5f, 1.5 scene.addActor(dyn); scene.addActor(st);
```

The above code creates a static box frame for the windmill and a croblades 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 reangular velocity indefinitely. However, if another object collided wit windmill would cease to function correctly because the turbine blade out of place. There are several options to make the turbine blade position when other bodies interact with it. One such approach might turbine have infinite mass and inertia. In this case, any interactions wit affect the turbine at all:

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

This example retains the previous behavior of the turbine spinning at velocity indefinitely. However, now the body's velocities cannot be constraints because the body has infinite mass and inertia. If a bod turbine blade, the collision would behave as if the turbine blade was a k

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

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

In both examples, the body's mass was set to 0, indicating that the bod so its linear velocity cannot be changed by any constraints. However, is body's inertia is configured to permit the body's angular velocity constraints around one principal axis or inertia. This provides a similar ϵ a revolute joint. The value of the inertia around the z-axis can be incre 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 mechanics, most interactions between bodies are typically solved Because of the law:

```
f = m*a (force = mass * acceleration)
```

Forces directly control a body's acceleration, but its velocity and posi For this reason control by force may be inconvenient if you need immed advantage of forces is that regardless of what forces you apply to the b the simulation will be able to keep all the defined constraints (joints and For example gravity works by applying a force to bodies.

Unfortunately applying large forces to articulated bodies at the reson system may lead to ever increasing velocities, and eventually to the fail maintain the joint constraints. This is not unlike a real world system, who ultimately break.

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

The PxForceMode member defaults to PxForceMode::eFORCE to approximately the possibilities. For example PxForceMode::eIMPUL impulsive force. PxForceMode::eVELOCITY_CHANGE will do the san the mass of the body, effectively leading to an instantaneous velocity of documentation of PxForceMode for the other possibilities.

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

There are further extension functions that compute the linear and angulathat would arise in the next simulation frame if an impulsive force or imput to be applied:

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

A use case for this function might be to predict an updated velocity fo that asset loading may be initiated in advance of the simulation frame to exceed a threshold velocity at the end of the frame. The impulsive fc simply the force and torque that are to be applied to the body multiplied the simulation frame. Neglecting the effect of constraint and contact for linear and angular velocity that are expected to arise in the next sire returned in deltaLinearVelocity and deltaAngularVelocity. The predicted then be computed with body.getLinearVelocity() + deltaLinearVelocity() + deltaLinearVelocity () + deltaLinearVelocity () + deltaLinearVelocity () + deltaLinearVelocity() + deltaLinearVelocity()

Gravity

Gravity is such a common force in simulations that PhysX makes it particle apply. For a scene-wide gravity effect, or any other uniform force field class' gravity vector using PxScene::setGravity().

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

Certain special effects can require that some dynamic actors are not into specify this set the flag:

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

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

An alternative to PxActorFlag::eDISABLE_GRAVITY is to use a zero gravity scene, then apply your own gravity force to rigid bodies, each used to create radial gravity fields, as demonstrated in SampleCustomC

Friction and Restitution

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

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

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

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

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

The coefficient of restitution of two colliding objects is a fractional valuation of speeds after and before an impact, taken along the line of imprestitution of 1 is said to collide elastically, while a coefficient of restitution inelastic.

Sleeping

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

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

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

```
void PxSimulationEventCallback::onWake(PxActor** actors, PxU32 co
void PxSimulationEventCallback::onSleep(PxActor** actors, PxU32 c
```

See the section *Callback Sequence* and the subsection *Sleep state* more information.

An actor goes to sleep when its kinetic energy is below a given thre time. Basically, every dynamic rigid actor has a wake counter which ge the simulation time step when the kinetic energy of the actor is b threshold. However, if the energy is above the threshold after a si counter gets reset to a minimum default value and the whole process the wake counter reaches zero, it does not get decremented any furth ready to go to sleep. Please note that a zero wake counter does not r has to be asleep, it only indicates that it is ready to go to sleep. Ther that might keep an actor awake for a while longer.

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

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

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

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

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 asle above hold, else it will be treated as awake.

In general, a dynamic rigid actor is guaranteed to be awake if at least (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 automatica

- PxRigidDynamic::setWakeCounter(), if the wake counter value is la
- PxRigidBody::setLinearVelocity(), ::setAngularVelocity(), if the velocity
- 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 also sets the wake counter to a positive value).

- PxRigidActor::setGlobalPose(), if the autowake parameter is set to
- Simulation gets disabled for a PxRigidActor PxActorFlag::eDISABLE_SIMULATION.
- PxScene::resetFiltering().
- PxShape::setSimulationFilterData(), if the subsequent re-filtering 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 def 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 step.
- The actor gets hit by a two-way interaction particle.

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

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

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

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

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 ch

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

```
void PxSimulationEventCallback::onWake(PxActor** actors, PxU32 co
void PxSimulationEventCallback::onSleep(PxActor** actors, PxU32 c
```

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

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

Sometimes it is desirable to detect transitions between awake and keeping track of the number of awake bodies. Suppose a sleeping body application, the counter is incremented, and during the next simula awake. Even though *B*'s sleep state did not change during simulatic since the previous *fetchResults()*, and so an *onWake()* event will be ge counter is incremented again in response to this event, its value will be

To use sleep state events to detect transitions, a record of the sleep interest has to be kept, for example in a hash. When processing an event 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 contromanipulate an actor's position or have it exactly follow a specific pascheme is provided by kinematic actors.

A kinematic actor is controlled using the PxRigidDynamic::setKinemat Each simulation step PhysX moves the actor to its target position, reg forces, gravity, collision, etc. Thus one must continually call setKinen time step, for each kinematic actor, to make them move along their movement of a kinematic actor affects dynamic actors with which it colli constrained with a joint. The actor will appear to have infinite mass ar dynamic actors out of the way.

To create a kinematic actor, simply create a regular dynamic actor the flag:

```
{\tt PxRigidBody::setRigidBodyFlag(PxRigidBodyFlag::eKINEMATIC, true);}
```

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

Caveats:

• It is important to understand the differ PxRigidDynamic::setKinematicTarget() and PxRigidActor::setG While setGlobalPose() would also move the actor to the desired properly interact with other objects. In particular, we the kinematic actor would not push away other dynamic actors in would go right through them. The setGlobalPose() function can still one simply wants to teleport a kinematic actor to a new position.

- A kinematic actor can push away dynamic objects, but nothing puresult, a kinematic can easily squish a dynamic actor against a state another kinematic actor. As a result, the squished dynamic 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
 PxSceneFlag::eENABLE_KINEMATIC_PAIRS
 ::eENABLE_KINEMATIC_STATIC_PAIRS gets set.

Active Transforms

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

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

When a scene's fetchResults() method is called an array of *PxActive1* generated, each entry in the array contains a pointer to the actor that m and its new transform. Because only actors that have moved will be inc approach is potentially much more efficient than, for example, analyzin scene individually.

The example below shows how to use active transforms to update a rer

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

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

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

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

Note: Since the target transform for kinematic rigid bodies is set by th 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 trans PhysX scene to an associated external object such as a render mesh.

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

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

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

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

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

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

Note: Since the target transform for kinematic rigid bodies is set by th 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 effectively imbues the dominant body in a pair with infinite mass. This mass modification within the constraint solver and, as such, can overrist of the bodies in a pair. Similar effects can be achieved through local m contact modification but dominance has the advantage of being har within the SDK so does not incur the additional memory and perforn contact modification.

Each actor must be assigned a dominance group ID. This is a 5-bit va 31]. As such, you are restricted to at-most 32 dominance groups. By de placed in dominance group 0. An actor can be assigned to a dominan following method on PxActor:

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

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

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

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

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

The user can define 3 different states for a given PxDominanceGroul indicates that both bodies have equal dominance. This is the default be

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 PxDominanceGroupF both sides of the PxDominanceGroupPair is also invalid. These values to be scales applied to the bodies' respective inverse mass and dominance value of 0 would therefore equate to an infinite mass body.

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

```
PxRigidDynamic* actorA = mPhysics->createRigidDynamic(PxTransform
PxRigidDynamic* actorB = mPhysics->createRigidDynamic(PxTransform
actorA->setDominanceGroup(1);
actorB->setDominanceGroup(2);
mScene->setDominanceGroupPair(1, 2, PxDominanceGroupPair(0.f, 1.f
```

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

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

As previously mentioned, dominance does not permit values other th dominance values are applied uniformly to both the inverse mass ε Joints and contacts through contact modification permit defining separand inverse inertia scales, which accept any values within the range [6] so can be used to achieve a wider range of effects than dominance can

Dominance can produce some very peculiar results if misused. For exa 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 car pushes body B into body C.

Solver Iterations

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

```
void PxRigidDynamic::setSolverIterationCounts(PxU32 minPositionIt
```

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

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

Immediate Mode

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

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

The API provides a function to perform contact generation:

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

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

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

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

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

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

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

4

Batching constraints reorders the provided constraints and produces be can be used by the solver to accelerate constraint solving by independent constraints and solving them in parallel using multip registers. This process is entirely optional and can bypassed if not des will change the order in which constraints are processed, which can cl of the solver.

The following methods is provided to create contact constraints:

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

This method can be provided with the contacts produced by PxGene contacts produced by application-specific contact generation approache

The following methods are provided to create joint constraints:

```
PX_C_EXPORT PX_PHYSX_CORE_API bool PxCreateJointConstraints(PxCon PX_C_EXPORT PX_PHYSX_CORE_API bool PxCreateJointConstraintsWithSh
```

The methods provide a mechanism for the application to define joi application to make use of PhysX PxConstraint objects, which create th

The following method solves the constraints:

```
PX_C_EXPORT PX_PHYSX_CORE_API void PxSolveConstraints(PxConstrain PxVec3* linearMotionVelocity, PxVec3* angularMotionVeloci
```

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

The following method is provided to integrate the bodies' final pose

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

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

Enhanced Determinism

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

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

To overcome this issue, PhysX provides PxSceneFlag::eENABLE_ENHANCED_DETERMINISM, which provide of determinism. Specifically, provided the application inserts the actors order, with this flag raised, the simulation of an island will be identica other islands in the scene. However, this mode sacrifices some perform additional determinism.

Axis locking

It is possible to restrict motion along or around specific world-space at PxRigidDynamicLockFlag. For example, the below code snippet der restrict a PxRigidDynamic to two dimensional simulation. In this ca PxRigidDynamic to rotate only around the Z-axis and to translate only axes:

```
PxRigidDynamic* dyn = physics.createRigidDynamic(PxTransform(PxVe
...
//Lock the motion
dyn->setRigidDynamicLockFlags(PxRigidDynamicLockFlag::eLOCK_LINEA
```

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

NVIDIA PhysX SDK 3.4.1 Documentation » previous | next User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Callback Sequence

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

Inside fetchResults(), among other things, the buffers are swapped. Moreover that properties of each object's internal simulation state are visible state. Some event callbacks happen before this swap, and som that happen before are:

- onTrigger
- onContact
- onConstraintBreak

When these events are received in the callback, the shapes, actors, et state they were in immediately before the simulation started. This is p these events were detected early on during the simulation, before obje (moved) forward. For example, a pair of shapes that get an onContact(are in contact will still be in contact when the call is made, even thou 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 PxSimulationEventCallback so that the various virtual functions may I desired. An instance of this subclass can then be registered per PxScene::setSimulationEventCallback or PxSceneDesc::simula Following these steps alone will ensure that constraint break event reported. One further step is required to report sleep and wake ev expense of reporting all sleep and wake events, actors identified as wo notification require the flag PxActorFlag::eSEND SLEEP NOTIFIES to to receive onContact and onTrigger events it is necessary to set a flag callback for all pairs of interacting objects for which events are require the filter shader callback can be found in Section *Collision Filtering*.

Simulation memory

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

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

After the self-explanatory function argument describing the size of the three function arguments are an identifier name, which identifies the typ the __FILE__ and __LINE__ location inside the SDK code where the all More details of these function arguments can be found in the PhysXAPI

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

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

Minimizing dynamic allocation is an important aspect of performar provides several mechanisms to control and analyze memory usag discussed in turn.

Scene Limits

The number of allocations for tracking objects can be minimized capacities of scene data structures, using either PxSceneDesc::limits scene or the function PxScene::setLimits(). It is useful to note that I represent hard limits, meaning that PhysX will automatically perform for 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 b size. The initial number of blocks allocated to the pool can be co PxSceneDesc::nbContactDataBlocks, while the maximum number of b be in the pool is governed by PxSceneDesc::maxNbContactData internally needs more blocks than nbContactDataBlocks then it will aur further blocks to the pool until the number of blocks reaches maxNbCo PhysX subsequently needs more blocks than the maximum number o simply start dropping contacts and joint constraints. When this happassed to the error stream in the PX_CHECKED configuration.

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

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

Scratch Buffer

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

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

In-place Serialization

PhysX objects cab be stored in memory owned by the application u deserialization mechanism. See *Serialization* for details.

PVD Integration

Detailed information about memory allocation can be recorded and disp Visual Debugger. This memory profiling feature can be configure trackOutstandingAllocations flag when calling PxCreatePhysics(), ar PxVisualDebuggerConnectionFlag::eMEMORY when connecting to 1 PxVisualDebuggerExt::createConnection().

Completion Tasks

A completion task is a task that executes immediately after PxScene::s If PhysX has been configured to use worker threads then PxScene simulation tasks on the worker threads and will likely exit before the w completed the work necessary to complete the scene update. As a concompletion task would first need to call PxScene::fetchResults(tru fetchResults blocks until all worker threads started during simulate() has work. After calling fetchResults(true), the completion task can perfor 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 PxSce details can be found in the PhysAPI documentation.

Synchronizing with Other Threads

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

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

Substepping

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

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

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

```
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-the modifications during the collide() phase will be seen immediately the advance() phase. This allows collide() to begin before the data require available and to run in parallel with game logic that generates inputs to particularly useful for animation logic generating kinematic targets, applying forces to bodies. The write-through properties are listed below:

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

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

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

Any other sequence of API calls is illegal. The SDK will issue error me can interleave the physics-dependent game logic between collide() and

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

fetchCollision() will wait until collide() has finished before it updates properties in the SDK. Once fetchCollision() has completed, any performed on the objects in the executing scene will be buffered and v until the simulation and a call to fetchResults() has completed. The swrite-through properties into account when computing the new sets poses for the actors being simulated.

Split fetchResults

The fetchResults() method is available in both a standard and split form offers some advantages over the standard fetchResult() method becauser to parallelize processing of contact reports, which can be expensionable scenes.

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

```
gSharedIndex = 0;
gScene->simulate(1.0f / 60.0f);

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

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

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

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

This feature is demonstrated in SnippetSplitFetchResults. In order to

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

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Tuning Shape Collision Behavior

Shapes used for contact generation influence the motion of the 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 the colliding stacking: contactOffset or and restOffset. Thev PxShape::setContactOffset() and PxShape::setRestOffset() respective 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 whe penetrating (unless collision detection between the two shapes is in soldisabled, 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 centimeter 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, of further motion in the direction of penetration. This would make the sime the simulation time step is selected to be tiny, which is not desir performance. Instead, we want the force at the contact to smo penetration increases until it reaches a value sufficiently high to penetrating motion. The distance at which this maximum force restDistance, because at this distance two shapes stacked on each oth equilibrium and come to rest. When the shapes are for some reason penetration increases are for some reason penetration.

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

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

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

Contact Modification

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

The callback occurs for all pairs of colliding shapes for which the use pair flag PxPairFlag::eMODIFY CONTACTS in the filter shader.

To listen to these modify callbacks, derive from the class PxContactMoc

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

And then implement the function on Contact Modify of PxContact Modify C

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

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

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

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

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

Uniformly scaling a body's inverse mass and inverse inertia by the salthe body behaving like a body that is either heavier or lighter dependenced. Providing inverse mass/inverse inertia scales < 1 results in the heavier; providing scales > 1 result in the body appearing lighter. Fo mass/inertia scales of 0.5 result in the body appearing to have double to inverse mass/inertia scales of 4 would result in the body appearing to horiginal mass. Providing inverse mass/inertia scale of 0 results in the body appearing to has infinite mass.

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

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

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

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

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

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

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

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

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

There are a couple of special requirements for the callback due to the farm deep inside the SDK. In particular, the callback should be thread In other words, the SDK may call onContactModify() from any thread a concurrently (i.e., asked to process sets of contact modification pairs sir

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

Contact reporting

Here is an example for a contact event function from SampleSubmarine

```
void SampleSubmarine::onContact(const PxContactPairHeader& pairHe
    const PxContactPair* pairs, PxU32 nbPairs)
{
    for(PxU32 i=0; i < nbPairs; i++)</pre>
        const PxContactPair& cp = pairs[i];
        if(cp.events & PxPairFlag::eNOTIFY_TOUCH_FOUND)
        {
            if((pairHeader.actors[0] == mSubmarineActor) ||
                (pairHeader.actors[1] == mSubmarineActor))
            {
                PxActor* otherActor = (mSubmarineActor == pairHea
                    pairHeader.actors[1] : pairHeader.actors[0];
                Seamine* mine = reinterpret_cast<Seamine*>(other.
                // insert only once
                if(std::find(mMinesToExplode.begin(), mMinesToExp
                    mMinesToExplode.end())
                    mMinesToExplode.push_back(mine);
                break;
            }
        }
    }
}
```

SampleSubmarine is a subclass of PxSimulationEventCallback. onCopair for which the requested contact events have been triggered. The only interested in eNOTIFY_TOUCH_FOUND events, which are rais shapes start to touch. In fact it is only interested in touch events of the is checked in the second if-statement. It then goes on to assume that the mine (which works in this example because the sample is configured contact reports will get sent when a submarine actor is involved). Aft mine to a set of mines that should explode during the next update.

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

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

- Use PxPairFlag::eNOTIFY_THRESHOLD_F
 ::eNOTIFY_THRESHOLD_FORCE_PERSISTS,
 ::eNOTIFY_THRESHOLD_FORCE_LOST instead
 ::eNOTIFY_TOUCH_FOUND etc.
- Specify the threshold force for a dynamic rigid PxRigidDynamic::setContactReportThreshold(). If the body collic object and the contact force is above the threshold, a report enabled according to the PxPairFlag setting of the pair). If two obdies both have a force threshold specified then the lower used.

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

Contact Reports and CCD

If continuous collision detection (CCD) with multiple passes is enabled, 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 toc *eNOTIFY_TOUCH_CCD* has to be raised for the collision pair. *eNOTIFY_TOUCH_CCD* events for the non primary impacts. For per the system can not always tell whether the contact pair lost touch in a CCD passes and thus can also not always tell whether the contact persisted. *eNOTIFY_TOUCH_CCD* just reports when the two collidetected 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 format for a given PxContactPair depending on certa depending on the shapes involved, the properties of the contacts, mat the contacts are modifiable.

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

```
void MySimulationCallback::onContact(const PxContactPairHeader& p
    const PxContactPair* pairs, PxU32 nbPairs)
{
    const PxU32 bufferSize = 64;
    PxContactPairPoint contacts[bufferSize];
    for(PxU32 i=0; i < nbPairs; i++)</pre>
        const PxContactPair& cp = pairs[i];
        PxU32 nbContacts = pairs[i].extractContacts(contacts, buf
        for(PxU32 j=0; j < nbContacts; j++)</pre>
            PxVec3 point = contacts[j].position;
            PxVec3 impulse = contacts[j].impulse;
            PxU32 internalFaceIndex0 = contacts[j].internalFaceIn
            PxU32 internalFaceIndex1 = contacts[j].internalFaceIn
            //...
        }
    }
}
```

This approach requires copying data to a temporary buffer in order second approach allows the user to iterate over the contact information their own copy:

```
void MySimulationCallback::onContact(const PxContactPairHeader& p
    const PxContactPair* pairs, PxU32 nbPairs)
{
    for(PxU32 i=0; i < nbPairs; i++)</pre>
        const PxContactPair& cp = pairs[i];
                    PxContactStreamIterator iter(cp.contactPatche
        const PxReal* impulses = cp.contactImpulses;
        PxU32 flippedContacts = (cp.flags & PxContactPairFlag::eI
        PxU32 hasImpulses = (cp.flags & PxContactPairFlag::eINTER
        PxU32 nbContacts = 0;
        while(iter.hasNextPatch())
        {
            iter.nextPatch();
            while(iter.hasNextContact())
            {
                iter.nextContact();
                PxVec3 point = iter.getContactPoint();
                PxVec3 impulse = hasImpulses ? dst.normal * impul
                PxU32 internalFaceIndex0 = flippedContacts ?
                    iter.getFaceIndex1() : iter.getFaceIndex0();
                PxU32 internalFaceIndex1 = flippedContacts ?
                    iter.getFaceIndex0() : iter.getFaceIndex1();
                //...
                nbContacts++;
            }
        }
    }
}
```

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

Extra Contact Data

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

Example code:

Continuous Collision Detection

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

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

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

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

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

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

```
PxRigidBody* body;
```

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

Once enabled, CCD only activates between shapes whose relative spesum of their respective CCD velocity thresholds. These veloci automatically calculated based on the shape's properties and support n

Contact Notification and Modification

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

CCD supports contact modification. To listen to these modify callback class PxCCDContactModifyCallback:

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

And then implement the function on Contact Modify of PxContact Modify C

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

This onContactModify callback operates using the same semantic collision detection contact modification callbacks. For further details, documentation on Callbacks.

As with discrete collision detection, CCD will only emit contact modifigiven pair if the user has specified the pair flag PxPairFlag::eMODIFY_ filter shader.

Triggers

Currently, shapes flagged with PxShapeFlag::eTRIGGER_SHAPE will CCD. However, it is possible to get trigger events from CCD by not flaggas PxShapeFlag::eTRIGGER_SHAPE and instead configuring the filte the following state for pairs involving trigger shapes:

```
pairFlags = PxPairFlag::eTRIGGER_DEFAULT |
PxPairFlag::eDETECT CCD CONTACT; return PxFilterFlag::eDEFA
```

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

Tuning CCD

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

- 1. PxSceneDesc.ccdMaxPasses: This variable controls the number of perform. This is defaulted to 1, meaning that all objects are attem to the TOI of their first contact. Any remaining time after the TOI of will be dropped. Increasing this value permits the CCD to run mureduces the likelihood of time being dropped but can increase the c
- 2. PxRigidBody::setMinCCDAdvanceCoefficient(PxReal advanceC method allows you to adjust the amount by which the CCD adv given pass. By default, this value is 0.15, meaning that CCD will ad the 0.15 * ccdThreshold, where ccdThreshold is a value compu acts as a lower-bound of the maximum amount of time that could before there is a chance that the object could have tunnelled. Tl

- 0.15 improves the fluidity of motion without risking missed collisi value can negatively impact fluidity but will reduce the likelihood of the end of a frame. Increasing this value may increase the likelihood only be set in the range [0,1].
- 3. Enabling the flag PxSceneFlag::eDISABLE_CCD_RESV PxSceneDesc.flags: Enabling this flag disables CCD resweeps. missed collisions as the result of ricochets but has the poter overhead of the CCD. In general, enabling this advancement mc that objects will not pass through the static environment but no long dynamic objects with CCD enabled will not pass through each-othe
- 4. PxRigidBody::setRigidBodyFlag(PxRigidBodyFlag::eENABLE_CCI true): Enabling this flag enables the application of friction forces i disabled by default. As the CCD operates using only linear motic inside CCD can cause visual artefacts.

Performance Implications

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

Limitations

The CCD system is a best-effort conservative advancement schern number of CCD substeps (defaulted to 1) and drops any remaining tir only dropped on high-speed objects at the moment of impact so it However, this artefact can become noticeable if you simulate an object

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

It is also recommended that you filter away CCD interactions between constrained together, e.g. limbs in the same ragdoll. Allowing CCD in limbs of the same ragdoll could increase the cost of CCD and also pot to be dropped unnecessarily. CCD interactions are automatically disable an articulation.

Raycast CCD

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

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

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

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

Speculative CCD

In addition to sweep-based CCD, PhysX also provides a cheaper but le called speculative CCD. This approach functions differently to the sw that it operates entirely as part of the discrete simulation by inflating co on object motion and depending on the constraint solver to ensure t 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 accelerates an actor (as a result of a collision or joint) such that the acthrough objects during that time-step, speculative CCD can result in tun

To enable this feature, raise PxRigidBodyFlag::eENABLE_SPECULA rigid body that requires CCD:

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

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

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

This feature can work in conjunction with the sweep-based CCD, e. kinematic has speculative CCD enabled but dynamic rigid bodies use § However, if speculative CCD is used on kinematics in conjunction with § it is important to ensure that interactions between the kinematic actor contacts and the CCD-enabled dynamic actors do not also enable § interactions otherwise the sweep-based CCD may overrule the specul 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 (Separal and distance-based collision detection to generate full contact manifol the potential contacts in one frame, so it lends itself better to state approach is stable for small contact offsets and rest offsets but may correct contact points when large offsets are used because it approx points in these situations by plane shifting.

2. Persistent Contact Manifold (PCM)

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

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

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

User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Joint Basics

A joint constrains the way two actors move relative to one another. A ty would be to model a door hinge or the shoulder of a character. Joints the PhysX extensions library and cover many common scenarios, be cases that are not met by the joints packaged with PhysX, you can im Since joints are implemented as extensions, the pattern for creatir different from other PhysX objects.

Creation of simple joints and limits is demonstrated in the SnippetJoint 9

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

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

One of the actors must be movable, either a *PxRigidDynamic* or a *PxA* other may be of one of those types, or a *PxRigidStatic*. Use a NU indicate an implicit actor representing the immovable global reference fr

Each localFrame argument specifies a constraint frame relative to the at Each joint defines a relationship between the global positions and original frames that will be enforced by the PhysX constraint solver. In this example joint constraints the origin points of the two frames to be coincident at coincide, but allows the two actors to rotate freely relative to one at 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 t

the orientations to vary freely.

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

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

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

```
scene->setVisualizationParameter(PxVisualizationParameter::eJOINT
scene->setVisualizationParameter(PxVisualizationParameter::eJOINT
...
joint->setConstraintFlag(PxConstraintFlag::eVISUALIZATION)
```

Force Reporting

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

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

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

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

Breakage

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

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

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

A constraint flag records whether a joint is currently broken:

bool broken = (joint->getConstraintFlags() & PxConstraintFlag::eBROKI

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

Projection

Under stressful conditions, PhysX' dynamics solver may not be able to the constraints specified by the joint. PhysX provides kinematic *project* bring violated constraints back into alignment even when the solver fair a physical process and does not preserve momentum or respect collist 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 values beyond which a joint will be projected, and set the constraint projected.

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

Very small tolerance values for projection may result in jittering around t

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

- world attachment or a rigid static actor with a projecting constraint
- kinematic actor with a projecting constraint

- all dominant dynamic actor (has projecting constraints and all of projecting towards this dynamic)
- dominant dynamic actor (same as above but there is at least one constraint as well)
- partially dominant dynamic actor (has at least one one-way pr towards this dynamic and at least one one-way projecting constrain actor)
- world attachment or a rigid static actor without any projecting const
- kinematic actor without any projecting constraints
- dynamic actor with or without two-way projecting constraints (among these, the one with the highest constraint count wins)

Limits

Some PhysX joints constrain not just relative rotation or translation, bu *limits* on the range of that motion. For example, in its initial configuratic allows free rotation around its axis, but by specifying and enabling a limbounds may be placed upon the angle of rotation.

Limits are a form of collision, and like collision of rigid body shapes, s requires a *contactDistance* tolerance specifying how far from t configuration may be before the solver tries to enforce it. Note that e starts before the limit is violated, so the role played by contactDis analogous to the role a positive contactDistance value plays in collision contact makes the limit less likely to be violated even at high relative because the limit is active more of the time, the joint is more expensive

Limit configuration is specific to each type of joint. To set a limit, geometry and set the joint-specific flag indicating that the limit is enable

Limits may be either *hard* or *soft*. When a hard limit is reached, relative stop dead if the limit is configured with zero restitution, or bounce if the zero. When a soft limit is violated, the solver will pull the joint back tow a spring specified by the limit's spring and damping parameters. By def and without restitution, so when the joint reaches a limit motion will sim softness for a limit, declare the limit structure and set the spring and dadirectly:

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

Note: Limits are not projected.

When using spring limits, the eACCELERATION flag is strongly recon will automatically scale the strength of the spring according to the mas objects that the limit is acting upon, and can substantially reduce the required for good, stable behavior.

Actuation

Some PhysX joints may be actuated by a motor or a spring implicitly PhysX solver. While driving simulations with actuated joints is more expapplying forces, it can provide much more stable control of simulating Prismatic Joint, and Revolute Joint for details

Note: The force generated by actuation is not included in the force re solver, nor does it contribute towards exceeding the joint's breakage fo

Note: Changing the drive parameters for a joint, or activating or deact does not wake sleeping bodies attached to the joint. If required, wake t manually.

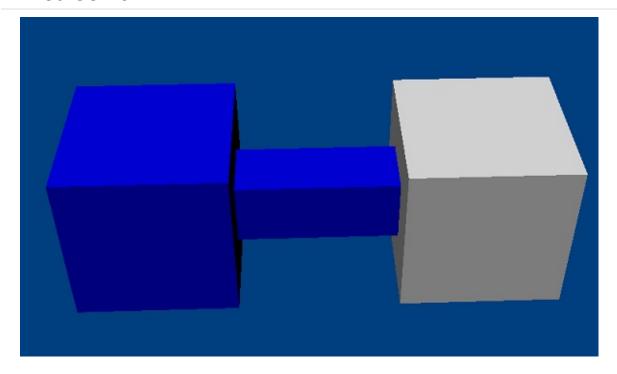
When using spring drives (in particular, drives on the D6 joint), the eAC is strongly recommended. This flag will automatically scale the stre according to the masses and inertias of objects that the limit is act substantially reduce the amount of tuning required for good, stable beha

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 hav ragdoll of masses 1 and 10, PhysX will typically resolve the joint by ch of the lighter body much more than the heavier one. You can apply a r the first body to make PhysX change the velocity of both bodies by ar ensure the same property holds for both linear and angular velocity, yo inertia scales in accordance with the bodies' inertias as well. Applying that the joint sees similar effective masses and inertias makes the solv which can make individual joints seem less rubbery or separated, a bodies appear less twitchy

Many applications that prioritize visual behavior over adherence to benefit from tuning these scale values. But if you use this feature, bear and inertia scaling is fundamentally nonphysical. In general momeonserved, the energy of the system may increase, the force reported incorrect, and non-physical tuning of breakage thresholds and for required.

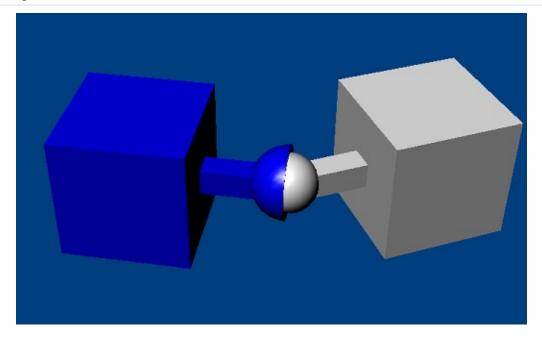
Fixed Joint



The fixed joint constrains two objects so that the positions and o constraint frames are the same.

Note: All joints are enforced by the dynamics solver, so although under the objects will maintain their spatial relationship, there may be some of alternative, which is cheaper to simulate and does not suffer from drift, single actor with multiple shapes. However fixed joints are useful, for e joint must be breakable or report its constraint force.

Spherical Joint



A spherical joint constrains the origins of the actor's constraint frames to

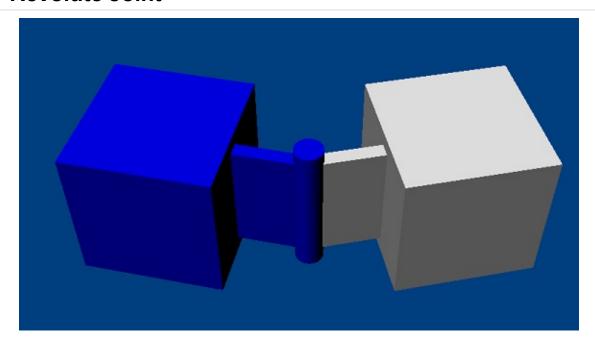
The spherical joint supports a cone limit, which constrains the angle b of the two constraint frames. Actor1's X-axis is constrained by a limit of the x-axis of actor0's constraint frame. The allowed limit values are the around the y- and z- axes of that frame. Different values for the y- are specified,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

```
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 jit

Note: Visualization of the limit surface can help considerably in under shape.

Revolute Joint



A revolute joint removes all but a single rotational degree of freedom fro axis along which the two bodies may rotate is specified by the commo frames and their common x-axis. In theory, all origin points along the 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 ϵ the y- and z- axes of the joint frames are coincident, and increases mov towards the z-axis:

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

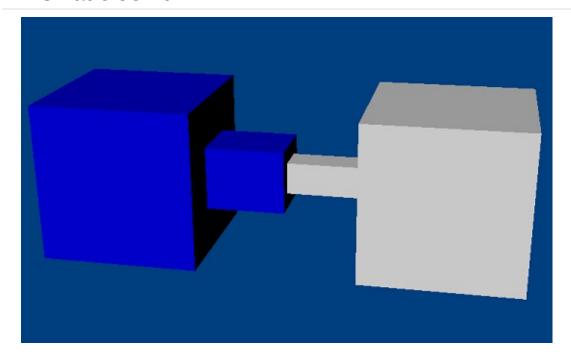
The joint also supports a motor which drives the relative angular velocitowards a user-specified target velocity. The magnitude of the force at may be limited to a specified maximum:

```
joint->setDriveVelocity(10.0f);
joint->setRevoluteJointFlag(PxRevoluteJointFlag::eDRIVE_ENABLED,
```

By default, when the angular velocity at the joint exceeds the target velocity as a brake; a freespin flag disables this braking behavior.

The drive force limit for a revolute joint may be interpreted either as a fedepending on the value of PxConstraintFlag::eDRIVE_LIMITS_ARE_FC

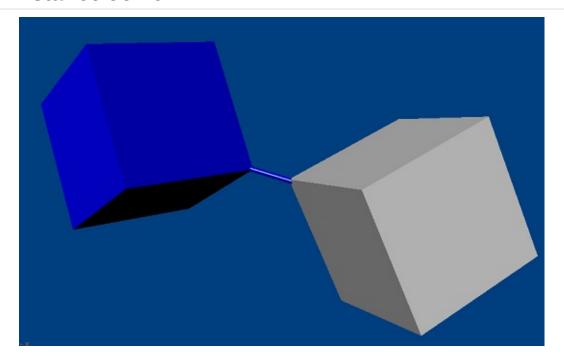
Prismatic Joint



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. supports a single limit with upper and lower bounds on the distance constraint frames' origin points:

```
joint->setLimit(PxJointLimitPair(-10.0f, 20.0f, 0.01f);
joint->setPrismaticJointFlag(PxPrismaticJointFlag::eLIMIT_ENABLED
```

Distance Joint



The distance joint keeps the origins of the constraint frames within distance. The range may have both upper and lower bounds, v separately by flags:

```
joint->setMaxDistance(10.0f);
joint->setDistanceJointFlag(eMAX_DISTANCE_ENABLED, true);
```

In addition, when the joint reaches the limits of its range motion beyond either be entirely prevented by the solver, or pushed back towards implicit spring, for which spring and damping parameters may be specifi

D6 Joint

The D6 joint is by far the most complex of the standard PhysX joints. I behaves like a fixed joint - that is, it rigidly fixes the constraint frame However, individual degrees of freedom may be unlocked to permit a rotation around the x-, y- and z- axes, and translation along these axes.

Locking and Unlocking Axes

To unlock and lock degrees of freedom, use the joint's setMotion function

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

Unlocking translational degrees of freedom allows the origin point of frame to move along a subset of the axes defined by actor0's corexample, unlocking just the X-axis creates the equivalent of a prismatic

Rotational degrees of freedom are partitioned as *twist* (around the constraint frame) and *swing* (around the Y- and Z- axes). Different effect unlocking various combinations of twist and swing.

- if just a single degree of angular freedom is unlocked, the result is to a revolute joint. It is recommended that if just one angular free should be the twist degree, because the joint has various configurations that are designed for this case.
- if both swing degrees of freedom are unlocked but the twist degree the result is a *zero-twist* joint. The x-axis of actor1 swings freely as of actor0 but twists to minimize the rotation required to align the creates a kind of isotropic universal joint which avoids the problem the problem of the proble

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 ac to remain orthogonal to the z-axis of actor0. In character application used to model an elbow swing joint incorporating the twist freedon 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 acts as the steering axis. Care must be taken with this combinants anisotropic behavior and singularities (beware the dreaded gimba π/2 radians (90 degrees), making the zero-twist joint a better behamost 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:

```
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:

• the point-on-line joint (with axis along the x-axis of actor0's constration:

Note: Angular projection is implemented only for the cases when two degrees of freedom are locked.

Limits

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

```
d6joint->setMotion(PxD6Axis::eX, PxD6Motion::eFREE);
d6joint->setMotion(PxD6Axis::eY, PxD6Motion::eLIMITED);
d6joint->setMotion(PxD6Axis::eZ, PxD6Motion::eLIMITED);
d6joint->setLinearLimit(PxJointLinearLimit(1.0f, 0.1f));
```

constrains the y- and z- coordinates of actor1's constraint frame to lie \(\) Since the x-axis is unconstrained, the effect is to constrain the origin of frame to lie within a cylinder of radius 1 extending along the x-axis of frame.

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

If both swing degrees of freedom are limited, a limit cone is generated, of the spherical joint. As with the spherical joint, very small or highly ϵ 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 maximu is π radians (180 degrees). If the other swing degree is free, the maximit is $\pi/2$ radians (90 degrees).

Drives

The D6 has a linear drive model, and two possible angular drive moc *proportional derivative* drive, which applies a force as follows:

```
force = spring * (targetPosition - position) + damping * (targetVelocity - \cdot\)
```

The drive model may also be configured to generate a proportional acc a force, factoring in the masses of the actors to which the joint is attadrive 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 the impulse, depending on PxConstraintFlag::eDRIVE_LIMITS_ARI
- acceleration drive flag

The drive attempts to follow the desired position input with the confiq damping properties. A physical lag due to the inertia of the driven body drive spring will occur; therefore, sudden step changes will result ove steps. Physical lag can be reduced by stiffening the spring or supplying

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

```
// set all translational degrees free
```

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

// set all translation degrees driven:

PxD6Drive drive(10.0f, -20.0f, PX_MAX_F32, true);
d6joint->setDrive(PxD6JointDrive::eX, drive);
d6joint->setDrive(PxD6JointDrive::eY, drive);
d6joint->setDrive(PxD6JointDrive::eZ, drive);

// Drive the joint to the local(actor[0]) origin - since no angul
// 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 intuitive representation free from singularities. For this reason, the D6 angular drive models - twist and swing and SLERP (Spherical Linear Int

The two models differ in the way they estimate the path in quaternion current orientation and the target orientation. In a SLERP drive, the directly. In a twist and swing drive, it is decomposed into separat components and each component is interpolated separately. Twist and many situations; however, there is a singularity when driven to 180 addition, the drive will not follow the shortest arc between two oriental hand, SLERP drive will follow the shortest arc between a pair of angular transport of the point of the point of the point of the path in quaternion and the target orientation. In a SLERP drive, the directly in a support of the path in quaternion and the target orientation. In a SLERP drive, the directly in a support of the path in quaternion and the target orientation. In a SLERP drive, the directly in a support of the path in quaternion and the target orientation. In a SLERP drive, the directly in a support of the path in quaternion and the target orientation. In a SLERP drive, the directly in a support of the path in quaternion and the target orientation. In a SLERP drive, the directly in a support of the path in quaternion and the path in quaternion a

The angular drive model has the following parameters:

- An angular velocity target specified relative to actor0's constraint
- An orientation target specified relative to actor0's constraint frame
- drive specifications for SLERP (slerpDrive), swing (swingE (twistDrive):
- spring amount of torque needed to move the joint to its t proportional to the angle from the target (not used for a velocity c

- damping applied to the drive spring (used to smooth out oscil drive target).
- forceLimit the maximum torque the drive can apply (note tha impulsive torque, depending on 1
 PxConstraintFlag::eDRIVE LIMITS ARE FORCES)
- acceleration drive flag. If this flag is set the acceleration (rathe applied by the drive is proportional to the angle from the target.

Best results will be achieved when the drive target inputs are consistreedom and limit constraints.

Note: if any angular degrees of freedom are locked, the SLERP drive ignored. If all angular degrees of freedom are unlocked, and parameter multiple angular drives, the SLERP parameters will be used.

Configuring Joints for Best Behavior

The behavior quality of joints in PhysX is largely determined by the at solver to converge. Better convergence can be achieved simply attributes of the PxRigidDynamic which controls the solver iteration coucan also be configured to produce better convergence.

- the solver can have difficulty converging well when a light ob between two heavy objects. Mass ratios of higher than 10 are be scenarios.
- when one body is significantly heavier than the other, make th second actor in the joint. Similarly, when one of the objects is sta the actor pointer is NULL) make the dynamic body the second acto

A common use for joints is to move objects around in the world. Best rewhen the solver has access to the velocity of motion as well as the char

- if you want a very stiff controller that moves the object to specific p consider jointing the object to a kinematic actor and use the function to move the actor.
- if you want a more springy controller, use a D6 joint with a driv desired position and orientation, and control the spring paran stiffness and damping. In general, acceleration drive is much easie drive.

When using mass scaling or when constraining bodies with infinite axes, the reduction in degrees of freedom of the rigid bodies continuation in floating point calculation can produce arbitrarily stiff of trying to correct unnoticeably small errors. This can appear, for example to perform 2D-simulation using infinite inertia to suppress velocity of simulation. In these cases, set the flag PxConstraintFlag::eDISABLE_P and set the minResponseThreshold on the constraint to a small value, result in such stiff constraint rows being ignored when encountered, an improve simulation quality.

Custom Constraints

It is also possible to add new joint types to PhysX. Use the exi PhysXExtensions library as a reference, and also the source for S which shows how to implement a Pulley Joint. Serializing custom objet the chapter *Serialization*, so the discussion here is limited to how to ε behavior in simulation. This is an advanced topic, and assumes a mathematics underlying rigid body simulation. The presentation here joint constrains two bodies; the case for a static body is equivalent to 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, execute in parallel and asynchronously, and should not access any 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 multi each function is called, PhysX passes a *constant block* which ca the joint configuration parameters (offsets, axes, limits etc).
- a static instance of PxConstraintShaderTable containing pointers to
- a class implementing the PxConstraintConnector interface, that cc joint to PhysX.

Defining Constraint Behavior

There are two functions that define the joint behavior: the *solver pr* which generates inputs to PhysX' velocity-based constraint solver, 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 using copy of the joint's constant block (so that the joint's copy may simulation without causing races).
- collision detection runs, and may wake bodies. If the joint connection simulation will ensure that either both bodies are awake, or neither
- for every joint connected to an awake body, the simulation calls the function.
- the solver updates body velocities and positions.
- if the constraint's ePROJECTION flag is set, the simulation calls function.

The Solver Preparation Function

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

The parameters are as follows:

- constraints is the output buffer of constraint rows.
- bodyAWorldOffset is the point, specified in world space as an offset bodyA, at which the constraint forces act to enforce the joint. The ignores this value as the information is already encoded in the composition when reporting forces it is necessary to choose a point at value considered to act. For PhysX joints, the attachment point of the 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 applie
 the purpose of resolving the joint. In the standard joints, these are j
 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 or a NULL pointer was supplied in constraint creation.
- *bB2w* is the transform of the second body. It is the identity trans static, or a NULL pointer was supplied in constraint creation.

The role of the solver preparation function is to populate the buffer o provide the point of application for force reporting, and provide properties. The return value is the number of Px1DConstraints gene buffer.

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

Each Px1D constraint constrains one degree of freedom between th structure looks like this:

```
struct Px1DConstraint
{
    PxVec3
                            linear0;
    PxReal
                            geometricError;
    PxVec3
                            angular0;
                            velocityTarget;
    PxReal
    PxVec3
                            linear1;
    PxReal
                            minImpulse;
    PxVec3
                            angular1;
    PxReal
                            maxImpulse;
    union
```

```
struct SpringModifiers
                           stiffness;
            PxReal
            PxReal
                           damping;
        } spring;
        struct RestitutionModifiers
            PxReal
                           restitution;
            PxReal
                           velocityThreshold;
        } bounce;
    } mods;
                          forInternalUse;
    PxReal
    PxU16
                          flags;
    PxU16
                          solveHint;
}
```

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

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

The constraint is treated as hard unless the Px1DConstraintFlag::eSPR
For both soft and hard constraints, the *solver velocity* for each row is the

```
v = body0vel.dot(lin0, ang0) - body1vel.dot(lin1, ang1)
```

Hard Constraints

For a hard constraint, the solver attempts to generate:

• a set of motion solver velocities vMotion for objects which, when the constraint errors, represented by the equation:

```
vMotion + (geometricError / timestep) = velocityTarget
```

 a set of post-simulation solver velocities vNext for the objects constraints:

```
vNext = velocityTarget
```

The motion velocities are used for integration and then discarded. T velocities are the values that getLinearVelocity() and getAngularVelocity

There are two special options for hard constraints, both most often limits: restitution and velocity biasing. They are set by the constraint flag and eKEEPBIAS, are mutually exclusive, and restitution takes priority (restitution is set, biasing is ignored).

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

```
restitution * -vCurrent
```

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:

```
vNext + (geometricError / timestep) = velocityTarget
```

This can be useful if, for example, the joint is approaching a limit but h it. If the target velocity is 0 and the geometric error is the distance ren the solver will constrain the velocity below that required to violate the lin The joint should then converge smoothly to the limit.

Soft Constraints

Alternatively, the solver can attempt to resolve the velocity constraint as In this case, the motion velocity vMotion and post-simulation velocity vMotion solver solves the equation:

```
F = stiffness * -geometricError + damping * (velocityTarget - v)
```

where F is the constraint force.

Springs are fully implicit: that is, the force or acceleration is a function velocity after the solve. There is one special option that applies only acceleration springs (PxConstraintFlag::eACCELERATION). With this o scale the magnitude of the force in accordance with the response effectively it implicitly solves the equation:

```
acceleration = stiffness * -geometricError + damping * (velocityT
```

Force Limits and Reporting

All constraints support limits on the minimum or maximum impulse ap There is a special flag for force limits: eHAS_DRIVE_FORCE_LIMIT. If force limits will be scaled by the tir PxConstraintFlag::eLIMITS_ARE_FORCES is set for the constraint.

The flag eOUTPUT_FORCE flag on a 1D constraint determines whether for this row should be included in the constraint force output. The repused internally to determine joint breakage. For example, if creating a angular drive that breaks when the stress on the linear part exceeds a flag for the linear equality rows but not the angular drive rows.

Solver Preprocessing

The joint solver attempts to preprocess hard constraints to improve solveHint value controls preprocessing for each row:

• if the constraint is a hard equality constraint with unbounded imp

impulse limits are -PX_MAX_FLT and PX_MAX_FLT) PxConstraintSolveHint::eEQUALITY.

- If one of the force limits is zero and the other unborned pxConstraintSolveHint::eINEQUALITY.
- for all soft constraints, and hard constraints with impulse limits oth set it to PxConstraintSolveHint::eNONE.

The solver does not check that the hint value is consistent with Px1DConstraint. Using inconsistent values may result in undefined behavior

The Projection Function

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

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

The Constraint Shader Table

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

```
PxConstraintVisualize visualize;
};
```

The constraint visualizer allows the joint to generate visualization information percentage interface. The functionality of this interface is towards the standard joints; examples of its use can be found in the extension of the standard joints; examples of its use can be found in the extension of the standard joints; examples of its use can be found in the extension of the standard joints; examples of its use can be found in the extension of the standard joints; examples of its use can be found in the extension of the standard joints.

Data Management

Next, define the class which lets PhysX manage the joint. This class the PxConstraintConnector interface.

To create a joint, call PxPhysics::createConstraint. The arguments to tl 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:

These functions are usually boilerplate; sample implementations car joints in the extensions library:

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

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

- onComShift() is called when the application calls setCMassLocalPactors connected by the joint. This is provided because the solv projection functions are defined using the frame of the underlying joint configuration is typically defined in terms of the actors.
- onOriginShift() is called when the application shifts the origin o necessary because some joints may have a NULL actor, signif attached to the world frame.
- getExternalReference() is used by PhysX to report simulation constraints, particularly breakage. The returned pointer is pass application in the event callback, along with the typeID which the appropriate type. The typeID sheach custom joint type, and different from any of the values in Px. If the joint also implements the PxBase interface, use the concre PxBase for the typeID.

NVIDIA PhysX SDK 3.4.1 Documentation »

<u>User's Guide »</u>

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

An articulation is a single actor comprising a set of links (each of wh rigid body) connected together with special joints. Every articulatic structure - so there can be no loops or breaks. Their primary use is n actuated characters. They support higher mass ratios, more accurate better dynamic stability and a more robust recovery from joint separa PhysX joints. However, they are considerably more expensive to simula

Although articulations do not directly build on joints, they use very s mechanisms. In this section we assume familiarity with PhysX joints.

Creating an Articulation

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

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

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

```
PxArticulationLink* link = articulation->createLink(parent, linkP
PxRigidActorExt::createExclusiveShape(*link, linkGeometry, materi
PxRigidBodyExt::updateMassAndInertia(*link, 1.0f);
```

Articulation links have a restricted subset of the functionality of rigid bobe kinematic, and they do not support damping, velocity clamping thresholds. Sleep state and solver iteration counts are properties of threather than the individual links.

Each time a link is created beyond the first, a *PxArticulationJoint* is created beyond the first parent.

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

Finally, add the articulation to the scene:

```
scene.addArticulation(articulation);
```

Articulation Joints

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

The swing limit is a hard elliptical cone limit which does not support strom movement perpendicular to the limit surface. You can set the lin follows:

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

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

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

The twist limit allows configuration of upper and lower angles:

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

and again you must explicitly enable it:

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

As usual with joint limits, it is good practice to use a sufficient limit cor that the solver will start to enforce the limit before the limit threshold is ϵ

Articulation joints are not breakable, and it is not possible to retrieve t applied at the joint.

Driving an Articulation

Articulations are driven through joint acceleration springs. You can target, an angular velocity target, and spring and damping parameter strongly the joint drives towards the target. You can also set compliance how strongly a joint resists acceleration. A compliance near zero incresistance, and a compliance of 1 indicates no resistance.

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

Note that with joint acceleration springs, the required strength of the using just the mass of the two bodies connected by the joint. By contras springs account for the masses of all the bodies in the articulation, and actuation at other joints. This estimation is an iterative process, co externalDriveIterations and internalDriveIterations properties of the PxA

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

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

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

```
PxTransform cA2w = parentPose.transform(joint.parentPose);
PxTransform cB2w = childPose.transform(joint.childPose);
transforms.cB2cA = transforms.cA2w.transformInv(transforms.cB2w
if(transforms.cB2cA.q.w<0)
    transforms.cB2cA.q = -transforms.cB2cA.q;</pre>
```

```
// 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 speci articulation is projected back together automatically. Projection is an ite the PxArticulation functions PxArticulation::setSeparation7 PxArticulation::setMaxProjectionIterations() control when projection occ 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 articuto sleep if each individual articulation link fulfills the sleep criteria.

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

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 vec

PxScene::shiftOrigin(const PxVec3& shift)

The positions of all objects in the scene and the corresponding data adjusted to reflect the new origin location (basically, the shift vector from all object positions). The intended use pattern for this API is to s that object positions move closer towards zero. Please note the responsibility to keep track of the summed total origin shift and adj to/from PhysX accordingly. Even though this method preserves som cached data, it is still an expensive operation and we recommend to use where distance related precision issues may arise in areas far from the modules of PhysX are used like the character controller or vehicle lib necessary to propagate the scene shift to those modules as well. Plea documentation of these modules for details.

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

Introduction

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

Using GPU Rigid Bodies

GPU rigid bodies are no more difficult to use than CPU rigid bodies. GF the exact same API and same classes as CPU rigid bodies. GPU rigic is enabled on a per-scene basis. If enabled, all rigid bodies occupying processed by the GPU. This feature is implemented in CUDA and requi or later compatible GPU. If no compatible device is found, simulation wi CPU and corresponding error messages will be provided.

This feature is split into two components: rigid body dynamics and broadenabled using PxSceneFlag::eENABLE_GPU_DYNAMICS at PxSceneDesc::broadphaseType to PxBroadPhaseType::eGPU reproperties are immutable properties of the scene. In addition, you must context manager and set the GPU dispatcher on the PxSceneDest requirement to make use of GPU-accelerated particles or clademonstrating how to enable GPU rigid body simulation is provided in The code example below serves as a brief reference:

```
PxCudaContextManagerDesc cudaContextManagerDesc;

gCudaContextManager = PxCreateCudaContextManager(*gFoundation, cu

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 c shape/body management and the GPU-accelerated constraint solver. T 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 MI body dynamics or combine GPU broad phase with GPU rigid body dynamics or combine GPU broad phase with GPU rigid body dynamics.)

What is GPU accelerated?

The GPU rigid body feature provides GPU-accelerated implementations

- Broad Phase
- Contact generation
- Shape and body management
- Constraint solver

All other features are performed on the CPU.

There are several caveats to GPU contact generation. These are as foll

- GPU contact generation supports only boxes, convex hulls, trian heightfields. Any spheres, capsules or planes will have contact guardeness those shapes processed on the CPU, rather than GPU.
- Convex hulls require PxCookingParam::buildGRBData to be set to required to perform contact generation on the GPU. If a hull to vertices or more than 32 vertices per-face is used, it will be proces the PxConvexFlag::eGPU_COMPATIBLE flag is used when the couthe limits are applied to ensure the resulting hull can be used on GI
- Triangle meshes require PxCookingParam::buildGRBData to be data required to process the mesh on the GPU. If this flag is not the GPU data for the mesh will be absent and any contact gener mesh will be processed on CPU.
- Any pairs requesting contact modification will be processed on the
- PxSceneFlag::eENABLE_PCM must be enabled for GPU contact
 performed. This is the only form of contact generation implement
 eENABLE_PCM is not raised, contact generation will be process
 pairs using the non distance-based legacy contact generation.

Irrespective of whether contact generation for a given pair is processe the GPU solver will process all pairs with contacts that request collisio filter shader.

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

The GPU rigid body solver provides full support for joints and conta performance is achieved using D6 joints because D6 joints are nativel GPU, i.e. the full solver pipeline from prep to solve is implemented on the types are supported by the GPU solver but their joint shaders are rur will incur some additional host-side performance overhead compared to

Tuning

Unlike CPU PhysX, the GPU rigid bodies feature is not able to dy buffers. Therefore, it is necessary to provide some fixed buffer sizes body feature. If insufficient memory is available, the system will is discard contacts/constraints/pairs, which means that behavior may be The following buffers are adjustable in PxSceneDesc::gpuDynamicsCor

```
struct PxgDynamicsMemoryConfig
{
        PxU32 constraintBufferCapacity; //!< Capacity of constrai
        PxU32 contactBufferCapacity; //!< Capacity of contact
        PxU32 tempBufferCapacity;
                                          //!< Capacity of temp buf
        PxU32 contactStreamCapacity; //!< Capacity of contact
        PxU32 patchStreamCapacity; //!< Capacity of the cont
PxU32 forceStreamCapacity; //!< Capacity of force bu
        PxU32 heapCapacity;
                                         //!< Initial capacity of
        PxU32 foundLostPairsCapacity; //!< Capacity of found an
        PxqDynamicsMemoryConfiq() :
                 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 apprigid bodies.

• constraintBufferCapacity defines the total amount of memory that constraints in the solver. If more memory is required, a warning further constraints will be created.

- contactBufferCapacity defines the size of a temporary contact constraint solver. If more memory is required, a warning is issued ε dropped.
- tempBufferCapacity defines the size of a buffer used for misce memory allocations used in the constraint solver.
- contactStreamCapacity defines the size of a buffer used to sto contact stream. This data is allocated in pinned host memory and If insufficient memory is allocated, a warning will be issued an dropped.
- patchStreamCapacity defines the size of a buffer used to store cor contact stream. This data is allocated in pinned host memory and If insufficient memory is allocated, a warning will be issued an dropped.
- forceStreamCapacity defines the size of a buffer used to report app to the user. This data is allocated in pinned host memory. If installocated, a warning will be issued and contacts will be dropped.
- heapCapacity defines the initial size of the GPU and pinned he Additional memory will be allocated if more memory is required. Th allocating memory can be relatively high so a custom heap allocate these costs.
- foundLostPairsCapacity defines the maximum number of found of GPU broad phase can produce in a single frame. This does not ling of pairs but only limits the number of new or lost pairs that can be of frame. If more pairs are detected or lost in a frame, an error is ended by the broad phase.

Performance Considerations

GPU rigid bodies can provide extremely large performance advantag bodies in scenes with several thousand active rigid bodies. However performance considerations to be taken into account.

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

D6 joints will provide best performance when used with GPU rigid bodie will be partially GPU-accelerated but the performance advantages will performance advantage exhibited by D6 joints.

Convex hulls with more than 64 vertices or with more than 32 vertices their contacts processed by the CPU rather than the GPU, so, if post counts within these limits. Vertex limits can be defined in cooking to convex hulls do not exceed these limits.

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

Modifying the state of actors forces data to be re-synced to the GPU, actors must be updated if the application adjusts global pose, velocities 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 accel processed on the CPU.

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

Introduction

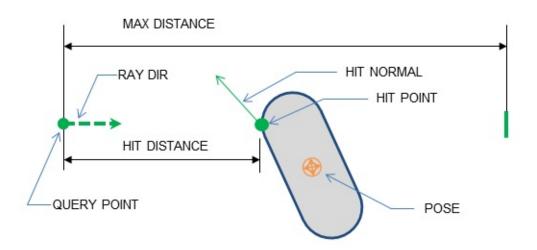
This chapter describes how to use PhysX' collision functionality with i objects. There are four main kinds of geometry queries:

- raycasts ("raycast queries") test a ray against a geometry object.
- sweeps ("sweep queries") move one geometry object along a line t of intersection with another geometry object.
- overlaps ("overlap queries") determine whether two geometry object
- penetration depth computations ("minimal translational distance quality here to "MTD") test two overlapping geometry objects to find the distance can be separated by the minimum distance.

In addition, PhysX provides helpers to compute the AABB of a geometry object.

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

Raycasts



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

The following code illustrates how to use a raycast query:

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 range. If the maximum distance is 0, a hit will only be returned if the shape, as detailed below for each geometry.

- geom is the geometry to test against.
- pose is the pose of the geometry.
- hitFlags specifies the values that should be returned by the que processing the query.
- maxHits is the maximum number of hits to return.
- hitInfo specifies the PxRaycastHit structure(s) into which the ray stored.
- The *anyHit* parameter is **deprecated**. It is equivalent to *PxHitF* which should be used instead.

The returned result is the number of intersections found. For ea *PxRaycastHit* is populated. The fields of this structure are as follows:

```
PxRigidActor*
                 actor;
PxShape*
                 shape;
PxVec3
                 position;
PxVec3
                normal;
PxF32
                distance;
PxHitFlags
                flags;
                faceIndex;
PxU32
PxF32
                u, v;
```

Some fields are optional, and the flags field indicates which members h result values. The query will fill fields in the output structure if the corres set in the input - for example, if the *PxHitFlag::ePOSITION* is set in the query will fill in the *PxRaycastHit::position* field, and set the *PxHitFlag::PxRaycastHit::flags*. If the input flag is not set for a specific member, may or may not contain valid data for that member. Omitting the *ePOSITION* flags in the input can sometimes result in faster queries.

For a raycast which is not initially intersecting the geometry object, the f as follows (optional fields are listed together with the flag that controls the state of the stat

 actor and shape are not filled (these fields are used only in scene-Scene Queries).

- position (PxHitFlag::ePOSITION) is the position of the intersection.
- normal (PxHitFlag::eNORMAL) is the surface normal at the point of
- *distance* (*PxHitFlag::eDISTANCE*) is the distance along 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 me intersections, it is a triangle index. For convex mesh intersection index. For other shapes it is always set to 0xfffffff.
- *u* and *v* (*PxHitFlag::eUV*) are the barycentric coordinates of the i fields (and the flag) are supported only for meshes and heightfields

The position field is related to the barycentric coordinates via the follow v0, v1 and v2 are the vertices from the hit triangle:

position =
$$(1 - u - v)*v0 + u*v1 + v*v2$$
;

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

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

Exceptions to the above behavior may apply if a ray starts inside an ot PhysX may not be able to compute meaningful output values for so cases the field will remain unmodified and the corresponding flag will a 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 N

For solid objects (sphere, capsule, box, convex) at most 1 result is a origin is inside a solid object:

• the reported hit distance is set to zero, and the PxHitFlag::eDIST/

the output.

- the hit normal is set to be the opposite of the ray's c PxHitFlag::eNORMAL flag is set in the output.
- the hit impact position is set to the ray's origin and the *PxHitFlag:*: set in the output.

If the start or end point of a ray is very close to the surface of the objec as being on either side of the surface.

Raycasts against Planes

For raycasts, a plane is treated as an infinite single-sided quad that ince (note that this is not the same as for overlaps). At most one result is a ray origin is behind the plane's surface, no hit will be reported ever intersects the plane.

If the start or end point of a ray is very close to the plane, it may be t either side of the plane.

Raycasts against Triangle Meshes

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

- if maxHits is 1 and *PxHitFlag::eMESH_ANY* is not set, the query wi intersection.
- if maxHits is 1 and *PxHitFlag::eMESH_ANY* is set, the query will intersection. Use this when it is sufficient to know whether or not the e.g. for line-of-sight queries or shadow rays.
- if maxHits is greater than 1, the query will return multiple intersection
 If more than maxHits intersection points exist, there is no guaran
 will include the closest. Use this for e.g. wall-piercing bullets that hi
 or where special filtering is required. Note that PxHitFlag::eMESI

be used in this case.

In general "any hit" queries are faster than "closest hit" queries, and "are faster than "multiple hits" queries.

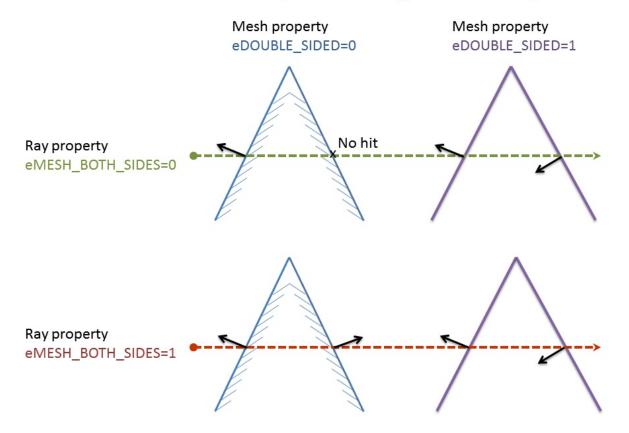
By default, back face hits (where the triangle's outward-facing normal product with the ray direction) are culled, and so for any triangle hit the will have a negative dot product with the ray direction. This behavior in the mesh instance's *PxMeshGeometryFlag::eDOUBLE_SIDED* flag *PxHitFlag::eMESH_BOTH_SIDES* flag:

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

For example a transparent glass window could be modeled as a doul that a ray would hit either side with the reported normal facing of direction. A raycast tracing the path of a bullet that may penetrate the find emerge from the back could use *eMESH_BOTH_SIDES* to find be 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.

Returned hit normals (with eMESH_MULTIPLE set)



To use *PxHitFlag::eMESH_BOTH_SIDES* for selected meshes rather the *PxQueryFilterCallback*.

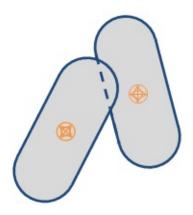
If the start or end point of a ray is very close to the surface of a triangle as being on either side of the triangle.

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

Raycasts against Heightfields

- Heightfields are treated the same way as triangle meshes with n shape space) in +y direction when thickness is <=0 and in -y direct is >0.
- Double-sided heightfields are treated the same way as double side

Overlaps



Overlap queries simply check whether two geometry objects over geometries must be a box, sphere, capsule or convex, and the other materials are convex.

The following code illustrates how to use an overlap query:

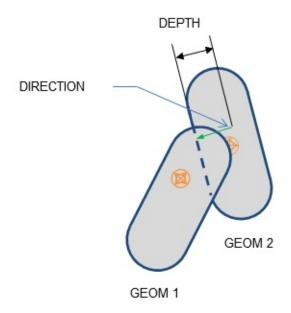
```
bool isOverlapping = overlap(geom0, pose0, geom1, pose1);
```

Overlaps do not support hit flags and return only a boolean result.

- A plane is treated as a solid half-space: that is, everything be considered part of the volume.
- Triangle meshes are treated as thin triangle surfaces rather than so
- Heightfields are treated as triangle surface extruded by their geometries that do not intersect with the heightfield surface | extruded space will report a hit.

If more than a boolean result is needed for meshes and hei *PxMeshQuery* API instead (see *PxMeshQuery*).

Penetration Depth



When two objects are intersecting, PhysX can compute the minimal dis by which the objects must be translated to separate them (this qua referred to as MTD, for *minimum translational distance*, as it is the vector by which translation will separate the shapes). One geometry objects sphere, capsule or convex mesh, and the other may be of any type.

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

The arguments are interpreted as follows:

- *direction* is set to the direction in which the first object should be tradepenetrate from the second.
- *distance* is set to the distance by which the first object should be tradepenetrate 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 and depth fields. Translating the first object by the depenetration vec depth will separate the two objects. If the function returns true, the always be positive or zero. If objects do not overlap, the function returns of the direction and distance fields are undefined.

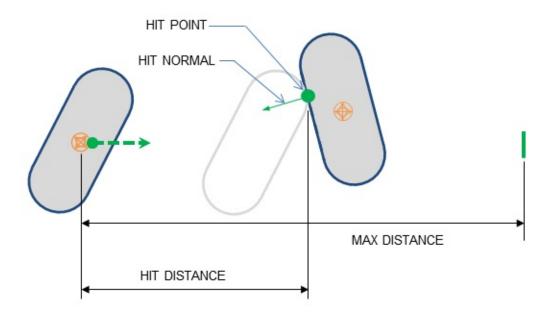
For simple (convex) shapes, returned results are accurate.

For meshes and heightfields, an iterative algorithm is used and dedic exposed in *PxExtensions*:

Here, *maxIter* is the maximum number of iterations for the algorithm, ar output argument which will be set to the number of iterations perform detected, *nb* is set to zero. The code will attempt at most maxIter iteration if a dependent of a dependent of the code will attempt at most maxIter iteration.

These functions only compute an approximate depenetration vector, at the amount of overlap between the geometry object and the mesh/hei particular, an intersection with a triangle will be ignored when the object the triangle, and if this holds for all intersecting triangles then no overlate 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 impround. PhysX only supports sweep queries where the first geometry object described 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:

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 murange, 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 geoused 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 dat impact is found.
- *inflation* inflates the first geometry with a shell extending outwa surface, making any corners rounded. It can be used to ensure a space is kept around the geometry when using sweeps to test wh possible.

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

```
PxRigidActor* actor;
PxShape* shape;
PxVec3 position;
PxVec3 normal;
PxF32 distance;
PxHitFlags flags;
PxU32 faceIndex;
```

- actor and shape are not filled (these fields are used only in scene Scene Queries).
- position (PxHitFlag::ePOSITION) is the position of the intersection
 multiple impact points, such as two boxes meeting face-to-face, Pt

- point arbitrarily. More detailed information for meshes or height field using the functions in *PxMeshQuery*.
- normal (PxHitFlag::eNORMAL) is the surface normal at the point o vector, pointing outwards from the hit object and backwards along t (in the sense that the dot product between the sweep direction and is negative).
- *distance* (*PxHitFlag::eDISTANCE*) is the distance along 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 not from the swept object. For triangle mesh and height field in triangle index. For convex mesh intersections it is a polygon index. is always set to 0xffffffff. For convex meshes the face index cor expensive. The face index computation can be disabled by not p query hit flag *PxHitFlag::eFACE_INDEX*. If needed the face in computed externally using the function *PxFindFaceIndex* which is 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 be considered part of the volume to sweep against.
- The same backface-culling rules as for raycasts apply for sweep difference that *eMESH_MULTIPLE* is not supported.

Initial Overlaps

Similarly to a raycast starting inside an object, a sweep may start with initially intersecting. By default PhysX will detect and report PxSweepHit::hadInitialOverlap() to see if the hit was generated by an in

For triangle meshes and height fields, backface culling is perform 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 dependentation 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 dependentation algorithm.
 - For other geometry types, the index is set to 0xfffffff.

This flag will incur additional processing overhead in the case of a addition, the following restrictions apply:

 PxHitFlag::eMTD is incompatible with PxHitFlag::ePRECI-PxHitFlag::eASSUME_NO_INITIAL_OVERLAP (see below). Using in conjunction with either of these flags will result in a warning be flag(s) that are incompatible with PxHitFlag::eMTD being ignored.

Testing for initial overlaps sometimes uses a specialized code performance penalty. If is it possible to guarantee that geometry obje overlapping, the check for overlaps can be s <code>PxHitFlag::eASSUME_NO_INITIAL_OVERLAP</code>. There are some restric this flag (also, see <code>Pitfalls</code>)

- Using *PxHitFlag::eASSUME_NO_INITIAL_OVERLAP* flag when initially overlap produces undefined behavior.
- PxHitFlag::eASSUME_NO_INITIAL_OVERLAP in combination 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 who overlap. If an overlap is detected, they are further culled by whether the behind the triangle, and if no triangles are found, the direction will be so sweep direction and the distance to 0.

Note: in most cases, translating the first geometry object by -normal*c separate the objects. However, an iterative dependentation algorithm is complete dependentation from the mesh in extreme cases. In this case to 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 a mesh is undefined when the eMTD flag is not set.

Precise Sweeps

PxHitFlag::ePRECISE_SWEEP enables more accurate sweep co potentially faster but less accurate solution is used). The ePRECISE_compatible with the inflation parameter, or with the flag PxHitFlag::eMTL

Sweeps against Height Fields

- Height fields are treated as thin triangle surfaces rather than solid c
- Thickness magnitude has no effect on initial overlap detection or po
- For single-sided height fields the normal of the hit will face in +y lc if thickness is < 0 and -y when thickness is > 0.
- Height fields are treated as double sided if either one of eD 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.

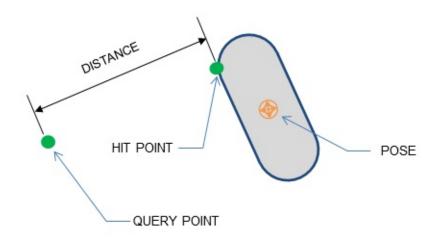
Pitfalls

There are some pitfalls to be aware of when using sweeps:

- Due to numerical precision issues, incorrect results may be returne have very large size disparities.
- Due to algorithmic differences, a sweep query may detect a difference overlapping shapes than an overlap query. In particular, it is not so an overlap check in order to determine the PxHitFlag::eIGNORE_INITIAL_OVERLAP flag. Applications that overlap/sweep/penetration depth information should use sweep

overlap testing and the *PxHitFlag::eMTD* flag.

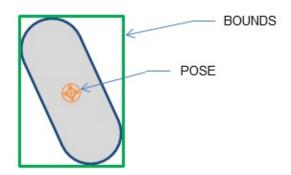
Additional PxGeometryQuery functions



The following function computes the distance between a point and a Only solid objects (box, sphere, capsule, convex) are supported:

```
PxReal dist = PxGeometryQuery::pointDistance(point, geom, pose, c
```

closestPoint is an optional output argument which returns the closest pc



The following function computes the axis-aligned bounding box (A geometry object, given its pose:

PxBounds3 bounds = PxGeometryQuery::getWorldBounds(geom, pose, in

The bounding box is scaled by the inflation value, which defaults to 1 specified.

PxMeshQuery

PhysX provides additional functionality for obtaining multiple results for height field overlaps, and for sweeping against arrays of triangles. On and capsules may be tested against meshes or heightfields using these

Mesh Overlaps

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

```
PxU32 triangleIndexBuffer[bufferSize];
PxU32 startIndex = 0;
bool bufferOverflowOccured = false;
PxU32 nbTriangles = PxMeshQuery::findOverlapTriangleMesh(sphereGe meshGeom triangle startInd

for(PxU32 i=0; i < nbTriangles; i++)
{
    PxTriangle tri;
    PxU32 vertexIndices[3];
    PxMeshQuery::getTriangle(meshGeom, meshPose, triangleIndexBuf ... // process triangle info
}</pre>
```

The findOverlapTriangleMesh method is used to extract the indices of the

- sphereGeom and spherePose specify the region to test for overlap
- meshGeom and meshPose specify the mesh and its pose.
- triangleIndexBuffer and triangleSize specify the output buffer and it
- *startIndex* is used to restart the query if the buffer size is exceed query for more triangles set this parameter to the number retrieved
- bufferOverflowOccured is set if more triangles would be returned f

would fit in the buffer.

Similar query functionality exists for height fields.

Sweeps against Triangles

Sometimes, for example, when using the mesh overlap API, it is conversed against groups of triangles. PhysX provides a function specifica with the following signature:

The arguments are interpreted as follows:

- unitDir, distance, geom and pose function identically to the first PxGeometryQuery::sweep(). distance is clamped to PX_MAX_SWI
- *triangleCount* is the number of triangles contained in the buffe 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. To optimization when repeatedly sweeping against the same set of triangle to test first.
- inflation functions identically to the inflation parameter of PxGeome
- doubleSided indicates whether the input triangles are double-sic equivalent to the PxMeshGeometryFlag::eDOUBLE_SIDED f suppresses backface culling, and for any hit the returned normal fa

sweep direction (see Raycasts against Triangle Meshes).

This function has extra limitations compared to the other sweep queries

- the geometry type must be either a sphere, a capsule or a box. C not supported.
- the function returns a single hit. Multiple hits (ar PxHitFlag::eMESH_MULTIPLE) are not supported.
- The function always returns the closest hit.
- The only supported flags are PxF PxHitFlag::eASSUME_NO_INITIAL_OVERLAP, PxHitFlag::eF PxHitFlag::eMESH_BOTH_SIDES and PxHitFlag::eMESH_ANY.

The function tests each input triangle in the order they are given. By d will test all triangles and return the closest sweep hit (if a hit has PxHitFlag::eMESH_ANY is used, the function will return as soon as a hit the remaining untested triangles). This flag can also be a PxHitFlag::eMESH_MULTIPLE, by calling the function PxHitFlag::eMESH_ANY, using as a starting point the previously re (whose index, between 0 and 'triangleCount', is available in sweepHit.fa

NVIDIA PhysX SDK 3.4.1 Documentation » previous | next User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Applications commonly need to efficiently query volumes in space or transported through space to determine what might be there. PhysX support this, one for objects already in a scene, and one for querying again AABBs. The scene query system is discussed in *Scene Queries*.

PxSpatialIndex

PxSpatialIndex is a BVH data structure that allows spatial queries to be the need to instantiate a PxScene. It supports insertion, removal ar objects defining a bounding box, and raycasts, sweeps, and overlap qu bounds.

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

SnippetSpatialIndex shows an example of how to use this class.

PxSpatialIndex has no internal locking, and there are special considera from multiple threads. Query operations (marked const in the interface) in parallel with update (non-const) operations, or update operations in other. When issuing query operations in parallel, it is important PxSpatialIndex defers some updates to its internal data structures unti In a single-threaded context this does not affect correctness or safety, from multiple threads simultaneously the internal updates may cause order to avoid these, call the flush() method to force the updates immediately. Between a call to flushUpdates() and any subsequent queries may be safely issued in parallel.

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

```
PxAgain onHit(PxSpatialIndexItem& item, PxReal distance, PxReal&
{
    PX_UNUSED(distance);
    Sphere& s = static_cast<Sphere&>(item);
    PxRaycastHit hitData;
```

```
// the ray hit the sphere's AABB, now we do a ray-sphere inte
    // the ray hit the sphere
    PxU32 hit = PxGeometryQuery::raycast(position, direction,
                                          PxSphereGeometry(s.radiu
                                          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)</pre>
    {
        closest = hitData.distance;
        hitSphere = &s;
        shrunkDistance = hitData.distance;
    }
    // and continue the query
    return true;
}
```

Note: Methods in PxGeometryQuery may report positive results wher within a numerical tolerance of intersection or impact. To obtain results when using PxSpatialIndex and when not using a culling hierarchy, the must be slightly padded. PxGeometryQuery::getWorldBounds adds this default.

PxSpatialIndex has the same performance characteristics as the sc using the PxPruningStructureType::eDYNAMIC_AABB_TREE optio correspond to moving objects, or there are many insertions and delet the tree may degrade over time. In order to prevent this, the tree may using the function *rebuildFull()*. Alternatively, a second tree may be but the background over many small steps, using the function *rebuildStep* same incremental rebuild step as performed by the scene's dynamic during *fetchResults()*. See *PxPruningStructureType* for details.

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

	· -		

Introduction

PhysX provides methods in PxScene to perform collision queries attached shapes in the scene. There are three types of queries: ray overlaps, and each can return either a single result, or multiple results each query traverses a culling structure containing the scene objects, test using the GeometryQuery functions (see *Geometry Queries*), an results. Filtering may occur before or after precise testing.

The scene uses two different query structures, one for *PxRigidStatic* ac for *PxRigidBody* actors (*PxRigidDynamic* and *PxArticulationLink*.) The be configured to use different culling implementations depending 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 alo follows:

In this code snippet a PxRaycastBuffer object is used to receive resul query. A call to raycast() returns true if there was a hit. hit.hadBlock is there was a hit. The distance for raycasts has to be in the [0, inf) range.

Raycasts results include position, normal, hit distance, shape and acto with UV coordinates for triangle meshes and heightfields. Before using PxHitFlag::ePOSITION, eNORMAL, eDISTANCE, eUV flags first, as it are not set.

Sweeps

A *PxScene::sweep()* query is geometrically similar to a raycast(): a Px swept from a specified initial pose in a direction unitDir with specified r find the points of impacts of the geometry with scene objects. The ma 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

Sweeps results include position, normal, hit distance, shape and actor for triangle meshes and heightfields.

Overlaps

PxScene::overlap() query searches a region enclosed by a specific overlapping objects in the scene. The region is specified as a transfc capsule or convex geometry.

A PxOverlapBuffer object is used to receive results from overlap() queri-

Overlaps results only include actor/shape and faceIndex since there is intersection.

Touching and blocking hits

For queries with multiple results we distinguish between *touching* and choice of whether a hit is touching or blocking is made by the user-in logic. Intuitively a blocking hit prevents further progress of a raycast or path, and a touching hit is recorded but allows the ray or sweep to conti hit query will return the closest blocking hit if one exists, together with that are closer. If there are no blocking hits, all touching hits will be return

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 re (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 indicating that there is no need to look for the clc encountered will do. This mode is most often used for boolean blc queries. Performance improvement may be a factor of 3 or more, deperor activate this mode use PxQueryFlag::eANY_HIT filter data flexQueryFilterData object, for instance:

Multiple hits

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

- To activate this mode for raycasts use the PxRaycastBuffer coprovided buffer for touching hits.
- In this mode all hits default to 'touching' type and are

PxRaycastBuffer::touches array.

For instance:

```
PxScene* scene;
PxVec3 origin = ...;
                                    // [in] Ray origin
                                    // [in] Normalized ray direc
PxVec3 unitDir = ...;
PxReal maxDistance = ...;
                                    // [in] Raycast max distance
const PxU32 bufferSize = 256;
                                    // [in] size of 'hitBuffer'
PxRaycastHit hitBuffer[bufferSize]; // [out] User provided buffe
PxRaycastBuffer buf(hitBuffer, bufferSize); // [out] Blocking and
// Raycast against all static & dynamic objects (no filtering)
// The main result from this call are all hits along the ray, sto
scene->raycast(origin, unitDir, maxDistance, buf);
for (PxU32 i = 0; i < buf.nbTouches; i++)</pre>
    animateLeaves(buf.touches[i]);
4
```

The same mechanism is used for overlaps (use PxOverlapBuffer with I sweeps (PxSweepBuffer with PxSweepHit[]).

Multiple hits with blocking hit

In the snippet for multiple hits above we only expected touching hits. If encountered along with touching hits, it will be reported in PxHitBuffer:: the touch buffer will contain only touching hits which are closer. This cc in scenarios such as bullets going through windows (breaking them on t of a tree (making them rustle) until they hit a blocking object (a concrete

```
// same initialization code as in the snippet for multiple hits
bool hadBlockingHit = scene->raycast(origin, unitDir, maxDistance
if (hadBlockingHit)
    drawWallDecal(buf.block);
for (PxU32 i = 0; i < buf.nbTouches; i++)
{
    assert(buf.touches[i].distance <= buf.block.distance);
    animateLeaves(buf.touches[i]);
}</pre>
```

- By default, hits are assumed to be touching when a touch buffer is filter callback should return PxQueryHitType::eBLOCK to denote th See *Filtering* for details.
- For overlap() queries all touching hits will be recorded even if encountered and PxQueryFlag::eNO_BLOCK flag is set.

Filtering

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

- a PxQueryFilterData structure, containing both PxQueryFlags and I
- an optional *PxQueryFilterCallback*

PxQueryFlag::eSTATIC, PxQueryFlag::eDYNAMIC

PxQueryFlag::eSTATIC and PxQueryFlag::eDYNAMIC flags control should include shapes from the static and/or dynamic query structure efficient way to filter out all static/dynamic shapes. For example an example applies forces to all dynamics in a region could use a spherical overlap PxQueryFlag::eDYNAMIC flag to exclude all statics since forces cal static objects. By default both statics and dynamics are included in quer

For instance:

PxQueryFlag::ePREFILTER, PxQueryFlag::ePOSTFILTER

Scene queries are performed in three phases: broad phase, midphase a

- Broad phase traverses the global scene spatial partitioning structure candidates for mid and narrow phases.
- midphase traverses the triangle mesh and heightfield internal cu find a smaller subset of the triangles in a mesh reported by the broa
- Narrow phase performs exact intersection tests (ray test for ray
 exact sweep shape tests or overlap tests for sweep() and overlap()

To implement custom filtering in queries, set the *PxQueryFlag::eF PxQueryFlag::ePOSTFILTER* flags and subclass *PxQueryFilterCallbac* filtering logic.

- Pre-filtering happens before midphase and narrow phase and al efficiently discarded before the potentially expensive exact collisio are more expensive for triangle meshes, heightfields, convexes than raycast and overlap tests involving only simple shapes capsules and boxes.)
- Post-filtering happens after the narrow phase test and can therefor the test (such as PxRaycastHit.position) to determine whethe discarded or not. These results can be accessed via the *hit* inpost-filtering callback (PxQueryFilterCallback::postFilter). static_cast<PxRaycastHit&>(hit), access data specific to a rasimilarly for overlaps (PxOverlapHit) and sweeps (PxSweepHit.)

The implementation of a filtering callback returns a *PxQueryHitType* res

- *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 calle PxHitCallback::nbTouches and PxHitCallback::touches parameters, ethat are no further (touchDistance <= blockDistance) than the closest

will be reported. For example, to record all hits from a raycast queTOUCH.

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

Note: eBLOCK should not be returned from user filters for overlap(). I result in undefined behavior, and a warning will be issued. If the PxQueryFlag::eNO_BLOCK flag is set, the eBLOCK will instead be aut converted to an eTOUCH and the warning suppressed.

PxQueryFlag::eANY_HIT

Use this flag to force the query to report the first encountered hit (whi closest) as a blocking hit. Performance may be more than three times on the scenario. Best gains can be expected for long raycasts/swe 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 return eTOUCH or in cases when no blocking hits are expected (in this case the performance hint.) All hits will then be reported as touching regardless or return value. The hit callback/buffer object provided to the query is requizero *PxHitBuffer::touches* buffer when this flag is used. Significant should only be expected for scenarios where the touching hit buffer ove

Note: this flag overrides the return value from pre and post-filter funct were previously returned as blocking will instead be returned as touchi

PxFilterData fixed function filtering

A fast, fixed-function filter is provided by *PxFilterData*, a 4*32-bit bitmas in filtering equation. Each shape has a bitmask (set via PxShape::set and the query also has a bitmask.

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

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

Or in other words:

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

```
enum ActiveGroup
{
    GROUP1 = (1<<0),
    GROUP2 = (1<<1),
    GROUP3 = (1<<2),
    GROUP4 = (1<<3),
    ...
};</pre>
```

When shapes are created, they can be assigned to the a group, for exa

```
PxShape* shape; // Previously created shape
```

```
PxFilterData filterData;
filterData.word0 = GROUP1;
shape->setQueryFilterData(filterData);
```

Or to multiple groups, for example GROUP1 and GROUP3:

```
PxShape* shape;  // Previously created shape

PxFilterData filterData;
filterData.word0 = GROUP1|GROUP3;
shape->setQueryFilterData(filterData);
```

When performing a scene query, select which groups are active for example GROUP2 and GROUP3 - as follows:

User defined hit callbacks for unbounded results

Queries can sometimes return a very large number of results (for exavery large objects or in areas with high object density), and it can expensive to reserve a sufficiently large memory buffer. The classes F PxSweepCallback and PxOverlapCallback provide efficient callback that such scenarios. For instance a *raycast* query with a PxRaycastCal return all touch hits via multiple virtual PxHitCallback::processTouches()

```
struct UserCallback : PxRaycastCallback
{
    UserData data;
    virtual PxAgain processTouches(const PxRaycastHit* buffer, Px
        // This callback can be issued multiple times and can be
        // to process an unbounded number of touching hits.
        // Each reported touching hit in buffer is guaranteed to
        // the final block hit after the query has fully executed
    {
        for (PxU32 i = 0; i < nbHits; i++)
            animateLeaves(buffer[i], data);
    virtual void finalizeQuery()
        drawWallDecal(this->block, data);
};
PxScene* scene;
                       // [in] Ray origin
// [in] Normalized ray direc
PxVec3 origin = ...;
PxVec3 unitDir = ...;
PxReal maxDistance = ...;
                                    // [in] Raycast max distance
UserCallback cb; cb.data = ...;
scene->raycast(origin, unitDir, maxDistance, cb); // see UserCall
```

In this code snippet the raycast() query will potentially invoke procestimes, with all touching hits already clipped to the globally nearest block

ullet Note that the query can be up to twice as expensive in case all ϵ

not fit in the provided touches buffer and a blocking hit was also for

• Also see PxQueryFlag::eNO_BLOCK

Batched queries

PhysX supports batching of scene queries via the *PxBatchQuery* interfmay simplify multi-threaded implementations.

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

- PxBatchQuery interface facilitates batching and execution of together. PxBatchQuery buffers raycast, overlap and swe PxBatchQuery::execute() is called.
- Use PxScene::createBatchQuery(const PxBatchQueryDesc& d PxBatchQuery object.
- The hardcoded filtering equation is not used for batched queries. In with two filter shaders, respectively running before (*PxBatchQue* and after (*PxBatchQueryPostFilterShader*) the exact per-shape *PxBatchQueryDesc::preFilterShader* and *PxBatchQueryDesc::post*
- BatchQueryFilterData::filterShaderData will be copied and passed via the constantBlock parameter.
- Results are written to user-defined buffers PxBatch
 PxBatchQueryDesc, in the same order queries were queued in object.
- The results and hits buffers for the each query type used (rayca: are specified separately.
- These buffers can be changed before each batch query execute produce a warning for batched queries with NULL results or I corresponding query type (raycast, overlap or sweep).

Volume Caching

PxVolumeCache provides a mechanism for accelerating scene q implements caching for objects within a specified volume and provide PxScene for executing raycasts, overlaps, and sweeps. PxVolumeCa performance boost when objects within the same localized region of multiple times, either within the same simulation frame or on a later fran

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 partic localized cloud.
- Multiple short range character controller raycasts within the sam character.
- Caching query results across multiple frames, the cache can be f volume on previous frame (possibly extruded in the anticipation movement) and then queried with a smaller volume.

The cache has a maximum capacity, specified separately for dynamic in *PxScene::createVolumeCache()*.

For purposes of multithreaded access, any operation on the cache co on the scene.

Filling the Cache

To fill the cache, call *PxVolumeCache::fill()*. This will query the overlapping with the volume defined by the geometry and transform at in an internal buffer up to the maximum sizes for static and dyna PxBoxGeometry, PxSphereGeometry and PxCapsuleGeometry a cacheVolume. The call will always refill both the static and dynamic int

if the new volume lies entirely within the previous cached volume. It returns the PxVolumeCache::FillStatus.

Subsequent queries against the cache (raycasts, overlaps, sweeps, for cache automatically using the same volume if the scene query sull updated since the last fill. The update status is tracked independent dynamics, so a query might only refill the cache for dynamics while re results for statics. If any attempt to fill or refill fails, the cache is subsequent query will attempt to fill it.

Querying the Cache

PxVolumeCache provides an API for raycasts, sweeps and overlaps to scene query API. The main difference in signatures is that *Single Ob*, supported for PxVolumeCache queries. Query results are PxVolumeCache::Iterator::shapes() callback, and the query may in multiple times to deliver multiple batches of results.

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

Since the cache refills automatically on any query where the scene h two conditions guarantee that a query against the cache that lies entire volume will always return exactly the same shapes as querying the states of the entirely within the cache volume (and the cache is valid) which overlap the cache volume will be returned. If a query is issued which fill() has never been called, an error is reported.

The cache also provides a low-level *forEach()* mechanism that iterate objects. If *forEach()* is executed on a cache for which *fill()* has never return without reporting an error. If the cache is invalid, forEach() will I that overlap the cached volume directly from the scene. This pro

allocation of a temporary buffer, and if the allocation fails, *forEach()* message and return.

This code snippet shows how to use PxVolumeCache:

```
PxScene* scene;
PxVec3 poi = ...;
                                     // point of interest
PxVec3 origin = ...;
                                     // [in] Ray origin
                                    // [in] Normalized ray direc
PxVec3 unitDir = ...;
PxReal maxDistance = ...;
                                    // [in] Raycast max distance
PxRaycastBuffer hit;
                                     // [out] Raycast results
const PxU32 maxStatics = 32, maxDynamics = 8;
// persistent cache, valid until invalidated by object movement,
// insertion or deletion
PxVolumeCache* cache = scene->createVolumeCache(maxStatics, maxDy
cache->setMaxNbStaticShapes(64); cache->setMaxNbDynamicShapes(16)
// fill the cache using a box geometry centered around the point
cache->fill(PxBoxGeometry(PxVec3(1.0f)), PxTransform(position));
. . .
// Perform multiple raycast queries using the cache
PxRaycastBuffer hit;
const bool status = cache->raycast(origin, unitDir, maxDistance,
// low level iterator for stored actor/shape pairs
struct UserIterator : PxVolumeCache::Iterator
    UserData userData;
    virtual void shapes(PxU32 count, const PxActorShape* actorSha
        for (PxU32 i = 0; i < count; i++)
           doSomething(actorShapePairs[i].actor, actorShapePairs[
   iter;
}
// invoke UserIterator::shapes() callback for all actor/shape pai
cache->forEach(iter);
```

Single Object Caching

Another special case mechanism for accelerating scene queries is sin using *PxQueryCache*.

- This cache can provide additional speedups and memory saving sweep queries in any operation mode.
- The cache object defines which shape should be tested first. Fo temporal coherence, this can provide significant performance gair to capture that coherence is simply to fill the cache object of a gi eBLOCK result (last blocking shape) from the previous frame.
- Note that it is likely incorrect to use a past touching hit (recorded 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 retur sight blocking shape for several frames. Using a *raycast* query will *PxQueryCache* object will allow PhysX to test a single shape - be internal spatial partitioning structures, and in case of a "cache hit" th bypassed entirely. For instance:

```
PxScene* scene;
PxVec3 origin = ...;
                                     // [in] Ray origin
                                    // [in] Normalized ray direc
PxVec3 unitDir = ...;
                                     // [in] Raycast max distance
PxReal maxDistance = ...;
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 sha
const PxQueryCache* cache = persistentCache.shape ? &persistentCa
// Perform a raycast query using the cache
const bool status = scene->raycast(origin, unitDir, maxDistance,
                                   PxHitFlags(PxHitFlag::eDEFAULT
```

```
PxQueryFilterData(), NULL, cac
if(status)
{
    // We hit a shape. Cache it for next frame.
    persistentCache.shape = hit.block.shape;
    persistentCache.faceIndex = hit.block.faceIndex;
}
else
{
    // We did not hit anything. Reset the cache for next frame.
    persistentCache = PxQueryCache();
}
```

Caching can also be useful in queries looking for the closest blocking hi eANY_HIT flag. In this case, testing the previously closest object first a shorten the query distance very early, leading to fewer total narrow pl and early out from the traversal.

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

Note: Overlaps do not support single hit blocking caches.

PxPruningStructureType

PhysX SDK offers different pruning structures which are used to acqueries. This paragraph describes the differences between them.

Generalities

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

The grid builds quickly, in O(n) time, with queries executing in between depending on how uniformly the objects are distributed in space, with case performance of O(N) when all objects are clustered in the same gr

The tree builds in O(n log(n)) time, but queries with a single result typicatime. Queries returning multiple results will traverse more of the tree, the a query returning all of the objects in the scene in O(n) time. The tradegeneration when the same topology is maintained too long as object and in pathological cases query performance may degrade to O(n) time

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

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

To avoid automatic resizing triggered by insertions into internal structures, reserve the space in advance. See *PxSceneDesc::maxNiPxSceneDesc::maxNbDynamicShapes*.

PxPruningStructureType::eNONE

The acceleration structure is similar to a hierarchical grid. Committing (full rebuild. This is a good choice if you expect to rarely or never update structure.

PxPruningStructureType::eSTATIC_AABB_TREE

The acceleration structure is a tree. Committing changes requires a f generally recommended, but can be a good choice for staticStructure if your scene are created on initialization, and not modified thereafter. If or remove static geometry, the default eDYNAMIC_AABB_TREE setting choice, although it has a higher memory footprint than that of eSTATIC_

PxPruningStructureType::eDYNAMIC_AABB_TREE

In this case, both the tree and the grid are used, and each query seal and the grid.

The tree is initially built by the first commit. Once a tree is built, conceeds as follows:: * the tree is refitted in accordance with update object it contains. * added objects are inserted into the grid. Such additionable objects currently in the grid, or changes to AABBs of objects in the grebuilt.

In addition, a new tree is incrementally built during fetchResults(), frames controlled by PxScene's dynamicTreeRebuildRateHint attribut starts, it includes all of the objects in the current tree and grid. Whe frames later, the new tree is refitted in accordance with any AABB ch since the build started, and then replaces the current tree. Any object since the start of the build remain in the grid.

To force a full immediate rebuild, call PxScene::forceDynamicTreeRet 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 refittin degrade the quality of the current tree

PxPruningStructure

Provides access to precomputed pruning structure used to acceler against newly added actors.

A pruning structure can be provided to PxScene::addActors. The a shapes will then be directly merged into the scenes AABB tree, with AABB tree recompute:

```
// Create pruning structure from given actors.
PxPruningStructure* ps = PxPhysics::createPruningStructure(&actor
// Add actors into a scene together with the precomputed pruning
PxScene::addActors(*ps);
ps->release();
```

A PxPruningStructure object can be serialized into a collection together

For usage of PxPruningStructure please refer to the snippet SnippetPru

A typical use case for PxPruningStructure is a large world scenario whe positioned actors get streamed in.

Merge process

The merge process into the scene query acceleration structure *PxPruningStructureType*: * eSTATIC_AABB_TREE - the pruning st directly into scene's AABBtree. This might unbalance the tree and it i recompute the static tree at some point. * eDYNAMIC_AABB_TR structure is merged into a temporary pruning structures until the scer AABB tree is computed.

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Introduction

PhysX support for vehicles has been significantly reworked in 3.> NxWheelShape class of 2.8.x, a more optimal integration of the corvehicle simulation code has been developed. More specifically, the v now sits outside the core SDK in a manner similar to PhysXExtens allows vehicles to be updated in a single pass as well as promotir approach to modeling vehicle data. Vehicles support has been ϵ suspension/wheel/tire modeling of 2.8.x to a more complete model that vehicle components including engine, clutch, gears, autobox, different quick chassis. glance suspensions, and Α at PxVehicleComponents.h will provide a flavor of the behaviors sur vehicles.

Algorithm

The PhysX Vehicle SDK models vehicles as collections of sprung mass represents a suspension line with associated wheel ar collections of sprung masses have a complementary representation as whose mass, center of mass, and moment of inertia matches exact coordinates of the sprung masses. This is illustrated below.

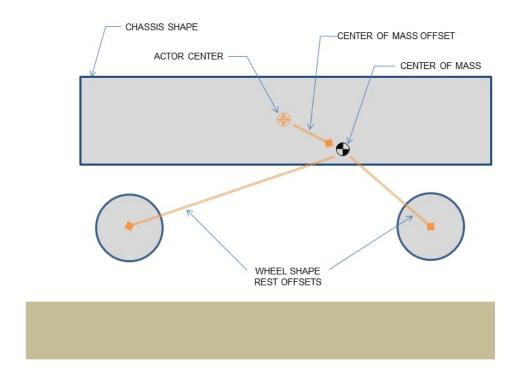


Figure 1a: Vehicle representation as a rigid body actor with shapes fo wheels. Note that the wheel rest offsets are specified relative to the

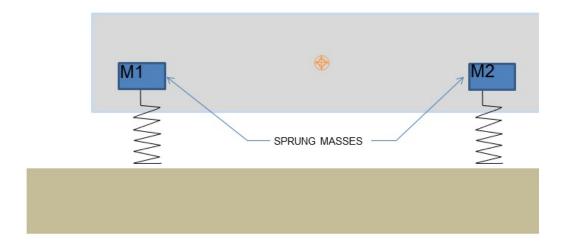


Figure 1b: Vehicle representation as a collection of sprung masses of

The relationship between the sprung mass and rigid body vehicle representationally formalized with the rigid body center of mass equations:

$$M = M1 + M2$$

 $Xcm = (M1 \times X1 + M2 \times X2)/(M1 + M2)$

where M1 and M2 are the sprung masses; X1 and X2 are the sprung n actor space; M is the rigid body mass; and Xcm is the rigid body center

The purpose of the PhysX Vehicle SDK update function is to compute strong forces using the sprung mass model and then to apply the aggregate the PhysX SDK rigid body representation in the form of a modified vervelocity. Interaction of the rigid body actor with other scene objects update is then managed by the PhysX SDK.

The update of each vehicle begins with a raycast for each suspensio starts just above the top of the tire at maximum compression and along the direction of suspension travel to a position just below the b maximum droop. This is shown in the diagram below.

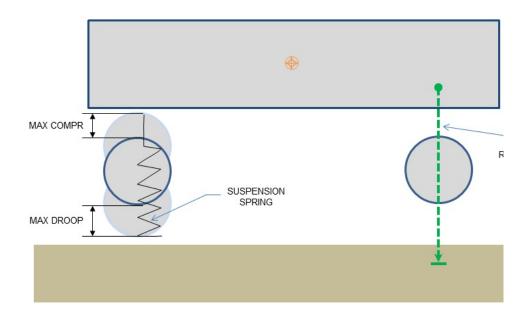


Figure 2: Suspension limits and suspension raycasts

The suspension force from each elongated or compressed spring is co to the aggregate force to be applied to the rigid body. Additionally, the sused to compute the load that is bearing down on the tire. This load is the tire forces that will be generated in the contact plane and then addeforce to be applied to the rigid body. The tire force computation actunumber of factors including steer angle, camber angle, friction, wheel rigid body momentum. The aggregated force of all tire and suspens applied to the rigid body actor associated with the vehicle so that the modified accordingly in the next PhysX SDK update.

In addition to being collections of sprung masses, PhysX vehicles also a drive models. The center of the drive model is a torsion clutch, which converses and the engine via forces that arise from differences in rotation sides of the clutch. At one side of the clutch is the engine, which is porthe accelerator pedal. The engine is modeled as a rigid body whos rotational and limited to a single degree of rotational freedom. At the clutch are the gearing system, the differential and the wheels. The 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 throt

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 foun *Guide*.

First Code

Vehicle SDK Initialization

Before using the vehicle SDK it must first be initialized in order to s threshold values from various tolerance scales. This is as straightfor following function:

```
PX_C_EXPORT bool PX_CALL_CONV PxInitVehicleSDK (PxPhysics& physics, PxSerializationRegistry* serializationRe
```

This function should be called after setting up the required PxPhysics instances. If vehicle serialization is required a PxSerializationRegistry be specified. A PxSerializationRegistry instance can be PxSerialization::createSerializationRegistry(), see *Serialization*.

The basis vectors of the vehicle simulation must also be configured and lateral tire slips may be unambiguously computed:

```
void PxVehicleSetBasisVectors(const PxVec3& up, const PxVec3& for
```

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

The rigid body actors associated with vehicles can be updated eithe velocity modifications or updated with an acceleration that is applied SDK simulate call. The following function can be used to select the requ

```
void PxVehicleSetUpdateMode(PxVehicleUpdateMode::Enum vehicleUpda
```

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

This needs to be called before the PxPhysics instance and PxFounce

released; that is, the order of shutdown is the reverse of the initializatio if serialization is required the PxSerializationRegistry specified for PxInito 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:

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

The shutdown code in SnippetVehicle4W is as follows:

```
PxCloseVehicleSDK();
```

Introduction To Vehicle Creation

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

```
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(numWhee vehDrive4W->setup(physics, veh4WActor, *wheelsSimData, driveSimDa wheelsSimData->free();
```

The code above first instantiates a PxVehicleWheelsSimData instance vehicle 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 PxVehic instance. This structure stores the configuration of the drive model and such as engine peak torque, clutch strength, gearing ratios, and A correction. Following this, a PxRigidDynamicActor is instantiated ar geometry for the wheels and chassis as well as dynamic propertion moment of inertia, and center of mass. The final step is to instantiate a instance and associate it with the actor and the vehicle configuration da

The functions setupWheelsSimulationData, setupDriveSimData and set actually quite involved and shall be discussed in setupWheelsSimulationData, setupDriveSimData and setupVehicleActo

Introduction To Vehicle Update

The PhysX Vehicles SDK utilizes batched scene queries to query the each tire. A more detailed discussion of PhysX batched scene queries Section *Batched queries*.

The following pseudo-code initializes a batched scene query with buffe a single vehicle with four wheels:

```
PxRaycastQueryResult sqResults[4];
PxRaycastHit sqHitBuffer[4];
PxBatchQueryDesc sqDesc(4, 0, 0);
sqDesc.queryMemory.userRaycastResultBuffer = sqResults;
sqDesc.queryMemory.userRaycastTouchBuffer = sqHitBuffer;
sqDesc.queryMemory.raycastTouchBufferSize = 4;
sqDesc.preFilterShader = myFilterShader;
PxBatchQuery* batchQuery = scene->createBatchQuery(sqDesc);
```

The PxBatchQuery instance is typically instantiated as part of the initial then reused each frame. It is possible to instantiate a PxBatchQuery vehicle or to instantiate a single PxBatchQuery instance with buffers lawheels of a batched array of vehicles. The only restriction is that all batched associated buffers configured at the start of a vehicle simulation until the end of the vehicle simulation frame.

PhysX vehicles make use of scene query filter shaders to eliminate int vehicle issuing the raycast and with any geometry that is not to be drivable surface. More details for how to set up "myFilterShader" abor Section *Filtering*.

For a batch containing just a single 4-wheeled vehicle the suspensic performed with the following pseudo-code:

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

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

The vehicles are updated with the following function call:

```
PxVehicleUpdates(timestep, gravity, frictionPairs, 1, vehicles, N
```

The function PxVehicleUpdates updates the internal dynamics of each wheel shapes of the vehicle's actor and applies either velocity or accel the actor, depending on the update mode chosen with PxVehicleSetl details can be found in Section *Wheel Pose* and Section *Vehicle Upda* frictionPairs is basically a lookup table that associates unique fr combinations of tire type and PxMaterial. The idea here is to allow t tuned for each surface type. This shall be discussed in more depth in S *on Drivable Surfaces*.

Snippets

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

- SnippetVehicle4W
- 2. SnippetVehicleTank
 - SnippetNoDrive
- 3. SnippetVehicleScale
- 4. SnippetVehicleMultiThreading
 - 5. SnippetVehicleWheelContactMod

Code snippets from each of these is used throughout the guide.

SnippetVehicle4W

SnippetVehicle4W demonstrates how to instantiate and update PxVehicleDrive4W. It creates a vehicle on a plane and then controls the performs a number of choreographed maneuvers such as accelerate handbrake, and turn.

SnippetVehicleTank

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

SnippetVehicleNoDrive

SnippetVehicleNoDrive demonstrates how to instantiate and update PxVehicleNoDrive. It creates a vehicle on a plane and then controls the performs a number of choreographed manoeuvres such as accelerate, and hard turns.

SnippetVehicleScale

SnippetVehicleScale demonstrates how to configure a PhysX vehicle w the chosen length scale. The snippet sets up a vehicle with meters as scale and then modifies the vehicle parameters so that they represen but with centimeters as the chosen length scale.

SnippetVehicleMultiThreading

SnippetVehicleMultiThreading demonstrates how to implement multi-the creates multiple vehicles on a plane and then concurrently simulate across multiple threads.

Advanced Concepts

Vehicle Creation

This Section discusses the configuration of vehicle simulation data an set up an actor that will represent the vehicle in the PhysX SDK. Sect *Vehicle Creation* identified three distinct phases of vehicle configuration wheel simulation data, configuration of drive simulation data and actor of these phases is discussed in turn.

setupWheelsSimulationData

The following code, taken from SnippetVehicle4W, PxVehicleWheelsSimData:

```
void setupWheelsSimulationData(const PxF32 wheelMass, const PxF32
    const PxF32 wheelRadius, const PxF32 wheelWidth, const PxU32
    const PxVec3* wheelCenterActorOffsets, const PxVec3& chassisC
    const PxF32 chassisMass, PxVehicleWheelsSimData* wheelsSimDat
{
    //Set up the wheels.
    PxVehicleWheelData wheels[PX MAX NB WHEELS];
        //Set up the wheel data structures with mass, moi, radius
        for(PxU32 i = 0; i < numWheels; i++)
            wheels[i].mMass = wheelMass;
            wheels[i].mMOI = wheelMOI;
            wheels[i].mRadius = wheelRadius;
            wheels[i].mWidth = wheelWidth;
        }
        //Enable the handbrake for the rear wheels only.
        wheels[PxVehicleDrive4WWheelOrder::eREAR_LEFT].mMaxHandBr
        wheels[PxVehicleDrive4WWheelOrder::eREAR_RIGHT].mMaxHandB
        //Enable steering for the front wheels only.
        wheels[PxVehicleDrive4WWheelOrder::eFRONT_LEFT].mMaxSteer
        wheels[PxVehicleDrive4WWheelOrder::eFRONT_RIGHT].mMaxStee
    }
```

```
//Set up the tires.
PxVehicleTireData tires[PX_MAX_NB_WHEELS];
    //Set up the tires.
    for(PxU32 i = 0; i < numWheels; i++)
        tires[i].mType = TIRE_TYPE_NORMAL;
    }
}
//Set up the suspensions
PxVehicleSuspensionData suspensions[PX_MAX_NB_WHEELS];
    //Compute the mass supported by each suspension spring.
    PxF32 suspSprungMasses[PX_MAX_NB_WHEELS];
    PxVehicleComputeSprungMasses
        (numWheels, wheelCenterActorOffsets,
         chassisCMOffset, chassisMass, 1, suspSprungMasses);
    //Set the suspension data.
    for(PxU32 i = 0; i < numWheels; i++)
    {
        suspensions[i].mMaxCompression = 0.3f;
        suspensions[i].mMaxDroop = 0.1f;
        suspensions[i].mSpringStrength = 35000.0f;
        suspensions[i].mSpringDamperRate = 4500.0f;
        suspensions[i].mSprungMass = suspSprungMasses[i];
    }
   //Set the camber angles.
    const PxF32 camberAngleAtRest=0.0;
    const PxF32 camberAngleAtMaxDroop=0.01f;
    const PxF32 camberAngleAtMaxCompression=-0.01f;
    for(PxU32 i = 0; i < numWheels; i+=2)
        suspensions[i + 0].mCamberAtRest = camberAngleAtRest
        suspensions[i + 1].mCamberAtRest = -camberAngleAtRes
        suspensions[i + 0].mCamberAtMaxDroop = camberAngleAtM
        suspensions[i + 1].mCamberAtMaxDroop = -camberAngleAt
        suspensions[i + 0].mCamberAtMaxCompression = camberAn
        suspensions[i + 1].mCamberAtMaxCompression = -camberA
    }
}
//Set up the wheel geometry.
PxVec3 suspTravelDirections[PX_MAX_NB_WHEELS];
PxVec3 wheelCentreCMOffsets[PX_MAX_NB_WHEELS];
```

```
PxVec3 suspForceAppCMOffsets[PX_MAX_NB_WHEELS];
    PXVec3 tireForceAppCMOffsets[PX_MAX_NB_WHEELS];
        //Set the geometry data.
        for(PxU32 i = 0; i < numWheels; i++)
            //Vertical suspension travel.
            suspTravelDirections[i] = PxVec3(0, -1,0);
            //Wheel center offset is offset from rigid body cente
            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,wheelCentre
            //Tire force application point 0.3 metres below
            //rigid body center of mass.
            tireForceAppCMOffsets[i] =
                PxVec3(wheelCentreCMOffsets[i].x,-0.3f,wheelCentre
        }
    }
    //Set up the filter data of the raycast that will be issued b
    PxFilterData gryFilterData;
    setupNonDrivableSurface(gryFilterData);
    //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, suspTravelDirect
        wheelsSimData->setWheelCentreOffset(i, wheelCentreCMOffse
        wheelsSimData->setSuspForceAppPointOffset(i, suspForceApp
        wheelsSimData->setTireForceAppPointOffset(i, tireForceApp
        wheelsSimData->setSceneQueryFilterData(i, qryFilterData);
        wheelsSimData->setWheelShapeMapping(i, i);
    }
}
```

The function PxVehicleComputeSprungMasses computes the sprususpension so that they collectively match the rigid body center of mass in the frame of the actor. It makes sense to perform PxVehicleComput the frame of the actor because the rigid body center of mass is always actor's frame. The vehicle suspension system, on the other hand, is spe of mass frame. As a consequence, the functions set\ setSuspForceAppPointOffset and setTireForceAppPointOffset all descr rigid body center of mass. The directness of this approach can make c body center of mass a bit more involved than might be expected. To sol function PxVehicleUpdateCMassLocalPose has been introduced, thou code above. This function recomputes and sets all suspension offse sprung masses and sets them in a way that preserves the natural frequ ratio of each spring.

Details of many of the parameters and functions above can be founc *Guide*. The function setupNonDrivableSurface, which sets up scene c each suspension raycast, shall be discussed in more detail in Section the link between TIRE_TYPE_NORMAL and tire friction shall be mac *Tire Friction on Drivable Surfaces*. Finally, the use of the function setWl 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);
```

```
//Gears
    PxVehicleGearsData gears;
    gears.mSwitchTime=0.5f;
    driveSimData.setGearsData(gears);
    //Clutch
    PxVehicleClutchData clutch;
    clutch.mStrength=10.0f;
    driveSimData.setClutchData(clutch);
    //Ackermann steer accuracy
    PxVehicleAckermannGeometryData ackermann;
    ackermann.mAccuracy=1.0f;
    ackermann.mAxleSeparation=
        wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheel
        wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheel
    ackermann.mFrontWidth=
        wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheel
        wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheel
    ackermann.mRearWidth=
        wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheel
        wheelsSimData->getWheelCentreOffset(PxVehicleDrive4WWheel
    driveSimData.setAckermannGeometryData(ackermann);
}
```

Details of many of the parameters and functions above can be found *Guide*.

Configuring PxVehicleDriveSimDataNW and PxVehicleDriveSimDataTa a very similar procedure, albeit with slightly different components. M found, for example, in SnippetVehicleTank.

setupVehicleActor

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

```
PxRigidDynamic* createVehicleActor
(const PxVehicleChassisData& chassisData,
PxMaterial** wheelMaterials, PxConvexMesh** wheelConvexMeshes, c
PxMaterial** chassisMaterials, PxConvexMesh** chassisConvexMeshe
```

```
PxPhysics& physics)
{
        //We need a rigid body actor for the vehicle.
        //Don't forget to add the actor to the scene after settin
        PxRigidDynamic* vehActor = physics.createRigidDynamic(PxT
        //Wheel and chassis query filter data.
        //Optional: cars don't drive on other cars.
        PxFilterData wheelQryFilterData;
        setupNonDrivableSurface(wheelQryFilterData);
        PxFilterData chassisQryFilterData;
        setupNonDrivableSurface(chassisQryFilterData);
        //Add all the wheel shapes to the actor.
        for(PxU32 i = 0; i < numWheels; i++)
        {
                PxConvexMeshGeometry geom(wheelConvexMeshes[i]);
                PxShape* wheelShape=PxRigidActorExt::createExclus
                wheelShape->setQueryFilterData(wheelQryFilterData
                wheelShape->setSimulationFilterData(wheelSimFilte
                wheelShape->setLocalPose(PxTransform(PxIdentity))
        }
        //Add the chassis shapes to the actor.
        for(PxU32 i = 0; i < numChassisMeshes; i++)</pre>
        {
                PxShape* chassisShape=PxRigidActorExt::createExcl
                chassisShape->setQueryFilterData(chassisQryFilter
                chassisShape->setSimulationFilterData(chassisSimF
                chassisShape->setLocalPose(PxTransform(PxIdentity
        }
        vehActor->setMass(chassisData.mMass);
        vehActor->setMassSpaceInertiaTensor(chassisData.mMOI);
        vehActor->setCMassLocalPose(PxTransform(chassisData.mCMOf
        return vehActor;
```

The significance of wheelSimFilterData, chassisSimFilterData, whee chassisQryFilterData shall be discussed in Section *Filtering*. Further, the ordering of the wheel shapes in the above code at PxVehicleWheelsSimData::setWheelShapeMapping is clarified in Section

Filtering

In this Section the concepts behind vehicle query and vehicle simulation described.

The key goal of scene query and simulation filtering for vehicles is to e are supported by suspension spring forces without interference f intersection. The requirements for filtering are then as follows:

```
    wheel shapes must not hit drivable surfaces
    suspension raycasts can hit drivable surfaces
    suspension raycasts must not hit the shapes of the vehicle iss
```

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

The snippets also apply simulation filter data to wheel shapes as follows

```
PxFilterData wheelSimFilterData;
wheelSimFilterData.word0 = COLLISION_FLAG_WHEEL;
```

```
wheelSimFilterData.word1 = COLLISION_FLAG_WHEEL_AGAINST;
...
wheelShape->setSimulationFilterData(wheelSimFilterData);
```

Finally, the following simulation filter data is applied to drivable surfaces

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

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

A remarkably similar process may be employed to configure the conquery filters. This is accomplished in the vehicle snippets with the follow

```
void setupDrivableSurface(PxFilterData& filterData)
{
    filterData.word3 = (PxU32)DRIVABLE_SURFACE;
}
void setupNonDrivableSurface(PxFilterData& filterData)
{
    filterData.word3 = UNDRIVABLE_SURFACE;
}
PxQueryHitType::Enum WheelRaycastPreFilter
(PxFilterData filterData0, PxFilterData filterData1.
 const void* constantBlock, PxU32 constantBlockSize,
PxHitFlags& queryFlags)
{
    //filterData0 is the vehicle suspension raycast.
    //filterData1 is the shape potentially hit by the raycast.
    PX_UNUSED(constantBlockSize);
    PX_UNUSED(constantBlock);
    PX_UNUSED(filterData0);
```

```
PX_UNUSED(queryFlags);
  return ((0 == (filterData1.word3 & DRIVABLE_SURFACE)) ?
    PxQueryHitType::eNONE : PxQueryHitType::eBLOCK);
}
```

Each vehicle wheel is given filter data configured with setupNonDipassed to the vehicle with:

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

The parameter filterData0 in WheelRaycastPreFilter corresponds qryFilterData passed to the vel PxVehiceWheelsSimData::setSceneQueryFilterData. The parameter other hand, corresponds to the query filter data of a shape potentially h the vehicle snippets the shape of the drivable ground plane has scer configured with the function setupDrivableSurface. This satisfies the suspension raycasts can hit drivable surfaces. Vehicle shapes, on the configured with setupNonDrivableSurface. This satisfies the restrictic raycasts must not hit the vehicle issuing the raycasts but also prev driving on any other vehicles that might be added to the scene. This ex readily be avoided by employing a more complex filter shader that perhaps IDs encoded in both the shape filter data and the filter data applied Care must be taken, however, to configure the filters to ensure that su only interact with the shapes of other vehicles.

Note: It is vital that WheelRaycastPreFilter returns PxQueryHitType::eraycast hit is allowed for the filter data pair. Using PxQueryHitType::eBl that each raycast returns either no hits or just the hit closest to the star raycast. This is important because PxVehicleSuspensionRaycasts and PxVehicleUpdates expect a one-to-one correspondence between each element in the PxRaycastQueryResult and PxRaycastHit arrays passe query.

Tire Friction on Drivable Surfaces

In this Section setting up tire types, drivable surface types, ar combinations of tire and surface type shall be discussed.

To implement a unique friction value for each combination of tire type at first necessary to assign tire types to tires. In Section *setupWheelsSit* type was assigned to each tire:

```
//Set up the tires.
PxVehicleTireData tires[PX_MAX_NB_WHEELS];
{
    //Set up the tires.
    for(PxU32 i = 0; i < numWheels; i++)
    {
        tires[i].mType = TIRE_TYPE_NORMAL;
    }
}</pre>
```

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 b table of friction lookup and the values is managed PxVehicleDrivableSurfaceToTireFrictionPairs. To make the feature n of the PxMaterial is element array actually asso PxVehicleDrivableSurfaceType instance. This allows multiple PxMa share the same surface type.

In the vehicle snippets the following code makes the association betwee surface type and then associates each combination of tire and surface value:

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

PxMaterial* surfaceMaterials[1];
    surfaceMaterials[0] = defaultMaterial;
```

```
PxVehicleDrivableSurfaceToTireFrictionPairs* surfaceTirePairs
    PxVehicleDrivableSurfaceToTireFrictionPairs::allocate(MAX_MAX_NUM_SURFACE_TYPES);

surfaceTirePairs->setup(MAX_NUM_TIRE_TYPES, MAX_NUM_SURFACE_T surfaceMaterials, surfaceTypes);

for(PxU32 i = 0; i < MAX_NUM_SURFACE_TYPES; i++)
{
    for(PxU32 j = 0; j < MAX_NUM_TIRE_TYPES; j++)
        surfaceTirePairs->setTypePairFriction(i,j,gTireFricti)
    }
}
return surfaceTirePairs;
}
```

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

There is no upper bound on the friction values used in the PhysX vehic the maximum value of friction that obeys the laws of physics is 1.0, t SDK purposefully does not enforce this rule. One reason for this is that is far from a complete description of a real vehicle, meaning that some taken with friction values to generate the desired behavior. A more con certainly provide greater accuracy given a specific set of vehicle param all clear that it would provide a greater range of editable and contro have the performance characteristics required for games. Another rea not clamped at 1.0 is that games typically simulate the physics upo comes at a cost to numerical accuracy, especially when there are a r tire effects that require KHz update frequencies. One source of numeri amplitude of oscillation of the suspension, which is governed in turn b the vehicle falls under gravity between each update. At KHz updat simulation artifact is acceptably small, but not at 60Hz. The last reas simply no need to impose the strict rules of friction on the vehicles SI interesting behaviors to be generated that would perhaps be impossible by the laws of rigid body and tire dynamics. Having said all this, however model simulated at 60Hz ought to have enough integrity that only small should be necessary. If very large friction values are required, say great 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 an between calls to PxVehicleUpdates. Editing any of these values while P still executing will lead to undefined behavior.

Vehicle Controls

In this Section setting the control values used to drive a vehicle shall be

The simplest and most direct way to set vehicle control values is to function:

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

One of the difficulties with vehicle dynamics in games is knowing ho controller data in a way that results in pleasing handling. Players, accelerate by pressing very quickly on the accelerator trigger in a happen in a real car. This rapid acceleration can have a counter-product the resulting wheel spin reduces the lateral and longitudinal forces that by the tire. To help overcome some of these problems some optior provided to filter the control data from keyboard and gamepad.

A solution to the problem of filtering controller input data is to assign a each button or pad. For analog values under digital control it is possible or decrease the analog value at a specified rate depending on whether on or off. For analog values under analog control it makes more sense

previous input value to the current input at a specified rate. A slight $\mathfrak c$ simple model is that the difficulty of achieving a large steer angle at large be modeled. One technique to achieve this would be to model the fo aligning moments and apply these to a steering linkage model. T complicated and quite difficult to tune. A simpler solution might be $\mathfrak c$ steer value by another value in range (0,1) that decreases at high s method has been implemented in the helper classes and functions.

Rise and fall rates for digital and analog control have bee SnippetVehicle4W:

```
PxVehicleKeySmoothingData gKeySmoothingData=
{
    {
        3.0f,
                  //rise rate eANALOG INPUT ACCEL
        3.0f, //rise rate eANALOG_INPUT_BRAKE
        10.0f, //rise rate eANALOG_INPUT_HANDBRAKE
        2.5f,  //rise rate eANALOG_INPUT_STEER_LEFT
2.5f,  //rise rate eANALOG_INPUT_STEER_RIGHT
    },
                //fall rate eANALOG_INPUT__ACCEL
//fall rate eANALOG_INPUT__BRAKE
        5.0f,
        5.0f,
        10.0f, //fall rate eANALOG_INPUT__HANDBRAKE
                //fall rate eANALOG_INPUT_STEER_LEFT
        5.0f,
        5.0f
                 //fall rate eANALOG_INPUT_STEER_RIGHT
    }
};
PxVehiclePadSmoothingData gPadSmoothingData=
{
    {
        6.0f,
                 //rise rate eANALOG_INPUT_ACCEL
                  //rise rate eANALOG_INPUT_BRAKE
        6.0f,
        12.0f,
                 //rise rate eANALOG_INPUT_HANDBRAKE
        12.0f,
2.5f,
                  //rise rate eANALOG INPUT STEER LEFT
        2.5f,
                 //rise rate eANALOG_INPUT_STEER_RIGHT
    },
                 //fall rate eANALOG_INPUT_ACCEL
        10.0f,
        10.0f,
                 //fall rate eANALOG_INPUT_BRAKE
        12.0f,
                  //fall rate eANALOG_INPUT_HANDBRAKE
        5.0f,
                  //fall rate eANALOG_INPUT_STEER_LEFT
```

```
5.0f //fall rate eANALOG_INPUT_STEER_RIGHT }
};
```

A look-up table has also been specified to describe the maximum ste speed:

```
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,
    PX_MAX_F32, PX_MAX_F32
};
PxFixedSizeLookupTable<8> gSteerVsForwardSpeedTable(gSteerVsForwardSpeedTable)
```

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

```
gVehicleInputData.setDigitalAccel(true);
gVehicleInputData.setDigitalBrake(true);
gVehicleInputData.setDigitalHandbrake(true);
gVehicleInputData.setDigitalSteerLeft(true);
gVehicleInputData.setDigitalSteerRight(true);
gVehicleInputData.setGearUp(true);
gVehicleInputData.setGearDown(true);
```

or in the event that a gamepad is used:

```
gVehicleInputData.setAnalogAccel(1.0f);
gVehicleInputData.setAnalogBrake(1.0f);
gVehicleInputData.setAnalogHandbrake(1.0f);
gVehicleInputData.setAnalogSteer(1.0f);
gVehicleInputData.setGearUp(1.0f);
gVehicleInputData.setGearDown(1.0f);
```

Here, gVehicleInputData is an instance of the vehicle SI PxVehicleDrive4WRawInputData.

The vehicle SDK offers two optional functions to smooth the keyboard and apply the smoothed input values to the PhysX vehicle. If the vehi digital inputs then the following function is used:

```
PxVehicleDrive4WSmoothDigitalRawInputsAndSetAnalogInputs(gKeySmoogSteerVsForwardSpeedTable, carRawInputs,timestep,isInAir,(PxV
```

while gamepad controllers employ the following code:

```
PxVehicleDrive4WSmoothAnalogRawInputsAndSetAnalogInputs(gCarPadSm gSteerVsForwardSpeedTable, carRawInputs,timestep,(PxVehicleDr
```

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

In Section *Introduction To Vehicle Update* the functions used to perform vehicle updates were introduced. In this Section these separate updates 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 PxVehicleSusp subsequent updates it is not strictly necessary to issue fresh rayca

vehicle caches raycast hit planes that can be re-used. It is recommen 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 ve geometry with high spatial coherence. Support for vehicles that require detail is discussed in Section *Level of Detail*.

There is some freedom in the order in which raycasts can be issued redynamics update. In a real-world situation it might be that raycasts c separate thread at the end of the update loop so that they are ready f the next. However, this really all depends on the threading environment of rigid body updates.

Wheel Pose

PxVehicleUpdates poses the wheels shapes of the vehicle's actor to t steer, camber, and rotation angles. The computed pose also attempts geometry exactly on the contact plane identified by the raycast that we suspension line. To perform this function the PhysX Vehicles SDK ne shapes of the actor correspond to each wheel of the vehicle. This is function PxVehicleWheelsSimData::setWheelShapeMapping.

Note: The vehicle SDK has a default mapping for each wheel that is @PxVehicleWheelsSimData::setWheelShapeMapping(i,i). This needs co layout of the shapes is different from the default pattern.

Note: PxVehicleWheelsSimData::setWheelShapeMapping(i,-1) can b setting the local wheel pose. This is particularly useful if a wheel has neactor geometry.

The wheel pose is always within the limits PxVehicleSuspensionData::mMaxDroop

PxVehicleSuspensionData::mMaxCompression. If the suspension requires the wheel to be placed beyond the compression limit the whe

the compression limit and a rigid body constraint will handle the differer simulate() call.

Vehicle State Queries

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

Persistent wheel data is stored in PxVehicleWheelsDynData, while per data is stored in PxVehicleDriveDynData. The most useful functions are

```
PX_FORCE_INLINE PxReal PxVehicleDriveDynData::getEngineRotationSp
PxReal PxVehicleWheelsDynData::getWheelRotationSpeed(const PxU32
PxReal PxVehicleWheelsDynData::getWheelRotationAngle(const PxU32
```

To record non-persistent simulation data so that it may be later be function argument must be passed to PxVehicleUpdates. The follo records non-persistent data for a single 4-wheeled car:

```
PxWheelQueryResult wheelQueryResults[4];
PxVehicleWheelQueryResult vehicleWheelQueryResults[1] = {{wheelQueryResults[timestep, gravity, frictionPairs, 1, vehicles, vehicle
```

Here, a PxVehicleWheelQueryResult array, whose length equals at le vehicles in the batched vehicles array, is passed to PxVehi PxVehicleWheelQueryResult instance has a pointer to a PxWheelQueryResult instance 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-personal This feature allows memory budgets to be targeted at the vehicles of high

More Advanced Concepts

Vehicle Telemetry

The purpose of telemetry data is to expose the inner dynamics of the curve tuning through the use of telemetry graphs. In this Section initialization rendering of telemetry data shall be discussed.

Telemetry data is recorded by calling the following function:

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

The function above is identical to PxVehicleUpdates with the excepti update a single vehicle at a time and takes an extra function argument t

Setting up the telemetry data is relatively straightforward. In addit telemetry data streams, the PxVehicleTelemetryData structure also sto the size, position, and color scheme of the graph. The following pseu and configures telemetry data for a 4-wheeled vehicle:

```
wheelGraphPosX,wheelGraphPosY,
backgroundColor,lineColorHigh,lineColorLow);
```

The sizes, positions, and colors are all values that will be used to rend exact values of these fields will depend on the coordinate system and c visualize the telemetry data.

In the above example, the coordinates have been configured to rende graph in the center of the screen under the assumption that (1,1) is the of the screen and (0,0) the bottom right-hand side of the screen. Scree also been specified for rendering data associated with each of the four v

The following enumerated lists detail the telemetry data that is collected

```
enum
{
    eCHANNEL_JOUNCE=0,
    eCHANNEL_SUSPFORCE,
    eCHANNEL_TIRELOAD,
    eCHANNEL_NORMALISED_TIRELOAD,
    eCHANNEL WHEEL OMEGA,
    eCHANNEL_TIRE_FRICTION,
    eCHANNEL TIRE LONG SLIP,
    eCHANNEL_NORM_TIRE_LONG_FORCE,
    eCHANNEL_TIRE_LAT_SLIP,
    eCHANNEL_NORM_TIRE_LAT_FORCE,
    eCHANNEL NORM TIRE ALIGNING MOMENT,
    eMAX NUM WHEEL CHANNELS
};
enum
{
    eCHANNEL_ENGINE_REVS=0,
    eCHANNEL_ENGINE_DRIVE_TORQUE,
    eCHANNEL CLUTCH SLIP,
    eCHANNEL ACCEL CONTROL,
    eCHANNEL_BRAKE_CONTROL,
    eCHANNEL_HANDBRAKE_CONTROL,
    eCHANNEL_STEER_CONTROL,
    eCHANNEL_GEAR_RATIO,
    eMAX_NUM_ENGINE_CHANNELS
};
```

Data is collected for suspension jounce, suspension force, tire load, nowheel rotation speed, tire friction, tire longitudinal slip, tire longitudinal slip, tire lateral force, and tire aligning moment. Data is also collect engine revs, engine drive torque, clutch slip, applied acceleration/brak and gear ratio. For each graph all associated data is collected in separathat can be accessed after the update is complete.

Prior to rendering the graph of a particular wheel and channel the folk is required:

```
PxF32 xy[2*PxVehicleGraph::eMAX_NB_SAMPLES];
PxVec3 color[PxVehicleGraph::eMAX_NB_SAMPLES];
char title[PxVehicleGraph::eMAX_NB_TITLE_CHARS];
myTelemetryData->getWheelGraph(wheel).computeGraphChannel(PxVehic 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 gr engine's graph data. It also stores a color for each sample by lineColorHigh and lineColorLow depending on the value of the sa channel stores the last 256 samples so that a history of each paramete on the screen.

The PhysX Vehicles SDK does not render the graphs. This is an ϵ application because each has its own system for rendering debug inforr

Vehicle Update Multi-Threaded

The PhysX Vehicles SDK can be used in a multi-threaded environment of performance improvements arising from parallelism. The update ste exactly as described in Section *Vehicle Update* but with an extra sefunction PxVehiclePostUpdates after all concurrent calls to PxVehicleSt and PxVehicleUpdates are complete. PxVehiclePostUpdates perform normally executed in PxVehicleUpdates but which are not possible to call when concurrency is employed.

PxVehicleSuspensionRaycasts is a thread-safe function and can be without any modifications to the calling code with the exception, of commanaging the tasks and threads that will execute the raycasts concurred hand, PxVehicleUpdates as used in Section *Vehicle Update* is not requires an extra PxVehicleConcurrentUpdateData array to be speconcurrently executed. When this extra data is specified PxVehicle number of writes to PhysX actors that are involved in the vehicle updat writes are stored in the PxVehicleConcurrentUpdateData array during to PxVehicleUpdates and then executed sequentially in PxVehiclePostL

Sample code can be found in SnippetVehicleMultiThreading.

Tire Shaders

It is possible to replace the default tire model used by PhysX vehicles w This requires a shader function that can be set per-vehicle along wit must be set per-wheel:

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

The shader function must implement this function prototype:

```
typedef void (*PxVehicleComputeTireForce)
(const void* shaderData,
  const PxF32 tireFriction,
  const PxF32 longSlip, const PxF32 latSlip, const PxF32 camber,
  const PxF32 wheelOmega, const PxF32 wheelRadius, const PxF32 rec
  const PxF32 restTireLoad, const PxF32 normalisedTireLoad, const
  const PxF32 gravity, const PxF32 recipGravity,
  PxF32& wheelTorque, PxF32& tireLongForceMag, PxF32& tireLatForce
```

The vehicle update code will call the shader function for each wheel w for that wheel.

Vehicle Types

The PhysX Vehicle SDK supports four types of vehicle: PxVehicleDriveNW. PxVehicleDriveTank and PxVehicleNoDrive. PxVehicleDrive4W will be the best choice for rally cars, street cars PxVehicleDriveNW is very similar to PxVehicleDrive4W except that it that it allows all wheels to be coupled to the differential. This genera differential models of PxVehicleDriveNW cannot match the range or c PxVehicleDrive4W. PxVehicleDriveTank implements a simple but effic constraining the left and right wheel speeds to mimic the effect of t PxVehicleNoDrive implements a vehicle that is simply a rigid body wit wheels and tires. The idea here is to allow custom drive models such ε hovercraft to be implemented using PhysX vehicles.

PxVehicleDrive4W

The class PxVehicleDrive4W has already been discussed in sor discussion so far has focused on 4-wheeled vehicles. In the PxVehicleDrive4W shall be discussed with special reference to instar and more than 4 wheels.

3-Wheeled Cars

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

void PxVehicle4WEnable3WTadpoleMode(PxVehicleWheelsSimData& wheel
 PxVehicleWheelsDynData& wheelsDynData, PxVehicleDriveSimData4'
void PxVehicle4WEnable3WDeltaMode(PxVehicleWheelsSimData& wheelsS
 PxVehicleWheelsDynData& wheelsDynData, PxVehicleDriveSimData4'

These functions ensure that no raycast hits are returned for the diadditionally do some other work to decouple the disabled wheel from disable ackermann correction, re-position the opposite remaining wheel to a supersisting the suspension of the opposite remaining wheel to the opposite remaining wheel the opposite remaining wheel

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 popularing 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 att of a PxShape that may pose no longer PxVehicleWheelsSimData::setWheelShapeMapping can be used requirement. The second requirement is that any wheel that has been a rotation This zero wheel speed. can be satisfie PxVehicleWheelsDynData::setWheelRotationSpeed for the relevant requirement is that disabled wheels must receive no drive torqu requirement can actually be ignored because it is automatically enforce code called by the PxVehicleUpdates function. For vehicles of type Px' requirement on drive torque is fulfilled by ensuring that PxVehicleNoDri is never called with a non-zero torque value. Further, the drive torque r readily fulfilled for vehicles of type PxVehicleDriveNW by ensuring that disconnected from the disabled wheel. This is achieved PxVehicleDifferentialNWData::setDrivenWheel.

Configuring the differential of a PxVehicle4W to ensure that no drive to a disabled wheel is a little more complex because there are many achieve this. If the wheel is not a driven wheel then disabling the v satisfies the drive torque requirement because such wheels can never k differential. On the other hand, if the wheel has index eFRONT_LEFT c or eREAR_LEFT or eREAR_RIGHT then the differential does need enforce the requirement. One way to do this is to set up the differential torque to only the rear(front) wheels if a front(rear) wheel has been dis readily implemented by selecting front-wheel drive mode or rear-wh appropriate:

```
PxVehicleDifferential4WData diff = myVehicle.getDiffData();
if(PxVehicleDrive4WWheelOrder::eFRONT_LEFT == wheelToDisable ||
```

```
PxVehicleDrive4WWheelOrder::eFRONT_RIGHT == wheelToDisable)
{
    if(PxVehicleDifferential4WData::eDIFF_TYPE_LS_4WD == diff.mTy
      PxVehicleDifferential4WData::eDIFF TYPE LS FRONTWD == diff.
      PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_4WD == diff.mT
      PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_FRONTWD == dif
          diff.mBias = 1.3f;
          diff.mRearLeftRightSplit = 0.5f;
          diff.mType = PxVehicleDifferential4WData::eDIFF_TYPE_LS
          //could also be PxVehicleDifferential4WData::eDIFF_TYPE
      }
else if(PxVehicleDrive4WWheelOrder::eREAR LEFT == wheelToDisable
    PxVehicleDrive4WWheelOrder::eREAR RIGHT == wheelToDisable)
{
    if(PxVehicleDifferential4WData::eDIFF_TYPE_LS_4WD == diff.mTy
       PxVehicleDifferential4WData::eDIFF_TYPE_LS_REARWD == diff.
       PxVehicleDifferential4WData::eDIFF TYPE OPEN 4WD == diff.m
       PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_REARWD == dif
          diff.mBias = 1.3f;
          diff.mFronteftRightSplit = 0.5f;
          diff.mType = PxVehicleDifferential4WData::eDIFF TYPE LS
          //could also be PxVehicleDifferential4WData::eDIFF TYPE
       }
myVehicle.setDiffData(diff);
```

In some situations limiting the drove torque to just the front or rear w acceptable. If only a single wheel has been disabled then it is possible mode where 3 wheels are driven. This can be achieved by modifying delivers torque to all four wheels (eDIFF_TYPE_LS_4WD or eDIFF_T so that torque is only delivered to 3 wheels:

```
diff.mFrontLeftRightSplit = 0.0f;
          }
          else
              diff.mFrontLeftRightSplit = 1.0f;
      }
else if(PxVehicleDrive4WWheelOrder::eREAR_LEFT == wheelToDisable
    PxVehicleDrive4WWheelOrder::eREAR_RIGHT == wheelToDisable)
{
    if(PxVehicleDifferential4WData::eDIFF_TYPE_LS_4WD == diff.mTy
       PxVehicleDifferential4WData::eDIFF_TYPE_OPEN_4WD == diff.m
          if(PxVehicleDrive4WWheelOrder::eREAR LEFT == wheelToDis
              diff.mRearLeftRightSplit = 0.0f;
          else
              diff.mRearLeftRightSplit = 1.0f;
       }
myVehicle.setDiffData(diff);
```

In some situations it will make sense to disable Ackermann steer correction wheel was able to steer. In particular, if the remaining wheel of the fron positioned so that it is at the center of the axle then it would almost a Ackermann correction would be disabled. This can be achieved by sett zero (PxVehicleAckermannGeometryData::mAccuracy). The role of correction, however, really needs to be determined on a case by case b

N-Wheeled Cars

In addition to removing wheels from a vehicle, it is also possit PxVehicleDrive4W with more than 4 wheels but with the caveat that only driven. As a consequence of this caveat the functionality of the extra limited compared to the first 4 wheels. More specifically, only the connected to the differential or the steering; that is, only the first block experience a drive torque or a steer angle and only the first block of 4 w

the Ackermann steering correction. As a consequence, the extra whee 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 doe ability to call PxVehicle4WEnable3WTadpoleMode or PxVehicle4WEn These functions, however, are hard-coded to disable one of the 4 wh connected to the steering and driven through the differential.

The following pseudo-code illustrates the key steps in the creation PxVehicleDrive4W vehicle:

```
PxVehicleWheelsSimData* wheelsSimData=PxVehicleWheelsSimData::all
PxVehicleDriveSimData4W driveSimData;
setupSimData(wheelsSimData, driveSimData);
PxVehicleDrive4W* car = PxVehicleDrive4W::allocate(6);
PxRigidDynamic* vehActor=createVehicleActor6W();
car->setup(&physics, vehActor, *wheelsSimData, driveSimData, 2);
```

PxVehicleDriveNW

While the PxVehicleDrive4W allows cars with any number of wheels simulated it only allows 4 of those wheels to be driven by engin differential. The vehicle type PxVehicleDriveNW has been introduced to limitation. This vehicle class makes use of the differential type PxVehic class that allows any or all of the vehicle's wheels to be coupled to the limitation that the torque available at the differential is always divided wheels that are coupled to the differential. The generality of PxVel advanced features such as limited slip differentials and Ackermann ameaning that only a simple equal-split differential model can be provided

The following pseudo-code illustrates the key steps in the creation PxVehicleDriveNW vehicle:

```
PxVehicleWheelsSimData* wheelsSimData=PxVehicleWheelsSimData::all
PxVehicleDriveSimDataNW driveSimData;
setupSimData(wheelsSimData, driveSimData);
PxVehicleDriveNW* car = PxVehicleDriveNW::allocate(6);
PxRigidDynamic* vehActor=createVehicleActorNW();
car->setup(&physics, vehActor, *wheelsSimData, driveSimData, 6);
```

PxVehicleDriveTank

The PhysX vehicle SDK also supports tanks through the use of the F class. Tanks are different to multi-wheeled vehicles in that the who through the differential in a way that ensures that all the wheels on the the same speed, and all the wheels on the right-hand have the same constraint on wheel speed mimics the effect of the caterpillar trac expense of simulating the jointed track structure. Adding the geometracks is as easy as adding an actor shape down each side and sett and query filters as appropriate for the tracks. The motion of the caterpi rendered with a scrolling texture, safe in the knowledge that all where speed, just as though they were properly constrained by the track rotation

Creating a PxVehicleDriveTank instance is very similar to creating a instance with the exception that tanks have no concept of extra w connected to the differential: all tank wheels are driven. The following c to set up a 12-wheeled tank:

```
PxVehicleWheelsSimData* wheelsSimData = PxVehicleWheelsSimData::a
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 different steering mechanism: the turning action of a tank arises from the and right wheel speeds, while cars turn by the action of a steering where the front wheels relative to the forward motion of the vehicle. This requires the set of helper classes and functions to smooth the control inputs:

- 1. PxVehicleDriveTankRawInputData
- 2. PxVehicleDriveTankSmoothDigitalRawInputsAndSetAnalogInputs
- 3. PxVehicleDriveTankSmoothAnalogRawInputsAndSetAnalogInputs

PhysX tanks currently support two drive models: eSTANDARD and eS

model eSPECIAL allows the tank tracks to rotate in different eSTANDARD does not. These two modes result in quite different tur model eSTANDARD simulates the usual turning action of a tank: push left(right) stick drives the left(right) wheels forward, while pulling back or applies the brake to the right(left) wheels. eSPECIAL, on the other hanc exotic turning action where pushing back on the right(left) stick drives the backwards. This can result in a turning circle focused at the center smallest possible turning circle of a tank in eSTANDARD will have a focuse of the caterpillar tracks, depending on whether the tank is turning less.

PxVehicleNoDrive

The class PxVehicleNoDrive has been introduced to provide a close backwards compatibility with the interface to the 2.8.x NxWheels essentially a rigid body with N suspension/wheel/tire units attache identical to that of a PxVehicleDrive4W which is permanently in neutrengine has no influence on the wheels and the wheels are coupled motion of the rigid body. This comes, of course, without the stackermann steering correction data, engine torque curve data etc. The can develop their own drive model on top of already existing vehicle a suspension raycasts, tire and suspension force computation, and Phys)

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

```
/**
\brief Set the brake torque to be applied to a specific wheel
*/
void setBrakeTorque(const PxU32 id, const PxReal brakeTorque);
/**
\brief Set the drive torque to be applied to a specific wheel
*/
void setDriveTorque(const PxU32 id, const PxReal driveTorque);
/**
\brief Set the steer angle to be applied to a specific wheel
*/
```

SI Units

The discussion so far has assumed that distance is measured in measured in kilograms, and that time is measured in seconds. Further of all relevant vehicle components have been set under the assumptic be adopted. An example of such a a default parameter is the maxim value. Inspection of the constructor for PxVehicleWheelData reveals a mMaxBrakeTorque. This number actually represents a value of 1500 Squared Per Second-Squared" (an alternative way of expressing this Metres"). An important question is how to set up a vehicle with mea units are not adopted. The purpose of this Section is to illustrate the particular, the case where distance is measured in centimeters rather used as an example. This particular deviation from the adoption of SI U most common one in game development, arising from the units of distance in modeling package.

Vehicle parameters whose value is dependent on the length scale fall i those that can theoretically be measured with a ruler and those with r involving combinations of other properties such as mass or time c distance. The former category includes data fields such as wheel resuspension droop, while the latter category includes data fields such as torque or wheel moment of inertia.

The following is an exhaustive list of vehicle parameters that cameasured solely from vehicle geometry:

PxVehicleChassisData::mCMOffset

PxVehicleAckermannGeometryData::mFrontWidth

PxVehicleAckermannGeometryData::mRearWidth

PxVehicleAckermannGeometryData::mAxleSeparation

PxVehicleWheelData::mRadius

PxVehicleWheelData::mWidth

PxVehicleSuspensionData::mMaxCompression

PxVehicleSuspensionData::mMaxDroop

PxVehicleWheelsSimData::setSuspForceAppPointOffset

PxVehicleWheelsSimData::setTireForceAppPointOffset

PxVehicleWheelsSimData::setWheelCentreOffset

It is useful to note that all the above parameters have default valuindependent of length scale they must always be set with measu corresponding length scale if a legal vehicle is to be successfully instantal.

Setting parameters that involve more complex combinations of length somore thought than those featured in the list above. A simple rule of parameter that has units linear with distance must be scaled by the number of length units invo distance must be scaled by the square of the number of length units that meter. A wheel braking torque of 1500 kilograms metres-squared per sexample, is equivalent to 1500*100*100 kilograms centimeters-squared. Consequently, when centimeters is used as the length scale after or wheel braking torque is 15000000 [kilograms centimeters-squared]. If inches are used as the length scale then a good initial general braking torque would be 1500*39.37*39.37 (= 2324995.35) [kilograms is second-squared].

Each non-dimensionless parameter has been described with the corres PxVehicleComponents.h. The following is an exhaustive list of vehicle $\frak x$ indirect expressions of distance scale:

PxVehicleEngineData::mMOI (kg m^2)

PxVehicleEngineData::mPeakTorque (kg m^2 s^-2)

PxVehicleEngineData::mDampingRateFullThrottle (kg m^2 s^-1)

```
PxVehicleEngineData::mDampingRateZeroThrottleClutchEngaged (kg m^PxVehicleEngineData::mDampingRateZeroThrottleClutchDisengaged (kg PxVehicleClutchData::mStrength (kg m^2 s^-1)
PxVehicleWheelData::mDampingRate (kg m^2 s^-1)
PxVehicleWheelData::mMaxBrakeTorque (kg m^2 s^-2)
PxVehicleWheelData::mMaxHandBrakeTorque (kg m^2 s^-2)
PxVehicleWheelData::mMOI (kg m^2)
PxVehicleChassisData::mMOI (kg m^2)
```

All but the last three of the above parameters have non-zero init associated constructors. This means that a good guess for their initial by multiplying the value expressed in SI Units with either the numl equivalent to 1 meter or the square of the number of length units that meter.

It is important to note that the wheel handbrake torque has a default val not all wheels respond to the handbrake torque. A good guess for the h simply the value of the wheel braking torque, perhaps multiplied by betwensure that the handbrake is stronger than the brake.

The wheel moment of inertia and chassis moment of inertia are typical the wheel radius and chassis dimensions so naturally reflect the length in the simulation. If values are taken from manufacturer data it is imposting the units of the manufacturer data are commensurate with the remaindata fields or to perform the appropriate unit conversion.

A number of functions also have parameters that are functions of following is an exhaustive list of such functions:

```
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 centimetre 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 casupplied must conform to the same unit system. Further, the default dastrictly 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 we be to decide if, say, a truck would have a stronger handbrake than the 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 pertonated that is, 5000 kilograms metres-squared per second-squared. If centimer 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 clathen performing the conversion to the chosen unit system.

The PhysX Vehicle SDK depends on a number of threshold values the length scale. These are set with the function PxInitVehicleSD

PxTolerancesScale values that have already been already configured f If PxInitVehicleSDK is not called prior to the first call to PxVehicleUpd 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 th levels of detail.

Extrapolation

The most obvious strategy for a vehicle that requires only a low level o (PxVehicleSuspensionRaycasts) performing stop raycasts (PxVehicleUpdates) for that vehicle. Instead of computing the ground vehicle's tires and computing the suspension and tire forces each and ϵ be acceptable to avoid these steps completely and let the PhysX SE body with the legacy momentum of the rigid body. After several fra wheels will likely either be hovering above the ground or intersecting t needs to be a strategy to decide how many PhysX SDK updates ca vehicle is once more updated properly by including it in the vehicle PxVehicleSuspensionRaycasts/PxVehicleUpdates. The details of any left to users of the vehicles SDK because it depends on a number distance from the camera; the spatial coherence of the world geometr the speed of the vehicle; and whether the audio or graphics fx for t important role.

Disable Wheels

If there exist vehicles with large wheel counts it might also be poss number of wheels that participate in the simulation PxVehicleWheelsSimData::disableWheel. An example might be a true Now, such a truck will clearly need to perform 18 raycasts, 18 tire for 18 updates of wheel rotation speed in order to complete the vehicle can be reduced to just 4 enabled wheels then it is clear that less cor required. It is important to note that when wheels are disabled they no leading to the control of the control o supporting the mass of the vehicle's rigid body. In the extreme case truck reduced to just 4 active wheels this will mean that the remaining e springs are only configured to support approximately 4/18 of the mass c body. To remedy this problem the mass of the rigid body will need t the enabled wheels and suspensions, among 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 w computational cost is to instantiate two versions of the vehicle with diffe The two vehicles can be easily swapped in the vehicles PxVehicleSuspensionRaycasts/PxVehicleUpdates as the required level and decreases. It is worth considering how this might work in the case truck mentioned earlier. The simplest strategy would be to first construction body and attach a PxShape instance for each of the 18 wheels of the Instantiating the required 18-wheeled version of the truck with PxVe PxVehicleNW::setup will automatically pose the shapes of all 18 whee The next step is to choose 4 of the 18 wheels to form the 4-wheeled v Many choices are available but the most obvious choice would be right/rear-left/rear-right wheels of the 18-wheeled truck. The 4-wheele be instantiated using the same rigid body as for the 18-wheeled versic of the PxShape instances to the rest pose of the 4-wheeled truck. If the wheeled version have been set up correctly the rest poses ought to k counterparts in the 18-wheeled version. A key point to note is that be vehicle apply forces to the same rigid body. Another key point to note wheeled vehicle is chosen only 4 of the 18 PxShape instances will updated, leaving 14 PxShape instances at either the rest local pose given to them when the 18-wheeled version was last used. In terms (

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 be versions of the same vehicle:

```
void PxVehicleComputeSprungMasses(const PxU32 nbSprungMasses,
    const PxVec3* sprungMassCoordinates, const PxVec3& centreOfMa
    const PxU32 gravityDirection, PxReal* sprungMasses);
void PxVehicleWheelsSimData::copy(const PxVehicleWheelsSimData& s
    const PxU32 trgWheel);
void PxVehicleSuspensionData::setMassAndPreserveNaturalFrequency(
void PxVehicleCopyDynamicsData(const PxVehicleCopyDynamicsMap& wh
    const PxVehicleWheels& src, PxVehicleWheels* trg);
```

The following pseudo-code hopefully makes clear how to apply these fifirst construct the lower LOD vehicle and then swap between the differe

```
PxVehicleDriveNW* instantiate4WVersion(const PxVehicleDriveNW& ve
    //Compute the sprung masses of the 4-wheeled version.
    PxReal sprungMasses[4];
        const PxReal rigidBodyMass = vehicle18W.getRigidDynamicAc
        const PxVec3 wheelCoords[4] =
            vehicle18W.mWheelsSimData.getWheelCentreOffset(0),
            vehicle18W.mWheelsSimData.getWheelCentreOffset(1),
            vehicle18W.mWheelsSimData.getWheelCentreOffset(2),
            vehicle18W.mWheelsSimData.getWheelCentreOffset(3)
        };
        const PxU32 upDirection = 1;
        PxVehicleComputeSprungMasses(4, wheelCoords, PxVec3(0,0,0
            sprungMasses);
    }
    //Set up the wheels simulation data.
    PxVehicleWheelsSimData* wheelsSimData4W = PxVehicleWheelsSimD
    for(PxU32 i = 0; i < 4; i++)
        wheelsSimData4W->copy(vehicle18W.mWheelsSimData, i, i);
        PxVehicleSuspensionData suspData = wheelsSimData4W->getSu
```

```
suspData.setMassAndPreserveNaturalFrequency(sprungMasses[
        wheelsSimData4W->setSuspensionData(i, suspData);
    wheelsSimData4W->setTireLoadFilterData(vehicle18W.mWheelsSimD
    //Make sure the correct shapes are posed.
   wheelsSimData4W->setWheelShapeMapping(0,0);
    wheelsSimData4W->setWheelShapeMapping(1,1);
    wheelsSimData4W->setWheelShapeMapping(2,2);
    wheelsSimData4W->setWheelShapeMapping(3,3);
    //Set up the drive simulation data.
    PxVehicleDriveSimDataNW driveSimData4W = vehicle18W.mDriveSim
    PxVehicleDifferentialNWData diff4W;
    diff4W.setDrivenWheel(0, true);
    diff4W.setDrivenWheel(1, true);
    diff4W.setDrivenWheel(2, true);
    diff4W.setDrivenWheel(3, true);
    driveSimData4W.setDiffData(diff4W);
    //Instantiate the 4-wheeled version.
    PxRigidDynamic* rigidDynamic =
        const_cast<PxRigidDynamic*>(vehicle18W.getRigidDynamicAct
    PxVehicleDriveNW* vehicle4W =
        PxVehicleDriveNW::create(&physics, rigidDynamic, *wheelsS
    //Delete the wheels simulation data now that we have copied t
    //vehicle.
   wheelsSimData4W->free();
    //Finished.
    return vehicle4W;
}
void swapToLowLodVersion(const PxVehicleDriveNW& vehicle18W, PxVe
    PxVehicleWheels** vehicles, PxU32 vehicleId)
{
   vehicles[vehicleId] = vehicle4W;
    PxVehicleCopyDynamicsMap wheelMap;
    wheelMap.sourceWheelIds[0]=0;
    wheelMap.sourceWheelIds[1]=1;
    wheelMap.sourceWheelIds[2]=2;
    wheelMap.sourceWheelIds[3]=3;
    wheelMap.targetWheelIds[0]=0;
    wheelMap.targetWheelIds[1]=1;
    wheelMap.targetWheelIds[2]=2;
```

```
wheelMap.targetWheelIds[3]=3;
    PxVehicleCopyDynamicsData(wheelMap, vehicle18W, vehicle4W);
}
void swapToHighLowVersion(const PxVehicleDriveNW& vehicle4W, PxVe
    PxVehicleWheels** vehicles, PxU32 vehicleId)
{
    vehicles[vehicleId] = vehicle18W;
    PxVehicleCopyDynamicsMap wheelMap;
    wheelMap.sourceWheelIds[0]=0;
    wheelMap.sourceWheelIds[1]=1;
    wheelMap.sourceWheelIds[2]=2;
    wheelMap.sourceWheelIds[3]=3;
    wheelMap.targetWheelIds[0]=0;
    wheelMap.targetWheelIds[1]=1;
    wheelMap.targetWheelIds[2]=2;
    wheelMap.targetWheelIds[3]=3;
    PxVehicleCopyDynamicsData(wheelMap, vehicle4W, vehicle18W);
}
```

Disable Raycasts

In some scenes it might be possible not to issue raycasts for each $v\varepsilon$ update. Depending on the geometry, this can lead to significant gains.

The PhysX vehicles SDK provides a simple mechanism to disable or ε update and per vehicle by specifying an array of booleans as a full PxVehicleSuspensionRaycasts. An alternative to disabling raycasts array would be to alter the array of vehicles passed to PxVehicleSusp that some vehicles scheduled for update in PxVehicleUpdates do no batched raycast prior to the update. It is anticipated that using the boolethe same vehicle array to be passed to both the raycast and update allowing simpler vehicle management.

Vehicles that participate in the batched raycast automatically store which are re-used each subsequent update until they are replaced by t next raycast. This means that it is not necessary to perform rayc

especially if the vehicle is moving slowly or the vehicle is far from vehicle remains on the same plane for several updates in a row. A updates preceded by a raycast decreases, the accuracy of the cach decreases, meaning that the likelihood of visibly poor wheel placem lack of accuracy in the cached hit planes means that some wheels mig or intersecting the ground if raycasts are not performed prior to each 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 partial description of its interaction with the scene. For example, as deletion the actor or shape or material hit by the last suspension rayc exist in the scene several updates later. For this reason, the vehicle rep for the shapes/actors/materials if a cached plane is used instead of fresh raycast. The documentation for PxWheelQueryResult describes the

The first update of any vehicle requires that a raycast is performed prio raycast is not performed prior to the first update then the vehicle will not an opportunity to cache its raycast hit planes. Further, after each call the vehicle also needs to perform a raycast prior to the next update. The that setToRestState clears the cached hit planes, meaning that the computed once more.

Use The Clutch in Estimate Mode

The vehicle SDK implements a mathematical model for the clutch that modes of operational eESTIMATE and **eBES** accuracy: eBEST POSSIBLE is chosen the SDK attempts to accurately update rotation speeds from their coupling through the clutch. It is worth n clutch model in PxVehicleDriveTank reduces to a particularly simple se have fast analytical solution. As a consequence, the vehicle SDK accuracy model for tanks and instead always opts to compute the bes In the case of PxVehicle4W only marginal performance gains car switching to eESTIMATE because at most only 4 wheels can ever clutch. The real performance gains from the estimated solution ar

PxVehicleNW instances with high wheel count.

lf eESTIMATE is chosen the quality of the estimate can PxVehicleClutchData::mEstimateIterations. As the value of this varia computational cost also increases and the estimated solution app possible solution. At particularly large values of mEstimateIteration estimated solution might even exceed that of the best possible solution any precision. On the other hand, particularly low values such as 1 mig inaccurate coupling between the engine and wheels. This can be par after a gear change or at standing starts or when the brakes are aggre such situations large angular velocity differences at the clutch result ir require computational effort to resolve. A poor estimate might, for oscillating engine rotation speeds after a gear change instead of the transitions. The magnitude of accuracy loss and its subsequent effect (are very difficult to quantify and really need tested for each vehicle and

It is recommended that eBEST_POSSIBLE is chosen for vehicles that of detail and that eESTIMATE is only chosen for vehicles that require locare must be taken when tuning PxVehicleClutchData::mEstimateIteral the loss of accuracy is acceptable for the required level of detail. In mar possible value of 1 will turn out to provide perfectly acceptable. Smc believable behavior, however, is only guaranteed if eBEST_POSSIBLE

Wheel Contact Beyond Raycasts

This Section describes the steps required to simulate wheel volumes sweeps and contact modification. Sample code can SnippetVehicleContactMod.

Section *Algorithm* described how scene query raycasts are used to suspension forces. Expanding on this theme, Section *Filtering* described query and simulation filtering to categorise scene shapes as either drive surfaces: drivable surfaces interact only with suspension raycasts, surfaces interact with wheels only through rigid body contact.

A variety of issues arise from the the system of raycasts and filtering One problem is that it may be impractical to author every shape in the either drivable or non-drivable: it is easy to imagine a landscape moc mesh that is partially drivable and partially non-drivable. Another problemore the extent of the wheel in the lateral and longitudinal directions. Figures 2a and 2b.

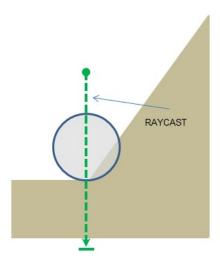


Figure 2a: The raycast ignores the overlap of the wheel's volume with plane.

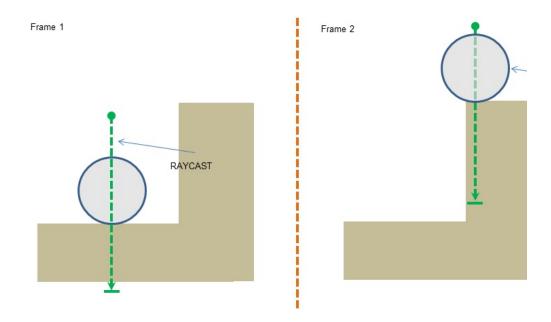
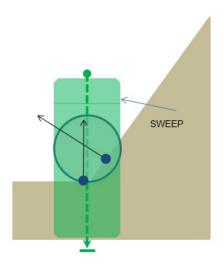


Figure 2b: The wheel rolls towards a wall in Frame 1 and is immediatel elevated surface in Frame 2.

The problem illustrated in Figure 2a can be solved by replacing rays Instead of performing a raycast along the suspension direction throug wheel, the shape representing the wheel is swept from its trans compression to its transform at maximum elongation. Sweeping a v scene means that all possible contact planes are considered. This is illustrated in Figure 2a can be solved by replacing rays Instead of performing a raycast along the suspension direction throug wheel, the shape representing the wheel is swept from its transform at maximum elongation.



In Figure 3, it is easy to see that there are multiple contact points und with a different normal. A decision needs to be made about which or accept as the driving surface and which to ignore. In some scenarios it take the first contact encountered by the sweep and ignore all others. It recommended to issue a blocking sweep. PhysX supports two type blocking and non-blocking. A detailed description of blocking and non-blocking and non-blocking in Section *Filtering*. In summary, however, a blocking sweep contact encountered by the sweept volume, while non-blocking sweeps encountered by the sweep. The scenario in Figure 3 suggests that a libe sufficient because it will return the inclined plane rather than the hole consequence, the vehicle will start to drive on the inclined plane. Some those depicted in Figure 2b, are more complex and require a non-blocking

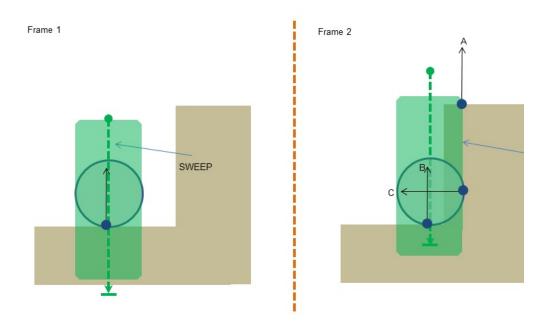


Figure 4: Judicious selection of sweep contacts and rigid body contacts and rigid body contacts are a wheel through a complex scene.

Figure 4 shows a wheel rolling along a horizontal plane towards a expected behavior is that the wheel continues to drive on the horiz blocked by the vertical plane. It turns out that this can be readily ack choice of sweep contacts and rigid body contacts. The first thing to not

will return the three contact planes labelled A, B and C in Figure 4. I rigid body contact between the wheel and the environment we will si contact planes B and C as rigid body contacts. The next step is to devaccepts contact plane B for the sweep and contact plane C for rigid combination will ensure that the wheel bounces off the vertical plane drive on the lower horizontal plane. The strategy adopted by Phy categorise sweep and rigid body contacts by comparing contact norm the suspension direction. The aim is to divide contact with the environ contact planes and non-drivable contact planes. This can be achieved threshold angles to categories contact points and normals.

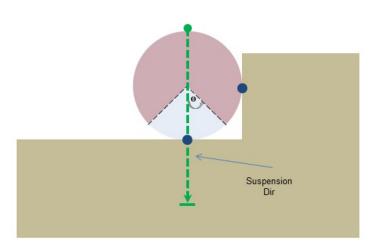


Figure 5: The position of sweep and rigid body contact points relative direction is used to filter the sweep and rigid body contacts. Sweep co blue zone are accepted as driving planes, while rigid body contacts in accepted as rigid body contact planes.

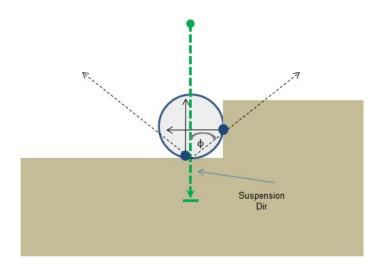


Figure 6: The angle between contact normal and the suspension directed categorised contact planes as either rigid body contacts or sweep conormals close to the suspension direction are accepted as driving planes far from the suspension direction are accepted as rigid body con

Figures 5 and 6 introduced two threshold angles that together allow sw contacts to be categorised using their position and normal. Having a drivable and non-drivable contact points and normals allows a relafiltering rules described in Section *Filtering*. The idea now is to set up s so that wheel shapes sweep against and collide with pretty much every The two threshold angles will filter and categorise sweep and rigic generate the desired behavior.

The threshold angles shown in Figure 5 and Figure 6 are configured function call:

```
void PxVehicleSetSweepHitRejectionAngles(const PxF32 pointRejectA
```

The code snippet SnippetVehicleContactMod demonstrates how to con non-blocking sweeps. This snippet can be configured to run with eithe modifying the BLOCKING_SWEEPS define. Running the

BLOCKING_SWEEPS demonstates that the situation depicted in Figure blocking sweeps to ensure that the elevated horizontal plane is not characteristic.

Suspension sweeps are issued with the following code:

```
//Suspension sweeps (instead of raycasts).
//Sweeps provide more information about the geometry under the wh
PxVehicleWheels* vehicles[NUM_VEHICLES] = {gVehicle4W[0], gVehicl
PxSweepQueryResult* sweepResults = gVehicleSceneQueryData->getSwe
const PxU32 sweepResultsSize = gVehicleSceneQueryData->getQueryRe
PxVehicleSuspensionSweeps(gBatchQuery, NUM_VEHICLES, vehicles, sw
```

In the event that non-blocking sweeps are implemented, the function accepts hits threshold rejects and sweep usina the PxVehicleSetSweepHitRejectionAngles. When blocking sweeps are ir single sweep contact is recorded. As a consequence, PxVehicleUr threshold angles and automatically works with the blocking sweep whether to use blocking or non-blocking sweeps is left to the dev depends on knowledge about the kinds of geometry that will be e vehicle. In some applications it will be sufficient to opt for the comp option of blocking sweeps, while other applications may expect the complex geometry and are prepared to accept the extra cost of non-blo-

Categorisation of rigid body contacts is implemented using contact mod Contact modification is described in Section *Contact Modification*. TI SDK provides the function PxVehicleModifyWheelContacts to accep points using the defined threshold angles. This function should be calk modification callback, which is owned by the application. Config modification callbacks involves a combination of simulation filter dashader. The implementation details, therefore, are left to applic SnippetVehicleContactMod illustrates one way to implement a co-callback using simulation filter data and the userdata pointers PxRigidDynamic. Other techniques are available using local knowledge In addition to adding sweeps and contact modification, the snippet also collision detection (CCD) to the wheel shapes. CCD is introduced in §

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 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 eith above the ground or intersecting the ground. Ideally, this parameter w from the 3D modeler.

mWidth:

This is the full width of the wheel in metres. This parameter has no be handling but is a very useful parameter to have when trying to render relating to the wheel/tire/suspension. Without this parameter it would compute coordinates for render points and lines that ensure their visil 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 dependin

mMOI:

This is the component of the wheel's moment of inertia about the ro values make it harder for the wheel to rotate about this axis, while lor it easier for the wheel to rotate about the rolling axis. Another way c is that a high MOI will result in less wheel spin when stamping or because it is harder to make the wheel spin. Conversely, lower variesult in more wheel spin when stamping on the accelerator.

If the wheel is approximately cylindrical then a simple formula compute MOI:

MOI = 0.5 * Mass * Radius * Radius

There is no reason, however, to rely on equations to compute thi strategy for tuning this number might to be start with the equation make small tweaks to the value until the handling is as desired.

mDampingRate:

This value describes how quickly a freely spinning wheel will come to damping rate describes the rate at which a freely spinning wheel lose speed. Here, a freely spinning wheel is one that experiences no force damping forces arising from the wheel's internal bearings. Higher dan result in the wheel coming to rest in shorter times, while lower dampir the wheel maintaining speed for longer. Values in range (0.25, 2) see values. Experimentation is always a good idea, even outside this range exercise some caution with very small damping rates. In particular, a exactly 0 should be avoided.

mMaxBrakeTorque:

This is the value of the torque applied to the wheel when the brake applied. Higher torques will lock the wheel quicker when braking, wh will take longer to lock the wheel. This value is strongly related to because the MOI determines how quickly the wheel will react to appli

A value of around 1500 is a good starting point for a vanilla wheel t will reveal typical braking torques. One difficulty is that these are oft manufacturers as braking horsepower or in "pounds inches". The here are in "Newton metres".

mMaxHandBrakeTorque:

This is the same as the max brake torque except for the handbrake rabrake. Typically, for a 4-wheeled car, the handbrake is stronger than t

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 respond this case, a value of 0 is required for the rear wheels. More exotic comight wish front and rear wheels to respond to steering. A value in racto somewhere between 30 degrees and 90 degrees seems like a goo but it really depends on the vehicle being simulated. Larger values of result in tighter turns, while smaller values will result in wider turns. But that large steer angles at large speeds are likely to result in the car lospinning out of control, just as would happen with a real car. A good wis to filter the steer angles passed to the car at run-time to generate s angles at larger speeds. This strategy will simulate the difficulty of act steer angles at high speeds (at high speeds the wheels resist the turn applied by the steering wheel).

mToeAngle:

This is the angle of the wheel (in radians) that occurs with no steer angle can be used to help the car straighten up after coming out of good number to experiment with but is best left at 0 unless deta 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 where the positive angles, and which the negative, it is straightforward to reither "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" configuration the opposite effect. Toe angles greater than a few degrees are best at

PxVehicleWheelsSimData

void setSuspTravelDirection(const PxU32 id, const PxVec3& dir):

This is the direction of the suspension in the downward direction in the configuration of the vehicle. A vector that points straight downwards is point.

void setSuspForceAppPointOffset(const PxU32 id, const PxVec3& offset

This is the application point of the suspension force, expressed as from the center of mass of the vehicle's rigid body. Another way of ϵ to start at the center of mass of the rigid body, then move along the ϵ point at the end off the offset vector is the point at which suspensiapplied.

In a real vehicle the suspension forces are mediated through the second through the second through the second through the second through the suspensive to simulate. As a consequence, instead of modeling the suspension strut, it makes sense to assume that the suspension strut point at which it applies the force to the rigid body. Choosing that needs careful consideration. At the same time, it opens up all second possibilities, freed from the constraints of the real world.

Deciding on the suspension force application point requires son suspension is very close to the wheel so the wheel center is a gor Consider a line through the wheel center and along the suspensior Somewhere along this line seems like an even better idea for the albeit not completely scientific. For a standard 4-wheeled car it make application point is somewhere above the wheel center but below the of the rigid body. It is probably above the wheel center because the mostly above this point. It can be assumed that it is somewhere belocenter of mass because otherwise vehicles would lean out of the turn the turn. This narrows down the application point to really quite a sknown line.

When editing the suspension force application point it is important to lowering the app point too far will result in cars leaning more into the

have a negative effect on handling because the inner wheel can tal that the response saturates, while the outer wheel ends up with rereduced turning force. The result is poor cornering. Conversely, setti too high will result in cornering that looks unnatural. The aim is to balance.

void setTireForceAppPointOffset(const PxU32 id, const PxVec3& offset)

This is almost the same as the suspension force app point except for longitudinal forces that develop on the tire. A good starting point is to suspension force application point. Only for really detailed editing is it tweaking the tire force app offset independently of the suspension for

void setWheelCentreOffset(const PxU32 id, const PxVec3& offset):

This is the center of the wheel at rest position, expressed as an offset 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 for typically be equally supported by each of the suspension spring suspension spring supports 1/4 of the total vehicle mass. If the cen moved forward then it would be expected that the front wheels would more mass than the rear wheels. Conversely, a center of mass neare ought to result in the rear suspension springs supporting more mass t

Note: In order to achieve stability at the desired rest pose it is received that the collection of sprung masses matches the mass and center the rigid body. There are two strategies that can be employed to at The first approach is to decide upon values for the individual sprur and work forwards to compute an equivalent value for the rigid both

center of mass. More specifically, the rigid body mass and center of be computed using the equations presented in Section *Algorithm &* applied to the vehicle's PxRigidDynamic instance. The second applied to the rigid body mass and center of mass of the vehicle's PxRiginstance and works backwards to compute and set the sprung manakes use of the function PxVehicleComputeSprungMasses that introduced in Section *setupWheelsSimulationData*.

mMaxCompression:

mMaxDroop:

These values describe the maximum compression and elongation ir spring can support. The total travel distance along the spring directic is the sum of mMaxCompression and mMaxDroop.

A simple way to illustrate the maximum droop and compression valu a car that is suspended in mid-air so that none of the wheels are touc The wheels will naturally fall downwards from their rest position ur droop is reached. The spring cannot be elongated beyond this poir that the wheel is pushed upward, first to its rest position, then further spring can no longer be compressed. The displacement from the re maximum compression of the spring.

It is important to choose the maximum compression value so that the placed where the visual mesh of the wheel intersects the v

mSpringStrength:

This value describes the strength of the suspension spring. The sprir profound influence on handling by modulating the time it takes for respond to bumps in the road and on the amount of load experienced

Key to understanding the effect of spring strength is the concept of a frequency. Consider a simple spring system, such as a pendulum sv

forth. The number of trips per second that the pendulum makes from right and then back again is called the natural frequency of the perpowerful pendulum spring will result in the pendulum swinging increasing the natural frequency. Conversely, increasing the pendulum result in a slower oscillation, thereby reducing the natural frequency.

In the context of a suspension spring supporting a fixed portion of v strength of the spring will affect the natural frequency; that is, the r spring can respond to changes in load distribution. Consider a car ta the car corners it leans in to the turn, putting more weight on the sus outside of the turn. The speed at which the spring reacts by as redistribute the load is controlled by the natural frequency. Ve frequencies, such as those on a racing car, will naturally produce because the load on the tires, and therefore the forces they can ger very rapidly. Very low natural frequencies, on the other hand, will taking a long time to straighten up even after the turn is complete. sluggish and unresponsive handling.

Another effect of strength and and natural frequency is the respon bump in the road. High natural frequencies can result in the car strongly and quickly to the bump, with the wheel possibly even leavishort while. This not only creates a bumpy ride but also periods of ti 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 the expected types of turn and terrain.

The natural frequency of the spring presents a challenge for compusmooth and stable simulation requires that the spring is updated at a greater than the spring's natural frequency. An alternative way of expression of the spring is the spring relative to the timestep of the simulation of the spring is the time the spring takes to complete a single of mathematically equal to the reciprocal of the natural frequency. In or stable simulation the spring must be sampled at several points during A natural consequence of this observation is that the simulation tis significantly smaller than the period of the spring. To discuss this furtly

introduce a ratio that describes the number of simulation updated during each spring oscillation. This ratio is simply the spring periodimestep

alpha = sqrt(mSprungMass/mSpringStrength)/timestep

where sqrt(mSprungMass/mSpringStrength) is the period of the s value of 1.0 means that the chosen timestep and spring properties or sample of the spring during each oscillation. As described above guaranteed to produce unstable behavior. In fact, the argument progression suggests a value of alpha significantly greater than 1.0 is essent smooth simulation. The exact value of alpha at which stability emerges to predict and depends on many other parameters. As a guide recommended that the timestep and spring properties are chosen produce an alpha value greater than 5.0; that is, a minimum of five sin per spring cycle.

When tuning a suspension spring it can be very useful to use man discover typical values used across a range of vehicle types. This discover typical values used across a range of vehicle types. This discover typical values used across a range of vehicle types. This discover typical values used across a range of vehicle types. This discover typical values used across a range of vehicle types. This discover typical values used across a range of vehicle types. This discover typical values used across a range of vehicle types. This discover typical values used across a range of vehicle types. This discover typical values used across a range of vehicle types. This discover typical values used across a range of vehicle types. This discover typical values used across a range of vehicle types. This discover types used across a range of vehicle types. This discover typical values used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types. This discover types used across a range of vehicle types used across a range of vehicle types. This discover types used across a range of

mSpringStrength = naturalFrequency * naturalFrequency * mSprun

Note: To achieve a spring that is theoretically correct, the values mSprungMass, mSpringStrength and mMaxDroop should be chos they obey the equation mSpringStrength*mMaxDroop = mSprungMass*gravitationalAcceleration. When this equation is sa spring is guaranteed to provide exactly zero force at maximum elo also to support the sprung mass at the rest pose (the rest pose is

PxVehicleWheelsSimDta::setWheelCentreOffset). It is often the cathat the visual requirements of the car are in conflict with its handli requirements. An example might be a visual requirement, imposed on both the rest pose and the suspension travel limits. In order to susual requirement and achieve a theoretically correct spring, the visual regularity must be equivalent to

mSprungMass*gravitationalAcceleration/mMaxDroop. If this value mSpringStrength does not meet the handling requirements of the gas there is a conflict that cannot be easily resolved. For this reason, the Vehicles module does not require the spring to be a theoretically particle that the spring eithe providing upward force before it hits maximum elongation or that it a non-zero force at maximum elongation. The effect on handling of visual appearance of the vehicle is often quite difficult to spot. In particle to spot the vehicle is often quite difficult to spot the disguises any imperfection.

mSpringDamperRate:

This describes the rate at which the spring dissipates the energy store

Key to the understanding of damper rate are the concepts of unde damping, and critical damping. An over-damped pendulum displar unable to make a single back-and-forth trip before it dissipates all its under-damped pendulum would be able to make at least a single back-and-expending all its energy.

For vehicle suspension springs, it is typically important to make sur 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 sl from the left suspension to the right suspension then back again. If the tire load, and the forces generated, would be extremely variable, re and uncontrollable handling. A very heavily over-damped spring, on will feel sluggish and unresponsive.

The concept of critical damping can be used to help tune the dan spring. It is helpful to introduce a value known as the damping ration mathematically describe the under-damping, critical damping an regimes.

dampingRatio = mSpringDamperRate/[2 * sqrt(mSpringStrength *
mSprungMass)]

A dampingRatio with value greater than 1.0 produces over-damperactly 1.0 generates critical damping, and a value less than 1.0 is a 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 a number to be subjectively applied to the damping ratio. From here can be directly computed by rearranging the equation above

mSpringDamperRate = dampingRatio * 2 * sqrt(mSpringStrength * 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 are likely to be unrealistic and will either produce sluggish or twitch difficult to put an exact figure on this but somewhere between 0.8 an a good starting point for the damping ratio.

mCamberAtRest:

mCamberAtMaxCompression:

mCamberAtMaxDroop:

These values describe the camber angle of the wheels as a functic spring compression. It is typical for the wheels of extended springs to that is, the left and right wheels almost seem to form the edges of viewed from the front or rear along the forward axis of the vehic springs, on the other hand, typically camber outwards; that is, they outer edges of an A shape when when viewed from the front or rear a

axis of the vehicle.

These three values allow the camber angle to be computed for any compression using simple linear interpolation. At rest, when the elongated or compressed, the camber angle is equal to mCamberA spring is compressed the camber is computed as a linear interpolation. When the spring camber is computed as a linear interpolation between mCarmCamberAtMaxDroop.

The camber angle is used by the default tire model and is passe argument to the tire shader. It is also used to set the local pose of t geometrically represents the wheel.

PxVehicleAntiRollBar

When a vehicle takes a corner the turning force causes the car to r suspension springs on the outside of the turn are compressed while springs on the inside of the turn are elongated. If the roll is so sever wheels completely leave the ground then there is a danger that the control of the vehicle. In such cases, there is even a danger that the onto its side. For less severe rolls there still remains a handling profrom the distribution of load between the inside and outside tires. T that the imbalance of the vehicle can lead to under-steer or over-stee

Anti-roll bars are commonly used to reduce the roll that natura cornering. They typically work as a torsion spring that applies a tominimise the difference in spring displacement for a pair of wheels. A car might feature a front and rear anti-roll bar. The front bar applies a the difference between the front-left and front-right wheels. Simila applies a torque to reduce the difference between the rear-left and rear-left and rear-left and rear-left.

The magnitude of the anti-roll torque is proportional to the difference displacement of the two wheels that are connected by the bar. The proportional to a stiffness parameter: stiffer bars generate more anti-r

As a general rule, under-steer can be reduced by increasing the stif anti-roll bar. Increasing the stiffness of the front anti-roll bar typical 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. It the computation independently computes the lateral and longitudina the force using linear equations. These independent forces are computed tire as a linear system so that the force in each direction can viewed as the product of a tire strength per unit slip and the slippage 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 the tires are able to resist greater horizontal forces when they experient load on a surface with high friction value. With this in mind the max force for a tire can be approximated as the product of the normal load value. The default PhysX Vehicle tire model employs a series of smitted to implement the normalization of the combined tire forces.

In addition to the lateral and longitudinal components of force a can arising from the camber angle of the tire, is also computed. Typica only a small correction to the effect of the lateral and longitudinal c camber force participates in the normalization process.

The following tire parameters describe the computation of the indepe longitudinal and camber components; that is, the first conceptual st computation. Reference is made throughout to the handling const normalization process.

mLongitudinalStiffnessPerUnitGravity:

The longitudinal stiffness describes the longitudinal forces that develongitudinal slip (in radians). Here, a variable that represents the long per unit gravity has been introduced in order to make the variable roledits to the value of gravitational acceleration. The longitudinal approximately the product of the longitudinal stiffness per unit longitudinal slip and the magnitude of gravitational acceleration:

longitudinalTireForce = mLongitudinalStiffnessPerUnitGravity * lonç gravity;

Increasing this value will result in the tire attempting to generate r force when the tire is slipping. Typically, increasing longitudinal stiffr car accelerate and brake. The total tire force available is limited by th so be aware that increases in this value might have no effect or expense of reduced lateral force.

mLatStiffX:

mLatStiffY:

These values together describe the lateral stiffness per unit lateral s the tire. The lateral stiffness of a tire has a role similar to the long (mLongitudinalStiffnessPerUnitGravity), except that it governs the lateral tire forces, and is a function of tire load. Typically, increasing will help the car turn more quickly. The total tire force available is lir on the tire so be aware that increases in this value might have no effect the expense of reduced longitudinal force.

Lateral stiffness is a little more complicated than longitudinal stiffne typically provide poor response under heavy load. Typical for car till lateral force against load that has linear response close to zero load greater loads. This means that at low tire loads the lateral stiffness.

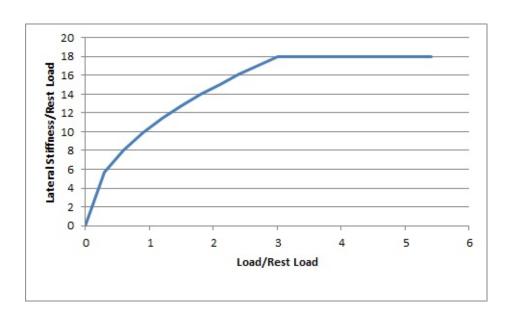
response to load; that is, more load results in more stiffness and more force. At higher tire loads the tire has a saturated response and is in applying more load will not result in more tire stiffness. In this latter reexpected that the tire would start slipping.

The combination of two values mLatStiffX and mLatStiffY describe a stiffness per unit load as a function of normalized tire load. The tire for employs a smoothing function which requires knowledge of the normal which the tire has a saturated response to tire load along with the lat unit load that occurs at this saturation point. A typical curve can be shelow.

The parameter mLatStiffX describes the normalized tire load above v a saturated response to tire load. The normalized tire load is sim divided by the load that is experienced when the vehicle is perfectly ϵ 2 for mLatStiffX, for example, means that when the tire has a twice its rest load it can deliver no more lateral stiffness no matter load is applied to the tire. In the graph below mLatStiffX has value 3.

The parameter mLatStiffY describes the maximum stiffness per unit radians) per unit rest load. The maximum stiffness is delivered when saturated load regime, governed in turn by mLatStiffX. In the graph has value 18.

The computation of the lateral stiffness begins by computing the loa then computing the normalized load in order to compute the number experienced by the tire. This places the tire somewhere along the X-below. The corresponding value on the Y-axis of the curve parallel stiff and mLatStiffY is queried to provide the lateral stiffness parallel than the lateral stiffness is then computed by multiples graph value by the rest load. This final value describes the lateral 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 lo Vehicle tires have a complicated response to longitudinal slil attempts to approximate this relationship.

Typically, tires have a linear response at small slips. This means tire is only slightly slipping it is able to generate a response force the slip increases. At greater values of slip, the force can ac decrease from the peak value that occurs at the optimum slip optimum slip the tire eventually starts behaving less and less efficie plateau of inefficiency.

The friction value for the combination of surface type and tire type been discussed in Section *Tire Friction on Drivable Surfaces*. friction versus longitudinal slip is used as a correction to the combination friction value and the graph's correction value. The tresponds to the final friction value.

The first two values describe the friction at zero tire slip: mFriction\[0] = 0, and mFriction\vsSlipGraph[0][1] = friction at zero slip.

The next two values describe the optimum slip and the friction a slip: mFrictionVsSlipGraph[1][0] = optimum slip, mFrictionVsSlip friction at optimum slip.

The last two values describe the slip at which the plateau of inefl and the value of the friction available at the plateau omFrictionVsSlipGraph[2][0] = slip at the start of the plateau omFrictionVsSlipGraph[2][1] = the friction available at the plateau of

In the graph below the following values have been used:

mFrictionVsSlipGraph[0][0] = 0.0

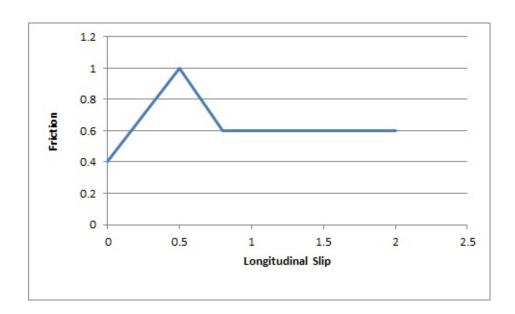
mFrictionVsSlipGraph[0][1] = 0.4

mFrictionVsSlipGraph[1][0] = 0.5

mFrictionVsSlipGraph[1][1] = 1.0

mFrictionVsSlipGraph[2][0] = 0.75

mFrictionVsSlipGraph[2][1] = 0.60



The friction values described here are used to scale the friction of the This means they should be in range (0,1) but this is not a strict requir the friction from the graph would be close to 1.0 in order to provide a to the ground surface friction.

A good starting point for this is a flat graph of friction vs slip with these

mFrictionVsSlipGraph[0][0]=0.0

mFrictionVsSlipGraph[0][1]=1.0

mFrictionVsSlipGraph[1][0]=0.5

mFrictionVsSlipGraph[1][1]=1.0

mFrictionVsSlipGraph[2][0]=1.0

mFrictionVsSlipGraph[2][1]=1.0

mCamberStiffnessPerUnitGravity:

The camber stiffness is analogous to the longitudinal and lateral stiffi it describes the camber thrust force arising per unit camber angle (in to the longitudinal stiffness, a camber stiffness per unit gravity has be

make the camber stiffness robust across different values of gravitatic. The independent camber force is computed as the camber angle camber stiffness multiplied by the gravitational acceleration:

camberTireForce = mCamberStiffnessPerUnitGravity * camberAng mType:

This parameter has been explained in Section *Tire Friction on Drivable*

PxVehicleEngineData

mMOI:

This the moment of inertia of the engine around the axis of rotation. L make it harder to accelerate the engine, while lower values make it eaccelerate 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. Th in Newton metres. A starting value might be around 600.

mMaxOmega:

This is the maximum rotational speed of the engine expressed in radi mDampingRateFullThrottle:

mDampingRateZeroThrottleClutchEngaged:

mDampingRateZeroThrottleClutchDisengaged:

These three values are used to compute the damping rate that i engine. If the clutch is engaged then the damping rate is an interpolation makes and makes and makes are used to compute the damping rate that i engine. If the clutch is engaged then the damping rate is an interpolation and makes are used to compute the damping rate that i engine makes are used to compute the damping rate that i engine makes are used to compute the damping rate that i engine makes are used to compute the damping rate that i engine makes are used to compute the damping rate that i engine makes are used to compute the damping rate that i engine makes are used to compute the damping rate is an interpolation of the damping rate is an interpol

gamepad or keyboard. At full throttle mDampingRateFullThrottle imDampingRateZeroThrottleClutchEngaged is applied at zero throttle the damping rate is an interpolation between mDampingRate mDampingRateZeroThrottleClutchDisengaged.

The three values allow a range of effects to be generated: good acce hampered by strong damping forces, tunable damping forces whe neutral gear during a gear change, and strong damping forces 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 unstakrates of 0.

mTorqueCurve:

This is a graph of peak torque versus engine rotational speed. Cars range of engine speeds that produce good drive torques, and other speed that produce poor torques. A skilled driver will make good us ensure that the car remains in the "good" range where the engine is Tuning this graph can have profound effects on gameplay.

The x-axis of the curve is the normalized engine speed; that is, the divided by the maximum engine speed. The y-axis of the curve is a n (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 r 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 t

top speed in that gear. Typically, the higher the gear, the lower the ge gear must always be given a value of 0, while reverse gear must have gear ratio. Typical values might be 4 for first gear and 1.1 for fifth gea

mFinalRatio:

The gear ratio used in the simulator is the gear ratio of the current gether the final ratio. The final ratio is a quick and rough way of changing the without having to edit each individual entry. Further, quoted gearing vermanufacturers typically mention ratios for each gear along with a final value might be around 4.

mSwitchTime:

The switch time describes how long it takes (in seconds) for a ge completed. It is impossible to change gear immediately in a real ca for example, require neutral to be engaged for a short time befo desired target gear. While the gear change is being completed the neutral. A good trick might be to penalize players that use an autom increasing the gear switch time.

If the autobox is enabled it is a good idea to set this value signific PxVehicleAutoBoxData::setLatency. If the autobox latency is smaller switch time then the autobox might decide to initiate a downwar immediately after an upward gear shift has been completed. This sit the car cycling between neutral and first gear with very short interlude

PxVehicleAutoBoxData

The autobox initiates gear changes up or down based on the rotat engine. If the engine is rotating faster than a threshold \times PxVehicleAutoBoxData then a gear increment will be initiated. On t the engine is rotating slower than a threshold value then the autob gear decrement. The autobox only initiates gear changes upward single gear at a time.

It is worth noting that if the autobox initiates a gear change then the a is automatically disconnected from the engine for the entire dura change. Manual gear changes (PxVehicleDriveDynData::star PxVehicleDriveDynData::mGearUpPressed

PxVehicleDriveDynData::mGearDownPressed) are not subject to thi is in keeping with typical real-world autobox behavior. The idea behi the engine wildly accelerating during the neutral phase of the gear avoiding damaging clutch slip when the clutch re-engages at the change.

The autobox will not try to initiate a gear change while an automatic change is still active.

too simplistic for the application's If the autobox İS PxVehicleGearsData can be readily disabled. The choices following revert to a manual gear model or to implement a custom autobox in t transition to specific a gear can PxVehicleDriveDynData::startGearChange, while single gear change with PxVehicleDriveDynData::mGearUpPressed PxVehicleDriveDynData::mGearDownPressed.

The autobox can be enabled or disabled PxVehicleDriveDynData::mUseAutoGears.

PxReal mUpRatios[PxVehicleGearsData::eGEARSRATIO_COUNT]:

The autobox will initiate a gear increment if the ratio of the engine in the maximum allowed engine rotation speed:

PxVehicleDriveDynData::getEngineRotationSpeed() / PxVehicleEngineData::mMaxOmega

is greater than the value smUpRatios[PxVehicleDriveDynData::getCurrentGear()]

PxReal mDownRatios[PxVehicleGearsData::eGEARSRATIO_COUNT]:

The autobox will initiate a gear decrement if the ratio of the engine the maximum allowed engine rotation speed:

PxVehicleDriveDynData::getEngineRotationSpeed() / PxVehicleEngineData::mMaxOmega

is less than the value stored in mUpRatios[PxVehicleDriveDynData::g

void setLatency(const PxReal latency):

After the autobox has initiated a gear change it will not attempt to initi change until the latency time has passed. It is a good idea to set this significantly higher than PxVehicleGearsData::mSwitchTime. If the lat than the gear switch time then the autobox might decide to initiate a c change immediately after an upward gear shift has been completed. can leave the car cycling between neutral and first gear with very sho 2nd gear.

PxVehicleClutchData

mStrength:

This describes how strongly the clutch couples the engine to the quickly differences in speed are eliminated by distributing torque to wheels.

Weaker values will result in more clutch slip, especially after cl stamping on the accelerator. Stronger values will result in reduced more engine torque delivered to the wheels.

This value is to be edited only for very fine tweaking of the vehicle. can be attributed to the numerical issues in the simulation at large some is a natural consequence of driving the car in an overly aggre value of 10 is a good starting point.

PxVehicleAckermannGeometryData

mAccuracy:

Ackermann correction allows better cornering by steering the left and with slightly different steer angles, as computed from simple trigonom it is impossible to engineer a steering linkage that will achieve the per steering correction. This value allows the accuracy of the Ackermann correction to be controlled. Choosing a value of 0 completely disables steer correction. A value of 1.0, on the other hand, achieves the impo perfect Ackermann correction.

mFrontWidth:

This is the distance in metres between the two front wheels.

mRearWidth:

This is the distance in metres between the two rear wheels.

mAxleSeparation:

This is the distance in metres between the center of the front axle and the rear axle.

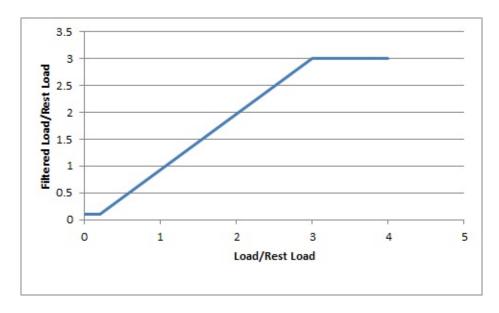
PxVehicleTireLoadFilterData

This is for very fine control of the handling, and corrects numerical is simulations at large timesteps.

At large simulation timesteps the amplitude of motion of the susper larger than it would be in real-life. This is unfortunately unavoidable surface this could mean that the simulation lifts the car further from would really happen. This could be quickly followed by the spicompressed than would be experienced with a real vehicle. A consoscillation is that the load on the tire is more variable than expected, the forces have more variability than expected. This filter aims numerical problem by smoothing the tire load with the aim of make smoother and more predictable.

A key concept is that of normalized tire loads. A normalized tire load load divided by the load experienced when the vehicle is in its rest of tire experiences more load than it does at rest then it has a normalized tire load less than 1.0. At rest, all tires obviously have a 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 filtere raw tire loads. The x-axis of the graph is "normalized tire load", while normalized load". "filtered tire Normalized graph mMinNormalisedLoad filtered produce normalized a mMinFilteredNormalisedLoad. Normalized loads greater than mMax produce a filtered normalized load of mMaxFilteredNormalisedL between mMinNormalisedLoad and mMaxNormalisedLoad normalized load in-between mMinFilteredNormalise mMaxFilteredNormalisedLoad, as computed by direct interpolation.



Choosing mMaxNormalisedLoad and mMaxFilteredNormalisedLomaximum load that will ever be used in the simulation. On the choosing mMinFilteredNormalisedLoad>0 and/or mMinNormalised the tire to potentially generate a non-zero tire force even when touching the ground at maximum droop.

The filtered load can be made identical to the computed tire load by

mMinNormalisedLoad=mMaxFilteredNormalisedLoad=0 mMaxNormalisedLoad=mMaxFilteredNormalisedLoad=1000.

Note: Tires may only generate forces if the tire is touching the greatire cannot be placed on the ground then the tire force is always of magnitude. A tire touching the ground at maximum suspension droother hand, has zero measured load because the spring generates at maximum droop. By editing PxVehicleTireLoadFilterData it is pospenerate tire forces even when there is very little load actually actitire.

PxVehicleDifferential4WData

mType:

A number of differential types are supported: 4-wheel drive with open wheel drive with limited slip, front-wheel drive with open differential, fr with limited slip, rear-wheel drive with open differential, rear-wheel dri slip.

mFrontRearSplit:

If a 4-wheel drive differential is chosen (open or limited slip) this optio drive torque to be split unevenly between the front and rear wheels. C of 0.5 delivers an equal split of the torque between the front and rear the total torque delivered to the front wheels is equal to the total torque the rear wheels. Choosing a value greater than 0.5 delivers more torce wheels, while choosing a value less than 0.5 delivers more torque to This value is ignored for front-wheel drive and rear-wheel drive differe

mFrontLeftRightSplit:

This is similar to the Front Rear Split but instead splits the torque that 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 0.1 torque to the front-right wheel. This parameter can be used to preven being delivered to a damaged or disabled wheel. This value is ignored drive.

mRearLeftRightSplit:

This is similar to mFrontLeftRightSplit except that it applies to the real of the front wheels. This value is ignored for front-wheel drive.

mFrontBias:

Limited slip differentials work by only allowing a certain difference speed to accumulate. This prevents the situation where one whee ends up taking all the available power. Further, by allowing a small directation speed to accumulate it is possible for the vehicle to expermitting the outside wheel to rotate quicker than the inside wheel.

This parameter describes the maximum difference in wheel rotatic allowed to accumulate. The front bias is the maximum of the two fro speeds divided by the minimum of the two front-wheel rotation sp ratio exceeds the value of the front bias the differential diverts torqu wheel to the slower wheel in an attempt to preserve the maximur rotation speed ratio.

This value is ignored except for front-wheel drive or four wheel drive ι

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 w

A good starting value is around 1.3.

mCentreBias:

This value is similar to the mFrontBias and mRearBias, except the sum of the front wheel rotation speeds and the sum of the rear wheel

This value is ignored except for four wheel drive with limited slip.

A good starting value is around 1.3.

PxRigidDynamic

Moment of Inertia:

The moment of inertia of the rigid body is an extremely important editing vehicles because it affects the turning and rolling of the vehicle

A good starting point for the moment of inertia of the rigid body is moment of inertia of the cuboid that bounds the chassis geometry. cuboid is W wide, H high, and L long then the moment of inertia for ϵ M is:

```
((L*L+H*H)*M/12, (W*W+L*L)*M/12, (H*H+W*W)*M/12)
```

However, this is only a rough guide. Tweaking each value will m around the corresponding axis, with higher values making it h rotational speed from tire and suspension forces.

Providing unphysical values for the moment of inertia will result in eitle behavior or extremely twitchy and perhaps even unstable behavior. inertia must at least approximately reflect the length scales of the sufforce application points.

This parameter should be viewed as one of the first go-to editable val

Center of mass:

Along with the moment of inertia, the center of mass is one of the fill

values and, as such, has a profound effect on handling.

To discuss the center of mass it is useful to consider a typical 4-whe a chassis mesh whose origin is at the center of the four whee requirement on the origin being at the center of the four wheels but following discussion a little simpler. It might be expected that the ce somewhere near this origin because vehicles are designed in a way load almost evenly between the four wheels. More specifically, it may that the center of mass needs to be a little above the base of the ch at the height of the wheels. After all, vehicles have higher mass bottom of the chassis due to density of the engine and other mechan a consequence, it is expected that the center of mass is nearer th chassis than the top, but definitely above the bottom. Without a par analysis of the chassis density distribution the exact location along the really a little arbitrary and subjective. Along the forward direction it m that the center of mass is a little nearer the front wheels than the rear of the mass of the front-located engine. Thinking about these factors of mass to be tweaked along the vertical and forward directions.

Tweaking the center of mass is really all about making incremental c the handling towards a desired goal. Moving the center of mass forw cornering because more load is distributed to the front tires. However the expense of reduced load on the rear tires, meaning that the car quickly only to spin out because the rear tires lose grip more quickly 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 moves nearer the front this means that more mass is support suspensions and less by the rear suspensions. This change needs to consistent way. It is possible to mathematically describe the relat center of mass and the mass split between the suspensions. How possibilities afforded by breaking this rigid link should allow more tween

Mass:

A typical car might have a mass of around 1500kg.

Troubleshooting

This Section introduces common solutions to common problems with ve

Jittery Vehicles

- 1. Have PxInitVehicleSDK and PxVehicleSetBasisVectors been call execution of PxVehicleUpdates? Check the error stream for warnin
- 2. Does the length scale of PxTolerancesScale match the length scale 100 if centimeters are used)? Update PxTolerancesScale::length as
- 3. Is the natural frequency of the spring too high/timestep of simul reliable simulation? See Section *PxVehicleSuspensionData* for update the natural frequency or timestep accordingly. Remembe can be updated per vehicle with PxVehicleWheelsSimData::setSub
- 4. Are the maximum suspension droop and compression set to valu suspension motion?

The Engine Rotation Refuses To Spin Quickly

- 1. Are the tires resisting the engine motion through excessive friction car very high above the ground and accelerate the engine to see wheels start to spin round.
- 2. Do the engine's moment of inertia, peak torque and damping rate scale? Note the documented SI units of each variable and recom appropriate.
- 3. Is the moment of inertia too large? A value of 1 or its equivalent in 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 standard car of around 1500kg.

- 5. Does the torque curve contain sensible values? Try a flat curve w having a y-value of 1.0.
- 6. Is the maximum engine angular speed a realistic value? Cor manufacturer data for typical values or revert to the default value fc
- 7. Are any of the damping rates too high? Reduce the damping rates

The Engine Spins But the Wheels Refuse To Spin

- 1. Is the vehicle in neutral gear? Connect the engine to the wheels by to first gear and disabling the autobox.
- 2. Does the differential deliver drive torque to the wheels (for PxV only)? Make sure that the differential is properly configured.
- 3. Is the brake or handbrake engaged? Ensure that the brake and had zero.
- 4. Do the wheels' moment of inertia and damping rates reflect the length documented SI units of each variable and recompute the values as
- 5. Are the wheels' moments of inertia too high? Recompute the w inertia.
- 6. Are the wheels' damping rates too high? Reduce the wheels' damp
- 7. Are the tires resisting the engine motion through excessive friction car very high above the ground and accelerate the engine to see 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 sometimes. Check the filtering configuration for shapes attached to the vehicle search for contacts involving shapes attached to the vehicle's actor
- 2. Is sufficient friction being delivered to the tire contact patch? Query experienced by the tires during the execution of PxVeh

- PxVehicleWheelsDynData::getTireFriction.
- 3. Do the suspension forces (and the loads on the tires) reflect the body actor? Query the suspension PxVehicleWheelsDynData::getSuspensionForce. A 4-wheeled generate suspension forces of approximately actorMass*gravity/4. masses of the vehicle suspensions to ensure that the driven significant tire load.
- the significant 4. Do tires Iongitudinal tire generate PxVehicleWheelsDynData::getTireLongSlip to check that the long tire is non-zero and approaches 1.0 when the wheels are spinn forward motion. Ensure that PxVehicleSetBasisVectors has bee correct forward vector if the longitudinal slip is vanishingly small. F forward vector has been set correctly PxVehicleWheelsDynData::getTireLongitudinalDir.
- 5. Is the tire longitudinal stiffness too small? Adjust the longitudinal s default value and test.
- 6. Is the mass of the vehicle's rigid body actor too large to be driven torque? Test that the mass of the actor is a sensible value and set a
- 7. Is the rigid body actor in a PhysX scene and is the scene being up the actor is not asleep and participates in the scene update.

The Vehicle Does Not Steer/Turn

- Is the moment of inertia of the vehicle too large so that it resis Check that the moment of inertia of the vehicle's rigid body actor i Use the moment of inertia of a box with width/height/length of the v 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 ve

- maximum steer angles of the steer wheels are sensible values.
- 3. Do the steer wheels have a sensible lateral sl PxVehicleWheelsDynData::getLatSlip to query the slip ang PxVehicleSetBasisVectors has been called with the correct forwar the lateral slips are vanishingly small. Further test that the basis vecorrectly 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? F damping rate, then the wheel damping rates and retest each time.
- Is the vehicle stuck in the same gear all the time? Disable the at gears manually to test if the autobox is failing to switch gear. (settings to make sure that it will automatically increase the gear rotation speeds.
- 3. Is the engine powerful enough to quickly accelerate the car? Increating of the engine.
- 4. Do the wheels have high moments of inertia that prevent significal 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 indicate, then the wheel damping rates and retest each time.
- 2. Does the vehicle's rigid body actor have a velocity damping v appropriate.

The Vehicle Turns Too Quickly/Too Slowly

1. Does the moment of inertia of the rigid body actor need twe component of the moment of inertia that corresponds to motion a Increasing the moment of inertia will slow the turn rate, decreas 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? of increase of the accelerator pedal value by filtering the conti Remember that aggressively pressing the accelerator pedal on a to lead to wheel spin.
- 2. Are the wheel moments of inertia too low? Increase the wheel more

The Wheels Spin Too Much When Cornering

1. Does the vehicle have a limited slip differential? If applicable set 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 autok the problem is probably that the latency of the autobox is shorter t performing a gear change. The autobox latency controls the minir between automated gear changes. After an automated gear cha autobox will not make another gear change decision until the passed. During a gear change the vehicle enters neutral gear a pedal is uncoupled from the engine, meaning that the engine will the gear change. When the vehicle enters the target gear at the change the autobox might decide immediately that the engine i target gear and immediately initiate a downwards gear change. The put the car back in neutral, meaning that the car spends a very leading to the shortest decide in the car spends a very leading to the shortest decide in the car spends a very leading to the shortest decide in the car spends a very leading to the shortest decide in the car spends a very leading to the shortest decide in the car spends a very leading to the shortest decide in the car spends a very leading to the shortest decide in the car spends a very leading to the shortest decide in the car spends a very leading to the shortest decide in the car spends a very leading to the shortest decide in the car spends a very leading to the ca

and never reaches its target gear. This will not happen if th (PxVehicleAutoBoxData::setLatency) is set significantly larger the time (PxVehicleGearsData::mSwitchTime).

The Vehicle Under-steers Then Over-steers

1. Is the vehicle on a bumpy surface? Edit the values in PxVehicleTir that the filtered normalized tire load has a flatter responsion.

The Vehicle Slows Down Unnaturally

1. Does the vehicle not slow down smoothly to rest? Take a look at 1 values to see if they are oscillating between positive and negative. oscillation then two options are available that can be used separate with each other. The first option i PxVehicleWheelsSimData::setSubStepCount to force more vehicle as the forward speed of the vehicle approaches zero. The secon PxVehicleWheelsSimData::setMinLongSlipDenominator to er denominator of the longitudinal slip never falls below a specified va

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

http://www.youtube.com/watch?v=_liGnV3PTiQ http://en.wikipedia.org

Tire Modeling

The default tire model employed by PhysX vehicles is discussed in CarSimEd documentation:

http://www.eggert.highpeakpress.com/ME485/Docs/CarSimEd.pdf

PhysX vehicles allow any tire model to be simulated by specifying discussed in *Tire Shaders*. A tire model commonly used in engineerin Pacejka tire model:

http://phors.locost7.info/phors21.htm

http://phors.locost7.info/phors22.htm

Engine Torque Curves

http://www.mitsubishi-fuso.com/en/technology/qanda/05_01.html?a5

Differentials

http://www.howstuffworks.com/differential.htm

Clutch

http://auto.howstuffworks.com/clutch.htm

Ackermann Steer Correction

http://en.wikipedia.org/wiki/Ackermann_steering_geometry

Tanks

http://www.wikihow.com/Drive-an-Abbot-SPG-%28Tank%29

General

http://phors.locost7.info/intro.htm

http://www.millikenresearch.com/rcvd.html

http://www.sciencedirect.com/science/book/9780750651127

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Introduction

The character controller (CCT) SDK is an external component built of SDK, in a manner similar to PhysXExtensions.

CCTs can be implemented in a number of ways: the PhysX impleme 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 n 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 poss provides the basic features common to all CCTs: character continteractions. It is a default starting point for users, a strong base that on later modify or customize if needed.

Kinematic Character Controller

The PhysX CCT is a kinematic controller. Traditionally, character controller directly works with input discorder control). A dynamic controller works with input velocities (2r forces (3rd order control).

In the past, games did not use a 'real' physics engine like the PhysX used a character controller to move a player in a level. These games, even Doom, had a dedicated, customized piece of code to implement and response, which was often the only piece of physics in the whole galittle physics, but a lot of carefully tweaked values to provide a controlling the player. The particular behavior it implemented is often and slide' algorithm, and it has been 'tweaked for more than a decade module is an implementation of such an algorithm, providing a robu behavior for character control.

The main advantage of kinematic controllers is that they do not suffer issues, which are typical for dynamic controllers:

- (lack of) continuous collision detection: typical physics engines us checks, leading to the notorious 'tunneling effect' that has commercial & non-commercial physics packages for years. This leading problems:
 - the tunneling effect itself : if the character goes too fast through a wall
 - as a consequence, the character's maximum velocity be lillimiting the game play possibilities)
 - even if it does not tunnel, the character might jitter when pus corner for example, because the physics engine keeps mc forth to slightly different positions.

- No direct control: a rigid body is typically controlled with impuls usually not possible to move it directly to its final position: insteac the delta position vector to impulses/forces, apply them, and hope will end up at the desired position. This does not always work well the physics engine uses an imperfect linear solver.
- Trouble with friction: when the character is standing on a ramp, it s infinite friction is needed here. When the character is moving for ramp, it should not slow down. One does not need any friction he the character is sliding against a wall, it should not slow down either friction is usually either 0 or infinite. Unfortunately the friction in engine might not be perfect, and it is easy to end up with either friction (the character slows down a tiny bit) or a very-large-but-not character slides very slowly on that ramp no matter how artificing parameters are). The conflicting requirements for ramps also mean is simply no way to perfectly model desired behavior.
- Trouble with restitution: typically, restitution should be avoided for character moves fast and collides with a wall, it should not bound when the character falls from a height and lands on the ground, fluoring should be prevented. But once again, even when the reservo, a physics engine can nonetheless make the CCTs bounce a related to the imperfect nature of the linear solver, it also has to a penetration-depth-based engines recover from overlap situations, seexcessive forces that separate the objects too much.
- Undesired jumps: characters must often stick to the ground, n physical behavior should be. For example characters in action ga fast, at unrealistic speeds. When they reach the top of a ramp, often makes them jump a bit, in the same way a fast car would jur San Francisco. But that is often not the desired behavior: instead the often stick to the ground regardless of its current velocity. I implemented using fixed joints, but this is an unnecessarily con problem that is easily prevented with kinematic controllers.

• Undesired rotations: a typical character is always standing up a However physics engines often have poor support for that sort of great deal of effort is often put into preventing a capsule around falling (it should always stands up on its tip). This is often impleme joints, and the resulting system is neither very robust nor very fast.

To summarize, a lot of effort can be spent on tweaking and disabling the features simply to emulate what is otherwise a much less complex pied 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 of all created controllers and allows characters from the same manager each other. Create the manager using the *PxCreateControllerManager*

```
PxScene* scene;  // Previously created scene
PxControllerManager* manager = PxCreateControllerManager(*scene);
```

Then, create one controller for each character in the game. At the t boxes (*PxBoxController*) and capsules (*PxCapsuleController*) are supported this way:

```
PxCapsuleControllerDesc desc;
...
<fill the descriptor here>
...
PxController* c = manager->createController(desc);
```

The manager class will keep track of all created controllers. They can time using the following functions:

```
PxU32 PxControllerManager::getNbControllers() const = PxController* PxControllerManager::getController(PxU32 index) =
```

To release a character controller, simply call its release function:

```
void PxController::release() = 0;
```

To release all created character controllers at once, either release the itself, or use the following function if you intend to keep using the management.

```
void PxControllerManager::purgeControllers() = 0;
```

The creation of a controller manager and its subsequent controlle SampleBridges.

Overlap Recovery Module

Ideally, character should not be created in an initial overlap state, i created in a position where they do not overlap the surrounding geo PxScene overlap functions can be used to check the desired volume prior to creating the character. By default the CCT module does not itself, and creating a character that initially overlaps the world's static (undesired and undefined behavior - like the character going throu example.

However, the overlap recovery module can be used to automa character's initial position. As long as the amount of overlap is reasor module should be able to relocate the character to a proper, collision-fre

The overlap recovery module can be useful in several other situation main cases:

- when the CCT is directly spawned or teleported in another object
- when the CCT algorithm fails due to limited FPU accuracy
- when the "up vector" is modified, making the rotated CCT shape c objects

When activated, the CCT module will automatically try to resolve th move the CCT to a safe place where it does not overlap other objects concerns static objects, dynamic objects are ignored by this module.

Enable or disable the overlap recovery module with this function:

```
void PxControllerManager::setOverlapRecoveryModule(bool flag);
```

By default the character controllers use precise sweep tests, whose a enough to avoid all penetration - provided the contact offset is not too s cases the overlap recovery module is not needed. When it is used thougan be switched to less accurate but potentially faster versions, a

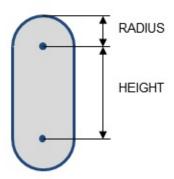
function:

```
void PxControllerManager::setPreciseSweeps(bool flag);
```

Character Volume

The character uses a bounding volume that is independent from alrea in the SDK. We currently support two different shapes around the chara

- An AABB, defined by a center position and an extents vector. To rotate. It always has a fixed rotation even when the player is (visuavoids getting stuck in places too tight to let the AABB rotate.
- A capsule, defined by a center position, a vertical height and a ra
 the distance between the two sphere centers at the end of the ca
 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 functio capsule).

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 purpexpand the volume by the size of this skin to get accurate debug visual defined in *PxControllerDesc::contactOffset* and later availat *PxController::getContactOffset()* function.

Volume Update

Sometimes it is useful to change the size of the character's volun example if the character can crouch, it might be required to reduction bounding volume so that it can then move to places he could not reach

For the box controller, the related functions are:

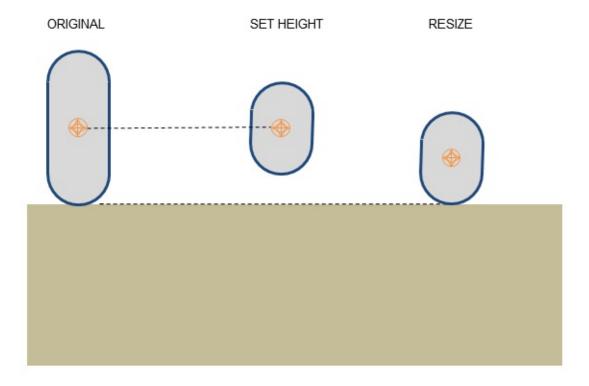
```
bool PxBoxController::setHalfHeight(PxF32 halfHeight)
bool PxBoxController::setHalfSideExtent(PxF32 halfSideExtent)
bool PxBoxController::setHalfForwardExtent(PxF32 halfForwardExten
```

And for the capsule controller:

```
bool PxCapsuleController::setRadius(PxF32 radius) = 0;
bool PxCapsuleController::setHeight(PxF32 height) = 0;
```

Changing the size of a controller using the above functions does not position. So if the character is standing on the ground (touching it) suddenly reduced without updating its position, the character will end the ground for a few frames until gravity makes it fall and touch the happens because the controllers positions are located at the center of than the bottom. Thus, to modify a controller's height and preserve its b must change both the height and position of a controller. The follow does that automatically:

```
void PxController::resize(PxF32 height) = 0;
```



It is important to note that volumes are directly modified without any ext might happen that the resulting volume overlaps some geometry ne when resizing the character to leave a crouch pose, i.e. when the size *increased*, it is important to first check that the character can indeed 'state of space above the character must be empty (collision free). It is recomvarious PxScene overlap queries for this purpose:

```
bool PxScene::overlap(...) = 0;
```

Updating the character's volume at runtime to implement a 'crouch' mo SampleNorthPole. Using overlap queries to leave the crouch pos SampleNorthPole::tryStandup() function.

Moving a Character Controller

The heart of the CCT algorithm is the function that actually moves chara

```
PxControllerCollisionFlags collisionFlags =
    PxController::move(const PxVec3& disp, PxF32 minDist, PxF32 e
    const PxControllerFilters& filters, const PxObstacleContext*
```

disp is the displacement vector for current frame. It is typically a commotion due to gravity and lateral motion when your character is moving are responsible for applying gravity to characters here.

minDist is a minimal length used to stop the recursive displacement algremaining distance to travel goes below this limit.

elapsedTime is the amount of time that passed since the last call to the

filters are filtering parameters similar to the ones used in the SDK. Use what the character should collide with.

obstacles are optional additional obstacle objects with which the chara Those objects are fully controlled by users and do not need to have objects. Note that touched obstacles are cached, meaning that the cinvalidated if the collection of obstacles changes.

collisionFlags is a bit mask returned to users to define collision eve during the move. This is a combination of PxControllerCollisionFlag flato trigger various character animations. For example your character miplaying a falling idle animation, and you might start the land anim PxControllerCollisionFlag::eCOLLISION DOWN is returned.

It is important to understand the difference between *PxCor PxController::setPosition*. The *PxController::move* function is the core of This is where the aforementioned 'collide-and-slide' algorithm takes pla will start from the CCT's current position, and use sweep tests to attention

required direction. If obstacles are found, it may make the CCT slide them. Or the CCT can get blocked against a wall: the result of the more the surrounding geometry. On the contrary, *PxController::setPosition* is function that will move the CCT to desired position no matter what, rethe CCT starts from, regardless of surrounding geometry, and even if the is in the middle of another object.

Both *PxController::move* and *PxController::setPosition* are SampleBridges.

Graphics Update

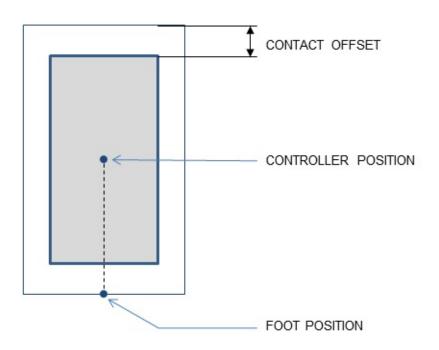
Each frame, after *PxController::move* calls, graphics object must be ke new CCT positions. Controllers' positions can be accessed using:

```
const PxExtendedVec3& PxController::getPosition() const;
```

This function returns the position from the center of the collision shape is used internally both within the PhysX SDK and by usual graphics A position and passing it to the renderer is illustrated in SampleBridge position uses double-precision, to make the CCT module work well with note that a controller never rotates so you can only access its position.

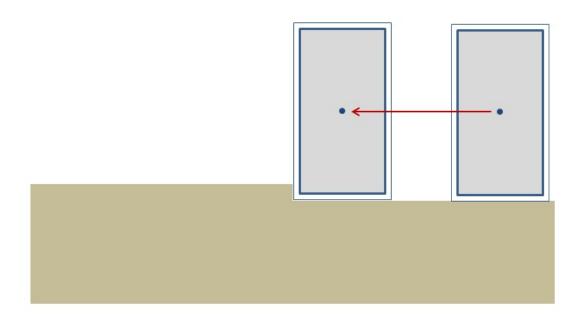
Alternative helper functions are provided to work using the character a.k.a. the foot position:

Note that the foot position takes the contact offset into account.

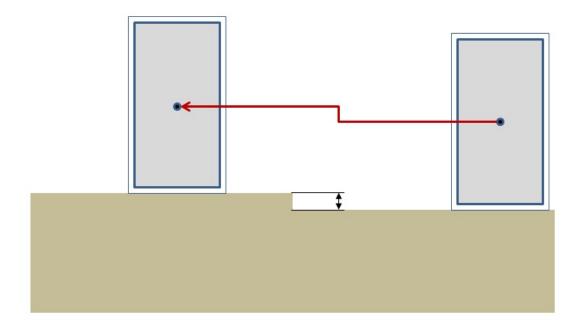


Auto Stepping

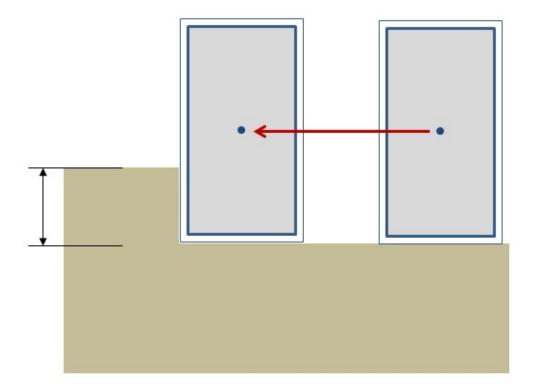
Without auto-stepping it is easy for a box-controlled character to get selevations of the ground mesh. In the following picture the small st character completely. It feels unnatural because in the real world a cl cross this small obstacle without thinking about it.



This is what auto-stepping enables us to do. Without any intervention f without them thinking about it) the box correctly steps above the minor c



However, if the obstacle is too big, i.e. its height is greater than the ste the controller cannot climb automatically, and the character gets stuck (



'Climbing' (over this bigger obstacle, for example) may also be implemed as an extension of auto-stepping. The step offset *PxControllerDesc::stepOffset* and later available through the *PxControl* function.

Generally speaking, the step offset should be kept as small as possible.

Climbing Mode

The auto-stepping feature was originally intended for box controllers blocked by small obstacles on the ground. Capsule controllers, thank nature, do not necessarily need the feature.

Even with a step offset of 0.0, capsules are able to go over small of rounded bottom produces an upward motion after colliding with a small

Capsules with a non-zero step-offset can go over obstacles higher th because of the combined effect of the auto-stepping feature and their this case the largest altitude a capsule can climb over is difficult to pre on the auto-step value, the capsule's radius, and even the magnitude ovector.

This is why there are two different climbing modes for capsules:

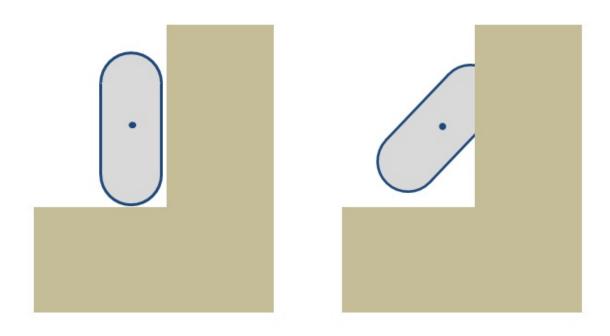
- PxCapsuleClimbingMode::eEASY: in this mode, capsules are not step offset value. They can potentially climb over obstacles higher to
- PxCapsuleClimbingMode::eCONSTRAINED: in this mode, an a make sure the capsule can not climb over obstacles higher than the

Up Vector

In order to implement the auto-stepping feature, the SDK needs to k vector. The up vector is defined in *PxControllerDesc::upDirection* a through the *PxController::getUpDirection()* function.

The up vector does not need to be axis-aligned. It can be arbitrary, m using the *PxController::setUpDirection()* function, allowing the charac spherical worlds. This is demonstrated in SampleCustomGravity.

Modifying the up vector changes the way the CCT library sees charaexample a capsule is defined by a *PxCapsuleControllerDesc::height*, wheight' along the up vector. Thus, changing the up vector effectively if from the point of view of the library. The modification happens immediat validate that the character does not overlap nearby geometry. It is the character to be penetrating some geometry right after the call. Using the module is recommended to solve these issues.



In the above picture the capsule on the left uses a vertical up vector a with the surrounding geometry. On the right the up vector has been set the capsule now penetrates the wall nearby. For most applications th constant, and the same for all characters. These issues will only appraying in spherical worlds (e.g. planetoids, etc).

Walkable Parts & Invisible Walls

By default the characters can move everywhere. This may not always particular, it is often desired to prevent walking on polygons whose s SDK can do this automatically thanks to a user-defined slope limit. *I* slope is higher than the limit slope will be marked as non walk-able, at let characters go there.

Two modes are available to define what happens when touching a nondesired mode is selected with the *PxControllerDesc::nonWalkableMode*

- *PxControllerNonWalkableMode::ePREVENT_CLIMBING* prevents moving up a slope, but does not move the character otherwise. The be able to walk laterally on these polygons, and to move down their
- PxControllerNonWalkableMode::ePREVENT_CLIMBING_AND_FC
 not only prevents the character from moving up non walk-able slo
 it to slide down those slopes.

The slope limit is defined in *PxControllerDesc::slopeLimit* and later av *PxController::getSlopeLimit()* function. The limit is expressed as the cost angle. For example this uses a slope limit of 45 degrees:

```
slopeLimit = cosf(PxMath::degToRad(45.0f));
```

Using slopeLimit = 0.0f automatically disables the feature (i.e. c everywhere).

This feature is not always needed. A common strategy is to disable it walls in the level, to restrict player's movements. The character modulations walls for you, if *PxControllerDesc::invisibleWallHeight* is non-zellibrary creates those extra triangles on the fly, and that parameter context (extruded in the user-defined up direction). A common problem is that the 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 *PxControllerDesc::maxJumµ* addresses this issue, by extending the size of the bounding volume do all potentially non-walkable triangles are properly returned by the col invisible walls are properly created - preventing the character from jump

A known limitation is that the slope limit mechanism is currently only enobjects. It is not enabled against dynamic objects, and in particular objects. It is also not supported for static spheres or static capsules.

Obstacle Objects

Sometimes it is convenient to create additional obstacles for the C without creating an actual SDK object. This is useful in a number example:

- the obstacles might only exist for a couple of frames, in which deleting SDK objects is not always efficient.
- the obstacles might only exist for stopping the characters, not t objects. This would be for example invisible walls around geor characters should collide with. In this case it may not be very eff invisible walls as SDK objects, since their interactions would then out for everything except the characters. It is probably more efficient additional invisible walls as external obstacles, that only characters
- the obstacles might be dynamic and updated with a variable times uses a fixed timestep. This could be for example a moving plat characters can stand.

At the time of writing the character controller supports box and controllers, namely *PxBoxObstacle* and *PxCapsuleObstacle*. To create the *PxObstacleContext* object using the following function:

```
PxObstacleContext* PxControllerManager::createObstacleContext()
```

Then manage obstacles with:

```
ObstacleHandle PxObstacleContext::addObstacle(const PxObstacle& o bool PxObstacleContext::removeObstacle(ObstacleHandle handle) = 0 bool PxObstacleContext::updateObstacle(ObstacleHandle handle, con
```

Typically *updateObstacle* is called right before the controllers' *move* call Using obstacles for moving platforms is illustrated in Sam

PLATFORMS_AS_OBSTACLES is defined in SampleBridgesSettings.h

Hit Callback

The *PxUserControllerHitReport* object is used to retrieve some controller's evolution. In particular, it is called when a character hits character, or a user-defined obstacle object.

When the character hits a shape, the *PxUserControllerHitReport::onS* invoked - for both static and dynamic shapes. Various impact paramet callback, and they can then be used to do various things like playing trails, applying forces, and so on. The use of *PxUserControllerHitRep* illustrated in SampleBridges. Note that this callback will only be calle character moving against a shape. It will *not* be called if a (dynam against an otherwise non-moving character. In other words, this will only *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 clafter touching a *PxShape*, a *PxController*, or a *PxObstacle*. This following functions:

```
PxControllerBehaviorFlags PxControllerBehaviorCallback::getBehavi
  (const PxShape& shape, const PxActor& actor) = 0;
PxControllerBehaviorFlags PxControllerBehaviorCallback::getBehavi
  (const PxController& controller) = 0;
PxControllerBehaviorFlags PxControllerBehaviorCallback::getBehavi
  (const PxObstacle& obstacle) = 0;
```

At the time of writing the following returned flags are supported:

PxControllerBehaviorFlag::eCCT_CAN_RIDE_ON_OBJECT defines if effectively travel with the object it is standing on. For example a charadynamic bridge should follow the motion of the PxShape it is standing SampleBridges). But it should not be the case if the character stands bottle rolling on the ground (e.g. the snowballs in SampleNorthPole). only controls the horizontal displacement communicated from an object The vertical motion is something slightly different, as many factors displacement: the *step offset* used to automatically walk over small be motion of underlying dynamic actors like e.g. the bridges in SampleBrid probably always been taken into account, etc.

PxControllerBehaviorFlag::eCCT_SLIDE defines if the character should standing on the object. This can be used as an alternative to the pr slope limit feature, to define non walk-able objects rather than non-walk-able objects rather than non-wal

PxControllerBehaviorFlag::eCCT_USER_DEFINED_RIDE simply disable related to controllers riding on objects. This can be useful to get the leg which can sometimes be necessary when porting to PhysX 3.x a property of the second se

around the PhysX 2.x character controller. The flag simply skips the ι lets users deal with this particular problem in their own application, ι 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 applyir points. However it is often not a very convincing solution.

The bounding volumes around characters are artificial (boxes, capsules so the forces computed by the physics engine between a boundir surrounding objects will not be realistic anyway. They will not p interaction between an actual character and these objects. If the bound compared to the visible character, maybe to make sure that its limbs r static geometry around, the dynamic objects will start moving (push volume) before the actual character touches them - making it look lil surrounded by some kind of force field.

Additionally, the pushing effect should not change when switching from a capsule controller. It should ideally be independent from the bounding

Pushing effects are usually dictated by gameplay, and sometimes required inverse kinematic solvers, which are outside of the scope of the CC1 simple use cases, it is for example difficult to push a dynamic box forw controller: since the capsule never hits the box exactly in the middle, ap rotate the box - even if gameplay dictates that it should move in a straig

Thus, this is an area where the CCT module should best be couple code, to implement a specific solution for a specific game. This coupl many different ways. For simple use cases it is enough the controller of the controller of the controller of the controller of the couple of the c

Note that the character controller does use overlap queries to determine nearby. Thus, SDK shapes that should interact with the characters (e the character should push) must have the PxShapeFlag::eSCENE_Ql set to true, otherwise the CCT will not detect them and characters will 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 should be fully controlled by users, and neither the PhysX SDK no should be allowed to move them.

The *PxControllerFilterCallback* object is used to define basic int characters. Its *PxControllerFilterCallback::filter* function can be used 1 PxController objects should collide at all with each other:

```
bool PxControllerFilterCallback::filter(const PxController& a, co
```

To make CCTs always collide-and-slide against each other, simply retur

To make CCTs always move freely through each other, simply return fal

Otherwise, customized and maybe gameplay-driven filtering rules can this callback. Sometimes the filtering changes at runtime, and two clallowed to go through each other only for a limited amount of time. Wh expires, the characters may be left in an overlapping state until they sagain towards each other. To automatically separate overlapping chara function can be used:

This function is an optional helper to properly resolve overlaps between should be called once per frame, before the *PxController::move* calls. To move the characters directly, but it will compute overlap information for will be used in the next *PxController::move* call.

Hidden Kinematic Actors

The CCT library creates a kinematic actor under the hood, for each combined when invoking the *PxController::move* function, the underlying hidden is also updated to reflect the CCT position in the physics scene.

Users should be aware of these hidden entities, since the total number scene will be higher than the number they created themselves. Addition back these potentially confusing unknown actors from scene-level collis

One possible strategy is to retrieve the controllers' kinematic actors function:

```
PxRigidDynamic* PxController::getActor() const;
```

Then mark these actors with a special tag, using the *PxRigidDynamic::* way the CCT actors can easily be identified (and possibly ignored) in contact reports.

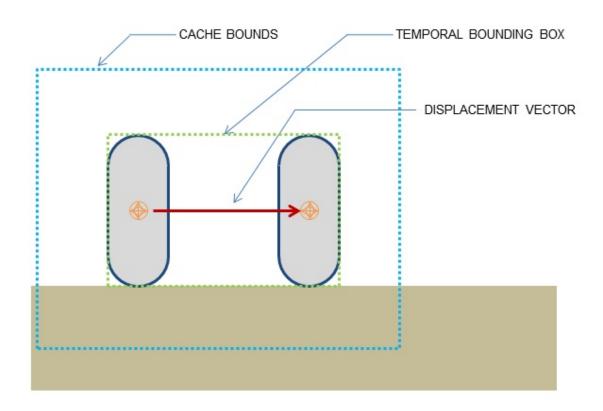
Time Stepping

Actors used internally by the CCT library follow the same rules as objects. In particular, they are updated using fixed or variable times 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 a collision queries. The temporal bounding box for a character is an character's motion (it contains the character's volume at both its start. The cached volume of space is determined by the size of the character bounding box, multiplied by a constant factor. This constant factor is character by *PxControllerDesc::volumeGrowth*. Each time a character resolution box is tested against the cached volume of space. If the motic within that volume of space, the contents of the cache are reused instalthrough PxScene-level queries.



In PhysX 3.3 and above, those caches should be automatically invalida object gets updated or removed. However it is also possible to matcaches using the following function:

void PxController::invalidateCache();

Prior to deciding if a character will travel with the motion of an object to character, a number of tests are automatically performed to decide if to object remains valid. These automatic validity tests mean that in the finot 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 cachenecessary to invalidate the cache. This holds true if the pair have consequence of an updated global pose or modified geometry.

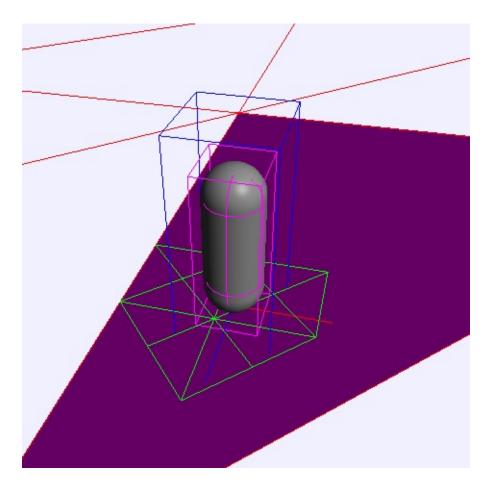
Runtime Tessellation

The CCT library is quite robust, but sometimes suffers from FPU accur character collides against large triangles. This can lead to characters r against those triangles, or even penetrating them. One way to effeproblems is to tessellate the large triangles at runtime, replacing the collection of smaller triangles. The library supports a built-in tessellatic with this function:

```
void PxControllerManager::setTessellation(bool flag, float maxEdg
```

The first parameter enables or disables the feature. The second paramaximum allowed edge length for a triangle, before it gets tessell smaller edge length leads to more triangles being created at runtil triangles get generated, the slower it is to collide against them.

It is thus recommended to disable the feature at first, and only enabl collision problems. When enabling the feature, it is recommended possible *maxEdgeLength* that does fix encountered problems.



In the screenshot, the large magenta triangle on which the characereplaced with the smaller green triangles by the tessellation mogeometry cache is represented by the blue bounding box. Note the triangles touching this volume of space are kept. Thus, the exact no produced by the tessellation code depends on both the *maxEdgeLen* the *PxControllerDesc::volumeGrowth* parameter.

Troubleshooting

This section introduces common solutions to common problems with the

Character goes through walls in rare cases

- 1. Try increasing *PxControllerDesc::contactOffset*.
- 2. Try enabling runtime tessellation with *PxControllerManager::setTes* a small *maxEdgeLength* first, to see if it solves the problem. Then as much as possible.
- 3. Try enabling overlap recovery m PxControllerManager::setOverlapRecoveryModule.

Tessellation performance issue

- 1. Try fine-tuning the *maxEdgeLength* parameter. Use the largest pos prevents tunneling issues.
- 2. Try reducing *PxControllerDesc::volumeGrowth*.

The capsule controller manages to climb over obstacles h step offset value

1. Try using PxCapsuleClimbingMode::eCONSTRAINED.

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Introduction

The PhysX particle feature has been deprecated in PhysX standalone library PhysX FleX is an alternative with a richer feature

PhysX 3 offers two particle system types - a generic particle system particle system. The generic particle system provides basic particle in with rigid actors. It can be used for objects that require collisions agains but for which inter-particle interactions are not needed. Examples in sparks or leaves. The SPH fluid particle system can be used for fluid approximate incompressibility and flowing behavior, such as liquids filling up a volume.

PhysX 3 takes care of collision detection and particle dynamics, while such as emitters, lifetime maintenance etc. need to be provided by the a

Creating Particle Systems

Both particle system classes *PxParticleSystem* and *PxParticleFPxParticleBase*, which is the common interface providing particle collision functionality. Particle systems inherit from *PxActor* and can be

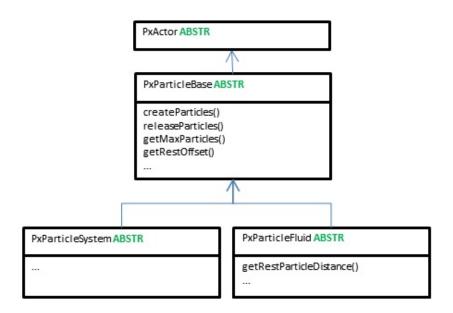


Figure 1: PxParticleSystem inherits all properties from PxParticleBase adds fluid specific properties

The following section shows how a particle system is created and adder

```
// set immutable properties.
PxU32 maxParticles = 100;
bool perParticleRestOffset = false;

// create particle system in PhysX SDK
PxParticleSystem* ps = mPhysics->createParticleSystem(maxParticle

// add particle system to scene, in case creation was successful
if (ps)
    mScene->addActor(*ps);
```

Note: The particle module has to be registered with *PxRegisterParticle* with static linking (non windows) before creating particle systems. *PxC* registers all modules by default as opposed to *PxCreateBasePhysics*.

Particle Management

Particle fixed systems reserve memory for a number PxParticleBase::getMaxParticles. Each of these particles can be addr index throughout it's lifetime. The of given range PxParticleBase::getMaxParticles]. In order to support a dynamic ar particles are marked as being valid or invalid. This is achieved by two particle range indicates the range within which particles may be valid. all particles are defined as being invalid. Within that range valid particle the flag *PxParticleFlag::eVALID*. Alternatively PhysX provides a bitma corresponding to a valid particle within the valid particle range. The bit array of 32-bit unsigned integers with enough elements to cover the vali

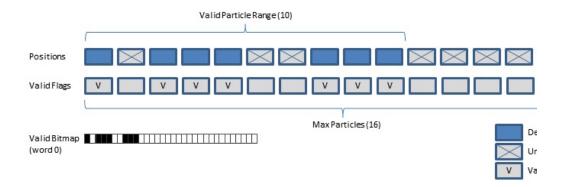


Figure 2: Scheme showing how valid particles are track

Creating Particles

The application specifies an index for each new particle at particle c application maintains its own representation of particles, and already trathen these indices may be re-used by PhysX. If the application does not 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 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 PxStridelterator template class to particle between the SDK and the application. This allows the particle data is more flexible by supporting interleaved arrays or padded data without f for reformatting. The stride iterator is configured by setting the type c and specifying the pointer to the first element.

Example for creating a few particles:

```
// declare particle descriptor for creating new particles
// based on numNewAppParticles count and newAppParticleIndices,
// newAppParticlePositions arrays and newAppParticleVelocity
PxParticleCreationData particleCreationData;
particleCreationData.numParticles = numNewAppParticles;
particleCreationData.indexBuffer = PxStrideIterator<const PxU32>(
particleCreationData.positionBuffer = PxStrideIterator<const PxVe
particleCreationData.velocityBuffer = PxStrideIterator<const PxVe
// create particles in *PxParticleSystem* ps
bool success = ps->createParticles(particleCreationData);
```

The indices specified for particle creation need to be unique and PxParticleBase::getMaxParticles().

In this example the stride iterator is used to set the same velocity for all is achieved by setting the stride to zero.

Note: For fluid particles it is necessary to spawn particles at distances *PxParticleFluid::getRestParticleDistance()* in order to achieve a regular otherwise particles will spread immediately in all directions.

Note: In PhysX 3 all particle access such as creating, releasing, upda particles can only be carried out while the simulation of the scene is no

Releasing Particles

Particles can be released by providing indices to the particle system. A versions of the PhysX SDK, particles get immediately released.

Example for releasing a few particles:

```
// declare strided iterator for providing array of indices corres
// 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 metho correspond to existing particles.

All particles can be released at once by calling:

```
ps->releaseParticles();
```

Since only a limited number of particle slots (*PxParticleBase::getll* available it might be appropriate to replace old particles with new achieved for instance by maintaining an application-side particle lifetimes reasons to release particles:

- Drains can be useful to remove particles that go to locations we needed anymore. See *Particle Drains*.
- The spatial data structure used for particles may overflow. Partic covered are marked and should be released. See *Particle Grid*.

Index Pool Extension

Example for allocating particle indices using the PhysX extensions libral

```
// create an index pool for a particle system with maximum partic
PxParticleExt::IndexPool* indexPool = PxParticleExt::createIndexP
// use the indexPool for allocating numNewAppParticles indices th
```

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 (onl *PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET*).

Particle updates that are carried out during the next scene simulation st

• Force updates: Results in a velocity change update according specified by *PxForceMode*.

Example for force update:

```
// specify strided iterator to provide update forces
PxStrideIterator<const PxVec3> forceBuffer(appParticleForces);

// specify strided iterator to provide indices of particles that
PxStrideIterator<const PxU32> indexBuffer(appParticleForceIndices

// specify force update on PxParticleSystem ps choosing the "forc
ps->addForces(numAppParticleForces, indexBuffer, forceBuffer, PxF
```

Reading Particles

The PhysX SDK does not provide to the user all simulated per-parti particle system by default. The application can specify the data it ne 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 defenabled if the particle system was created with per particle rest (
 Creating Particle Systems.
- PxParticleReadDataFlag::eCOLLISION_NORMAL_BUFFER: Off b
- PxParticleReadDataFlag::eDENSITY_BUFFER: Only available for 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 static during the last simulation step.
- PxParticleFlag::eCOLLISION_WITH_DYNAMIC: Shows whether with a dynamic rigid body during the last simulation step.
- PxParticleFlag::eCOLLISION_WITH_DRAIN: Shows whether a par rigid actor shape that was marked as a drain (Particle Drains).
- PxParticleFlag::eSPATIAL_DATA_STRUCTURE_OVERFLOW: S particle had to be omitted when building the SDK internal space (Particle Grid).

Particle collision normals represent contact normals between particl

surfaces. A non-colliding particle has a zero collision normal. Collision e.g. for orienting the particle visualization according to their contact with

Particle densities provided by particle fluids can be used for rendering has a value of zero for a particle that is completely isolated. It has a particle that has a particle neighborhood with a mean spacing PxParticleFluid::getRestParticleDistance().

Particle data can only be read while the scene simulation is not executaccess to the SDK buffers a *PxParticleReadData* instance needs to be SDK. It has the following properties:

- numValidParticles: Total number of valid particles for the corr system.
- validParticleRange: The index range of valid particles in the particle
- *validParticleBitmap:* Bitmap of valid particle locations.
- positionBuffer, positionBuffer, velocityBuffer, restOffsetBu collisionNormalBuffer: Strided iterators for particle properties.

Additionally particle fluids provide PxParticleFluidReadData with

• *densityBuffer:* Strided iterator for particle densities.

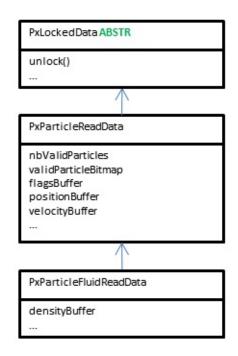


Figure 3: PxParticleReadData and PxParticleFluidReadI

Example of how to access particle data:

```
// lock SDK buffers of *PxParticleSystem* ps for reading
PxParticleReadData* rd = ps->lockParticleReadData();
// access particle data from PxParticleReadData
if (rd)
{
    PxStrideIterator<const PxParticleFlags> flagsIt(rd->flagsBuff
    PxStrideIterator<const PxVec3> positionIt(rd->positionBuffer)
    for (unsigned i = 0; i < rd->validParticleRange; ++i, ++flags
        if (*flagsIt & PxParticleFlag::eVALID)
        {
            // access particle position
            const PxVec3& position = *positionIt;
        }
    }
    // return ownership of the buffers back to the SDK
    rd->unlock();
```

}

Example of how to use the valid particle bitmap to access particle dat the locking and unlocking):

Parameter Guide

There are three types of particle system parameter. Some need to be particle system is created and cannot be changed afterwards. Some are particle system is not part of a scene and others can be changed following description covers parameter that either cannot be set at induce a performance overhead when changed.

maxParticles:

The maximum number of particles that can be added to a particle sys smaller the value, the smaller the memory footprint of the particle sys be. Can only be set on particle system creation.

PxParticleReadDataFlags:

Specifies a subset of simulation properties which are returned to the a simulation. See *Reading Particles*. As few read data flags should be a order to save memory and improve performance by avoiding unneces data copying. Parameter can only be changed while particle system is scene.

gridSize:

A hint for the PhysX SDK to choose the particle grouping granularity f tests and parallelization. See *Particle Grid*. Parameter can only be ch 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 is part of a scene induces a large performance overhead.

PxParticleBaseFlag::eCOLLISION_WITH_DYNAMIC_ACTORS:

Enable/disable collision with dynamic rigids. Changing this parameter particle system is part of a scene induces a performance overhead.

PxParticleBaseFlag::eCOLLISION_TWOWAY:

Enable/disable twoway interaction between particles and rigid bodies parameter while the particle system is part of a scene induces a performance overhead.

PxParticleBaseFlag::ePER_PARTICLE_COLLISION_CACHE_HINT:

Enable/disable internal collision caches. Changing this parameter whisystem is part of a scene induces a performance overhead.

Particle Dynamics

externalAcceleration:

Acceleration applied to each particle at each time step. The scene grandded to the external acceleration by default can be disabled using PxActorFlag::eDISABLE_GRAVITY.

maxMotionDistance:

The maximum distance a particle can travel during one simulation ste may hurt performance, while low values may restrict the particle veloc order to improve performance it's advisable to set this to a low value a increase it until particles can move fast enough to achieve the target of Parameter can only be changed while particle system is not part of a

damping:

Velocity damping constant, which is globally applied to each particle. particularly useful when using particles for smoke to prevent ballistic I individual particles which can look odd.

particleMass:

Mass used for two way interaction with rigid bodies (*PxParticleBaseFlag::eCOLLISION_TWOWAY*) and different force moreontext of *PxParticleBase::addForces*. This mass property doesn't had on the fluid dynamics simulation.

PxParticleBaseFlag::ePROJECT_TO_PLANE, projectionPlaneDistance:

Parameter to configure the projection mode which confines particles t projection is enabled particles can only move in a plane. This can be in the context of a 2D-Game.

Collision with Rigid Actors

restOffset:

Defines the minimum distance between particles and the surface of ri maintained by the collision system. Parameter can only be changed v system is not part of a scene.

PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET:

Enables/disables per-particle rest offsets. Memory can be saved by to particle rest offsets off. Per-particle rest offsets should only be enable represent objects of significantly varying size, for example in the contested. See *Per-particle Rest Offsets*. Can only be set on particle sys

contactOffset:

Defines the distance at which contacts between particles and rigid ac The contacts are internally used to avoid jitter and sticking. It needs to restOffset. A good value to start with is about twice the size of the res Parameter can only be changed while particle system is not part of a

restitution:

Restitution used for particle collision. This parameter defines how strc bounce of rigid actors.

dynamicFriction:

Dynamic friction used for particle collision. This parameter defines ho particles slide over rigid actor surfaces. The lower the value is to 0, th particles slide. One is the maximal value supported.

staticFriction:

Static friction used for particle collision. This parameter is similar to dy but defines how easily particles start to slide over a surface. Values la are supported.

simulationFilterData:

Filter data used to filter collisions between particles and rigid bodies. *Filtering*.

PxParticleBaseFlag::eCOLLISION_TWOWAY:

The collision two-way flag allows enabling/disabling two-way interaction bodies and particles. The particle mass parameter defines the strengt interaction. The flag can only be changed while the particle system is scene.

Fluid (PxParticleFluid)

The SPH simulation can be tricky to tweak for good results. As this si (see *References*) uses an explicit integration scheme it only provides s a certain parameter sub-space. A good set of parameter values deper size of the simulation and the external forces applied (such as gravil starting points for parameter values below assume a time step size of a gravity around 10 [m/s^2]. Using a *damping* value larger than zero parameter sub-space, for example useful when implementing a smoke of

restParticleDistance:

Defines the resolution of the particle fluid. It defines the approximate an eighboring particles will adopt within a fluid volume at rest. For the particle assumption mentioned above, the particle rest distance sho smaller than 0.05 [m]. Parameter can only be changed while particle apart of a scene.

stiffness:

The stiffness (or gas constant) influences the calculation of the pressing Low values of stiffness make the fluid more compressible (i.e., spring values make it less compressible. The stiffness value has a significan numerical stability of the simulation; setting very high values will resul Reasonable values are usually between 1 and 200.

viscosity:

Viscosity controls a fluid's thickness. For example, a fluid with a high behave like treacle, while a fluid with low viscosity will be more like was viscosity value scales the force to reduce the relative velocity of partic fluid. Both, too high and too low values will typically result in instabiliti values are usually between 5 and 300.

Collision Handling

By default, particles will collide with any shapes inside the PxScene t They will attempt to maintain a fixed distance from these shape PxParticleBase::setRestOffset().

Collision Filtering

Filtering particle versus rigid body collisions can be useful to a performance overhead or simply to avoid undesired collisions.

For the following examples filtering is useful:

- Avoid particles colliding with trigger shapes (this is already *PxDefaultSimulationFilterShader*)
- Configure a drain shape to exclusively collide with particles
- Have particles collide with a proxy shape as opposed to the shape collisions

Filter information for particles can be specified PxParticleBase::setSimulationFilterData(). Instructions for how to setup be found here: Collision Filtering.

Per-particle Rest Offsets

It is also possible to set a rest offset per-particle, using *PxParticleBase* order provide In to per-particle PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET needs to be smaller than the offsets must be per-system va PxParticleBase.getRestOffset().

Particle Drains

Using drains is a good method for keeping the particle count and spi Placing drains around the area of interest in which a particle syster maintain good performance of the particle simulation. The area of example, also be moved with the player.

Example of how to flag a *PxShape* rbShape as a drain:

```
rbShape->setFlag(PxShapeFlag::ePARTICLE_DRAIN, true);
```

Particles that collide with a drain are PxParticleFlag::eCOLLISION_WITH_DRAIN and may be released.

Best Practices / Troubleshooting

Particle Grid and Spatial Data Structure Overflow

The PhysX SDK uses a grid to subdivide the particles of a particle groups. This is done to accelerate proximity queries and for paralleliza grid size parameter needs to be experimentally *PxParticleBase::setGridSize()* for best performance. When doing the visualize the grid using *PxVisualizationParameter::ePARTICLE_SYS* grid size values might result in spatial data structure overflow, since cells is limited to about 1000. Large grid size values on the other hand performance due to ineffective spatial queries or lack of parallelization c

In case of overflow, some particles will stop colliding with rigid actors in particles are marked with *PxParticleFlag::eSPATIAL_DATA_STRUCT* and should be released.

GPU/CUDA Acceleration

PhysX 3 supports GPU acceleration. This allows for larger and moleffects while retaining good performance levels. To achieve this gaphysx::PxGpuDispatcher for the scene we want to add the particle system.

A particle system can be configured for GPU simula *PxParticleBaseFlag::eGPU*. Toggling GPU acceleration while the partic a scene might have a bad impact on performance since its state needs from the GPU device memory. It is therefore better to so *PxParticleBase::setParticleBaseFlag()* before adding the particle system

Particle data can be read directly from the GPU PxParticleBase::lockParticleReadData(PxDataAccessFlag::eDEVICE) PxParticleFluid::lockParticleFluidReadData(PxDataAccessFlag::eDEVIC used to render particles directly with CUDA Graphics Interop.

Convex, Triangle and Height field meshes are automatically mirrored in when the corresponding shapes are within the proximity of a GPU ε system. This may cause some undesired performance hiccups which comirroring the meshes explicitly, as shown in this example:

```
#if PX_WINDOWS
    // mirror PxTriangleMesh triangleMesh providing the correspon
    // the desired scene.
    PxParticleGpu::createTriangleMeshMirror(triangleMesh, *cudaCo
    // later release the obsolete mirror
    PxParticleGpu::releaseTriangleMeshMirror(triangleMesh, *cudaC
#endif
```

On Kepler and above GPUs, the triangle meshes can be cached performance. The amount of memory to be allocated for caching can be

```
PxParticleGpu::setTriangleMeshCacheSizeHint(const class PxScene&
```

The triangle mesh cache will be shared among all the particle syste scene. The optimal size depends on the scene (i.e. triangle mesh d distribution). The cache usage statistics can be queried and analyze cache size hint.

Sample Discussion

The SampleParticles shows both particle system types being used: *F* used for small debris and smoke, while *PxParticleFluid* is used for a war provides example implementations of various aspects described in this

- SampleParticles::createParticleSystem, SampleParticles::createFl create particle systems.
- ParticleSystem::createParticles creates particles within a particle sy
- ParticleSystem::update shows how to read, update, release particl with spatial data structure overflows.
- SampleParticlesFilterShader is an example for setting up collision f
- SampleParticles::createDrain shows how to setup a rigid body s drain.
- SampleBase::onInit illustrates how to setup GPU/CUDA acceleration

The sample makes use of various helper classes:

- ParticleSystem: Encapsulates a PxParticleSystem or PxParticleI manages application side data such as particle lifetimes and orient facilitates creating and releasing particles and double buffers asynchronous rendering.
- RenderParticleSystemActor: Owns a ParticleSystem and p functionality.
- ParticleEmitterRate: Emits particles at a specified rate (#particles p
- ParticleEmitterPressure: Emits particles maintaining a certain dista
- SampleParticles::Emitter: Connects an emitter as described RenderParticleSystemActor.
- SampleParticles::Raygun: Provides functionality for the ray forc debris, particle debris and smoke emission.

In the sample, the smoke effect is achieved by using a *PxParticleSyst* Each particle is rendered as a point sprite with a smoke texture. The when the particles get close to the end of their lifespan. The smoke p the scene, which can be seen when roaming the smoke with the generated for the craters, as well as for the ray-gun impacts. The rea effect could be increased by using a particle fluid in order to get the expand. This is typically useful for indoor scenes or ground fog like particles get into pooling situations.

Two kinds of debris are shown in the sample. Larger chunks of debrusing convex-shaped rigid bodies. Smaller but more abundant chunks particles, which helps performance. The particle based debris is render 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 simp

- 1. Assign an initial random rotation matrix to each particle.
- 2. Change this rotation matrix proportional to the linear velocity of par

The implementation of this approach can be ParticleSystem::initializeParticlesOrientations and ParticleSystem::mode

References

Particle-Based Fluid Simulation for Interactive Applications
Matthias Muller, David Charypar and Markus Gross, Eurogrpahics/Siggi 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

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Introduction

The PhysX clothing feature has been deprecated in PhysX versior and APEX clothing features are replaced by the standalone NvClot

Realistic movement of character clothing greatly improves the player PhysX 3 cloth feature is a complete and high-performance solution to clothing. It provides local space simulation for high accuracy and stabil to reduce stretching, collision against a variety of shapes, as well as p and inter-collision to avoid the cloth penetrating itself or other clc simulation can be offloaded to CUDA capable GPUs for better performance solutions than the CPU is able to handle.

PhysX 3 cloth is a rewrite of the PhysX 2 deformables, tailored character cloth. Softbodies, tearing, and two-way interaction have be behavior and performance for cloth simulation have been improved.

Simulation Algorithm

For one PhysX simulation frame, the cloth solver runs for multiple iteration iterations is determined by the solver frequency parameter and the time. Each iteration integrates particle positions, solves constrain character and self-collision. Cloth inter-collision is performed once per-instances in the scene have been stepped forward. Local frame, moti collision shapes are interpolated per iteration from the per-frame value user.

Solver Frequency

The size of the iteration time step is inversely proportional to the iterations:

```
cloth.setSolverFrequency(240.0f);
```

The solver frequency is specified as iterations per second. A solver freq corresponds to 4 iterations per frame at 60 frames per second. In gen become more accurate if higher solver frequency value is used. However grows roughly linearly with solver frequency. Typically this value is between

The number of iterations for each frame is derived using the simulation simulation time-step. PhysX tries to handle variable time-steps c variations of the time-step into account during position integration ϵ damping parameters like constraint stiffness. While this reduces th artifacts due to varying time step sizes, use of variable time step sizecommended.

Particle Integration

The first step in a cloth iteration predicts the new particle position by position, velocity and external acceleration. While a particle state con position and the position before the last iteration, the particle velocity in

computed by dividing the position delta by the delta time of the previous

Local Space Simulation and Inertia Scale

Each PxCloth actor has a transformation that transforms particles from world space positions. For example:

```
cloth.setGlobalPose(PxTransform(PxVec3(1.0f, 0.0f, 0.0f));
```

will change the cloth's world space position to (1,0,0). Now compare that

```
cloth.setTargetPose(PxTransform(PxVec3(1.0f, 0.0f, 0.0f));
```

, which also changes the cloth's position to the same place. So what's d

PxCloth::setGlobalPose() only moves the cloth, but PxCloth::set generates acceleration (inertia) due to the position change. The amount frame acceleration affects the cloth particles can be controlled using a example to impart half the local frame acceleration to the particles use:

```
cloth.setInertiaScale(0.5f);
```

Scaling inertia effects individually per translation and rotation axis is als the family of PxCloth::set*InertiaScale() methods. Limiting the amount accelerations affect particles can be especially useful for fast moving ch

Note: Using setGlobalPose() is equivalent to using setTargetPos() wh 0. In this case, the cloth does not receive any acceleration due to frame

Constraints

After the particle positions have been integrated, a set of different const simulate stretch, shear and bending forces, as well as to confine the within a certain region.

Distance Constraints

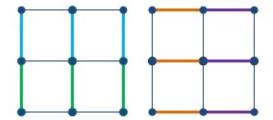


Figure 1. Typical configuration for vertical (left), and horizontal (right) str

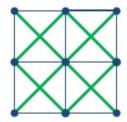


Figure 2. Typical configuration for shearing distance const

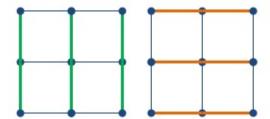


Figure 3. Typical configuration for vertical (left), and horizontal (right) be note bending constraints typically span more than one edge in

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 constraints into 4 types (see *PxClothFabricPhaseType*), each of which with different stiffness parameters.

Below is an example of stiffness settings for each constraint type:

```
cloth.setStretchConfig(PxClothFabricPhaseType::eVERTICAL, PxCloth
cloth.setStretchConfig(PxClothFabricPhaseType::eHORIZONTAL, PxClo
cloth.setStretchConfig(PxClothFabricPhaseType::eSHEARING, PxCloth
cloth.setStretchConfig(PxClothFabricPhaseType::eBENDING, PxClothS
```

Sometimes it is desirable that distance constraints are not enforce stiffness parameter allows only correcting a portion of the edge le iteration, for example to reduce the strength of bending constraints. stiffness can be used for edges that are only moderately stretched of example, a dress can be made to stretch when the character is taking I behave correctly during pirouettes.

The following code sets up the vertical constraints such that when edge more than 60% or stretched more than 120%, a stiffness of 0.8 will be stiffness of 0.4 = 0.8 * 0.5 will be used:

```
PxClothStretchConfig stretchConfig;
stretchConfig.stiffness = 0.8f;
stretchConfig.stiffnessMultiplier = 0.5f;
stretchConfig.compressionLimit = 0.6f;
stretchConfig.stretchLimit = 1.2f;
cloth.setStretchConfig(PxClothFabricPhaseType::eVERTICAL, stretch
```

Note: Stretch settings for horizontal and vertical directions are specificant to be used to handle stretching along the gravity (vertical) direct

Tether Constraints

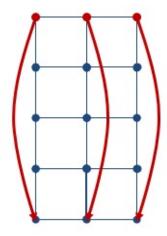


Figure 4. Example tether constraint configuration

The distance constraints are solved only once per iteration without con-The most visible artifact of this approximation is that the cloth Increasing solver frequency reduces the stretching, but results in intime.

PhysX 3.3 introduces tether constraints as a solution to avoid stretchine fast motion. Tether constraints prevent stretching by limiting the distantone away from their anchor particles. This constraint adds very computation to the solver, so it is more effective than increasing the nun

The tether constraints are automatically generated by the constraints are automatically generated by the constraints. The constraints are automatically generated by the constraints are set to zero, telling the corresponding particles are non-simulated anchor particles whose possible from users. Changing inverse masses after the fabric has been affect which anchor particles are used for the tether constraints.

Motion Constraints

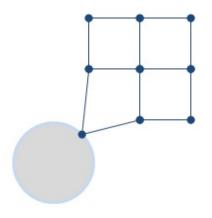


Figure 5. Example motion constraint

One can fully constrain a point to user specified position with zero inver 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 animation system can sketch the overall movement of a cloth while th are handled by the cloth simulation.

PxClothParticleMotionConstraint structure holds the position and radiu each particle, and motion constraints can be specified as follows:

```
PxClothParticleMotionConstraints motionConstraints[] = {
    PxClothParticleMotionConstraints(PxVec3(0.0f, 0.0f, 0.0f), 0.
    PxClothParticleMotionConstraints(PxVec3(0.0f, 1.0f, 0.0f), 1.
    PxClothParticleMotionConstraints(PxVec3(1.0f, 0.0f, 0.0f), 1.
    PxClothParticleMotionConstraints(PxVec3(1.0f, 1.0f, 0.0f), FL
};
cloth.setMotionConstraints(motionConstraints);
```

If the sphere radius becomes zero or negative, the corresponding parti sphere center and the inverse particle mass is set to zero for the neabove example, the first particle will fully lock to the constraint position and third particle will remain within the sphere radius. The last p constrained.

The motion constraint sphere radius can be globally scaled and bias transition between simulated and animated states. See *PxClothMotionC* details.

Separation Constraints

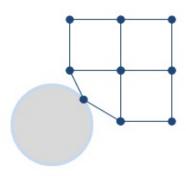


Figure 6. Example separation constraint

Separation constraints work exactly the opposite way to the motion constraints to stay outside of a sphere. When particle movement is modera motion constraints (e.g. sleeves around an arm), separation constraints represent the character's collision shape more accurately than using constraints, separation constraints can be placed slightly inside the characteristic to be the distance from the sphere center to the surface of the characteristic can be placed.

See PxClothParticleSeperationConstraint and PxCloth::setSeparationConstraint and PxCl

Collision Detection

Each cloth object supports collision with spheres, capsules, planes, complanes) and triangles. By default these shapes are all treated separa PhysX rigid body scene, however collision against other PxScene actors using the PxClothFlag::eSCENE COLLISION flag.

Collision shapes are specified in local coordinates for the next frame be scene. An independent and complete collision stage is performed as iteration, using shape positions interpolated from the values at the beg

of the frame. Sphere and capsule collision supports continuous collisior an acceleration structure to cull far-away particles early in the collisior 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 spechange 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 I multiple capsules, which can be useful for modeling characters (upper a up from capsules can share the sphere at the knee). Sharing of spheresimulation more efficient and robust, so is highly encouraged.

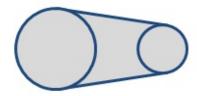


Figure 7. A tapered capsule collision shape formed by two conne-

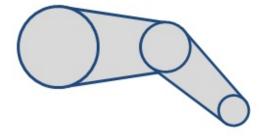


Figure 8. A leg shape formed by using two tapered capsules, each shar middle

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 triangles should form a closed patch near the cloth for consistent collist particle collides against its closest triangle expanded to an infinite plane

The order of planes and triangles should remain unchanged (apart frough the *PxCloth::removeCollisionPlane/Triangle()* method) as t interpolated between simulation frames.

Continuous Collision Detection

Besides discrete collision which resolves particles inside shapes a iteration, continuous collision detection is supported and can be enabled

```
// Enable continuous collision detection
cloth.setClothFlag(PxClothFlag::eSWEPT_CONTACT, true);
```

Continuous collision is around 2x more computationally expensive that but it is necessary to detect collision between fast moving objects. C analyzes the trajectory of particles and capsules to determine when a contact, the particle is moved with the shape until the en

Note: The SIMD collision path handles sets of 4 particles in parallel. I advantageous to spatially group cloth particles so that they are likely to same set of shapes.

Virtual Particle Collision

Virtual particles provide a way of improving cloth collision without ir resolution. They are called 'virtual' particles because they only exist processing stage and do not have their position, velocity or mass e regular particles, they can be thought of as providing additional samp surface.

During collision processing each virtual particle is created from thre using barycentric interpolation. It is then tested for discrete collision lik and the collision impulse is redistributed back to the original particinterpolation.

Section Adding Virtual Particles explains the necessary steps to use this

Friction and Mass Scaling

Coulomb friction can be enabled and will be applied for particle a collisions by setting a friction coefficient between 0 and 1:

```
cloth.setFrictionCoefficient(0.5f);
```

Additionally, there is an option to artificially increase the mass of coll temporary increase in mass can help reduce stretching along edges the pulled over a collision shape. The effect is determined by the relative the particle and collision shape and a user defined coefficient. A value starting point but users are encouraged to experiment with this value:

```
cloth.setCollisionMassScale(20.0f);
```

Self-Collision of a Single Cloth Actor

The particles of a cloth actor can collide among themselves. To enab behavior, one should set both self-collision distance and self-collision s values:

```
cloth.setSelfCollisionDistance(0.1f);
cloth.setSelfCollisionStiffness(1.0f);
```

Self-collision distance defines the diameter of a sphere around each solver ensures that these spheres do not overlap during simulat stiffness defines how strong the separating impulse should be.

Self-collision distance should be smaller than the smallest distance bein the rest configuration. If the distance is larger, self-collision may vio constraints and result in jittering.

When such a configuration cannot be avoided (e.g. due to irregular ir one can assign additional rest positions:

```
cloth.setRestPositions(restPositions);
```

Collision between two particles is ignored if their rest-positions are collision distance. However, a large collision distance and use of significantly degrade performance of self-collision, so should be used specifically

Self-collision performance for high-resolution cloth instances can be ir self-collision to a subset of all particles (see *PxCloth::setSelfCollisionInc*

Inter-Collision between Multiple Cloth Actors

Different cloth actors can be made to interact with each other whe enabled. The parameters for inter-collision are set for all cloth instances

```
scene.setInterCollisionDistance(0.5f);
scene.setInterCollisionStiffness(1.0f);
```

The definition of distance and stiffness values are the same as a instances that specify a particle subset for self-collision use the sam collision.

Best Practices / Troubleshooting

Performance Tips

The runtime of the cloth simulation scales approximately linearly with t particles and the solver frequency: Simulating a higher resolution mesh and increasing stretch stiffness and collision handling fidelity v frequencies increase the time it takes to simulate one frame. Addi performance drop somewhere below 3000 particles for the GPU solver next section. As a rough guideline, a dozen cloth instances with 2000 p solver frequency of 300Hz can be simulated in real-time as part of a gar

Convex collision and triangle collision do not use any mid phase acc 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-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 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 exceptio When no supported GPU is available PhysX will issue a warning simulations will be run on CPU.

When the cloth is simulated using CUDA, the GPU simulation results c graphics API by requesting CUDA device pointers to the particle data:

cloth.lockParticleData(PxDataAccessFlag::eDEVICE);

To take full advantage of the GPU hardware there should be at le instances as streaming multiprocessors (SMs). This means it is a simulate clothing as multiple instances (e.g. shirts and skirt) rather that instance.

GPU PhysX performance is better when the particle data of a cloth memory. The number of particles that fit into shared memory depends collision shapes, whether continuous collision or self-collisions are en the GPU version. For GPUs supporting SM 2.0 and above, about 250 into shared memory. If particles don't fit into shared memory the automatically stream particles through global memory, which incurs cost.

Furthermore, the limited size of shared memory requires the number c to be clamped to 500 when GPU simulation is enabled.

Fast-Moving Characters

Consistent collision handling for fast-moving characters can be difficul and rotations are best handled by tying the cloth local simulation fran transformation. The inertia effects of the local frame transformations using the inertia scale settings.

If the cloth tunnels collision shapes during fast character animations, solver frequency or enabling swept contacts (see *PxClothFlag::eSWEP*

Avoiding Stretching

Due to the iterative nature of the distance constraint solver, high restretch undesirably under strong gravity even if the stretch stiffne Increasing the solver frequency mitigates the stretching, but tether con better suited to eliminate stretching efficiently.

Avoiding Jitter

Under certain configurations, different constraint types can violate eaconstrain the particle positions. For example, a motion constraint can further from the anchor particle than the tether constraint permits, o pinched between two overlapping collision shapes. Over-constraining can result in jitter and should be avoided. In some situations jitter and increasing the solver frequency or by reducing the corresponding consti

PVD Support

Cloth particle positions, distance constraints, and collision shapes are I lines, and wireframes respectively in PVD. The SDK does not have a used to create the fabric, and this mesh can't be displayed in PVD ei can display individual sets of distance constraints instead of all at onc *Mode* to *Single Phase* in the *Preferences* dialog and use the *Cloth Phase* phase to display. The *Particle Scale* slider in the same dialog affects th ordinary and virtual cloth particles as well. All properties of a selected viewed in the *Inspector* panel of PVD.

Snippet Discussion

The following paragraph describes code of the cloth snippet provided w

The cloth constraint connectivity and rest values are stored in (*PxClothFabric*), separate from the cloth actor (*PxCloth*). The separatio from particles allows the same fabric data to be reused for multip reducing cooking time and storage requirements. *PxClothFabricCreate* library, creates a fabric from a triangle or quad mesh (see *PxClothMe* actor itself is created through the physics instance (*PxPhysics*) and ne a scene (*PxScene*) in order to be simulated. Once the cloth actor is assign simulation settings such as collision data, constraint stiffness, so self-collision. The *createCloth* function in the cloth snippet performs the

The *stepPhysics* function advances the simulation by one frame. It firs local frame, which rotates around the y-axis. The collision shapes are n coordinates, but their positions are specified in cloth local coordinates, to be updated every frame. The following sections detail some of the avand show how to configure them.

Note: The cloth module has to be registered with *PxRegisterCloth* on static linking (non windows) before creating cloth objects. *PxCreatePhy* modules by default as opposed to *PxCreateBasePhysics*.

Filling in PxClothMeshDesc

The first task to create a cloth is to fill in the *PxClothMeshDesc* struprogrammatically creates a regular grid of cloth particles connected Below is a simpler example on how to create a cloth from a simple m single quad.

```
PxClothParticle vertices[] = {
    PxClothParticle(PxVec3(0.0f, 0.0f, 0.0f), 0.0f),
    PxClothParticle(PxVec3(0.0f, 1.0f, 0.0f), 1.0f),
```

```
PxClothParticle(PxVec3(1.0f, 0.0f, 0.0f), 1.0f),
    PxClothParticle(PxVec3(1.0f, 1.0f, 0.0f), 1.0f)
};

PxU32 primitives[] = { 0, 1, 3, 2 };

PxClothMeshDesc meshDesc;
meshDesc.points.data = vertices;
meshDesc.points.count = 4;
meshDesc.points.stride = sizeof(PxClothParticle);

meshDesc.invMasses.data = &vertices->invWeight;
meshDesc.invMasses.count = 4;
meshDesc.invMasses.stride = sizeof(PxClothParticle);

meshDesc.quads.data = primitives;
meshDesc.quads.count = 1;
meshDesc.quads.stride = sizeof(PxU32) * 4;
```

Each particle is defined by its position in local coordinates and its inv the inverse mass to zero indicates that the particle is not simulated. Ins fixed in local space or kinematically constrained to user specified pos mass of simulated particles can normally be set to any fixed positive val

The *PxClothMeshDesc* structure allows positions and inverse masses arrays or interleaved like in the code above. The mesh can consist of or both. The cooker prefers quad meshes over triangle meshes when and classifying constraint types. The extensions library theref *PxClothMeshQuadifier* helper class to extract quads from a triangle mesh

Creating Fabric

Given the mesh descriptor, a call to *PxClothFabricCreate* in the extent the generation of constraints and the creation of the *PxClothFabric* structure.

```
PxClothFabric* fabric = PxClothFabricCreate(physics, meshDesc, Px
```

The third parameter indicates the direction of gravity, which is used as the direction of 'horizontal' or 'vertical' constraints.

The *PxClothFabric* class describes internal solver data for a cloth. For constraints consisting of two particle indices and a rest-length are creand stored in the fabric data. Multiple cloth instances of the same mesh fabric instance.

Creating Cloth

A *PxCloth* object is created using a fabric instance and the initial palatike all actors, the cloth instance is simulated as part of a scene:

```
PxTransform pose = PxTransform(PxIdentity);
PxCloth* cloth = physics.createCloth(pose, fabric, vertices, PxCl scene.addActor(cloth);
```

The first parameter specifies the initial pose. The second input is a created by the cooker. The third input provides initial particle posses. Typically this array is the same as the one referenced by the used to create the fabric. Note that the rest configuration (such as the distance constraint) is computed from PxClothMeshDesc, so the initial do not affect rest configuration. The last parameter is a set of flag simulation and continuous collision detection to be enabled. The defau options.

Specifying Collision Shapes

The following code illustrates how to add two spheres of different ratapered capsule between them:

```
// Two spheres located on the x-axis
PxClothCollisionSphere spheres[2] =
{
    PxClothCollisionSphere( PxVec3(-1.0f, 0.0f, 0.0f), 0.5f ),
    PxClothCollisionSphere( PxVec3( 1.0f, 0.0f, 0.0f), 0.25f )
};
cloth.setCollisionSpheres(spheres, 2);
cloth.addCollisionCapsule(0, 1);
```

Planes can be added through *PxCloth::addCollisionPlane()* methoconsidered for collision unless they are referenced by a convex shape following code shows how to setup a typical upward facing ground origin:

```
cloth.addCollisionPlane(PxClothCollisionPlane(PxVec3(0.0f, 1.0f, cloth.addCollisionConvex(1 << 0); // Convex references the first</pre>
```

Planes may be efficiently updated after construction PxCloth::setCollisionPlanes() function.

Finally, triangles are added using the *PxCloth::setCollisionTriangl* example, the following code adds a tetrahedron made of four triangles:

Note: The snippet adds collision convex and capsule once in the *cree* and then updates collision spheres, planes and triangles every frame in function.

Adding Virtual Particles

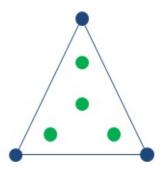


Figure 9. Four virtual particles (green) expressed as the weighted contribution triangle's particles, virtual particles provide a better sampling of the classical improves collision detection.

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 from a linear combination of the referenced particles. The following is a table that can be used to create a distribution of 4 virtual particles on a t

```
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 virtua 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;</pre>
```

```
// 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 the back any particle data. The *lockParticleData()* provides read and option the particle positions of the current and previous iteration. As an exa code applies some external acceleration to each part setParticleAccelerations():

```
PxClothParticleData* data = cloth.lockParticleData(PxDataAccessFl
float dt = cloth.getPreviousTimeStep();
for(PxU32 i = 0, n = cloth.getNbParticles(); i < n; ++i)
{
    data->previousParticles[i].pos -= particleAccelations[i] * dt
}
data->unlock();
```

References

- [1] Mueller, Matthias and Heidelberger, Bruno and Hennix, Marcus Position based dynamics. Academic Press, Inc., p. http://dx.doi.org/10.1016/j.jvcir.2007.01.005
- [2] Kim, Tae-Yong and Chentanez, Nuttapong and Mueller-Fischer, Mattachments a method to simulate inextensible clothing in Eurographics Association. p. 305--310 2012 http://dl.acid=2422356.2422399

NVIDIA PhysX SDK 3.4.1 Documentation » previous | next User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

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 interiormation directly into the application's view. For that purpose, F interface to extract visual debug information as a set of basic rendering points, lines, triangles and text. These primitives can then be rendered the application render objects.

Usage

To enable debug visualization, the global visualization scale has to be value first:

```
PxScene* scene = ...
scene->setVisualizationParameter(PxVisualizationParameter::eSCALE
```

Then the individual properties that should be visualized can be enab positive value:

```
scene->setVisualizationParameter(PxVisualizationParameter::eACTOR
```

In the example, the actor world axes will be visualized. The scale used be the product of the global scale (1.0 in this example) and the proper example). Please note that for some properties the scale factor does geometry, for example, will not be scaled since the size is defined by the Furthermore, for some objects, visualization has to be enabled corresponding object instances too (see *PxActorFlag: PxShapeFlag::eVISUALIZATION*, ...).

After a simulation step, the visualization primitives can then be extracted

```
const PxRenderBuffer& rb = scene->getRenderBuffer();
for(PxU32 i=0; i < rb.getNbLines(); i++)
{
    const PxDebugLine& line = rb.getLines()[i];
    // render the line
}</pre>
```

Note: Do not extract render primitives while the simulation is running.

The amount of debug visualization data might be too vast to create scenes. In cases where only a localized area is of interest, there is t culling box for debug visualization via *PxScene::setVisualizationCullingI*

Note that simply enabling debug visualization (PxVisualizationParam have a significant performance impact, even when all the other indiflags are disabled. Thus, make sure debug visualization is disabled it builds.

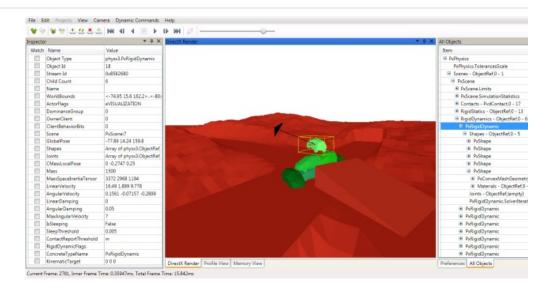
NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

PVD



The PhysX Visual Debugger (PVD) provides a graphical view of the includes various tools to inspect and visualize variables of eve Additionally it can also record and visualize memory and timing data.

PVD can be downloaded from: http://supportcenteronline.com/ics/s 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 configu In order to reduce memory footprint and code size, it is not enab configuration.

The SDK outputs the PVD debugging data in form of a stream. PVD sustream 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 convenient way to collect PVD data. In this mode the stream can be we depending on network speed and scene complexity. In network more TCP/IP server and must therefore be launched before the SDK tries to default listening port is 5425:

```
use namespace physx;

PxPvd* pvd = PxCreatePvd(*foundation);
PxPvdTransport* transport = PxDefaultPvdSocketTransportCreate(PVD pvd->connect(*transport,PxPvdInstrumentationFlag::eALL);

PxPhysics* physics = PxCreatePhysics(PX_PHYSICS_VERSION, *gFounda

//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 reco in case your platform or system setup does not support a network conn streams are often faster than network sockets and therefore a performance is more important than real-time viewing. Streams store

loaded by drag&drop or over the File->Load menu in PVD:

```
use namespace physx;

PxPvd* pvd = PxCreatePvd(*foundation);
PxPvdTransport* transport = PxDefaultPvdFileTransportCreate(filen pvd->connect(*transport,PxPvdInstrumentationFlag::eALL);

PxPhysics* physics = PxCreatePhysics(PX_PHYSICS_VERSION, *gFounda

//After releasing PxPhysics, release the PVD physics->release();
pvd->release();
transport->release();
```

Advanced Setup

Connection Flags

To optimize the stream size we provide flags to enable specific featurinfluence on PVD's and the SDK's performance:

- PxPvdInstrumentationFlag::eDEBUG: Transfer all debug data inspect objects. This flag has usually the biggest impact on the stre
- PxPvdInstrumentationFlag::ePROFILE: Transfer timing inforr profiling zones in our SDK.
- PxPvdInstrumentationFlag::eMEMORY: Transfer memory usage

Setup to transfer only profiling data over network:

```
pvd->connect(*transport, PxPvdInstrumentationFlag::ePR0FILE);
```

Visualizing Externals and Extended Data

Joints are implemented as an extension to the SDK constraints and the handling to get transmitted to PVD. Both joint and contact data can ir size significantly. Visualizing it in PVD is therefore disabled by default. I following API calls:

```
mScene->getScenePvdClient()->setScenePvdFlags(PxPvdSceneFlag::eTR
```

or set the flags separately:

```
mScene->getScenePvdClient()->setScenePvdFlag(PxPvdSceneFlag::eTRA
```

Visualizing SceneQuery

Visualizing SceneQuery in PVD is disabled by default since queries increase the stream size significantly. To enable it use following API call

```
mScene->getScenePvdClient()->setScenePvdFlag(PxPvdSceneFlag::eTRA
```

Custom PvdClient

Implement the PvdClient interface if your application needs to react the disconnection from PVD, or if you plan to send custom PVD events from the second to the contact and constraint visionPvdConnected/onPvdDisconnected callbacks to avoid potential memoverhead in the SDK:

```
// 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 represe
        // 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
    //impleament 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 connect can call Ps::Foundation::error() or Ps::Foundation::getErrorHandler report your error message. These functions will send error mautomatically.

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 h PxPvd::connect() call, and it overrides the potentially already existin That is, if users call PxSetProfilerCallback() with their own user profiler initialize PVD with PxPvdInstrumentationFlag::ePROFILE, then the user lost. Similarly, initializing PVD first then calling PxSetProfilerCallback() profiling results vanish.

In case both PVD's internal profiling and a user's custom profiling are n time, it is recommended to initialize PVD first, then call PxSetProfilerC own profiler. In your implementation, call the PVD profiling functions r after performing your own profiling operations:

```
// Then re-route to PVD implementation
mPvd->zoneEnd(profilerData, eventName, detached,
}
};
```

This is illustrated in SnippetCustomProfiler.

NVIDIA PhysX SDK 3.4.1 Documentation » previous | next User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Interface

In this chapter we will have a quick look at the statistics information to every simulation step. Usually, this information can be explored in Debugger but we do offer a PhysX API method as well to allow applica data directly. After a simulation step and a call to *PxScene::fetchResu* statistics for the processed step can be retrieved *PxScene::getSimulationStatistics()* interface. The method copies the provided *PxSimulationStatistics* structure. For details about the individual refer to the API documentation.

Note: Do not fetch the simulation statistics while the simulation is run

Usage

The provided simulation statistics is mainly meant to help investigate p It provides a quantitative summary of the work done, i.e., the nur combination of objects which have been processed in the current si example, if you encounter performance spikes in certain frames, tl statistics might give some insight into possible causes. For instance:

- Has a large amount of volumes been added or removed from the single step? You could try to distribute the addition/removal of object simulation steps or maybe there is a particle system in the scene very small.
- Are there suddenly many more collision pairs processed than explose caused by a badly configured collision pair filter or maybe some been accidentally raised.
- etc.

Please keep in mind that the simulation statistics are currently less what the scene contains but rather what got processed. So it is only detect whether objects have been configured and arranged properly.

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

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 - Re corresponds to the PhysX API. It is therefore suitable for manumodification for debugging purposes. It offers platform independence a loading data that was serialized with a previous PhysX SDK version. AF is not expected to be used in performance critical situations.

The binary serialization approach on the other hand supports instance objects directly from memory without copying data. This in-place deservel suited for performance critical real time situations. However, this approvide as the binary format is specific to a given platform and PhysX S provides functionality to convert binary serialized data from authoring platforms to ease the asset management.

Note: *cooking* also generates a binary output stream. The primary pu however, is to translate from a user format to a format suitable for the 5 so it is not considered a serialization mechanism. Loading a cooked me involves allocation and endian conversion. As a consequence, it is muchan PhysX' binary serialization mechanism. See *Shapes* for more details.

The following documentation will discuss how to use both serialization show how to build collections of PhysX objects and how these collections and deserialized. Further it will show how dependencies to other application side objects can be re-established when deserializing.

PhysX also supports extending serialization to custom types, such as This is described in more detail in Section *Extending Serialization*.

First Code

The following code creates and serializes a rigid dynamic using both formats:

```
// Create a material, a shape and a rigid dynamic
PxSphereGeometry geometry(1.0f);
PxMaterial* material = PxGetPhysics().createMaterial(0.0f, 0.0f,
PxShape* shape = PxGetPhysics().createShape(geometry, *material);
PxTransform t = PxTransform(PxIdentity);
PxRigidDynamic* dynamic = PxCreateDynamic(PxGetPhysics(), t, geom
PxSerializationRegistry* registry = PxSerialization::createSerial
// Create a collection and all objects for serialization
PxCollection* collection = PxCreateCollection();
collection->add(*dynamic);
PxSerialization::complete(*collection, *registry);
// Serialize either to binary or RepX
PxDefaultFileOutputStream outStream("serialized.dat");
// Binary
    PxSerialization::serializeCollectionToBinary(outStream, *coll
//~Binary
// RepX
    PxSerialization::serializeCollectionToXml(outStream, *collect
//~RepX
```

Most operations related to serialization require an instance of *PxSe* which provides information on how to serialize PhysX types. In order to object, it needs to be added to a *PxCollection*. If an object has deper PhysX objects, they need to be serialized as well. *PxSerialization::cor* required objects to the collection.

The following code deserializes the rigid dynamic and adds it to a scene

```
PxSerializationRegistry* registry = PxSerialization::createSerial
```

```
// Binary
    // Open file and get file size
   FILE* fp = fopen("serialized.dat", "rb");
   fseek(fp, 0, SEEK_END);
    unsigned fileSize = ftell(fp);
    fseek(fp, 0, SEEK_SET);
   // Allocate aligned memory, load data and deserialize
   void* memory = malloc(fileSize+PX_SERIAL_FILE_ALIGN);
   void* memory128 = (void*)((size_t(memory) + PX_SERIAL_FILE_AL
    fread(memory128, 1, fileSize, fp);
    fclose(fp);
    PxCollection* collection = PxSerialization::createCollectionF
//~Binary
// RepX
    // Load file and deserialize collection - needs cooking libra
    PxDefaultFileInputData inputData("serialized.dat");
    PxCollection* collection = PxSerialization::createCollectionF
//~RepX
scene->addCollection(*collection);
```

When deserializing a binary serialized collection, the data first needs memory block that is aligned to 128 bytes. The memory block may 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 resu be added to the scene with *PxScene::addCollection*.

In-depth Discussion

Collections

The serialization system makes use of a class *PxCollection*, which man objects deriving from *PxBase*. Each collection represents a set of c maintain a mapping between IDs of type *PxSerialObjectId* and object IDs may be defined by the application. One caveat here is that the II within a collection, but do not have to be unique across different collection required by the application, it is the application's responsibility to ensure

Here is an example of how to iterate over a collection, for instance objects intended for serialization have all been added to the collectic PhysX' dynamic typing mechanism can be used to classify the objects:

```
PxCollection* collection;
PxU32 size = collection->getNbObjects();
for(PxU32 i=0; i<size; i++)
{
    PxBase* object = collection->getObject(i);
    if(!object->is<PxActor>())
        continue;

    switch((PxConcreteType)object->getConcreteType())
    {
        case PxConcreteType::eRIGID_DYNAMIC:
        ...
    }
}
```

Note: In order to simplify releasing object within a collection, PhysXE: contains a function to remove and release all objects from a collection: *PxCollectionExt::releaseObjects*.

Note: Releasing an object within a collection invalidates the mapping objects.

A collection is said to be *complete* if no contained objects depend on ϵ the collection. For example, an actor, a shape with a box geometry, and shape would together form a complete collection. The same collection ι would be incomplete.

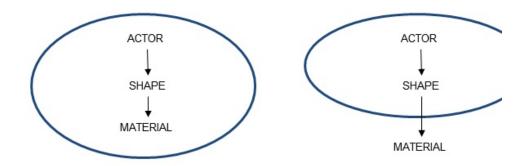


Figure 1: Left: Complete Collection, Right: Incomplete Coll

For a formal definition please refer to *Complete*.

Both complete and incomplete collections can be serialized, but whe incomplete collection, references to objects which were not serialize resolved. The following two sections describe how PhysX collections can deserialized using the binary format or RepX. The first section shows complete collections, and the second section shows how to deacollections.

Serializing Complete Collections

This code snippet shows how to prepare a collection of PhysX objection (e.g. an actor, its shapes, and the materials and meshes they reference

```
PxPhysics* physics;
PxRigidDynamic* dynamic = PxCreateDynamic(...);

// Cre
PxSerializationRegistry* registry = PxSerialization::createSerial

PxCollection* collection = PxCreateCollection();
// Cr
collection->add(*dynamic);
// Ad
```

Instead of using *PxSerialization::complete* it is possible to manuall required for serialization. All objects the *PxRigidDynamic* references added and then all objects referenced by the newly added objects would as well and so forth. See definitions: *Requires*, *Complete*.

By default *PxSerialization::complete* follows references from joints to t from actors to their joints. The *followJoint* parameter can be used to che of *PxSerialization::complete* to add the joints attached to each actor. The actor-joint chains to be added to the collection.

When all the necessary objects have been added to a collection, create of the PxOutputStream interface, then serialize the collection:

```
PxColletion* collection;
                                                           // Comp
PxSerializationRegistry* registry;
                                                           // Regi
                                                           // Impl
PxOutputStream& outStream = ...;
// Serialize
// Binary
    PxSerialization::serializeCollectionToBinary(outStream, *coll
//~Binary
// RepX
    PxSerialization::serializeCollectionToXml(outStream, *collect
//~RepX
// Collection and registry can be released if they are no longer
// Note that releasing the collection will not release the contai
collection->release();
registry->release();
```

Note: Serialization of objects in a scene that is simultaneously being supported and leads to undefined behavior.

The following code shows how to deserialize a collection from a memor

```
PxSerializationRegistry* registry;
                                                           // Regi
PxCooking* cooking;
                                                           // Cook
                                                           // inst
// Deserialize
// Binary
   void* memory128 = ...;
                                                           // A 12
                                                           // load
    PxCollection* collection = PxSerialization::createCollectionF
//~Binary
// RepX
    PxInputData& inputData = ...;
                                                           // Impl
    PxCollection* collection = PxSerialization::createCollectionF
//~RepX
```

To add all the objects to the scene and release the collection and registi

See *Serializable* for the exact set of conditions a collection must sat serialized. These conditions can be checked with *PxSerialization::isSerialization:*

Serializing Incomplete Collections

Another common use case is where a collection of actors and joints - see the deserialized multiple times, with each instance sharing the sa 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 deserial

Collection B is *incomplete*, since it contains references to objects in A. In the serialized format will remember each reference to an object in A us (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 resolved. Although collection B is incomplete, it is also said to be collection A. For a formal definition of complete please refer to *Complete*.

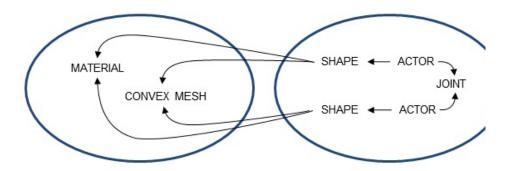


Figure 2: Left: Collection *A* with Sharable Objects, Right: Collection *B* Concretely, to serialize and deserialize an incomplete collection:

- At serialization time, provide IDs for all objects in collection A that objects in collection B.
- When deserializing, provide a collection with matching IDs for all tl were referenced by objects in B.

Here are examples of how the application can provide identities (*P* express requirements of one collection to another. This can be do adding the object with:

```
PxCollection* collection;
PxTriangleMesh* triMesh;
PxSerialObjectId triMeshId = 1;

collection->add(*triMesh, triMeshId);
```

Or set the ID after adding the object:

```
collection->add(*triMesh);
collection->addId(*triMesh, triMeshId);
```

There is a helper function to generate IDs for all objects in a collectio IDs yet:

```
PxSerialObjectId baseId = 1;
PxSerialization::createSerialObjectIds(*collection, baseId);
```

Already used ID values will be skipped by *createSerialObjectIds*, as valready have IDs.

After providing correct IDs, all required objects have been added to t serialized, but without adding the objects that are intended to be referer function in *PxSerialization* supports completing a collection relative to a

```
PxSerializationRegistry* registry;

PxCollection* collectionB;
PxCollection* collectionA;

PxSerialization::complete(*collectionB, *registry, collectionA);
```

Serialization example:

```
// Add convexes to collection
for(PxU32 i=0;i<mNbConvexes;i++)</pre>
    convexCollection->add(*convexes[i]);
// Create IDs for the convexes, starting with 1
PxSerialization::createSerialObjectIds(*convexCollection, PxSeria
// Serialize the convexes along with their IDs
// Binary
   PxSerialization::serializeCollectionToBinary(convexStream, *co
//~Binary
// RepX
   PxSerialization::serializeCollectionToXml(convexStream, *conve
//~RepX
// Add actors to other collection
for(PxU32 i=0;i<mNbActors;i++)</pre>
    actorCollection->add(*actors[i]);
// Add all required objects except the convexes
PxSerialization::complete(*actorCollection, *registry, convexColl
// Serialize the actors with references to convexCollection
// Binary
    PxSerialization::serializeCollectionToBinary(actorStream, *ac
                                                  convexCollection
//~Binary
// RepX
    PxSerialization::serializeCollectionToXml(actorStream, *actor
                                               convexCollection);
//~RepX
// Release collections and registry
convexCollection->release();
actorCollection->release();
registry->release();
```

Deserialization example:

```
PxPhysics* physics; // The physics SDK ob
PxSerializationRegistry* registry // Registry for seria
```

```
PxCooking* cooking;
                                            // Cooking lib needed
PxScene* scene;
                                            // The scene into whi
// Deservalize convexes along with their IDs (no external depende
// Binary
   void* convexMemory128;
                                            // Aligned memory con
   PxCollection* convexCollection =
        PxSerialization::createCollectionFromBinary(convexMemory1
//~Binarv
// RepX
    PxInputData& convexInputData = ...; // Implemented by the
    PxCollection* convexCollection =
        PxSerialization::createCollectionFromXml(convexInputData,
//~RepX
// Deserialize actors referencing the convexCollection
// Binary
   void* actorMemory128;
                                            // Aligned memory con
    PxCollection* actorCollection =
       PxSerialization::createCollectionFromBinary(actorMemory12
//~Binarv
// RepX
    PxInputData& actorInputData = ...; // Implemented by the
    PxCollection* actorCollection =
        PxSerialization::createCollectionFromXml(actorInputData,
                                                 convexCollection
//~RepX
// Release convex collection
convexCollection->release();
// Add actors to scene and release collection and registry
scene->addCollection(*actorCollection);
actorCollection->release();
registry->release();
```

The next example shows how to deal with situations where the serialized objects that are not serialized and deserialized but created by other meaning the serial objects.

```
PxSerializationRegistry* registry; // Registry for serializable PxMaterial** materials; // Created procedurally by ap
```

```
PxRigidDynamic** actors; // An array of mNbConvexes ac PxOutputStream& actorStream; // Output stream for the acto
// Add materials with IDs to collection
PxCollection* materialCollection = PxCreateCollection();
for(PxU32 i=0;i<mNbMaterials;i++)</pre>
    materialCollection->add(*materials[i], PxSerialObjectId(i+1))
// Create actor collection, complete and serialize
PxCollection* actorCollection = PxCreateCollection();
for(PxU32 i=0;i<mNbActors;i++)</pre>
    actorCollection->add(*actors[i]);
PxSerialization::complete(*actorCollection, *registry, materialCo
// Binary
    PxSerialization::serializeCollectionToBinary(actorStream, *ac
                                                    materialCollecti
//~Binary
// RepX
    PxSerialization::serializeCollectionToXml(actorStream, *actor
                                                 materialCollection)
//~RepX
actorCollection->release();
                                  // Note that materialColl
materialCollection->release();
registry->release();
```

Deserialization:

```
// Binary
   void* actorMemory128;
                                         // aligned memory contai
    PxCollection* actorCollection =
        PxSerialization::createCollectionFromBinary(actorMemory12
        materialCollection);
//~Binary
// RepX
    PxInputData& actorInputData = ...; // Implemented by the ap
    PxCollection* actorCollection =
        PxSerialization::createCollectionFromXml(actorInputData,
                                                 materialCollecti
//~RepX
materialCollection->release();
scene->addCollection(*actorCollection);
actorCollection->release();
registry->release();
```

Reference Counting of Deserialized Objects

This section assumes the background in *Reference Counting*.

Objects that are created by deserialization are always created with a application needs to give up by explicitly calling *release()*. The inforr application gave up a reference to an object is **not** preserved on serializ

See *Shapes* for a discussion of the method *PxRigidActorExt::crea* which automatically releases the initial reference to the shape, leaving reference. Again, the information that this reference has been released serialization.

Example for shapes:

```
PxOutputStream& outStream; // Output stream for the coll PxSerializationRegistry* registry; // Registry for serializable PxRigidActor* actor; // Any actor

// Creating shapes in different ways implies different rules for
```

```
// Shape is automatically released when actor gets released
PxShape* shapeA = PxRigidActorExt::createExclusiveShape(*actor, .
// Shape is either created as "shared" or "exclusive" and needs t
// the application
PxShape* shapeB = PxGetPhysics().createShape(...);
actor->attachShape(*shapeB);
// Create collection with actor and shapes and serialize
PxCollection* collection = PxCreateCollection();
collection->add(*actor);
collection->add(*shapeA);
collection->add(*shapeB);
PxSerialization::serializeCollectionToBinary(outStream, *collecti
collection->release();
// Releasing actors and shapes
actor->release();  // Releases actor and shapeA (automatically)
shapeB->release();  // Releases shapeB (necessary since shapeB w
// Deserialize collection
void* memory128 = ...; // Aligned memory for serialized data
collection = PxSerialization::createCollectionFromBinary(memory12
// Release actors and release ALL shapes (necessary since shape c
// not preserved across serialization
for(PxU32 i = 0; i < collection->getNbObjects(); i++)
{
    switch ( collection->getObject(i).getConcreteType() )
    {
        case PxConcreteType::eRIGID DYNAMIC:
        case PxConcreteType::eRIGID_STATIC:
            static_cast<PxActor&>(collection->getObject(i)).relea
            break;
        case PxConcreteType::eSHAPE:
            static cast<PxShape&>(collection->getObject(i)).relea
            break;
    }
}
```

Note: There is a PhysXExtensions function to release all objects with *PxCollectionExt::releaseObjects*.

Reconnecting PhysX and Game-Objects

Here is an example of how to fix up references with gameplay objects I of a collection:

```
PxPhysics* physics;
                                        // The physics SDK object
PxCooking* cooking;
                                        // Cooking library needed
PxSerializationRegistry* registry;
                                       // Registry for serializa
// Deserialize objects along with IDs
// Binary
   void* memory128;
                                       // Aligned memory contain
    PxCollection* collection =
        PxSerialization::createCollectionFromBinary(memory128, *r
//~Binary
// RepX
    PxInputData& inputData = ...;  // Implemented by the ap
    PxCollection* collection =
        PxSerialization::createCollectionFromXml(actorInputData,
                                                 materialCollecti
//~RepX
// Receive a list of all deserialized IDs
#define MAX IDS 100
PxSerialObjectId idBuffer[MAX_IDS];
PxU32 numIds = collection->getIds(idBuffer, MAX IDS);
// iterate over the list to patch up gameplay objects
for (PxU32 i = 0; i < numIds; i++)
{
    PxActor* actor = collection->find(idBuffer[i])->is<PxActor>()
    if (actor)
        // this assumes that findGamePlayObjectFromId is able to
        // the corresponding game play object from a PxSerialObje
        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 PxCollectionExt::createCollection(PxPhysics& sdk)
PxCollectionExt::createCollection(PxScene& scene):

```
PxPhysics* physics;
                                   // The physics SDK object
PxScene* scene;
                                   // The physics scene
PxSerializationRegistry* registry; // Registry for serializable
PxOutputStream& outStream;
                                 // The user stream doing the
// 1) Create a collection from the set of all objects in the phys
    multiple scenes.
PxCollection* everythingCollection = PxCollectionExt::createColle
// 2) Create a collection from all objects in the scene and add i
    to everythingCollection.
PxCollection* collectionScene = PxCollectionExt::createCollection
everythingCollection->add(collectionScene);
collectionScene->release();
// 3) Complete collection
PxSerialization::complete(*everythingCollection, *registry);
// 4) serialize collection and release it
// Binary
    PxSerialization::serializeCollectionToBinary(outStream, *ever
//~Binary
// RepX
    PxSerialization::serializeCollectionToXml(outStream, *everyth
//~RepX
everythingCollection->release();
registry->release();
```

Deserialization is as previously:

```
PxScene* scene; // The physics scene
PxCooking* cooking; // Cooking library needed fo
PxSerializationRegistry* registry; // Registry for serializable
```

Serializability

This section contains various definitions to describe serializability of a a collection can be successfully serialized and deserialized, optionally references collection, can be queried by calling *PxSerialization::isSerial*

Requires

An object **A** requires another object **B** if **A** maintains a reference to **B** the established for successfully deserializing **A**. This implies that **B** needs before **A**.

Here is the table of the relationship **requires** of all PhysX objects:

joints	require their actors and constraint
rigid actors	require their shapes
shapes	require their materials and mesh (triangle mesh, conve field), if any
articulations	require their links and joints
aggregates	require their actors
cloth actors	require their cloth fabric

Subordinate

Subordinates are objects that cannot be instantiated without being objects. An articulation link, for example, can only be instantiated as particulation.

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 describilized 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

Memory Management

Management of memory blocks containing deserialized objects is left user's responsibility to:

- allocate the memory block. Note that it must be proper PX_SERIAL_FILE_ALIGN (128) bytes boundary.
- fill the block with serialized data, typically by loading it from disk.
- deallocate the memory block when the objects within have been rel

Although the user owns the memory block, the PhysX runtime own objects it contains. Concretely, calling release() on an object tha deserialization will cause its destructor to run, but will not deallocate block is deallocated before the destructors have run for all the object PhysX runtime will likely crash. For more information about how deseries to be released see *Reference Counting of Deserialized Objects*.

Versioning

The binary serialized data is typically specific to the version of the SE with. However, a SDK version can load the data of older SDK versions didn't change. This is usually the case with bugfix releases. The compare listed in the code documentation of *PX_BINARY_SER PxSerialization.h*.

Retargeting to other Platforms

Binary serialized data is platform-specific, and when serialized it a platform on which it was created. The binary converter in the extensio data from one platform to another. Typically assets are serialized on an (Windows, Mac OS X and Linux). The serialized data can then be retarg to a console or any other runtime platform.

The converter requires meta-data for the source and target platforn

information about the binary layout of objects for that platform. To ob the function provided in the extensions library for each platform:

```
void PxSerialization::dumpBinaryMetaData(PxOutputStream& stream,

◀
```

On each target platform, run it once and keep generated data around. of pre-built binary metadata is included with the PhysX SDK at [path SDK]/Tools/BinaryMetaData.

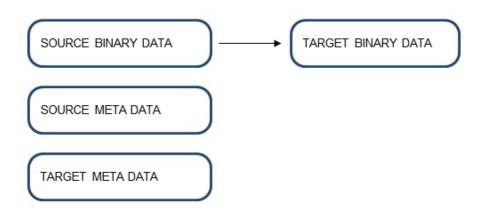


Figure 3: Schema of Retargeting

Assuming that the extensions library has been initialized, conversion follows:

```
PxSerializationRegistry* registry;
                                        // Registry for serializa
PxInputStream& srcMetadata;
                                        // metadata for the 'from
                                        // (e.g. PxDefaultFileInp
                                        // metadata for the 'to'
PxInputStream& dstMetadata;
PxInputStream& srcAsset;
                                        // stream containing sour
                                        // size of the source ass
PxU32 srcAssetSize;
PxOutputStream& dstAsset;
                                        // output stream for reta
PxBinaryConverter* converter = PxSerialization::createBinaryConve
converter->setMetaData(srcMetadata, dstMetadata);
converter->convert(srcAsset, srcAssetSize, dstAsset);
```

The convert tool is at [path to installed PhysX SDK]/Snippets/SnippetC how to convert PhysX 3 serialized binary files from one platform compiles and runs on authoring platforms (Windows, MacOs and Linux)

SnippetConvert is a simple command-line tool supporting the following (

```
--srcMetadata=<filename> Defines source metadata file
--dstMetadata=<filename> Defines target metadata file
--srcBinFile=<filename> Source binary file to convert
--dstBinFile=<filename> Outputs target binary file
--generateExampleFile=<filename> Generates an example file
--verbose Enables verbose mode
```

Object Names

Some SDK objects, such as shapes and actors, can be given PxShape::setName() and PxActor::setName() functions. By default the serialized. The 'exportNames' parameter PxSerialization::serializeCollectionToBinary() can be set to true in ordinames along with the objects.

API-level Serialization (RepX) Specifics

RepX stands for Representation X and is the ASCII-XML serialization f As opposed to binary serialization, the RepX XML serialization is not ir in performance critical or memory constrained situations. The following specifics of the RepX XML serialization system.

Upgrading RepX Data

Upgrading RepX data from an older PhysX version to a newer one i implicitly when deserializing old RepX data with a newer PhysX SDK ar resulting PxCollection.

Example for upgrading a RepX stream:

Object Names

As opposed to binary serialization, the object names that can be PxShape::setName() and PxActor::setName() functions, are alway serialized format. On deserialization with PxSerialization::createCollectinames can be recovered by setting the <math>PxStringTable parameter.

If *PxStringTable* parameter is set, the names will live within the memory by the string table. The string table must not be released unless it can the names will not be accessed any more.

Caching Cooked Geometry Data

In order to facilitate faster instantiation of XML data, it is possible to serialization to store the cooked triangle and convex mesh data along The cooked data caching can be enabled by passing a *PxCookexSerialization::serializeCollectionToXml(...)*. The cached cooked data format is incompatible with the current SDK version.

Common Use Cases

API-level RepX serialization should be used whenever compatibility and are important. The PhysX plug-ins for the DCC tools 3ds Max and I export PhysX objects. The resulting RepX files can then be deserialized the PhysX runtime. This is useful for rapid prototyping or for general assets if performance is not of a big concern. For quick loading of as convert RepX data into binary serialized data. RepX is also usef situations with unwanted behavior without the need to provide the who this, the application may be connected to the PhysX Visual Debug records the scene of interest. A representative frame can then be say from within PVD (see PVD).

Binary serialization should be used in performance and memory con The main target use-case is streaming in chunks of a large game loaded into memory at once. Creating and loading save games is anot could be optimized by using binary serialization. PhysX objects in bina be sent over the network to enable efficient game state synchronization

Snippet Discussion

The following snippets illustrate common operations such as male serialization, deserialization and re-targeting of binary data.

SnippetSerialization

SnippetSerialization shows binary and XML serialization of a scene jointed rigid bodies representing a chain. This is done in a way that allow of multiple chains while sharing the shape and the material across all constants shows how to create and populate collections, specify IDs to dependencies, serialize collections, deserialize collections and add actual simulation.

The snippet also shows how to allocate a data block aligned to demonstrates how to copy binary serialized data into it. It further der data blocks containing the binary deserialized collections must be material corresponding objects are not needed anymore and have been released

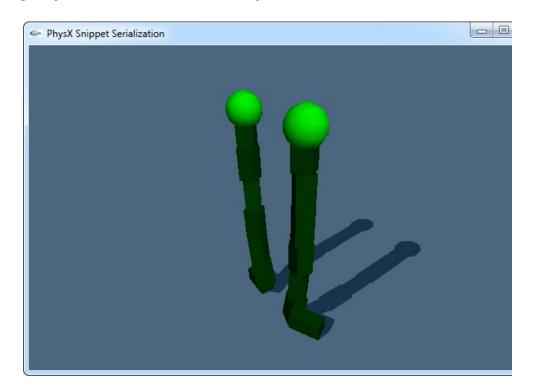


Figure 4: SnippetSerialization

SnippetConvert

SnippetConvert illustrates how binary serialized data can be re-targeted platform to a runtime platform such as a console. The snippet is a sin tool that can load a binary serialized data file along with meta data file and destination platforms and then output a converted binary data file 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 Debugger application and display the content of serialized collect snippet's source documentation for more details.

Best practices / Troubleshooting

- Concurrent simulation and serialization is not supported and leading.
- If releasing PhysX objects leads to crashes or errors it is possible is releasing some objects twice. The following two reasons should A potential source of error is to release PhysX objects without u referencing these objects. 2.) Shapes that where created through application reference automatically released on creation. If such a and deserialized the creation history will be lost. It might be con extension function *PxCollectionExt::releaseObjects* because it dea cases as required. See *Reference Counting of Deserialized Objects*
- If accessing binary deserialized PhysX objects, including accesses causes crashes it might be due to the premature release of the holds the deserialized objects.
- If binary files are too large and/or too slow to load it might be that a been serialized multiple times. An example of a shared asset might referenced by multiple shapes. The solution is to separate shared a separate collection. See *Serializing Incomplete Collections*.
- If loading PhysX objects from RepX files is too slow two things should be used instead? Even for debugg sense to convert RepX files into binary serialized data by re-serial binary approach. 2.) Meshes tend to load very slowly from serialization offers an option to cache cooked mesh data by in-linithe RepX file. If such a cache is present and valid, the load significantly faster. See Caching Cooked Geometry Data.

PVD

The PhysX Remote Debugger provides the functionality to export single scenes as RepX files. The resulting files can be used to playback a sna state. In many cases this is sufficient to isolate an issue. The option comenu of PVD: [Menu > File > Export Current Frame To RepX]

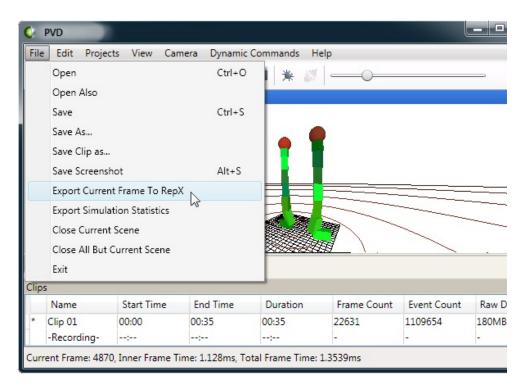


Figure 5: RepX Functionality in PVD

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Introduction

The PhysX serialization system (*Serialization*) is extendable to cu application were to require a new joint type, for example, the serialization extended to add support for serialization of that new joint type.

The following document contains some recipes and example code that serialization may be extended to custom types. It doesn't cover extension mechanisms. It is therefore advisable to look into the follow example for more details:

PhysXVehicle library (PhysXVehicle/src)

Overview

Both binary and RepX serialization can be extended for custom type custom type for serialization it must first inherit from *PxBase*. This allow custom type to be added to a *PxCollection*, which is a pre-requisite for core serialization functionality needs to be provided by implementing interface. The template *PxSerializerDefaultAdapter* provides a default in can be specialized for the custom type as required. In order to support an additional *PxRepXSerializer* interface needs to be implemented. The relies on automatic code generation using clang. Scripts to run the code examples can be found in (Tools/PhysXMetaDataGenerator).

Binary Serialization of Custom Classes

Serialization and deserialization of a custom class can be achieved steps:

- 1. Define a *PxConcreteType* and type info for the custom class. Make is unique.
- 2. The custom class needs to inherit from *PxBase* and implement it's
- Instance PxSerializerDefaultAdapter<T> and implement specialized necessary.
- 4. If retargeting to other platforms is needed, implement *getBinaryMet*
- 5. Register the adapter and metadata, see *PX_NEW_SERIALI PxSerializationRegistry::registerSerializer*
 - PxSerializationRegistry::registerBinaryMetaDataCallback. Note the need to be unregistered before PxSerializationRegistry::release application is responsible for custom type serializer allocation and continuous continuous application.

For pointer members the following needs to be done (Note that reference currently not supported):

- 6. Implement *PxSerializer::requires*. It should enumerate *PxBase* ob object depends for deserialization. See *Requires*.
- 7. For a member pointer to another *PxBase* object, register the implementation of *PxSerializer::registerReferences*. The in *PxSerializer::requires* may be used to help with this.
- 8. Resolve references in the implementation of *PxSerializer::c PxDeserializationContext::resolveReference*, *translatePxBase*.
- 9. Make sure that *PxSerializer::isSubordinate* returns whether the serialized along with an owner object. See *Subordinate*.
- 10. Export non PxBase data by implementing PxSerializer::expc

- PxSerializationContext::writeData, alignData.
- 11. Import non *PxBase* data in the implementation of *PxSerializer:: PxDeserializationContext::readExtraData, alignExtraData.*

Note: In checked builds (PX_CHECKED defined as 1) metadata defir against serialized data. If metadata definitions are missing warnings ar error stream during re-targeting (*PxBinaryConverter::convert*). To avoic all unused memory in custom serialized class instances should be mar pattern. This can be done with *Cm::markSerializedMem* from CmUtils.

Note: The memory of a deserialized class instance should not be deamemory is embedded in the memory buffer containing the serialized da PxBaseFlag::eOWNS_MEMORY can used to decide whether the object be deallocated or not.

Example for a custom class:

```
#include "extensions/PxSerialization.h"
#include "common/PxTypeInfo.h"
#include "common/PxMetaData.h"
#include "common/PxSerializer.h"
#include "common/PxSerialFramework.h
using namespace physx;
const PxType customClassType = PxConcreteType::eFIRST_USER_EXTENS
PX_DEFINE_TYPEINFO(CustomClass, customClassType);
class CustomClass : public PxBase
    friend class PxSerializerDefaultAdapter<CustomClass>;
public:
    // constructor setting up PxBase object
    CustomClass()
    : PxBase(customClassType, PxBaseFlag::eOWNS_MEMORY | PxBaseFl
    {}
    // constructor called on deserialization
    CustomClass(PxBaseFlags baseFlags) : PxBase(baseFlags) {}
```

```
virtual ~CustomClass() {}
    //PxBase
    virtual const char* getConcreteTypeName() const { return "Cus
    virtual bool isKindOf(const char* name) const
        return !strcmp("CustomClass", name) || PxBase::isKindOf(n
    //~PxBase
    //PxSerializationRegistry::registerBinaryMetaDataCallback
    static void getBinaryMetaData(PxOutputStream& stream)
        PX_DEF_BIN_METADATA_VCLASS(stream, CustomClass)
        PX_DEF_BIN_METADATA_BASE_CLASS(stream, CustomClass, PxBas
        PX DEF BIN METADATA ITEM(stream, CustomClass, PxRigidDyna
            PxMetaDataFlag::ePTR)
        PX_DEF_BIN_METADATA_ITEM(stream, CustomClass, char, mBuf,
        PX_DEF_BIN_METADATA_ITEM(stream, CustomClass, PxU32, mSiz
        PX DEF BIN METADATA EXTRA ITEMS(stream, CustomClass, char
    //~PxSerializationRegistry::registerBinaryMetaDataCallback
private:
    PxRigidDynamic* mActor; //add in requires
    char* mBuf;
                              //extra data
                              //size of mBuf
   PxU32 mSize:
};
//PxSerializerDefaultAdapter
template<>
void PxSerializerDefaultAdapter<CustomClass>::requires(PxBase& ob
                                                       PxProcessP
{
   CustomClass* custom = obj.is<CustomClass>();
   PX_ASSERT(custom);
    c.process(*custom->mActor);
}
template<>
void PxSerializerDefaultAdapter<CustomClass>::registerReferences(
```

```
{
    CustomClass* custom = obj.is<CustomClass>();
    PX_ASSERT(custom);
    s.registerReference(obj, PX_SERIAL_REF_KIND_PXBASE, size_t(&o
    s.registerReference(*custom->mActor, PX_SERIAL_REF_KIND_PXBAS
}
template<>
void PxSerializerDefaultAdapter<CustomClass>::exportExtraData(PxB
                                                               PxS
{
    CustomClass* custom = obj.is<CustomClass>();
    PX_ASSERT(custom);
    s.alignData(PX_SERIAL_ALIGN);
    s.writeData(custom->mBuf, custom->mSize);
}
template<>
PxBase* PxSerializerDefaultAdapter<CustomClass>::createObject(PxU
                                                               con
{
    CustomClass* custom = new (address) CustomClass(PxBaseFlag::e
    address += sizeof(CustomClass);
    // resolve references
    context.translatePtr(custom->mActor);
    // import extra data
    custom->mBuf = context.readExtraData<char*, PX_SERIAL_ALIGN>(
    // return deserialized object
    return custom;
//~PxSerializerDefaultAdapter
void registerCustomClassBinarySerializer(PxSerializationRegistry&
{
    registry.registerSerializer(customClassType, PX NEW SERIALIZE
    registry.registerBinaryMetaDataCallback(CustomClass::getBinar
}
void unregisterCustomClassBinarySerializer(PxSerializationRegistr
{
    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 C*PxSerializer and PxSerializerDefaultAdapter<T> required exclusive serialization may be left empty.
- 2. Create a custom RepX serializer that implements the *PxRepXS 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. No 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: *Ph. System*.

Example for a custom class:

```
// when using the PhysX API metadata system readAllProper
        // all properties automatically
        readAllProperties(inArgs, inReader, object, inAllocator,
        return PxCreateRepXObject(object);
    }
    virtual void objectToFileImpl(const CustomClass* obj, PxColle
                                  XmlWriter& inWriter, MemoryBuff
                                  PxRepXInstantiationArgs&)
    {
        // when using the PhysX API metadata system writeAllPrope
        // all properties automatically
        writeAllProperties(obj, inWriter, inTempBuffer, *inCollec
    }
    // this can return NULL if fileToObject(...) is overwritten w
    virtual CustomClass* allocateObject(PxRepXInstantiationArgs&)
};
void registerCustomClassRepXSerializer(PxSerializationRegistry& r
{
    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(c
}
```

Note: Implementing a PxRepXSerializer is currently not practical with internal PhysXExtension header "SnRepXSerializerImpl.h".

PhysX API Metadata System

This system produces a set of objects that are analogues of the

descriptors in the PhysX system, all based on the public interface heuristically finds functions that start with get/set and, through a series 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
 - o {ptype} get{pname}(enumType idx);
 - void set{pname}(enumType idx, const {ptype}& prop);
- Dual indexed property (like above, but with two enumeration indexe
- 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 following recipe:

- CustomTypeExtensionAPI.h
 - Add all the types that should be exported to gUserPhysXTypes
 - Add the unnecessary types to gAvoidedPhysXTypes. It metadata information for these types.
 - Be sure to append the included files for these types.
- runClang_[windows|osx|linux].[bat|sh] (e.g. runClang_windows.bat)

- Set definition folder for these autogenerated files and set the s
- Specify the filename of autogenerated files. Then it will gen files:

include/CustomTypeAutoGeneratedMetaDataObjectNames.h
include/CustomTypeAutoGeneratedMetaDataObjects.h
src/CustomTypeAutoGeneratedMetaDataObjects.cpp

- CustomTypeMetaDataObjects.h
 - CustomTypePropertyInfoName has to be CustomTypeAutoGeneratedMetaDataObjects.h has to be in The file will then export the properties of the custom class ar for implementing the custom RepX serializer.
- CustomTypeMetaDataObjects.cpp
 - This file is optional. It is only required when custom properties

PxVehicle serialization is a useful example. With Source/PhysXVehicle the structure of the files is as follows:

```
src/PhysXMetaData/include/PxVehicleMetaDataObjects.h
src/PhysXMetaData/src/PxVehicleMetaDataObjects.cpp
../../Tools/PhysXMetaDataGenerator/PxVehicleExtensionAPI.h
../../Tools/PhysXMetaDataGenerator/generateMetaData.py
```

Running the script will auto-generate the following files:

src/PhysXMetaData/include/PxVehicleAutoGeneratedMetaDataObjectNam src/PhysXMetaData/include/PxVehicleAutoGeneratedMetaDataObjects.h src/PhysXMetaData/src/PxVehicleAutoGeneratedMetaDataObjects.cpp

1. PxVehicleExtensionAPI.h: The type DisabledPropertyEntry is used which do not require export. CustomProperty is for properties customized and gUserPhysXTypes is for general properties that ne

- 2. runClang_[windows|osx|linux].[bat|sh]: The target director src/PhysXMetaData, and the target name is PxVehicle.
- 3. PxVehicleMetaDataObjects.h: It defines the custom propert PxVehicleAutoGeneratedMetaDataObjects.h
- 4. PxVehicleMetaDataObjects.cpp: It implements the custom propertie

Note: The properties defined in PxVehicleAutoGeneratedMetaDataOl written to the RepX file automatically if PxVehicleMetaDataObjects.h is custom RepX serializer.

NVIDIA PhysX SDK 3.4.1 Documentation » previous | next User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

Introduction

This chapter covers a number of best practices for the PhysX SDK to ϵ and fixing frequently encountered issues.

Debugging

The PhysX SDK contains a few debugging helpers. They can be used scenes are properly set up.

Use checked builds and the error stream

The PhysX SDK has different build configurations: Debug, Checked, F make sure that the scene is properly set up without warnings or err Debug or Checked builds, and monitor the error callback. Please *Reporting* chapter for details. Note that some checks can be expensive not performed in Release or Profile builds. If the SDK silently fails or Release build, please switch to Debug or Checked builds to ensure th an uncaught error.

Visualizing physics data

Use the PhysX Visual Debugger (PVD) to see what PhysX is seeing physics data is what you expect it to be. Please refer to the *PhysX Visu* chapter for details. Note that this is only available in Debug, Checked at

Visualizing physics data (2)

An alternative to PVD is the built-in debug visualization system. Please *Visualization* chapter for details. This option is available with all build co

Limiting coordinates

Bugs in applications, or issues in content creation, can sometime placement at unexpected coordinates. We recommend PxSceneDesc::sanityBounds, to generate reports when objects are in beyond what your application expects, or when application code more unexpected positions. Note that these bounds only apply to application

coordinates, not updates by the simulation engine.

Performance Issues

The PhysX SDK has been optimized a lot in the past dot releases. I 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, F identify performance bottlenecks, please use Profile builds and PxPvdInstrumentationFlag::ePROFILE only, since enabling the othe might negatively affect performance. Please refer to the PhysX Visual chapter for details.

Use release builds for final performance tests

The PhysX SDK has different build configurations: Debug, Checked, ReRelease builds are the most optimal. If you encounter a performance other builds, please switch to Release builds and check if the problem is

Disable debug visualization in final/release builds

Debug visualization is great for debugging but it can have a signimpact. Make sure it is disabled in your final/release builds. Please *Visualization* chapter for details.

Debug visualization is very slow

Debug visualization can be very slow, because both the code gatheri and the code rendering it is usually not optimal. Use a culling box to data the SDK gathers and sends to the renderer. Please refer to the $\mathcal L$ chapter for details.

Consider using tight bounds for convex meshes

By default PhysX computes approximate (loose) bounds around con PxConvexMeshGeometryFlag::eTIGHT_BOUNDS enables smaller/tight are more expensive to compute but can result in improved simulation policy of convex objects are interacting with each other. Please refer to the for details.

Use scratch buffers

The PxScene::simulate function accepts optional scratch buffers the reduce temporary allocations and improve simulation performance. F *Simulation* chapter for details.

Use the proper mid-phase algorithm

PxCookingParams::midphaseDesc can be used to select the desired m It is a good idea to try the different options and see which one w Generally speaking the new PxMeshMidPhase::eBVH34 introduced better performance for scene queries against large triangle meshes. Geometry chapter for details.

Use the proper narrow-phase algorithm

PxSceneFlag::eENABLE_PCM enables an incremental "persistent algorithm, which is often faster than the previous implementation. P default algorithm since PhysX 3.4, but you can also try to enable it ir like 3.3.

Use the proper broad-phase algorithm

PhysX also supports two different broad-phase implementation PxSceneDesc::broadPhaseType. The different implementations have vaccharacteristics, and it is a good idea to experiment with both and fine best for you. Please refer to the *Rigid Body Collision* chapter for details broad-phases.

Use the scene-query and simulation flags

If a shape is only used for scene-queries (raycasts, etc), disable its shape is only used for simulation (e.g. it will never be raycasted a scene-query flag. This is good for both memory usage and performan 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 s lot of dynamic objects, try to increase the PxSceneDesc::dynamicTr 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. first writing the data to a file or a memory buffer, and then passing the d

The "Well of Despair"

One common use-case for a physics engine is to simulate fixed independent of the frame rate that the application is rendered at. If capable of being rendered at a higher frequency than the simulation of has the option to render the same simulation state, interpolate frame sometimes it is not possible to render the scene at a frequency high simulation frequency. At this point, the options are to either run the physical larger time-step or to simulate multiple, smaller sub-steps. The larger preferable solution because changing the size of time-steps in a physical significantly change perceived behavior. However, when using a sub-significantly change perceived behavior. However, when using a sub-significantly always be aware of the potential that this has to damage perfectives.

As an example, let's imagine a game that is running using v-sync at 60 simulating a large number of physics bodies and, as a result, the p expensive. In order to meet the 60FPS requirement, the entire frame I

within ~16ms. As already mentioned, the physics is reasonably exposenario, takes 9ms to simulate 1/60th of a second. If the game was e.g. as a result of some OS activity, saving a check-point or loading a level, we may miss the deadline for 60FPS. If this happens, we must steps in the physics to catch up the missed time in the next frame. previous frame took 50ms instead of 16ms, we must now simulate 3 si to simulate all the elapsed time. However, each sub-step takes ~9ms, we will take ~27ms to simulate 50ms. As a result, this frame also deadline for 60FPS, meaning that the frame including v-sync took 33 must now simulate 2 sub-steps in the next frame, which takes ~18ms a 16ms deadline. As a result, we never manage to recover back to 60FP our decision to sub-step as a result of a spike has resulted in our applic a performance trough indefinitely. The application is capable of simula at 60FPS but becomes stuck in the so-called "physics well of desp substepping.

Problems like this can be alleviated in several ways:

- Decouple the physics simulation from the game's update/render local physics simulation becomes a scheduled event that occurs at a fix can make player interaction in the scene more difficult and may in must be well-thought through. However, using multiple scenes (or "important" objects, one asynchronous for "unimportant" objects) can be update/render local physics simulation from the game's update/render local physics simulation from the game's update/render local physics simulation becomes a scheduled event that occurs at a fix can make player interaction in the scene more difficult and may in the
- Permit the game to "drop" time when faced with a short-tern introduce visible motion artifacts if spikes occur frequently.
- Introduce slight variations in time-step (e.g. instead of simulating simulating a range between 1/50th and 1/60th). This can introduc into the simulation so should be used with caution. If this is done, must be simulated can potentially be amortized over several fra slightly larger time-steps.
- Consider simplifying the physics scene, e.g. reducing object count adjusting iteration counts etc. Provided physics simulation is a s

Pruner Performance for Streamed Environments

PhysX provides multiple types of pruners, each of which aimed at sp. These are:

- Static AABB tree
- Dynamic AABB tree

By default, the static AABB tree is used for the static objects in the er dynamics AABB tree is used for the dynamic objects in the environme approach works well but it must be noted that creating the static AAB expensive. As a result, adding, removing or moving any static objects will result in the static AABB tree being fully recomputed, which can ir performance cost. As a result, we recommend the use of dynamics A static and dynamic pruners in games which stream in the static envirc scene query performance against newly added objects can be in PxPruningStructure, which can precompute the AABB structure of in offline.

Performance Implications for Multi-Threading

The PhysX engine is designed from the ground-up to take advan architectures to accelerate physics simulation. However, this does not threads are always better. When simulating extremely simple so additional worker threads can detrimentally affect performance. This core, PhysX operates around a task queue. When a frame's simulation dispatches a chain of tasks that encapsulate that frame of physics sin stages of the physics pipeline, work can be performed in parallel of threads. However, if there is insufficient work, there will be little or no pathis case, the use of additional worker threads may detrimentally a because the various phases of the pipeline may be run by different working incur some additional overhead depending on the CPU archite

running on just a single worker thread. As a result, developers st performance of the engine with their expected physics loads with di threads to maximize their performance and make sure that they are r the available processing resources for their game.

Memory allocation

Minimizing dynamic allocation is an important aspect of performance provides several mechanisms to control memory usage.

Reduce allocation used for tracking objects by presizing the capaci structures, using either PxSceneDesc::limits before creating the sce PxScene::setLimits(). When resizing, the new capacities will be at required to deal with the objects currently in the scene. These variety preallocation and do not represent hard limits, so if you add more of 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 b size. You can control the current and maximum size of the nbContactDataBlocks and maxNbContactDataBlocks members of Px5 will never allocate more than the maximum number of blocks specifi insufficient memory it will instead simply drop contacts or joint constrain how many blocks are currently in use with the getNbContactBlocksU find out the maximum number that have ever been getMaxNbContactDataBlocksUsed() method.

Use PxScene::flushSimulation() to reclaim unused blocks, and to shrin data structures to the size presently required.

To reduce temporary allocation performed during simulation, provide ph block in the simulate() call. The block may be reused by the appetchResults() call which marks the end of simulation. The size of the multiple of 16K, and it must be 16-byte aligned.

Character Controller Systems using Scene Queries and Pe Computation

Implementing a Character Controller (CCT) is a common use case for Query (SQ) system. A popular approach is to use sweeps to impleme and to improve robustness by using Geometry Queries (GQ) to compupenetrations that occur due to object movement that does not account 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
- 2. If no initial overlap is detected, move the CCT shape to the positior adjust the trajectory of the CCT by removing the motion relative to of the hit.
- 3. Repeat Steps 1 and 2 until the goal is reached, or until an SÇ detects an initial overlap.
- 4. If an SQ-Sweep in Step 1 detects an initial overlap, use the GQ computation function to generate a direction for dependentation. Mc out of penetration and begin again with Step 1.

Limitations and Problems

Step 4 of the algorithm above can sometimes run into trouble due differences in SQ-Sweep, SQ-Overlap and and GQ-Penetration Department certain initial conditions it is possible that the SQ system will deterrobjects is initially overlapping while the GQ -Penetration Depth com them as disjoint (or vice-versa). Penetration depth calculations involved operate by shrinking the convex hull and performing distance calculations and the shrunken convex hull. To understand the conditions occurs and how to resolve the artefacts, please refer to the diagram below. Each diagram represents the initial conditions of two shall controller shape (red boxes), a convex obstacle (black boxes), at the

the algorithm above is executed. In the diagrams, the outermost recta the convex hull as seen by the SQ algorithms; the inner black box represents the shrunken convex shape and the black box with roun shrunken convex shape inflated by the amount by which we shrunk. boxes are used by the GQ-Penetration Depth computation. Although the convex hull obstacles, the issue is not exclusive to the convex hull sha similar for other shape types as well.

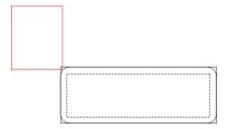


Diagram 1: CCT Shape Barely Touches an Obstacle

In **Diagram 1**, the red box of the CCT is barely touching the outermo convex obstacle. In this situation the SQ-Sweep will report an initial o Penetration Depth function will report no hit, because the red box is not box with rounded corners.

To resolve this, inflate the CCT shape for the GQ-Penetration Depth cathat it detects an overlap and returns a valid normal. Note that after shape, the GQ-Penetration Depth function will report that the shapes a deeply than they actually are, so take this additional penetration i depenetrating in Step 4. This may result in some clipping around the co convex objects but the CCT's motion should be acceptable. As the corr more acute, the amount of clipping will increase.

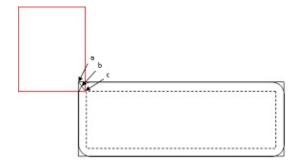


Diagram 2: CCT Overlaps an Obstacle Slightly

Diagram 2 shows a case where the CCT initially overlaps the outer bla SQ system, but does not overlap the shrunken shape seen by the GQ calculator. The GQ-Penetration Depth system will return the penetration point b but not from point c to point a. Therefore the CCT may clip through the convex hull after dependentation. This can be corrected in Step 4.

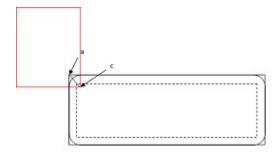


Diagram 3: CCT Overlaps an Obstacle Significantly

As can been seen from **Diagram 3**, if the CCT penetrates sufficiently to the shrunken shape seen by GQ, the GQ-Penetration Depth calcular penetration from point c to point a. In this case, the GQ-Penetration E used without modification in Step 4. However, as this condition we categorize without additional computational cost, it is best to influence the commended in Step 4 and then subtract this inflation from the redepth.

Unified MTD Sweep

A recent addition to the scene query sweeps is the flag PxHitFlag::e used in conjunction with default sweeps to generate the MTD (Mi Direction) when an initial overlap is detected by a sweep. This flag generate an appropriate normal under all circumstances, including case may detect an initial overlap but calling a stand-alone MTD function m still may suffer from accuracy issues with penetration depths but, in above around corners/edges, it will report a distance of 0 and the corn This can be used to remove components of the sweep moving into the and then re-sweeping when attempting to implement a CCT. The

compound MTDs for meshes/heightfields, which means that it reports penetrates the shape from the entire mesh rather than just an individual MTD exists.

Quantizing HeightField Samples

Heightfield samples are encoded using signed 16-bit integers for the y-l converted to a float and multiplied by PxHeightFieldGeometry::heightS space scaled coordinates. Shape transform is then applied on top to location. The transformation is performed as follows (in pseudo-code):

```
localScaledVertex = PxVec3(row * desc.rowScale, PxF32(heightSampl
    col * desc.columnScale)
worldVertex = shapeTransform( localScaledVertex )
```

The following code snippet shows one possible way to build quantize space heightfield coordinates from world space grid heights stored in te

```
const PxU32 ts = ...; // user heightfield dimensions (ts = terrai
// create the actor for heightfield
PxRigidStatic* actor = physics.createRigidStatic(PxTransform(PxId
// iterate over source data points and find minimum and maximum h
PxReal minHeight = PX_MAX_F32;
PxReal maxHeight = -PX_MAX_F32;
for(PxU32 s=0; s < ts * ts; s++)</pre>
    minHeight = PxMin(minHeight, terrainData.verts[s].y);
   maxHeight = PxMax(maxHeight, terrainData.verts[s].y);
}
// compute maximum height difference
PxReal deltaHeight = maxHeight - minHeight;
// maximum positive value that can be represented with signed 16
PxReal quantization = (PxReal)0x7fff;
// compute heightScale such that the forward transform will gener
// to the source
// clamp to at least PX_MIN_HEIGHTFIELD_Y_SCALE to respect the Ph
PxReal heightScale = PxMax(deltaHeight / quantization, PX_MIN_HEI
PxU32* hfSamples = new PxU32[ts * ts];
PxU32 index = 0;
```

```
for(PxU32 col=0; col < ts; col++)</pre>
{
   for(PxU32 row=0; row < ts; row++)</pre>
       PxI16 height;
       height = PxI16(quantization * ((terrainData.verts[(col*ts
           deltaHeight));
       PxHeightFieldSample& smp = (PxHeightFieldSample&)(hfSampl
       smp.height = height;
       smp.materialIndex0 = userValue0;
       smp.materialIndex1 = userValue1;
       if (userFlipEdge)
           smp.setTessFlag();
   }
}
// Build PxHeightFieldDesc from samples
PxHeightFieldDesc terrainDesc;
terrainDesc.format
                          = PxHeightFieldFormat::eS16_TM;
terrainDesc.nbColumns
                        = ts;
terrainDesc.nbRows
                          = ts;
terrainDesc.samples.data = hfSamples;
terrainDesc.samples.stride = sizeof(PxU32); // 2x 8-bit material
terrainDesc.thickness = -10.0f; // user-specified heightfie
                         = PxHeightFieldFlags();
terrainDesc.flags
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.g = 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. Usin convex meshes is mainly useful for performance, but it can also red pairs coming out of the broad-phase, which decreases the amount of manage these pairs.

Use scratch buffers

See the above chapter about Performance Issues for details. Scrat shared between multiple sub-systems (e.g. physics and rendering), improve memory usage. PhysX will not use less memory per-se, but it it.

Flush simulation buffers

Call the PxScene::flushSimulation function to free internal buffers u computations. But be aware that these buffers are usually allocated c subsequent frames, so releasing the memory might trigger new re-allonext simulate call, which can decrease performance. Please refer *memory* chapter for details.

Use preallocation

Use PxSceneDesc::limits to preallocate various internal arrays. Prea necessary size for internal buffers may use less memory overall the resizing strategy of dynamic arrays. Please refer to the *Simulation n* details.

Tweak cooking parameters

Some cooking parameters have a direct impact on memory us PxMeshPreprocessingFlag::eDISABLE_ACTIVE_EDGES_PRECOMPL

PxCookingParams::suppressTriangleMeshRemapTable,

PxBVH33MidphaseDesc::meshCookingHint,

PxBVH33MidphaseDesc::meshSizePerformanceTradeOff,

PxBVH34MidphaseDesc::numTrisPerLeaf, PxCookingParar

PxCookingParams::gaussMapLimit and PxCookingParams::buildTrianç be modified to choose between runtime performance, cooking perfor usage.

Share shape and mesh data

Share the same PxConvexMesh and PxTriangleMesh objects betwee instances if possible. Use shared shapes if possible. Please refer *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 shape is only used for simulation (e.g. it will never be raycasted a scene-query flag. This is good for both memory usage and performan 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 This can artificially prevent the objects from spinning quickly, which n and wrong in some cases. Please use PxRigidDynamic::setMax increase the maximum allowed angular velocity.

Overlapping objects explode

Rigid bodies created in an initially overlapping state may explode, bec to resolve the penetrations in a single time-step, which can lead to large use PxRigidBody::setMaxDepenetrationVelocity to limit the de-penetr reasonable value (e.g. 3.0).

Rigid bodies are jittering on the ground

Visualize the contacts with the visual debugger. If the jittering is cause appear and disappear from one frame to another, try to increase (PxShape::setContactOffset).

Piles or stacks of objects are not going to sleep

PxSceneFlag::eENABLE_STABILIZATION might help here. This is not jointed objects though, so use PxRigidDynamic::setStabilizate enable/disable this feature on a per-object basis. It should be safe to 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 simil
 number of solver iterations, but the performance impact is local
 object rather than the simulation island it is a part of. So it can
 overall. Note that the order in which constraints are created is i
 have 4 constraints named A, B, C, D, and you want to create tl
 Creating them in the AAAABBBBCCCCDDDD order will not improv
 creating them in the ABCDABCDABCDABCD order will.
- Consider using joint projection. This might help for simple cases objects are connected. Please refer to the *Joints* chapter for details
- Use smaller time steps. This can be an effective way to improalthough it can be an expensive solution. Instead of running 1 sir time-step dt and N solver iterations, consider trying N simulation ca dt/N and 1 solver iteration.
- Consider tweaking inertia tensors. In particular, for ropes or chains
 the PxJoint::setInvMassScale and PxJoint::setInvInertiaScale func
 effective. An alternative is to compute the inertia tel
 PxRigidBodyExt::setMassAndUpdateInertia) with an artificially inc
 then set the proper mass directly afterwards (using PxRigidBody::s
- Consider adding extra distance constraints. For example in a rope to create an extra distance constraint between the two ends of the stretching. Alternatively, one can create distance constraints between N+2 in the chain.
- Use spheres instead of capsules. A rope made of spheres will be rope made of capsules. The positions of pivots can also affect s pivots at the spheres' centers is more stable than placing their surfaces.
- Use articulations. Perhaps not surprisingly, articulations are much articulated objects. They can be used to model better ropes, br

ragdolls out-of-the-box, without the need for the above workaroun the *Articulations* chapter for details. They are more expensive though.

GPU Rigid Bodies

Collision detection with PxSceneFlag::eENABLE_GPU_DYNAMICS w GPU for all convex-convex, convex-box, box-box, convex-mesh, box anb 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 compati requests contact modification, the triangle mesh was not cooked with G (PxCookingParams::buildGrbData) or if the triangle mesh makes u materials.

Aggregates are used to lighten the load on broad phases. When runni the CPU, aggregates frequently improve performance by reducing the broad phase algorithm. However, there is some cost when aggregate these overlaps must be processed by a separate module. When using the use of aggregates generally result in performance regressions beca of aggregate overlaps occurs on the CPU and, while using aggregat load on the GPU broad phase, the amount by which they improve performance is frequently smaller than the cost of processing the aggre

Determinism

The PhysX SDK can be described as offering limited determinism. between platforms due to differences in hardware maths precision and the compiler reoders instructions during optimization. This means the different between different platforms, different compilers operating on the between optimized and unoptimized builds using the same compiler on However, on a given platform, given the exact same sequence of even exact scene using a consistent time-stepping scheme, PhysX is ex deterministic results. In order to achieve this determinism, the application the scene in the exact same order each time and insert the actors in PxScene. There are several other factors that can affect determinism s (e.g. variable) time-stepping scheme is used or if the application does same sequence of API calls on the same frames, the PhysX simulation

In addition, the PhysX simulation can produce divergent behavior if ar simulation has varied. Even the addition of a single actor that is not existing set of actors in the scene can produce divergent results.

PhysX provides a mechanism to overcome the issue of divergent b configurations as a result of additional actors being added or actors b the scene that do not interact with the other actors in the scene. This enabled by raising PxSceneFlag::eENABLE_ENHANCED_DE PxSceneDesc::flags prior to creating the scene. Enabling this m performance concessions to be able to offer an improved level of application must still follow all the requirements to achieve deted described previously in order for this mechanism to produce consistent

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

This guide describes how to upgrade applications that have an integrat using PhysX 3.x. As the changes are numerous and significant, the lev in upgrading to PhysX 3 should be carefully assessed before state 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 shown PhysX 2.

Compartments

PhysX 2 scenes supported scene compartments. A separate compassigned to simulating rigid bodies, deformables or fluids. The compainments in parallel and the scene code contained some extra lobetween compartments. Compartments were added as an afterthoug was not originally designed to support interaction between rechnologies. This design deficiency was addressed from the ground comparments were no longer needed.

One missing detail is separate time steps are no longer directly suppor workaround is to create multiple PxScenes and step them at diffe scenario the force exchange implementation would be entirely up to 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 environmental cloth, soft bodies, inflatable balloons and plastic deformable performance and code quality reasons, 3.3 temporarily stopped soft 2.8 deformable features in favor of a much simpler and higher simulation engine. In PhysX 3 dot releases, we will be incrementally action as environmental simulation. For the time being there is no soft applications of PhysX 2 deformables.

NxUtilLib

The assorted utility functions that were in this library was either modeleted. Sweep, overlap and ray tests are available in PxGeometryQu diagonalization is in PxDiagonalize(). Density computation from mass point unit manipulation routines are gone. Geometrical helpers are in ge

Anisotropic Friction

Friction on a surface in PhysX 2 could be configured to be stronger in o another. This is no longer supported in PhysX 3, and there is no know will give comparable behavior.

Basics

SDK Header

In PhysX 2, the symbols of the SDK could be included in the user's rethrough 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 ϵ 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

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:

Callback Classes

PhysX 2 callback classes are listed below, followed by the correspondir there is one:

NxUserAllocator	PxAllocatorCallback	
NxUserOutputStream	PxErrorCallback	
NxUserContactReport	PxSimulationEventCallback	
NxUserNotify	PxSimulationEventCallback	
NxUserTriggerReport	PxSimulationEventCallback	

The following PhysX 2 callback classes have no PhysX 3 direct equivalent

NxUserRaycastReport	Ray casting Results. Results are now passed t PxHitBuffer object.
NxUserEntityReport	Sweep and Overlap results. Results are now pusing a PxHitBuffer object.
NxStream	Data serialization. Serialized data is now writte binary buffers.

Below is a list of new callback classes that offer functionality that did n yet:

PxBroadPhaseCallback	Broad-phase related events.
PxSimulationFilterCallback	Contact filtering.
PxUserControllerHitReport	Reports character controller events.
PxControllerBehaviorCallback	Customizes behavior of character cont
PxContactModifyCallback	Modification of contact constraints.
PxCCDContactModifyCallback	Modification of CCD contact constraint
PxConstraintConnector	Custom constraints.
PxProcessPxBaseCallback	Serialization.
PxQueryFilterCallback	Scene query filtering.
PxSpatialLocationCallback	Scene Queries against PxSpatialIndex
PxSpatialOverlapCallback	Scene Queries against PxSpatialIndex

Memory Management

NxUserAllocator is renamed to PxAllocatorCallback. An important char The SDK now requires that the memory that is returned be 16-byte platforms malloc() returns memory that is 16-byte aligned, but on Wifunction _aligned_malloc() provides this capability.

Debug Rendering

Debug visualization formerly provided by NxScene::getDebugRenderab by PxScene::getRenderBuffer() and related functions.

Error Reporting

NxUserOutputStream is now called PxErrorCallback, but works the san separate reportAssertViolation() function. Asserts are only contained which only ships with the source release and go directly to platform hoo

Type Casting

PhysX 2 style downcasting:

```
NxSphereShape * sphere = shape->isSphere();
```

is replaced by the following template syntax:

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

Multithreading

Compared to PhysX 2, there are now more situations where it is legal to multiple threads. See the section on Multithreading for details.

While PhysX 2 simulation threads were managed internally by the 5

could simply specify the number to use, PhysX 3 allows the application the simulation's thread scheduling. It is also possible for the application tasks and submit them to the SDK's default scheduler. See TaskManagement for details.

Startup and Shutdown

PxCreatePhysicsSDK() has been renamed PxCreatePhysics(), and the slightly changed. A foundation instance must first be created PxCreateFoundation().

Extensions

A lot of non-essential utility code has been moved to the extensions lik NxActor::addForceAtPos() is now exposed as PxRigidBodyExt::add former function appears to be missing, look there. It is avail PxInitExtensions().

Heightfields

Heightfields now need to be pre-cooked like convexes and meshes. P can be set to use the same internal collision logic as meshes so behavior.

Cooking

The PhysX 2 cooking library was created by calling:

```
NxCookingInterface *gCooking = NxGetCookingLib(NX_PHYSICS_SDK_VER
gCooking->NxInitCooking();
```

It can now be accessed through a single PxCreateCooking() call. Cook are slightly changed, e.g. NxCookTriangleMesh() is nc cooking.cookTriangleMesh().

Serialization

PhysX 3 has two serialization systems: 'RepX' based on XML, and a s fast binary data. Neither approach is similar to PhysX 2's save-to-desc a based serialization code, though the PhysX 3 'RepX' serialization is s NxUStream.

API Design Changes

Changed Actor Hierarchy

PhysX 2 only had a single actor class, and it was possible to call a instance of this class even if it wasn't applicable to the kind of actor objection (a subscription) could be called on static actors which did not have a linear physical subscription (a subscription) and PxParticleSystem are now subclasses of PxActor.

Actor Creation

In PhysX 2, the objects inside each scene were created by the scene cl 3, objects are created by PxPhysics, and need to be added to a sc subsequent step by calling:

mScene->addActor(actor);

Material Indexes

PhysX 2 uses so-called material indexes for stored materials. Me supported in PhysX 3 only to specify per-triangle materials in meshes other cases the material object is referenced directly.

Continuous Collision Detection

PhysX 2 uses CCD skeleton meshes for CCD. PhysX 3 no longer new skeleton related code can simply be removed.

Pose Description

In PhysX 2 pose is specified using a matrix. In PhysX 3, pose is PxTransform type that consists of a PxVec3 for translation and a P Constructors are provided to convert 4x4 matrices to PxTransform matrices from quaternions, as well as conversely.

Shape Description

PhysX 2 has multiple subclasses of NxShape, one for each type corresponding NxShapeDesc classes. PhysX 3 has only a single PxSh a PxGeometry object is passed on creation. To determine the geomet call PxShape::getGeometryType(). To extract a PxGeometry object unknown type, use PxShape::getGeometry().

Skin Width

PhysX 2's NX_SKIN_WIDTH and NxShapeDesc::skinWidth was PxShape::setContactOffset() and setRestOffset(). See *Tuning Shape ContactOffset*().

Joints

The D6 driveType in PhysX 2 no longer exists in PhysX 3. Now driv spring-like: if you want position drive you set the 'spring' value nor velocity drive you set the damping field non-zero, and if you set both 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 NxSpringAndDamperEffector, you shou 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 distant which it becomes active. It functions similarly to the contactOffset pardetection.

Time Stepping

PhysX 2 had two different time stepping modes: NX_TIMESTEP_FIXE into fixes steps) and NX_TIMESTEP_VARIABLE (user specified steps) to the setTiming() function. This controlled SDK-internal substepping c the proper size of the next time step, and called an internal simulate elapsed time.

PhysX 3 discards with the substepping code altogether, and expose simulate function directly:

```
mScene->simulate(mStepSize);
```

In PhysX 2 it was legal to call simulate with a timestep of zero to for various side-effects of simulation. PhysX 3 neither requires nor supports

The fetchResults function stayed the same, however there is no more fl simulation to fetch, as there is now only a single simulation.

Simulation Parameters

The global speeds below which objects go NX_DEFAULT_SLEEP_LIN_VEL_SQUARED NX_DEFAULT_SLEEP_ANG_VEL_SQUARED are gone. PhysX 3 ins body function PxRigidDynamic::setSleepThreshold() which is an ene more similar to the PhysX 2 NX_DEFAULT_SLEEP_ENERGY.

The global NX_BOUNCE_THRESHOLD is PxSceneDesc::bounceThresholdVelocity.

The NX_DYN_FRICT_SCALING, NX_STA_FRICT_SCALING scaling removed. These values should now be pre-baked into friction coefficient

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 filterir 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 restrimemory cannot be accessed by filtering code so that it can be execute on GPUs with optimal performance. If performance is not a priority, a can be achieved via conventional callbacks (PxSimulationFilterCallback

When migrating PhysX 2 code, note that we properly PxDefaultSimulationFilterShader in PhysX 3, which emulates a portion behavior. Start by checking if this class is sufficient. As this is an expounded code is available and may be extended or customized.

To migrate your fixed function PhysX 2 filtering code on your own, you r its exact behavior and implement it as a callback or shader. Let us lool mechanisms and make some recommendations for porting:

The first function stored explicit shape pairs in a hash, and a look indicating to filter or not. The second did the same for actor pairs. Beca size of the pair hash, implementing this mechanism as a shader will difficult in practice, but implementing as a callback should be trivial usi such as the STL hash_map where Key is a struct holding the two point bit flag.

Another scheme provided by PhysX 2 were collision groups:

This approach let the user assign shapes to one of 32 collision groups pair of groups be assigned a boolean pair flag. This approach lend shader based implementation. To do this, you should reserve a wor filterData (say word0) to hold the group index, and assign this as bef matrix to hold the group pair bits, and a function to set it:

```
NxU32 groupCollisionFlags[32];
//init all group pairs to true:
for (unsigned i = 0; i < 32; i ++)
    groupCollisionFlags[i] = 0xffffffff;
void setU32CollisionFlag(NxU32 groups1, NxU32 groups2, bool enabl
{
    NX_ASSERT(groups1 < 32 \&\& groups2 < 32);
    if (enable)
        //be symmetric:
        groupCollisionFlags[groups1] |= (1 << groups2);</pre>
        groupCollisionFlags[groups2] |= (1 << groups1);</pre>
    else
        groupCollisionFlags[groups1] &= ~(1 << groups2);</pre>
        groupCollisionFlags[groups2] &= ~(1 << groups1);</pre>
    }
}
```

Unfortunately it is not possible to change this state after the scene because if the matrix could change during simulation, it would force an existing contact pairs to be refiltered. In a large simulation, this could t amount of computation. Therefore the matrix must be initialized to its fil

scene is created, like this:

```
PxSceneDesc desc;
...
desc.filterShaderData = groupCollisionFlags;
desc.filterShaderDataSize = 32 * sizeof(PxU32);
scene = sdk.createScene(desc);
```

Finally, you need to code the filter shader to access this data:

```
PxFilterFlags FilterShader(
    PxFilterObjectAttributes attributesO, PxFilterData filterData
    PxFilterObjectAttributes attributes1, PxFilterData filterData
    PxPairFlags& pairFlags, const void* constantBlock, PxU32 cons
{
    // let triggers through, and do any other prefiltering you ne
    if(PxFilterObjectIsTrigger(attributes0) || PxFilterObjectIsTr
    {
        pairFlags = PxPairFlag::eTRIGGER_DEFAULT;
        return PxFilterFlag::eDEFAULT;
    // generate contacts for all that were not filtered above
    pairFlags = PxPairFlag::eCONTACT_DEFAULT;
    PxU32 ShapeGroup0 = filterData0.word0 & 31;
    PxU32 ShapeGroup1 = filterData1.word0 & 31;
    PxU32* groupCollisionFlags = (PxU32*)constantBlock;
    if ((groupCollisionFlags[ShapeGroup0] & (1 << ShapeGroup1)) =</pre>
        return PxFilterFlag::eSUPPRESS;
    else
        return PxFilterFlag::eDEFAULT;
}
```

Scene Queries

The API for scene query functions that return multiple i PxScene::raycast(...)) has changed. In PhysX 3, raycast/overlap/sweep pre-allocated buffer or a callback class as a parameter in order intersections. If you do not know the maximum number of intersectio can inherit from PxHitCallback and override processTouches virtual fur arbitrary number of intersections via multiple callbacks using only a Please refer to the Scene Query section of the guide for more details ar

Raycasts

The interface for making raycasts was changed in PhysX 3. Now your origin (PxVec3) and a direction (PxVec3) instead of a NxRay that comb PhysX 2.

Overlaps

Routines like overlapSphereShapes, overlapAABBShapes, o overlapCapsuleShapes are now all covered with PxScene::overlapCxSphereGeometry, PxBoxGeometry or PxCapsuleGeometry as a first

Sweep Tests

PhysX 2 provides a linearCapsuleSweep that takes two points to define spherical ends. In PhysX 3 we have a general sweep() routine that tale and an initial PxTransform position. Capsules were defined in PhysX 2 should be converted to an initial transformation (PxTransform) that composition and PxQuat for rotation. PxCapsuleGeometry's length is along space.

User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous | next

• This guide highlights all significant parts of the API that have char release. An application with a working integration of the older version be able to easily migrate to the newer version by following these pc

Math Classes

The static createIdentity() and createZero() methods are now depre removed in a future release. The preferred method is to use PxMat33(PxIdentity), PxMat44(PxIdentity), PxQuat(PxIdentity), PxTrafor identity transforms, and PxMat33(PxZero) and PxMat44(PxZero) for

Scene Query API

- The Scene Query API underwent significant changes. The highlight
 - Former raycastAny, raycastMultiple, raycastSingle API ca into a single PxScene::raycast call
 - Same for overlaps and sweeps
 - Same for PxBatchQuery and PxVolumeCache
 - For PxScene queries a deprecated backwards cor was added to aid the transition
 - This mapping will be removed in the next defeater.
 - There are now dedicated callback and buffer classes f results, replacing PxRaycastHit array and count parameters
 - Same for sweeps and overlaps
 - See PxRaycastBuffer, PxSweepBuffer,
 PxRaycastCallback, PxSweepCallback, PxOverlap
 - The way results are returned is now more robust and transparently handle unbounded number of results allocations.
 - Header PxSceneQueryFiltering.h was renamed to PxSceneQueryReport.h to PxQueryReport.h
 - PxHitFlag::eIMPACT changed to PxHitFlag::ePOSITION
 - PxRaycastHit.impact renamed to PxRaycastHit.pos
 PxSweepHit.impact)
 - PxQueryFlag::eNO_BLOCK and PxQueryFlag::eANY_HIT

- The following classes were renamed
 - PxSceneQueryHit -> PxQueryHit
 - PxSceneQueryFlags -> PxHitFlags
 - PxSceneQueryHitType -> PxQueryHitType
 - PxSceneQueryFilterData -> PxQueryFilterData
 - PxSceneQueryFilterCallback -> PxQueryFilterCallb
 - PxSceneQueryFilterFlags -> PxQueryFlags
 - PxSceneQueryCache -> PxQueryCache
 - PxCCTNonWalkableMode -> PxControllerNonWalk
 - PxControllerFlags -> PxControllerCollisionFlags
 - PxCCTHit -> PxControllerHit
 - PxConstraintDominance -> PxDominanceGroupPa
 - PxActorTypeSelectionFlags -> PxActorTypeFlags
 - PxFindOverlapTriangleMeshUtil -> PxMeshOverlap
 - Old versions are #defined to new versions to simpli
 #defines are deprecated and will be phased out.
- queryClient parameter was removed from raycast/sweep.
 list and added to PxQueryFilterData
 - The fix is to simply pass the sate PxQueryFilterData::clientId
- PxBatchQueryDesc now requires 3 parameters at construmaxRaycastsPerExecute, PxU32 maxSweepsPerEmaxOverlapsPerExecute
 - Each of these numbers is an upper bound of PxBatchQuery::raycast(), sweep() and overlap() cate 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 mel
 3.3 these are moved to PxBatchQueryMemory.
 - It is now possible to set a new batch query memor each execute
 - userRaycastHitBuffer has been renamed to userRa
 - raycastHitBufferSize has been renamed to raycast
 - same for overlaps and sweeps (user Sweep Hit Buffer Size, user Overlap Hit Buffer, overlap)
 - A code snippet below illustrates the migration for th
- PxQueryFilterData constructors are now explicit. This mean was possible to write
 - scene->raycast(..., PxQueryFlag::eDYN PxQueryFlag::eSTATIC, ...), causing PxQuery implicitly constructed by the compiler
 - now it is required to explicitly
 raycast(...,PxQueryFilterData(PxQueryFlag::eDYNPxQueryFlag::eSTATIC), ...)
 - This change was made to improve type safety and while reading the code employing implicit construct
- PxRaycastBufferN, PxOverlapBufferN and PxSweepBuffe convenience
 - A buffer object with space for 10 touching hits ar can now be conveniently declared as PxRaycastBu
- PxRaycastHit and PxSweepHit now inherit from PxLocatio
 PxSceneQueryImpactHit)

- bool PxLocationHit::hadInitialOverlap() function was addesswept shape was overlapping at sweep distance=0 or if a at distance=0.
- Functionality of PxSceneQueryFlag::eINITIAL_C PxSceneQueryFlag::eINITIAL_OVERLAP_KEEP was PxHitFlag::eASSUME_NO_INITIAL_OVERLAP 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 f so will result in undefined behavior, and a warning \(\)
 - If the PxQueryFlag::eNO_BLOCK flag is set, the el be automatically converted to an eTOUCH suppressed.
- Sweeps in 3.3 execute using a new faster code path, ir reduced precision. If you encounter precision issue experienced in earlier versions of PhysX, use ePRECIS enable the backwards compatible more accurate sweep co
- Snippets demonstrating API migration:

Former raycastSingle call:

```
PxRaycastHit hit;
bool hadHit = scene->raycastSingle(..., hit, ...);
if (hadHit) doStuff(hit);
```

Is now:

```
PxRaycastBuffer buf;
Bool hadHit = scene->raycast(..., buf, ...);
if (hadHit) doStuff(buf.block);
```

Former raycastAny call:

```
PxSceneQueryHit hit;
bool hadHit = scene->raycastAny(hit);
if (hadHit) doStuff(hit);
```

Is now:

```
PxRaycastBuffer buf; // declare a hit buffer with room for a sing
PxFilterData fdAny; fdAny.flags |= PxQueryFlag::eANY_HIT;
bool hadHit = scene->raycast(buf, PxHitFlags(), fdAny);
if (hadHit) doStuff(buf.block);
```

Former Multiple call:

```
PxRaycastHit buffer[N];
bool hasBlock;
PxI32 result = Scene->raycastMultiple(buffer, N, hasBlock);
if (result == -1)
    handleOverflow();
else
{
    if (hasBlock)
    {
        doBlocking(buffer[result-1]);
        doTouches(buffer, result-1);
    }
    else
    {
        doTouches(buffer, result);
    }
}
```

Is now:

```
PxRaycastBufferN<N> buf;
scene->raycast(buf);
if (buf.hasBlock)
    doBlocking(buf.block);
doTouches(buf.touches, buf.nbTouches);
```

or:

Former batch query memory setup code in 3.2:

```
const PxU32 maxRaycastHits = 16, maxRaycastQueries = 8;
PxRaycastOueryResult* resultBuffer = new PxRaycastOueryResult[max
PxRaycastHitBuffer* hitBuffer = new PxRaycastHit[maxRaycastHits];
PxBatchQueryDesc desc; // required no arguments, there was no saf
                       // of gueries per batch (not hits per guer
desc.userRaycastResultBuffer = resultBuffer;
desc.userRaycastHitBuffer = hitBuffer;
desc.raycastHitBufferSize = maxRaycastHits;
PxBatchQuery* bq = PxCreateBatchQuery(desc);
for (PxU32 iQuery = 0; iQuery < maxRaycastQueries; iQuery++)</pre>
    bq->raycastSingle(...); // up to 8 raycast queries are allowe
                            // call but there was no overflow che
bq->execute();
for (PxU32 iResult = 0; iResult < nQueries; iResult++)</pre>
    for (PxU32 iHit = 0; iHit < resultBuffer[i].nbHits; iHit++)</pre>
    {
        bool isBlocking = (iHit == resultBuffer[i].nbHits &&
            (resultBuffer[iResult].hits[iHit].flags & PxSceneQuer
        processHit(resultBuffer[iResult].hits[iHit], isBlocking);
    }
}
```

Batch query setup code in 3.3:

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 SI separate PPU thread, this design allows higher flexibility, such as examultiple SPU and PPU threads simultaneously, better control of para allows the user to fine tune thread load balancing. Here's one possible 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) { mBatc
    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++)</pre>
    BQThread* t = PX_NEW(BQThread);
    t->start();
    mThreads.pushBack(t);
}
// submit batches
for (PxU32 iThread = 0; iThread < nThreads; iThread++)</pre>
{
    // 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)</pre>
    mThreads[i]->mBatchCompleted.wait();
    mThreads[i]->mBatchCompleted.reset();
    releaseBatch(mThreads[i]->mBatch);
}
// terminate batch threads
for (PxU32 i=0; i<mThreads.size(); ++i)</pre>
{
    mThreads[i]->signalQuit();
    mThreads[i]->mBatchReady.set();
    mThreads[i]->waitForQuit();
    PX DELETE(mThreads[i]);
}
```

Whether the batch is executed on SPU or PPU is determine PxBatchQueryDesc::runOnSpu or PxBatchQuery::setRunOnSpu(bool) query is executed on SPU:

```
PxBatchQueryDesc desc;
...
desc.runOnSpu = true;
...
```

Core PhysX

- The following methods require that the corresponding objects hav scene. Calling these methods for objects which are not in a s undefined behavior. In the CHECKED build configuration an erro sent.
 - addForce/addTorque/clearForce/clearTorque() on a PxRigidI
 - isSleeping/wakeUp/putToSleep() on a PxRigidDynamic, I PxCloth
 - PxScene::resetFiltering() and the deprecated counterparts
 PxParticleBase
- The sleep behavior of dynamic rigid bodies has changed significances are:
 - The wakeUp() method of PxRigidDynamic and PxArticular wake counter parameter. Use the newly intro setWakeCounter() instead to set a specific value.
 - Putting a dynamic rigid actor to sleep will clear any pending
 - Switching a dynamic actor to kinematic will put the immediately.
 - Switching a kinematic actor back to dynamic will not affec (previously the actor was woken up).
 - Calling wakeUp/putToSleep() on a kinematically controlled not valid any longer. The sleep state of a kinematic actor based on whether a target pose has been set (see API c isSleeping() for details).
 - A call to PxRigidBody::setCMassLocalPose() does not wa anymore. Add a call to PxRigidBody::wakeUp() to get the ol

- Note: this also affects related methods in PhysX PxRigidBodyExt::updateMassAndInertia() etc.
- Adding or removing a PxConstraint to/from the scene do connected actors up automatically anymore (note: this app PhysX Extensions as well).
- If a non-zero velocity or force is PxRigidBody::setLinearVelocity(), ::setAngularVelocity(), ::addTorque(), the actor will get woken up automatically ever parameter is false.
- PxRigidBody::clearForce() and ::clearTorque() do not hav 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 seve
 - 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(). Replac found in PxShapeExt.
 - PxShape now has the same reference counting semantics materials, so that release() releases the user reference, at 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 serial
 - return results from scene queries which previously specifically specifically 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 PxActo inheriting from PxBase. Furthermore, two kinds of deletion distinguished: user release and memory release. The following pseudocode for the transition from the previous to the new API:

old API:

```
class MyObserver : public PxObserver
{
public:
    virtual void onRelease(const PxObservable& observable);
}
MyObserver myObs;
PxRigidDynamic* d = create...;
d->registerObserver(myObs);
```

new API:

```
class MyDelListener : public PxDeletionListener
{
public:
    virtual void onRelease(const PxBase* observable, void* userDa
        PxDeletionEventFlag::Enum deletionEvent);
}

MyDelListener myDelListener;
PxPhysics* physics = create...;
PxRigidDynamic* d = create...;
physics->registerDeletionListener(myDelListener, PxDeletionEventF
PxBase* b = d;
physics->registerDeletionListenerObjects(myDelListener, &b, 1);
```

 The contactStream in PxContactPair is now stored in a variable contact stream. This is used to save memory. As such, you can no it to a PxContactPoint* and access the data. Instead, you PxContactPair::extractContacts or us a PxContactStreamIterator to Please see the callbacks section of the user guide for further inform

- The friction API and behavior for dynamic rigid bodies has changed
 - Friction mode flags eENABLE_ONE_DIRECTIONAL_I eENABLE_TWO_DIRECTIONAL_FRICTION have beer PxFrictionType::Enum PxSceneDesc::frictionType.
 - PxSceneDesc::contactCorrelationDistance has been depre longer has an influence on how many friction anchors are cr frame, only on when they are removed in later frames. The very minor change in friction behavior.
- PxShape::resetFiltering() and PxParticleBase::resetFiltering() have
 Please use one of the new overloaded methods PxScene::resetFilt
- PxClientBehaviorBit and PxActorClientBehaviorBit have be PxClientBehaviorFlag and PxActorClientBehaviorFlag respectively.
- PxActorTypeSelectionFlag and PxActorTypeSelectionFlags have PxActorTypeFlag and PxActorTypeFlags respectively.
- PxConstraintDominance has been renamed to PxDominanceGroup
- The parameter 'spring' on articulation joints has been renamed 'stifl
- The parameter 'tangentialSpring' on articulation joints has 'tangentialStiffness'.
- PxConstraintFlag::Type has been renamed to PxConstraintFlag::Er
- Discrete contact reports are no longer produced for PxPairFlag::eDETECT_DISCRETE_CONTACT raised in the filter discrete contact generation would always have been performed presence of the PxPairFlag::eDETECT_DISCRETE_CONTACT

potentially improves performance when using specific shapes for which would have previously generated discrete contacts and the the solver.

 Trigger reports are no longer produced for PxPairFlag::eDETECT_DISCRETE_CONTACT raised in the PxPairFlag::eTRIGGER_DEFAULT has been modified to 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 getConcreteType() replaces getType(). Alternatively, to dynar 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 '
- Dominance settings no longer apply to joints. To achieve setInvMassScale. For example if actor0 in the joint is to affer conversely, use setInvMassScale0(0.0f), setInverseInertiaScale0(0.0f).

PhysX Character Controller

- When creating a PxControllerManager, a reference to a PxScene | As a consequence, creating a controller from a PxControllerN requires the controller descriptor as an argument.
- On PxControllerManager::release(), all associated PxObstacleCol get deleted automatically. Make sure to not access PxObstacleCon the corresponding manager has been released.

PhysX Vehicles

- A new struct has been introduced to hold the enumerated lis 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 lis PxVehicleDrive4WControl::eANALOG_INPUT_ACCEL. The change
 - PxVehicleDrive4W::eANALOG_INPUT_ACCEL
 PxVehicleDrive4WControl::eANALOG_INPUT_ACCEL
 - PxVehicleDrive4W::eANALOG_INPUT_BRAKE
 PxVehicleDrive4WControl::eANALOG_INPUT_BRAKE
 - PxVehicleDrive4W::eANALOG_INPUT_HANDBRAKE
 PxVehicleDrive4WControl::eANALOG_INPUT_HANDBRAKI
 - PxVehicleDrive4W::eANALOG_INPUT_STEER_LEFT
 PxVehicleDrive4WControl::eANALOG_INPUT_STEER_LEF
 - PxVehicleDrive4W::eANALOG_INPUT_STEER_RIGHT
 PxVehicleDrive4WControl::eANALOG_INPUT_STEER_RIG
 - PxVehicleDrive4W::eMAX_NUM_DRIVE4W_ANALOG_INPI PxVehicleDrive4WControl::eMAX_NB_DRIVE4W_ANALOG

- A new struct has been introduced to hold the enumerated lis 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_LE
 PxVehicleDriveTankWheelOrder::e1ST_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_1ST_FROM_FRONT PxVehicleDriveTankWheelOrder::e1ST_FROM_FRONT_RIC
 - PxVehicleDriveTank::eTANK_WHEEL_2ND_FROM_FRONT PxVehicleDriveTankWheelOrder::e2ND_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_2ND_FROM_FRONT PxVehicleDriveTankWheelOrder::e2ND_FROM_FRONT_RIV
 - PxVehicleDriveTank::eTANK_WHEEL_3RD_FROM_FRONT PxVehicleDriveTankWheelOrder::e3RD_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_3RD_FROM_FRONT_RIPART
 PxVehicleDriveTankWheelOrder:: e3RD_FROM_FRONT_RIPART
 - PxVehicleDriveTank::eTANK_WHEEL_4TH_FROM_FRONT PxVehicleDriveTankWheelOrder::e4TH_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_4TH_FROM_FRONT PxVehicleDriveTankWheelOrder::e4TH FROM FRONT RIC
 - PxVehicleDriveTank::eTANK_WHEEL_5TH_FROM_FRONT PxVehicleDriveTankWheelOrder::e5TH_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_5TH_FROM_FRONT PxVehicleDriveTankWheelOrder::e5TH_FROM_FRONT_RIC
 - PxVehicleDriveTank::eTANK_WHEEL_6TH_FROM_FRONT PxVehicleDriveTankWheelOrder::e6TH_FROM_FRONT_LE
 - PxVehicleDriveTank::eTANK_WHEEL_6TH_FROM_FRONT

- PxVehicleDriveTankWheelOrder::e6TH FROM FRONT RIC
- PxVehicleDriveTank::eTANK_WHEEL_7TH_FROM_FRONT PxVehicleDriveTankWheelOrder::e7TH_FROM_FRONT_LE
- PxVehicleDriveTank::eTANK_WHEEL_7TH_FROM_FRONT
 PxVehicleDriveTankWheelOrder::e7TH_FROM_FRONT_RICE
- PxVehicleDriveTank::eTANK_WHEEL_8TH_FROM_FRONT
 PxVehicleDriveTankWheelOrder::e8TH_FROM_FRONT_LE
- PxVehicleDriveTank::eTANK_WHEEL_8TH_FROM_FRONT PxVehicleDriveTankWheelOrder::e8TH_FROM_FRONT_RIC
- PxVehicleDriveTank::eTANK_WHEEL_9TH_FROM_FRONT PxVehicleDriveTankWheelOrder::e9TH_FROM_FRONT_LE
- PxVehicleDriveTank::eTANK_WHEEL_9TH_FROM_FRONT PxVehicleDriveTankWheelOrder::e9TH_FROM_FRONT_RIC
- PxVehicleDriveTank::eTANK_WHEEL_9TH_FROM_FRONT_RICE
 PxVehicleDriveTankWheelOrder::e9TH_FROM_FRONT_RICE
- A new struct has been introduced to hold the enumerated lis PxVehicleDriveTank::eANALOG_INPUT_ACCEL. The changes are
 - PxVehicleDriveTank::eANALOG_INPUT_ACCEL
 PxVehicleDriveTankControl::eANALOG_INPUT_ACCEL
 - PxVehicleDriveTank::eANALOG_INPUT_BRAKE_LEFT
 PxVehicleDriveTankControl::eANALOG_INPUT_BRAKE_LE
 - PxVehicleDriveTank::eANALOG_INPUT_BRAKE_RIGHT
 PxVehicleDriveTankControl::eANALOG_INPUT_BRAKE_RIGHT
 - PxVehicleDriveTank::eANALOG_INPUT_THRUST_LEFT
 PxVehicleDriveTankControl::eANALOG_INPUT_THRUST_L
 - PxVehicleDriveTank::eANALOG_INPUT_THRUST_RIGHT
 PxVehicleDriveTankControl::eANALOG_INPUT_THRUST_F
 - PxVehicleDriveTank::eMAX_NUM_DRIVETANK_ANALOG_I

PxVehicleDriveTankControl::eMAX_NB_DRIVETANK_ANAL

- A new struct has been introduced to hold the enumerated lis PxVehicleDriveTank::eDRIVE_MODEL_STANDARD. The changes
 - PxVehicleDriveTank::eDRIVE_MODEL_STANDARD
 PxVehicleDriveTankControlModel::eSTANDARD
 - PxVehicleDriveTank::eDRIVE_MODEL_SPECIAL
 PxVehicleDriveTankControlModel::eSPECIAL
- A new struct has been introduced to hold the enumerated lise
 eVEHICLE TYPE DRIVE4W. The changes are
 - eVEHICLE TYPE DRIVE4W -> PxVehicleTypes::eDRIVE4\(\)
 - eVEHICLE TYPE DRIVETANK -> PxVehicleTypes::eDRIVE
 - eVEHICLE TYPE NODRIVE -> PxVehicleTypes::eNODRIV
 - eMAX_NUM_VEHICLE_TYPESPxVehicleTypes::eMAX_NB_VEHICLE_TYPES
- A new struct has been introduced to hold the enumerated lis 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_TIRELOAI
 - PxVehicleGraph::eCHANNEL_WHEEL_OMEGA
 PxVehicleWheelGraphChannel::eWHEEL_OMEGA

- PxVehicleGraph::eCHANNEL_TIRE_FRICTION
 PxVehicleWheelGraphChannel::eTIRE_FRICTION
- PxVehicleGraph::eCHANNEL_TIRE_LONG_SLIP
 PxVehicleWheelGraphChannel::eTIRE_LONG_SLIP
- PxVehicleGraph::eCHANNEL_NORM_TIRE_LONG_FORCE
 PxVehicleWheelGraphChannel::eNORM_TIRE_LONG_FOF
- PxVehicleGraph::eCHANNEL_TIRE_LAT_SLIP
 PxVehicleWheelGraphChannel::eTIRE_LAT_SLIP
- PxVehicleGraph::eCHANNEL_NORM_TIRE_LAT_FORCE
 PxVehicleWheelGraphChannel::eNORM_TIRE_LAT_FORCE
- PxVehicleGraph::eCHANNEL_NORM_TIRE_ALIGNING_M(
 PxVehicleWheelGraphChannel::eNORM_TIRE_ALIGNING_
- PxVehicleGraph::eMAX_NUM_WHEEL_CHANNELS
 PxVehicleWheelGraphChannel::eMAX_NB_WHEEL_CHAN
- A new struct has been introduced to hold the enumerated lis PxVehicleGraph::eCHANNEL_ENGINE_REVS. The changes are
 - PxVehicleGraph::eCHANNEL_ENGINE_REVS
 PxVehicleDriveGraphChannel::eENGINE_REVS
 - PxVehicleGraph::eCHANNEL_ENGINE_DRIVE_TORQUE
 PxVehicleDriveGraphChannel::eENGINE_DRIVE_TORQUE
 - PxVehicleGraph::eCHANNEL_CLUTCH_SLIP
 PxVehicleDriveGraphChannel::eCLUTCH_SLIP
 - PxVehicleGraph::eCHANNEL_ACCEL_CONTROL
 PxVehicleDriveGraphChannel::eACCEL_CONTROL
 - PxVehicleGraph::eCHANNEL_BRAKE_CONTROL
 PxVehicleDriveGraphChannel::eBRAKE_CONTROL
 - PxVehicleGraph::eCHANNEL_HANDBRAKE_CONTROL
 PxVehicleDriveGraphChannel::eHANDBRAKE_CONTROL

- PxVehicleGraph::eCHANNEL_STEER_LEFT_CONTROL
 PxVehicleDriveGraphChannel::eSTEER_LEFT_CONTROL
- PxVehicleGraph::eCHANNEL_STEER_RIGHT_CONTROL
 PxVehicleDriveGraphChannel::eSTEER_RIGHT_CONTROL
- PxVehicleGraph::eCHANNEL_GEAR_RATIO
 PxVehicleDriveGraphChannel::eGEAR_RATIO
- PxVehicleGraph::eMAX_NUM_ENGINE_CHANNELS
 PxVehicleDriveGraphChannel::eMAX_NB_DRIVE_CHANNE
- A new struct has been introduced to hold the enumerated lis 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 PxVehicleUp function argument. A simple example of how to construct, use, ar 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:

```
void updateVehicles(const PxF32 timestep, const PxVec3& gravi
    const PxVehicleDrivableSurfaceToTireFrictionPairs& fricPa
    PxVehicleWheels** vehicles, PxU32 numVehicles, std::vecto
{
    //Count the total number of wheels.
    unsigned int numWheels = 0;
    for(unsigned int i = 0; i < numVehicles; i++)
    {
        numWheels += vehicles[i]->mWheelsSimData.getNbWheels(
      }

    //Allocate buffers to store results for each vehicle and
```

```
PxVehicleWheelQueryResult* vehicleWheelQueryResults =
        new PxVehicleWheelQueryResult[numVehicles];
    PxWheelQueryResult* wheelQueryResults = new PxWheelQueryR
    PxU32 wheelCount = 0;
    for(PxU32 i = 0; i < numVehicles; i++)</pre>
        vehicleWheelQueryResults[i].nbWheelQueryResults =
            vehicles[i]->mWheelsSimData.getNbWheels();
        vehicleWheelQueryResults[i].wheelQueryResults = &whee
        wheelCount += vehicles[i]->mWheelsSimData.getNbWheel
    }
    //Update the array of vehicles.
    PxVehicleUpdates(timestep, gravity, fricPairs, numVehicle
        vehicleWheelQueryResults);
    //Test if each vehicle is in the air.
    for(PxU32 i = 0; i < numVehicles; i++)</pre>
    {
        if(!PxVehicleIsInAir(vehicleWheelQueryResults[i]))
        {
            for(PxU32 j = 0; j < vehicleWheelQueryResults[i].</pre>
                if(vehicleWheelQueryResults[i].wheelQueryResu
                {
                    hitActors.push_back
                         (vehicleWheelQueryResults[i].wheelQue
                }
            }
        }
    }
    delete[] vehicleWheelQueryResults;
    delete[] wheelQueryResults;
}
```

- The following accessors to non-persistent data associated with eac replaced as follows
 - PxVehicleWheelsDynData::getSuspLineStart
 PxWheelQueryResult::suspLineStart
 - PxVehicleWheelsDynData::getSuspLineDir
 PxWheelQueryResult::suspLineDir

- PxVehicleWheels::getSuspRaycast -> PxWheelQueryResu
 PxWheelQueryResult::suspLineDir, PxWheelQueryResult::s
- PxVehicleWheelsDynData::getTireDrivableSurfaceShape
 PxWheelQueryResult::tireContactShape
- PxVehicleWheelsDynData::getTireDrivableSurfaceMaterial
 PxWheelQueryResult::tireSurfaceMaterial
- PxVehicleWheelsDynData::getTireDrivableSurfaceType
 PxWheelQueryResult::tireSurfaceType
- PxVehicleWheelsDynData::getTireDrivableSurfaceContactPoint
 PxWheelQueryResult::tireContactPoint
- PxVehicleWheelsDynData::getTireDrivableSurfaceContactN
 PxWheelQueryResult::tireContactNormal
- PxVehicleWheelsDynData::getTireFriction
 PxWheelQueryResult::tireFriction
- PxVehicleWheelsDynData::getSuspJounce
 PxWheelQueryResult::suspJounce
- PxVehicleWheelsDynData::getSuspensionForce
 PxWheelQueryResult::suspSpringForce
- PxVehicleWheelsDynData::getTireLongitudinalDir
 PxWheelQueryResult::tireLongitudinalDir
- PxVehicleWheelsDynData::getTireLateralDir
 PxWheelQueryResult::tireLateralDir
- PxVehicleWheelsDynData::getTireLongSlip
 PxWheelQueryResult::longitudinalSlip
- PxVehicleWheelsDynData::getTireLatSlip ->PxWheelQueryF
- $\circ \ \ PxVehicleWheelsDynData::getSteer \ -> \ PxWheelQueryResu$
- PxVehicleWheels::isInAir -> PxWheelQueryResult::isInAir
- PxVehicleWheels::setWheelShapeMapping
 PxVehicleWheels::getWheelShapeMapping
 have
 been

- PxVehicleWheelsSimData::setWheelShapeMapping PxVehicleWheelsSimData::getWheelShapeMapping
- PxVehicleWheels::setSceneQueryFilterData
 PxVehicleWheels::getSceneQueryFilterData have been
 PxVehicleWheelsSimData::setSceneQueryFilterData
 PxVehicleWheelsSimData::getSceneQueryFilterData
- PxVehicle4WEnable3WTadpoleMode and PxVehicle4WEnable3 take an extra function argument: a non-const r PxVehicleWheelsDynData
- PxVehicleWheels::isInAir() has been replaced with Px\ PxVehicleWheelQueryResult& vehWheelQueryResults)
- PxVehicleDrive4WSmoothAnalogRawInputsAndSetAnalogInputs n function argument "const bool isVehicleInAir". This can be ca function PxVehicleIsInAir
- To improve api consistency PxVehicleTelemetryData::getNumWh PxVehicleTelemetryData::getNbWheelGraphs
- To improve api consistency PX_MAX_NUM_WHEI PX_MAX_NB_WHEELS
- To improve api consistency PxVehicleGraph::eMAX_NUM_TITL PxVehicleGraph::eMAX_NB_TITLE_CHARS
- PxVehicleTireData::mCamberStiffness has been PxVehicleTireData::mCamberStiffnessPerUnitGravity. PxVehicleTireData::mCamberStiffnessPerUnitGravity should be equivalent to the old value of PxVehicleTireData::mCamberStiffnessPerUnitGravity().
 advantage of using PxVehicleTireData::mCamberStiffnessPerUnitGravity().

 PxVehicleComputeTireForceDefault has been removed from the Custom tire shaders that call PxVehicleComputeTireForce implemented by taking a copy of PxVehicleComputeTireForceDefacopy instead.

- The mechanism to activate CCD per shape has change PxShapeFlag::eUSE SWEPT BOUNDS that was used in 3.2 to a its per shape has been removed. In place is PxRigidBodyFlag::eENABLE CCD that is set per rigid actor. Sett 3.3 actor approximately the effe has same PxShapeFlag::eUSE SWEPT BOUNDS on all the actor's shapes in
- PxPairFlag::eSWEPT_INTEGRATION_LINEAR has been PxPairFlag::eCCD_LINEAR in PhysX 3.3.
- PxSceneFlag::eENABLE_SWEPT_INTEGRATION flag in 3.2 has PxSceneFlag::eENABLE_CCD in PhysX 3.3.
- A simple example of how to enable CCD on a specific shape is demonstrates creating a body consisting of a large box and a smathe box is only used in discrete collision detection and the sphe CCD. The simulation filter shader shown here requires that the shapes be flagged with eCCD_RESPONSE to generate a (PxPairFlag::eCCD_LINEAR). Likewise, the filter shader shown he that the filter data of both shapes need to be flagged with eDISCF in order to generate a collision response (PxPairFlag::eRESOLV final remark is that the following shader requires that shapes of stat data with flags eDISCRETE_RESPONSE | eCCD_RESPONSE in and collision response from pairs that involve a static actor ar dynamic actor:

```
struct CCDFilterTest
{
    enum FilterFlags
    {
        eDISCRETE_RESPONSE = 1 << 0
        eCCD_RESPONSE = 1 << 1
    };</pre>
```

```
static PxFilterFlags filterShader(
        PxFilterObjectAttributes attributes0,
        PxFilterData filterData0,
        PxFilterObjectAttributes attributes1,
        PxFilterData filterData1,
        PxPairFlags& pairFlags,
        const void* constantBlock,
        PxU32 constantBlockSize)
    {
        pairFlags = PxPairFlags(0);
        PxU32 combo = filterData0.word0 & filterData1.word0;
        if(combo & eDISCRETE_RESPONSE)
        {
            pairFlags |= PxPairFlag::eRESOLVE_CONTACTS;
        if(combo & eCCD_RESPONSE)
            pairFlags |= PxPairFlag::eCCD_LINEAR;
        return PxFilterFlags();
    }
};
PxRigidDynamic* dyn = getPhysics().createRigidDynamic(PxTrans
PxBoxGeometry box;
box.halfExtents = PxVec3(1.f, 1.f, 1.f);
PxSphereGeometry sphere;
sphere.radius = 0.75f;
PxShape* boxShape = dyn->createShape(box, getDefaultMaterial(
PxShape* sphereShape = dyn->createShape(sphere, getDefaultMat
PxFilterData data = boxShape->getSimulationFilterData();
data.word0 |= CCDFilterTest::eDISCRETE_RESPONSE;
boxShape->setSimulationFilterData(data);
data = sphereShape->getSimulationFilterData();
data.word0 |= CCDFilterTest::eCCD_RESPONSE;
sphereShape->setSimulationFilterData(data);
dyn->setRigidBodyFlag(PxRigidBodyFlag::eENABLE_CCD, true);
getActiveScene().addActor(*dyn);
```

PhysX Visual Debugger

- A new flag has been introduced to configure the visualizing of cons
 PxVisualDebuggerFlag::eTRANSMIT CONSTRAINTS;
- A new function has been introduced to configure PxVisualDebugge
 PxVisualDebugger::setVisualDebuggerFlags(PxVisualDebugge
- A new funtion has been introduced to send error stream to PVD:
 PxVisualDebugger::sendErrorMessage((PxErrorCode::Enum c message, const char* file, PxU32 line);
- The following functions were renamed:

PxVisualDebugger::getPvdConnectionFactory()
PxVisualDebugger::getPvdConnection(); PxVisualDebugger::getPvdDataStream();

- The PVD connect function changed to the same method as previous PxVisualDebuggerExt::connect -> PxVisualDebuggerExt::create
- The constraint, contacts and scene queries visualizing can all PxVisualDebuggerFlag in 3.3. Here is an example for how to enathe contacts:

mPhysics->getVisualDebugger()>setVisualDebuggerFlags(PxVisualDebuggerFlag::eTRANSMIT_
PxVisualDebuggerFlag::eTRANSMIT_CONSTRAINTS);

PhysX Cloth

There have been substantial changes to the PhysX 3.3 cloth so performance and behavior. This has resulted in a reorganization of his stored and processed in the cloth fabric. Prior to PhysX 3.3 the cloth so organize edge constraints into independent groups. In PhysX 3.3 it is not to decompose constraints into fibers, instead edge constraints now exare solved in larger, independent sets. Interface changes are detailed by

- Previously there were multiple solver types to choose from 1 constraints such as eFAST, eSTIFF, eBENDING, PxClothPhaseSolverConfig::SolverType). There is now one type of constraints, this is a flexible distance constraint with controls to ad certain ranges of compression and stretch (see PxClothStretch such as bending are now achieved by the way distance constraint geometrically, rather than through a specialized bending solver.
- To reduce stretching a new constraint type has been added called '
 These constraints do not act along edges of the mesh, but attachments between particles that enforce a maximum distance b
 See PxClothFabric::getTetherAnchors().
- Cloth cooking which was previously part of the PxCooking library I the extension library, see PxClothFabricCooker:

```
// PhysX 3.2.x
cooking->cookClothFabric(meshDesc, gravity, outputStream);
// PhysX 3.3
PxClothFabricCooker cooker(meshDesc, gravity, useGeodesicTeth cooker.save(outputStream, false);
```

• The PxClothCollisionData parameter has been removed from PxPl The collision shapes can now be added after cloth PxCloth::addCollisionSphere and PxCloth::addCollisionCapsule.

- PxCloth::wakeUp() does not have a parameter anymore. Use the method setWakeCounter() instead to set a specific value.
- PxCloth::setDampingCoefficient now takes a PxVec3 instead of a the damping per axis.
- PxCloth::setPhaseSolverConfig() has been renamed to PxCloth::se
- PxCloth::lockClothReadData() has been renamed to PxCloth::lockF
- PxClothFabricTypes.h has been removed, this header has t PxClothFabric.h

RepX Serialization

Substantial changes were made to the PhysX 3.3 serialization inte collections and references between collections have been unified for serialization.

- The RepX and RepXUpgrader libraries have been removed. RepX provided through PhysXExtensions.
- RepXCollection has been replaced with PxCollection, which is the class for both RepX and binary serialization in 3.3. Collections are deserialization with PxSerialization::createCollectionFromXml(). En be created with PxCreateCollection(). Serialization into RepX f through PxSerialization::serializeCollectionToXml().
- TRepXId has been replaced with PxSerialObjectId.
- RepXIdToRepXObjectMap and RepXObject have been representation of the serial serial pxSerialObjectId values.
- RepXExtension was removed. Serialization and deserialization of sachieved through the PxRepXSerializer interface.
- RepXUtility and PxToolkit functionality has been replaced with v PxSerialization, PxCollection and PxScene.
 - A PxCollection with all PxPhysics-level objects such as s materials (formally referred to as buffers) can b PxCollectionExt::createCollection(PxPhysics&).
 - Similarly PxCollectionExt::createCollection(PxScene&) can a collection of PxScene-level objects.
 - o Dependencies between objects and collections can

- PxSerialization::complete().
- The objects of a collection can be added to PxScene::addCollection().
- Operations on files are generally handled with all PxOutputStream and PxInputData. Default implementation PxDefaultFileOutputStream and PxDefaultFileInputData.
- RepXUpgrader::upgradeCollection was removed. RepX data ca newer PhysX versions by deserializing and re-serializing PxSerialization::createCollectionFromXml(), PxSerialization::serializeCollectionToXml().
- Serialization functionality requires a PxSerializationRegistry instacreated 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, PxClos cleanup.
- Custom class RepX serialization is supported in 3.3, more inforr Serialization.

Binary Serialization

The binary serialization interface has been refactored and unifie serialization interface.

- Most serialization functionality requires an instance of PxSerializationRegistry. It is application managed and can PxSerialization::createSerializationRegistry() and re PxSerializationRegistry::release().
- The base class for serializable types has been renamed from PxBase. Most of the serialization functionality moved to a separate A PxSerializer instance per serializable type is re-PxSerializationRegistry. All PhysX and PhysXExtension serializal by default.
- PxCollection has been reworked.
 - PxCollection::serialize() and PxCollection::deserialize() w
 PxSerialization::createCollectionFromBinary()
 PxSerialization::serializeCollectionToBinary() in PhysXExte
 - PxSerializable::collectForExport() has been replaced with I PxSerialization::complete() helps to add required objected dependencies. PxSerializable::isSerializable() should be u a collection can be successfully serialized.
- PxUserReferences was removed: PxCollection instances can now resolve dependencies between collections on PxSerialization::complete() supports creating collections with exte to other collections.
- PxSerialObjectRef has been replaced with PxSerialObjectId.
- PxCollectForExportSDK() and PxCollectForExportScene() function

- ŗ
- PxDumpMetaData() was replaced with PxSerialization::dumpBinary
- PxBinaryConverter moved **PhysXCooking** The from to PxCooking::createBinaryConverter() repla was PxSerialization::createBinaryConverter().
- PhysXVehicles supports binary serialization. PxSerializationReg provided to PxInitVehicleSDK() for vehicle serialization, PxClos cleanup.
- Custom class binary serialization is supported in 3.3, more inforr Serialization.

PhysX TaskManager

 The pxtask namespace has been removed and all it's types are r physx namespace with a Px prefix, for example pxtask::LightCpi physx::PxLightCpuTask

NVIDIA PhysX SDK 3.4.1 Documentation » previous | next User's Guide »

NVIDIA PhysX SDK 3.4.1 Documentation » User's Guide »

previous

- This guide highlights all significant parts of the API that have char release. An application with a working integration of the older version be able to easily migrate to the newer version by following these pc
- Functionality shared with the APEX SDK was moved into a se 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 remove PxRigidActorExt::createExclusiveShape() replaces this method.

PxSceneFlag::eDEPRECATED_TRIGGER_TRIGGER_REPC

PxSceneFlag::eDEPRECATED_TRIGGER_TRIGGER_REPORTS is do 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. 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 pa receives a pointer to a PxPvd object, used for connecting Phy debugger.
- PxActor::isRigidStatic, isRigidDynamic, isParticleSystem, isArticulationLink, isCloth, isRigidActor, isRigidBody, isParticle removed. Use corresponding PxBase::is() with class template p casting.
- PxContactPairFlag::eINTERNAL_HAS_FACE_INDICES is obsole removed.
- Trigger shapes will no longer send notification events for interaction shapes. For PhysX 3.4 there is the option to re-enable the PxSceneFlag::eDEPRECATED_TRIGGER_TRIGGER_REPORTS no longer be available in PhysX 3.5. It is recommended to eDEPRECATED_TRIGGER_TRIGGER_REPORTS and instead u non-trigger shape, both with the same geometry and local pose, notifications for overlaps between trigger shapes.
- Implementations of PxSimulationEventCallback will have to primplementation of the newly added method onAdvance() to avoid c
- The deprecated method PxPhysics::createHeightField(const Px has been removed. Please use PxCooking::create PxHeightFieldDesc&, PxPhysicsInsertionCallback&) instead. The can be obtained through PxPhysics::getPhysicsInsertionCallback().
- The deprecated flag PxActorTypeSelectionFlag/PxActorTypeSelec removed. Please use PxActorTypeFlag/PxActorTypeFlags instead.

- The deprecated class PxFindOverlapTriangleMeshUtil has been re PxMeshOverlapUtil instead.
- The deprecated flag PxConstraintFlag::eREPORTING has been reports are now always generated.
- The deprecated flag PxConstraintFlag::eDEPRECATED_32_CC been removed.
- PxRegisterHeightFields() now registers unified heightfields. T heightfields, call PxRegisterLegacyHeightFields(). Legacy heig deprecated and will be removed in a future PhysX release.
- The following deprecated simulation event flags have been remove

 - PxContactPairFlag::eDELETED_SHAPE_0, ::eDELETED_S
 (use PxContactPairFlag::eREMOVED_SHAPE_0, ::eREMC instead)
- PxContactPair now reports separate pointers for contactPatches, contactImpulses rather than reporting a single pc PxContactStreamIterator parses. The interface for PxContactStreamodified accordingly. See the PxContactPair::extractContacts in

further guidance on how to iterate over this contact data if required.

Contact Generation

- PCM contact generation is now used by default. Legacy § generation can be re-enabled by clearing the PxSceneFlag::eE PxSceneDesc::flags.
- Unified heightfields are now the default heightfield collision approx mirrors the way in which mesh contact gen functions so per heightfields to be used interchangeably with negligible behavior legacy heightfield collision approach can be use PxRegisterLegacyHeightFields().
- When unified heightfields are in use, the bounds of heightfield extruded by "thickness". If legacy heightfield collision is used, the extruded by thickness.

PhysX Cooking

- The deprecated
 PxMeshPreprocessingFlag::eREMOVE_UNREFERENCED_VERT
 ::eREMOVE_DUPLICATED_TRIANGLES have been removed. M
 up by default unless PxMeshPreprocessingFlag::eDISABLE CLEA
- PxCookingParams::meshSizePerformanceTradeOff
 PxCookingParams::meshCookingHint have been moved to PxBVI since they only affect the BVH33.
- The PxGaussMapLimit.h file has been I PxGetGaussMapVertexLimitForPlatform function has been move but the function is now deprecated, along with the PxPlatform enu now an explicit PxCookingParams::gaussMapLimit parameter. As PhysX 3.4 is concerned there is nothing to do other than rer PxGaussMapLimit.h, and perhaps including PxCooking.h instead if
- Legacy convex hull
 (PxConvexMeshCookingType::eINFLATION_INCREMENTAL_HUL all cases. To cook a convex mesh without inflation PxConvexMeshCookingType::eQUICKHULL algorithm must be us does not support inflation.

Reference Counting

- In previous releases, isReleasable() for shareable objects (shape convex meshes, cloth fabrics, materials and heightfields) would release() had been called on the object, which was only allowed reference counts can be manually incremented with acquired decremented with release(), and so the fact that release() has carreliable indicator of whether it can be called again.
- As a consequence of the above, applications must ensure they counted reference to each shareable object in a collection PxCollectionExt::releaseObjects. The main case in which this might is when using PxRigidActor::createShape(), since in that case or counted reference to the shape. In this specific case, the releaseExclusiveShapes to PxCollectionExt::releaseObjects may b
- Since there is no unique user release for shareable objects, the USER RELEASE events when release() is called.

PhysX Visual Debugger

• PxVisualDebugger is deprecated, and new PxPvd has been introc are mentioned in *PhysX Visual Debugger (PVD)*.

Scene queries

- PxPruningStructure enum has been renamed to PxPruningStructur
- Deprecated type PxSceneQueryHit has been removed. Pleasinstead.
- Deprecated type PxSceneQueryFilterData has been remove PxQueryFilterData instead.
- Deprecated type PxSceneQueryFilterCallback has been removed.
 PxQueryFilterCallback instead.
- Deprecated type PxSceneQueryCache has been remove PxQueryCache instead.
- Deprecated types PxSceneQueryFlag(s) has been removed. Pleas instead.
- Deprecated scene query functions have been removed (e.g. PxSi etc). To make the transition easier they are still available in PxSc part of PhysXExtensions. A previous PxScene::raycastAny(...) call use PxSceneQueryExt::raycastAny(PxScene, ...), or PxScene::raycastAny(PxScene, ...)
- PxHitFlag::eFACE_INDEX was introduced. In order to receive sweeps against convex geometry, the flag needs to be set.
- PxHitFlag::eDISTANCE has been deprecated, since the distance and its computation cannot be skipped. Please simply avoid using on. The flag has no effect and it will be removed in the next version
- The "anyHit" parameter of the PxGeometryQuer PxShapeExt::raycast() functions has been removed.
 PxHitFlag::eMESH_ANY instead.
- PxMeshQuery::sweep() now respects PxHitFlag::eMESH_BOTH_ previously used that flag when calling that function, it was ignored,
 3.4 might start generating different results compared to 3.3. If ke behaviour is important, please disable PxHitFlag::eMESH_BOT

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
- Spatial index feature is marked as deprecated, it will be removed in

PxExtensions

• The signatures for the PxComputeMeshPe PxComputeHeightFieldPenetration functions have changed. The olavailable but they are now deprecated. It is recommended to trafunctions (with the same names but a different signature).

NVIDIA PhysX SDK 3.4.1 Documentation » User's previous Guide »



Built December 15, 2017

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::ePOSTFILTEF
 - PxQueryFlag::eANY_HIT
 - PxQueryFlag::eNO BLOCK
 - PxFilterData fixed function filtering
- User defined hit callbacks for unbounded results
- Batched queries
- Volume Caching
 - Filling the Cache
 - Querying the Cache
- Single Object Caching
- PxPruningStructureType
 - Generalities
 - PxPruningStructureType::eNONE
 - PxPruningStructureType::eSTATIC AABB TREE
 - PxPruningStructureType::eDYNAMIC_AABB_TREE
- PxPruningStructure
 - Merge process
- Vehicles
 - Introduction
 - Algorithm
 - First Code

- Vehicle SDK Initialization
- Introduction To Vehicle Creation
- Introduction To Vehicle Update
- Snippets
 - SnippetVehicle4W
 - SnippetVehicleTank
 - SnippetVehicleNoDrive
 - SnippetVehicleScale
 - SnippetVehicleMultiThreading
- Advanced Concepts
 - Vehicle Creation
 - Filtering
 - Tire Friction on Drivable Surfaces
 - Vehicle Controls
 - Vehicle Update
- More Advanced Concepts
 - Vehicle Telemetry
 - Vehicle Update Multi-Threaded
 - Tire Shaders
 - Vehicle Types
 - SI Units
 - Level of Detail
 - Wheel Contact Beyond Raycasts
- Tuning Guide
 - PxVehicleWheelData
 - PxVehicleWheelsSimData
 - PxVehicleSuspensionData
 - PxVehicleAntiRollBar
 - PxVehicleTireData
 - PxVehicleEngineData

- PxVehicleGearsData
- PxVehicleAutoBoxData
- PxVehicleClutchData
- PxVehicleAckermannGeometryData
- PxVehicleTireLoadFilterData
- PxVehicleDifferential4WData
- PxRigidDynamic
- Troubleshooting
 - Jittery Vehicles
 - The Engine Rotation Refuses To Spin Quickly
 - The Engine Spins But the Wheels Refuse To Spin
 - The Wheels Are Spinning But The Vehicle Does Not Move
 - The Vehicle Does Not Steer/Turn
 - The Acceleration Feels Sluggish
 - The Vehicle Does Not Slow Down When Not Accelerating
 - The Vehicle Turns Too Quickly/Too Slowly
 - The Wheels Spin Too Much Under Acceleration
 - The Wheels Spin Too Much When Cornering
 - The Vehicle Never Goes Beyond First Gear
 - The Vehicle Under-steers Then Over-steers
 - The Vehicle Slows Down Unnaturally
 - The Vehicle Climbs Too Steep Slopes

References

- Anti-roll Bar
- Tire Modeling
- Engine Torque Curves
- Differentials
- Clutch
- Ackermann Steer Correction
- Tanks

- General
- Character Controllers
 - Introduction
 - Kinematic Character Controller
 - Creating a character controller
 - Overlap Recovery Module
 - Character Volume
 - Volume Update
 - Moving a Character Controller
 - Graphics Update
 - Auto Stepping
 - Climbing Mode
 - Up Vector
 - Walkable Parts & Invisible Walls
 - Obstacle Objects
 - Hit Callback
 - Behavior Callback
 - Character Interactions: CCT-vs-dynamic actors
 - Character Interactions: CCT-vs-CCT
 - Hidden Kinematic Actors
 - Time Stepping
 - Invalidating Internal Geometry Caches
 - Runtime Tessellation
 - Troubleshooting
 - Character goes through walls in rare cases
 - Tessellation performance issue
 - The capsule controller manages to climb over obstacles h
 offset value
- Particles (deprecated)
 - Introduction

- Creating Particle Systems
- Particle Management
 - Creating Particles
 - Releasing Particles
 - Index Pool Extension
 - Updating Particles
 - Reading Particles
- Parameter Guide
 - Particle Dynamics
 - Collision with Rigid Actors
 - Fluid (PxParticleFluid)
- Collision Handling
 - Collision Filtering
 - Per-particle Rest Offsets
 - Particle Drains
- Best Practices / Troubleshooting
 - Particle Grid and Spatial Data Structure Overflow
- GPU/CUDA Acceleration
- Sample Discussion
- References
- Cloth
 - Introduction
 - Simulation Algorithm
 - Solver Frequency
 - Particle Integration
 - Local Space Simulation and Inertia Scale
 - Constraints
 - Distance Constraints
 - Tether Constraints
 - Motion Constraints

- Separation Constraints
- Collision Detection
- Continuous Collision Detection
- Virtual Particle Collision
- Friction and Mass Scaling
- Self-Collision of a Single Cloth Actor
- Inter-Collision between Multiple Cloth Actors
- Best Practices / Troubleshooting
 - Performance Tips
 - Using GPU Cloth
 - Fast-Moving Characters
 - Avoiding Stretching
 - Avoiding Jitter
 - PVD Support
- Snippet Discussion
 - Filling in PxClothMeshDesc
 - Creating Fabric
 - Creating Cloth
 - Specifying Collision Shapes
 - Adding Virtual Particles
 - Accessing Particle Data
- References
- Debug Visualization
 - Introduction
 - Usage
- PhysX Visual Debugger (PVD)
 - o 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
- o CCD
- PhysX Visual Debugger
- PhysX Cloth
- RepX Serialization
- Binary Serialization
- PhysX TaskManager
- Migrating From PhysX SDK 3.3 to 3.4
 - Deprecated APIs
 - PxRigidActor::createShape
 - PxSceneFlag::eDEPRECATED_TRIGGER_TRIGGER_RI
 - PhysX particles
 - PhysX cloth
 - Core PhysX
 - Contact Generation
 - PhysX Cooking
 - Reference Counting
 - PhysX Visual Debugger
 - Scene queries
 - PxExtensions