
Contents:

- **User's Guide**
 - The Basics
 - Startup and Shutdown
 - Scenes, Materials, and Actors
 - Shapes and Geometries
 - Joints
 - Rigid Body Dynamics
 - Data Access and Buffering
 - Task Management
 - GPU Resource Management
 - Scene Queries
 - Geometry Queries
 - Callbacks and Customization
 - Aggregates
 - Serialization
 - Simulation Statistics
 - Advanced Rigid Body Topics
 - Persistent Contact Manifold
 - Coulomb Friction
 - Particles
 - Cloth
 - Vehicles
 - Character Controllers
 - PhysX Visual Debugger (PVD)
 - Migration Guide From PhysX SDK 2.x to 3.x
 - Migrating from PhysX SDK 3.1 to 3.2

Welcome to the PhysX SDK version 3! With this second major rewrite of the SDK, we are excited to bring you a great number of enhancements, including numerous API improvements. Because so much has changed in the API, we recommend even experienced PhysX users to read through this guide to familiarize themselves with the new programming interface.

Users migrating from PhysX 2 will find the *Migration Guide From PhysX SDK 2.x to 3.x* chapter of particular interest.

Building and Running the Samples on Windows

This guide explains the code from a series of samples:

- The **Hello World Sample** shows a minimal physx setup.
- The **North Pole Sample** demonstrates basic functionality covered in *Startup and Shutdown*, *Scenes, Materials, and Actors*, *Shapes and Geometries*, and *Rigid Body Dynamics*.
- The **Bridges Sample** demonstrates joints, described in the chapter *Joints*, as well as the character controller explained in *Character Controllers*.
- The **Vehicle Sample** demonstrates the use of vehicles described in the chapter *Vehicles*.
- The **Particles Sample** demonstrates the use of particle effects, which is the topic of *Particles*.
- The **Character Cloth Sample** demonstrates clothing functionality, described in the chapter *Cloth*.
- The **Custom Gravity Sample** demonstrates using the character controller in a context where the vertical direction is not fixed.
- Finally, the **Submarine Sample** shows off a number of advanced features, which receive a treatment in the chapters *Scene Queries*, *Callbacks and Customization*, *Advanced Rigid Body Topics*, and *Task Management*.

On Windows, PhysX requires either Visual Studio 2008 with Service Pack 1 or higher, or Visual Studio 2010.

To see the samples, open the Visual Studio solution called `Samples.sln`. This includes the samples listed above within the project `Samples`, plus some additional projects which implement common operations.

Build Settings

While not particularly important for understanding the Sample code, to build your own PhysX app, you will need to add some include paths and libraries to your project makefile or IDE. The include files and libraries are located in the root folders "Include" and "Lib" respectively. A number of these include directories and libraries are of course only needed if the corresponding component, such as the Vehicle library, is used. You will also need to add the appropriate platform specific extension for libs (e.g. ".lib" or ".a") and the appropriate relative path prefixes (e.g. "../..") for includes.

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

Library Redistribution

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

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

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

Build Configurations

The SDK has three build configurations available to all licensees, designed for different stages of development and deployment.

- the *checked* build contains code to detect invalid parameters, API race conditions, and other incorrect uses of the API which might otherwise cause mysterious crashes or failures in simulation.
- the *profile* build omits the checks, but still has PVD and memory instrumentation.
- the *release* build is built for minimal footprint and maximum speed. It omits most checks and instrumentation.

Simulation works the same way in all of them, and all are compiled with high optimization levels. In addition, you can build components of PhysX which ship with source in a debug configuration, which is compiled without optimizations. The debug build can be useful for error analysis, but contains asserts used for SDK development which some customers may find too intrusive for daily use.

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

Note: PhysX libraries of different build configurations (e.g. the DEBUG version of PhysXVehicle and the CHECKED version of PhysXVisualDebuggerSDK) should never be mixed in an application because this will also result a CRT conflict.

The API

The PhysX SDK exposes an application programming interface (API) composed primarily of abstract interface classes. The header files that belong to the public API are located in a top level directory named "Include". Classes, enumerations and functions defined by the public API have the prefix Px. We are committed to keeping this API stable and backwards-compatible from one minor release to the next, to protect the investment you make in your integration code.

Note: There are currently two sections of the public API which do not have the Px prefix: the RepX serialization library for structured data which has the prefix RepX, and the PhysX Visual Debugger connection library which has the prefix Pvd.

The PhysX libraries also expose some classes and functions that are not part of the public API, such as container and platform abstractions that we find practical to reuse in our samples. These are largely undocumented, and can be recognized because they do not have the Px prefix of the public API. While they are technically available to users, most users either will not need them or will have their own versions, and we are not committed to backwards-compatibility of this code between PhysX versions.

Multithreading

PhysX provides efficient multithreaded implementations of its simulation and raycasting functionality. See [Task Management](#) for details of integrating PhysX threading into an application.

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

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

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

Read and write calls may be made to objects in a scene that is being simulated. See [Data Access and Buffering](#) for details.

PhysX Visual Debugger

The PhysX Visual Debugger (PVD) provides a graphical view of the PhysX scene together with various tools to inspect and visualize variables of every PhysX object. Additionally it can also record and visualize memory and timing data. See [PhysX Visual Debugger \(PVD\)](#) for details.

Startup functionality is common to all of the samples and is shared in the project *SampleBase*. Open the main file *PhysXSample.cpp* and navigate to the function *onInit*, which the sample calls on startup to initialize PhysX.

Note that the file includes the entire PhysX API in a single header. You may also selectively include just the headers you need, but *PxPhysicsAPI.h* includes everything to help you get started faster:

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

First, create a *PxFoundation* object:

```
static PxDefaultErrorCallback gDefaultErrorCallback;
static PxDefaultAllocator gDefaultAllocatorCallback;

mFoundation = PxCreateFoundation(PX_PHYSICS_VERSION, gDefaultAl
if(!mFoundation)
    fatalError("PxCreateFoundation failed!");
```

Every PhysX module requires a *PxFoundation* instance to be available. The required parameters are a version ID, an allocator callback and an error callback. *PX_PHYSICS_VERSION*, is a macro predefined in our headers to enable PhysX to check for a version mismatch between the headers and the corresponding SDK DLLs. Usually, the allocator callback and error callback are specific to the application, but PhysX provides default implementations that make it easy to get started. The sections below will describe these two parameters in more detail. (The actual sample code supports an advanced memory allocator that tracks allocations instead of the default, but we have omitted that detail here.)

Now create the top-level *PxPhysics* object:

```
bool recordMemoryAllocations = true;
mProfileZoneManager = &PxProfileZoneManager::createProfileZoneM
if(!mProfileZoneManager)
```

```
fatalError("PxProfileZoneManager::createProfileZoneManager  
mPhysics = PxCreatePhysics(PX_PHYSICS_VERSION, *mFoundation,  
PxTolerancesScale(), recordMemoryAllocations, mProf  
if(!mPhysics)  
fatalError("PxCreatePhysics failed!");
```

Again, the version ID has to be passed in. The `PxTolerancesScale` parameter makes it easier to author content at different scales and still have PhysX work as expected, but to get started simply pass a default object of this type. The `recordMemoryAllocations` parameter specifies whether to perform memory profiling. The optional profile zone manager, typically created with `PxProfileZoneManager::createProfileZoneManager()`, enables the performance profiling capabilities of the PhysX Visual Debugger.

The Allocator Callback

In order to perform dynamic allocation, the SDK requires an implementation of the *PxAllocatorCallback* interface. PhysX provides default implementations for all supported platforms. Here is the windows version:

```
#include <malloc.h>
class PxDefaultAllocator : public PxAllocatorCallback
{
    void* allocate(size_t size, const char*, const char*, int)
    {
        return _aligned_malloc(size, 16);
    }

    void deallocate(void* ptr)
    {
        _aligned_free(ptr);
    }
};
```

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

The three unused parameters to `allocate()` are an identifier which identifies the type of allocation, and the `__FILE__` and `__LINE__` location inside the SDK code where the allocation was made. Refer to `PxAllocatorCallback::allocate()` to find out more about them.

The Error Callback

The error callback *PxErrorCallback* is a user-defined class which the SDK requires in order to pass error messages to the application. There is only a single function to implement, `reportError`. This function should log the passed message, or print it on the application's output console. For the more serious error codes `eABORT`, `eINVALID_PARAMETER`, `eINVALID_OPERATION`, and `eOUT_OF_MEMORY`, breaking into the debugger may be a more appropriate choice. Whatever you do, do not just ignore the messages.

Again PhysX provide a complete default implementation, which will print the error message. The error code `PxErrorCode::eABORT` represents an unrecoverable error. In order that this error not be missed, the default implementation will print the message once every second.

Cooking

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

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

The `PxCookingParams` struct configures the cooking library to target different platforms, use non-default tolerances or produce optional outputs.

The cooking library generates data through a streaming interface. In the samples, implementations of streams are provided in the `PxToolkit` library to read and write from files and memory buffers.

Extensions

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

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

Optional SDK Components

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

- Articulations
- Height Fields

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

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

    if(!physics)
        return NULL;

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

    return physics;
}
```

Note that this will only save memory when linking PhysX as a static library, as we rely on the linker to strip out the unused code.

Windows delay load dll

The PhysXCommon dll is marked as delay loaded inside of the PhysX and PhysXCooking project. So it is possible to have a delay loaded PhysXCommon, PhysX and PhysXCooking dlls. If you need to load a different dll, it is possible to create a PxDelayLoadHook and define the name of PhysXCommon dll that should be loaded by PhysX dll and PhysXCooking dll, please see example:

```
class SampleDelayLoadHook: public PxDelayLoadHook
{
    virtual const char* GetPhysXCommonDEBUGDllName() const
    virtual const char* GetPhysXCommonCHECKEDDllName() const
    virtual const char* GetPhysXCommonPROFILEDDllName() const
    virtual const char* GetPhysXCommonDllName() const { ret
} gDelayLoadHook;
```

Now the hook must be set to PhysX and PhysXCooking:

```
PxDelayLoadHook::SetPhysXInstance(&gDelayLoadHook);
PxDelayLoadHook::SetPhysXCookingInstance(&gDelayLoadHook);
```

Shutting Down

To dispose of any PhysX object, call its `release()` method. This will destroy the object, and all contained objects. The precise behavior depends on the object type being released, so refer to the reference guide for details. To shut down physics entirely, simply call `release()` on the `PxPhysics` object, and this will clean up all of the physics objects:

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

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

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

The most important PhysX objects are *scenes* and *actors*. A scene is PhysX' representation of the world, and actors are the individual elements of that world. To simulate a physical world, create a scene and populate it with actors. A scene also supports geometric queries, such as raycasts and volume overlap checks, against the actors it contains. See [Scene Queries](#) for more details.

The basic use of rigid actors is outlined below and discussed further in [Rigid Body Dynamics](#), see the chapters [Cloth](#) and [Particles](#) for other types of actors.

The Scene

The PxScene object is the representation of the world in PhysX. Creating the scene requires a number of immutable parameters to be specified in the PxSceneDesc struct. The values of these parameters may vary between samples, so each sample has the opportunity to set values using the customizeSceneDesc() function:

```
static PxDefaultSimulationFilterShader gDefaultFilterShader;

PxScene* mScene;

PxSceneDesc sceneDesc(mPhysics->getTolerancesScale());
sceneDesc.gravity = PxVec3(0.0f, -9.81f, 0.0f);
customizeSceneDesc(sceneDesc);

if(!sceneDesc.cpuDispatcher)
{
    mCpuDispatcher = PxDefaultCpuDispatcherCreate(mNbThreads);
    if(!mCpuDispatcher)
        fatalError("PxDefaultCpuDispatcherCreate failed!");
    sceneDesc.cpuDispatcher = mCpuDispatcher;
}
if(!sceneDesc.filterShader)
    sceneDesc.filterShader = &gDefaultFilterShader;

#ifdef PX_WINDOWS
if(!sceneDesc.gpuDispatcher && mCudaContextManager)
{
    sceneDesc.gpuDispatcher = mCudaContextManager->getGpuDispat
}
#endif

mScene = mPhysics->createScene(sceneDesc);
if (!mScene)
    fatalError("createScene failed!");
```

For mandatory fields, default values are set if not supplied by the application:

- a realistic gravity vector to act along the -y axis of the world.
- the SDK's default implementation of the CpuDispatcher object, which maps simulation tasks to threads. mNbThreads is the number of threads that it should use -- in this sample we set it to 1.
- on windows, a GpuDispatcher to use for CUDA-accelerated features.
- the SDK's default implementation of PxSimulationFilterShader, a user-definable collision filtering mechanism.

PxPhysics::createScene() then creates the scene object.

The Simulation Loop

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

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

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

    mAccumulator -= mStepSize;

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

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

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

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

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

It is possible to read and write from the scene during simulation. The samples take advantage of this to perform rendering work in parallel with

physics. Until `fetchResults()` returns, the results of the current simulation step are not available. So running rendering in parallel with simulation renders the actors as they were when `simulate()` was called. After `fetchResults()` returns, all these functions will return the new, post-simulate state. See the chapter *Data Access and Buffering* for more details about reading and writing while the simulation is running.

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

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

Simulation Memory

Much of the memory PhysX uses for simulation is held in a pool of blocks, each 16K in size. You can control the current and maximum size of the pool with the `nbContactDataBlocks` and `maxNbContactDataBlocks` members of `PxSceneDesc`. PhysX will never allocate more than the maximum number of blocks specified, and if there is insufficient memory it will instead simply drop contact or joint constraints. You can reclaim unused blocks with the scene's `flush()` method, find out how many blocks are currently in use with the `getNbContactBlocksUsed()` method, and find out the maximum number that have ever been used with the `getMaxNbContactDataBlocksUsed()` method.

In order to minimize the allocations performed during simulation, you may provide physx with a memory block in the `simulate()` call. This block will be used for allocation of temporary data during simulation. Its size must be a multiple of 16K, and it must be 16-byte aligned.

Note that there are currently special restrictions for PS3, which are discussed in the platform-specific section of this guide.

Materials

All physical objects have at least one material, which defines the friction and restitution properties used to resolve a collision with the objects. To create a material, call `PxPhysics::createMaterial()`:

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

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

PhysX objects whose collision geometry is a triangle mesh or a heightfield (see *Shapes and Geometries*) can have a material per triangle.

Rigid Actors

Rigid actors are of two principal kinds: *static* and *dynamic*, corresponding to the PhysX classes `PxRigidStatic` and `PxRigidDynamic`. Static actors are immovable by the simulation, whereas dynamic actors have their positions updated by the simulation when `simulate()` is called. Dynamic actors may be controlled either directly by the application updating their position on a frame-by-frame basis (such actors are called *kinematic*), or by the simulation engine according to Newton's Laws of Motion.

PhysX provides helper methods in `PxSimpleFactory.h` to quickly create simple actors. Actors may also be imported using a binary serialization mechanism (see chapter [Serialization](#)) which bypasses much of the processing involved in actor creation. To procedurally create and simulate more complex actors, follow these steps:

1. create a `PxRigidStatic` or `PxRigidDynamic` object through the `PxPhysics` object, specifying its pose (orientation and position) in the world
2. create one or more `PxShapes` to define the collision geometry of the actor
3. for dynamics, update the mass and inertia properties of the actor
4. customize the properties of the actor and shapes as necessary
5. add the actor to the scene

For example, to throw a simulated sphere from a specific position with an initial speed, create a dynamic actor following the first three steps above:

```
PxRigidDynamic* aSphereActor = thePhysics->createRigidDynamic(PxShape* aSphereShape = aSphereActor->createShape(PxSphereGeomePxRigidBodyExt::updateMassAndInertia(*aSphereActor, sphereDensi
```

or equivalently:

```
PxRigidBody* aSphereActor = PxCreateDynamic(*thePhysics, PxMaterial, sphereDensity);
```

Then specify an initial linear velocity vector:

```
aSphereActor->setLinearVelocity(velocity);
```

And add the actor to a scene:

```
aScene->addActor(*aSphereActor);
```

To create a static ground plane and add it to the simulation:

```
PxRigidBody* plane = PxCreatePlane(*mPhysics, PxPlane(PxVec3(
if (!plane)
    fatalError("create shape failed!");
mScene->addActor(*plane);
```

PxShape and PxGeometry classes are described in more detail in [Shapes and Geometries](#). For more about specifying mass properties, see [Rigid Body Dynamics](#).

Simulation Clients

The so-called multi-client functionality of the SDK strives to solve problems that can arise if the PhysX SDK is used in an environment with multiple distinct software components. It makes it possible to selectively hide simulated objects from all but the component that has created them, to avoid confusing and in the worst case crashing the other components that have been written without anticipating the presence of "foreign" simulation objects.

For example, imagine a game that is written with the convention that all `PxActors` store an index into an array in their `userData` fields at creation time, that is then used to look up the game entity object that is among other things responsible for rendering the physics actor. This game could have code that queries the `PxScene` for all the actors contained, and then casts the `userData` of each retrieved actor into an index, which it uses to do an array lookup. Further, imagine that the author of the game decides to make use of a third party library which provides some physical special effect. This library also creates `PxActors` in the same scene as the game, so that they will automatically interact with the game's actors. Unfortunately the actors belonging to the library do not use the convention of the game when it comes to the `userData` field -- in fact the library uses the `userData` field to some other number. One can easily imagine that unless the game code is changed, this could lead to the game trying to index into its entity array using `userData` values that it did not create, resulting in undefined behavior. The solution in this case would be to let the game mark its own actors in a distinct way from the actors of the library so that the foreign actors could be efficiently skipped when performing the `userData` lookups. Clearly one needs to guard against this problem any time the application receives actors back from the SDK, including for example notifications and scene queries. In theory the game could achieve this without dedicated PhysX support by doing a hash table lookup of any `PxActor` pointers it receives back from the SDK, but this is less efficient than putting a little bit of logic into the PhysX

SDK.

Each middleware library or application component that uses the PhysX SDK for simulation is referred to as a 'client'. Each client is identified with a PxClientID, which is an 8 bit scalar value. The PhysX SDK supports up to PX_MAX_CLIENTS clients, which is currently defined to be 128, though we only foresee needing a few clients at most. The main client in a multi-client context, or the sole client in an application that was not written to be multi-client aware, is the default client with a PxClientID of PX_DEFAULT_CLIENT.

Additional clients can be created by calling:

```
PxClientID myClient = PxScene::createClient();
```

This generates a new client identifier. In the above example, the game would implicitly be the default client, and the third party library would generate an additional clientID for itself like this. If multiple scenes exist, this client creation and all additional book keeping described below must be performed independently for each scene.

Any actor that is created can now be assigned to a client right after creation using:

```
PxActor::setOwnerClient(PxClientID ownerClient);
```

If this is not done, the PX_DEFAULT_CLIENT is the owner by default. Thus, in our example above, the third party library would need to mark all of the actors it creates as part of its special effect in this way. Actors created by the game are thus marked correctly by default, no change to the game code is needed.

Many object retrieval methods of the PhysX SDK employ a clientID parameter so that the actors retrieved are limited to the ones belonging to the PX_DEFAULT_CLIENT by default, so that foreign actors are automatically omitted. For example, if the game had used the efficient

active transforms method to retrieve updated actors for rendering, it would need no further changes. The method:

```
virtual PxActiveTransform* PxScene::getActiveTransforms(Px
```

limits the returned actor scope to the default client unless another client is specified explicitly. If on the other hand the game used the simple method `PxScene::getActors()`, a simple change is needed because `getActors` does not filter its results by client:

```
void retrieveGameActors()
{
    PxActorTypeSelectionFlags desiredTypes = PxActorTypeSel
        | PxActorTypeSelectionFlag::eRIGID_DYNAMIC;
    PxU32 count = sharedScene->getNbActors(desiredTypes);
    PxActor** buffer = new PxActor*[count];
    sharedScene->getActors(desiredTypes, buffer, count);
    for(PxU32 i = 0; i < count; i++)
    {
        if (buffer[i]->getOwnerClient() == PX_DEFAULT_C
        {
            // further process this actor

            ...
        }
    }
    delete buffer;
}
```

A further key method by which components using the PhysX SDK receive objects back is the use of callbacks. The main callback class of the SDK is `PxSimulationEventCallback`. A derived class can be implemented by each client and passed to the SDK using:

```
PxScene::setSimulationEventCallback(PxSimulationEventCallback*
```

This is in contrast to specifying the callback using:

```
PxSimulationEventCallback* PxSceneDesc::simulationEventCal
```

which does not permit the flexibility to specify a client explicitly and is thus equivalent to calling `setSimulationEventCallback()` with the default client parameter. In our example, the game would set the callback for its own event listener class using either method, and the third party library could set an additional listener with its own client ID as the parameter. Such a setup would guarantee that each component only receives events related to its own actors.

In more complex scenarios, the need may arise to loosen this strict segregation of events. It is plausible for example to imagine that the third party middleware library needs to receive notification that its special effect objects got in contact with the game's collision environment because in this case it wants to spawn some additional graphical or audible contact effect. To do this, the library must ask the SDK to send foreign contact events to its own event callback:

```
sharedScene->setClientBehaviorBits(libraryClient, PxClientBehav
```

In addition, the game must opt in all the actors that could be sent to the `libraryClient`:

```
gameActor->setClientBehaviorBits(PxActorClientBehaviorBit::eREP
```

In both function calls, flags for multiple event types can of course be OR-ed together. Besides contacts, flags are available for triggers, scene queries and constraint break events. Scene queries deserve special attention because they don't reach the user through the `PxSimulationEventCallback`. Instead, the owner of a synchronous scene query is defined by a parameter, for example as in `PxScene::raycastAny()`. For batched queries the owner is specified in the batch query creation descriptor: `PxBatchQueryDesc::ownerClient`.

Shapes

Shapes describe the spatial extent and collision properties of actors. A `PxShape` is owned by the `PxActor` through which it was created, and is released along with the actor. Each shape contains a `PxGeometry` object and a reference to a `PxMaterial`, which must both be specified upon creation. A shape may also have a transform relative to its parent actor. To create an actor with several shapes (sometimes referred to as a *compound*), just call `PxRigidActor::createShape()` several times. There are no persistence requirements on the `PxGeometry` instance that is passed as a function argument in `PxRigidActor::createShape()`. The PhysX SDK takes a copy of the `PxGeometry` object rather than maintaining a reference to it.

There are some restrictions on the geometry types that may be specified for a shape, depending on the type of the parent actor. Actors of type `PxRigidStatic` may have any kind of supported shape; that is, sphere, capsule, box, convex mesh, triangle mesh, plane or heightfield. The shapes permitted for actors of type `PxRigidDynamic` depends on whether or not the actor is kinematic. `PxRigidDynamic` actors that have been set up as kinematic may be given any of the supported shapes, just as with `PxRigidStatic` actors. Dynamic actors, on the other hand, will not accept heightfields or triangle meshes or planes. Actors of type `PxArticulationLink` have the same restrictions as non-kinematic `PxRigidDynamic` instances in that they are also forbidden to accept heightfields and triangle meshes and planes. To complete the discussion it is worth noting that actors of type `PxCloth`, `PxParticleFluid` and `PxParticleSystem` do not accept shapes at all because their collision is handled by systems special to these actor types.

The PhysX SDK supports contacts between all possible combinations of shape pair except for combinations where both shapes are a plane or a heightfield or a triangle mesh. The collision of two meshes, for example, is not supported. Similarly, collision between a plane and a heightfield or between two heightfields remains unsupported by the sdk.

Simulation Shapes and Scene Query Shapes

PxShape instances are configured by default to participate in the intersection tests that determine the contacting features of shape pairs. They are also configured by default to participate in scene query tests. It is possible to configure PxShape instances to participate or not in both types of test. This can be done before or after the shape's actor has been added to the scene.

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

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

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

```
void enableShapeInShapePairIntersectionTests(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);
}
```


Trigger Shapes

PxShape instances can be configured as trigger shapes. Trigger shapes play no part in the simulation of the scene (though they can be configured to participate in scene queries). Instead, their role is to report that there has been an overlap with another shape. This does not involve generating the contact features of the intersection. As a result, contact reports are not available for trigger shapes. A further point to note is that because trigger points play no part in the simulation it makes no sense for the `eSIMULATION_SHAPE` and `eTRIGGER_SHAPE` flags to be simultaneously raised. To avoid any ambiguity the sdk will act to prevent these flags being simultaneously raised; that is, if the `eSIMULATION_SHAPE(eTRIGGER_SHAPE)` flag is raised then attempts to raise the `eTRIGGER_SHAPE(eSIMULATION_SHAPE)` flag will be rejected. When this occurs an error is passed to the error stream.

Trigger shapes have been used in `SampleSubmarine` to determine if the submarine has reached the treasure. In the following code the `PxActor` representing the treasure has its 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 `SampleSubmarine` through the implementation of `PxSimulationEventCallback::onTrigger` in the `PxSampleSubmarine` class, a sub-class of `PxSimulationEventCallback`:

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

```
        continue;

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

The code above iterates through all pairs of overlapping shapes that involve a trigger shape. If it is found that the treasure has been touched by the submarine then the flag gTreasureFound is set true.

Kinematic triangle meshes (planes, heighfields)

It is possible to create a kinematic `PxRigidDynamic` which can have a triangle mesh (plane, heighfield) shape. If this shape has a simulation shape flag, this actor must stay kinematic. If you change the flag to not simulated, you can switch even the kinematic flag.

To setup kinematic triangle mesh see following code:

```
PxRigidDynamic* meshActor = getPhysics().createRigidDynamic(PxT
PxShape* meshShape;
if(meshActor)
{
    meshActor->setRigidDynamicFlag(PxRigidDynamicFlag::eKIN

    PxTriangleMeshGeometry triGeom;
    triGeom.triangleMesh = triangleMesh;
    meshShape = meshActor->createShape(triGeom, defaultMate
    getScene().addActor(*meshActor);
}

```

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

```
PxRigidDynamic* meshActor = getPhysics().createRigidDynamic(PxT
PxShape* meshShape;
if(meshActor)
{
    meshActor->setRigidDynamicFlag(PxRigidDynamicFlag::eKIN

    PxTriangleMeshGeometry triGeom;
    triGeom.triangleMesh = triangleMesh;
    meshShape = meshActor->createShape(triGeom, defaultMate
    getScene().addActor(*meshActor);

    PxConvexMeshGeometry convexGeom = PxConvexMeshGeometry(
    convexShape = meshActor->createShape(convexGeom, default
    convexShape->setFlag(PxShapeFlag::eSIMULATION_SHAPE, fa
}

// ... now switch to dynamic

```

```
meshShape->setFlag(PxShapeFlag::eSIMULATION_SHAPE, false);  
convexShape->setFlag(PxShapeFlag::eSIMULATION_SHAPE, true);  
meshActor->setRigidDynamicFlag(PxRigidDynamicFlag::eKINEMATIC,
```



Geometries

The `PxGeometry` class defines a volume or surface with a fixed position and orientation. Typically, as when used in a shape or scene query, a transform specifies the frame in which the geometry is interpreted.

For bulk objects, such as a convex mesh, triangle mesh or height field, PhysX allows multiple `PxGeometry` objects to refer to a single mesh or height field, and supports per-instance scaling.

Note: Each mesh (or height field) is reference counted, and the reference count refers to the number of `PxShapes` whose geometries reference the mesh, rather than the number of `PxGeometry` objects.

Spheres

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

Capsules

A `PxCapsuleGeometry` is centered at the origin. It is specified by a radius and a half-height 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, the shape needs a relative transform that rotates it around the Z-axis by a quarter-circle. By doing this, the capsule will extend along the Y-axis of the actor instead of the X-axis. Setting up the shape and actor is otherwise the same as for the sphere:

```
PxRigidDynamic* aCapsuleActor = thePhysics->createRigidDynamic(
PxTransform relativePose(PxQuat(PxHalfPi, PxVec(0,0,1))));
PxShape* aCapsuleShape = aCapsuleActor->createShape(PxCapsuleGe
```

```
PxRigidBodyExt::updateMassAndInertia(*aCapsuleActor, capsuleDen  
aScene->addActor(aCapsuleActor);
```

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

Boxes

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

```
PxShape* aBoxShape = aBoxActor->createShape(PxBoxGeometry(a/2,
```

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

Planes

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

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

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

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

Convex Meshes

Creating a `PxConvexMesh` requires cooking. It is assumed here that the cooking library has already been initialized (see [Startup and Shutdown](#).)

The following steps explain how to create a simple square pyramid.

First, define the vertices of the convex object:

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

Then construct a description of the convex data layout:

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

Now use the cooking library to construct a PxConvexMesh:

```
PxToolkit::MemoryOutputStream buf;  
if(!cooking.cookConvexMesh(convexDesc, buf))  
    return NULL;  
PxToolkit::MemoryInputData input(buf.getData(), buf.getSize());  
PxConvexMesh* convexMesh = thePhysics->createConvexMesh(input);
```

Finally, create a shape using a PxConvexMeshGeometry which instances the mesh:

```
PxShape* aConvexShape = aConvexActor->createShape(PxConvexMeshG
```

A user can optionally provide a per-instance PxMeshScale in the PxConvexMeshGeometry. The default scale is the identity.

The default convex hull generation code is selected when using the PxConvexFlag::eCOMPUTE_CONVEX flag alone. The algorithm tries to create a convex hull as close to the source vertices as possible. This can sometimes fail when the source data is geometrically challenging, for example if it contains a lot of vertices close to each-other, etc. An error is reported to the error stream in case of failure. If this happens, the best option is to switch to an alternative hull generation routine using the

PxConvexFlag::eCOMPUTE_CONVEX|PxConvexFlag::eINFLATE_CONVEX flags, both together. This allows the code to inflate the source data by a margin - defined by PxCookingParams::skinWidth -, which gives the code more freedom to correct the problematic geometry. Alternatively it is possible for users to provide an already created hull, by filling up both PxConvexMeshDesc::points and PxConvexMeshDesc::triangles, and omitting the PxConvexFlag::eCOMPUTE_CONVEX flag. Some checks are still performed to make sure the provided hull is valid, so the cooking call can still fail at that point.

In any case the number of vertices and the number of convex polygons in the final cooked hull are both limited to 256.

Height Fields

As the name suggests, terrains can be described by just the height values on a regular, rectangular sampling grid:

```
PxHeightFieldSample* samples = (PxHeightFieldSample*)alloc(size)
```

Each sample consists of a 16 bit integer height value, two materials (for the two triangles in the samples rectangle) and a tessellation flag. The flag and materials refer to the cell below and to the right of the sample point, and indicate along which diagonal to split it into triangles, and the materials of those triangles. A special predefined material `PxHeightFieldMaterial::eHOLE` specifies a hole in the height field. See the reference documentation for `PxHeightFieldSample` for more details.

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

```
PxHeightFieldDesc hfDesc;  
hfDesc.format      = PxHeightFieldFormat::eS16_TM;  
hfDesc.nbColumns   = numCols;  
hfDesc.nbRows      = numRows;  
hfDesc.samples.data = samples;
```

```
hfDesc.samples.stride      = sizeof(PxHeightFieldSample);  
PxHeightField* aHeightField = thePhysics->createHeightField(hfD
```

Now create a PxHeightFieldGeometry and a shape:

```
PxHeightFieldGeometry hfGeom(aHeightField, PxMeshGeometryFlags(  
PxShape* aHeightFieldShape = aHeightFieldActor->createShape(hfG
```

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

The variant of createShape() used here specifies an array of materials for the height field, which will be indexed by the material indices of each cell to resolve collisions with that cell. The single-material variant of createShape() may be used instead, but the height field material indices must all be a single value or the special value eHOLE.

Triangle Meshes

Creating a PxTriangleMesh requires cooking. It is assumed here that the cooking library has already been initialized (see [Startup and Shutdown](#).)

Like graphical triangle meshes, a collision triangle mesh consists of a collection of vertices and the triangle indices. Triangle mesh creation requires use of the cooking library:

```
PxTriangleMeshDesc meshDesc;  
meshDesc.points.count      = nbVerts;  
meshDesc.points.stride    = sizeof(PxVec3);  
meshDesc.points.data      = verts;  
  
meshDesc.triangles.count   = triCount;  
meshDesc.triangles.stride  = 3*sizeof(PxU32);  
meshDesc.triangles.data    = indices32;
```

```
PxToolkit::MemoryOutputStream writeBuffer;  
bool status = cooking.cookTriangleMesh(meshDesc, writeBuffer);  
if(!status)  
    return NULL;  
  
PxToolkit::MemoryInputData readBuffer(writeBuffer.getData(), wr  
return physics.createTriangleMesh(readBuffer);
```

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

Shapes with triangle mesh geometries may only be created for static and kinematic actors:

```
PxShape* aTriMeshShape = aTriMeshActor->createShape(PxTriangleM
```

The user can optionally specify a per-instance `PxMeshScale` in the `PxTriangleMeshGeometry`. The default scale is the identity.

Like height fields, triangle meshes support per-triangle material indices. To use per-triangle materials for a mesh, provide the indices to the cooking library in the mesh descriptor, and use the multi-material form of `createShape()`.

Mesh Scaling

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

The following code creates a shape with a `PxTriangleMesh` scaled by a factor of x along 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::createIdentity());
PxTriangleMeshGeometry geom(myTriMesh, scale);
PxShape* myTriMeshShape = myActor->createShape(geom, *myMaterial
```

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

```
PxMeshScale scale(PxVec3(x,y,z), PxQuat quat(PxPi*0.25f, PxVec3
PxConvexMeshGeometry geom(myTriMesh, scale);
PxShape* myConvexMeshShape = myActor->createShape(geom, *myMater
```

Height fields can also be scaled, using scale factors stored in `PxHeightFieldGeometry`. In this case the scale is assumed to be along the

axes of the rows, columns and height directions of the height field. The scaling of is demonstrated in SampleNorthPole in SampleNorthPoleBuilder.cpp:

```
PxHeightFieldGeometry hfGeom(heightField, PxMeshGeometryFlags()  
PxShape* hfShape = hfActor->createShape(hfGeom, getDefaultMater
```

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

PxGeometryHolder

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

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

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

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

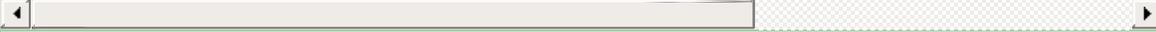
`PxGeometryHolder` is a union-like class that allows the return of a `PxGeometry` object by value, regardless of type. Its use is illustrated in the `createRenderObjectFromShape()` function in `PhysXSample.cpp`:

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

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

```
bool testForOverlap(const PxShape& s0, const PxShape& s1)
{
    return PxGeometryQuery::overlap(s0.getGeometry().any(), PxS
                                   s1.getGeometry().any(), PxS
}

```



Detaching Shapes

In the North Pole Sample, some of the shapes detach on contact with a snowball. To request notification of this event, the sample sets a flag in the simulation filter function:

```
if (needsContactReport(filterData0, filterData1))
{
    pairFlags |= PxPairFlag::eNOTIFY_TOUCH_FOUND;
}
```

When this flag is set for a pair, the initial collision of that pair will generate a callback through `PxSimulationEventCallback::onContact()`. The implementation of this callback in the sample simply records which detachable shapes were touched during simulation. `needsContactReport()` is a helper function which returns true if one of the shapes is detachable and the other is marked as a snowball. It determines this by testing the flags in each shape's `simulationFilterData`, which were set on shape creation using the `setDetachable()` and `setSnowball()` functions. Collision filtering is discussed in more detail in the [Callbacks and Customization](#) section.

After simulation, the sample iterates over the list of touched detachable shapes and detaches each one from its owning actor. Since a `PxShape` must belong to a `PxActor`, the sample creates a new actor, whose global pose is that of the original shape. The geometry and material for the new shape are retrieved from the original shape:

```
PxRigidDynamic* newActor = mPhysics->createRigidDynamic(pose);

PxMaterial* mat;
shape->getMaterials(&mat, 1);
PxGeometryHolder geometry = shape->getGeometry();
PxTransform newActorPose = PxShapeExt::getGlobalPose(*shape);

PxRigidDynamic* newActor = PxCreateDynamic(*mPhysics, newActorP
shape->release());
```

Shape Vertex and Triangle Queries

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

```
RenderBaseActor* PhysXSample::createRenderObjectFromShape(PxSha
```

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

It is possible to get information about triangle from a triangle mesh using the getTriangle function, You may also get the information about adjacency triangles for the triangle, for this you need to cook your triangle meshes with cooking parameter buildTriangleAdjacencies, otherwise the adjacency information is not created and stored.

Convex Meshes

A convex mesh contains an array of vertices, an array of faces, and an index buffer which concatenates the vertex indices for each face. To unpack a convex mesh, the first step is to extract the shared convex mesh:

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

Then obtain references to the vertex and index buffers:

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

Now iterate over the array of faces to triangulate them:

```
PxU32 offset = 0;  
for(PxU32 i=0;i<nbPolygons;i++)
```

```

{
    PxHullPolygon face;
    bool status = convexMesh->getPolygonData(i, face);
    PX_ASSERT(status);

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

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

    offset += face.mNbVerts;
}

```

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

Triangle Meshes

Triangle meshes contain arrays of vertices and index triplets which define the triangles by indexing into the vertex buffer. The arrays can be accessed directly from the shared triangle mesh:

```

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

```

The indices may be stored with either 16-bit or 32-bit values, specified when the mesh was originally cooked. To determine the storage format at

runtime, use the API call:

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

Assuming that the triangle indices are stored in 16-bit format, find the *j*th vertex of the *i*th 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 direct access to the height field samples is not provided. Instead, calls are provided to render the samples to a user-supplied buffer.

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

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

The height field has three scaling parameters:

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

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

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

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

```
const PxU32 nbVerts = nbRows * nbCols;
PxHeightFieldSample* sampleBuffer = new PxHeightFieldSample[nbV
hf->saveCells(sampleBuffer, nbVerts * sizeof(PxHeightFieldSampl
```

The samples are stored in row-major order; that is, row0 is stored first, followed by row1, then row2, and so on. Thus the sample corresponding to the i th row and the j th column is $i \cdot \text{nbCols} + j$.

Evaluate the scaled vertices of the height field as follows:

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

Then tessellate the field from the samples as required.

Joint Basics

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

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

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

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

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

Each *localFrame* argument specifies a constraint frame relative to the actor's global pose. Each joint defines a relationship between the global positions and origins of the constraint frames that will be enforced by the PhysX constraint solver. In this example, the revolute joint constrains the origin points of the two frames to be coincident and their x-axes to coincide, but allows the two actors to rotate freely relative to one another around this common axis.

PhysX supports six different joint types:

- a **fixed** joint locks the orientations and origins rigidly together
- a **distance** joint keeps the origins within a certain distance range
- a **spherical** joint (also called a *ball-and-socket*) keeps the origins together, but allows the orientations to vary freely.
- a **revolute** joint (also called a *hinge*) keeps the origins and x-axes of the frames together, and allows free rotation around this common axis.
- a **prismatic** joint (also called a *slider*) keeps the orientations identical, but allows the origin of each frame to slide freely along the common x-axis.
- a **D6** joint is a highly configurable joint that allows specification of individual degrees of freedom either to move freely or be locked together. It can be used to implement a wide variety of mechanical and anatomical joints, but is somewhat less intuitive to configure than the other joint types. This joint is covered in detail below.

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

Note: *As in the rest of the PhysX API, all joint angles for limits and drive targets are specified in radians*

Visualization

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

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

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

Force Reporting

The joint may be configured to report each frame the force that was required to hold it together. To enable this behavior, set the joint's reporting flag. The force may then be retrieved after simulation with a call to `getForce()`:

```
joint->setConstraintFlag(PxConstraintFlag::eREPORTING)
...

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 breaking force and torque may be specified, and if the force or torque required to maintain the joint constraint exceeds this threshold, the joint will break. Breaking a joint generates a simulation event (see `PxSimulationEventCallback::onJointBreak`), and the joint no longer partakes in simulation, although it remains attached to its actors until it is deleted.

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

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

A constraint flag records whether a joint is currently broken:

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

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

```
class MySimulationEventCallback
{
    void onConstraintBreak(PxConstraintInfo* constraints, PxU32 count)
    {
        for(PxU32 i=0; i<count; i++)
        {
            PxJoint* joint = reinterpret_cast<PxJoint*>(constraints[i].externalReference);
            ...
        }
    }
}
```

Projection

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

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

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

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

Limits

Some PhysX joints constrain not just relative rotation or translation, but can also enforce *limits* on the range of that motion. For example, in its initial configuration the revolute joint allows free rotation around its axis, but by specifying and enabling a limit, lower and upper bounds may be placed upon the angle of rotation.

Limits are a form of collision, and like collision of rigid body shapes, stable limit behavior requires a *contactDistance* tolerance specifying how close to the limit the joint configuration may be before the solver tries to enforce it. A high tolerance makes the limit less likely to be violated even at high relative velocity, but because the limit is active more of the time, the joint is more expensive to simulate.

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

```
revolute->setLimit(PxJointLimitPair(-PxPi/4, PxPi/4, 0.1f)); //  
revolute->setRevoluteJointFlag(PxRevoluteJointFlag::eLIMIT_ENAB
```

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

```
PxJointLimitPair limitPair(-PxPi/4, PxPi/4, 0.1f));
```

```
limitPair.spring = 100.0f;  
limitPair.damping = 20.0f;  
revolute->setRevoluteJointLimit(limitPair);  
revolute->setRevoluteJointFlag(PxRevoluteJointFlag::eLIMIT_ENAB
```

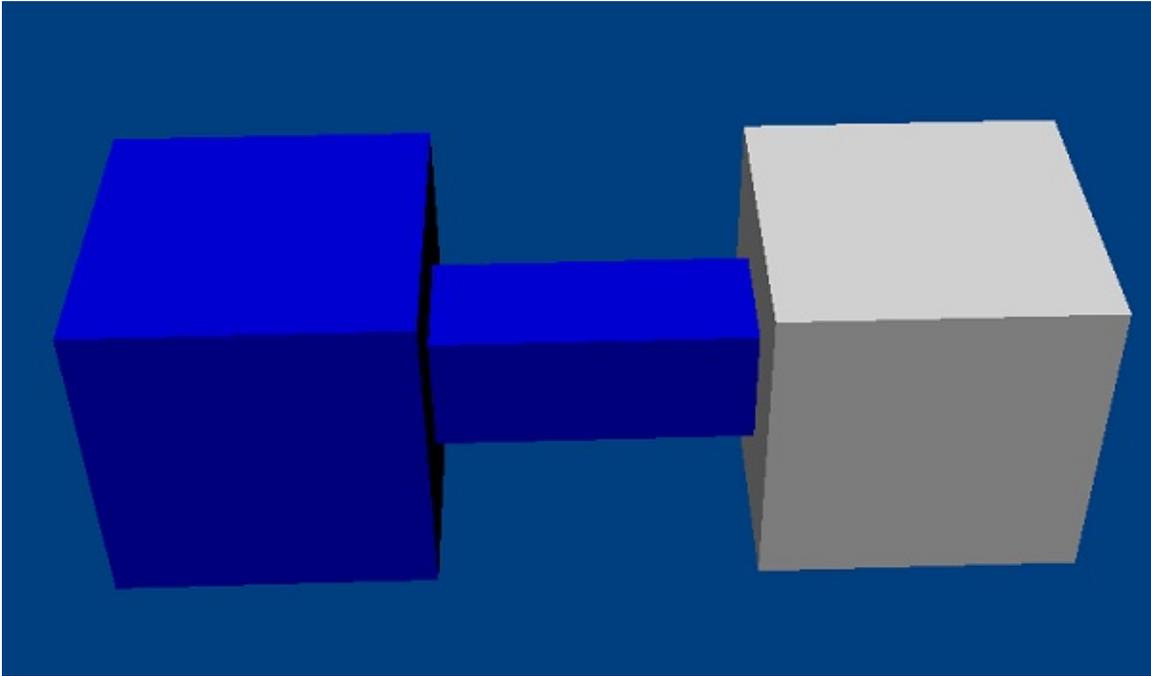
Note: *Limits are not projected.*

Actuation

Some PhysX joints may be actuated by a motor or a spring implicitly integrated by the PhysX solver. While driving simulations with actuated joints is more expensive than simply applying forces, it can provide much more stable control of simulation. See the specific documentation on the D6 and revolute joints for details

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

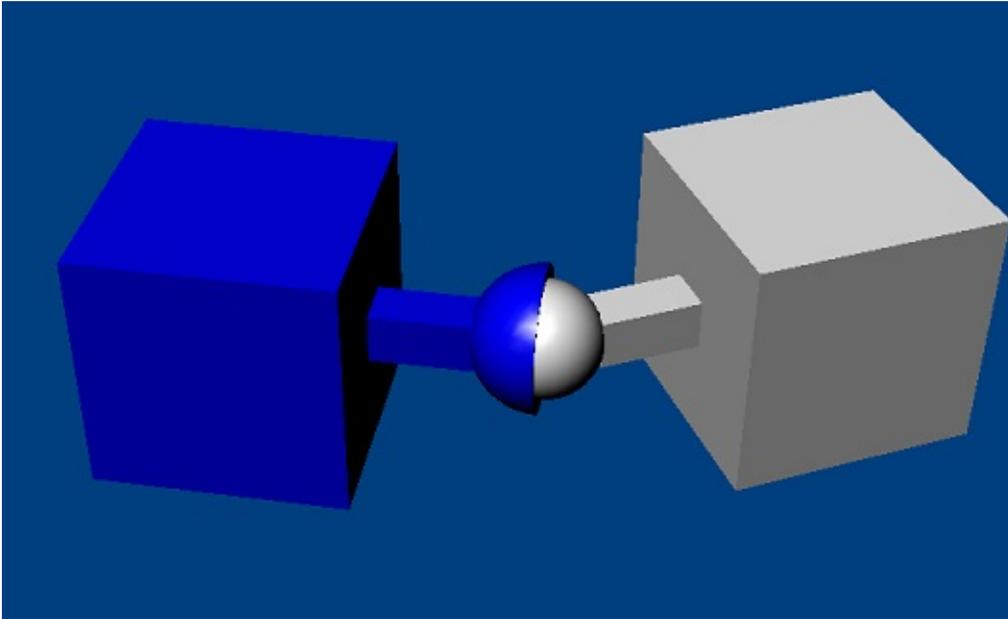
Fixed Joint



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

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

Spherical Joint



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

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

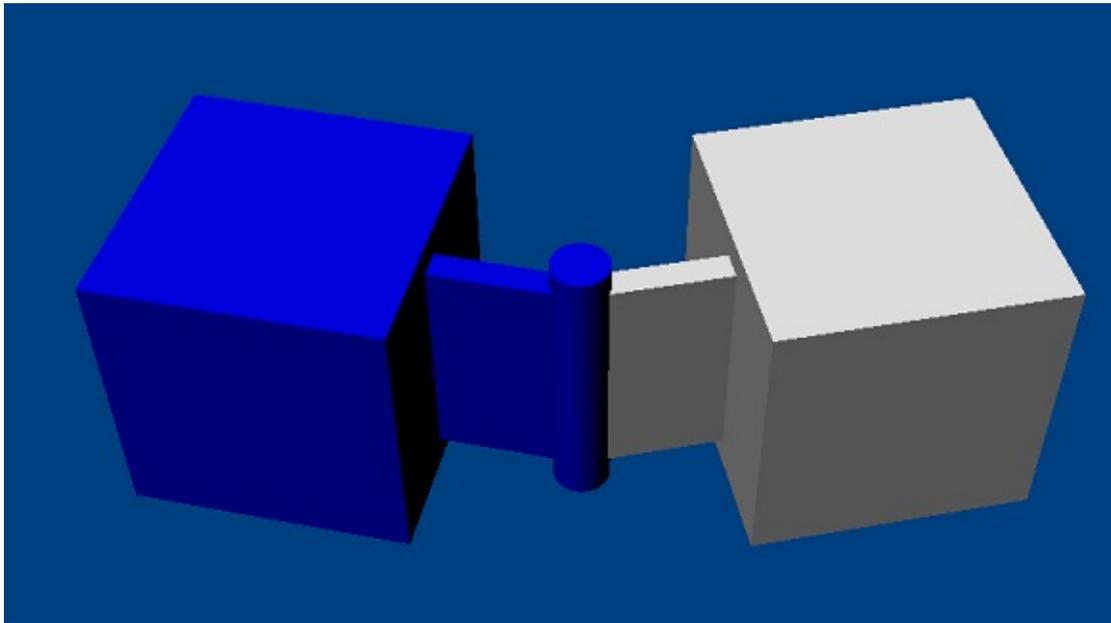
```
joint->setLimitCone(PxJointLimitCone(PxPi/2, PxPi/6, 0.01f);  
joint->setSphericalJointFlag(PxSphericalJointFlag::eLIMIT_ENABL
```

Note that very small or highly elliptical limit cones may result in solver jitter.

Note: *Visualization of the limit surface can help considerably in*

understanding its shape.

Revolute Joint



A revolute joint removes all but a single rotational degree of freedom from two objects. The axis along which the two bodies may rotate is specified by the common origin of the joint frames and their common x-axis. In theory, all origin points along the axis of rotation are equivalent, but simulation stability is best in practice when the point is near where the bodies are closest.

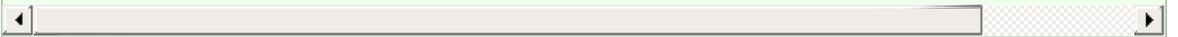
The joint supports a rotational limit with upper and lower extents. The angle is zero where the y- and z- axes of the joint frames are coincident, and increases moving from the y-axis towards the z-axis:

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

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

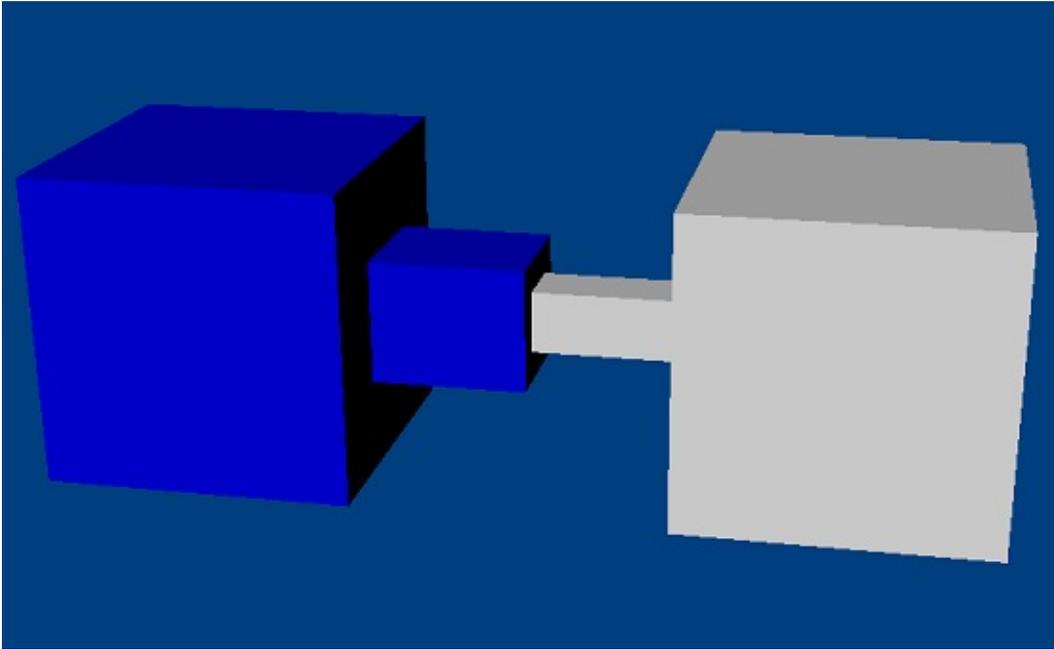
```
joint->setDriveVelocity(10.0f);
```

```
joint->setRevoluteJointFlag(PxRevoluteJointFlag::eDRIVE_ENABLED
```



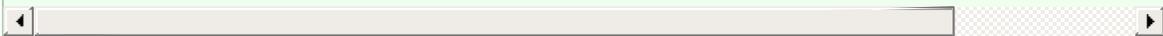
By default, when the angular velocity at the joint exceeds the target velocity the motor acts as a brake; a freespin flag disables this braking behavior.

Prismatic Joint

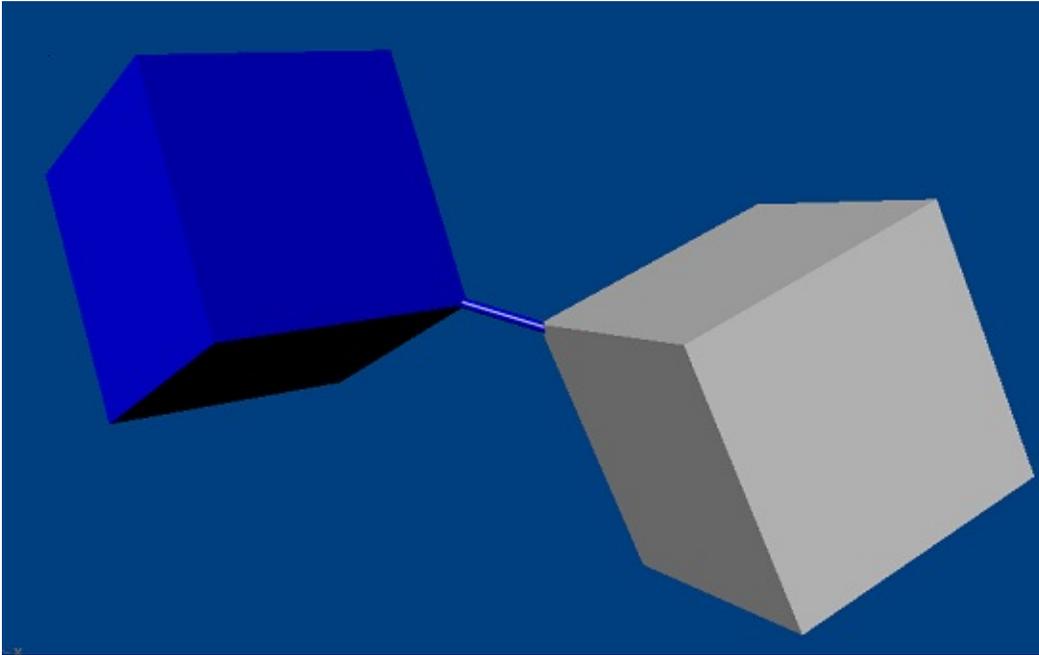


A prismatic joint prevents all rotational motion, but allows the origin of actor1's constraint frame to move freely along the x-axis of actor0's constraint frame. The prismatic joint supports a single limit with upper and lower bounds on the distance between the two constraint frames' origin points:

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



Distance Joint



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

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

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

D6 Joint

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

Locking and Unlocking Axes

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

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

Unlocking translational degrees of freedom allows the origin point of actor1's constraint frame to move along a subset of the axes defined by actor0's constraint frame. For example, unlocking just the X-axis creates the equivalent of a prismatic joint.

Rotational degrees of freedom are partitioned as *twist* (around the X-axis of actor0's constraint frame) and *swing* (around the Y- and Z- axes.) Different effects are achieved by unlocking various combinations of twist and swing.

- if just a single degree of angular freedom is unlocked, the result is always equivalent to a revolute joint. It is recommended that if just one angular freedom is unlocked, it should be the twist degree, because the joint has various configuration options and optimizations that are designed for this case.
- if both swing degrees of freedom are unlocked but the twist degree remains locked, the result is a *zero-twist* joint. The x-axis of actor1 swings freely away from the x-axis of actor0 but twists to minimize the rotation required to align the two frames. This creates a kind of isotropic universal joint which avoids the problems of the usual

'engineering style' universal joint (see below) that is sometimes used as a kind of twist constraint. There is a nasty singularity at π radians (180 degrees) swing, so a swing limit should be used to avoid the singularity.

- if one swing and one twist degree of freedom are unlocked but the remaining swing is kept locked, a *zero-swing* joint results (often also called a *universal* joint.) If for example the SWING1 (y-axis rotation) is unlocked, the x-axis of actor1 is constrained to remain orthogonal to the z-axis of actor0. In character applications, this joint can be used to model an elbow swing joint incorporating the twist freedom of the lower arm or a knee swing joint incorporating the twist freedom of the lower leg. In vehicle applications, these joints can be used as 'steered wheel' joints in which the child actor is the wheel, free to rotate about its twist axis, while the free swing axis in the parent acts as the steering axis. Care must be taken with this combination because of anisotropic behavior and singularities (beware the dreaded gimbal lock) at angles of $\pi/2$ radians (90 degrees), making the zero-twist joint a better behaved alternative for most use cases.
- if all three angular degrees are unlocked, the result is equivalent to a spherical joint.

Three of the joints from PhysX 2 that have been removed from PhysX 3 can be implemented as follows:

- The cylindrical joint (with axis along the common x-axis of the two constraint frames) 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's constraint frame) is given by the combination:

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

```
d6joint->setMotion(PxD6Axis::eZ, PxD6Motion::eFREE);  
d6joint->setMotion(PxD6Axis::eTWIST, PxD6Motion::eFREE);  
d6joint->setMotion(PxD6Axis::eSWING1, PxD6Motion::eFREE);  
d6joint->setMotion(PxD6Axis::eSWING2, PxD6Motion::eFREE);
```

- the point-on-line joint (with axis along the x-axis of actor0's constraint frame) is given by the combination:

```
d6joint->setMotion(PxD6Axis::eX, PxD6Motion::eFREE);  
d6joint->setMotion(PxD6Axis::eTWIST, PxD6Motion::eFREE);  
d6joint->setMotion(PxD6Axis::eSWING1, PxD6Motion::eFREE);  
d6joint->setMotion(PxD6Axis::eSWING2, PxD6Motion::eFREE);
```

Limits

Instead of specifying that an axis is free or locked, it may also be specified as limited. The 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 of the translational degrees of freedom. The limit constrains the distance between the origins of the constraint 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(PxJointLimit(1.0f, 0.1f));
```

constrains the y- and z- coordinates of actor1's constraint frame to lie within the unit disc. Since the x-axis is unconstrained, the effect is to constrain the origin of actor1's constraint frame to lie within a cylinder of radius 1 extending along the x-axis of actor0's constraint frame.

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

If both swing degrees of freedom are limited, a limit cone is generated,

identical to the limit of the spherical joint. As with the spherical joint, very small or highly elliptical limit cones may result in solver jitter.

If only one swing degree of freedom is limited, the corresponding angle from the cone limit is used to limit rotation. If the other swing degree is locked, the maximum value of the limit is π radians (180 degrees). If the other swing degree is free, the maximum value of the limit is $\pi/2$ radians (90 degrees.)

Drives

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

$$\text{force} = \text{spring} * (\text{targetPosition} - \text{position}) + \text{damping} * (\text{targetVelocity} - \text{velocity})$$

The drive model may also be configured to generate a proportional acceleration instead of a force, factoring in the masses of the actors to which the joint is attached. Acceleration drive is often easier to tune than force drive.

The linear drive model for the D6 has the following parameters:

- target position, specified in actor0's constraint frame
- target velocity, specified in actor0's constraint frame
- spring
- damping
- forceLimit - the maximum force the drive can apply
- acceleration drive flag

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

reduced by stiffening the spring or supplying a velocity target.

With a fixed position input and a zero target velocity, a position drive will spring about that 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:

PxDrive 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 angu

d6joint->setDrivePosition(PxTransform::createIdentity());
d6joint->setDriveVelocity(PxVec3::createZero());
```

Angular drive differs from linear drive in a fundamental way: it does not have a simple and intuitive representation free from singularities. For this reason, the D6 joint provides two angular drive models - twist and swing and SLERP (Spherical Linear Interpolation).

The two models differ in the way they estimate the path in quaternion space between the current orientation and the target orientation. In a SLERP drive, the quaternion is used directly. In a twist and swing drive, it is decomposed into separate twist and swing components and each component is interpolated separately. Twist and swing is intuitive in many situations; however, there is a singularity when driven to 180 degrees swing. In addition, the drive will not follow the shortest arc between two orientations. On the other hand, SLERP drive will follow the shortest arc between a pair of angular configurations, but may cause unintuitive changes in the joint's twist and swing.

The angular drive model has the following parameters:

- An angular velocity target specified relative to actor0's constraint frame
- An orientation target specified relative to actor0's constraint frame
- drive specifications for SLERP (slerpDrive), swing (swingDrive) and twist (twistDrive):
- spring - amount of torque needed to move the joint to its target orientation proportional to the angle from the target (not used for a velocity drive).
- damping - applied to the drive spring (used to smooth out oscillations about the drive target).
- forceLimit - maximum torque applied when driving towards a velocity target (not used for an orientation drive)
- acceleration drive flag. If this flag is set the acceleration (rather than the force) applied by the drive is proportional to the angle from the target.

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

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

Configuring Joints for Best Behavior

The behavior quality of joints in PhysX is largely determined by the ability of the iterative solver to converge. Better convergence can be achieved simply by increasing the attributes of the PxRigidBody which controls the solver iteration count. However, joints can also be configured to produce better convergence.

- the solver can have difficulty converging well when where a light object is constrained between two heavy objects. Mass ratios of higher than 10 are best avoided in such scenarios.
- when one body is significantly heavier than the other, make the lighter body the second actor in the joint. Similarly, when one of the objects is static or kinematic (or the actor pointer is NULL) make the dynamic body the second actor.

A common use for joints is to move objects around in the world. Best results are obtained when the solver has access to the velocity of motion as well as the change in position.

- if you want a very stiff controller that moves the object to specific position each frame, consider jointing the object to a kinematic actor and use the `setKinematicTarget` function to move the actor.
- if you want a more springy controller, use a D6 joint with a drive target to set the desired position and orientation, and control the spring parameters to increase stiffness and damping. In general, acceleration drive is much easier to tune than force drive.

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

Applying Forces and Torques

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

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

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

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

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

```
void PxRigidBody::addForce(const PxVec3& force, PxForceMode::En
void PxRigidBody::addTorque(const PxVec3& torque, PxForceMode::

void PxRigidBodyExt::addForceAtPos(PxRigidBody& body, const PxV
void PxRigidBodyExt::addForceAtLocalPos(PxRigidBody& body, cons
void PxRigidBodyExt::addLocalForceAtPos(PxRigidBody& body, cons
void PxRigidBodyExt::addLocalForceAtLocalPos(PxRigidBody& body,
```

The `PxForceMode` member defaults to `PxForceMode::eFORCE` to apply simple forces. There are other possibilities. For example

PxForceMode::eIMPULSE will apply an impulsive force. PxForceMode::eVELOCITY_CHANGE will do the same, but also ignore the mass of the body, effectively leading to an instantaneous velocity change. See the API documentation of PxForceMode for the other possibilities.

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

Gravity

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

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

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

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

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

Setting the Velocity

To immediately get a body moving in a certain direction, set its velocity:

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

During simulation, PhysX will modify the velocity of an object in accordance with gravity and other applied forces, and override the velocity if it is in conflict with a joint or a collision constraint. For example, if a ball is resting on a table and has a downward velocity, this will be clamped to 0. If you set the velocity of a chain link that is jointed to a bunch of other chain links, that velocity will be diminished, and the other chain links' velocity will be increased, to permit the chain to stay together.

Kinematic Actors

Sometimes controlling an actor using forces or constraints is not sufficiently robust, precise or flexible. For example moving platforms or character controllers often need to manipulate an actor's position or have it exactly follow a specific path. Such a control scheme is provided by kinematic actors.

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

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

```
PxRigidDynamic::setRigidDynamicFlag(PxRigidDynamicFlag::eKINEMA
```

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

Caveats:

- It is important to understand the difference between `PxRigidDynamic::setKinematicTarget()` and `PxRigidActor::setGlobalPose()` here. While `setGlobalPose()` would also move the actor to the desired position, it would not make that

actor properly interact with other objects. In particular, with `setGlobalPose()` the kinematic actor would not push away other dynamic actors in its path, instead it would go right through them. The `setGlobalPose()` function can still be used though, if one simply wants to teleport a kinematic actor to a new position.

- A kinematic actor can push away dynamic objects, but nothing pushes it back. As a result, a kinematic can easily squish a dynamic actor against a static actor, or against another kinematic actor. As a result, the squished dynamic object can deeply penetrate the geometry it has been pushed into.
- There is no interaction or collision between kinematic actors and static actors. However, it is possible to request contact information for these cases if `PxSceneFlag::eENABLE_KINEMATIC_PAIRS` or `::eENABLE_KINEMATIC_STATIC_PAIRS` gets set.

Sleeping

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

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

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

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

See the section [Callbacks and Customization](#) for more information. An actor goes to sleep when its kinematic energy is below a given threshold for a certain time. This threshold can be manipulated using the following methods:

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

Note: Kinematic actors go to sleep immediately if no target pose is set before a simulation step.

Objects automatically wake up when touched by an awake object, or the application changes the position or velocity. To explicitly wake up a sleeping object, or force an object to sleep, use:

```
void PxRigidDynamic::wakeUp(PxReal wakeCounterValue=PX_SLEEP_IN  
void PxRigidDynamic::putToSleep();
```



The API reference documents exactly which methods cause an actor to be woken up.

Solver Accuracy

When the motion of a rigid body is constrained either by contacts or joints, the constraint solver comes into play. The solver satisfies the constraints on the bodies by iterating over all the constraints restricting the motion of the body a certain number of times. The more iterations, the more accurate the results become. The solver iteration count defaults to 4 position iterations and 1 velocity iteration. Those counts may be set individually for each body using the following function:

```
void PxRigidBody::setSolverIterationCounts(PxU32 minPosition
```



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

Fast Rotation

Objects shaped like a pencil are difficult to simulate because they can store a lot of energy while rotating around a short axis, which is then converted to a very high rotational velocity when they start to rotate around a longer axis. High rotational velocities can lead to problems because certain linear approximations of the rotational motion fail to hold. For this reason the SDK automatically limits the rotational velocity of a body to a user definable maximum value. Because this may prevent intentional fast rotation in objects such as wheels, the user can override it on an per body basis:

```
void PxRigidBody::setMaxAngularVelocity(PxReal maxAngVel);
```

Mass Properties

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

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

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

The second uses the mass of the bottom snowball only, resulting in a bigger inertia. Later on, the center of mass is moved to the bottom of the actor. This approximation is by no means physically correct, but the resulting snowman feels a bit more filled.

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

center of mass squared).

The last snowman's mass properties are set up manually. The sample uses rough values for the moment of inertia to create a specific desired behavior. The diagonal tensor has a low value in X, and high values in Y and Z, producing a low resistance to rotation around the X-axis and high resistance around Y and Z. As a consequence, the snowman will wobble back and forth only around the X axis.

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

Damping

The rates at which rigid bodies dissipate angular and linear momentum are governed by damping rates. In the PhysX SDK two damping rates may be specified for a rigid body:

```
void PxRigidDynamic::setLinearDamping(PxReal linDamp);  
void PxRigidDynamic::setAngularDamping(PxReal angDamp);
```

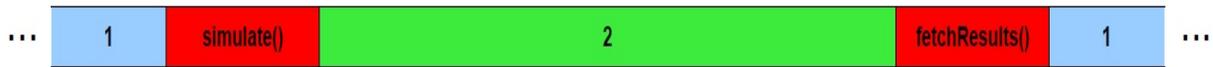
With a linear damping value of `linDamp` a rigid body will experience a damping force each update equal to $-\text{linDamp} \times \text{velocity}$. Similarly, a rigid body with damping rate `angDamp` will experience a damping torque $-\text{angDamp} \times \text{angularVelocity}$. The damping forces and torques always act against the velocity and angular velocity and are applied in a way that ensures that a moving and rotating rigid body will asymptotically approach the rest state in the absence of other forces.

This chapter will discuss read/write access to PhysX objects which are in a scene, specifically the time frame when access is allowed and how/when the changes will take effect. For objects in different scenes or outside a scene, the restrictions are given in the multithreading guidelines in *The Basics*.

General Data Access Rules

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

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



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

In the second time slot the simulation is running and in the process, reading and changing the state of objects. Concurrent access from the user might corrupt the state of the objects or lead to data races or inconsistent views in the simulation code. Hence the simulation code's view of the objects is protected from API writes, and any attributes the simulation updates are buffered to allow API reads. The consequences will be discussed in detail in the next section.

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

Double Buffering

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

From the user perspective, API changes are reflected immediately. For example, if the velocity of a rigid body is set and then queried, the new velocity will be returned. Similarly, if an object is created while the simulation is running, it can be accessed/modified as any other object. However, these changes are buffered so that the simulation code sees the object state as it was when *PxScene::simulate()* was called. For instance, changes to the filter data of an object while the simulation is running are ignored for collision pair generation of the running step, and will only affect for the next simulation step.

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

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

Events involving removed objects

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

- *PxSimulationEventCallback::onWake()*, *::onSleep()* events will not get fired if an object is involved which got deleted/removed during the

running simulation.

- *PxSimulationEventCallback::onContact()*, *::onTrigger()* events will get fired if an object is involved which got deleted/removed during the running simulation. The deleted/removed object will be marked as such (see *PxContactPairHeaderFlag::eDELETED_ACTOR_0*, *PxContactPairFlag::eDELETED_SHAPE_0*, *PxTriggerPairFlag::eDELETED_SHAPE_TRIGGER*). Furthermore, if *PxPairFlag::eNOTIFY_TOUCH_LOST*, *::eNOTIFY_THRESHOLD_FORCE_LOST* events were requested for the pair containing the deleted/removed object, then these events will be created.

Support

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

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

Memory Considerations

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

PxTask is a subsystem for managing compute resources for PhysX and APEX. It manages CPU and GPU compute resources, as well as SPU units on PlayStation3, by distributing Tasks to a user-implemented dispatcher and resolving Task dependencies such that Tasks are run in a given order.

Middleware products typically do not want to create CPU threads for their own use. This is especially true on consoles where execution threads can have significant overhead. In the PxTask model, the computational work is broken into jobs that are submitted to the game's thread pool as they become ready to run.

The following classes comprise the PxTask CPU resource management.

TaskManager

A TaskManager manages inter-task dependencies and dispatches ready tasks to their respective dispatcher. There is a dispatcher for CPU tasks, GPU tasks, and SPU tasks assigned to the TaskManager.

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

CpuDispatcher

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

PxTask includes a default CpuDispatcher implementation, but we prefer applications to implement this class themselves so PhysX and APEX can efficiently share CPU resources with the application.

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

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

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

The PxExtensions library has default implementations for all dispatcher types, the following code snippets are taken from SampleParticles and SampleBase and show how the default dispatchers are created. *mNbThreads* which is passed to *PxDefaultCpuDispatcherCreate* defines how many worker threads the CPU dispatcher will have.

Best performance is usually achieved if the number of threads is equal to the available hardware threads of the platform you are running on:

```
PxSceneDesc sceneDesc(mPhysics->getTolerancesScale());  
[...]  
// create CPU dispatcher with mNbThreads worker threads
```

```
mCpuDispatcher = PxDefaultCpuDispatcherCreate(mNbThreads);
if(!mCpuDispatcher)
    fatalError("PxDefaultCpuDispatcherCreate failed!");
sceneDesc.cpuDispatcher = mCpuDispatcher;
#ifdef PX_WINDOWS
    // create GPU dispatcher
    pxTask::CudaContextManagerDesc cudaContextManagerDesc;
    mCudaContextManager = pxTask::createCudaContextManager(cuda
    sceneDesc.gpuDispatcher = mCudaContextManager->getGpuDispat
#endif
    [...]
    mScene = mPhysics->createScene(sceneDesc);
```

Note: CudaContextManagerDesc support appGUID now. It only works on release build. If your application employs PhysX modules that use CUDA you need to use a GUID so that patches for new architectures can be released for your game. You can obtain a GUID for your application from Nvidia. The application should log the failure into a file which can be sent to NVIDIA for support.

CpuDispatcher Implementation Guidelines

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

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

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

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

BaseTask

BaseTask is the abstract base class for all PxTask task types. All task run() functions will be executed on application threads, so they need to be careful with their stack usage, use a little stack as possible, and they should never block for any reason.

Task

The Task class is the standard task type. Tasks must be submitted to the TaskManager each simulation step for them to be executed. Tasks may be named at submission time, this allows them to be discoverable. Tasks will be given a reference count of 1 when they are submitted, and the TaskManager::startSimulation() function decrements the reference count of all tasks and dispatches all Tasks whose reference count reaches zero. Before TaskManager::startSimulation() is called, Tasks can set dependencies on each other to control the order in which they are dispatched. Once simulation has started, it is still possible to submit new tasks and add dependencies, but it is up to the programmer to avoid race hazards. You cannot add dependencies to tasks that have already been dispatched, and newly submitted Tasks must have their reference count decremented before that Task will be allowed to execute.

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

APEX uses the Task class almost exclusively to manage CPU resources. The ApexScene defines a number of named Tasks that the modules use to schedule their own Tasks (ex: start after LOD calculations are complete, finish before the PhysX scene is stepped).

LightCpuTask

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

PhysX 3.0 uses LightCpuTasks almost exclusively to manage CPU resources. For example, each stage of the simulation update may consist of multiple parallel tasks, when each of these tasks has finished execution it will decrement the reference count on the next task in the update chain. This will then be automatically dispatched for execution when its reference count reaches zero.

Note: Even when using LightCpuTasks exclusively to manage CPU resources, the TaskManager startSimulation() and stopSimulation() calls must be made each simulation step to keep the GpuDispatcher synchronized.

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

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

```
class Crab: public ClassType, public physx::pxtask::LightCpuTask
{
public:
    Crab(SampleSubmarine& sample, const PxVec3& crabPos, Render
    ~Crab();
    [...]
```

```

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

[...]
}

```

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

```

pxtask::LightCpuTask& task = &mCrab;
mCpuDispatcher->submitTask(task);

```

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

```

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

```

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

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

```
// assume all tasks have a refcount of 1 and are submitted to t
// 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 para
//
// 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 refc
// 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();
}
```

PxTask provides both a CUDA heap manager and an execution scheduler. To use the execution scheduler (GpuDispatcher), you need to break up your CUDA pipeline into GpuTasks. Each GpuTask can be one of three flavors: Host-to-Device copies, CUDA kernels, or Device-to-Host copies. The execution scheduler will submit your tasks to the device in an order that optimizes the device throughput.

The following classes comprise the PxTask GPU resource management.

GpuTasks

GpuTask derives from Task (not BaseTask), so they must be submitted to a TaskManager for scheduling. GpuTasks are dispatched to the TaskManager's given GpuDispatcher when they are ready to run, and have a specialized launchInstance() method rather than run().

GpuTasks come in three flavors. They are *HostToDevice*, *Kernel*, and *DeviceToHost*. Each GpuTask must declare their flavor by returning a valid value from getTaskHint(). The GpuDispatcher must know the type of operations each task will perform in order to optimally combine work of multiple tasks, for example the dispatcher should run all available *HostToDevice* tasks before running all *Kernel* tasks, and all *Kernel* tasks before running any *DeviceToHost* tasks. This provides maximal kernel overlap and the least number of CUDA flushes.

CudaContextManager

The application should allocate one `CudaContextManager` for each CUDA context it wishes PhysX/APEX to use. In most typical scenarios, only a single CUDA context is required, as the user only has one GPU, or in the cases where two CUDA capable GPUs are present, only one is used for game physics.

If the application is using CUDA itself and wants to share its CUDA context with PhysX, it can provide its context to the `CudaContextManager` constructor. Otherwise the `CudaContextManager` will allocate its own context.

Note: If the application allocates its own CUDA context and gives it to a `CudaContextManager`, the context must be detached from any threads when the `CudaContextManager` takes ownership.

Each `CudaContextManager` provides a heap for managing memory allocated within the CUDA context (both device and host memory), and a `GpuDispatcher` instance for scheduling CUDA work on the device managed by the context. A scene's `TaskManager` `GpuDispatcher` pointer determines the CUDA context used by that scene.

The `CudaContextManager` has ownership of the CUDA context until it is released. Any code which wishes to use CUDA APIs must acquire the context from the `CudaContextManager` for the duration of the CUDA API calls.

The `CudaContextManager` also provides functions which perform Graphics/CUDA Interop mappings in a version safe manner. The intent is that application code can call these functions without needing to include CUDA headers.

Note: After allocating a `CudaContextManager`, it is required that you

check the return value of `contextIsValid()` and to discard the instance if it returns false. The CUDA context will not be used at all if this method fails. At runtime, the application should also poll the return of `GpuDispatcher::failureDetected()`. If it returns True, the CUDA context has encountered a non-recoverable error and no further CUDA work is possible.

GpuDispatcher

Every `CudaContextManager` instance creates and owns a `GpuDispatcher` instance. The `GpuDispatcher` is responsible for scheduling `GpuTasks` for the given context in the most efficient means possible.

On SM architectures 1.0 through 1.3, this mostly means overlapping the kernels in one task with copies (`HostToDevice` or `DeviceToHost`) in other tasks. On SM 2.0 and higher architectures, the `GpuDispatcher` will try to maximize the kernel overlap (out-of-order completion).

These types of overlap optimizations are only possible when more than one `GpuTask` is ready to run at the same time, so if parts of your CUDA pipeline are parallelizable with itself, you should break the work into multiple `GpuTasks` and set dependencies accordingly.

Dependencies between `GpuTasks` determines the order they are dispatched through the `GpuDispatcher`, which enforces the order those tasks are submitted to CUDA. Note however that independent tasks may be rearranged for maximum efficiency (see `GpuTasks` section for details).

If you have a task which cannot start until actual CUDA results are available (this almost always implies a `DeviceToHost` copy), then one of your `GpuTasks` must call the `GpuDispatcher::addCompletionPrereq(BaseTask&)` method to register a *CUDA completion* dependency on the given task. The `GpuDispatcher` will immediately increment the reference count of the given Task and then decrement it again when all of the work submitted up to CUDA at the time `addCompletionPrereq()` was called has been completed.

Note: Since `addCompletionPrereq()` takes a `BaseTask` reference, CUDA completion tasks may be `GpuTasks`, `LightCpuTasks` or normal `Tasks`.

Warning: CUDA completion dependencies are volatile in that they are

not created until your GpuTask pipeline is essentially finished running. In order to prevent your completion task from starting before the CUDA completion dependency can be added to its reference count, it is recommended practice to add normal *startAfter* dependency on the GpuTask that will set the CUDA completion dependency.

The GpuTask "run" function, `launchInstance()` takes two parameters, a CUDA stream and a kernel index. `launchInstance()` will always be called with the same CUDA stream while the kernel index will increment from 0 until the GpuTask is finished. Returning false from `launchInstance` tells the GpuDispatcher that your GpuTask is done launching CUDA work and should be released.

The reason for the iterative kernel launching is that optimal kernel scheduling on SM 2.0 GPUs requires kernels in different streams to be launch interleaved to minimize the blocking that occurs when kernels have to wait for their own stream to become idle.

Note: HostToDevice and DeviceToHost GpuTasks can ignore the `kernelIndex` and simply perform all of their copies, then return false. There is no benefit to interleaving copies in streams.

GpuTasks have three flavors to avoid serious performance pitfalls, especially when multiple pipelines are sharing a single CUDA context. When GpuTasks are dispatched they are binned by the GpuDispatcher by flavor and the dispatcher itself runs in a three-phase state machine: HtoD -> Kernel -> DtoH -> HtoD, etc. At each phase, the GpuDispatcher interleaves `launchInstance()` calls to all running GpuTasks in that phase until there are no more tasks remaining.

Note: All of the work submitted to CUDA in one round trip of the state machine is referred to as a GpuTaskBatch. It should roughly correlate to a single submission of work to the GPU, unless you submitted more work than could fit and caused an implicit flush.

Note: The GpuDispatcher has a utility kernel to perform memory copies as part of the Kernel GpuTasks. See GpuDispatcher::launchCopyKernel()

When a GpuTask's launchInstance() function returns false, the GpuDispatcher immediately releases the task and allows its dependency resolutions to schedule new GpuTasks of the same flavor so those can be dispatched in the same pass through the state machine (if the newly dispatched GpuTasks do not match the current GpuDispatcher phase, they are binned). In short, three kernel GpuTasks that each launch a single kernel and have linear dependencies will be executed in exactly the same way as one kernel GpuTask which launches three kernels (one per launchInstance() call).

Note: Now that the concept of the GpuDispatcher state machine has been introduced, it is worth noting that CUDA completions are queued within the GpuDispatcher. The completion tasks' reference counts are incremented immediately, but the CUDA work is not flushed until the GpuDispatcher transitions from the DtoH to HtoD phase. This is an optimization tailored for multiple pipeline situations where we want to minimize the number of CUDA flushes (aka, maximize the amortization).

To make GpuTask scheduling flexible, the GpuDispatcher manages the assignment of CUDA streams to each task. It does this by managing a pool of free CUDA streams, and a few extra flags and fields on each GpuTask. The logic works like this:

1. If a GpuTask has no GpuTask predicate dependencies (essentially means it is the first GpuTask in the pipeline), it is assigned a new CUDA stream from the free pool.
2. When a GpuTask completes launching CUDA work and is released, it passes its CUDA stream to its first dependent GpuTask.
3. If a GpuTask has more than one dependent tasks, the remaining tasks are given new CUDA streams (from the free pool) and have a flag set on them that informs the GpuDispatcher it must perform a WFI (wait-for-idle) on the GPU before that task's CUDA work is allowed to run.
4. If a GpuTask

has more than one GpuTask predicate dependency (a joining of two CUDA streams), it takes the CUDA stream of the first predicate and is marked as requiring a WFI.

Note: The GpuDispatcher implements the WFI by issuing a non-blocking cuEventRecord in stream 0. For all architectures up to SM2.0, this causes all previously launched CUDA work to complete before any later submitted work to start - a sync point.

The GpuDispatcher does not release CUDA streams back to the free pool until it knows that all of the TaskManagers that are using the GpuDispatcher have finished submitting GpuTasks. This is the primary reason the TaskManager calls the startSimulation() and stopSimulation() methods of the GpuDispatcher (the other is to initialize profiling), and also why the TaskManager itself needs to be notified of simulation start and stop, even if LightCpuTasks are being used for all non-Gpu work.

BlockingWait

CUDA completion notifications are handled by a special BlockingWait thread of the GpuDispatcher. The main GpuDispatcher thread queues the completion task references and the CUDA EventRecord they are waiting for, and the BlockingWait thread waits for each of them in turn and then decrements the reference count of the provided task. The BlockingWait thread is either in a blocking wait for completion tasks or in a blocking wait for a CUDA EventRecord.

Note: Profiling events for the blocking wait thread are prefixed with GDB, while profiling events for the GpuDispatcher dispatch thread are prefixed with GD.

Copy Engine Kernel

The GpuDispatcher has a utility kernel that can execute an arbitrary number of memory copies or memory sets in parallel on the GPU itself. This is the recommended method for performing all host-to-device, device-to-host, device-to-device copies.

1. By using the utility kernel from a Kernel flavor GpuTask, you avoid the need for creating DeviceToHost or HostToDevice flavor GpuTasks.
2. By batching together many copies into a single kernel launch, you save driver overhead and save space inside your push buffer.
3. By only using Kernel flavored GpuTasks, there is a much greater opportunity to overlap both copies and kernels on SM2.0 architectures and greater.
4. By using a copy kernel, the copies will show up in your profile.

See `GpuDispatcher::launchCopyKernel()` for the details on using the utility kernel. The primary point to remember is that the copy descriptors are read from host memory if you perform more than one copy at a time, so the descriptors must be allocated from page locked memory.

CUDA Profiling

PxTask, more specifically the GpuDispatcher, supports profiling at the CTA level. This requires a number of preparation steps:

1. Your application must register kernel names with the GpuDispatcher using the registerKernelNames() method. It returns an index to the start of an ID range, each kernel should use this index as an offset when constructing IDs for use with fillKernelEvent().
2. Each kernel must call KERNEL_START_EVENT(profileBuffer, kernelID) and KERNEL_STOP_EVENT(profileBuffer, kernelID) at the appropriate times (start and stop of the kernel). This requires the inclusion of PsPAGpuEventSrc.h
3. At run-time, the kernel ID and current profiling buffer must be passed to each kernel (see GpuDispatcher::getCurrentProfileBuffer()). Note that the profile buffer will be NULL unless you are actively collecting events.

The application needs to tune a couple of parameters for their needs, both in PsPAGPUEventSrc.h.

ENABLE_CTA_PROFILING - this flag toggles CTA level event collection globally. Turning it off removes the code (and probably register) overhead in each kernel, and disables the allocation of profile buffers and their copying overhead.

NUM_CTAS_PER_PROFILE_BUFFER - needs to be larger than the count of CTAs launched in your largest batch of kernel launches. The BlockingWait thread will issue a warning if it finds that CTA events were lost because the buffer was too small.

NUM_CTA_PROFILE_BUFFERS - the number of profile buffers allocated. This should be the number of GpuTaskBatches you intend to launch each simulation step. If it is too small, later GpuTaskBatches will not have CTA

level profiling.

Warning: After changing `ENABLE_CTA_PROFILING` or `NUM_CTAS_PER_PROFILE_BUFFER`, you must rebuild all of your CUDA source files.

CTA profiling works on a `GpuTaskBatch` basis, and is only active when the `GpuDispatcher` notices an event collector is attached and active. At the start of each batch the `GpuDispatcher` allocates an idle profile buffer, issues a small `memset` kernel to clear the 8 word header of the buffer, then issues a saturation kernel in order to record the clock on each SM. These clocks are used by the event viewer to line up the SM events, though this only seems to work well on SM 2.0. The `GpuDispatcher` then executes `GpuTasks` as normal until the state machine finishes all three phases (HtoD, Kernel, DtoH). At the end the `GpuDispatcher` issues another `Saturate` kernel to delineate the end of all of the DtoH copy times, then issues a DtoH copy for the profile buffer itself, then finally issues a blocking `cuEventRecord` to flush everything to the GPU for execution.

The profile buffer ID is sent to the `BlockingWait` thread along with the blocking `EventRecord`, and once the event has completed the `BlockingWait` thread is responsible for parsing the CTA events and emitting profiling data to the collector.

Note that much effort was devoted to minimize the performance impact of collecting all this data, but some penalties are unavoidable. Also, the copy back to the host can be substantial depending on the value you chose for `NUM_CTAS_PER_PROFILE_BUFFER`.

Note: The CTA based profiling requires global memory atomics, and thus SM architecture 1.1 or above. SM 1.0 cards will see the `GpuTaskBatch` bar, which is measured via non-blocking `cuEventRecords`, but they will not see any CTAs.

Note: By default, the BlockingWait thread will parse the CTA events in the profile buffer and attempt to piece together the run-times of each kernel. This is done with a simplistic heuristic and does not work very well with short kernels and SM clock skews. It seems to work best with SM 2.0 devices. By changing `EMIT_CTA_EVENTS` to 1, near the top of `BlockingWait.cpp`, the BlockingWait thread will instead emit one profile bar for each CTA, giving you a more truthfull picture of how the SMs were used by each kernel. You will still need to account for the SM clock skew in your analysis.

PhysX provides methods in PxScene to perform collision queries against the actors in the scene. The queries come in three types: raycasts, sweeps and overlaps. Moreover, each query type has several variants. The API supports batching of queries via the *PxBatchQuery* interface, which may provide significant speedups for some platforms and scenarios. In particular, on PS3 only batched queries are SPU-accelerated.

Raycast queries

A raycast intersects a user-defined ray with the whole scene. PhysX supports three types of raycasts.

- *raycastAny*
- *raycastSingle*
- *raycastMultiple*

raycastAny returns a single boolean result. It is optimized for cases where the hit shape and impact point are not important. A typical use case would be AI line-of-sight queries.

The simplest *raycastAny* call looks like this:

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

// Raycast against all static & dynamic objects (no filtering)
// The main result from this call is the boolean 'status'
bool status = scene->raycastAny(origin, unitDir, maxDistance, hit);
```

raycastSingle returns a single result: the closest touched shape along the ray, if any, along with the exact hit information. This might be used for simple bullets, for example.

The simplest *raycastSingle* call looks like this:

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

// [in] Define what parts of PxRaycastHit we're interested in
```

```

const PxSceneQueryFlags outputFlags = PxSceneQueryFlag::eDISTAN
// Raycast against all static & dynamic objects (no filtering)
// The main result from this call is the closest hit, stored in
bool status = scene->raycastSingle(origin, unitDir, maxDistance

```

raycastMultiple finds all the objects touched by the ray, along with all the corresponding hits. For example this could be used for armor-piercing bullets.

The simplest *raycastMultiple* call looks like this:

```

PxScene* scene;
PxVec3 origin = ...; // [in] Ray origin
PxVec3 unitDir = ...; // [in] Normalized ray dir
PxReal maxDistance = ...; // [in] Raycast max distan

// [in] Define what parts of PxRaycastHit we're interested in
const PxSceneQueryFlags outputFlags = PxSceneQueryFlag::eDISTAN

bool blockingHit; // [out] Tells whether hit
const PxU32 bufferSize = 256; // [in] size of 'hitBuffer'
PxRaycastHit hitBuffer[bufferSize]; // [out] Results will be s

// Raycast against all static & dynamic objects (no filtering)
// The main result from this call are all hits along the ray, s
PxI32 nbHits = scene->raycastMultiple(origin, unitDir, maxDista
hitBuffer, bufferSize, bl

```

Notes:

- Solid objects (sphere, capsule, box, convex) are defined as closed (i.e. they include their boundaries.)
- A plane is a closed half-space
- Heightfields are also closed and solid

When raycasting against solid objects, the rays include their endpoints. Any intersection between a ray and a solid object results in a hit report.

For solid objects (sphere, capsule, box, convex) and heightfields PhysX will report a hit:

1. if the start point or the end point is inside a solid object or heightfield.
2. if the start point or the end point is on the surface of a solid object or heightfield.

In the case that the start point is inside a solid object or heightfield:

1. the reported hit distance is set to zero.
2. the hit normal is in the opposite direction to the ray.
3. the hit impact position is the start point of the ray.

For Planes: If the start point is behind the plane's surface, no hit will be reported even in the case that the ray intersects the plane. If the start point is on the plane, a hit with zero distance will be reported.

There are two kinds of meshes: double-sided meshes and single-sided meshes. Backface culling is enabled for single-sided meshes, disabled for double-sided ones. A double-sided mesh is defined with the `PxMeshGeometryFlag::eDOUBLE_SIDED` flag.

If the start point or end point is on the surface for both kinds of meshes, PhysX will report a hit. If the start point is on the back of a mesh triangle, and the ray intersects the triangle, PhysX will:

1. report a hit for the double-sided mesh.
2. report no hit for the single-sided mesh.

To summarize, here are the tables for the definition of raycast behavior.

Solid Shape & Heightfield

	Start Pt Inside	Start Pt On the Surface	Start Pt Outside
End Pt Inside	HIT	HIT	HIT
End Pt On the			

Surface	HIT	HIT	HIT
End Pt Outside	HIT	HIT	MISS

Plane

	Start Pt Behind	Start Pt On the Plane	Start Pt in Front
End Pt Behind	MISS	HIT	HIT
End Pt On the Plane	MISS	HIT	HIT
End Pt in Front	MISS	HIT	MISS

Double-Sided Mesh

	Start Pt Behind Triangle	Start Pt On the Triangle	Start Pt in Front of Triangle
End Pt Behind Triangle	MISS	MISS	HIT
End Pt On the Triangle	HIT	MISS	HIT
End Pt in Front of Triangle	HIT	MISS	MISS

Single-Sided Mesh

	Start Pt Behind Triangle	Start Pt On the Triangle	Start Pt in Front of Triangle
End Pt Behind Triangle	MISS	MISS	HIT
End Pt On the Triangle	MISS	MISS	HIT

**End Pt in Front
of Triangle**

MISS

MISS

MISS

Sweep Queries

A sweep query sweeps a shape from a given point along a given direction, and collisions with scene objects are reported. As with raycasts, PhysX supports three kinds of sweep tests with the same distinctions and limitations:

- *sweepAny*
- *sweepSingle*
- *sweepMultiple*

Each kind of sweep test has two versions: geometry sweeping and compound geometry sweeping. The compound version sweeps all specified geometry objects through space and finds all rigid actors that get hit along the sweep. Each result contains data as specified by the `outputFlags` field. A typical use case would be character sweeping queries.

The distance of sweep must be larger than 0. It will be clamped to `PX_MAX_SWEEP_DISTANCE` which is defined in file `PxScene.h`.

sweepAny is optimized for cases in which a simple boolean result is enough. All that matters is that there was a hit, and the exact impact point or which shape has been touched is not important.

The simplest *sweepAny* call looks like this:

```
PxScene* scene;
PxGeometry geometry = ...;           // [in] Geometry of object
PxTransform pose = ...;              // [in] Pose of the object
PxVec3 unitDir = ...;               // [in] Normalized sweep direction
PxReal maxDistance = ...;          // [in] Sweep max distance
PxSweepHit hit;                     // [out] Sweep results

// [in] Define what parts of PxSweepHit we're interested in
const PxSceneQueryFlags outputFlags = PxSceneQueryFlag::eDISTANCE;

// Sweep against all static & dynamic objects (no filtering)
```

```
// The main result from this call is the boolean 'status', return
bool status = scene->sweepAny(geometry, pose, unitDir, maxDistance);
```

sweepSingle returns a single result: the closest touched shape along the sweep direction, if any, along with the exact hit information.

The simplest *sweepSingle* call looks like this:

```
PxScene* scene;
PxGeometry geometry = ...;           // [in] Geometry of object
PxTransform pose = ...;              // [in] Pose of the object
PxVec3 unitDir = ...;               // [in] Normalized sweep direction
PxReal maxDistance = ...;           // [in] Sweep max distance
PxSweepHit hit;                      // [out] Sweep results

// [in] Define what parts of PxSweepHit we're interested in
const PxSceneQueryFlags outputFlags = PxSceneQueryFlag::eDISTANCE;

// Sweep against all static & dynamic objects (no filtering)
// The main result from this call is the boolean 'status', return
bool status = scene->sweepSingle(geometry, pose, unitDir, maxDistance);
```

sweepMultiple finds all the objects touched by the sweep volume, along with all the corresponding hits.

The simplest *sweepMultiple* call looks like this:

```
PxScene* scene;
PxGeometry geometry = ...;           // [in] Geometry of object
PxTransform pose = ...;              // [in] Pose of the object
PxVec3 unitDir = ...;               // [in] Normalized sweep direction
PxReal maxDistance = ...;           // [in] Sweep max distance
PxSweepHit hit;                      // [out] Sweep results

// [in] Define what parts of PxSweepHit we're interested in
const PxSceneQueryFlags outputFlags = PxSceneQueryFlag::eDISTANCE;

bool blockingHit;                    // [out] Tells whether hit
const PxU32 bufferSize = 256;       // [in] size of 'hitBuffer'
PxSweepHit hitBuffer[bufferSize];    // [out] Results will be stored here
```

```
// Sweep against all static & dynamic objects (no filtering)
// The return value is the number of hits in the buffer, or -1
PxI32 nbHits = scene->sweepMultiple((geometry, pose, unitDir, m
```

The currently supported input shapes are boxes, spheres, capsules and convex.

Notes:

- Solid objects (sphere, capsule, box, convex) are defined as closed (i.e. they include their boundaries.)
- A plane is a closed half-space
- Triangle Mesh is defined as thin triangle surface.
- Heightfield is defined as thin triangle surface, the thickness of heightfield is ignored for sweeping test.
- Sweeping volumes are defined as closed (i.e. they include their boundaries.)

PhysX does not use tolerance when sweeping. Instead, users can add their own tolerance to the sweeping volume by using a larger sweeping geometry or longer sweeping distance before filling the query for PhysX. For example, use a large tolerance to get more broad results, use a small tolerance to get more accurate results.

Sweeps with Initial Intersection

By default PhysX sweeps return an undefined result if the initial position of the swept volume intersects another shape. Setting the *PxSceneQueryFlag::eINITIAL_OVERLAP* flag specifies that additional tests will be performed to ensure a defined result in this case. When this flag is set and an initial overlap is found, the *PxSceneQueryFlag::eINITIAL_OVERLAP_KEEP* flag determines whether the overlap generates a hit result or is ignored. If a hit result is generated, the distance is set to zero, and the returned normal is set to the opposite of the sweep direction. If none of the flags are set the behaviour is undefined.

There is a slight performance hit for initial overlap checks.

Overlap Queries

In overlap queries, a shape is collided against the objects in the scene, and any touching object is reported. PhysX only supports two kinds of overlap tests:

- *overlapAny*
- *overlapMultiple*

overlapAny returns a single boolean, and is optimized for testing whether a given volume of space is empty or not.

The simplest *overlapAny* call looks like this:

```
PxScene* scene;
PxGeometry geometry = ...;           // [in] Geometry of object
PxTransform pose = ...;              // [in] Pose of the object
PxShape* hit;                        // [out] Overlap results

// Overlap against all static & dynamic objects (no filtering)
// The main result from this call is the boolean 'status', return
bool status = scene->overlapAny(geometry, pose, hit);
```

overlapMultiple returns the set of all overlapping objects.

The simplest *overlapMultiple* call looks like this:

```
PxScene* scene;
PxGeometry geometry = ...;           // [in] Geometry of object
PxTransform pose = ...;              // [in] Pose of the object
const PxU32 bufferSize = 256;       // [in] size of 'hitBuffer'
PxShape* hitBuffer[bufferSize];     // [out] Results will be stored here

// Overlap against all static & dynamic objects (no filtering)
// The return value is the number of hits in the buffer, or -1
PxI32 hitNum = scene->overlapMultiple(geometry, pose, hitBuffer);
```

There is no "overlapSingle" case here because overlap queries do not have a given direction, and thus we cannot define a "closest" or "first" hit in this case.

The currently supported input shapes are boxes, spheres, capsules and convex.

Notes:

- Solid objects (sphere, capsule, box, convex) are defined as closed (i.e. they include their boundaries.)
- A plane is a closed half-space
- Triangle Mesh is defined as thin triangle surface.
- Heightfield is defined as extruded triangle surface with thickness. Overlap geometries who do not intersect with heightfield surface but are within the extruded space also report a overlap hit.
- Overlapping volumes are defined as closed.

PhysX does not use tolerance when overlapping. Instead, users can add their own tolerance to the overlapping volume by using a larger overlapping geometry before fill the query for PhysX. For example, use a large tolerance to get more broad results, use a small tolerance to get more accurate results.

Filtering

There are several ways to filter out undesired shapes from scene queries. Each scene query accepts the following filtering-related parameters:

- a `PxSceneQueryFilterData` structure, containing both `PxSceneQueryFilterFlags` and `PxFilterData`
- an optional `PxSceneQueryFilterCallback`

The first level of filtering is given by the `PxSceneQueryFilterFlag::eSTATIC` and `PxSceneQueryFilterFlag::eDYNAMIC` flags. These flags control whether the query takes static and/or dynamic shapes into account. This is the most efficient way to filter out all static shapes. For example an explosion effect which applies forces within a region could use the `overlapMultiple` query with a sphere shape, and the `PxSceneQueryFilterFlag::eDYNAMIC` flag to only consider dynamic objects, since it is useless to apply forces to static objects.

Returning to the initial raycast code snippets, a `raycastAny` call against static shapes only would be written like this:

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

// [in] Define filter for static objects only
PxSceneQueryFilterData filterData(PxSceneQueryFilterFlag::eSTATIC);

// Raycast against static objects only
// The main result from this call is the boolean 'status'
bool status = scene->raycastAny(origin, unitDir, maxDistance, hit, filterData);
```

In case of triangle meshes it is possible to receive multiple hits per mesh. To

enable triangle mesh multiple hits set
PxSceneQueryFilterFlag::eMESH_MULTIPLE flag. This flag can be set in the scene query filter data and can be also set/cleared in pre-filter shader.

The second level of filtering is controlled by the *PxFilterData*, a 128-bit bitmask used in a built-in filtering equation. Each shape has a bitmask, set using *PxShape::setQueryFilterData()*, and the query also has a bitmask.

The query data is used differently by batched and unbatched queries (see below for batched queries.) For unbatcher queries, the following rules are applied:

- If the query's bitmask is all zero, the shape is kept
- Otherwise, if the bitwise-AND value of the query's bitmask and the shape's bitmask is zero, the shape is skipped

Or in other words:

```
PxU32 keep = (query.word0 & object.word0) | (query.word1 & obje
```

The hardcoded filtering equation avoids the function call overhead of the filtering callback, while still providing reasonable filtering capabilities. This is similar to the "active groups" from previous versions of PhysX. For example, one can simply use the first word of filterData (word0) to emulate the behavior of previous PhysX versions. The active groups could be defined like this:

```
enum ActiveGroup
{
    GROUP1    = (1<<0),
    GROUP2    = (1<<1),
    GROUP3    = (1<<2),
    GROUP4    = (1<<3),
    ...
};
```

When shapes are created, they can be put in a single group, for example

GROUP1:

```
PxShape* shape; // Previously created shape

PxFilterData filterData;
filterData.word0 = GROUP1;
shape->setQueryFilterData(filterData);
```

Or in several groups, for example GROUP1 and GROUP3:

```
PxShape* shape; // Previously created shape

PxFilterData filterData;
filterData.word0 = GROUP1|GROUP3;
shape->setQueryFilterData(filterData);
```

Then when performing a scene query, select which groups are active for the query - for example GROUP2 and GROUP3 here:

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

// [in] Define what parts of PxRaycastHit we're interested in
const PxSceneQueryFlags outputFlags = PxSceneQueryFlag::eDISTANCE;

// [in] Raycast against GROUP2 and GROUP3
PxSceneQueryFilterData filterData = PxSceneQueryFilterData();
filterData.data.word0 = GROUP2|GROUP3;

bool status = scene->raycastSingle(origin, unitDir, maxDistance,
```

A built-in equation is never flexible enough, so the last level of filtering is provided by a filtering callback. The filtering callback must be passed to the query as a *PxSceneQueryFilterCallback*. Set the *PxSceneQueryFilterFlag::ePREFILTER* and *PxSceneQueryFilterFlag::ePOSTFILTER* flags to determine whether to

filter before accurate per-shape collision, afterward, or both. Filtering early allows shapes to be efficiently discarded before the potentially expensive collision test. On the other hand, the results of that test may be required in order to determine whether a shape should be discarded or not.

The implementation of the filtering callback must return a *PxSceneQueryHitType*. These types define three different kinds of behavior:

- *eNONE* indicates that the shape must simply be discarded from any further processing
- *eBLOCK* indicates that the shape is a "blocking hit", and any shapes located further away can be ignored
- *eTOUCH* indicates that the shape is a "touching hit", and while this shape will be recorded and included in the query's report, the query will continue to seek hits further away

The *eNONE* and *eBLOCK* types correspond to the intuitive definition of a standard filtering mechanism: discard the shape (*eNONE*) or keep it (*eBLOCK*). *eTOUCH* provides support for scenarios such as bullets going through windows (breaking them on their way), or through the leaves of a tree (making them rustle). That is, cases where it is useful to apply various effects to touched objects, without actually blocking the bullet. Note that *eTOUCH* is only useful for scene queries reporting multiple hits. For queries returning a single result, touching hits are simply ignored (similar to *eNONE*).

To use filter-style querying in unbatched queries, similar to that performed by the simulation filter shader and for batched queries, add a *filterData* field to the query callback object and call your filter function there.

Caching

Scene queries can sometimes be accelerated using *PxSceneQueryCache* objects. This is especially true for *raycastAny*, *raycastSingle* and *sweepSingle* queries. The cache object defines which shape - or even, in the case of triangle meshes, which triangle - should be tested first. For queries with high temporal coherence, this can provide significant performance gains. A good strategy to capture that coherence is simply to fill the cache object of a given query with the results (last touched shape, last touched triangle) from the previous frame.

For example there is a high probability that an AI visibility query will return the same vision-blocking shape for several frames. Using *raycastAny* with a properly filled *PxSceneQueryCache* object will allow PhysX to test a single shape - or a single triangle! - before traversing internal pruning structures, and in the case of a "cache hit" the pruning structures can be bypassed entirely. Caching in such a scenario works like this:

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

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

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

// Perform raycast query using the cache
PxSceneQueryHit hit;
const bool status = scene->raycastAny(origin, unitDir, maxDistance, hit, cache);
if(status)
{
    // We hit a shape. Cache it for next frame.
    persistentCache.shape = hit.shape;
}
```

```
    persistentCache.faceIndex = hit.faceIndex;
}
else
{
    // We did not hit anything. Reset the cache for next frame.
    persistentCache = PxSceneQueryCache();
}

```

Caching can also be useful in queries looking for the "closest" hit. Here, testing the previously closest object first can allow PhysX to shorten the query distance very early, leading to fewer collision tests overall.

PhysX cannot detect stale pointers, so the application is responsible for updating caches when shapes are deleted.

Batched queries

Batched queries provide an interface, *PxBatchQuery*, where queries can be batched together and executed all at once. *PxBatchQuery* buffers the raycast, overlap and sweep queries until *PxBatchQuery::execute()* is called.

Use *PxScene::createBatchQuery* to create *PxBatchQuery* object.

The hardcoded filtering equation is not used for batched queries. Instead it is replaced with two filter shaders, respectively running before (*PxBatchQueryPreFilterShader*) and after (*PxBatchQueryPostFilterShader*) the accurate collision tests. Set the *preFilterShader* and *postFilterShader* in *PxBatchQueryDesc* correspondingly.

The *filterShaderData* will be copied into *PhysX* and passed to the filter shader by the *constantBlock* parameter.

Results are written to the user-defined buffers in *PxBatchQueryDesc*, in the same order the queries were given to the *PxBatchQuery* object. The results buffer and hits buffer for the needed query type must be set. The SDK ignores batched queries with NULL results buffer or NULL hits buffer.

PS3 specific limitations and how to write an SPU Query Filter Shader are captured in the User's PS3 Guide "SPU Simulation Restrictions" and "SPU Query Filter Shaders" chapters respectively.

The chapter *Scene Queries* shows how to query a PhysX scene or rather its objects for raycast, sweep and overlap hits. However, sometimes the objects to query are known up front and only the hit location on the object surface, for example, is of interest. Sometimes those objects might not even be PhysX actors or shapes but simple geometric volumes like spheres or boxes. PhysX exposes an API for such basic query functionality through the *PxGeometryQuery* and *PxMeshQuery* interfaces.

Geometry Objects

In the following sections, the mentioned simple geometry objects will consist of two parts. One is a descriptor for the geometric shape, the other is a world transform to specify the position and rotation of the object. The former gets defined by the same class which is used for *PxShape* objects already, that is, *PxGeometry*. See [Shapes and Geometries](#) for more information on this class.

PxGeometryQuery

Three types of tests are offered for simple geometric volumes: raycast, sweep and overlap tests.

As an example, let us assume we are interested in knowing whether a given box hits a given sphere if the box moves a certain distance along a specified direction (and if the sphere does get hit, we would like to know the first point of contact on its surface). With the *PxGeometryQuery* API the code would look like this:

```
PxSweepHit hitInfo;  
bool gotHit = PxGeometryQuery::sweep(unitDir, distance,  
                                     boxGeom, boxPose,  
                                     sphereGeom, spherePose,  
                                     hitInfo);
```

The first two parameters of the provided method are the normalized direction and the distance of the movement. The third and fourth parameter describe the geometry object to sweep. In the example, this would be a *PxBoxGeometry* object and its world transform. The fifth and sixth parameter describe the geometry object to test against. This would be a *PxSphereGeometry* object together with its global pose, in our example. The last parameter in the code snippet will hold the hit information if the method returns true, i.e., if the box does hit the sphere on its motion path.

For limitations and detailed parameter descriptions please refer to the corresponding API documentation.

PxMeshQuery

For triangle meshes and heightfields, additional functionality is provided to extract the triangles which lie inside a specified geometric volume. The following example code will show the API usage for a scenario, where the mesh triangles which lie within a given spherical volume should get detected:

```
PxU32 triangleIndexBuffer[bufferSize];
PxU32 startIndex = 0;
bool bufferOverflowOccured = false;
PxU32 nbTriangles = PxMeshQuery::findOverlapTriangleMesh(sphere
                                                         meshGe
                                                         triang
                                                         startI

for(PxU32 i=0; i < nbTriangles; i++)
{
    PxTriangle tri;
    PxU32 vertexIndices[3];
    PxMeshQuery::getTriangle(meshGeom, meshPose, triangleIn

    ... // process triangle info
}
```

The *findOverlapTriangleMesh* method is used to extract the indices of the triangles. The first two parameters describe the geometry object which defines the volume to look for triangles in. In the given scenario, this would be a *PxSphereGeometry* object and its world transform. The next two parameters specify the mesh/heightfield to test against. This would be a *PxTriangleMeshGeometry* object and its global pose, in this example. The fifth parameter is a user buffer to store the indices of the found triangles to, followed by a parameter holding the maximum size of this buffer. The second to last parameter can be used together with the last parameter to handle cases where the provided buffer was not large enough to store all the found triangle indices. The last parameter is a boolean which gets set to true if not all triangle indices could get stored. In that case, the method can get

called once again setting the second to last parameter (*startIndex*) to the number of triangle indices extracted so far such that the next block of triangle indices which did not fit into the user buffer can get fetched. The following loop in the sample code then iterates over the found triangle indices and uses the *getTriangle* interface to translate an index to a triangle and optionally triangle vertex indices.

More in-depth information about the interface is provided in the API documentation.

In this chapter we will look at a number of callback functions the SDK provides that let the user listen to simulation events and even customize some parts of the simulation. The callback functions are implemented as member functions of classes which the user is expected to subclass, providing custom implementations for the functions. This is the same mechanism that we already explained in the chapter *The Basics*, to let the user provide custom allocators and error notification.

Simulation Events

The simplest type of simulation callbacks are the events. The application may simply listen in on these without needing to react in any way. There is only one restriction on the code you may put in the callbacks: Any SDK state change from them is disallowed. This may be a bit surprising given that normally the SDK is set up to be double buffered, and writes are permitted while the simulation is running in the background, as any new state is written to an inactive state backbuffer. However, these event callbacks are not called from within the simulation thread, but rather from inside `fetchResults()`. At that point some buffered operations have already been processed, so the situation is more fragile. All write operations to the scene should be buffered and carried out after `fetchResults()` returns.

Inside `fetchResults()` among other things we 'swap the buffers'. This means that we copy the objects' simulation results to their API-visible state. Some event callbacks happen before this swap, and some after. The events that happen before are:

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

When these events get received, the shapes, actors, etc. will still be in the state they were in before the simulation ran. This is preferable, because these events were detected early on during the simulation, before objects were integrated (moved) forward. For example, a pair of shapes that get an `onContact()` to report that they are in contact will still be in contact when the call is made, even though after `fetchResults()` returns, they may have bounced apart again.

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

- `onSleep`

- onWake

Sleep information is updated after objects have been integrated, so it makes sense to send these events after the swap.

You 'listen' to any of these events by doing two things: First, define a callback function by subclassing `PxSimulationEventCallback`. Not all of its member functions have to be implemented, only the ones to be overwritten. For constraint break events, this is the only thing that needs to be done. For sleep and wake events, you must raise the flag `PxActorFlag::eSEND_SLEEP_NOTIFICATIONS` on all actors from which notifies are desired. To get `onContact` and `onTrigger` events, in the filter shader callback, set a flag for all pairs of interacting objects for which you wish to receive these events. See the section on using collision filtering below for details.

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

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

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

                break;
            }
        }
    }
}
```

SampleSubmarine is a subclass of PxSimulationEventCallback. onContact receives the pair for which the requested contact events have been triggered. The above function is only interested in eNOTIFY_TOUCH_FOUND events, which are raised whenever two shapes start to touch. In fact it is only interested in touch events of the submarine -- which is checked in the second if-statement. It then goes on to assume that the second actor is a mine (which works in this example because the sample is configured such that no other contact reports will get sent when a submarine actor is involved). After that, it adds the mine to a set of mines that should explode during the next update.

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

Frequently, users are only interested in contact reports, if the force of impact is larger than a certain threshold. This allows to reduce the amount of reported pairs which need to get processed. To take advantage of this option the following additional configurations are necessary:

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

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

Collision Filtering

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

Before we can understand what the sample does to achieve this, we need to understand the possibilities of the SDK filtering system. Because filtering potentially interacting pairs happens in the deepest parts of the simulation engine, and needs to be applied to all pairs of objects that come near each other, it is particularly performance sensitive. The simplest way to implement it would be to always call a callback function to each potentially interacting pair, where the application, based on the two object pointers could determine, using some custom logic -- like consulting its game data base -- whether the pair should interact. Unfortunately this quickly becomes too slow if done for a very large game world, especially if the collision detection processing happens on a remote processor like the GPU or an other kind of vector processor with local memory, which would have to suspend its parallel computations, interrupt the main processor that runs game code, and have it execute the callback before it can continue. Even if it were to be executed on a CPU, it would likely be done so simultaneously on multiple cores or hyperthreads, and thread safe code would have to be put in place to make sure that concurrent access to shared data is safe. Far better is to use some kind of fixed function logic that can execute on the remote processor. This is what we did in PhysX 2.x -- unfortunately the simple group based filtering rules we provided were not flexible enough to cover all applications. In 3.0, we introduce both a shader system, which lets the developer implement an arbitrary system of rules using code that runs on the vector processor (and is therefore not able to access any eventual

game data base in main memory), which is more flexible than 2.x fixed function filtering, but just as efficient, and a totally flexible callback mechanism where the filter shader calls a CPU callback function that is able to access any application data, at the cost of performance -- see `PxSimulationFilterCallback` for details. The best part is that an application can decide on a per-pair basis to make this speed vs. flexibility tradeoff.

Let us look at the shader system first: Here is the filter shader implemented by `SampleSubmarine`:

```
PxFilterFlags SampleSubmarineFilterShader(  
    PxFilterObjectAttributes attributes0, PxFilterData filt  
    PxFilterObjectAttributes attributes1, PxFilterData filt  
    PxPairFlags& pairFlags, const void* constantBlock, PxU3  
{  
    // let triggers through  
    if(PxFilterObjectIsTrigger(attributes0) || PxFilterObjec  
    {  
        pairFlags = PxPairFlag::eTRIGGER_DEFAULT;  
        return PxFilterFlag::eDEFAULT;  
    }  
    // generate contacts for all that were not filtered abo  
    pairFlags = PxPairFlag::eCONTACT_DEFAULT;  
  
    // trigger the contact callback for pairs (A,B) where  
    // the filtermask of A contains the ID of B and vice ve  
    if((filterData0.word0 & filterData1.word1) && (filterDa  
        pairFlags |= PxPairFlag::eNOTIFY_TOUCH_FOUND;  
  
    return PxFilterFlag::eDEFAULT;  
}
```

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

`SampleSubmarineFilterShader()` will be called for all pairs of shapes that

come near each other -- more precisely: for all pairs of shapes whose axis aligned bounding boxes in world space are found to intersect for the first time. All behavior beyond that is determined by what `SampleSubmarineFilterShader()` returns.

The arguments of `SampleSubmarineFilterShader()` include `PxFilterObjectAttributes` and `PxFilterData` for the two objects, and a constant block of memory. Note that the pointers to the two objects are NOT passed, because those pointers refer to the computer's main memory, and that may, as we said, not be available to the shader, so the pointers would not be very useful, as dereferencing them would likely cause a crash. `PxFilterObjectAttributes` and `PxFilterData` are intended to contain all the useful information that one could quickly glean from the pointers. `PxFilterObjectAttributes` are 32 bits of data, that encode the type of object: For example `PxFilterObjectType::eRIGID_STATIC`, `::eRIGID_DYNAMIC`, or even `::ePARTICLE_SYSTEM`. Additionally, it lets you find out if the object is kinematic, or a trigger.

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

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

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

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

Let us look at what the above shader does:

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

This means that if either object is a trigger, then perform default trigger behavior (notify the application about start and end of touch), and otherwise perform 'default' collision 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.word
    pairFlags |= PxPairFlag::eNOTIFY_TOUCH_FOUND;

return PxFilterFlag::eDEFAULT;
```

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

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

```

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

```

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

```

void setupFiltering(PxRigidActor* actor, PxU32 filterGroup, PxU
{
    PxFilterData filterData;
    filterData.word0 = filterGroup; // word0 = own ID
    filterData.word1 = filterMask; // word1 = ID mask to f
    const PxU32 numShapes = actor->getNbShapes();
    PxShape** shapes = (PxShape**)SAMPLE_ALLOC(sizeof(PxSha
    actor->getShapes(shapes, numShapes);
    for(PxU32 i = 0; i < numShapes; i++)
    {
        PxShape* shape = shapes[i];
        shape->setSimulationFilterData(filterData);
    }
    SAMPLE_FREE(shapes);
}

```

This sets up the `PxFilterData`s of each shape belonging to the passed actor. Here are some examples how this is used in `SampleSubmarine`:

```

setupFiltering(mSubmarineActor, FilterGroup::eSUBMARINE, Filter
setupFiltering(link, FilterGroup::eMINE_LINK, FilterGroup::eSUB
setupFiltering(mineHead, FilterGroup::eMINE_HEAD, FilterGroup::

```

```
setupFiltering(heightField, FilterGroup::eHEIGHTFIELD, FilterGr  
setupFiltering(mCrabBody, FilterGroup::eCRAB, FilterGroup::eHEI
```

This scheme is probably too simplistic to use in a real game, but it shows the basic usage of the filter shader, and it will ensure that `SampleSubmarine::onContact()` is called for all interesting pairs.

An alternative group based filtering mechanism is provided with source in the extensions function `PxDefaultSimulationFilterShader`. And, again, if this shader based system is too inflexible, consider using the callback approach provided with `PxSimulationFilterCallback`.

Contact Modification

Sometimes users would like to have special contact behavior. For example to implement sticky contacts, give objects the appearance of floating or swimming inside each other, or making objects go through apparent holes in walls. A simple approach to achieve such effects is to let the user change the properties of contacts after they have been generated by collision detection, but before the contact solver. Because both of these steps occur within the scene `simulate()` function, a callback must be used.

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

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

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

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

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

Basically, every pair of shapes comes with an array of contact points, that have a number of properties that can be modified, such as position, contact

normal, and separation. For the time being, friction properties of the contacts cannot be modified. See `PxContactPoint` and `PxContactPointAux` for properties that can be modified.

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

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

Active Transforms

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

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

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

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

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

// update each render object with the new transform
for (PxU32 i=0; i < nbActiveTransforms; ++i)
{
    MyRenderObject* renderObject = static_cast<MyRenderObje
    renderObject->setTransform(activeTransforms[i].actor2Wo
}
```

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

Introduction

An aggregate is a collection of actors. Aggregates do not provide extra simulation or query features, but allow you to tell the SDK that a set of actors will be clustered together, which in turn allows the SDK to optimize its spatial data operations. A typical use case is a ragdoll, made of N different body parts, with each part a PhysX actor. Without aggregates, this gives rise to N broad-phase entries for the ragdoll. It is typically more efficient to represent the rag doll in the broad phase as a single entity, and do internal collisions in a second pass if necessary.

Creating an Aggregate

Create an aggregate from the *PxPhysics* object:

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

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

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

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

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

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

Populating an Aggregate

Adds an actor to an aggregate as follows:

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

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

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

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

Similarly, to remove the aggregate from the scene:

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

Releasing an Aggregate

To release an aggregate:

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

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

Amortizing Insertion

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

Serialization is the process by which a collection of PhysX objects is stored in a persistent form outside the PhysX runtime, such as on disk. Deserialization is the reverse process, i.e. the loading of those objects into another instance of the PhysX runtime. PhysX 3 features two serialization APIs:

- API-level serialization to RepX, a versioned XML data format.
- Binary serialization, which serializes objects into a block of memory from which the PhysX runtime can later load them without allocation or copying.

Both serialization systems use their own meta data representation of PhysX data structures. The meta data used by RepX captures the PhysX objects at the API-level. It allows forward conversions of serialized data from earlier PhysX 3 versions to later ones. The binary serialization meta data captures the internal data structures of PhysX objects and can be used to convert binary representations between different platforms.

Note: *cooking* also generates a binary output stream. However the primary purpose of cooking is to translate from a user format to a format suitable for the SDK runtime, and so it is not considered a serialization mechanism. Loading a cooked mesh from a stream involves allocation and endian conversion, so is much less efficient than PhysX' binary serialization mechanism. See [Shapes and Geometries](#) for more details about cooking.

Binary Serialization

Binary Serialization allows the creation of memory blocks from which PhysX can later construct objects. The PhysX runtime constructs the objects in place, making this an efficient mechanism for loading objects. You may instance collections of objects simply by making multiple copies of a memory block and deserializing them.

The data is specific to a platform and SDK version. When exported from the runtime it is always targeted at the platform on which it was created, although PhysX can retarget it at another platform in a post-processing step. This allows the conversion of binary assets from authoring platforms (Windows, MacOs and Linux) to other platforms.

Framework Classes

- *PxSerializable* is the base class for the objects that can be serialized.
- *PxCollection* is a collection of *PxSerializable* objects.
- *PxSerialObjectRef* is a 64 bit type, which is used as a reference to a serialized object.
- *PxUserReferences* is a map from object references to serializable objects.

Serializing Objects

The simplest scenario is serializing a complete object graph (for example, an actor, its shapes, and the materials and meshes they reference.)

To serialize objects, add them to a collection:

```
PxRigidBody* dynamic = PxCreateDynamic(...);  
  
...  
  
PxCollection* collection = physics->createCollection();
```

```
dynamic->collectForExport(*collection);
...
material->collectForExport(*collection);
...
mesh->collectForExport(*collection);
```

In general, you need to manually add to a collection all of the objects which you want serialized. However, for certain objects `collectForExport()` automatically adds other objects to the collection

Rigid Actors	shapes owned by the actor
Articulations	links and joints owned by the articulation
Cloth	the cloth fabric

When all the objects have been added, create an implementation of the `PxOutputStream` interface, then serialize the collection:

```
PxOutputStream& s = ...;
collection->serialize(s);
collection->release();
```

To deserialize, first create a collection, then populate it by deserializing from a memory block:

```
void* memory128 = ...;
PxCollection* collection = physics->createCollection();
collection->deserialize(memory128, NULL, NULL);
```

To add all the objects to the scene and release the collection:

```
physics->addCollection(*collection, scene);
collection->release();
```

Memory Management

Management of memory blocks containing deserialized objects is left to users. It is the user's responsibility to:

- allocate the memory block. Note that it must be properly aligned, to a `PX_SERIAL_FILE_ALIGN` (128) bytes boundary.
- fill the block with serialized data, typically by loading it from disk.
- deallocate the memory block when the objects within have been released by PhysX.

Although the user owns the memory block, the PhysX runtime owns any deserialized objects it contains. Concretely, calling `release()` on an object that was created by deserialization will cause its destructor to run, but will not deallocate its memory. If you deallocate the block before the destructors have run for all the objects it contains, the PhysX runtime will likely crash.

Traversing Collections

You can iterate over a collection, for example to ensure the objects you intend to serialize have all been added by `collectForExport()`. When doing so you can use PhysX' dynamic typing mechanism to classify the objects:

```
PxCollection* collection;
PxU32 size = collection->getNbObjects();
for(PxU32 i=0; i<size; i++)
{
    PxSerializable* object = collection->getObject(i);
    if(!object->is<PxActor>())
        continue;

    switch((PxConcreteType)object->getConcreteType())
    {
    case PxConcreteType::eRIGID_DYNAMIC:
        ...
    }
}
```

Partial Serialization

The above code in ([serializingObjects](#)) serializes complete object graphs. Another common use case is where a collection of actors and joints - say, a rag doll - will be deserialized multiple times, with each instance sharing the same materials and meshes. To achieve this, serialize two collections:

- a collection of the materials and meshes that will be deserialized just once
- a collection of actors and joints which will be copied and deserialized multiple times

The second of these will be a *partial* object graph: there will be some objects which are not serialized with the collection, but to which the collection will contain references.

The application has to take two steps in order to deal with partial object graphs:

1. On serialization: Provide consistent reference identities (*PxSerialObjectRef*) to serializable objects for the referencing and referenced collections.
2. On deserialization: Provide the referencing collections with appropriate information to reestablish the references to the deserialized objects in referenced collections.

PxCollection provides two functions for declaring object references before serializing:

- *PxCollection::setObjectRef* specifies a reference to an object that is in the collection. The reference will be serialized along with the collection and recreated when it is deserialized to support lookup for objects within the collection.
- *PxCollection::addExternalRef* specifies a reference for an object that will not be serialized with the collection, but is referenced by an object

the collection contains. You must provide a lookup for this reference when deserializing the collection.

They are used as follows:

```
PxConvexMesh** convexes;           // An array of mNbConvexes conv
PxRigidDynamic** actors;          // An array of mNbConvexes acto

PxPhysics* physics;               // The physics SDK object
PxOutputStream& convexStream;     // Output stream for the convex
PxOutputStream& actorStream;      // Output stream for the actor

PxCollection* convexCollection = physics->createCollection();
PxCollection* actorCollection = physics->createCollection();

for(PxU32 i=0;i<mNbConvexes;i++)
{
    convexes[i]->collectForExport(*convexCollection);
    convexCollection->setObjectRef(convexes[i], (PxSerialObject
    actorCollection->addExternalRef(convexes[i], (PxSerialObjec
}

// serialize the convexes and the references in their collectio
convexCollection->serialize(convexStream);
convexCollection->release();

// Add actors to collection
for(PxU32 i=0;i<mNbConvexes;i++)
    actors[i]->collectForExport(*actorCollection);
actorCollection->serialize(actorStream);
actorCollection->release();
```

PhysX 3 expects references (*PxSerialObjectRef*) to be unique per collection. The application has to make sure they are consistent across different collections for making partial object graphs deserialize correctly. On deserialization *PxUserReferences* container objects can be used to handle associations between references and deserialized objects. When you deserialize a collection you may supply

- a *PxUserReferences* object to the deserializer to resolve the collection's external references.

- a *PxUserReferences* object which the deserializer populates with the object references that were serialized with the collection.

The two *PxUserReferences* objects are optional arguments to the deserializer, and may be the same. To deserialize the collections:

```
PxPhysics* physics;           // The physics SDK object
PxScene* scene;              // the scene into which the obj
void* convexMemory128;      // aligned memory containing se
void* actorMemory128;       // aligned memory containing se

PxCollection* convexCollection = physics->createCollection();
PxCollection* actorCollection = physics->createCollection();
PxUserReferences* convexRefs = physics->createUserReferences();

// deserialize the convexes, populating convexRefs with the ser
// that where specified with PxCollection::setObjectRef

convexCollection->deserialize(convexMemory128, convexRefs, NULL
physics->addCollection(*convexCollection, scene);
convexCollection->release();

// deserialize the actors, using convexRefs to resolve referenc
// that where specified with PxCollection::addExternalRef

actorCollection->deserialize(actorMemory128, NULL, convexRefs);
physics->addCollection(*actorCollection, scene);
actorCollection->release();

convexRefs->release();
```

If there are references in the collection to objects not contained within it, and they cannot be resolved using the *PxUserReferences* passed at deserialization time, an error occurs and deserialization is aborted.

You can add references to a *PxUserReferences* object manually with *PxUserReferences::setObjectRef(...)*, as well as via deserialization. This may be useful, for example, if you are using a mixture of serialized and procedurally created objects - for example, a predefined material library for your application that is not itself serialized but which serialized objects

must reference.

You can also use *PxUserReferences* to find objects in a collection in order to fix up references with gameplay objects:

```
PxPhysics* physics;           // The physics SDK object
void* memory128;             // aligned memory containing se

PxCollection* collection = physics->createCollection();
PxUserReferences* userRefs = physics->createUserReferences();

// deserialize objects and fill userRefs with objects for which
// PxCollection::setObjectRef was called before serialization
collection->deserialize(memory128, userRefs, NULL);

// receive a list of all deserialized objects which have user r
#define MAX_USER_REFS 100
PxSerialObjectAndRef userRefBuffer[MAX_USER_REFS];
userRefs->getObjectRefs(userRefBuffer, MAX_USER_REFS);

// iterate over the list to path up gameplay objects
for (PxU32 i = 0; i < userRefs->getNbObjectRefs(); i++)
{
    PxActor* actor = userRefBuffer[i].serializable->is<PxActor>
    if (actor)
    {
        // this assumes that findGamePlayObjectFromRef is able
        // the corresponding game play object from a PxSerialOb
        actor->userData = findGamePlayObjectFromRef(userRefBuff
    }
}
```

In order to iterate over user references and external references of a collection the following two methods can be used:

- *PxCollection::getObjectRefs* creates a *PxUserReferences* instance containing all the user references that were set with *PxCollection::setObjectRef(...)*.
- *PxCollection::getExternalRefs* creates a *PxUserReferences* instance containing all the external references that were added with *PxCollection::addExternalRef(...)*.

Note that both methods don't provide the corresponding references for deserialized collections. Serialized user references can only be obtained by the first *PxUserReferences* argument to *PxCollection::deserialize(...)*. The set of external references that are needed to deserialize a collection is expected to be managed by the application. There is currently no support to query the external references needed to deserialize a collection.

Serializing Everything

PhysX provides two utility functions for serializing the entirety of the PhysX runtime: *PxCollectForExportSDK* and *PxCollectForExportScene*:

```
PxPhysics* physics;    // The physics SDK object
PxScene* scene;       // The physics scene
PxOutputStream& s;    // The user-defined stream doing the act

// 1) create a collection
PxCollection* collection = physics->createCollection();

// 2) collect objects to serialize
PxCollectForExportSDK(*physics, *collection);    // Collects all
PxCollectForExportScene(*scene, *collection);    // Collects all

// 3) serialize collection and release it
collection->serialize(s);
collection->release();
```

Deserialization is as previously:

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

void* memory128 = ...; // a 128-byte aligned buffer previous
PxCollection* collection = physics->createCollection();
collection->deserialize(memory128, NULL, NULL);
physics->addCollection(*collection, scene);
collection->release();
```

Object Names

Some objects, such as shapes and actors, can be given names using the *PxShape::setName()* and *PxActor::setName()* functions. The SDK does not own those names, i.e. the strings remain in user memory. When serializing objects, you may choose whether to preserve those names in the serialized data, or to discard them to create smaller data. If you choose to preserve the names, they will be serialized along with the objects themselves when calling *PxCollection::serialize()*. On deserialization, the names will live within the user-provided memory block.

Use the 'exportNames' parameter of the *PxCollection::serialize()* function to control this behavior.

Retargeting

Binary serialized data is platform-specific, and when serialized it always targets the platform on which it was created. The binary converter in the cooking library retargets data from one platform to another. So to deploy data for PS3, XBox etc, typically you will serialize on PC, then use the converter to retarget for each platform.

The converter requires meta-data for the source and target platforms, which contains information about the binary layout of objects for that platform. To obtain metadata, use the function provided in the extensions library for each platform:

```
void PxDumpMetaData(PxOutputStream& stream, const PxPhysics& ph
```

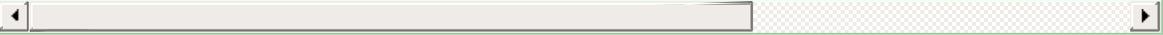
On each target platform, run it once and keep generated data around.

Assuming you have initialized the cooking library, convert data as follows:

```
PxErrorCallback* myErrorCallback; // an error callback im  
PxInputStream& srcMetadata; // metadata for the 'fr
```

```
PxInputStream& dstMetadata;           // metadata for the 'to
PxInputStream& srcAsset;              // stream containing so
PxU32 srcAssetSize;                 // size of the source a
PxOutputStream& dstAsset;           // output stream for re

PxBinaryConverter* converter = cooking->createBinaryConverter(m
converter->setMetaData(srcMetadata, dstMetadata);
converter->convert(srcAsset, srcAssetSize, dstAsset);
```



API-level Serialization with RepX

RepX stands for Representation X and is the XML serialization format for PhysX 3. This format is intended to be a user-level format meaning the data is in a format that matches the public API. It is also intended to be backwards compatible, thus assets saved in RepX for version 3.0 of PhysX will load in version 3.1, 3.2, etc. We do not intend for RepX to be used in performance critical or memory constrained situations.

RepX itself consist of a core module with a set of extensions. Extensions are responsible for transforming objects coming from the outside world (called 'live' objects) into a sort of key-value pair format. The library takes care of serializing/deserializing this format. A RepX collection is a set of objects that are transformed into the key-value pair format but still held in memory.

RepX collections may depend on objects in other collections in order to be fully realized into live objects. To facilitate this, RepX, similar to the binary serialization system, has an Id assignment system where 64 bit identifiers are assigned to objects. This happens when a live object is added to a collection and the id defaults to the memory address of the object. This design was intended so that one would serialize various buffers (convex mesh, triangle mesh, height field, etc.) into one collection file, and then using the same id map serialize a set of scene objects into another RepX file. Then the user could deserialize the buffer collection using the original ids once, but deserialize the scene RepX collection multiple times requesting RepX to generate new ids upon deserialization into live objects.

To use RepX, there are two headers you need. The first is RepX.h and this is absolutely required as it describes the base types. The second is RepXUtility.h and this makes using RepX *much* easier, but requires the PhysX extensions to be loaded to work. Even if you do not intend to use RepXUtility.h we recommend you use it for examples on how to do things like:

1. Create a collection with the various extensions loaded.
2. Copy objects into the collection.
3. Instantiate a collection into a scene.
4. Convert a RepX collection into a binary collection.
5. Upgrade a RepX collection from a past version to the current version (requires RepXUpgrader project).

Here is an example:

```
RepXCollection* theCollection = createCollection(physics.getTol  
RepXIdToRepXObjectMap* theIdMap = RepXIdToRepXObjectMap::creat  
addToRepXCollectionNF( theCollection, theIdMap, thePhysicsObjec  
addObjectsToScene(theCollection, physics, cooking, scene, mStri  
theCollection->destroy();  
theIdMap->destroy();
```

A RepX (Representation X) collection is PhysX's forward-compatible storage format. Using RepX you can store PhysX assets in a way that we guarantee support for them in future versions of PhysX. RepX also includes facilities for instantiating a set of assets multiple times into a scene and for of course upgrading a RepX collection from an older version to a newer version as well as converting a RepX collection to a PhysX binary collection. Currently RepX's storage format is ASCII-XML.

RepX.h contains the base collection definitions and RepXUtility.h contains functionality that cannot be included in RepX because it relies on PhysXExtensions. Users who have PhysXExtensions compiled into their SDK should use RepXUtility wherever possible and users who do not should still look to the RepXUtility header for examples on how to do the base operations.

RepX is a base key-value data store that is specialized towards the various PhysX datatypes via extensions. An extension needs to provide the capability to go from a 'live' object to a RepX data store value and back. Extensions are created and registered when the collection is created and destroyed when the collection is destroyed. So, to extend RepX to store a

different datatype, be it a custom joint or specific game information users will need to implement a RepX extension.

RepX identifies live types via a 'fat pointer' combination of a void* and a const char* type name. This name is used to link the void* pointer to the appropriate extension necessary to serialize the pointer. RepX types also have a user-supplied id that is used to link dependent types to other objects in the collection. This id defaults to the memory address of the object if not supplied. All of the world-to-RepX functions along with the RepX-to-world functions take an id map. One constraint that RepX has is that a base object needs to be added to the collection before any dependent object. Thus PxConvexMesh needs to be added before the PxConvexMeshGeometry that refers to it.

To create a RepX collection, we have provided a few createCollection functions in RepXUtility.h. These functions create all known extensions for both the core RepX types and types that rely on PhysXExtensions.h and then create a new collection. The various overloads are for passing in custom allocators and for creating a collection and immediately deserializing a data source back to a RepX collection.

Saving to RepX involves making a decision about whether to save the PxPhysics objects (which I will later refer to as 'buffers') separate from the scene objects. Saving them separate adds complexity but gives you the option of instantiating the scene objects, perhaps with different global transforms several times into the PhysX scene.

In any case, we provide three functions in RepXUtility.h which allow various combinations of use cases. These functions build the RepX collection using the object's base address as its id.

1. addSDKItemsToRepX - add all the buffers present in the PxPhysics object (PxConvexMesh, PxHeightField, PxTriangleMesh and PxMaterial) to the RepX collection.
2. addSceneItemsToRepX - add all the objects in a given PxScene

(PxRigidDynamic, PxRigidStatic, PxArticulation, any PxJoints, etc.), to a RepX collection.

3. `addItemToRepX` - calls `addSDKItemsToRepX` followed by `addSceneItemsToRepX`.

When RepX instantiates an object it asks the appropriate extension to create a live object. This is achieved through use of the given key-value data object. RepX then calls a callback passed into the instantiation function with the newly created object. It is then the responsibility of the callback to add the object to a PxScene if desired. We have wrapped up the common case of this operation with a function `addObjectsToScene`.

For the use case where you want to instantiate buffer and scene RepXCollections separately, you would need to call a more low level function (also available in `RepXUtility.h`), `instantiateCollection` for the scene objects with a flag, `inAddOriginalIdsToObjectMap` set to false. This tells RepX to just use the newly generated object's address as its id. You could call this several times with the same scene RepX collection safely but you need to share the id map between all instantiations so that RepX can find the buffers instantiated with `inAddOriginalIdsToObjectMap` set to true.

Upgrading a RepX collection from an older version to a newer one is easy. You need to link with the `RepXUpgrader` static library and call the appropriate `physx::repx::RepXUpgrader::upgradeCollection` function.

Implementing your own custom extension is a bit more involved. You need to assign an ascii name to your extension and implement the `RepXExtension` interface. You will then need to add that extension to the rest of the extensions when a RepX collection is created. When your target objects are added to the `RepXCollection`, they will need to be tagged with your extension's ascii name.

Converting between RepX and Binary Serializable Collections

PhysX provides a couple of convenience functions in RepXUtility.h to ease the conversion between binary and RepX data.

- *addObjectsToPxCollection(...)* takes a *RepXCollection* and instantiates the objects contained in the physics SDK. In the process it creates two *PxCollection* instances, one for *PxPhysics* serializables and one for *PxScene* serializables. Optionally a *PxUserReferences* can be provided that is populated with RepX ids of *PxPhysics* serializables. Note that unlike with *PxCollection::deserialize(...)* the resulting *PxCollection* instances will also have the deserialized ids available with *PxCollection::getObjectRefs(...)*. Additionally, the *PxCollection* containing the *PxScene* serializables will have the external references added, that can be queried with *PxCollection::getExternalRefs*.
- *deserializeFromRepX(...)* does the same as *addObjectsToPxCollection(...)* but takes a repX stream as input.
- *pxCollectionToRepXCollection(...)* takes a *PxCollection* and returns a *RepXCollection* on success. The in/out parameter *inAnonymousNameStart* serves as the first value used for 64 bit RepX Ids assigned to the resulting RepX objects and is incremented as needed.
- *serializeToRepX(...)* does the same as *pxCollectionToRepXCollection(...)* but additionally serializes the collection to an output stream in RepX format.

Example for serializing a *PxCollection* to a RepX stream:

```
PxCollection* collection = physics.createCollection();
PxCollectForExportSDK(physics, *collection); //collect buffer o
PxCollectForExportScene(scene, *collection); //collect scene le

PxDefaultFileOutputStream outputStream(pathToRepXFile);
PxU64 start = 0x80000000; //start of reference ids for exported
serializeToRepX(outputStream, collection, start);

collection->release();
```

Example for deserializing a PxCollection from a RepX stream:

```
PxDefaultFileInputData data(pathToRepXFile);

PxCollection* bufferCollection = physics.createCollection();
PxCollection* sceneCollection = physics.createCollection();
PxStringTable* stringTable = NULL; //we are not interested in o
PxUserReferences* externalRefs = NULL; //we assume there are no
PxUserReferences* userRefs = NULL; //would be used to receive r

deserializeFromRepX(data, physics, cooking, stringTable, extern
physics.addCollection(*sceneCollection, scene); //add the scene

bufferCollection->release();
sceneCollection->release();
```

Example for upgrading a RepX stream:

```
PxDefaultFileInputData data(pathTo30RepXFile); //load an older
RepXCollection* collection = createCollection(data, foundation.
RepXCollection* upgraded = &RepXUpgrader::upgradeCollection(*co
PxDefaultFileOutputStream outputStream(pathToNewRepXFile);
upgraded->save(outputStream); //save the result to file
```


Interface

In this chapter we will have a quick look at the statistics information that PhysX collects every simulation step. Usually, this information can be explored in the PhysX Visual Debugger but we do offer a PhysX API method as well to allow applications to access the data directly. After a simulation step and a call to *PxScene::fetchResults()*, the simulation statistics for the processed step can be retrieved through the *PxScene::getSimulationStatistics()* interface. The method copies the data to a user provided *PxSimulationStatistics* structure. For details about the individual members please refer to the API documentation.

Note: Do not fetch the simulation statistics while the simulation is running.

Usage

The provided simulation statistics is mainly meant to help investigate performance issues. It provides a quantitative summary of the work done, i.e., the number of objects or combination of objects which have been processed in the current simulation step. For example, if you encounter performance spikes in certain frames, then the simulation statistics might give some insight into possible causes. For instance:

- Has a large amount of volumes been added or removed from the broadphase in one single step? You could try to distribute the addition/removal of objects over a couple of simulation steps or maybe there is a particle system in the scene whose grid size is very small.
- Are there suddenly many more collision pairs processed than expected? This could be caused by a badly configured collision pair filter or maybe some *PxPairFlags* have been accidentally raised.
- etc.

Please keep in mind that the simulation statistics are currently less a measurement of what the scene contains but rather what got processed. So it is only partially helpful to detect whether objects have been configured and arranged properly.

Continuous Collision Detection

When continuous collision detection (or CCD) is turned on, the affected rigid bodies will not go through other objects at high velocities (a problem also known as tunneling). To 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_SWEPT_INTEGRATION;
...
```

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::eRESOLVE_CONTACTS;
    pairFlags |= PxPairFlag::eSWEPT_INTEGRATION_LINEAR;
    return PxFilterFlags();
}

...

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

3. Swept bounds need to be enabled for each shape that requires CCD:

```
PxShape* shape;
...
shape->setFlag(PxShapeFlag::eUSE_SWEPT_BOUNDS, true);
```


Articulations

An articulation is a single actor comprising a set of links (each of which behaves like a rigid body) connected together with special joints. Every articulation has a tree-like structure - so there can be no loops or breaks.

Articulations are a somewhat experimental feature of the SDK, under active research. Currently their primary use is modeling physically actuated characters. They support higher mass ratios, more accurate drive models, have better dynamic stability and a more robust recovery from joint separation than standard PhysX joints. However, they are considerably more expensive - the CPU budget on XBox360 for the collision and dynamics of a 20-link articulation is in the region of 0.5 CPU-milliseconds.

Although articulations do not directly build on joints, they use very similar configuration mechanisms. We assume in this section that you are already familiar with using 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 the parent of the initial link), and the pose of the new link:

```
PxsArticulationLink* link = articulation->createLink(parent, li  
link->createShape(linkGeometry, material);  
PxRigidBodyExt::updateMassAndInertia(*link, 1.0f);
```

Articulation links have a restricted subset of the functionality of rigid bodies. They may not be kinematic, and they do not support damping, velocity clamping, or contact force thresholds. Sleep state and solver

iteration counts are properties of the entire articulation rather than the individual links.

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

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

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

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

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

```
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 `contactDistance` value that the solver will start to enforce the limit before the limit threshold is exceeded.

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

Driving an Articulation

Articulations are driven through joint acceleration springs. You can set a position target, a velocity target, and spring and damping parameters that control how strongly the joint drives towards the target. You can also set compliance values, indicating how strongly a joint resists acceleration. A compliance near zero indicates very strong resistance, and a compliance of 1 indicates no resistance.

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

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

PxArticulation class.

Articulation Projection

When any of the joints in an articulation separate beyond a specified threshold, the articulation is projected back together automatically. Projection is an iterative process, and the PxArticulation attributes *separationThreshold* and *projectionIterations* control when projection occurs and trade cost for robustness.

Substepping

You may want the simulation frequency of physx to be higher than the frame rate of your application, to allow for higher fidelity simulation or better stability. The simplest way to do this is just to call `simulate()` and `fetchResults()` multiple times:

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

The code in `Samples/SampleBase/SampleStepper.cpp` in the sample framework demonstrates a different approach using completion tasks.

Using Completion Tasks

If you submit a completion task to the scene in the `simulate()` call, the simulation will decrement its reference count when simulation completes, which (assuming there are no outstanding references) will cause the task to run. The completion task first calls `fetchResults` and performs any per-substep work:

```
mScene->fetchResults(true);
mSample->onSubstep(mSubStepSize);
```

Since a task may not submit itself as a completion to `simulate()`, the completion tasks are double buffered. To start another simulation step, the completion task registers the *other* task with the task manager (which also sets the reference count to 1), calls `simulate()` again if necessary:

```
StepperTask &s = ownerTask == &mCompletion0 ? mCompletion1 : mC
```

```
s.setContinuation(*mScene->getTaskManager(), NULL);  
mScene->simulate(mSubStepSize, &s);
```

Finally, it releases the reference which prevents the new completion task from executing:

```
s.removeReference();
```

Synchronizing with Other Threads

An important consideration for substepping is that `simulate()` and `fetchResults()` are classed as write calls on the scene, and it is therefore illegal to read from or write to a scene while those functions are running. PhysX does not lock its scene graph, but it will report an error in the checked build if it detects that multiple threads make concurrent calls to the same scene, unless they are all read calls.

To synchronize with the rendering thread the sample stepper holds an extra reference to the first completion task until the `renderDone()` method is invoked, so that the renderer can safely read the scene in parallel with simulation. On completion of all substeps, the stepper signals a synchronization object which may be checked with a `wait()` method.

Custom Constraints

Constraint is a more general term for joints. Constraints use shaders for the same reason as contact filtering: There is a requirement to inject performance sensitive custom code into the SDK. While joints were native objects of the PhysX 2.x API, PhysX 3.0 only supports a fully customizable constraint object in the core API, and all 2.x joint types are implemented using this mechanism as extensions. Let us take a short look at how this works. Once the reader understands, he will be in a good position to create his own joint types. You should read the chapter on joints before you try to understand their workings, however.

When you call `PxJointCreate()`, the extensions library first fills out a `PxConstraintDesc` object, which is a bunch of parameters for constraint creation. Here is the code for a spherical joint:

```
PxConstraintDesc nxDesc;
nxDesc.actor[0]           = desc.actor[0];
nxDesc.actor[1]           = desc.actor[1];
nxDesc.flags              = desc.constraintFlags;
nxDesc.linearBreakImpulse = desc.breakForce;
nxDesc.angularBreakImpulse = desc.breakTorque;

nxDesc.solverPrep         = SphericalJointSolverP
nxDesc.project            = SphericalJointProject
nxDesc.visualize         = SphericalJointVisuali

nxDesc.dataSize          = sizeof(SphericalJoint
nxDesc.connector         = joint->getConnector()
```

The first few settings are self explanatory ... like the actors to connect, when the joint should break, and so on. The next three are three callback functions -- user defined shaders. (See the section on filter shaders to find out what shaders are, and the rules that apply to them.) They contain the code that mathematically defines the behavior of the joint. Every time the joint needs to be solved, the simulation will call these functions.

Finally, the 'connector' is a class of additional user defined joint specific functionality that are not called from the solver directly, and are not shaders.

Lastly, the filled out descriptor is used to create the constraint object:

```
PxConstraint* constraint = physics.createConstraint(nxDesc);
```


Introduction

PhysX SDK 3.2 provides two types of collision detection:

1. Default collision detection

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

2. Persistent Contact Manifold (PCM)

PCM is a fully distance-based collision detection system, which uses the Gilbert-Johnson-Keerthi algorithm (GJK) and the Expanding Polytope Algorithm (EPA) to calculate one contact at the point of deepest penetration. It then uses an incremental method to build up a contact manifold by recycling contacts from past frames, accumulating up to four contacts per frame. This approach is quite efficient in terms of performance and memory. However, it can reduce stacking stability when simulating with large timesteps, tall stacks and small objects. As this approach is distance-based, it will generate the correct contact points for arbitrary contact offsets/rest offsets.

Note that PCM is under active development, and is not yet as mature as the default collision system.

Enabling the Persistent Contact Manifold

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

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


Introduction

PhysX SDK 3.2 provides two different types of friction models:

1. Default friction model

The default friction model is constraint-based friction model. It is efficient and leads to very stable friction behavior even at relatively low solver iterations. However, as it is constraint-based, this approach can lead to very strong static friction, which can diverge from the analytical results expected for a given friction coefficient. This makes it ideal for simulations in which accuracy can be traded off for performance; a situation common in games.

2. Coulomb friction model

The Coulomb friction model is based on the Coulomb laws of friction. It is less efficient than the default friction model and it can require more solver iterations to achieve stable stacking behavior. However, its friction behaviors are much closer to the analytical results than the default friction model so could be preferable for applications where accurate friction behavior is important. The Coulomb model comes in two variants, one-directional and two-directional. The one-directional approach is similar in terms of workload to the default friction model and can be stable at similar solver iteration counts. Its behavior is closer to the analytical results but diverges slightly. The one-directional approach is a simplification of the full Coulomb model, in which the friction for a given point of contact is reduced to a one-directional friction model. This simplification allows us to reduce the number of iterations required for convergence but is not as accurate as the full two-directional model. Therefore, in order to simulate stable stacking, we need to increase the number of solver iterations in the two-directional model.

Enabling The Coulomb Friction Model

To enable the Coulomb friction model, set one of the two flags in the `PxSceneDesc::flags`:

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

or:

```
sceneDesc.flags |= PxSceneFlag::eENABLE_TWO_DIRECTIONAL_FRICTIO
```


Introduction

PhysX 3 offers two particle system types - a generic particle system and an SPH fluid particle system. The generic particle system provides basic particle motion and collision with rigid actors. It can be used for objects that require collisions against the environment, but for which inter-particle interactions are not needed. Examples include small debris, sparks or leaves. The SPH fluid particle system can be used for fluid effects that require approximate incompressibility and flowing behavior, such as liquids or fog and smoke filling up a volume.

PhysX 3 takes care of collision detection and particle dynamics, while auxiliary facilities such as emitters, lifetime maintenance etc. need to be provided by the application.

SampleParticles shows both particle system types being used: *PxParticleSystem* is used for small debris and smoke, while *PxParticleFluid* is used for the waterfall.

Creating Particle Systems

Sample Reference:

```
PxParticleSystem* SampleParticles::createParticleSystem(...)  
PxParticleFluid* SampleParticles::createFluid(...)
```

Both particle system classes *PxParticleSystem* and *PxParticleFluid* inherit from *PxParticleBase*, which is the common interface providing particle manipulation and collision functionality. Particle systems inherit from *PxActor* and can be added to a scene.

The following section shows how a particle system is created and added:

```
// set immutable properties.  
PxU32 maxParticles = 100;  
bool perParticleRestOffset = false;  
  
// create particle system in PhysX SDK  
PxParticleSystem* ps = mPhysics->createParticleSystem(maxPartic  
  
// add particle system to scene, in case creation was successfu  
if (ps)  
    mScene->addActor(*ps);
```

Particle fluids can be created in a similar fashion.

There are three types of particle system properties. Some need to be specified when the particle system is created and can't be changed afterwards. Some are mutable while the particle system is not part of a scene. Others can be changed at any time.

Immutable properties of *PxParticleBase* that need to be specified at creation:

- *maxParticles*: The maximum number of particles that can be added to a particle system. The smaller the value, the smaller the memory

footprint of the particle system is going to be.

- *particleBaseFlags*,
PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET:
Enables/disables per-particle rest offsets. Memory can be saved by turning per particle rest offsets off.

Properties of *PxParticleBase* which are immutable when the particle system is part of a scene:

- *maxMotionDistance*: The maximum distance a particle can travel during one simulation step. High values may hurt performance, while low values may restrict the particle velocity too much.
- *gridSize*: A hint for the PhysX SDK to choose the particle grouping granularity for proximity tests and parallelization. See [particleGrid](#).
- *restOffset*: Defines the minimum distance between particles and the surface of rigid actors that is maintained by the collision system.
- *contactOffset*: Defines the distance at which contacts between particles and rigid actors are created. The contacts are internally used to avoid jitter and sticking. It needs to be larger than *restOffset*.
- *particleReadDataFlags*: Specifies a subset of simulation properties which are returned to the application after simulation. See [readingParticles](#).
- *particleBaseFlags*, *PxParticleBaseFlag::eGPU*: Enable/disable GPU acceleration.
- *particleBaseFlags*, *PxParticleBaseFlag::eCOLLISION_TWOWAY*: Enable/disable two-way interaction between rigid bodies and particles.

Properties of *PxParticleFluid* which are immutable when the particle system is part of a scene:

- *restParticleDistance*: Defines the resolution of the particle fluid.

Mutable properties of *PxParticleBase*:

- *restitution*: Restitution used for particle collision.
- *dynamicFriction*: Dynamic friction used for particle collision.

- *staticFriction*: Static friction used for particle collision.
- *damping*: Velocity damping constant, which is globally applied to each particle.
- *externalAcceleration*: Acceleration applied to each particle at each time step. The scene gravity which is added to the external acceleration by default can be disabled using *PxActorFlag::eDISABLE_GRAVITY*.
- *particleBaseFlags*, *PxParticleBaseFlag::eENABLED*: Enables/disables particle simulation.
- *particleBaseFlags*, *PxParticleBaseFlag::ePROJECT_TO_PLANE*: Enables/disables projection mode which confines particles to a plane.
- *projectionPlaneNormal*, *projectionPlaneDistance*: Defines plane for the projection mode.
- *particleMass*: Mass used for two way interaction with rigid bodies.
- *simulationFilterData*: Filter data used to filter collisions between particles and rigid bodies. See [collisionFiltering](#).

Mutable properties of *PxParticleFluid*:

- *stiffness*: The stiffness (or gas constant) influences the calculation of the pressure force field. Low values of stiffness make the fluid more compressible (i.e., springy), while high values make it less compressible. The stiffness value has a significant impact on the numerical stability of the simulation; setting very high values will result in instability. Reasonable values are usually between 1 and 200.
- *viscosity*: Viscosity controls a fluid's thickness. For example, a fluid with a high viscosity will behave like treacle, while a fluid with low viscosity will be more like water. The viscosity value scales the force to reduce the relative velocity of particles within the fluid. Reasonable values are usually between 5 and 300.

Creating Particles

Sample Reference:

```
void ParticleSystem::createParticles(...)
```

PhysX 3 itself has no built-in emitters. Instead, it simply provides an interface to create particles with initial properties. Specifying particle indices and positions is mandatory, while velocities and rest offsets may be specified optionally. In order to provide per-particle rest offsets *PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET* needs to be set. The rest offsets are not allowed to be larger than *PxParticleBase.getRestOffset()*. Per-particle flags can be provided but do not serve any purpose in PhysX 3, since all particle flags are read only.

Particles in PhysX 3 can be accessed with constant array indices throughout their lifetime. The application specifies an index for each particle on creation. Usually the application maintains its own representation of particles which already have associated indices that can be reused for PhysX. If the application does not have appropriate indices at its disposal, it can use an index pool provided by the PhysX extensions library *PxParticleExt::IndexPool* as explained here: [indexPool](#).

Note: In PhysX 3 all particle access such as creating, releasing, updating and reading particles can only be carried out while the simulation of the scene is not being executed.

Example for creating a few particles:

```
// declare particle descriptor for creating new particles
// based on numNewAppParticles count and newAppParticleIndices,
PxParticleCreationData particleCreationData;
particleCreationData.numParticles = numNewAppParticles;
particleCreationData.indexBuffer = PxStrideIterator<const PxU32
particleCreationData.positionBuffer = PxStrideIterator<const Px
```

```
// create particles in *PxParticleSystem* ps
bool success = ps->createParticles(particleCreationData);
```

The indices specified for particle creation need to be unique and within the limit of *PxParticleBase::getMaxParticles()*.

Note: For fluid particles it is necessary to spawn particles at distances close to *PxParticleFluid::getRestParticleDistance()* in order to achieve a regular emission, otherwise particles will spread immediately in all directions. The sample implements two types of emitters - a constant rate emitter, and a constant pressure emitter, which is suitable for fluid particles.

Releasing Particles

Sample reference:

```
void ParticleSystem::update(...)
```

Particles can be released by providing indices to the particle system. As opposed to older versions of the PhysX SDK, particles get immediately released.

Example for releasing a few particles:

```
// declare strided iterator for providing array of indices corresponding to
// particles that should be removed
PxStrideIterator<const PxU32> indexBuffer(appParticleIndices);

// release particles in *PxParticleSystem* ps
ps->releaseParticles(numAppParticleIndices, indexBuffer);
```

It is a requirement that the indices passed to the release method are unique and correspond to existing particles.

All particles can be released at once by calling:

```
ps->releaseParticles();
```

Since only a limited number of particle slots (*PxParticleBase::getMaxParticles()*) are available it might be appropriate to replace old particles with new ones. This can be achieved for instance by maintaining an application-side particle lifetime. There are other reasons to release particles:

- Drains can be useful to remove particles that go to locations where they are not needed anymore. See [particleDrains](#).
- The spatial data structure used for particles may overflow. Particles

that cannot be covered are marked and should be released. See [particleGrid](#).

Index Pool Extension

Example for allocating particle indices using the PhysX extensions library:

```
// create an index pool for a particle system with maximum part
PxParticleExt::IndexPool* indexPool = PxParticleExt::createInde

// use the indexPool for allocating numNewAppParticles indices
// for particle creation throughout the particle system lifetim
// is smaller than numNewAppParticles, the maxParticles limit w
PxU32 numAllocated = indexPool->allocateIndices(numNewAppPartic

// in order to reuse particle slots, the indices should be hand
// indexPool after the particles have been released
indexPool->freeIndices(numAppParticleIndices, PxStrideIterator<

// if no further index management is needed, the pool should be
indexPool->release();
```



Updating Particles

The following per-particle updates are carried out immediately:

- Position updates: Teleporting particles from one location to another.
- Velocity updates: Directly altering the velocities of particles.
- Rest offset updates: Changes particle rest offsets (only available with *PxParticleBaseFlag::ePER_PARTICLE_REST_OFFSET*).

Particle updates that are carried out during the next scene simulation step:

- Force updates: Results in a velocity change update according to a vector unit specified by *PxForceMode*.

Example for force update:

```
// specify strided iterator to provide update forces
PxStrideIterator<const PxVec3> forceBuffer(appParticleForces);

// specify strided iterator to provide indices of particles that
PxStrideIterator<const PxU32> indexBuffer(appParticleForceIndices);

// specify force update on PxParticleSystem ps choosing the "force"
ps->addForces(numAppParticleForces, indexBuffer, forceBuffer, P
```

Reading Particles

Sample reference:

```
void ParticleSystem::update(...)
```

The PhysX SDK does not provide to the user all simulated per-particle properties of a particle system by default. The application can specify the data it needs by configuring `PxParticleBase::particleReadDataFlags`:

- `PxParticleReadDataFlag::ePOSITION_BUFFER`: On by default.
- `PxParticleReadDataFlag::eFLAGS_BUFFER`: On by default.
- `PxParticleReadDataFlag::eVELOCITY_BUFFER`: Off by default.
- `PxParticleReadDataFlag::eREST_OFFSET_BUFFER`: Off by default.
- `PxParticleReadDataFlag::eCOLLISION_NORMAL_BUFFER`: Off by default.
- `PxParticleReadDataFlag::eDENSITY_BUFFER`: Only available for particle fluids and off by default.

Particle flags provide more information on individual particles:

- `PxParticleFlag::eVALID`: If set, the particle was created beforehand and not yet released. If not set, the particle slot does not contain a valid particle. All other properties are invalid in this case and should be ignored.
- `PxParticleFlag::eCOLLISION_WITH_STATIC`: Shows whether a particle collided with a rigid static during the last simulation step.
- `PxParticleFlag::eCOLLISION_WITH_DYNAMIC`: Shows whether a particle collided with a dynamic rigid body during the last simulation step.
- `PxParticleFlag::eCOLLISION_WITH_DRAIN`: Shows whether a particle collided with a rigid actor shape that was marked as a drain (`particleDrains`).
- `PxParticleFlag::eSPATIAL_DATA_STRUCTURE_OVERFLOW`:

Shows whether a particle had to be omitted when building the SDK internal spatial data structure (`particleGrid`).

Particle collision normals represent contact normals between particles and rigid actor surfaces. A non-colliding particle has a zero collision normal. Collision normals are useful e.g. for orienting the particle visualization according to their contact with rigid actors.

Particle densities provided by particle fluids can be used for rendering. A particle density has a value of zero for a particle that is completely isolated. It has a value of one for a particle that has a particle neighborhood with a mean spacing corresponding to `PxParticleFluid::getRestParticleDistance()`.

Particle data can only be read while the scene simulation is not executing. In order to get access to the SDK buffers a `PxParticleReadData` instance needs to be acquired from the SDK. It has the following properties:

- `numValidParticles`: Total number of valid particles for the corresponding particle system.
- `validParticleRange`: The index range of valid particles in the particle buffers.
- `validParticleBitmap`: Bitmap of valid particle locations.
- `positionBuffer`, `positionBuffer`, `velocityBuffer`, `restOffsetBuffer`, `flagsBuffer`, `collisionNormalBuffer`: Strided iterators for particle properties.

Additionally particle fluids provide `PxParticleFluidReadData` with

- `densityBuffer`: Strided iterator for particle densities.

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->fla
    PxStrideIterator<const PxVec3> positionIt(rd->positionB

    for (unsigned i = 0; i < rd->validParticleRange; ++i, +
    {
        if (*flagsIt & PxParticleFlag::eVALID)
        {
            // access particle position
            const PxVec3& position = *positionIt;
        }
    }

    // return ownership of the buffers back to the SDK
    rd->unlock();
}

```

Example of how to use the valid particle bitmap to access particle data (without showing the locking and unlocking):

```

if (rd->validParticleRange > 0)
{
    // iterate over valid particle bitmap
    for (PxU32 w = 0; w <= (rd->validParticleRange-1) >> 5;
    {
        for (PxU32 b = rd->validParticleBitmap[w]; b; b
        {
            PxU32 index = (w << 5 | Ps::lowestSetBi

            // access particle position
            const PxVec3& position = rd->positionBu

        }
    }
}

```

Particle Drains

Sample reference:

```
void SampleParticles::createDrain()
```

The sample uses a *PxShape* plane, marked as a drain, to delete particles. The drain represents a lake that the fluid particles flow into, and is used as a general safety net in case any particles escape from the level.

Drains are generally a good method for keeping the particle count and spread under control. Placing drains around the area of interest in which a particle system is used helps to maintain good performance of the particle simulation. The area of interest could, for example, also be moved with the player.

Example of how to flag a *PxShape* *rbShape* as a drain:

```
rbShape->setFlag(PxShapeFlag::ePARTICLE_DRAIN, true);
```

Particles that collide with a drain are marked with *PxParticleFlag::eCOLLISION_WITH_DRAIN* and may be released.

Particle Grid and Spatial Data Structure Overflow

Sample reference:

```
void ParticleSystem::update(...)
```

The PhysX SDK uses a grid to subdivide the particles of a particle system into spatial groups. This is done to accelerate proximity queries and for parallelization purposes. The grid size parameter needs to be experimentally adjusted with *PxParticleBase::setGridSize()* for best performance. When doing this it is helpful to visualize the grid using *PxVisualizationParameter::ePARTICLE_SYSTEM_GRID*. Small grid size values might result in spatial data structure overflow, since the number of grid cells is limited to about 1000. Large grid size values on the other hand might result in poor performance due to ineffective spatial queries or lack of parallelization opportunities.

In case of overflow, some particles will stop colliding with rigid actors in the scene. These particles are marked with *PxParticleFlag::eSPATIAL_DATA_STRUCTURE_OVERFLOW* and should be released.

Collision Filtering

Sample reference:

```
PxFilterFlags SampleParticlesFilterShader(...)
```

Filtering particle versus rigid body collisions can be useful to avoid unnecessary performance overhead or simply to avoid undesired collisions. The sample filter shader is setup to

- Avoid particles colliding with trigger shapes (this is also the default filter shader behavior)
- Just have the particles collide with the drain shape
- Have two capsules with different radii to represent a force field around the laser. Each capsule interacts with just one particle system instance, smoke or water, to achieve effects of different strengths.

Filter information for particles can be specified by calling *PxParticleBase::setSimulationFilterData()*.

GPU/CUDA Acceleration

Sample reference:

```
void SampleBase::onInit()
```

PhysX 3 supports GPU acceleration. This allows for larger and more detailed particle effects while retaining good performance levels. To achieve this gain we must use a *pxtask::GpuDispatcher* for the scene we want to add the particle system to:

```
#ifdef PX_WINDOWS
    // create cuda context manager
    pxtask::CudaContextManagerDesc cudaContextManagerDesc;
    pxtask::CudaContextManager* cudaContextManager = pxtask
#endif

    PxSceneDesc sceneDesc(mPhysics->getTolerancesScale());
    //...
#ifdef PX_WINDOWS
    if (cudaContextManager)
        sceneDesc.gpuDispatcher = cudaContextManager->g
#endif
    //...
    physicsSdk->createScene(sceneDesc);
```

A particle system can be configured for GPU simulation by setting *PxParticleBaseFlag::eGPU*. Toggling GPU acceleration while the particle system is part of a scene might have a bad impact on performance since its state needs to be copied to or from the GPU device memory. It is therefore better to set the flag with *PxParticleBase::setParticleBaseFlag()* before adding the particle system to the scene.

Convex, Triangle and Height field meshes are automatically mirrored in the GPU memory when the corresponding shapes are within the proximity of a GPU accelerated particle system. This may cause some undesired performance hiccups which can be prevented by mirroring the meshes

explicitly, as shown in this example:

```
#ifdef PX_WINDOWS
    // mirror PxTriangleMesh triangleMesh providing the cor
    PxParticleGpu::createTriangleMeshMirror(triangleMesh, *

    // later release the obsolete mirror
    PxParticleGpu::releaseTriangleMeshMirror(triangleMesh,
#endif
```



Additional SampleParticles Information

The performance level of the particle sample varies by platform. Therefore different particle loads are chosen for different platforms, as can be seen in:

```
void SampleParticles::onInit()
```

The sample makes use of various helper classes:

- *ParticleSystem*: Encapsulates a *PxParticleSystem* or *PxParticleFluid* instance and manages application side data such as particle lifetimes and orientations for debris. It facilitates creating and releasing particles and double buffers particle data for asynchronous rendering.
- *RenderParticleSystemActor*: Owns a *ParticleSystem* and provides rendering functionality.
- *ParticleEmitterRate*: Emits particles at a specified rate (#particles per second).
- *ParticleEmitterPressure*: Emits particles maintaining a certain distance between them.
- *SampleParticles::Emitter*: Connects an emitter as described above with a *RenderParticleSystemActor*.
- *SampleParticles::Raygun*: Provides functionality for the ray force field, rigid body debris, particle debris and smoke emission.

In the sample, the smoke effect is achieved by using a *PxParticleSystem* without gravity. Each particle is rendered as a point sprite with a smoke texture. The sprites fade away when the particles get close to the end of their lifespan. The smoke particles collide with the scene, which can be seen when roaming the smoke with the ray-gun. Smoke is generated for the craters, as well as for the ray-gun impacts.

Two kinds of debris are shown in the sample. Larger chunks of debris are represented using convex-shaped rigid bodies. Smaller but more abundant chunks are represented by particles, which helps performance. The particle

based debris is rendered using instanced meshes. It is spawned in the craters and at the ray-gun impact location.

In order to give the chunks the appearance of a tumbling motion a simple trick is used.

1. Assign an initial random rotation matrix to each particle.
2. Change this rotation matrix proportional to the linear velocity of particle.

The implementation of this approach can be found in the following functions:

```
void ParticleSystem::initializeParticlesOrientations()  
void ParticleSystem::modifyRotationMatrix(...)
```


Introduction

PhysX 3 cloth is a rewrite of the PhysX 2 deformables, tailored towards simulating character cloth. Softbodies, tearing, two-way interaction, and world collision have been removed, while behavior and performance for cloth simulation have been improved.

Creating Cloth Fabric

The *PxClothFabric* class describes the constraint structure for a cloth. Constraints, for example a distance constraint, consist of two particle indices and a rest-length. These constraints are chained together as *fibers*, where two consecutive constraints of a fiber share a particle, multiple fibers are then further grouped into *sets*. By imposing the restriction that fibers of the same set do not overlap nor contain cycles, the fibers of a set can be solved in parallel. Each fabric may be shared between multiple cloth objects.

The cloth solver operates on one set of fibers at a time. This is referred as a solver *phase*, each phase has its own constraint type, such as stretching, bending or shearing, and an associated stiffness value.

The simplest way to create a fabric is to use the cloth cooking API. Given a *PxClothMeshDesc*, the cooker will create a set of constraints for a regular manifold triangle or quad mesh. The cooker will assign rest lengths and rest angles from the mesh automatically and it will organise constraints into vertical stretch, horizontal stretch, bending, or shear phases.

Below is an example function that creates a fabric using the cooking API and memory streams:

```
PxClothFabric* createFabric(PxPhysics &physics, PxCooking &cook
{
    // In this example, we cook the fabric on the fly throu
    // Note that we can also use a file stream and pre-cook
    PxToolkit::MemoryOutputStream wb;
    PX_ASSERT(desc.isValid());

    // Cook the fabric data into memory buffer
    if (!cooking.cookClothFabric(desc, gravityDir, wb))
        return 0;

    // Read fabric from memory stream
    PxToolkit::MemoryInputData rb(wb.getData(), wb.getSize(
    return physics.createClothFabric(rb);
```

```
}
```

Note: The direction of gravity is provided as a hint to the cooker, 'vertical' constraints will be placed parallel to this vector.

Creating Cloth Collision Data

In contrast to PhysX 2 deformables, PhysX 3 cloth does not collide with rigid bodies. Instead, each cloth object supports collision with spheres, capsules, planes and convex shapes (groups of planes), these shapes are all treated separately to the main PhysX rigid body scene.

Capsules are defined by a pair of indices into the spheres array and each sphere may have a different radius thus forming a tapered capsule. Sharing a sphere between two capsules is supported and can be useful for modelling characters (upper and lower leg made up from capsules can share the sphere at the knee), this sharing is encouraged because it helps make the simulation more efficient and robust.

Sphere and capsule shapes must be specified at cloth construction time. The following example shows how to set up the *PxClothCollisionData* object for a single capsule consisting of two spheres of radius 0.5 and 0.25:

```
// Two spheres located on the x-axis
PxClothCollisionSphere spheres[2] =
{
    {PxVec3(-1.0f, 0.0f, 0.0f), 0.5f},
    {PxVec3( 1.0f, 0.0f, 0.0f), 0.25f}
};

// A tapered capsule
PxU32 capsulePairs[] = { 0, 1 };

PxClothCollisionData collisionData;
collisionData.spheres = spheres;
collisionData.numSpheres = 2;
collisionData.pairIndexBuffer = capsulePairs;
collisionData.numPairs = 1;
```

Planes can be added through *PxCloth::addCollisionPlane()* method at any time after creation, but will not be considered for collision unless they are referenced by a convex shape. For example, the following code shows how to setup a typical upward facing ground plane through the origin:

```
PxClothCollisionPlane p;  
p.normal = PxVec3(0.0f, 1.0f, 0.0f);  
p.distance = 0.0f;  
  
PxU32 convexMask = 1; // Convex references the first plane only  
  
cloth.addCollisionPlane(p);  
cloth.addCollisionConvex(convexMask);
```

Planes may be efficiently updated after construction using the *PxCloth::setCollisionPlanes()* function.

Continuous Collision Detection

Besides discrete collision which resolves particles inside shapes at the end of each iteration, continuous collision detection is supported and can be enabled by calling:

```
// Enable continuous collision detection  
cloth.setClothFlag(PxClothFlag::eSWEPT_CONTACT, true);
```

Continuous collision is around 2x more computationally expensive than discrete collision, but it is necessary to detect collision between fast moving objects. Continuous collision analyzes the trajectory of particles and capsules to determine when a contact occurs. After the first time of contact, the particle is moved with the capsule until the end of the iteration.

Note: The SIMD collision path handles sets of 4 particles in parallel. It is therefore advantageous to spatially group cloth particles so that they are likely to collide with the same set of shapes.

Creating Cloth

With the fabric and collision data ready the *PxCloth* object can be created and added to the scene as below:

```
// Create a cloth object and add to the scene
Px Cloth* cloth = physics.createCloth( pose, fabric, particlePos
scene.addActor(cloth);
```

The *particlePositions* parameter is an array of *PxClothParticle* structures containing the initial particle position and the reciprocal of the particle mass. The initial positions will typically be the same as those provided during cooking and the mass of the particle may be set uniformly or non-uniformly as appropriate. For fixed position particles the inverse mass should be set to zero.

Simulation Overview

For one PhysX simulation frame, the cloth solver runs for multiple iterations. The number of iterations is determined by the solver frequency parameter and the simulation frame time. Each iteration integrates particle positions and solves distance constraints, motion constraints, and character collision. Local frame, motion constraints and collision shapes are interpolated per iteration from the per-frame values specified by the user.

Particle Integration

A particle state consists of the current position and the position before the last iteration. The particle velocity can be computed by dividing the position delta by the delta time of the previous iteration.

Particle positions are stored in local space, and accelerating the local frame affects the particles. The amount by which the local frame acceleration affects the cloth particles can be controlled using an inertia scale, for example to impart half the local frame acceleration to the particles use:

```
cloth.setInertiaScale(0.5f);
```

Limiting the amount that local frame changes affect particles can be especially useful for fast moving characters.

Even though using variable time-steps is generally not recommended, the simulation tries to handle variable time-steps carefully. Change in time-step is taken into account for position integration, and external forces are integrated using a smoothed time-step to avoid jittering.

Constraint Solving

The solver is run for a fixed number of iterations per simulation frame and can only enforce the constraints approximately. Two solver modes are provided with a different balance of speed and convergence rate. A semi-implicit solver, specified with `PxClothPhaseSolverConfig::eSTIFF` or a Gauss-Seidel style solver, specified with `PxClothPhaseSolverConfig::eFAST`.

If the distance constraints are not solved accurately enough, the cloth becomes stretchy. Because stretching cloth is most obvious under gravity, it is wise to pick the more accurate but about 2.5x slower semi-implicit solver for vertical stretch phases. If the built in cooking API is used to construct the fabric then these vertical constraints will be placed in the `PxClothFabricPhaseType::eSTRETCHING` phase according to the direction of gravity specified at cooking time.

The faster solver is usually good enough for the other phase types because approximate bending constraints for example are much less noticeable. Below is an example setup for multiple solver phases:

```
PxClothPhaseSolverConfig config;

// Use the semi-implicit solver for vertical distance constrain
config = cloth.getPhaseSolverConfig(PxClothFabricPhaseType::eST
config.solverType = PxClothPhaseSolverConfig::eSTIFF;
config.stiffness = 1.0f;
cloth.setPhaseSolverConfig(PxClothFabricPhaseType::eSTRETCHING,

// Use Gauss-Seidel solver for horizontal constraints
config = cloth.getPhaseSolverConfig(PxClothFabricPhaseType::eST
config.solverType = PxClothPhaseSolverConfig::eFAST;
config.stiffness = 1.0f;
cloth.setPhaseSolverConfig(PxClothFabricPhaseType::eSTRETCHING_

// Use Gauss-Seidel solver for shearing constraints
config = cloth.getPhaseSolverConfig(PxClothFabricPhaseType::eSH
config.solverType = PxClothPhaseSolverConfig::eFAST;
```

```
config.stiffness = 1.0f;
cloth.setPhaseSolverConfig(PxClothFabricPhaseType::eSHEARING, c

// Use bending solver for angle based bending constraints
config = cloth.getPhaseSolverConfig(PxClothFabricPhaseType::eBE
config.solverType = PxClothPhaseSolverConfig::eBENDING;
config.stiffness = 0.5f;
cloth.setPhaseSolverConfig(PxClothFabricPhaseType::eBENDING, co
```

Sometimes it is even desirable that distance constraints are not enforced rigorously. The stiffness parameter allows only correcting a portion of the edge length residual per iteration, for example to reduce the strength of bending constraints. A separate, lower stiffness can be used for edges that are only moderately stretched or compressed. For example, a dress can be made to stretch when the character is taking large steps, but still behave correctly during pirouettes.

The following code shows how to set up a phase such that when edges are compressed between 60% and 100% of the rest-length, a stiffness of $0.4 = 0.8 * 0.5$ will be used. If the edge is compressed more than 60% or if the edge is stretched, a stiffness of 0.8 will be used:

```
PxClothPhaseSolverConfig config;
config.solverType = PxClothPhaseSolverConfig::eFAST;
config.stiffness = 0.8f;
config.stretchStiffness = 0.5f;
config.stretchLimit = 0.6f;
```

Virtual Particles

Virtual particles provide a way of improving cloth collision without increasing the cloth resolution. They are called 'virtual' particles because they only exist during the collision processing stage and do not have their position, velocity or mass explicitly stored like regular particles, they can be thought of as providing additional samples on the collision surface.

A virtual particle is defined by 3 particle indices and an index into a weights table, the weights table defines the barycentric coordinates used to create a virtual particle position from a linear combination of the referenced particles. The following is an example weights table that can be used to create a distribution of 4 virtual particles on a triangle:

```
static PVec3 weights[] =
{
    // Center point
    PVec3(1.0f / 3, 1.0f / 3, 1.0f / 3),

    // Center of sub triangles
    PVec3(2.0f / 3, 1.0f / 6, 1.0f / 6),
    PVec3(1.0f / 6, 2.0f / 3, 1.0f / 6),
    PVec3(1.0f / 6, 1.0f / 6, 2.0f / 3),
};
```

During collision processing each virtual particle is tested for collision like a regular particle and the collision impulse is redistributed back to the original particles using reverse interpolation.

The code below shows an example of how to set up the virtual particles for a *PxClothMeshDesc*:

```
bool Test::ClothHelpers::createVirtualParticles(PxCloth& cloth,
{
    if(!numSamples)
        return false;

    PxU32 numFaces = meshDesc.triangles.count;
```

```

PxU8* triangles = (PxU8*)meshDesc.triangles.data;

PxU32 numParticles = numFaces * numSamples;
SampleArray<PxU32> virtualParticleIndices;
virtualParticleIndices.reserve(4 * numParticles);

for (PxU32 i = 0; i < numFaces; i++)
{
    for (int s = 0; s < numSamples; ++s)
    {
        PxU32 v0, v1, v2;

        if (meshDesc.flags & PxMeshFlag::e16_BIT
        {
            PxU16* triangle = (PxU16*)trian
            v0 = triangle[0];
            v1 = triangle[1];
            v2 = triangle[2];
        }
        else
        {
            PxU32* triangle = (PxU32*)trian
            v0 = triangle[0];
            v1 = triangle[1];
            v2 = triangle[2];
        }

        virtualParticleIndices.pushBack(v0);
        virtualParticleIndices.pushBack(v1);
        virtualParticleIndices.pushBack(v2);
        virtualParticleIndices.pushBack(s);
    }
    triangles += meshDesc.triangles.stride;
}

cloth.setVirtualParticles(numParticles, virtualParticle

return true;
}

```

Friction and Mass Scaling

Coloumb friction can be enabled and will be applied for particle and virtual particle collisions by setting a friction coefficient between 0 and 1:

```
cloth.setFrictionCoefficient(0.5f);
```

Additionally, there is an option to artificially increase the mass of colliding particles, this temporary increase in mass can help reduce stretching along edges that are being tightly pulled over a collision shape. The effect is determined by the relative normal velocity of the particle and collision shape and a user defined coefficient. A value of 20 is reasonable starting point but users are encouraged to experiment with this value:

```
cloth.setCollisionMassScale(20.0f);
```

Motion Constraints

Motion constraints lock the movement of each particle inside a sphere. For example, an animation system can sketch the global movement of a cloth while the fine scale details are handled by the cloth simulation.

A global scale and bias is also applied to each sphere radius. If the sphere radius becomes zero or negative, the corresponding particle is locked at the sphere center and the inverse particle mass is set to zero for the next iteration.

Separation constraints work the opposite way, forcing a particle to stay outside of a sphere. For cloth simulations with moderate particle movement, this can be used to represent the character's shape more accurately than using capsules alone.

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

The vehicle SDK can be thought of as having two separate components: the core vehicle SDK and an optional set of utility classes and functions that are provided as a reference solution to common problems in game vehicle dynamics. The utility classes and functions may be found in `PxVehicleUtilControl.h`, `PxVehicleUtilSetup.h`, and `PxVehicleUtilTelemetry.h`, while the remaining files make up the core vehicle SDK. It is important to note that these utility classes are provided only as an example solution to common game vehicle problems. It is quite possible that many developers will be able to use these utility classes directly without modification. However, it is expected that many developers will use these classes as a starting point for integrating the physx core vehicle SDK with gameplay code. An example might be the code that filters the raw inputs from a gamepad controller in order to generate meaningful acceleration, brake and steer values that can be passed to the core SDK. The implementation provided might be unsuitable for some developers because it is written with specific assumptions about the kind of smoothing required to generate enjoyable gameplay. It is perfectly reasonable that some developers will consider these assumptions to be at odds with the gameplay requirements of their project. Developers who find

themselves in this situation are invited to use the given implementation as a reference or even to independently develop an alternative if they wish. Other utility classes include data structures that record telemetry data in real time and helper functions to enable 3-wheeled cars.

In the following Sections the steps required to create and update a PhysX vehicle shall be discussed. The use of filter shaders to mark shapes as drivable and non-drivable shall be presented, as well as a demonstration of how to set up the friction of different combinations of tire type and drivable surface type. Following this, the recording and visualization of vehicle telemetry data is introduced with reference to tuning and debugging vehicle behaviors. Additionally, some of the key functions to query the internal vehicle state shall be introduced. The last Section details the steps required to set up 3-wheeled and N-wheeled cars and tanks. References to example code in SampleVehicle is made throughout.

Vehicle SDK Initialization

Before using the vehicle sdk it must first be initialized in order to set up a number of threshold values from various tolerance scales. This is as straightforward as calling the following function:

```
PX_C_EXPORT bool PX_CALL_CONV PxInitVehicleSDK(PxPhysics& phys
```

This function should be called after setting up the required PxPhysics and PxFoundation instances.

As expected, the vehicle sdk also has a shutdown process which needs to be invoked:

```
PX_C_EXPORT void PX_CALL_CONV PxCloseVehicleSDK();
```

This needs to be called before the PxPhysics instance and PxFoundation instance are released; that is, the order of shutdown is the reverse of the initialisation order.

Vehicle Creation

In this Section the process of vehicle creation shall be decomposed into an intuitive sequence of simple procedures. Each of these procedures will be explained in turn.

The first step in the process is to configure the data structures that completely describe the vehicle and its components:

```
PxVehicleWheelsSimData* wheelsSimData=PxVehicleWheelsSimData::a
PxVehicleDriveSimData4W driveSimData;
```

The above code instantiates the data structures for the wheels of a 4-wheeled car, as well as the data structures for non-wheel data such as engine, clutch, gears etc. With the exception of geometry data, the data structures are all given the default values of a "typical" car at construction. Geometry data must be explicitly set to successfully complete the creation of a PhysX vehicle, while the data fields with default values might be inappropriate for the vehicle under consideration. Some code is therefore required to configure the data to the requirements of the car in the sample:

```
PxVehicleChassisData chassisData;
createVehicle4WSimulationData
    (chassisMass, chassisConvexMesh,
     20.0f, wheelConvexMeshes4, wheelCentreOffsets4,
     *wheelsSimData, driveSimData, chassisData);
```

This code introduces the `PxVehicleChassisData` struct that is used to configure the PhysX actor that represents the vehicle in the PhysX SDK, and then calls a function to configure `wheelsSimData`, `driveSimData` and `chassisData` with meaningful data:

```
void createVehicle4WSimulationData
(const PxF32 chassisMass, PxConvexMesh* chassisConvexMesh,
 const PxF32 wheelMass, PxConvexMesh** wheelConvexMeshes, const
 PxVehicleWheelsSimData& wheelsData, PxVehicleDriveSimData4W& d
```

```

{
    //Extract the chassis AABB dimensions from the chassis
    const PxVec3 chassisDims=computeChassisAABBDimensions(c

    //The origin is at the center of the chassis mesh.
    //Set the center of mass to be below this point and a l
    const PxVec3 chassisCMOffset=PxVec3(0.0f,-chassisDims.y

    //Now compute the chassis mass and moment of inertia.
    //Use the moment of inertia of a cuboid as an approxima
    PxVec3 chassisMOI
        ((chassisDims.y*chassisDims.y + chassisDims.z*c
         (chassisDims.x*chassisDims.x + chassisDims.z*c
         (chassisDims.x*chassisDims.x + chassisDims.y*c
    //A bit of tweaking here. The car will have more respo
    //y-component of the chassis moment of inertia.
    chassisMOI.y*=0.8f;

    //Let's set up the chassis data structure now.
    chassisData.mMass=chassisMass;
    chassisData.mMOI=chassisMOI;
    chassisData.mCMOffset=chassisCMOffset;

    //Work out the front/rear mass split from the cm offset
    //This is a very approximate calculation with lots of a
    //massRear*zRear + massFront*zFront = mass*cm
    //massRear      + massFront      = mass
    //Rearrange (2)
    //massFront = mass - massRear
    //Substitute (3) into (1)
    //massRear(zRear - zFront) + mass*zFront = mass*cm
    //Solve (4) for massRear
    //massRear = mass(cm - zFront)/(zRear-zFront)
    //Now we also have
    //zFront = (z-cm)/2
    //zRear = (-z-cm)/2
    //Substituting (6a-b) into (5) gives
    //massRear = 0.5*mass*(z-3cm)/z
    const PxF32 massRear=0.5f*chassisMass*(chassisDims.z-3*
    const PxF32 massFront=chassisMass-massRear;

    //Extract the wheel radius and width from the wheel con
    PxF32 wheelWidths[4];
    PxF32 wheelRadii[4];
    computeWheelWidthsAndRadii(wheelConvexMeshes,wheelWidth

    //Now compute the wheel masses and inertias components

```

```

//http://en.wikipedia.org/wiki/List_of_moments_of_inert
PxU32 wheelMOIs[4];
for(PxU32 i=0;i<4;i++)
{
    wheelMOIs[i]=0.5f*wheelMass*wheelRadii[i]*wheel
}
//Let's set up the wheel data structures now with radiu
PxVehicleWheelData wheels[4];
for(PxU32 i=0;i<4;i++)
{
    wheels[i].mRadius=wheelRadii[i];
    wheels[i].mMass=wheelMass;
    wheels[i].mMOI=wheelMOIs[i];
    wheels[i].mWidth=wheelWidths[i];
}
//Disable the handbrake from the front wheels and enabl
wheels[PxVehicleDrive4W::eFRONT_LEFT_WHEEL].mMaxHandBra
wheels[PxVehicleDrive4W::eFRONT_RIGHT_WHEEL].mMaxHandBr
wheels[PxVehicleDrive4W::eREAR_LEFT_WHEEL].mMaxHandBrak
wheels[PxVehicleDrive4W::eREAR_RIGHT_WHEEL].mMaxHandBra
//Enable steering for the front wheels and disable for
wheels[PxVehicleDrive4W::eFRONT_LEFT_WHEEL].mMaxSteer=P
wheels[PxVehicleDrive4W::eFRONT_RIGHT_WHEEL].mMaxSteer=
wheels[PxVehicleDrive4W::eREAR_LEFT_WHEEL].mMaxSteer=0.
wheels[PxVehicleDrive4W::eREAR_RIGHT_WHEEL].mMaxSteer=0

//Let's set up the tire data structures now.
//Put slicks on the front tires and wets on the rear ti
PxVehicleTireData tires[4];
tires[PxVehicleDrive4W::eFRONT_LEFT_WHEEL].mType=TIRE_T
tires[PxVehicleDrive4W::eFRONT_RIGHT_WHEEL].mType=TIRE_
tires[PxVehicleDrive4W::eREAR_LEFT_WHEEL].mType=TIRE_TY
tires[PxVehicleDrive4W::eREAR_RIGHT_WHEEL].mType=TIRE_T

//Let's set up the suspension data structures now.
PxVehicleSuspensionData susps[4];
for(PxU32 i=0;i<4;i++)
{
    susps[i].mMaxCompression=0.3f;
    susps[i].mMaxDroop=0.1f;
    susps[i].mSpringStrength=35000.0f;
    susps[i].mSpringDamperRate=4500.0f;
}
susps[PxVehicleDrive4W::eFRONT_LEFT_WHEEL].mSprungMass=
susps[PxVehicleDrive4W::eFRONT_RIGHT_WHEEL].mSprungMass
susps[PxVehicleDrive4W::eREAR_LEFT_WHEEL].mSprungMass=m
susps[PxVehicleDrive4W::eREAR_RIGHT_WHEEL].mSprungMass=

```

```

//We need to set up geometry data for the suspension, w
//We already know the wheel centers described as offset
//From here we can approximate application points for t
//Lets assume that the suspension travel directions are
//Also assume that we apply the tire and suspension for
PxVec3 suspTravelDirections[4]={PxVec3(0,-1,0),PxVec3(0
PxVec3 wheelCentreCMOffsets[4];
PxVec3 suspForceAppCMOffsets[4];
PxVec3 tireForceAppCMOffsets[4];
for(PxU32 i=0;i<4;i++)
{
    wheelCentreCMOffsets[i]=wheelCentreOffsets[i]-c
    suspForceAppCMOffsets[i]=PxVec3(wheelCentreCMOf
    tireForceAppCMOffsets[i]=PxVec3(wheelCentreCMOf
}

//Now add the wheel, tire and suspension data.
for(PxU32 i=0;i<4;i++)
{
    wheelsData.setWheelData(i,wheels[i]);
    wheelsData.setTireData(i,tires[i]);
    wheelsData.setSuspensionData(i,susps[i]);
    wheelsData.setSuspTravelDirection(i,suspTravelD
    wheelsData.setWheelCentreOffset(i,wheelCentreCM
    wheelsData.setSuspForceAppPointOffset(i,suspFor
    wheelsData.setTireForceAppPointOffset(i,tireFor
}

//Now set up the differential, engine, gears, clutch, a

//Diff
PxVehicleDifferential4WData diff;
diff.mType=PxVehicleDifferential4WData::eDIFF_TYPE_LS_4
driveData.setDiffData(diff);

//Engine
PxVehicleEngineData engine;
engine.mPeakTorque=500.0f;
engine.mMaxOmega=600.0f;//approx 6000 rpm
driveData.setEngineData(engine);

//Gears
PxVehicleGearsData gears;
gears.mSwitchTime=0.5f;
driveData.setGearsData(gears);

```

```

//Clutch
PxVehicleClutchData clutch;
clutch.mStrength=10.0f;
driveData.setClutchData(clutch);

//Ackermann steer accuracy
PxVehicleAckermannGeometryData ackermann;
ackermann.mAccuracy=1.0f;
ackermann.mAxleSeparation=wheelCentreOffsets[PxVehicleDrive4W::FRONT];
ackermann.mFrontWidth=wheelCentreOffsets[PxVehicleDrive4W::FRONT];
ackermann.mRearWidth=wheelCentreOffsets[PxVehicleDrive4W::REAR];
driveData.setAckermannGeometryData(ackermann);
}

```

In the above code, tire types have been specified for the front and rear vehicles. These tire types have an influence on the friction that the tires will experience on different surface types. This will be discussed in more detail in Section [Tire Friction On Drivable Surfaces](#).

The next step is to create a PhysX actor that will be used to represent the vehicle and its collision geometry as a rigid body actor in the PhysX sdk:

```

PxRigidDynamic* vehActor=createVehicleActor4W(chassisData,wheelData);

```

This function creates a PhysX rigid body actor, sets up all the shapes for the wheels and chassis, configures the rigid body mass and inertia, and sets collision and scene query filters for wheel and chassis shapes. The filtering shall be discussed in more detail in Section [Filtering](#). When adding shapes to the vehicle an ordering of the shapes is assumed such that the wheels are added first and in a specific order, followed by the chassis shapes. This is documented in the vehicle SDK, along with a description of further vehicle SDK functions that open up the possibility of having any shape order. This is discussed in more detail in Section [Vehicle Actor](#).

The final step is to create a PhysX vehicle:

```

PxVehicleDrive4W* car = PxVehicleDrive4W::allocate(4);
car->setup(&physics,vehActor,*wheelsSimData,driveSimData,0);

```

This code creates a car with 4 driven wheels and no non-driven wheels, as described by `wheelsSimData/driveSimData`, and with a rigid body represented in the PhysX SDK by `vehActor`. Two representations of the vehicle (`PxVehicleDrive4W` and `PxRigidDynamic`) are necessary because vehicles are simulated in two distinct steps. In the first step the vehicle simulation code runs outside of the core PhysX SDK. This code computes tire and suspension forces/torques that are applied to each vehicle's rigid body actor, in addition to simulating the internal dynamics of the vehicle. These rigid body actors are subsequently updated inside the PhysX SDK, taking into account the applied vehicle dynamics forces/torques as well as the collision shapes of the actor that represent the combination of chassis and wheels. More details of these two update steps are discussed in [Section Vehicle Update](#).

Vehicle Actor

The actor representing the vehicle in the PhysX SDK is configured with this function:

```
void setupActor
(PxRigidDynamic* vehActor,
 const PxFilterData& vehQryFilterData,
 const PxGeometry** wheelGeometries, const PxTransform* wheelLo
 const PxGeometry** chassisGeometries, const PxTransform* chass
 const PxVehicleChassisData& chassisData,
 PxPhysics* physics)
{
    //Add all the wheel shapes to the actor.
    for(PxU32 i=0;i<numWheelGeometries;i++)
    {
        PxShape* wheelShape=vehActor->createShape(*wheel
        wheelShape->setQueryFilterData(vehQryFilterData
        wheelShape->setSimulationFilterData(wheelCollFi
        wheelShape->setLocalPose(wheelLocalPoses[i]);
    }

    //Add the chassis shapes to the actor.
    for(PxU32 i=0;i<numChassisGeometries;i++)
    {
        PxShape* chassisShape=vehActor->createShape(*ch
        chassisShape->setQueryFilterData(vehQryFilterDa
        chassisShape->setSimulationFilterData(chassisCo
        chassisShape->setLocalPose(chassisLocalPoses[i]
    }

    vehActor->setMass(chassisData.mMass);
    vehActor->setMassSpaceInertiaTensor(chassisData.mMOI);
    vehActor->setCMassLocalPose(PxTransform(chassisData.mCM
}
}
```

Each wheel and chassis shape is assigned a query filter data and a simulation filter data. Additionally, each shape is given a local pose. For wheels this local pose is just the rest pose of the wheel. The local pose of each wheel is subsequently reset after each vehicle update pass to reflect the wheel's current position along the suspension travel direction. See Section

[Vehicle Update](#) for more details.

The above code adds the wheel shapes in the order specified in `PxVehicleDrive4W::eWheelOrdering`, followed by the shapes of the chassis. This is the default shape ordering expected by the PhysX vehicle SDK. The freedom to order the shapes in the actor, however, might not be available in all games. With this in mind it is possible to set up a mapping between each wheel and the position of the corresponding shape in the actor's shapes array:

```
void PxVehicleWheels::setWheelShapeMapping(const PxU32 wheelId,
```

Tire Friction on Drivable Surfaces

In this Section setting up tire types, drivable surface types, and tire friction on combinations of tire and surface type shall be discussed.

In Section [Vehicle Creation](#) tire types were assigned to the front and rear tires:

```
PxVehicleTireData tires[4];
tires[PxVehicleDrive4W::eFRONT_LEFT_WHEEL].mType=TIRE_TYPE_SLIC
tires[PxVehicleDrive4W::eFRONT_RIGHT_WHEEL].mType=TIRE_TYPE_SLI
tires[PxVehicleDrive4W::eREAR_LEFT_WHEEL].mType=TIRE_TYPE_WETS;
tires[PxVehicleDrive4W::eREAR_RIGHT_WHEEL].mType=TIRE_TYPE_WETS
```

In addition to tire types, drivable surface types can be associated with PxMaterial instances:

```
mSurfaceTirePairs=PxVehicleDrivableSurfaceToTireFrictionPairs::
```

This code creates a mapping, stored in a PxVehicleDrivableSurfaceToTireFrictionPairs instance, that associates the nth PxMaterial pointer in the drivableSurfaceMaterials array with the nth PxVehicleDrivableSurfaceType in the drivableSurfaceTypes array. As a consequence, PxVehicleDrivableSurfaceToTireFrictionPairs permits a mapping between each specified PxMaterial instance and the integer value stored in PxVehicleDrivableSurfaceType. It is not necessary to store all possible PxMaterial instances in the hash table: unrecognised PxMaterial pointers are automatically associated with a surface type having integer value zero.

Each combination of surface type and tire type can be assigned a unique friction value. This allows different types of tires to react differently to the same surface conditions. Wet tires, for example, would be expected to behave very differently on wet tarmac to slick tires. This code refines the

friction for each possible combination in SampleVehicle:

```
for(PxU32 i=0;i<MAX_NUM_SURFACE_TYPES;i++)
{
    for(PxU32 j=0;j<MAX_NUM_TIRE_TYPES;j++)
    {
        mSurfaceTirePairs->setTypePairFriction(i,j,gTir
    }
}
```

Here, a table of friction values has been hard-coded:

```
//Tire model friction for each combination of drivable surface
static PxF32 gTireFrictionMultipliers[MAX_NUM_SURFACE_TYPES][MA
{
    //WETS  SLICKS  ICE  MUD
    {0.95f, 0.95f, 0.95f, 0.95f}, //MUD
    {1.10f, 1.15f, 1.10f, 1.10f}, //TARMAC
    {0.70f, 0.70f, 0.70f, 0.70f}, //ICE
    {0.80f, 0.80f, 0.80f, 0.80f} //GRASS
};
```

In this table the friction of slick tires on tarmac has been assigned a value of 1.15. Slick tires on mud, on the other hand, have been assigned a friction value of 1.10. By tuning this table, along with the tire types assigned to each vehicle and the surface types associated with each material, it is possible to modify the tire grip to the specifications of any game.

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

characteristics required for games. Another reason that friction is not clamped at 1.0 is that games typically simulate the physics update at 60Hz. This comes at a cost to numerical accuracy, especially when there are a number of transient tire effects that require KHz update frequencies. One source of numerical accuracy is the amplitude of oscillation of the suspension, which is governed in turn by the distance that the vehicle falls under gravity between each update. At KHz update frequencies this simulation artifact is acceptably small, but not at 60Hz. The last reason is that there is simply no need to impose the strict rules of friction on the vehicles SDK. This can allow interesting behaviors to be generated that would perhaps be impossible when constrained by the laws of rigid body and tire dynamics. Having said all this, however, the implemented model simulated at 60Hz ought to have enough integrity that only small tweaks above 1.0 should be necessary. If very large friction values are required, say greater than 2.0, then it is likely that something is wrong with the update order or perhaps very unphysical vehicle data has been used.

The PxMaterial instances discussed here are the results of per-wheel raycasts. Raycast filtering shall be discussed in more detail in Section [Filtering](#). The relationship between raycast execution and vehicle updates shall be introduced in Section [Vehicle Update](#).

Filtering

In this Section the concepts of wheel raycast filtering shall be introduced. Following this, the steps required to add filter data to a drivable surface and configure the filter data of vehicle shapes will be described.

The key goal of filtering for vehicles is to configure raycasts and collisions in such a way that raycasts along the suspension travel directions interact with drivable surfaces while also ensuring that the wheel shapes themselves never collide with drivable surfaces. This allows vehicles on drivable surfaces to be supported by spring forces arising from raycast intersections without interference from wheel shape intersection.

A simulation filter is required for the wheel shapes to ensure they do not collide with drivable surfaces:

```
PxFilterData wheelCollFilterData;  
wheelCollFilterData.word0=COLLISION_FLAG_WHEEL;  
wheelCollFilterData.word1=COLLISION_FLAG_WHEEL_AGAINST;
```

Similarly, a simulation filter is required for the chassis shapes but this time it is desired that the chassis shapes do collide with drivable surfaces:

```
PxFilterData chassisCollFilterData;  
chassisCollFilterData.word0=COLLISION_FLAG_CHASSIS;  
chassisCollFilterData.word1=COLLISION_FLAG_CHASSIS_AGAINST;
```

The combination of simulation filter shader:

```
if ((filterData0.word0 != 0 || filterData1.word0 != 0) &&  
    !(filterData0.word0&filterData1.word1 || filterData1.word0&filterData0.word1))  
    return PxFilterFlag::eSUPPRESS;
```

with the definitions of the
COLLISION_FLAG_DRIVABLE_OBSTACLE_AGAINST and

COLLISION_FLAG_WHEEL_AGAINST flags enforces the rule that the wheel shapes do not collide with the drivable surfaces, while also allowing the chassis shapes to collide with drivable surfaces.

In addition to simulation filter data, it is necessary to configure query filter data for the vehicle wheel raycasts. Drivable surfaces, such as the terrain and ramp obstacles, are assigned query filter data as follows:

```
PxFilterData qryFilterData;  
SampleVehicleSetupDrivableShapeQueryFilterData(&qryFilterData);
```

Non-drivable surfaces, such as dynamic boxes and pendula in SampleVehicle, are also assigned query filter data:

```
PxFilterData qryFilterData;  
SampleVehicleSetupNonDrivableShapeQueryFilterData(&qryFilterData);
```

In SampleVehicle, the vehicle shapes have been assigned query filter data that forbids vehicles being treated as drivable surfaces:

```
PxFilterData vehQryFilterData;  
SampleVehicleSetupVehicleShapeQueryFilterData(&vehQryFilterData);
```

It is not at all necessary to stop cars driving on other cars. In SampleVehicle this choice has been made for reasons of simplicity but it would be straightforward to configure the scene queries to allow cars to drive on other cars. Care must be taken, however, to configure the filters to ensure that wheel raycasts only interact with the shapes of other vehicles. If a raycast was to intersect the geometry of the vehicle that issued the raycasts then there is a strong possibility that the intersection with the drivable terrain would be rejected in favour of a deeper intersection with the vehicle geometry.

Similar to the filter shader described above, a query filter shader must also

be specified:

```
static PxSceneQueryHitType::Enum SampleVehicleWheelRaycastPreFi
    PxFilterData filterData0,
    PxFilterData filterData1,
    const void* constantBlock, PxU32 constantBlockSize,
    PxSceneQueryFilterFlags& filterFlags)
{
    //filterData0 is the vehicle suspension raycast.
    //filterData1 is the shape potentially hit by the rayca
    PX_UNUSED(filterFlags);
    PX_UNUSED(constantBlockSize);
    PX_UNUSED(constantBlock);
    PX_ASSERT(filterData0.word3 & SAMPLEVEHICLE_UNDRIVABLE_
    return ((0 == (filterData1.word3 & SAMPLEVEHICLE_DRIVAB
}
```

The vehicle SDK ensures that filterData0 in the above function is always assigned the query filter data of the shape of the wheel that is issuing the raycast. For the case of SampleVehicle this means that filterData0 will always have the properties of a non-drivable surface because wheel shapes have been set up to be non-drivable in the sample.

Deployment of the query filters will be discussed in Section [Vehicle Update](#).

Vehicle Update

It has already mentioned that vehicles are updated in two stages: specific vehicle code that updates the vehicle internal dynamics and computes forces/torques to apply to the vehicle's rigid body representation, followed by an SDK update that accounts for the applied forces/torques as well as collision with other scene bodies. In this Section these separate update phases will be discussed in some detail.

Beginning with the vehicle dynamics update, it is necessary to initialise and execute batched raycasts for suspension line intersection with drivable shape surfaces:

```
void SampleVehicle_VehicleManager::suspensionRaycasts(PxScene*
{
    //Create a scene query if we haven't already done so.
    if(NULL==mSqWheelRaycastBatchQuery)
    {
        mSqWheelRaycastBatchQuery=mSqData->setUpBatched
    }
    //Raycasts.
    PxVehicleSuspensionRaycasts(mSqWheelRaycastBatchQuery,m
}
```

Creating a batched scene query is discussed elsewhere in the guide and in the api documentation. The key points to note here are that `SampleVehicleWheelRaycastPreFilter` is used as the shader for the the batched scene query, and that buffers have been pre-allocated to provide enough memory for a single raycast hit for each active wheel.

Having completed setting up the scene queries, the scene queries can now be issued for all vehicles:

```
PxVehicleSuspensionRaycasts(mSqWheelRaycastBatchQuery,mNumVehic
```

There is some freedom in the order in which raycasts can be issued relative to the vehicle dynamics update. In a real-world situation it might be that raycasts can be issued on a separate thread at the end of the update loop so that they are ready for the beginning of the next. However, this really all depends on the threading environment and the ordering of rigid body updates. As always, a few rules must be observed. The first rule is that there must be a one-to-one correspondence between raycast completion and updates for each vehicle. The second rule is that a raycast must be issued before an update is allowed. That does not mean that the raycast must be issued first in a game's update loop, only that the first raycasts must be completed before the first update and then each must be done in turn. The last rule is that the raycasts and updates must be performed sequentially if they involve the same vehicles.

With the raycasts complete the update of the vehicles can now be performed:

```
void SampleVehicle_VehicleManager::update(const PxF32 timestep,
{
    //Update.
    PxVehicleUpdates(timestep, gravity, *mSurfaceTirePairs, mN
}
```

In Section [Tire Friction On Drivable Surfaces](#) it was discussed how to set up friction for different combinations of drivable surface type and tire type. The parameter `mSurfaceTirePairs` in the above code shows the application of the friction values and of the mapping of `PxMaterial` instances to integer values representing surface type.

A second code-path has been implemented to allow the update of each vehicle and to record telemetry data for a single specific vehicle:

```
void SampleVehicle_VehicleManager::updateAndRecordTelemetryData
(const PxF32 timestep, const PxVec3& gravity, PxVehicleWheels*
{
    PX_ASSERT(focusVehicle && telemetryData);
```

```
//Update the vehicle for which we want to record debug
PxVehicleUpdateSingleVehicleAndStoreTelemetryData(times

//Update the remaining vehicles.
PxVehicleWheels* vehicles[MAX_NUM_4W_VEHICLES];
PxU32 numVehicles=0;
for(PxU32 i=0;i<mNumVehicles;i++)
{
    if(focusVehicle!=mVehicles[i])
    {
        vehicles[numVehicles]=mVehicles[i];
        numVehicles++;
    }
}
PxVehicleUpdates(timestep,gravity,*mSurfaceTirePairs,nu
}
```

The function:

```
//Update the vehicle for which we want to record debug data.
PxVehicleUpdateSingleVehicleAndStoreTelemetryData(timestep,grav
```

updates a single vehicle and collects telemetry data for that vehicle. The remaining cars, those from whom telemetry data is not collected, are updated in the usual way. Telemetry data shall be discussed in more detail in Section [Telemetry](#).

To ensure thread safety with the telemetry data collection it is important that the update with telemetry data and the regular update run sequentially. Telemetry data is unlikely to be collected in the final build of a real game so this ought to have no significant impact on release performance.

The code discussed so far in this Section performs raycasts for each wheel and uses the results to compute tire and suspension forces to apply to each rigid body actor. Simultaneous to the computation of tire and suspension forces, wheel and engine rotation speeds are also updated. None of this code, however, advances the vehicle through the game world. This advancement is performed by the PhysX SDK update, where vehicles are

represented by rigid body actors. There isn't too much to say here except that the SDK update and the vehicle dynamics update must run sequentially. It ought not to matter in which order they are updated in a game update loop or which is updated first. It is worth bearing in mind, though, that the vehicle dynamics update poses the wheel shapes along the suspension travel using the results of the suspension line raycasts. This might have an impact on the desired ordering of vehicle update and vehicle render.

Tire Shaders

It is possible to replace the default tire model used by PhysX vehicles with custom models. This requires a shader function that can be set per-vehicle along with shader data that must be set per-wheel:

```
void PxVehicleWheelsDynData::setTireForceShaderFunction(PxVehic  
void PxVehicleWheelsDynData::setTireForceShaderData(const PxU32
```

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 r  
const PxF32 restTireLoad, const PxF32 normalisedTireLoad, cons  
const PxF32 gravity, const PxF32 recipGravity,  
PxF32& wheelTorque, PxF32& tireLongForceMag, PxF32& tireLatFor
```

The vehicle update code will call the shader function for each wheel with the shader data for that wheel.

Telemetry

The purpose of the telemetry data is to expose the inner dynamics of the car and aid handling tuning through the use of telemetry graphs. In this Section initialisation, collection, and rendering of telemetry data shall be discussed.

In Section [Vehicle Update](#) the possibility of collecting telemetry data for a single vehicle was introduced with this code:

```
//Update the vehicle for which we want to record debug data.  
PxVehicleUpdateSingleVehicleAndStoreTelemetryData(timestep, grav
```

Setting up the telemetry data is relatively straightforward:

```
mTelemetryData4W = PxVehicleTelemetryData::allocate(4);  
  
const PxF32 graphSizeX=0.25f;  
const PxF32 graphSizeY=0.25f;  
const PxF32 engineGraphPosX=0.5f;  
const PxF32 engineGraphPosY=0.5f;  
const PxF32 wheelGraphPosX[4]={0.75f, 0.25f, 0.75f, 0.25f};  
const PxF32 wheelGraphPosY[4]={0.75f, 0.75f, 0.25f, 0.25f};  
const PxVec3 backgroundColor(255, 255, 255);  
const PxVec3 lineColorHigh(255, 0, 0);  
const PxVec3 lineColorLow(0, 0, 0);  
mTelemetryData4W->setup  
    (graphSizeX, graphSizeY,  
     engineGraphPosX, engineGraphPosY,  
     wheelGraphPosX, wheelGraphPosY,  
     backgroundColor, lineColorHigh, lineColorLow);
```

The sizes, positions, and colors are all values that are used to render the graphs. The exact values of these fields will depend on the coordinate system and color coding being used to visualize the telemetry data. In the above example, the coordinates have been configured to render an engine-related graph in the centre of the screen. Screen coordinates have also been specified for rendering data associated with each of the four wheels. The coordinates used in the vehicle sample visualize the graph data of the front

left wheel at the top left-hand side of the screen, the front right wheel at the top right-hand side of the screen, and the rear wheels at the bottom left and right.

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

The data for a particular graph channel is computed as follows:

```
telemetryData.getWheelGraph(i).computeGraphChannel(activewheelG
```

This code computes a sequence of screen coords in the format [x0,y0,x1,y1,x2,y2,...xn,yn] that represent the points of the specified graph channel of the engine's graph data. In SampleVehicle only a single graph channel is rendered at a time with the possibility of iterating through the channels from the keyboard or gamepad. It is important to note that the data for all the channels (see the above enumerated lists) is available so it is possible to visualize simultaneously many channels for each graph. The above function also stores a color for each graph point. The idea here is that each graph channel is assigned a value that separates high and low values, along with a color that is assigned to points with high values, and a color that is assigned to points with low values. This is a useful feature if multiple channels are being simultaneously visualized. In SampleVehicle the value that separates high and low values for each channel is hard-coded within the optional setup function PxVehicleTelemetryData::setup. With this function the jounce channel, for example, uses zero as the demarcation between high value and low value. This allows spring compression and elongation to be easily recognised from the different colors used to represent these two states. Users are free to set up graphs as required if the provided graph setup is unsuited to their needs.

The friction and surface type experienced by each tire is also printed to screen. The surface and tire types are retrieved as follows:

```
tireTypes[i]=focusVehicle.mWheelsSimData.getTireData(i).mType;  
surfaceTypes[i]=focusVehicle.mWheelsDynData.getTireDrivableSurf
```

In Section [Vehicle State Queries](#) there is further discussion of vehicle internal state queries.

Vehicle State Queries

It has already been mentioned that a PhysX vehicle is a multi-shape actor with a single shape for each wheel and multiple shapes for the chassis. It has also been mentioned that the local poses of the wheels are set during the vehicle dynamics update to reflect the results of the suspension raycasts. The transforms used to render the vehicle can readily be found by querying the shapes of the vehicle:

```
PxShape* carShapes[PX_MAX_NUM_WHEELS+1];
const PxVehicleWheels& vehicle=*mVehicleManager.getVehicle(carI
const PxU32 numShapes=vehicle.getRigidDynamicActor()->getNbShap
vehicle.getRigidDynamicActor()->getShapes(carShapes, numShapes);
```

Further data is recorded each update and stored in the vehicle. Much of this data is useful for triggering audio and graphics effects based on the state of the vehicle. For example, particles with a specific texture might be emitted from a wheel if the wheel is found to be spinning at high speed on a specific surface type. The following functions would then be useful:

```
PxReal PxVehicleWheelsDynData::getTireLongSlip(const PxU32 tireI
PxReal PxVehicleWheelsDynData::getTireLatSlip(const PxU32 tireI
PxU32 PxVehicleWheelsDynData::getTireDrivableSurfaceType(const
```

Alternatively, the audio system could be fed with engine revs data using this function:

```
PX_FORCE_INLINE PxReal PxVehicleDriveDynData::getEngineRotation
{
    return mEnginespeed;
}
```

For skidding audio and particle effects the following functions are probably worth noting:

```
PxReal PxVehicleWheels::computeForwardSpeed() const;  
PxReal PxVehicleWheels::computeSidewaysSpeed() const;
```

Vehicle Controls

In this Section setting the control values used to drive a vehicle shall be discussed. These are accelerate, brake, handbrake, steer, gear change up, and gear change down. The simplest way to do this is to use the following function:

```
void PxVehicleDriveDynData::setAnalogInput(const PxReal analogV
```

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

A solution to the problem of filtering controller input data is to assign a rise and fall rate to each button or pad. For analog values under digital control it is possible to simply increase or decrease the analog value at a specified rate depending on whether the digital input is on or off. For analog values under analog control it makes more sense to blend from the previous input value to the current input at a specified rate. A slight complication to this simple model is that the difficulty of achieving a large steer angle at large speed must also be modelled. One technique to achieve this would be to model the forces from the tires' aligning moments and apply these to a steering linkage model. This sounds rather complicated and quite difficult to tune. A simpler solution might be to scale the filtered steer value by another value in range (0,1) that decreases at high speed. This simpler method has been implemented in the helper classes and functions.

Rise and fall rates for digital and analog control have been hard-coded in SampleVehicle:

```
PxVehicleKeySmoothingData gKeySmoothingData=
{
    {
        3.0f,    //rise rate eANALOG_INPUT_ACCEL
        3.0f,    //rise rate eANALOG_INPUT_BRAKE
        10.0f,   //rise rate eANALOG_INPUT_HANDBRAKE
        2.5f,    //rise rate eANALOG_INPUT_STEER_LEFT
        2.5f,    //rise rate eANALOG_INPUT_STEER_RIGHT
    },
    {
        5.0f,    //fall rate eANALOG_INPUT__ACCEL
        5.0f,    //fall rate eANALOG_INPUT__BRAKE
        10.0f,   //fall rate eANALOG_INPUT__HANDBRAKE
        5.0f,    //fall rate eANALOG_INPUT_STEER_LEFT
        5.0f     //fall rate eANALOG_INPUT_STEER_RIGHT
    }
};

PxVehiclePadSmoothingData gCarPadSmoothingData=
{
    {
        6.0f,    //rise rate eANALOG_INPUT_ACCEL
        6.0f,    //rise rate eANALOG_INPUT_BRAKE
        12.0f,   //rise rate eANALOG_INPUT_HANDBRAKE
        2.5f,    //rise rate eANALOG_INPUT_STEER_LEFT
        2.5f,    //rise rate eANALOG_INPUT_STEER_RIGHT
    },
    {
        10.0f,   //fall rate eANALOG_INPUT_ACCEL
        10.0f,   //fall rate eANALOG_INPUT_BRAKE
        12.0f,   //fall rate eANALOG_INPUT_HANDBRAKE
        5.0f,    //fall rate eANALOG_INPUT_STEER_LEFT
        5.0f     //fall rate eANALOG_INPUT_STEER_RIGHT
    }
};
```

A look-up table has also been specified to describe the maximum steer as a function of 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
};
PxFixedSizeLookupTable<8> gSteerVsForwardSpeedTable(gSteerVsFor

```

Using a `PxVehicleDrive4WRawInputData` instance it is straightforward to record the user inputs in the event a keyboard is used:

```

rawInputData.setDigitalAccel(mAccelKeyPressed);
rawInputData.setDigitalBrake(mBrakeKeyPressed);
rawInputData.setDigitalHandbrake(mHandbrakeKeyPressed);
rawInputData.setDigitalSteerLeft(mSteerLeftKeyPressed);
rawInputData.setDigitalSteerRight(mSteerRightKeyPressed);
rawInputData.setGearUp(mGearUpKeyPressed);
rawInputData.setGearDown(mGearDownKeyPressed);

```

or in the event that a gamepad is used:

```

rawInputData.setAnalogAccel(mAccel);
rawInputData.setAnalogBrake(mBrake);
rawInputData.setAnalogHandbrake(mHandbrake ? 1.0f : 0.0f);
rawInputData.setAnalogSteer(mSteer);
rawInputData.setGearUp(mGearup);
rawInputData.setGearDown(mGeardown);

```

Here, `rawInput` data is an instance of the vehicle SDK helper class `PxVehicleDrive4WRawInputData`.

The vehicle SDK offers two optional functions to smooth the keyboard or gamepad data and apply the smoothed input values to the PhysX vehicle:

```

if(mUseKeyInputs)
{
    PxVehicleDrive4WSmoothDigitalRawInputsAndSetAnalogInput
}

```

```
else  
{  
    PxVehicleDrive4WSmoothAnalogRawInputsAndSetAnalogInputs  
}
```

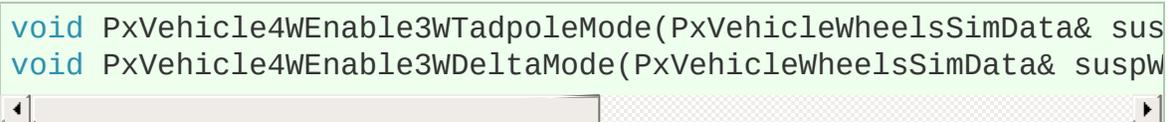


3-Wheeled, N-Wheeled Cars, and N-Wheeled Tanks

The primary focus until now has been on how to create, simulate and query 4-wheeled cars. This Section shall focus instead on cars that either have only 3 wheels or more than 4 wheels. To complete the discussion many-wheeled tanks shall be introduced.

Utility functions have been provided to quickly configure 3-wheeled cars. The basic idea is to start with a 4-wheeled car and then disable one of the wheels:

```
void PxVehicle4WEnable3WTadpoleMode(PxVehicleWheelsSimData& sus  
void PxVehicle4WEnable3WDeltaMode(PxVehicleWheelsSimData& suspW
```



These functions ensure that no raycast hits are returned for that wheel and additionally do some other work to decouple the disabled wheel from the differential, disable ackermann correction, re-position the opposite remaining wheel to the centre of the axle, and adjust the suspension of the opposite remaining wheel to compensate for the missing suspension of the disabled wheel. Further wheels could in theory be removed with custom code to create a vehicle with 1 or 2 effective wheels. At that point, however, extra balancing code would be required to prevent the vehicle falling over.

In addition to removing wheels from a vehicle, it is also possible to construct a PxVehicleDrive4W with more than 4 wheels. The functionality of the extra wheels, however, is slightly limited compared to the first 4 wheels. More specifically, only the first 4 wheels are connected to the differential or the steering; that is, only the first block of 4 wheels can experience a drive torque or a steer angle. As a consequence, the extra wheels play an identical role to the rear wheels of a 4-wheeled car that has front-wheel drive or the front wheels of a 4-wheeled car that has rear-wheel drive. Adding extra wheels does not preclude the ability to call PxVehicle4WEnable3WTadpoleMode or

PxVehicle4WEnable3WDeltaMode. These functions, however, are hard-coded to disable one of the 4 wheels that could be connected to the steering and driven through the differential.

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

```
PxVehicleWheelsSimData* wheelsSimData=PxVehicleWheelsSimData::a
PxVehicleDriveSimData4W driveSimData;
setupSimData(wheelsSimData,driveSimData);
PxVehicleDrive4W* car = PxVehicleDrive4W::allocate(6);
PxRigidBody* vehActor=createVehicleActor6W();
car->setup(&physics,vehActor,*wheelsSimData,driveSimData,2);
```

The PhysX vehicle SDK also supports tanks through the use of the PxVehicleDriveTank class. Tanks are different to multi-wheeled vehicles in that the wheels are all driven through the differential in a way that ensures that all the wheels on the left-hand side have the same speed, and all the wheels on the right-hand have the same speed. This extra constraint on wheel speed mimics the effect of the caterpillar tracks but avoids the expense of simulating the jointed track structure. Adding the geometry of the caterpillar tracks is as easy as adding an actor shape down each side and setting up the collision and query filters as appropriate for the tracks. The motion of the caterpillar tracks could be rendered with a scrolling texture, safe in the knowledge that all wheels have the same speed, just as though they were properly constrained by the track rotation.

Creating a PxVehicleDriveTank instance is very similar to creating a PxVehicleDrive4W instance with the exception that tanks have no concept of extra wheels that are not connected to the differential: all tank wheels are driven. The following code illustrates how to set up a 12-wheeled tank:

```
PxVehicleWheelsSimData* wheelsSimData = PxVehicleWheelsSimData:
PxVehicleDriveSimData4W driveSimData;
setupTankSimData(wheelsSimData,driveSimData);
PxVehicleDriveTank* tank = PxVehicleDriveTank::allocate(12);
PxRigidBody* vehActor=createVehicleActor12W();
```

```
tank->setup(&physics, vehActor, *wheelsSimData, tankDriveSimData, 1
```

Controlling a tank is quite different to controlling a car because tanks have a completely different steering mechanism: the turning action of a tank arises from the difference in left and right wheel speeds, while cars turn by the action of a steering wheel that orientates the front wheels relative to the forward motion of the vehicle. This requires quite a different set of helper classes and functions to smooth the control inputs. Instead of a `PxVehicleDrive4WRawInputData` for N-wheeled cars, `PxVehicleDriveTankRawInputData` is used for N-wheeled tanks. Further, instead of `PxVehicleDrive4WSmoothDigitalRawInputsAndSetAnalogInputs/PxVehicleD` for N-wheeled cars, `PxVehicleDriveTankSmoothDigitalRawInputsAndSetAnalogInputs` and `PxVehicleDriveTankSmoothAnalogRawInputsAndSetAnalogInputs` are used to smooth the keyboard and gamepad inputs for tanks.

PhysX tanks currently support two drive models: `eDRIVE_MODEL_STANDARD` and `eDRIVE_MODEL_SPECIAL`. The drive model `eDRIVE_MODEL_SPECIAL` allows the tank tracks to rotate in different directions, while `eDRIVE_MODEL_STANDARD` does not. These two modes result in quite different turning actions. Drive model `eDRIVE_MODEL_STANDARD` simulates the usual turning action of a tank: pushing forward on the left(right) stick drives the left(right) wheels forward, while pulling back on the right(left) stick applies the brake to the right(left) wheels. `eDRIVE_MODEL_SPECIAL`, on the other hand, simulates a more exotic turning action where pushing back on the right(left) stick drives the right(left) wheels backwards. This can result in a turning circle focused at the centre of the tank. The smallest possible turning circle of a tank in `eDRIVE_MODEL_STANDARD` will have a focus at a point along one of the caterpillar tracks, depending on whether the tank is turning left or right.

Tuning Guide

This Section describes the effect of the editable vehicle parameters of the data structures in PxVehicleComponents.h.

PxVehicleWheelData

mRadius:

This is the distance in metres between the centre of the wheel and the outside rim of the tire. It is important that the value of the radius closely matches the radius of the render mesh of the wheel. Any mismatch will result in the wheels either hovering above the ground or intersecting the ground. Ideally, this parameter will be exported from the 3D modeller.

mWidth:

This is the full width of the wheel in metres. This parameter has no bearing on the handling but is a very useful parameter to have when trying to render debug data relating to the wheel/tire/suspension. Without this parameter it would be difficult to compute coordinates for render points and lines that ensure their visibility. Ideally, this parameter will be exported from the 3D modeller.

mMass:

This is the combined mass of the wheel and the tire in kg. Typically, a wheel has mass between 20Kg and 80Kg but can be lower and higher depending on the vehicle.

mMOI:

This is the component of the wheel's moment of inertia about the rolling axis. Larger values make it harder for the wheel to rotate about this axis, while easier values make it easier for the wheel to rotate

about the rolling axis. Another way of expressing this is that a high MOI will result in less wheel spin when stamping on the accelerator because it is harder to make the wheel spin. Conversely, lower values of MOI will result in more wheel spin when stamping on the accelerator.

If the wheel is approximately cylindrical then a simple formula can be used to compute MOI:

$$\text{MOI} = 0.5 * \text{Mass} * \text{Radius} * \text{Radius}$$

There is no reason, however, to rely on equations to compute this value. A good strategy for tuning this number might be to start with the equation above and then make small tweaks to the value until the handling is as desired.

mDampingRate:

This value describes how quickly a freely spinning wheel will come to rest. The damping rate describes the rate at which a freely spinning wheel loses rotational speed. Here, a freely spinning wheel is one that experiences no forces except for the damping forces arising from the wheel's internal bearings. Higher damping rates result in the wheel coming to rest in shorter times, while lower damping rates result in the wheel maintaining speed for longer. Values in range (0.25, 2) seem like sensible values. Experimentation is always a good idea, even outside this range. Always exercise some caution with very small damping rates. In particular, a damping rate of exactly 0 should be avoided.

mMaxBrakeTorque:

This is the value of the torque applied to the wheel when the brakes are maximally applied. Higher torques will lock the wheel quicker when braking, while lower torques will take longer to lock the wheel. This value is strongly related to the wheel MOI because the MOI determines how quickly the wheel will react to applied torques.

A value of around 1500 is a good starting point for a vanilla wheel but a google search will reveal typical braking torques. One difficulty is that these are often expressed by manufacturers as braking horsepower or in "pounds inches". The values required here are in "Newton metres".

mMaxHandBrakeTorque:

This is the same as the max brake torque except for the handbrake rather than the brake. Typically, for a 4-wheeled car, the handbrake is stronger than the brake and is only applied to the rear wheels. A value of 4000 for the rear wheels is a good starting point, while a value of 0 is necessary for the front wheels to make sure they do not react to the handbrake.

mMaxSteer:

This is the value of the steer angle of the wheel (in radians) when the steering wheel is at full lock. Typically, for a 4-wheeled car, only the front wheels respond to steering. In this case, a value of 0 is required for the rear wheels. More exotic cars, however, might wish front and rear wheels to respond to steering. A value in radians equivalent to somewhere between 30 degrees and 90 degrees seems like a good starting point but it really depends on the vehicle being simulated. Larger values of max steer will result in tighter turns, while smaller values will result in wider turns. Be aware, though, that large steer angles at large speeds are likely to result in the car losing traction and spinning out of control, just as would happen with a real car. A good way to avoid this is to filter the steer angles passed to the car at run-time to generate smaller steer angles at larger speeds. This strategy will simulate the difficulty of achieving large steer angles at high speeds (at high speeds the wheels resist the turning forces applied by the steering wheel).

mToeAngle:

This is the angle of the wheel (in radians) that occurs with no steer applied. The toe angle can be used to help the car straighten up after

coming out of a turn. This is a good number to experiment with but is best left at 0 unless detailed tweaks are required.

To help the car straighten up apply a small negative angle to one of the front wheels and a small positive angle to the other front wheel. By choosing which wheel takes the positive angles, and which the negative, it is straightforward to make the wheels either "toe'in" or "toe'out". A "toe-in" configuration, the front wheels pointing slightly towards each other, should help the car straighten up after a turn but at the expense of making it a little harder to turn in the first place. A "toe-out" configuration can have the opposite effect. Toe angles greater than a few degrees are best avoided.

PxVehicleWheelsSimData

`void setSuspTravelDirection(const PxU32 id, const PxVec3& dir):`

This is the direction of the suspension in the downward direction in the rest configuration of the vehicle. A vector that points straight downwards is a good starting point.

`void setSuspForceAppPointOffset(const PxU32 id, const PxVec3& offset):`

This is the application point of the suspension force, expressed as an offset vector from the center of mass of the vehicle's rigid body. Another way of expressing this is to start at the center of mass of the rigid body, then move along the offset vector. The point at the end of the offset vector is the point at which suspension forces will be applied.

In a real vehicle the suspension forces are mediated through the suspension strut. These are often incredibly complex mechanical systems that are computationally expensive to simulate. As a consequence, instead of modelling the details of the suspension strut, it makes sense to assume that the suspension strut has an effective point at which it applies the force to the rigid body. Choosing that point,

however, needs careful consideration. At the same time, it opens up all sorts of tweaking possibilities, freed from the constraints of the real world.

Deciding on the suspension force application point requires some thought. The suspension is very close to the wheel so the wheel center is a good starting point. Consider a line through the wheel center and along the suspension travel direction. Somewhere along this line seems like an even better idea for the application point, albeit not completely scientific. For a standard 4-wheeled car it makes sense that the application point is somewhere above the wheel center but below the centre of mass of the rigid body. It is probably above the wheel centre because the suspension is mostly above this point. It can be assumed that it is somewhere below the rigid body centre of mass because otherwise vehicles would lean out of the turn rather than in to the turn. This narrows down the application point to really quite a small section of a known line.

When editing the suspension force application point it is important to bear in mind that lowering the app point too far will result in cars leaning more into the turn. This can have a negative effect on handling because the inner wheel can take so much load that the response saturates, while the outer wheel ends up with reduced load and reduced turning force. The result is poor cornering. Conversely, setting the app point too high will result in cornering that looks unnatural. The aim is to achieve a good balance.

```
void setTireForceAppPointOffset(const PxU32 id, const PxVec3& offset):
```

This is almost the same as the suspension force app point except for the lateral and longitudinal forces that develop on the tire. A good starting point is to duplicate the suspension force application point. Only for really detailed editing is it advised to start tweaking the tire force app offset independently of the suspension force app offset.

```
void setWheelCentreOffset(const PxU32 id, const PxVec3& offset):
```

This is the centre of the wheel at rest position, expressed as an offset vector from the vehicle's centre of mass.

PxVehicleSuspensionData

mSprungMass:

This is the mass in kg that is supported by the suspension spring.

A vehicle with rigid body centre of mass at the centre of the four wheels would typically be equally supported by each of the suspension springs; that is, each suspension spring supports 1/4 of the total vehicle mass. If the centre of mass was moved forward then it would be expected that the front wheels would need to support more mass than the rear wheels. Conversely, a centre of mass nearer the rear wheels ought to result in the rear suspension springs supporting more mass than at the front.

mMaxCompression:

mMaxDroop:

These values describe the maximum compression and elongation in metres that the spring can support. The total travel distance along the spring direction that is allowed is the sum of mMaxCompression and mMaxDroop.

A simple way to illustrate the maximum droop and compression values is to consider a car that is suspended in mid-air so that none of the wheels are touching the ground. The wheels will naturally fall downwards from their rest position until the maximum droop is reached. The spring cannot be elongated beyond this point. Now consider that the wheel is pushed upwards, first to its rest position, then further pushed until the spring can no longer be compressed. The displacement from the rest position is the maximum compression of the spring.

It is important to choose the maximum compression value so that the wheel is never placed where the visual mesh of the wheel intersects the visual meshes of the car chassis. Ideally, these values will be exported from the 3d modeller.

mSpringStrength:

This is the strength of the suspension spring in Newtons per metre. The spring strength has a profound influence on handling by modulating the time it takes for the vehicle to respond to bumps in the road and on the amount of load experienced by the tire.

Key to the understanding the effect of spring strength is the concept of a spring's natural frequency. Consider a simple spring system, such as a pendulum swinging back and forth. The number of trips per second that the pendulum makes from full left to full right and then back again is called the natural frequency of the pendulum. A more powerful pendulum spring will result in the pendulum swinging faster, thereby increasing the natural frequency. Conversely, increasing the pendulum mass will result in a slower oscillation, thereby reducing the natural frequency.

In the context of a suspension spring supporting a fixed portion of vehicle mass, the strength of the spring will affect the natural frequency; that is, the rate at which the spring can respond to changes in load distribution. Consider a car taking a corner. As the car corners it leans in to the turn, putting more weight on the suspensions on the outside of the turn. The speed at which the spring reacts by applying forces to redistribute the load is controlled by the natural frequency. Very high natural frequencies, such as those on a racing car, will naturally produce twitchy handling because the load on the tires, and therefore the forces they can generate, is varying very rapidly. Very low natural frequencies, on the other hand, will result in the car taking a long time to straighten up even after the turn is complete. This will produce sluggish and unresponsive handling.

Another effect of strength and natural frequency is the response of a car to a bump in the road. High natural frequencies can result in the car responding very strongly and quickly to the bump, with the wheel possibly even leaving the road for a short while. This not only creates a bumpy ride but also periods of time when the tire is generating no forces. Weaker springs will result in a smoother trip over the bump, with weaker but more constant tire forces. A balance must be found to tune the car for the expected types of turn and terrain.

The natural frequency of the spring presents a challenge for computer simulation. A smooth and stable simulation requires that the spring is updated at a frequency much greater than the spring's natural frequency. An alternative way of expressing this is to consider the period of the spring relative to the timestep of the simulation. The period of the spring is the time the spring takes to complete a single oscillation, and is mathematically equal to the reciprocal of the natural frequency. In order to achieve a stable simulation the spring must be sampled at several points during each oscillation. A natural consequence of this observation is that the simulation timestep must be significantly smaller than the period of the spring. To discuss this further it is helpful to introduce a ratio that describes the number of simulation updates that will occur during each spring oscillation. This ratio is simply the spring period divided by the timestep

$$\alpha = \text{sqrt}(m\text{SprungMass}/m\text{SpringStrength})/\text{timestep}$$

where $\text{sqrt}(m\text{SprungMass}/m\text{SpringStrength})$ is the period of the spring. An alpha value of 1.0 means that the chosen timestep and spring properties only allow a single sample of the spring during each oscillation. As described above, this is almost guaranteed to produce unstable behaviour. In fact, the argument presented so far suggests a value of alpha significantly greater than 1.0 is essential to produce a smooth simulation. The exact value of alpha at which stability emerges is very difficult to predict and depends on many other parameters. As a guide, however, it is recommended that the timestep and spring

properties are chosen so that they produce an alpha value greater than 5.0; that is, a minimum of five simulation updates per spring cycle.

When tuning a suspension spring it can be very useful to use manufacturer data to discover typical values used across a range of vehicle types. This data is not always readily available. An alternative strategy would be to think in terms of the natural frequency of the spring by imagining how quickly the car would oscillate up and down if it was dropped onto the ground from a height of, say, 0.5m. The springs of a typical family car have natural frequency somewhere between 5 and 10; that is, such a car would make 5-10 oscillations per second if gently dropped to the ground. If the mass supported by the spring is already known then the spring strength can be calculated from the following equation

$$\text{mSpringStrength} = \text{naturalFrequency} * \text{naturalFrequency} * \text{mSprungMass}$$

mSpringDamperRate:

This describes the rate at which the spring dissipates the energy stored in the spring.

Key to the understanding of damper rate are the concepts of under-damping, over-damping, and critical damping. An over-damped pendulum displaced from rest is unable to make a single back-and-forth trip before it dissipates all its energy, while an under-damped pendulum would be able to make at least a single back-and-forth trip. A critically damped pendulum makes exactly a single back-and-forth trip before expending all its energy.

For vehicle suspension springs, it is typically important to make sure that the spring has a damper rate that produces over-damping but not by too much. When cornering, for example, it is important that the spring doesn't over-respond by shifting the weight from the left suspension to the right suspension then back again. If this happened

the tire load, and the forces generated, would be extremely variable, resulting in twitchy and uncontrollable handling. A very heavily over-damped spring, on the other hand, will feel sluggish and unresponsive.

The concept of critical damping can be used to help tune the damping rate of the spring. It is helpful to introduce a value known as the damping ratio, which helps to mathematically describe the under-damping, critical damping and over-damping regimes.

$$\text{dampingRatio} = \text{mSpringDamperRate} / [2 * \text{sqrt}(\text{mSpringStrength} * \text{mSprungMass})]$$

A dampingRatio with value greater than 1.0 produces over-damping, a value of exactly 1.0 generates critical damping, and a value less than 1.0 is under-damped. It can be useful to first think about whether the spring will be under-damped or over-damped, then think about how far it will be from critical damping. This process allows a number to be subjectively applied to the damping ratio. From here the damping rate can be directly computed by rearranging the equation above

$$\text{mSpringDamperRate} = \text{dampingRatio} * 2 * \text{sqrt}(\text{mSpringStrength} * \text{mSprungMass})$$

A typical family car is probably slightly over-damped, having dampingRatio with value perhaps just over 1.0. A guideline would be that values very far from critical damping are likely to be unrealistic and will either produce sluggish or twitchy handling. It is difficult to put an exact figure on this but somewhere between 0.8 and 1.2 seems like a good starting point for the damping ratio.

PxVehicleTireData

mLongitudinalStiffnessPerUnitGravity:

The longitudinal tire force is approximately the product of the longitudinal stiffness per unit longitudinal slip (in radians) per unit

gravity and the longitudinal slip and the magnitude of gravitational acceleration.

Increasing this value will result in the tire attempting to generate more longitudinal force when the tire is slipping. Typically, increasing longitudinal stiffness will help the car accelerate and brake. The total tire force available is limited by the load on the tire so be aware that increases in this value might have no effect or even come at the expense of reduced lateral force.

mLatStiffX:

mLatStiffY:

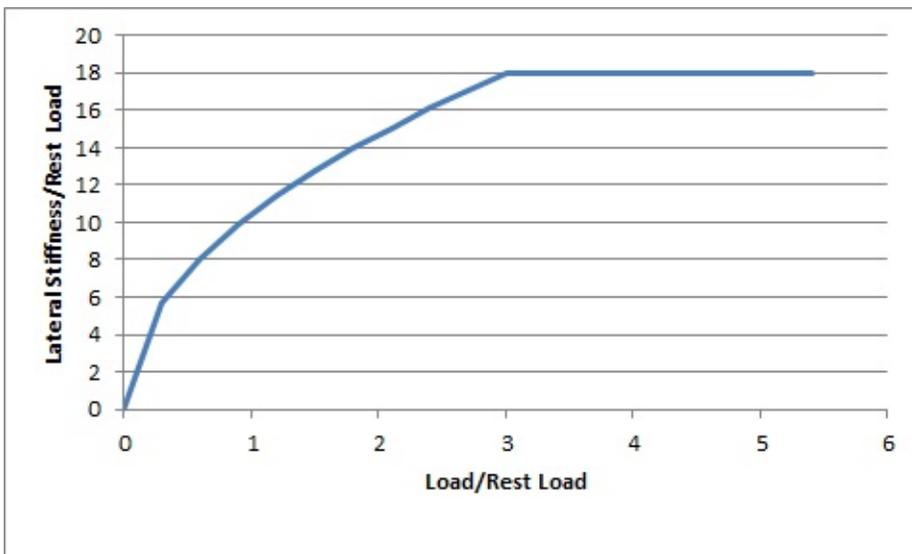
These values together describe the lateral stiffness per unit lateral slip (in radians) of the tire. The lateral stiffness of a tire has a role similar to the longitudinal stiffness, except that it governs the development of lateral tire forces, and is a function of tire load. Typically, increasing lateral stiffness will help the car turn more quickly. The total tire force available is limited by the load on the tire so be aware that increases in this value might have no effect or even come at the expense of reduced longitudinal force.

The combination of the two values mLatStiffX and mLatStiffY describe a graph of lateral stiffness as a function of normalised tire load. Typical for car tires is a graph that has linear response close to zero load but saturates at greater loads. This means that at low tire loads the lateral stiffness has a linear response to load; that is, more load results in more stiffness. At higher tire loads the tire has a saturated response and is in a regime where applying more load will not result in more tire stiffness. In this latter regime it would be expected that the tire would start slipping.

The parameter mLatStiffX describes the normalised tire load above which the tire has a saturated response to tire load. The normalised tire load is simply the tire load divided by the load experienced when the

vehicle is perfectly at rest. A value of 2 for mLatStiffX means that when the the tire has a load more than twice its rest load it can deliver no more lateral stiffness no matter how much extra load is applied to the tire.

The parameter mLatStiffY describes the maximum stiffness per unit of lateral slip (in radians) per unit rest load. The maximum stiffness is delivered when the tire is in the saturated load regime, governed in turn by mLatStiffX.



A good starting value for mLatStiffX is somewhere between 2 and 3. A good starting value for mLatStiffY is around 18 or so.

mFrictionVsSlipGraph[0][0]:

mFrictionVsSlipGraph[0][1]:

mFrictionVsSlipGraph[1][0]:

mFrictionVsSlipGraph[1][1]:

mFrictionVsSlipGraph[2][0]:

mFrictionVsSlipGraph[2][1]:

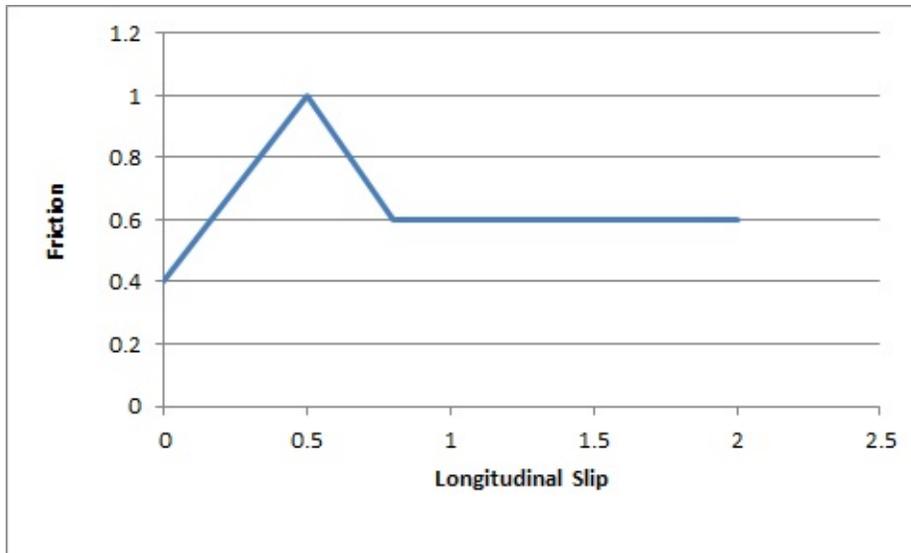
These six values describe a graph of friction as a function of longitudinal slip. Vehicle tires have a complicated response to longitudinal slip and this graph attempts to quickly describe this relationship.

Typically, tires have a linear response at small slips. This means that when the tire is only slightly slipping it is able to generate a response force that grows as the slip increases. At greater values of slip, the force can actually start to decrease from the peak value that occurs at the optimum slip. Beyond the optimum slip the tire eventually stops behaving less and less efficiently and hits a plateau of inefficiency.

The first two values describe the friction at zero tire slip: $mFrictionVsSlipGraph[0][0] = 0$, and $mFrictionVsSlipGraph[0][1] =$ friction at zero slip.

The next two values describe the optimum slip and the friction at the optimum slip: $mFrictionVsSlipGraph[1][0] =$ optimum slip, $mFrictionVsSlipGraph[1][1] =$ friction at optimum slip.

The last two values describe the slip at which the plateau of inefficiency begins and the value of the friction available at the plateau of inefficiency: $mFrictionVsSlipGraph[2][0] =$ slip at the start of the plateau of inefficiency, $mFrictionVsSlipGraph[2][1] =$ the friction available at the plateau of inefficiency.



The friction values described here are used to scale the friction of the ground surface. This means they should be in range (0,1) but this is not a strict requirement. Typically, the friction from the graph would be close to 1.0 in order to provide a small correction to the ground surface friction.

A good starting point for this is a flat graph of friction vs slip with these values:

`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`

`mCamberStiffness:`

This value is currently unused.

mType:

This parameter has been explained in Section [Tire Friction on Drivable Surfaces](#).

PxVehicleEngineData

mPeakTorque:

This is the maximum torque that is ever available from the engine. This is expressed in Newton metres. A starting value might be around 600.

mMaxOmega:

This is the maximum rotational speed of the engine expressed in radians per second.

mDampingRateFullThrottle:

mDampingRateZeroThrottleClutchEngaged:

mDampingRateZeroThrottleClutchDisengaged:

These three values are used to compute the damping rate that is applied to the engine. If the clutch is engaged then the damping rate is an interpolation between mDampingRateFullThrottle and mDampingRateZeroThrottleClutchEngaged, where the interpolation is governed by the acceleration control value generated by the gamepad or keyboard. At full throttle mDampingRateFullThrottle is applied, while mDampingRateZeroThrottleClutchEngaged is applied at zero throttle. In neutral gear the damping rate is an interpolation between mDampingRateFullThrottle and mDampingRateZeroThrottleClutchDisengaged.

The three values allow a range of effects to be generated: good acceleration that isn't hampered by strong damping forces, tunable

damping forces when temporarily in neutral gear during a gear change, and strong damping forces that will bring the vehicle quickly to rest when it is no longer being driven by the player.

Typical values in range (0.25,3). The simulation can become unstable with damping rates of 0.

mTorqueCurve:

This is a graph of peak torque versus engine rotational speed. Cars typically have a range of engine speeds that produce good drive torques, and other ranges of engine speed that produce poor torques. A skilled driver will make good use of the gears to ensure that the car remains in the "good" range where the engine is most responsive. Tuning this graph can have profound effects on gameplay.

The x-axis of the curve is the normalised engine speed; that is, the engine speed divided by the maximum engine speed. The y-axis of the curve is a multiplier in range (0,1) that is used to scale the peak torque.

PxVehicleGearsData

mNumRatios:

This is the number of the gears of the vehicle, including reverse and neutral. A standard car with 5 forward gears would, therefore, have a value of 7 after accounting for reverse and neutral.

mRatios:

Each gear requires a gearing ratio. Higher gear ratios result in more torque but lower top speed in that gear. Typically, the higher the gear, the lower the gear ratio. Neutral gear must always be given a value of 0, while reverse gear must have a negative gear ratio. Typical values might be 4 for first gear and 1.1 for fifth gear.

mFinalRatio:

The gear ratio used in the simulator is the gear ratio of the current gear multiplied by the final ratio. The final ratio is a quick and rough way of changing the gearing of a car without having to edit each individual entry. Further, quoted gearing values from manufacturers typically mention ratios for each gear along with a final ratio. A typical value might be around 4.

mSwitchTime:

The switch time describes how long it takes (in seconds) for a gear change to be completed. It is impossible to change gear immediately in a real car. Manual gears, for example, require neutral to be engaged for a short time before engaging the desired target gear. While the gear change is being completed the car will be in neutral. A good trick might be to penalise players that use an automatic gear box by increasing the gear switch time.

PxVehicleClutchData

mStrength:

This describes how strongly the clutch couples the engine to the wheels and how quickly differences in speed are eliminated by distributing torque to the engine and wheels.

Weaker values will result in more clutch slip, especially after changing gear or stamping on the accelerator. Stronger values will result in reduced clutch slip, and more engine torque delivered to the wheels.

This value is to be edited only for very fine tweaking of the vehicle. Some clutch slip can be attributed to the numerical issues in the simulation at large timesteps, while some is a natural consequence of driving the car in an overly aggressive manner. A value of 10 is a good starting point.

PxVehicleAckermannGeometryData

mAccuracy:

Ackermann correction allows better cornering by steering the left and right wheels with slightly different steer angles, as computed from simple trigonometry. In practice, it is impossible to engineer a steering linkage that will achieve the perfect Ackermann steering correction. This value allows the accuracy of the Ackermann steering correction to be controlled. Choosing a value of 0 completely disables Ackermann steer correction. A value of 1.0, on the other hand, achieves the impossible dream of 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 centre of the front axle and the centre of the rear axle.

PxVehicleTireLoadFilterData

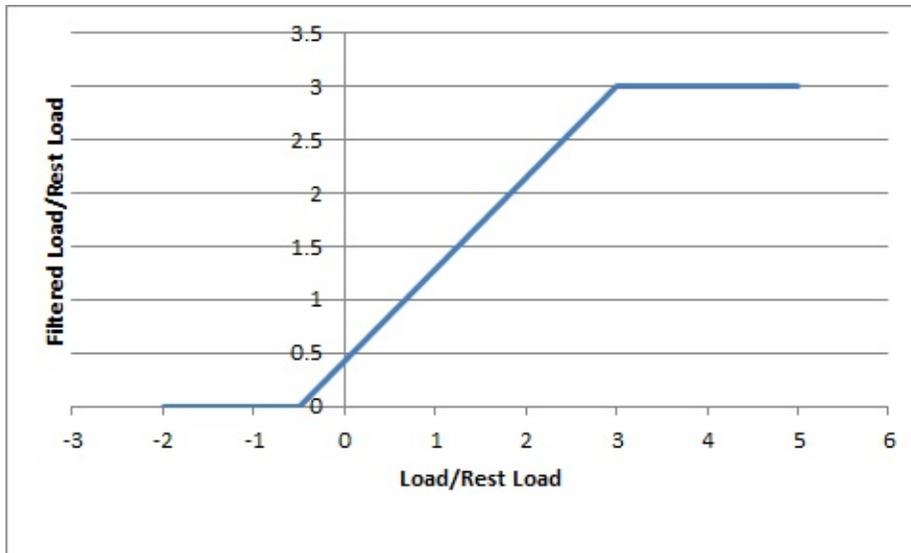
This is for very fine control of the handling, and corrects numerical issues inherent in simulations at large timesteps.

At large simulation timesteps the amplitude of motion of the suspension springs is larger than it would be in real-life. This is unfortunately unavoidable. A consequence of this oscillation is that the load on the tire is more variable than expected, and the available tire forces have more variability than expected. On a bumpy surface this could mean that the simulation lifts the wheel off the ground, while in reality it should have stayed on the ground and delivered turning force. This filter aims to correct this numerical problem by smoothing the tire

load, and perhaps even allowing turning force when the wheel is off the ground.

A key concept is that of normalised tire loads. A normalised tire load is just the actual load divided by the load experienced when the vehicle is in its rest configuration. If a tire experiences more load than it does at rest then it has a normalised tire load greater than 1.0. Similarly, if a tire has less load than it does at rest then it has a normalised tire load less than 1.0. At rest, all tires obviously have a normalised tire load of exactly 1.0. Another key idea is that when the wheel is just off the ground it can have a small negative load. Now, negative loads are a rather artificial result of the modelling mathematics and should be neglected entirely, but as already discussed, it might be beneficial to produce some tire force even when the tire is off the ground.

The values here describe points on a 2d graph that generates filtered tire loads from raw tire loads. The x-axis of the graph is "normalised tire load", while the y-axis of the graph is "filtered normalised tire load". Normalised loads less than `mMinNormalisedLoad` produce a filtered normalised load of 0. Normalised loads greater than `mMaxNormalisedLoad` produce a filtered normalised load of `mMaxFilteredNormalisedLoad`. Load in-between `mMinNormalisedLoad` and `mMaxNormalisedLoad` produce a filtered normalised load in-between 0 and `mMaxNormalisedLoad`, as computed by direct interpolation.



Choosing negative values for `mMinNormalisedLoad` results in small turning forces even when the tire is slightly off the ground. Additionally, choosing `mMaxNormalisedLoad` and `mMaxFilteredNormalisedLoad` limits the maximum load that will ever be used in the simulation. To disable the correcting effect of this graph choose `mMinNormalisedLoad=0`, `mMaxNormalisedLoad=1000`, and `mMaxFilteredNormalisedLoad=1000`.

PxVehicleDifferential4WData

`mType`:

A number of differential types are supported: 4-wheel drive with open differential, 4-wheel drive with limited slip, front-wheel drive with open differential, front-wheel drive with limited slip, rear-wheel drive with open differential, rear-wheel drive with limited slip.

`mFrontRearSplit`:

If a 4-wheel drive differential is chosen (open or limited slip) this option allows the drive torque to be split unevenly between the front and rear wheels. Choosing a value of 0.5 delivers an equal split of the torque between the front and rear wheels; that is, the total torque delivered to the front wheels is equal to the total torque delivered to

the rear wheels. Choosing a value greater than 0.5 delivers more torque to the front wheels, while choosing a value less than 0.5 delivers more torque to the rear wheels. This value is ignored for front-wheel drive and rear-wheel drive differentials.

mFrontLeftRightSplit:

This is similar to the Front Rear Split but instead splits the torque that is available for the front wheels between the front-left and front-right wheels. A value greater than 0.5 delivers more torque to the front-left wheel, while a value less than 0.5 delivers more torque to the front-right wheel. This parameter can be used to prevent any torque being delivered to a damaged or disabled wheel. This value is ignored for rear-wheel drive.

mRearLeftRightSplit:

This is similar to mFrontLeftRightSplit except that it applies to the rear wheels instead of the front wheels. This value is ignored for front-wheel drive.

mFrontBias:

Limited slip differentials work by only allowing a certain difference in wheel rotation speed to accumulate. This prevents the situation where one wheel is slipping but ends up taking all the available power. Further, by allowing a small difference in wheel rotation speed to accumulate it is possible for the vehicle to easily corner by permitting the outside wheel to rotate quicker than the inside wheel.

This parameter describes the maximum difference in wheel rotation speed that is allowed to accumulate. The front bias is the maximum of the two front-wheel rotation speeds divided by the minimum of the two front-wheel rotation speeds. When this ratio exceeds the value of the front bias the differential diverts torque from the faster wheel to the slower wheel in an attempt to preserve the maximum allowed wheel rotation speed ratio.

This value is ignored except for front-wheel drive or four wheel drive with limited slip.

A good starting value is around 1.3.

mRearBias:

This is similar to mFrontBias except that it refers to the rear wheels.

This value is ignored except for rear-wheel drive or four wheel drive with limited slip.

A good starting value is around 1.3.

mCentreBias:

This value is similar to the mFrontBias and mRearBias, except that it refers to the sum of the front wheel rotation speeds and the sum of the rear wheel rotation speeds.

This value is ignored except for four wheel drive with limited slip.

A good starting value is around 1.3.

PxRigidDynamic

Moment of Inertia:

The moment of inertia of the rigid body is an extremely important parameter when editing vehicles because it affects the turning and rolling of the vehicle.

A good starting point for the moment of inertia of the rigid body is to work out the moment of inertia of the cuboid that bounds the chassis geometry. If the bounding cuboid is W wide, H high, and L long then the moment of inertia for a vehicle of mass 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 modify the motion around the corresponding axis, with higher values making it harder to induce rotational speed from tire and suspension forces.

Providing unphysical values for the moment of inertia will result in either very sluggish behaviour or extremely twitchy and perhaps even unstable behaviour. The moment of inertia must at least approximately reflect the length scales of the suspension and tire force application points.

This parameter should be viewed as one of the first go-to editable values.

Center of mass:

Along with the moment of inertia, the center of mass is one of the first go-to editable values and, as such, has a profound effect on handling.

To discuss the center of mass it is useful to consider a typical 4-wheeled vehicle with a chassis mesh whose origin is at the centre of the four wheels. There is no requirement on the origin being at the center of the four wheels but it does make the following discussion a little simpler. It might be expected that the center of mass lies somewhere near this origin because vehicles are designed in a way that spreads the load almost evenly between the four wheels. More specifically, it might be expected that the center of mass needs to be a little above the base of the chassis rather than at the height of the wheels. After all, vehicles have higher mass density near the bottom of the chassis due to density of the engine and other mechanical systems. As a consequence, it is expected that the center of mass is nearer the bottom of the chassis than the top, but definitely above the bottom. Without a particularly detailed analysis of the chassis density distribution the exact location along the vertical axis is really a little arbitrary and subjective. Along the forward direction it might be

expected that the center of mass is a little nearer the front wheels than the rear wheels because of the mass of the front-located engine. Thinking about these factors allows the center of mass to be tweaked along the vertical and forward directions.

Tweaking the center of mass is really all about making incremental changes that tune the handling towards a desired goal. Moving the center of mass forwards should help cornering because more load is distributed to the front tires. However, this comes at the expense of reduced load on the rear tires, meaning that the car might turn more quickly only to spin out because the rear tires lose grip more quickly. Small changes followed by tests on the handling are required.

When setting the center of mass it is important to bear in mind that the suspension sprung mass values might require simultaneous updating. If the center of mass moves nearer the front this means that more mass is supported by the front suspensions and less by the rear suspensions. This change needs to be reflected in a consistent way. It is possible to mathematically describe the relationship between center of mass and the mass split between the suspensions. However, the editing possibilities afforded by breaking this rigid link should allow more tweaking options.

Mass:

A typical car might have a mass of around 1500kg.

Introduction

The goal of the character controller SDK is to provide users with a default character controller built on top of the NVIDIA PhysX SDK. Roughly it has to support two things:

- Character control
- Character interactions

This covers a very high number of features, which can be implemented in numerous ways. The goal is not to implement all of them (which would be a daunting task), the goal is to give a default/sample implementation that people can use as a starting point. For example the character's bounding volume could in theory be anything, from a box to an inverted pyramid. We will not implement all of them. In our initial implementation we support two common bounding volumes: an AABB and a capsule.

Kinematic Character Controller

One might wonder why we did not use the physics engine directly to implement the character controller. Here is the story.

In the past games did not use 'real' physics engines. However they still used a character controller to move a player in a level. These games, such as Quake or even Doom, had a dedicated, customized piece of code to implement collision detection and response, which was often the only piece of physics in the whole game. It actually had little physics, but a lot of carefully tweaked values to provide a good feeling while controlling the player. The particular behavior it implemented is often called the 'collide and slide' algorithm, and it has been 'tweaked for more than a decade'. The result is that players expect to find the same well-known behavior in new games, and providing them with anything else is often dangerous. This is especially true if provided behavior is not as robust and stable as before. And this is exactly what happens if you use a typical physics engine directly, to control players.

In particular, here is a (non exhaustive) list of typical problems you run into when using a physics engine for character controllers:

- (lack of) continuous collision detection: Typical physics engines use discrete collision checks, leading to the notorious 'tunneling effect' that has plagued various commercial & non-commercial physics packages for years. This leads to three main problems:
 - the tunneling effect itself : if your character goes too fast it might tunnel through a wall
 - as a consequence, the maximum velocity of your character might be limited (hence also limiting the game play possibilities)
 - even if you do not tunnel, the character might jitter when pushed forward in a corner for example, because the engine

keeps moving it back and forth to slightly different positions.

- No direct control: a rigid body is typically controlled with impulses or forces. It is usually not possible to move it directly to its final position, you first have to convert the delta position vector to impulses/forces, apply them, and hope that the character will be where you wanted it to be as a result. Usually it does not work too well, in particular when the physics engine uses an imperfect linear solver.
- Trouble with friction: When the character is standing on a ramp, you do not want it to slide. You want infinite friction here. When the character is moving forward on that same ramp, you do not want it to slow down. You want no friction here. When the character is sliding against a wall, you do not want it to slow down either. You want no friction here as well. Usually it is either 0 or infinite. However the friction model might not be perfect, and what you actually get is very little friction (you can still feel the character slowing down) or a very-big-but-not-infinite one (the character slides very slowly on that ramp no matter how artificially big the friction parameters are). The conflicting requirements for ramps also mean that usually there is simply no way to perfectly model desired behavior.
- Trouble with restitution: You should avoid restitution. When the character moves fast and collides with a wall, you do not want it to bounce away from it. When the character falls from a height and lands on the ground, flexing his legs, you definitely do not want any bounce to happen. But once again, even when the restitution is exactly zero, you can nonetheless sometimes get a small bump. This is not only related to the imperfect nature of the linear solver, it also has to do with how typical penetration-depth-based engines recover from overlap situations, sometimes applying excessive forces that separate the objects too much.
- Undesired jumps: You often want a character to stick to the ground, no matter what the physical behavior should be. For example characters

in action games tend to move fast, at unrealistic speeds. When they reach the top of a ramp, the physics engine often makes them jump a bit, in the same way a fast car would jump in the streets of San Francisco. But that is often not what you want: you want the character to stick to the ground regardless of its current velocity. This is sometimes implemented using fixed joints, which is an excessively complex solution to a very simple problem that can be solved without the complexity of a physics engine.

- Undesired rotations: Finally, a character is always standing up and never rotating. However physics engines often have poor support for that sort of constraints, and a great deal of effort is often put into preventing a capsule around the character from falling (it should always stand up on its tip). This is again often implemented using artificial 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 physics engine's features simply to emulate what's otherwise a much less complex piece of custom code. It is natural to instead keep using that simple piece of custom code.

Creating a character controller

You first have to decide what bounding volume you want around your characters. At the time of writing, only boxes (*PxBoxController*) and capsules (*PxCapsuleController*) are supported.

Then you need to create a controller manager somewhere in your app. You only need one of them; it will keep track of all created controllers and allow your character to interact with other characters created by the same manager. You can create this manager using the *PxCreateControllerManager* function:

```
PxFoundation* foundation;           // Previously created Foundatio
PxControllerManager* manager = PxCreateControllerManager(*found
```

Then you create one controller for each movable character in the game. This is done like this, for a capsule controller:

```
PxPhysics& sdk; // Previously created PxPhysics object
PxScene* scene; // Previously created scene
PxCapsuleControllerDesc desc;
<fill the descriptor here>
PxController* c = manager->createController(sdk, scene, desc);
```

Updating a character controller

Each frame, move your characters using the following function:

```
PxU32 collisionFlags = PxController::move(const PxVec3& disp, P
```

disp is the displacement vector for current frame. It is typically a combination of vertical motion due to gravity and lateral motion when your character is moving. Note that this is a displacement vector, i.e. a first order control. This is not an impulse vector (2nd order control) or a force (3rd order control). Please also note that users are responsible for applying gravity to characters here.

minDist is a minimal length used to stop the recursive displacement algorithm early when remaining distance to travel goes below this limit.

elapsedTime is the amount of time that passed since the last call to the move function.

filters are filtering parameters similar to the ones used in the SDK. These allow customization of filtering and control what the character is colliding with.

obstacles are optional, additional obstacles that the character should collide with. Those objects are fully controlled by users and do not need to have counterpart SDK objects.

collisionFlags is a bit mask returned to users to define collision events that happened during the move. This is a combination of PxControllerFlag flags. It can be used to trigger various character animations. For example your character might be falling while playing a falling idle animation, and you might start the land animation as soon as PxControllerFlag::eCOLLISION_DOWN is returned.

Obstacle objects

Sometimes it is convenient to create additional obstacles for the CCT to collide with, without creating an actual SDK object. This is useful in a number of situations. For example:

- the obstacles might only exist for a couple of frames, in which case creating and deleting SDK objects is not always efficient.
- the obstacles might only exist for stopping the characters, not the SDK's dynamic objects. This would be for example invisible walls around geometry, that only the characters should collide with. In this case it may not be very efficient to create the invisible walls as SDK objects, since their interactions would then have to be filtered out for everything except the characters. It is probably more efficient to create those additional invisible walls as external obstacles, that only characters can interact with.
- the obstacles might be dynamic and updated with a variable timestep, while the SDK uses a fixed timestep. This could be for example a moving platform on which the characters can stand.

At the time of writing the character controller supports box and capsule *PxObstacle* objects, namely *PxBoxObstacle* and *PxCapsuleObstacle*. To create those, first create a *PxObstacleContext* object using the following function:

```
PxObstacleContext* PxControllerManager::createObstacleContext()
```

Then manage obstacles with:

```
ObstacleHandle PxObstacleContext::addObstacle(const PxObstacle&  
bool PxObstacleContext::removeObstacle(ObstacleHandle handle)  
bool PxObstacleContext::updateObstacle(ObstacleHandle handle, c
```

Typically *updateObstacle* is called right before the controllers' *move* calls.

Graphics update

Each frame, you need to keep your graphics object in sync with the position of the character controller. You can access a controller's position using:

```
const PxExtendedVec3& PxController::getPosition() const;
```

This function returns the position from the center of the collision shape, since this is what is used internally both within the PhysX SDK and by usual graphics APIs. So if you just use this position in your world matrix and pass this to the renderer, the character should collide and slide smoothly against the world. This is illustrated in `SampleBridges`. Note that the position uses double-accuracy, to make the CCT module work well with large worlds. Also note that a controller never rotates so you can only access its position.

Alternative helper functions are provided if you need to work using the character's bottom position, a.k.a. the foot position:

```
const PxExtendedVec3& PxController::getFootPosition() const;  
bool PxController::setFootPosition(const PxExtendedVec3& positi
```

Character Volume

The character uses a bounding volume that is independent from already existing shapes in the SDK. That way we are free to implement a dedicated collision volume for the character controller (e.g. an ellipsoid), even if the corresponding physics shape does not exist.

We currently support two different shapes around the character:

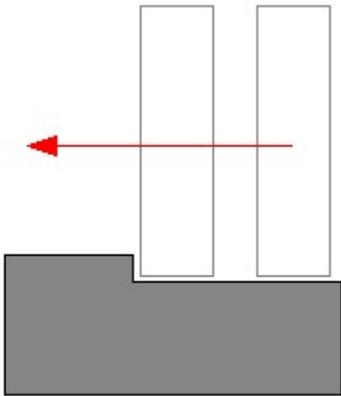
- An AABB, defined by a position and an extents vector. The AABB does not rotate. It always has a fixed rotation even when the player is (visually) rotating. This avoids getting stuck in places too tight to let the AABB rotate, for example.
- A capsule, defined by a position, a height and a radius. The capsule has a better behavior when climbing stairs for example. However it might be slightly more expensive in terms of CPU time.

Note: In versions prior to 2.3 there was an `PxSphereController`, this has been removed since the `PxCapsuleController` is more robust and provides the same functionality (zero length capsule).

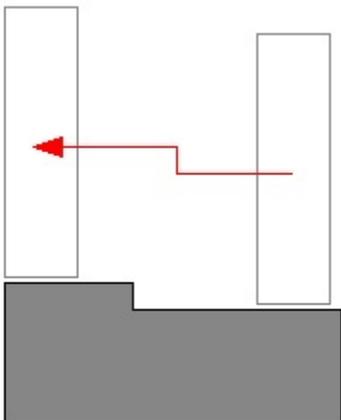
A small skin is maintained around the character's volume, to avoid numerical issues that would otherwise happen if we let it touch other shapes. The size of this skin is user-defined. If you render the character's volume for debug purpose, remember to expand the volume by the size of this skin to get accurate debug visualization. This skin is defined in `PxControllerDesc::contactOffset` and later available through the `PxController::getContactOffset()` function.

Auto Stepping

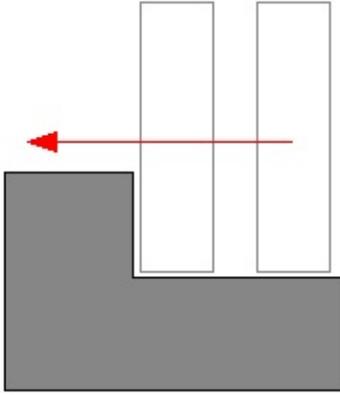
Without auto-stepping it is easy for a box-controlled character to get stuck against slight elevations of the ground mesh. In the following picture the small step would stop the character completely. It feels unnatural because in the real world you would just cross this small obstacle without thinking about it.



This is what auto-stepping enables us to do. Without any intervention from the player (i.e. without him thinking about it) the box correctly steps above the minor obstacle.



However if the obstacle is too big, i.e. its height is greater than the `stepOffset` parameter, the controller can not climb automatically, and the character gets stuck (correctly this time):



'Climbing' (over this bigger obstacle, for example) may also be implemented in the future, as an extension of auto-stepping. The step offset is defined in `PxControllerDesc::stepOffset` and later available through the `PxController::getStepOffset()` function.

In order to implement the auto-stepping feature, the SDK needs to know about your 'up' vector. The up vector is defined in `PxControllerDesc::upDirection` and later available through the `PxController::getUpDirection()` function.

Walkable Parts

By default the character can move anywhere. This is not what you usually want. In particular, you often want to disallow walking on polygons whose slope is high. The SDK can do this automatically thanks to a user-defined slope limit. All polygons whose slope is higher than the limit slope will be marked as non walk-able, and the SDK will not let characters go there.

In the future it may be possible to tag each triangle of a mesh as walk-able or not. (This is not implemented yet)

Two modes are available to select what happens when touching a non walk-able part. The desired mode is selected with the *PxControllerDesc::nonWalkableMode* enum. *PxCCTNonWalkableMode::ePREVENT_CLIMBING* prevents the character from moving up a slope, but does not move the character otherwise. *PxCCTNonWalkableMode::eFORCE_SLIDING* forces characters to slide down non walk-able slopes.

The slope limit is defined in *PxControllerDesc::slopeLimit* and later available through the *PxController::getSlopeLimit()* function.

The limit is expressed as the cosine of desired limit angle. For example this uses a slope limit of 45 degrees:

```
slopeLimit = cosf(PxMath::degToRad(45.0f));
```

If you use `slopeLimit = 0.0f`, the feature is automatically disabled (i.e. you can go anywhere).

This feature is not always needed. A common strategy is to disable it and place invisible walls in the level, to restrict player's movements. The character module can also create those walls for you, if *PxControllerDesc::invisibleWallHeight* is non-zero. In this case the library creates those extra triangles on the fly, and that parameter controls their

height (extruded in the user-defined up direction). A common problem is that those invisible walls are only created when non-walkable triangles are found. It is possible for a jumping character to go over them, if its bounding volume is too small and does not collide with the non-walkable triangles below him. The *PxControllerDesc::maxJumpHeight* parameter addresses this issue, by extending the size of the bounding volume downward. That way all potentially non-walkable triangles are properly returned by the collision queries, and invisible walls are properly created - preventing the character from jumping on them.

The slope limit is ignored if the touched shape is attached to a dynamic or kinematic rigid body. Further, the slope limit is ignored if the touched shape is a sphere or capsule attached to a static body. Extraction of a contact normal from heightfields, triangle meshes, convex meshes and boxes is more readily supported than with spheres and capsules so these shape types are all involved in the slope limit calculations, provided they are attached to a static body.

Volume Update

Sometimes it is useful to change the size of the character's volume at runtime. For example if your character can crouch, you might want to reduce the height of its bounding volume so that it can then move to places he could not reach otherwise.

The character library supports runtime volume updates. However volumes are directly modified without any extra tests, so it might happen that the resulting volume overlaps some geometry nearby. To avoid this you must first use the already existing 'placement API' from the SDK, to check that the volume of space you want to occupy is actually empty. Only then can you increase the size of the character's volume.

Related functions for the AABB:

```
bool PxBBoxController::setHalfHeight(PxF32 halfHeight)
bool PxBBoxController::setHalfSideExtent(PxF32 halfSideExtent)
bool PxBBoxController::setHalfForwardExtent(PxF32 halfForwardExt
```

Related functions for the capsule:

```
bool PxCapsuleController::setRadius(PxF32 radius) = 0;
bool PxCapsuleController::setHeight(PxF32 height) = 0;
```

Related 'placement API' function:

```
bool PxScene::overlapAny(...) = 0;
```

It is important to keep in mind that changing the size of a controller using those functions does not actually change its position. So if your character is standing on the ground (touching it), and you suddenly reduce its height without updating its position, the character will end up levitating above the ground for a few frames until gravity makes it fall and touch the ground again. This happens because the controllers positions are located at the

center of the shapes, rather than the bottom. Thus, to modify a controller's height and preserve its bottom position, one must change both the height and position of a controller. The following helper function does that automatically:

```
void PxController::resize(PxF32 height) = 0;
```

Updating the character's volume at runtime to implement a 'crouch' motion is illustrated in `SampleNorthPole`.

Hit callback

You can define a *PxUserControllerHitReport* callback that will let you retrieve some information about controller's evolution. In particular, it is possible to get called when the character hits a shape, another character, or a user-defined obstacle object.

When the character hits a shape, the *onShapeHit* callback is called - for both static and dynamic shapes. Various impact parameters are sent to the callback, and they can then be used in your application to do various things like playing sounds, rendering trails, applying forces, and so on.

When the character hits another character, i.e. another object controlled by a character controller, the *onControllerHit* callback is called. This happens when the player collides with an NPC, for example.

When the character hits a user-defined obstacle, the *onObstacleHit* callback is called.

Behavior callback

You can define a *PxControllerBehaviorCallback* callback that will let you customize the character's behavior after touching a *PxShape*, a *PxController*, or a *PxObstacle*. This is done using the following functions:

```
PxU32 PxControllerBehaviorCallback::getBehaviorFlags(const PxSh  
PxU32 PxControllerBehaviorCallback::getBehaviorFlags(const PxCo  
PxU32 PxControllerBehaviorCallback::getBehaviorFlags(const PxOb
```

Returned `PxU32` should be a combination of *PxControllerBehaviorFlag*. At the time of writing the following flags are supported:

PxControllerBehaviorFlag::eCCT_CAN_RIDE_ON_OBJECT defines if the character can effectively travel with the object it is standing on. For example a character standing on a dynamic bridge should follow the motion of the *PxShape* it is standing on. But it should not be the case if the character stands on, say a *PxShape* bottle rolling on the ground.

PxControllerBehaviorFlag::eCCT_SLIDE defines if the character should slide or not when standing on the object. This can be used as an alternative to the previously discussed slope limit feature, to define non walk-able objects rather than non-walkable parts. It can also be used to make a capsule character fall off a platform's edge automatically, when the center of the capsule crosses the platform's edge.

Character interactions

It is tempting to let the physics engine push objects by applying forces at contact points. However it is often not good for game play. The bounding volume around characters are artificial (boxes, spheres, etc) and you do not want the pushing effect to change when you switch from a box controller to a capsule controller. You usually want a pushing effect dictated by game play. So it might actually be better to apply artificial forces to objects in the *onShapeHit* callback. It is also difficult to push a box forward with a capsule: since you never hit the box exactly in the middle, applied force tends to rotate it - even if all you want is to push it in a straight line. Thus the recommended approach so far is to use custom code inside the callback to push away dynamic objects.

Hidden kinematic Actors

The character controller library creates a kinematic actor under the hood, for each controlled character. The user should be aware of this, since the total number of actors in the scene is more than they created, and they might receive unknown actors from the scene when doing collision queries.

When you invoke the *move* function to move a `PxController`, the underlying hidden kinematic `PxActor` is also updated. The kinematic actors can be retrieved using the following function:

```
PxRigidBody* PxController::getActor() const;
```

Time Stepping

Actors used internally by the character controller library follow the same rules as any other PhysX objects. In particular, they are updated using fixed or variable timesteps. This can be troublesome because the PxController objects are otherwise typically updated using variable time steps (usually using the elapsed time between two rendering frames). So the PxController objects are not always perfectly in sync with their kinematic actors when using fixed time steps.

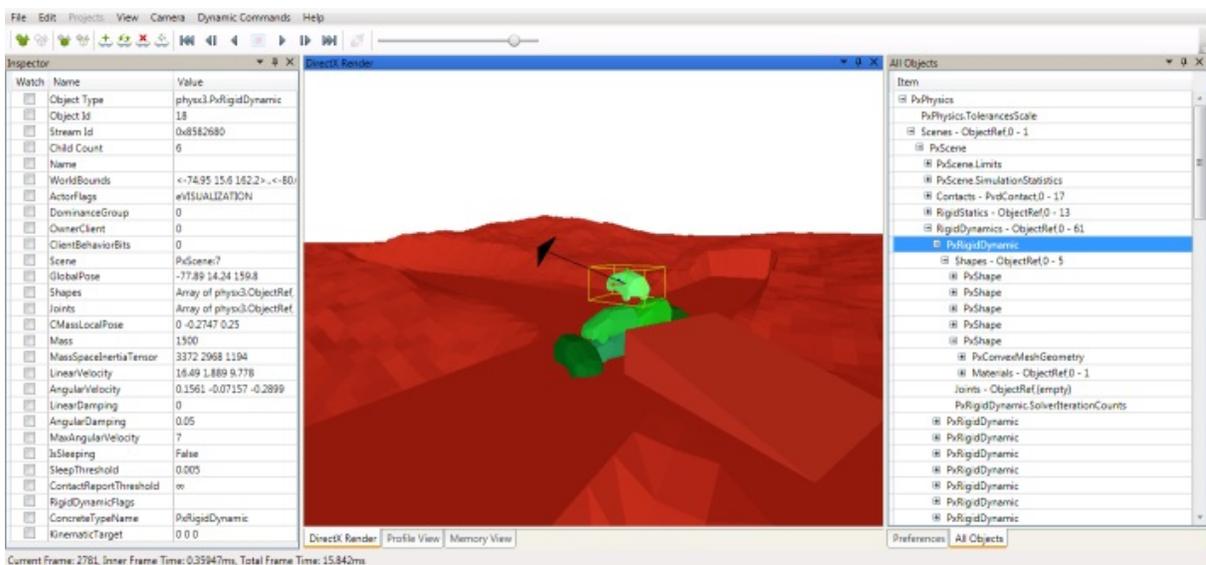
Invalidating internal caches

The character controller library caches the geometry around each character, in order to speed up collision queries. In PhysX 3.3 and above, those caches should be automatically invalidated when a cached object gets updated. However it is also possible to manually flush those caches using the following function:

```
void PxController::invalidateCache();
```

It might be necessary do to so when changing the filtering parameters between calls to `PxController::move()`.

PVD



The PhysX Visual Debugger (PVD) provides a graphical view of the PhysX scene and includes various tools to inspect and visualize variables of every PhysX object. Additionally it can also record and visualize memory and timing data.

PVD can be downloaded from: <http://developer.nvidia.com/physx-visual-debugger>

Questions regarding the usage of the GUI should all be answered by its detailed built-in help.

Basic Setup (SDK Side)

PVD integration is enabled in the debug, checked and profiling configurations of the SDK. In order to reduce memory footprint and code size, it is not enabled in the release configuration.

The SDK outputs the PVD debugging data in form of a stream. PVD supports reading the stream either from a TCP/IP network socket or from a file.

Network Setup

Streaming to TCP/IP is supported on almost all platforms, and is usually the most convenient way to collect PVD data. In this mode the stream can be watched in real-time, depending only on network speed and scene complexity. In network mode PVD acts as a TCP/IP server and must therefore be launched before the SDK tries to connect to it. The default listening port is 5425:

```
// check if PvdConnection manager is available on this platform
if(mPhysics->getPvdConnectionManager() == NULL)
    return;

// setup connection parameters
const char*    pvd_host_ip = "127.0.0.1"; // IP of the PC whi
int           port         = 5425;       // TCP port to conn
unsigned int   timeout     = 100;       // timeout in milli
                                                    // consoles and rem
PxVisualDebuggerConnectionFlags connectionFlags = PxVisualDebug

// and now try to connect
PVD::PvdConnection* theConnection = PxVisualDebuggerExt::create
    pvd_host_ip, port, timeout, connectionFlags));

// remember to release the connection by manual in the end
if (theConnection)
    theConnection->release();
```



File Setup

Streaming to file is an alternative to network streams. This is the recommended fall-back in case your platform or system setup does not support a network connection to PVD. File streams are often faster than network sockets and therefore a good alternative if performance is more important than real-time viewing. Streams stored as files can be loaded by drag&drop or over the File->Load menu in PVD:

```
// check if PvdConnection manager is available on this platform
if(mPhysics->getPvdConnectionManager() == NULL)
    return;

// setup connection parameters
const char* filename = "C:\\PvdCapture.pxd2"; // filename
PxVisualDebuggerConnectionFlags connectionFlags = PxVisualDebug

// and now try to connect
PVD::PvdConnection* theConnection = PxVisualDebuggerExt::create
    filename, connectionFlags);

// remember to release the connection by manual in the end
if (theConnection)
    theConnection->release();
```



Advanced Setup

Connection Flags

To optimize the stream size we provide flags to enable specific features. This has both influence on PVD's and the SDK's performance:

- **PxVisualDebuggerConnectionFlag::Debug:** Transfer all debug data to visualize and inspect objects. This flag has usually the biggest impact on the stream's size.
- **PxVisualDebuggerConnectionFlag::Profile:** Transfer timing information of various profiling zones in our SDK.
- **PxVisualDebuggerConnectionFlag::Memory:** Transfer memory usage data of our SDK.

Setup to transfer only profiling data over network:

```
PVD::PvdConnection* theConnection = PxVisualDebuggerExt::create  
    pvd_host_ip, port, timeout, PxVisualDebuggerConnectionFlag:
```

Visualizing Externals and Extended Data

Joints are implemented as an extension to the SDK constraints and therefore need special handling to get transmitted to PVD. Both joint and contact data can increase the stream size significantly. Visualizing it in PVD is therefore disabled by default. To enable them use following API calls:

```
mPhysics->getVisualDebugger()->setVisualizeConstraints(true);  
mPhysics->getVisualDebugger()->setVisualDebuggerFlag(PxVisualDe
```

Connection Handler

Implement the PvdConnectionHandler interface if your application needs to

react upon connection or disconnection from PVD, or if you plan to send custom PVD events from your application. It is recommended to toggle the contact and constraint visualization in the onPvdConnected/onPvdDisconnected callbacks to avoid potential memory and compute overhead in the SDK. The use of PvdConnectionHandler is demonstrated in our samples and the external joint implementation:

```
// derive from PvdConnectionHandler
struct MyConnectionHandler : public physx::debugger::comm::PvdC
{
    virtual void onPvdSendClassDescriptions( physx::debugger::c
    {
        // send your custom PVD class descriptions from here
        // this then allows PVD to correctly identify and repre
        // custom data that is sent from your application to a
        // example in JointConnectionHandler
    }
    virtual void onPvdConnected( physx::debugger::comm::PvdConn
    {
        // do something when successfully connected
        // e.g. enable contact and constraint visualization
    }
    virtual void onPvdDisconnected( physx::debugger::comm::PvdC
    {
        // handle disconnection
        // e.g. disable contact and constraint visualization
    }
};

// register custom handler
MyConnectionHandler myConnectionHandler;
if(mPhysics->getPvdConnectionManager())
    mPhysics->getPvdConnectionManager()->addHandler(&myConnecti
```

This guide describes how to upgrade applications that have an integration of PhysX 2.x to using PhysX 3.x.

Changed Actor Hierarchy

In PhysX 2, PhysX provided one actor class, and you were able to call any method on objects of this class even if that did not make sense. For example, you were able to call `isSleeping()` on static actors which do not have any sleep logic. In PhysX 3, we have a hierarchy of Actor classes, with each subclass only providing methods that truly apply to the specific subclass. This may mean that you have to change your code so that it either holds Actor references of the proper subtype, or you cast to the proper type before making certain calls.

Collision Filtering

In PhysX 2, we had multiple fixed function, custom mechanisms for configuring if a pair of shapes desires collisions to be detected. An example was collision groups. You were able to assign each shape to a fixed number of groups, and then set if a particular pair of groups should collide.

We found that for some people this approach was not flexible enough. In PhysX 3, the user is able to write his own program code to implement custom filtering. This is like a callback function from the engine, with the limitation that arbitrary memory may not be accessed. We made this restriction so that the filtering code can be executed on PS3 SPU's or on GPU's with optimal performance. If performance is not a top concern for users, they can also opt to use conventional callbacks (`PxSimulationFilterCallback`).

When migrating PhysX 2 code, note that we provide the class `PxDefaultSimulationFilterShader` in PhysX 3, which emulates a portion of PhysX 2 filtering behavior. Start by checking if this class is sufficient. As this is an extension class, the source code is available and may be extended or customized.

To migrate your fixed function PhysX 2 filtering code on your own, you need to be aware of its exact behavior and implement it as a callback or shader. Let us look at the precise 2.8 mechanisms and make some recommendations for porting:

```
virtual void NxScene::setShapePairFlags ( NxShape & shapeA,
    NxShape & shapeB,
    NxU32  nxContactPairFlag           //0 or NX_IGNORE_PAIR
)

virtual void NxScene::setActorPairFlags ( NxActor & actorA,
    NxActor & actorB,
    NxU32  nxContactPairFlag
)
```

The first function stored explicit shape pairs in a hash, and a lookup returned the bit indicating to filter or not. The second did the same for actor pairs. Because of the arbitrary size of the pair hash, implementing this mechanism as a shader with fixed memory is difficult in practice, but implementing as a callback should be trivial using a data structure such as the STL hash_map where Key is a struct holding the two pointers and Data is the bit flag.

Another scheme provided by PhysX 2 were collision groups:

```
virtual void  NxShape::setGroup (NxCollisionGroup collisionGroup)
virtual void  NxScene::setGroupCollisionFlag (  NxCollisionGroup
    NxCollisionGroup  group2,
    bool  enable
    )
```

This approach let the user assign shapes to one of 32 collision groups, and then let each pair of groups be assigned a boolean pair flag. This approach lends itself better to a shader based implementation. To do this, you should reserve a word of each shape's filterData (say word0) to hold the group index, and assign this as before. Next, define a matrix to hold the group pair bits, and a function to set it:

```
NxU32 groupCollisionFlags[32];

//init all group pairs to true:
for (unsigned i = 0; i < 32; i ++ )
    groupCollisionFlags[i] = 0xffffffff;

void setU32CollisionFlag(NxU32 groups1, NxU32 groups2, bool enable)
{
    NX_ASSERT(groups1 < 32 && groups2 < 32);
    if (enable)
    {
        //be symmetric:
        groupCollisionFlags[groups1] |= (1 << groups2);
        groupCollisionFlags[groups2] |= (1 << groups1);
    }
    else
```

```

        {
            groupCollisionFlags[groups1] &= ~(1 << groups2)
            groupCollisionFlags[groups2] &= ~(1 << groups1)
        }
    }

```

Unfortunately it is not possible to change this state after the scene is created. This is because if the matrix could change during simulation, it would force an arbitrary amount of existing contact pairs to be refiltered. In a large simulation, this could be an unacceptable amount of computation. Therefore the matrix must be initialized to its final state before the scene is created, like this:

```

PxSceneDesc desc;
...
desc.filterShaderData = groupCollisionFlags;
desc.filterShaderDataSize = 32 * sizeof(PxU32);
scene = sdk.createScene(desc);

```

Finally, you need to code the filter shader to access this data:

```

PxFilterFlags FilterShader(
    PxFilterObjectAttributes attributes0, PxFilterD
    PxFilterObjectAttributes attributes1, PxFilterD
    PxPairFlags& pairFlags, const void* constantBlo
{
    // let triggers through, and do any other prefi
    if(PxFilterObjectIsTrigger(attributes0) || PxFi
    {
        pairFlags = PxPairFlag::eTRIGGE
        return PxFilterFlag::eDEFAULT;
    }
    // generate contacts for all that were not filt
    pairFlags = PxPairFlag::eCONTACT_DEFAULT;

    PxU32 ShapeGroup0 = filterData0.word0 & 31;
    PxU32 ShapeGroup1 = filterData1.word0 & 31;
    PxU32* groupCollisionFlags = (PxU32*)constantBl

    if ((groupCollisionFlags[ShapeGroup0] & (1 << S
        return PxFilterFlag::eSUPPRESS;

```

```
else  
    return PxFilterFlag::eDEFAULT;  
}
```

Material Indexes

PhysX 2 used so-called material indexes for stored materials. Material indices are not supported in PhysX 3.0, you'll have to use the object directly. This should be a rather trivial change, unless you are storing a lot of material indices and you want to avoid storing a full reference. In this case you should create your own index array data structure from which you can fetch the pointer before passing it into the API.

Continuous Collision Detection

PhysX 2 uses CCD skeleton meshes for CCD. PhysX 3 no longer needs this data so all skeleton related code can simply be removed.

Pose Description

In PhysX 2 pose was described using a simple matrix. Now the only way to describe a pose is to prepare a `PxTransform` structure that contains a `PxVec3` for translation and a `PxQuat` for rotation. If the user code uses matrices natively, you will need to convert the matrices into a translation and a rotation component, and then convert the rotation matrix into a quaternion. See the class `PxMat33` for helpers to facilitate this conversion, as well as the inverse.

Shape Description

PhysX 2 uses special geometry descriptor to set some required shape parameters (one descriptor for every type of shape, e.g. `NxBoxShapeDesc`). In PhysX 3, descriptors are no longer used - you should create a required geometry class object (e.g. `PxBoxGeometry`) and pass it to `PxShape` using a `setGeometry` method. This means that the application shape creation logic should be slightly modified. In most cases the new approach is more flexible and needs less code to implement.

In PhysX 2, objects that have different geometry (Box, Capsule, and Sphere) were objects of different classes - now they are all `PxShape` objects. That means that if you want to figure out which geometry is assigned to some object you should call `getGeometryType` method. That means that the old way to have every geometry type classes inherited from different shapes classes (e.g. `NxBoxShape`) will not work now - you will have all geometry type classes inherited from `PxShape` class and should find out which geometry type it actually has internally.

Joins

The D6 driveType in PhysX 2 no longer exists in PhysX 3. Now drive for D6 is always spring-like: if you want position drive you set the 'spring' value non-zero, if you want velocity drive you set the damping field non-zero, and if you set both you get a damped spring. Some specialized joints like NxJointDriveDesc, NxJointLimitSoftDesc (PhysX 2 names) now were moved to Extensions (see the extensions folder inside PhysX 3 include directory).

Also, if you have used the deleted NxSpringAndDamperEffector, you should now use a joint with a spring property.

Time Stepping

The PhysX 2 SDK supported substeps, such that the simulation would automatically subdivide arbitrary elapsed time values into a variable number of fixed size simulation steps. This functionality is no longer provided in the SDK because many users found the precise logic too opaque and had difficulty integrating it with real world applications to get the behaviors they needed. Instead, we only provide the simulate code that used to be the logic for each substep. It is now the responsibility of the user to choose a stable step size, and to subdivide arbitrary 'wall clock' elapsed times into multiple such fixed size simulation steps, and then call `PxScene::simulate()` one or more times each frame with this step size. Because the user now owns the control logic, they are able to tweak all of the simulation control code, such as force application.

In PhysX 2 it was legal to call simulate with a timestep of zero to force the execution of various side-effects of simulation. PhysX 3 neither requires nor supports this.

Scene Queries

Working with functions that return a buffer of objects (e.g. `raycastMultiple`) has changed. In PhysX 3, functions that returns multiple objects want to get a pre-allocated buffer and buffer size as a parameter. You do not know how many elements should be needed when you are making a call, so you will need to make a guess. If the number of elements in the buffer is not enough, you will be informed about it and should re-allocate your buffer.

It is a good performance optimization idea to use a buffer with a reasonable size first, try with this buffer and if -1 is returned, resize the buffer and try again. But most of the time the initial buffer will be large enough. Choosing a reasonable start size is not always easy ... here are some ideas: You could profile the app and see the sizes that come up in practice. Also, whenever space runs out, you could reallocate with double the size, and then keep this doubled buffer around for the next time queries are performed. Finally, you should add an upper bound on how large the buffer can get in total, and do some error reporting if this size is exceeded.

Raycasts

The interface for making raycasts was changed in PhysX 3. Now you should pass an origin (`PxVec3`) and a direction (`PxVec3`) instead of a `NxRay` that combined these fields in PhysX 2.

Overlaps

Routines like `overlapSphereShapes`, `overlapAABBShapes`, `overlapOBBShapes`, `overlapCapsuleShapes` are now all covered with `PxScene::overlapMultiple` (passing in a `PxSphereGeometry`, `PxBoxGeometry` or `PxCapsuleGeometry` as a first parameter).

Sweep Tests

The only migration problem that we have met is with capsule sweeps tests. PhysX 2 provides a `linearCapsuleSweep` that takes two points to define the capsule's two spherical ends. In PhysX 3 we have a general `sweepMultiple()` routine that takes a `PxGeometry` and an initial position as a `PxTransform`. The capsules defined as two points should be converted to initial transformation (`PxTransform`) that consists of `PxVec3` for position and `PxQuat` for rotation. The capsule's length is now along the x axis of this local frame.

Compartments

PhysX 2 scenes featured sub-scenes called compartments. Each compartment simulated rigid bodies, deformables or fluids. The compartments could be simulated in parallel and the scene contained some extra logic to permit objects from different compartments to interact with each other. Compartments were an afterthought to permit an SDK that was not designed for native parallelism, and did not provide native interoperability between separate simulation technologies. Both of these deficits were addressed from the ground up in PhysX 3, meaning that PhysX 3 scenes provide support for intra-scene parallelism and full object interaction across different simulation objects automatically without having to configure this manually.

There is one detail feature however which compartments provided for which we currently do not have an answer: Because interaction between objects in different compartments was relatively weak and done with exchange of external forces, it was possible to step each compartment with a different time step. It is not possible to step parts of a PhysX 3 scene with different time steps. A workaround could be to create multiple scenes and step them at different rates. This however will completely lose the interactions. The user would have to exchange forces between the scenes to mimic the weak interactions provided between compartments in PhysX 2. This however is not a reliable approach and we do not recommend it. A better though eventually more costly approach is to simulate the entire scene at the smallest time step size of all the former compartments. This approach is not as unattractive as it seems, as the simulation code in PhysX 3 often has far better performance than it did in PhysX 2.

Deformables

PhysX 2 featured a quite full featured and very flexible deformable mesh simulation component: It was able to simulate environmental cloth such as banners or flags, clothing, soft bodies, inflatable balloons, and plastic deformation of rigid metal. Unfortunately this great flexibility meant that the implementation code was quite complex and the performance optimization for low end consoles a daunting challenge. We realized that that the flexibility of the solution prevented us from delivering a truly excellent solution in performance and behavior in the relatively few specific areas of application that our users were typically using. In PhysX 3 we decided to shelve the general deformable simulation code in favor of focusing only on a clothing solution which we identified as the number one application. PhysX 3 clothing differs primarily in that it sacrifices environmental interaction in favor of high speed simulation. In PhysX 3 dot releases, we will be incrementally adding back more features such as environmental interactions while making certain that performance does not suffer. For the time being, for many applications of PhysX 2 deformables there simply is no upgrade path in PhysX 3.

-
- This guide highlights all significant parts of the API that have changed in the last dot release. An application with a working integration of the older version of PhysX should be able to easily migrate to the newer version by following these pointers.

Foundation and Common

- PxCreatePlane used to take an (n,d) pair, now it takes a PxPlane. The first three elements are the same, but the d parameter must be negated.
- PxStream has been split into PxOutputStream and PxInputStream, and these must be implemented anew by the user if APIs are used that need them.
- There are new helpers PxShortestRotation, and PxDiagonalize.
- The serialization API (PxCollection, PxSerializable) has seen major changes with work still ongoing. As this API is not yet finalized we cannot provide migration information yet.

Geometry

We added a number of new capabilities to the geometry library, but existing code should not need to be adapted. New things include:

- PxTransformFromSegment
- PxConvexMesh::getLocalBounds
- PxTriangleMesh::getLocalBounds
- PxHeightField::modifySamples, getTriangleNormal
- PxMeshQuery
- PxTransformFromPlaneEquation, PxPlaneEquationFromTransform
- PxTriangle

Core PhysX

- The way to create the Physics SDK has changed. The `PxCreatePhysics` call now should be preceded by a call to `PxCreateFoundation`, which takes the allocator and error callback parameters that used to go into `createPhysics`. Instead `PxCreatePhysics` now takes a profile zone manager created using `PxProfileZoneManager::createProfileZoneManager`. It is important to create and pass this profile zone manager in order to set up the Visual Remote Debugger for profiling.
- Perhaps the most fundamental change for the API is the rewrite of the type casting system. The `PxActor::is(PxActorType)` style functions are replaced with `isKindOf(const char * typeName)`, `const char* getConcreteTypeName()`, `is<Class>()` template functions (e.g. `is<PxCloth>()`), and specially named inline functions (e.g. `PxActor::isCloth()`).
- `PxAggregate::getMaxSize` has been renamed to `getMaxNbActors`, and `getCurrentSize` has been renamed `getNbActors` for clarity.
- The `PxBatchQuery` raycast parameters have been reordered.
- There is a new user implemented interface class `PxConstraintVisualizer`, and the `PxConstraintVisualize` callback function now takes one of those instead of a `PxRenderBuffer`.
- `PxContactPoint::featureIndex0/1` has been renamed `internalFaceIndex0/1`, and `PxContactSet::getFeatureIndex0/1` has been renamed `getInternalFaceIndex0/1`.
- The class `PxContactStreamIterator` has been deleted, and replaced by a completely rewritten contact query mechanism. It is best if the user reads up on the new API and implements it from scratch, it is quite dissimilar to the previous incarnation. See the class `PxSimulationEventCallback` and in particular `::onContact` to start to refactor code.
- The empty default implementations of event callbacks have been removed to prevent confusion. This means that users must provide

their own (blank) implementations of all events in a user class they subclass, even ones they are not interested in. Providing default blank implementations in the SDK exposed the user to hard to find bugs if we changed an event signature from one release to another.

- Users of continuous collision detection should note that `PxPairFlag::eSWEPT_CONTACT_GENERATION` and `PxPairFlag::eSWEPT_INTEGRATION_FULL` no longer exist. We have rewritten the CCD algorithm such that it no longer has its own contact generation, and it no longer supports angular sweeps. Hence use of `eSWEPT_CONTACT_GENERATION` should simply be removed, and `eSWEPT_INTEGRATION_FULL` should be replaced with `eSWEPT_INTEGRATION_LINEAR`.
- `RbPairStatsType::eSWEPT_CONTACT_PAIRS` removed together with swept contact generation.
- We also stopped supporting anisotropic friction, so `PxMaterialFlag`, `PxCombineMode`, and all function calls of `PxMaterial` relating to anisotropic friction have been removed. Users should remove the corresponding calls.
- `PxPhysics::getMetaData` been changed into `PxGetSDKMetaData` to make it be an optionally linkable component.
- `PxPhysics::releaseUserReferences`, `releaseCollection` deprecated. Users should simply call `release()` on the appropriate object.
- `PxPhysics::addCollection` now takes a reference parameter instead of a pointer to indicate that a null pointer is not an option.
- The functions `PxPhysics::createParticleSystem` and `createParticleFluid` take parameters directly rather than descriptors.
- The functions `PxPhysics::createTriangleMesh`, `createConvexMesh`, and the stream version of `createClothFabric` now take a `PxInputStream` rather than a general `PxStream`, which has been removed. To migrate these function calls, the user must implement the new `PxInputStream` class.
- The `PxPhysics::createClothFabric` creation parameters for the non-stream version have changed a lot. The number `nbFibers` and `nbIndices` parameters have been removed because it always equals

nbSets (which is new). particleIndices has been renamed indices. restValues and nbRestValues are newly added. None of the parameters may be NULL.

- PxPhysics::getProfileZoneManager returns a pointer instead of a reference.
- PxRigidDynamic::moveKinematic has been renamed to setKinematicTarget.
- The prototype of PxGetFoundation has been moved from the Physics API to the Foundation API.
- PxScene::overlapAny has been removed. Use instead overlapMultiple with hitBufferSize = 1.
- The pruning structure options PxPruningStructure::eOCTREE and eQUADTREE have been removed, because they are now in general inferior in performance to the AABB_TREE options. The PxSceneDesc::maxBounds, upAxis, and subdivisionLevel properties were also removed as part of this because they were only used for these two pruning modes.
- We removed PxScene::getNbAttachments, getAttachments, this was part of the removed deformable simulation feature.
- PxSweepCache is deprecated in this release and will be removed in the next. We found that it provides no significant benefit.

There are also a number of new features:

- There is a new API for optional components: PxRegisterArticulations, PxRegisterPCM, PxRegisterHeightFields, PxCreateBasePhysics, PxGetSDKMetaData. This way some rarely used features can avoid getting linked into the executable on certain platforms.
- There is a new function PxGetPhysics.
- There are new functions PxScene::set/getTimestamp(). Saving and loading the time stamp makes the simulation be more deterministic.
- There are new PxSceneDesc::contactReportStreamBufferSize and PxScene::get/setNbContactDataBlocks,

`getMaxNbContactDataBlocksUsed` functions (and `PxSceneDesc::nbContactDataBlocks`, `maxNbContactDataBlocks`) for some advanced memory management possibilities.

- There are new flags `PxSceneFlag::` modes `eENABLE_ONE_DIRECTIONAL_FRICTION`, `eENABLE_TWO_DIRECTIONAL_FRICTION`, `eDISABLE_CONTACT_REPORT_BUFFER_RESIZE`, `eENABLE_PCM`, to enable a new friction model and a new contact generation algorithm, among other things.
- The new flag `PxSceneQueryFlag::eINITIAL_OVERLAP_KEEP` controls whether how scene query should behave when it comes to initial overlaps.
- `PxShape::getGeometry` is a more general way to retrieve shape geometry.

Cooking

- The biggest change to cooking is that the functions `PxCooking::cookTriangleMesh`, `cookConvexMesh`, and `cookClothFabric` now take the new class `PxOutputStream` not and not a `PxStream` as before. Users must implement a subclass of `PxOutputStream`. Note that we provide a default implementation in the form of `PxDefaultFileOutputStream`.
- The `ConvX` tool has been replaced by the new class `PxBinaryConverter` (see `PxCooking::createBinaryConverter`) function which works the same way to convert binary serialized files between platforms.
- The `PxCreateCooking` function now takes a foundation reference instead of a pointer to denote that a null pointer value is not valid.

Deformables

- The deformables feature that has appeared as deprecated in the early versions of PhysX 3.x has been removed. We decided to reboot deformables as the more streamlined and simplified Cloth feature.

Particles

- The only significant change to the particles API is that it no longer uses descriptors for creation. The classes `PxParticleBaseDesc`, and `PxParticleFluidDesc` have been deleted, and the create functions take the parameters immediately.
- There are also two new functions: `PxParticleBase::setParticleReadDataFlag` and `PxParticleFluid::setRestParticleDistance`.

Cloth

- The function `PxCloth::setParticles` can now also update the previous particle state. This lets the user easily define the velocity of the particles as the difference between the two positions. The cloth simulation really does not store velocities explicitly, but rather only two consecutive positions, hence this somewhat odd interface.
- `PxCloth::getCollisionData` now also takes a `planesBuffer`, and a `convexMaskBuffer`. These buffers retrieve additional collision information, but passing `NULL` should make it possible to ignore them.
- `PxClothFabric::scale/getRestlengths` have been renamed to `scale/getRestvalues`.
- Aside from this, there are a number of new features:
 - There is a new struct `PxClothParticleSeparationConstraint` (see also `PxCloth::get/setSeparationConstraints`, `getNbSeparationConstraints`). This is a new separations constraints feature that enforces each particle to stay inside a sphere.
 - The new function `PxCloth::clearInterpolation` lets the user easily set the effective velocity of particles to zero. This should be called whenever the character animation is discontinuous.
 - There are new `PxCloth::get/setParticleAccelerations`, `getNbParticleAccelerations` functions that effectively permit the user to apply forces to individual particles.
 - New `PxCloth::getNb/set/remove/addCollisionPlane(s)` and `PxCloth::getNb/remove/addCollisionConvex` permit the definition of convex shapes built from planes.
 - There are a bunch of new cloth properties to experiment with

such as `PxCloth::get/setInertiaScale`,
`PxCloth::get/setFrictionCoefficient`,
`PxCloth::get/setDragCoefficient`,
`PxCloth::get/setCollisionMassScale`.

- The `PxClothFabric` now exposed a number of new internal arrays with getter functions.
- The new `PxClothMeshVertFlag` and `PxClothMeshDesc::vertFlags` lets attached vertices be defined for the cooker from which it will try to generate zero stretch constraint chains. This feature is still experimental and these settings are not required for general vertex attachment.

Extensions

- We provide new default implementations for user implemented functionality in the form of `PxDefaultMemoryOutputStream`, `PxDefaultMemoryInputData`, `PxDefaultFileOutputStream`, and `PxDefaultFileInputData`.
- The class `PxDefaultSimulationFilterShader` was significantly extended with `set/PxGetGroupCollisionFlag`, `set/PxGetGroup`, `set/PxGetFilterOps`, `get/PxSetFilterBool`, `get/PxsetFilterConstants`, and `set/PxGetGroupsMask`, this should implement 2.8 style group filtering for users migrating from that old version.
- `PxDumpMetaData` now dumps to a stream, not to a file, for increased flexibility.
- New versions of `PxRigidBodyExt::updateMassAndInertia`, and `setMassAndUpdateInertia` were added that permit specifying individual shape densities, not just a uniform density. Old versions are still available for uniform density.
- The new helpers `PxCloneStatic`, `PxCloneDynamic`, and `PxScaleRigidActor` have been added.

Virtual Remote Debugger Interface

- The function `PxVisualDebuggerExt::connect` has been renamed `createConnection`.
- `PxVisualDebugger::updateCamera` now takes names for cameras directly and it is not necessary to allocate cameras. Unseen names passed to the function will automatically be registered.
- Added new functions `PxVisualDebugger::setVisualizeConstraints`, `isVisualizingConstraints` to let user determine if joints should be visualized in PVD.
- The function `PxVisualDebugger::setJointVisualizationScale` has been deleted. Joints can now be scaled in PVD using the Preferences -> Gizmo Scale setting.

Character Controller

- `PxBoxControllerDesc` representation changed from an extents vector to separate floats, which are hopefully more intuitive. Applicable accessors have been renamed.
- The default mode of the capsule controller was changed from `PxCapsuleClimbingMode::eCONSTRAINED` to `PxCapsuleClimbingMode::eEASY`. This is to be consistent with the default behavior of e.g. box controllers, which are blocked by obstacles greater than the `stepOffset`. Also, `stepOffsets` greater than the capsule controller's entire height are now forbidden.
- The `PxControllerDesc::upDirection` is now a vector rather than a 3 value enum. `PxCCTUpAxis` has been removed. This enables arbitrary up vectors for character controllers -- they no longer have to be aligned to the world axes. Applicable accessors have been updated.
- The various functions changing the height or radius of a controller do not automatically update the controller's position anymore, as some of them did on the past. There is now a new helper function, `PxController::resize()`, to do this.
- The function `PxController::move` now needs the user to pass an elapsed time parameter. This is needed to get proper interactions between the character (may be updated with variable timesteps) and kinematic platforms controlled by the SDK (updated with fixed timesteps). When the character jumps away from a moving kinematic platform, the character controller code needs to compute its velocity. It needs a correct elapsed time value to do that. If the controller being migrated never needs to jump away from moving kinematic platforms, this parameter is not used, and an arbitrary value may be passed.
- The function `PxController::move` also needs a `PxControllerFilters`, a new class. This stores all kinds of filtering information to filter out collisions with potentially colliding objects, including a filter callback. This class has a constructor to initialize itself to defaults, one can just pass a default object to get default behavior.

- Finally the function `PxController::move` now also takes an optional `PxObstacleContext`, which you can leave `NULL` to keep your prior behavior. This object could be used to define obstacles like invisible walls for the character controller that you do not need to add to the simulation for efficiency reasons.
- Some versions of PhysX 3.1 still shipped with a dynamic character controller which admittedly did not work as well as the kinematic controller, and there was a lot of overlap between the functionality. We decided to stop maintaining this code path and it was removed from 3.2. The great number of improvements to the kinematic character controller should make it possible for applications to migrate over to it.
- There is a new function `PxControllerManager::computeInteractions(elapsedTime)`. This is needed to properly resolve interactions between characters when they overlap (which can happen from time to time, for various reasons). One can call this pretty much at any time, once per frame. It needs a proper elapsed time between frames value here so that interactions are resolved in a way that do not depend on the frame rate. Nothing bad would happen if one passed a fixed time value, though. If there is only one character in the scene, it is guaranteed that characters will never overlap, then this function need not be called.

Meanwhile a great many new features have been added to the character controller:

- See the new `PxCCTNonWalkableMode` and `PxController::get/setNonWalkableMode()` for two options for how to handle non-walkable areas.
- See the new `PxControllerState`, and `PxController::getState(PxControllerState)` for basic state information like the currently touching shape, what the controller is standing on, etcetera.
- See the new `PxControllerStats` and `PxController::getStats(PxControllerStats)` for simulation statistics like the number of collision iterations performed.

- See the new new `PxControllerManager::createObstacleContext()` (and also `PxControllerObstacleHit` and `PxUserControllerHitReport::onObstacleHit`) for a way to collide with user defined obstacles outside of the simulation scene. This is covered in more detail on the character controller page of this manual.
- The class `PxControllersHit` now has contact point information between controllers (position, normal).
- We added `PxControllerDesc::density`, and `scaleCoeff` variables to let user have more control over kinematic actor that character controller creates.
- The new `PxControllerDesc::volumeGrowth` lets the user control caching of geometry to improve performance.
- The new `PxControllerBehaviorCallback` (see also `PxControllerDesc::behaviorCallback`) lets the user set behavior flags for object interaction to specify whether the controller should ride along on the object it stands on or slide along its surface.
- There is a new `PxControllerDesc::groupsBitmask` (and `PxController::get/setGroupsBitmask()`) to control collision filtering.
- Convenience functions like `PxController::getScene()` and the new `PxController::getFootPosition()` to retrieve foot position (bottom point of shape) are now available.
- There is the new `PxControllerDebugRenderFlags` (see also `PxControllerManager::setDebugRenderingFlags()`) to control what geometry to send to debug renderer, plus `PxControllerManager::getRenderBuffer()` to get debug render buffer.
- There is a new `PxExtendedVec3::toVec3()` to convert an extended precision position to a regular `PxVec3`.



Contents:

- **The Basics**
 - Building and Running the Samples on Windows
 - Build Settings
 - Library Redistribution
 - Build Configurations
 - The API
 - Multithreading
 - PhysX Visual Debugger
- **Startup and Shutdown**
 - The Allocator Callback
 - The Error Callback
 - Cooking
 - Extensions
 - Optional SDK Components
 - Windows delay load dll
 - Shutting Down
- **Scenes, Materials, and Actors**
 - The Scene
 - The Simulation Loop
 - Simulation Memory
 - Materials
 - Rigid Actors
 - Simulation Clients
- **Shapes and Geometries**
 - Shapes

- Simulation Shapes and Scene Query Shapes
- Trigger Shapes
- Kinematic triangle meshes (planes, heighfields)
- Geometries
 - Spheres
 - Capsules
 - Boxes
 - Planes
 - Convex Meshes
 - Height Fields
 - Triangle Meshes
- Mesh Scaling
- PxGeometryHolder
- Detaching Shapes
- Shape Vertex and Triangle Queries
 - Convex Meshes
 - Triangle Meshes
 - Height Fields
- Joints
 - Joint Basics
 - Visualization
 - Force Reporting
 - Breakage
 - Projection
 - Limits
 - Actuation
 - Fixed Joint
 - Spherical Joint
 - Revolute Joint
 - Prismatic Joint
 - Distance Joint
 - D6 Joint
 - Locking and Unlocking Axes
 - Limits

- Drives
 - Configuring Joints for Best Behavior
- Rigid Body Dynamics
 - Applying Forces and Torques
 - Gravity
 - Setting the Velocity
 - Kinematic Actors
 - Sleeping
 - Solver Accuracy
 - Fast Rotation
 - Mass Properties
 - Damping
- Data Access and Buffering
 - General Data Access Rules
 - Double Buffering
 - Events involving removed objects
 - Support
 - Memory Considerations
- Task Management
 - TaskManager
 - CpuDispatcher
 - CpuDispatcher Implementation Guidelines
 - BaseTask
 - Task
 - LightCpuTask
- GPU Resource Management
 - GpuTasks
 - CudaContextManager
 - GpuDispatcher
 - BlockingWait
 - Copy Engine Kernel
 - CUDA Profiling
- Scene Queries
 - Raycast queries

- Sweep Queries
 - Sweeps with Initial Intersection
- Overlap Queries
- Filtering
- Caching
- Batched queries
- Geometry Queries
 - Geometry Objects
 - PxGeometryQuery
 - PxMeshQuery
- Callbacks and Customization
 - Simulation Events
 - Collision Filtering
 - Contact Modification
 - Active Transforms
- Aggregates
 - Introduction
 - Creating an Aggregate
 - Populating an Aggregate
 - Releasing an Aggregate
 - Amortizing Insertion
- Serialization
 - Binary Serialization
 - Framework Classes
 - Serializing Objects
 - Memory Management
 - Traversing Collections
 - Partial Serialization
 - Serializing Everything
 - Object Names
 - Retargeting
 - API-level Serialization with RepX
 - Converting between RepX and Binary Serializable Collections

- Simulation Statistics
 - Interface
 - Usage
- Advanced Rigid Body Topics
 - Continuous Collision Detection
 - Articulations
 - Creating an Articulation
 - Articulation Joints
 - Driving an Articulation
 - Articulation Projection
 - Substepping
 - Using Completion Tasks
 - Synchronizing with Other Threads
 - Custom Constraints
- Persistent Contact Manifold
 - Introduction
 - Enabling the Persistent Contact Manifold
- Coulomb Friction
 - Introduction
 - Enabling The Coulomb Friction Model
- Particles
 - Introduction
 - Creating Particle Systems
 - Creating Particles
 - Releasing Particles
 - Index Pool Extension
 - Updating Particles
 - Reading Particles
 - Particle Drains
 - Particle Grid and Spatial Data Structure Overflow
 - Collision Filtering
 - GPU/CUDA Acceleration
 - Additional SampleParticles Information
- Cloth

- Introduction
- Creating Cloth Fabric
- Creating Cloth Collision Data
- Continuous Collision Detection
- Creating Cloth
- Simulation Overview
- Particle Integration
- Constraint Solving
- Virtual Particles
- Friction and Mass Scaling
- Motion Constraints
- Vehicles
 - Vehicle SDK Initialization
 - Vehicle Creation
 - Vehicle Actor
 - Tire Friction on Drivable Surfaces
 - Filtering
 - Vehicle Update
 - Tire Shaders
 - Telemetry
 - Vehicle State Queries
 - Vehicle Controls
 - 3-Wheeled, N-Wheeled Cars, and N-Wheeled Tanks
 - Tuning Guide
 - PxVehicleWheelData
 - PxVehicleWheelsSimData
 - PxVehicleSuspensionData
 - PxVehicleTireData
 - PxVehicleEngineData
 - PxVehicleGearsData
 - PxVehicleClutchData
 - PxVehicleAckermannGeometryData
 - PxVehicleTireLoadFilterData
 - PxVehicleDifferential4WData

- PxRigidDynamic
- Character Controllers
 - Introduction
 - Kinematic Character Controller
 - Creating a character controller
 - Updating a character controller
 - Obstacle objects
 - Graphics update
 - Character Volume
 - Auto Stepping
 - Walkable Parts
 - Volume Update
 - Hit callback
 - Behavior callback
 - Character interactions
 - Hidden kinematic Actors
 - Time Stepping
 - Invalidating internal caches
- PhysX Visual Debugger (PVD)
 - PVD
 - Basic Setup (SDK Side)
 - Network Setup
 - File Setup
 - Advanced Setup
 - Connection Flags
 - Visualizing Externals and Extended Data
 - Connection Handler
- Migration Guide From PhysX SDK 2.x to 3.x
 - Changed Actor Hierarchy
 - Collision Filtering
 - Material Indexes
 - Continuous Collision Detection
 - Pose Description
 - Shape Description

- Joints
- Time Stepping
- Scene Queries
- Raycasts
- Overlaps
- Sweep Tests
- Compartments
- Deformables
- Migrating from PhysX SDK 3.1 to 3.2
 - Foundation and Common
 - Geometry
 - Core PhysX
 - Cooking
 - Deformables
 - Particles
 - Cloth
 - Extensions
 - Virtual Remote Debugger Interface
 - Character Controller