Programmer's Guide

APEX™

November 2008

Table of Contents

Introduction

Motivation

The APEX SDK is designed to address three important issues facing game physics:

1. Significant programmer involvement is required.
2. Game physics content typically gets designed to the game's "min-spec" system.
3. Game engine performance limitations.

The first problem arises because the traditional interface to middleware physics is an API. It's designed for programmers. These API's need to be "general purpose", so they expose a very "low-level" interface: rigid bodies, shapes, joints, particles, impulses, and collisions. They are basically, a "toolbox" of all the primitive components of a physics simulation. This gives the game programmer a lot of control, but in return, it requires a lot of non-trivial work. This work requires broad physics programming experience, and is often not budgeted by game developers. This is similar, in many ways, to the specialized work of writing an efficient rendering engine on top of D3D. With graphics, artists don't directly create vertex buffer objects in the editor, but for physics that's what physics engine integration requires. Even when authoring tools (like plug-ins for max/maya) are made available to attempt to reduce the programmer involvement that's required, it's the same "low-level toolbox" that is exposed to the artists. If the artists want to create content at a higher level of abstraction – not using these primitive building blocks – then you're back to requiring a lot of programmer involvement again.

The second problem arises because there are huge performance differences between the consoles, and between different generations of PC CPUs, and GPUs. So, if game developers want their game to take advantage of higher end hardware, when it's available, to improve the quality, or scale of their game physics, custom authoring is required for each platform or hardware configuration. The result of this is that, in practice, only the "lowest common denominator" content gets created, and users don't benefit from better hardware.

The third problem arises because many game engines make the assumption that the world is largely static: that there are many static objects, but few dynamic objects Therefore, they tend to use very "heavy weight" data structures for their moving objects: For example: Using an "Actor" class for each and every crate, barrel, or piece of debris flying around the level. So even though the physics system might be able to handle very complex simulations, the overhead of the game engine (the scene graph, the AI, the rendering) makes it impossible, in practice, to use more than a few dozen objects in the level.

The APEX Solution

APEX addresses each of the above problems as follows:

1. **Significant programmer involvement is required to take a relatively abstract PhysX-SDK and create a lot of meaningful content.**

   APEX provides a high-level interface to artists and content developers. This reduces the need for programmer time, adds automatic physics behavior to familiar objects, and leverages multiple low-level PhysX-SDK features with a single easy-to-use authoring interface.

2. **Game physics content typically gets designed to the game's "min-spec" system.**

   APEX requires each functional module to provide one or more ways to "scale the content" when running on better-than-min-spec systems, and to do this without requiring a lot of extra work from the game developer (artist or programmer, but especially programmer).

3. **Game engine performance limitations.**

   APEX avoids many of the game engine bottlenecks by allowing the designer to identify the physics that is important to the game logic, and what can be sent directly to the renderer, bypassing the fully generic path through the game engine. It also allows the game engine to treat an APEX asset as a single game object, even though it may actually comprise many hundreds or even thousands of low-level physics components.

The APEX SDK comprises a set of C++ header files that define the SDK public API, a set of binary libraries that implement the functionality, as well as documentation of the API and of the functionality provided. Source code to the libraries may be provided to certain developers, according to business considerations. The APEX SDK also includes a set of tools (binary executables and DCC plugins) for content creation.

The APEX architecture is a modular one. Modules implement intuitive, specific purpose, high-level physics technology at a level of abstraction that is appropriate for content creators (artists and level designers). Modules typically manage multiple physics primitive elements (e.g.: rigid bodies, shapes, joints, etc..), and may manage multiple simulation types (e.g.: rigid body, fluid, cloth, soft-body).

Definitions

Term	Definition
Scalable Content	In the context of APEX, content is said to be scalable whenever it can adapt to running on a variety of systems with various degrees of physics simulation capability (e.g.: CPU, CPU+PPU, CPU+GPU, PS3, Xbox360), and demonstrate benefit from running on systems with increased physics simulation capability.
APEX Asset	An APEX asset is a data structure that is created at game authoring time, can be serialized to non-volatile storage, loaded by the APEX run-time, and used to create instances (called actors) in the scene.
APEX Authorable Object	An authorable object is a definition of an asset data structure, the specification of how it will be authored, the specification of the runtime functionality, and the software that implements all this. This software includes, at a minimum, an Asset Authoring class, an Asset class, and an Actor (or instance) class.
APEX Module	An APEX module is software component, consisting of a group of related APEX Authorable Objects. It includes a run-time component and an authoring component. The run-time component for each APEX Module is shipped as a separate library or DLL. The authoring component may be provided by multiple tools, and/or may be combined in a single tool with other authoring functionality. The specification of Modules is driven primarily by architecture and engineering, with input from marketing and product management.
APEX Framework	The APEX Framework comprises all parts of APEX that are not Modules. The Framework may contain Authorable Objects.
APEX Feature	An APEX Feature is a group of inter-related APEX Authorable Objects that provide functionality to the game developer to facilitate the creation of a particular type of scalable content. An APEX Feature may span multiple Modules, and may include subsets of Modules. The specification of Features is driven primarily by marketing and product management, with input from architecture and engineering.
Scalability	The degree to which an APEX Feature enables content created for it to adapt to running on a variety of systems with various degrees of physics simulation capability, and to demonstrate benefit from running on systems with increased physics simulation capability.
Scalable Parameter	A member of an APEX asset that is authored as a range of values, rather than as a unique value, in order to provide scalability. At run-time, a value in the authored range is selected based on the capabilities of the system.

Directory Structure

The APEX top level directory contains:

• framework	Everything that's not a module
• module	APEX module specific code
• shared	Example source code that may be useful to the developer
• bin	Platform and SDK version specific DLLs
• lib	Platform and SDK version specific libs
• compiler	Platform and SDK version specific build scripts
• docs	Documentation
• samples	Sample applications
• tools	Authoring tools

At the lowest level of the directory hierarchy, the following naming conventions are used:

.../public	This is the set of public (external) interface classes which hide underlying implementation. They can be used by any application, by tools, or internal APEX code.
.../include	Non-public header files, including classes that implement public interface classes.
.../src	Source code.

An APEX binary distribution consists of a set of DLL's (Windows) and libraries (all platforms) as well as the header files of the Public API (APEX/framework/public and APEX/module/xx/public). It will also include tools (executables) and samples (source & executables).

Each APEX Module and the APEX Framework is in a separate library (and Windows DLL). An application can thus link (or load) only the APEX Modules that the application will be using in order to conserve code space. A further benefit of separate libraries is that it is easy for NVIDIA or third-party developers to create new APEX Modules that can be easily used in an existing APEX application.

Version Numbers

The APEX Framework and all APEX Modules are independently versioned using a 2-part version number, consisting of a major number and a minor number, in the format "major.minor", for example: "1.3".

Module and framework libraries and DLL's with equal major numbers are binary compatible, regardless of minor number. Modules and framework with different major numbers are not compatible. For example, framework 1.2, module_A 1.0, and module_B 1.5 can all interoperate.

This compatibility is accomplished by ensuring that the framework "Nx" and "Ni" public interfaces, as well as the module "Ni" interfaces, never change across different minor numbers (same major number). The module "Nx" interfaces can change anytime. Module loading interfaces must always be backward compatible to enable detecting major number mismatches when the framework loads a module.

Asset definitions within a module may change from version to version of the module, but new versions of the NxApexAssetAuthoring and NxApexAsset derived classes must support deserializing all old versions of the asset.

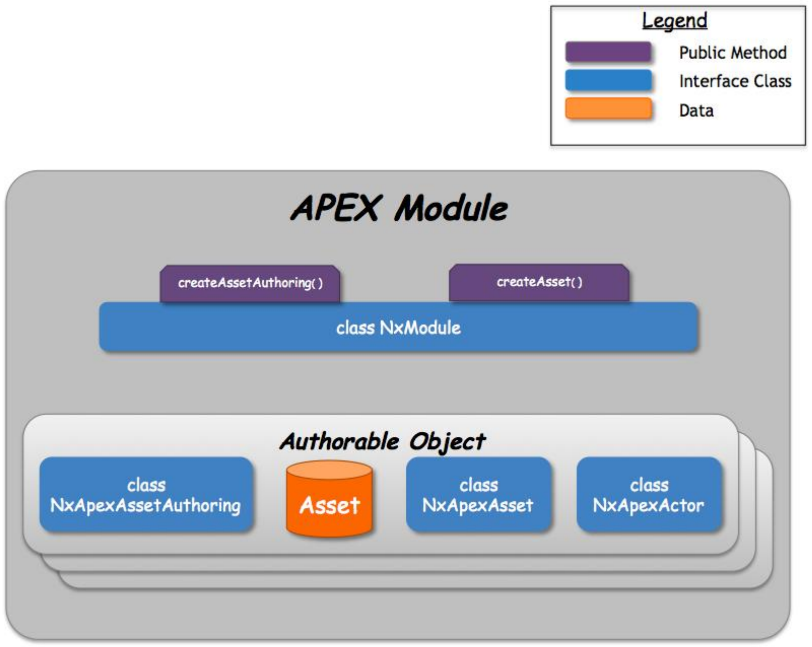
Assets and Actors

Authorable Objects

The Authorable Object is the heart of the APEX architecture. It is a data structure, the specification of how it will be authored, the specification of the runtime functionality associated with it, and the software which implements this functionality.

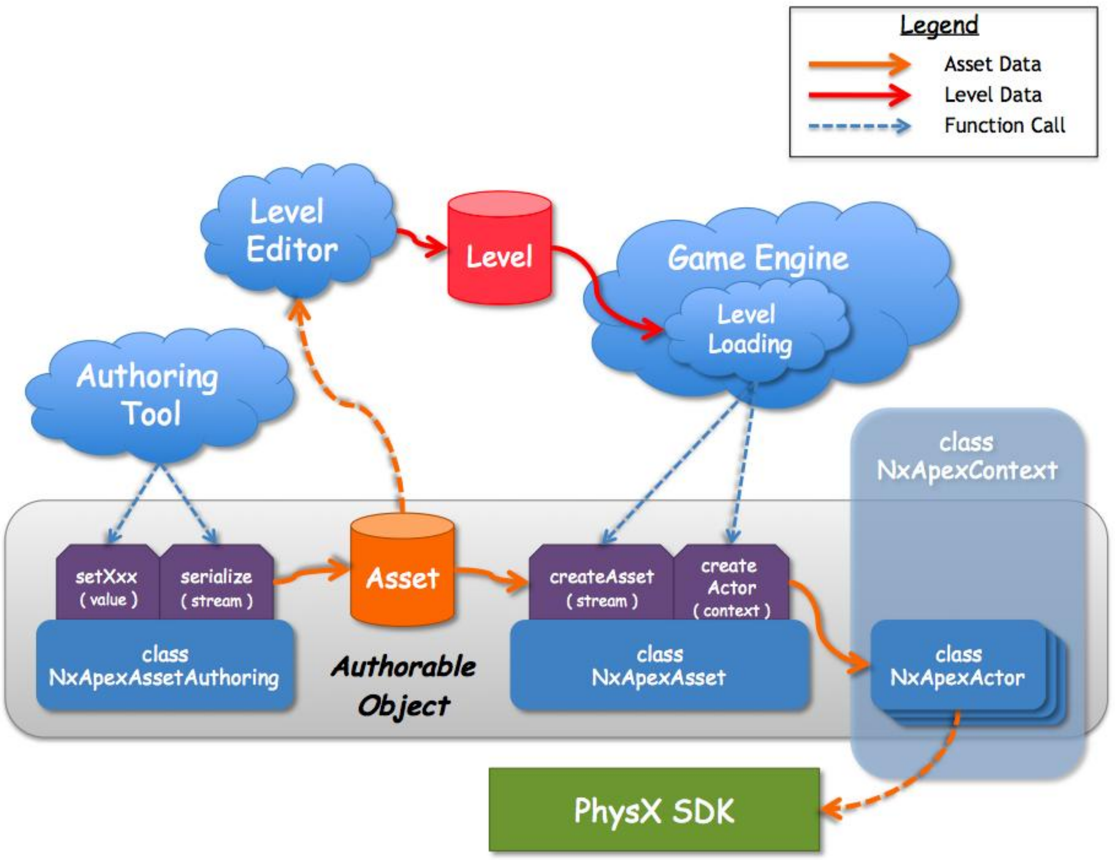
APEX Modules implement one or more types of "authorable object". For each authorable object, the module must implement 3 classes, derived from each of the 3 following base classes: NxApexAssetAuthoring, NxApexAsset, and NxApexActor. The derived classes should be named: NxXxxYyyAssetAuthoring, NxXxxYyyAsset, and NxXxxYyyActor, where "Xxx" is the module name, and "Yyy" is the object name. The object name ("Yyy") is optional if the module only contains one authorable object.

Thus, for every Asset class, there must be a corresponding Actor class, and a corresponding AssetAuthoring class.



Asset and AssetAuthoring classes are created, respectively, by the createAssetAuthoring() and createAsset() methods of the module, whereas actor classes are created by the asset.

Asset Data Flow



Level Editor and Level Loading

Level editing and loading functionality is outside the scope of what is provided by APEX. The game engine must provide this functionality.

APEX requires that the level loading system be able to instantiate named assets. The level loading system must be able to handle instances with and without a position in the level. For example, particle systems do not have a position, but particle emitters do.

Asset Authoring Class

The classes derived from NxApexAssetAuthoring provided methods for authoring and serializing asset data.

These public interface classes only derive from the NxApexAssetAuthoring base class. However, the implementation classes derive from their corresponding public interface classes, the implementation base class (apexAssetAuthoring), and the corresponding Asset implementation class. Only the implementation class derives from the corresponding Asset implementation class. The Public interface classes, NxApexAssetAuthoring and NxApexAsset, do not derive from each other. Inheritance from the Asset implementation class is necessary because the Asset implementation class contains the all the member data.

The NxApexAssetAuthoring class simply adds the serialize() method and various setXxx() methods that are used (typically by the authoring tool) to author the asset by setting the data members that can be subsequently serialized.

Asset Class

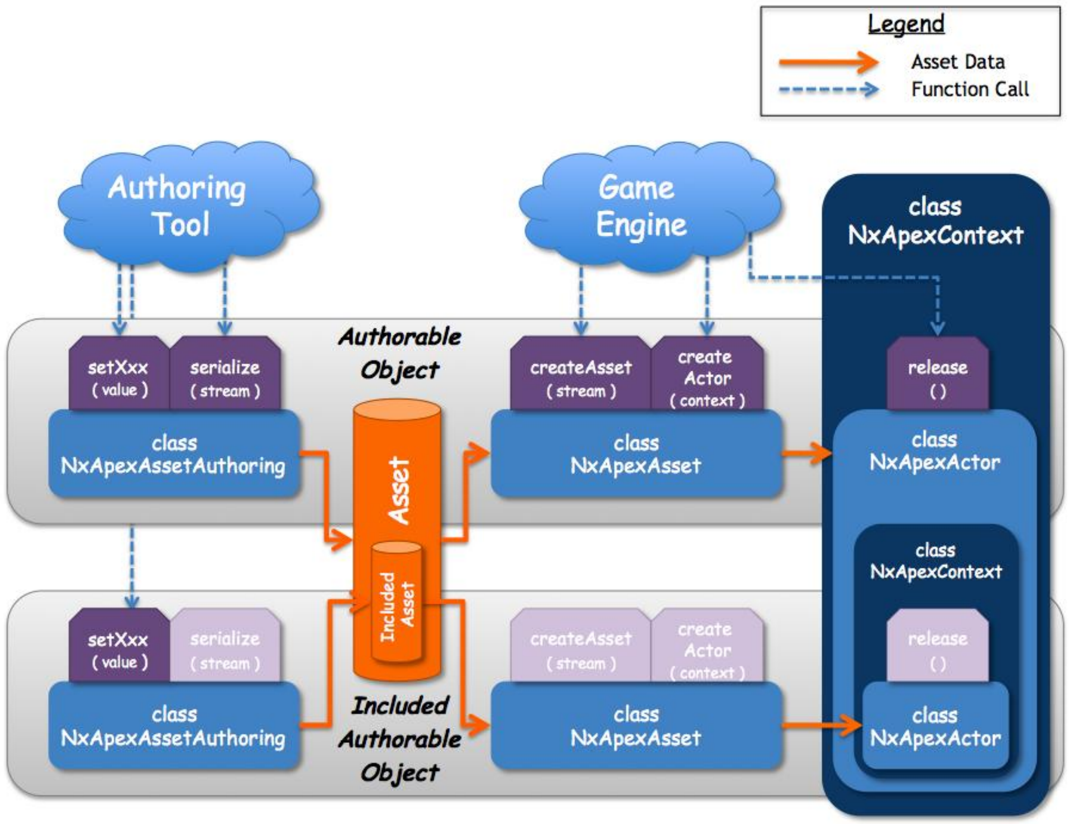
The classes derived from NxApexAsset are containers for authorable data. They always implements the deSerialize() and createActor() methods. The member variables in the implementation class contain all the asset data.

All assets must contain a name. When an asset is created or deserialized, the name must be registered with the Named Resource Provider (see below), together with a pointer to the asset (an NxApexAsset*). This allows the asset to be referenced by name by other assets.

Asset Inclusion

The classes derived from NxApexAsset are containers for authorable data. They always implements the deSerialize() and createActor() methods. The member variables in the implementation class contain all the asset data.

It is sometimes useful to implement authorable objects that are generic enough to be included in other authorable objects. For example, a "generic emitter" might be included in a "particle emitter". These objects may even be located in different modules.

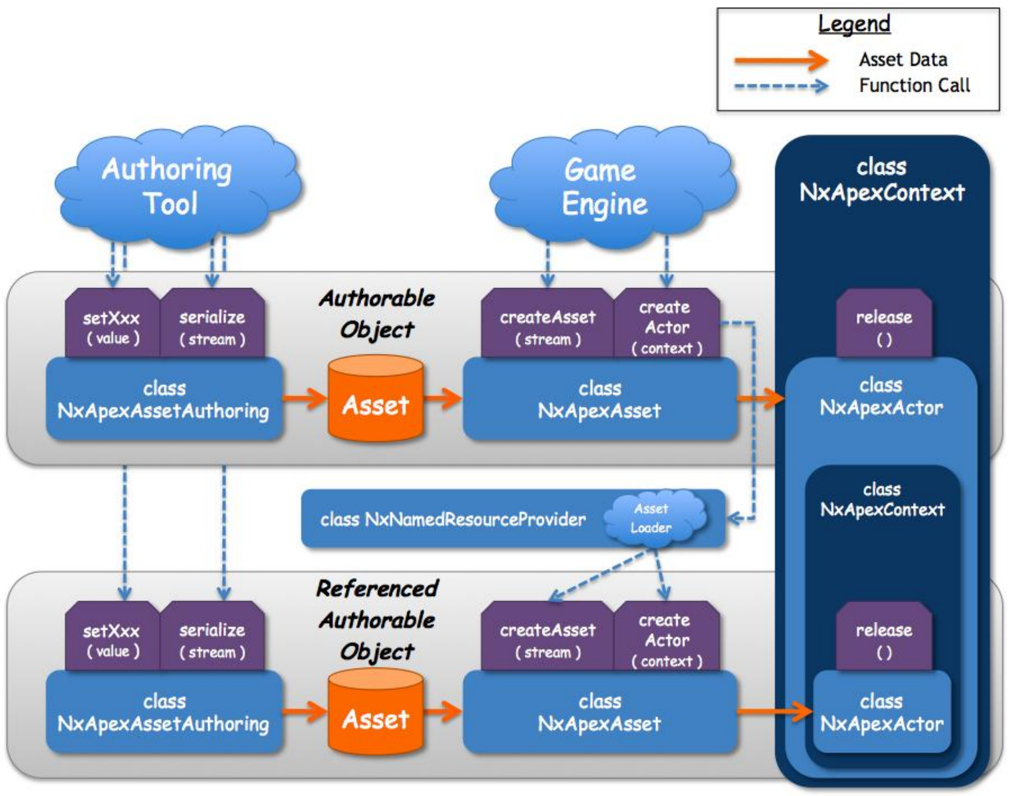


The AssetAuthoring, Asset, and Actor classes of the included object must be automatically created by the corresponding classes of the "including" object when they are created. Furthermore, the serialize(), deSerialize(), createActor(), releaseActor(), and release() methods of the included object must be automatically called by the corresponding classes of the including object.

The including Actor must furthermore derive from NxApexContext. This will be the context in which the included Actor must be created, as shown above. This will be a different context than the one in which the including actor is created.

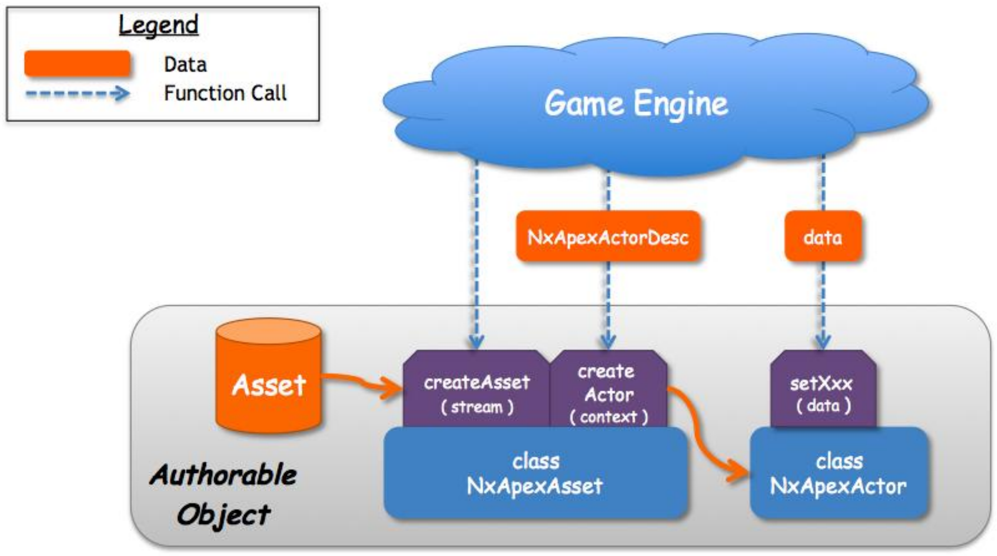
Asset Referencing

An alternative to "Asset Inclusion" is "Asset Referencing". In asset referencing, an asset contains (at authoring time) the name of another asset. This could be another asset type defined in the same module, an asset type defined in another module, or even an asset type defined in the framework. Asset names can be resolved (into a pointer to an asset (NxApexAsset*) using the Named Resource Provider. If the named asset has already been created, the pointer is returned immediately, but if the name cannot be resolved, the Named Resource Provider calls back to the application to give it a chance to create the named asset. If the application fails to create the named asset, a failure indication is returned and the name resolution fails. This, in turn, allows the referencing asset to fail actor creation.



Actor Class

The NxApexActor class represents a runtime instance of the authorable data.



The Asset class is the primary source of configuration data for the Actor. All data that can be authored must come from the Asset. However, runtime data may also be provided by the game engine. Runtime data that is needed for Actor creation should be provided in a descriptor structure passed to the createActor() method, for example, initial position. This data is typically provided by the game engine's level loading code. Runtime data that is not needed for Actor creation (i.e.: that can be provided later) should be provided by calling "set" methods of the Actor.

Even in cases where it only makes sense to have a single instance of the Actor class, the run-time functionality of the authorable object must still be placed in an Actor class rather than in the Asset class.

All instances of the Actor class are "owned" by the Asset that created them. (See discussion of "ownership" below.) Thus, the Asset implements the createActor() and releaseActor() methods.

Actor Context

All instances of the Actor class also have a "context". The concept of a context is completely different than the owner. The "context" is the context in which the Actor is created or instantiated, and it will be a class that derives from NxApexContext. A pointer to the context of an Actor is a standard argument to the NxApexAsset::createActor() method.

For most Actors, the context will be an instance of the NxApexScene class. However, some Actors may be created in other contexts, such as the NxApexSdk. Some Actors may even be created in the context of another Actor. The type of an Actor's context is defined on a Asset/Actor type basis.

The NxApexContext class provides a method to add and remove actors: addActor(), and removeActor(). It also provides a method to release all actors in the context, releaseActors(), which all actors from the context and calls their release() methods. This allows the context itself to be easily released. NxApexContext also provides an iterator method, iterateActors(), to iterate over the set of actors in the context.

Framework Interfaces

Initialization

In order to use APEX, the first thing that must be done by the application is to create a single instance of the NxApexSDK class:

```
#include "NxApex.h"

extern NxPhysicsSDK*         gPhysicsSDK;
extern NxCookingInterface*   gCooking;
NxApexSDKDesc                apexDesc;
NxSDKCreateError             errorCode;
NxApexSDK*                   gApexSDK;

        apexDesc.allocator = &gPhysicsSDK->getFoundationSDK().getAllocator();
        apexDesc.outputStream = gPhysicsSDK->getFoundationSDK().getErrorStream();
        apexDesc.physXSDK = gPhysicsSDK;
        apexDesc.cooking = gCooking;
        apexDesc.renderResourceManager = new UserRenderResourceManager();;

        gApexSDK = NxCreateApexSDK(apexDesc, &errorCode);
```

The PhysX SDK must have already been initialized, so you that you have a NxPhysicsSDK*. Most functionality of APEX is provided directly or indirectly through the NxApexSDK class. For example, scene creation and APEX module initialization is performed by calling methods of the NxApexSDK class.

Loading Modules

The NxApexSDK class provides a single generic public API for loading modules, called createModule(). Providing a single generic API makes it possible for third-parties to develop modules and ship module binaries (libs and DLL's) that are compatible with a particular version of the APEX SDK and of the PhysX SDK.

The module is identified by its filename, without path and without suffix/extension, which is the first argument to createModule(). The second second argument to createModule() is a pointer to an error result. The APEX SDK will ensure that the loaded module matches the APEX and PhysX SDK versions. Module specific parameters, if any, must be passed to the module using a subsequent call to the module's init() method.

Here is an example:

```
#include "NxApexSDK.h"
#include "NxModuleDestructible.h"

extern NxApexSDK            gApexSDK;
NxSDKCreateError            errorCode;
NxModuleDestructibleDesc    moduleDesc;
NxModuleDestructible       *gModuleDestructible;

#if MODULE_STATIC_LINKED
    instantiateModuleDestructible();
#endif

    gModuleDestructible = dynamic_cast<NxModuleDestructible*>
        ( gApexSDK->createModule("Destructible", &errorCode) );

    assert(gModuleDestructible);

    moduleDesc.destructibleActorGroup = 1;
    gModuleDestructible->init(moduleDesc);
```

Creating Scenes

The APEX scene (class NxApexScene) represents a unique physics world in which various types of objects can be instantiated. It mirrors the PhysX SDK NxScene class. An instance of the NxApexScene class should be created for each NxScene in which APEX will be used.

The APEX scene provides a centralized interface for advancing the simulation and accessing the rendering data output from APEX.

The NxApexScene derives from NxApexContext, and therefore can contain (or be the context for) APEX actors. In fact, all *asset::create*Actor() methods must take an NxApexScene argument.

The NxApexScene operates in similar fashion to the PhysX SDK NxScene, however, the APEX scene can exist, and even contain APEX actors, without having an associated PhysX SDK NxScene. But in order to actually perform simulation in the PhysX SDK, an NxScene must be associated with the APEX scene, either by providing a pointer to it in the NxApexSceneDesc descriptor structure, or by using the setPhysXScene() method.

```
NxApexScene*           gApexScene;
NxApexSceneDesc        apexSceneDesc;

        // Create the APEX scene...
        apexSceneDesc.scene = (NxScene *)0;
        gApexScene = gApexSDK->createScene(apexSceneDesc);
```

```
extern NxScene*        scene;

        // Associate the PhysX SDK NxScene...
        gApexScene->setPhysXScene(scene);
```

After associating an NxScene, the application must not delete the NxScene until after it has been disassociated by calling setPhysicsScene() with a null pointer:

```
        gApexScene->setPhysicsScene((NxScene *)0);

        // Now, it's safe to release the NxScene...
```

The APEX SDK owns the APEX scene. According to APEX ownership rules, this means that the NxApexSDK class provides a public releaseScene() method. In turn, NxApexScene provides a public release() method, which simply calls NxApexSDK::releaseScene(this). It also provides a protected destroy() method that is called by releaseScene().

Loading Assets

At run-time, asset creation is typically performed by the game engine's "level loader" using the Named Resource Provider's user defined NxResourceCallback::requestResource() method. The level loader typically processes a set of asset instances that have been placed in the level. At a minimum, the game engine must be capable of storing the following information for each asset instance in the level: name of the asset, the type of the asset, and position and orientation of the instance. For each asset instance found in a level, the level loader should query the NRP to determine whether or not the named asset has been already loaded. It should use the NxResourceProvider::getResource() method to do so. If the asset has not been loaded, a callback to NxResourceCallback::requestResource() will be immediately triggered to request that the asset be loaded. Either way, getResource() will return a void pointer to the loaded asset that can subsequently be used to create the instance of the asset: the Actor.

To load APEX assets, getResource() callback should first create an NxApexStream for reading the asset from non-volatile storage. If a filename is required, the name of the asset can be used for the base part of the filename, and the namespace can be used to determine the suffix (file extension) and optionally the directory path.

If the APEX Default Stream implementation is being used (e.g.: the APEX samples), a stream can be created with the NxApexSDK::createStream() method:

```
        NxApexStream* stream = gApexSDK->createStream(
            "asset.pda",
            NxApexStreamFlags::OPEN_FOR_READ );
```

Otherwise, an instance of a user defined stream class (derived from the pure virtual NxApexStream base class) should be created.

Next, the streaming asset creation method of the module is used to de-serialize (read) the asset from the stream. A reference to the stream must be provided, as well as the name of the asset, for example:

```
        NxDestructibleAsset* asset =
            gModuleDestructible->createDestructibleAsset( *stream, name );
```

Instantiating Actors

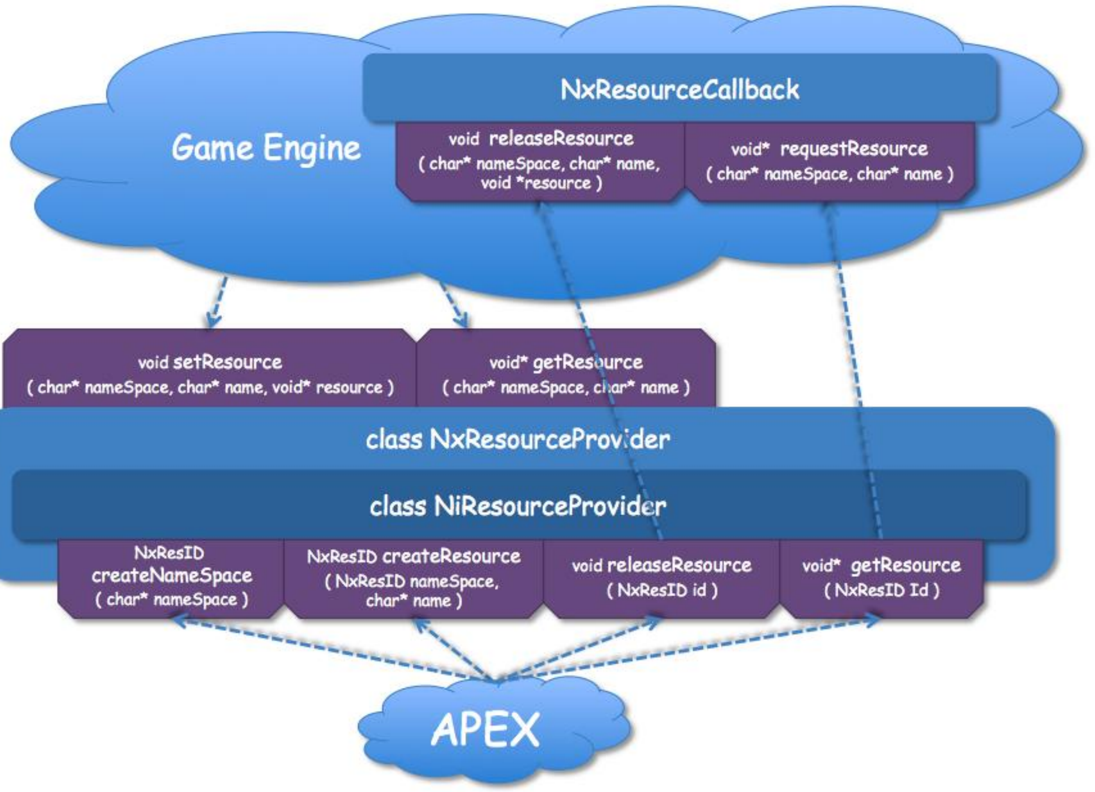
Once an asset has been created, de-serialized, and registered with the Named Resource Provider, instantiating it in a scene is as easy as calling its createXxxActor() method. Additional information, such as its global pose (position and orientation in the scene), that is commonly available to the level loader can be provided in a descriptor data structure that is passed to the createXxxActor() method along with a pointer to the APEX Scene:

```
        NxDestructibleActorDesc destructibleDesc;

        /* Set the NxMat34 pose */
        destructibleDesc.globalPose = pose;

        /* Create the destructible actor in the APEX scene */
        NxDestructibleActor* actor =
            asset->createDestructibleActor( destructibleDesc, *gApexScene );
```

Named Resource Provider

The Named Resource Provider (NRP) is a public interface class in the APEX Framework (class NxApexResourceProvider). It's purpose is to provide name to address mapping for APEX assets (NxApexAsset and derived classes) and for miscellaneous game engine resources that APEX authorable objects refer to by name, such as materials/textures. It also provides a callback interface to allow the application to load the named asset and thus resolve the mapping.

The NRP maintains mappings from a namespace/name pair to a resource pointer. Namespaces and names are both null terminated character strings (char*), and resource pointers are void pointers (void*). Internally to APEX, an integer (NxU32) resource identifier is uniquely associated with every namespace/name pair. The resource identifier is used to efficiently access the current value of a resource pointer, without requiring the NRP to perform string operations everytime the resource pointer is queried. The resource identifier space will be global; resource identifier values will not be re-used in different namespaces.


NxResourceCallback
Game Engine
void releaseResource (char* nameSpace, char* name, void *resource)
void* requestResource (char* nameSpace, char* name)
void setResource (char* nameSpace, char* name, void* resource)
void* getResource (char* nameSpace, char* name)
class NxResourceProvider
class NiResourceProvider
NxResID createNameSpace (char* nameSpace)
NxResID createResource (NxResID nameSpace, char* name)
void releaseResource (NxResID id)
void* getResource (NxResID Id)
APEX

Every module, when created, should create a new namespace for each type of authorable object (asset) that it implements. The namespace for an authorable object must be documented, and must not be changed, because it is through this name that game developers, as well as other modules, will reference assets of this type. At run-time, the NRP will associate an integer (NxU32) namespace identifier with each namespace, to optimize calls to createResource().

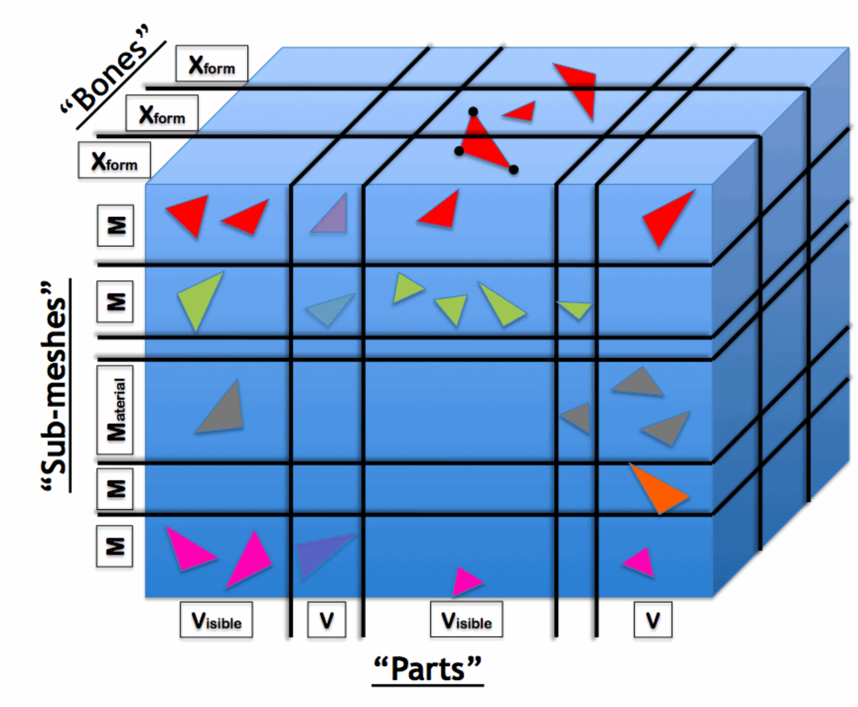
Whenever any asset is referenced by name (e.g.: when some other asset is loaded), the APEX module must use the createResource() method to create a mapping from the namespace/name pair to a unique resource identifier. This only needs to be done once, as resource identifiers are permanently (for the lifetime of the APEX SDK) associated with the namespace/name pair. The referencing object should save the assigned resource identifier for subsequent use with the getResource() method to query the current value of the resource pointer. APEX modules must, however, not save the resource pointer for future use. Rather, they must call getResource() every time they need to access the resource pointer. This allows the application to call setResource() at any time to change the value of the resource pointer. When the referencing object is done referencing the resource (e.g.: when the referencing object is deleted), it must call releaseResource(), which will call the NxResourceCallback::releaseResource() callback if the number of references is now zero.

The public interface consists of a single method, setResource(), and two callback functions, requestResource() and releaseResource(), that are implemented in a user defined subclass of the pure virtual base class, NxResourceCallback. The application may call setResource() at any time, to create a mapping between a namespace/name pair and a resource pointer. This can be safely done before the createNameSpace() or createResource() calls are made by the APEX modules. If the application has not yet called setResource() to create a mapping (for a particular namespace/name pair) when getResource() is called for the first time from within APEX, then the NRP will call the requestResource() callback to request the resource pointer to use for the mapping. The callback must return a value, however, the value may be 0 (NULL). The callback will only be called once for a given namespace/name pair. If the application wants to subsequently change the mapping, it must call setResource(). This may be done as often as the application wishes, and getResource() will always return the correct (most recent) resource pointer. The user class implementing NxResourceCallback must be provided in the descriptor when the APEX SDK is initialized, in the call to NxCreateApexSDK().

Render Mesh Asset

The NxRenderMeshAsset class, and the associated NxRenderMeshAssetAuthoring and NxRenderMeshActor classes, implement a general purpose triangle mesh asset. These classes are provided by the APEX Framework, and can be used, by named reference, or by inclusion by assets in other modules.

The NxRenderMeshAsset class supports a wide variety of vertex attributes: position, normal, tangent, binormal, color, bone indices, and up to 4 sets of [U,V] coordinates.



The mesh can be subdivided in two orthogonal ways: by "part" and by "sub-mesh". Furthermore, each vertex of the mesh can be associated with one or more "bones" (transformation matrices).

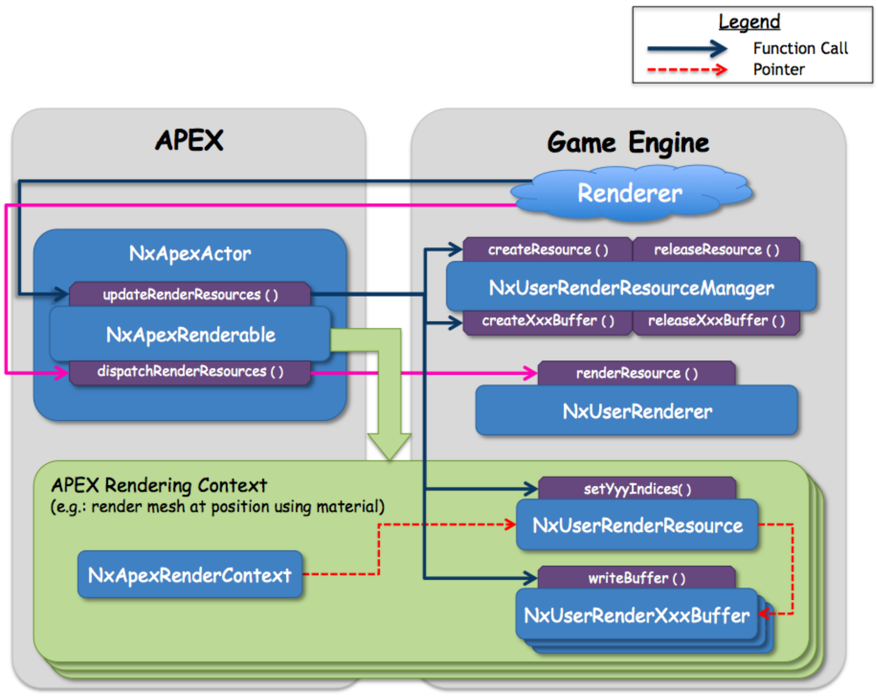
Each "part" is a unique subset of the triangles of the mesh for which rendering can be individually enabled or disabled. This is called "visibility", and is controlled, per instance, using the NxRenderMeshActor::setVisibility(..) method. A part can span several sub-meshes (see below), such that it can use different materials for different triangles.

Each "sub-mesh" is a unique subset of the vertices and triangles of the mesh that is assigned a particular material name. At run-time, the material name is mapped to a user-defined pointer, using the Named Resource Provider. This pointer is subsequently passed to the rendering engine through the Render Resources Interface. A sub-mesh can contain triangles from many different parts.

Each "bone" is a transform matrix (position, rotation, and scale) that can be used at run-time to transform the position of the vertices of the mesh. Each vertex can be associated with one or more bones using the "bone index" vertex semantic. If a vertex is associated with more than one bone, the "bone weight" semantic is also used to provide a weighting factor that determines the proportion in which the vertex position is transformed by each bone. If all the vertices of a part are associated with a single common bone, the part will be rigid. If they are associated with different bones, the part will be "skinned".

Render Resources Interface

The APEX Framework provides a standard interface for modules to interface with the game's rendering engine. The purpose of the interface is to pass data (e.g.: vertex buffers, index buffers, transform buffers) from APEX to the rendering engine for rendering. APEX does not directly render anything (i.e.: no calls to D3D or OpenGL), nor does specify or control how or when things are rendered. It simply seeks to pass data that it generates or updates to the game's rendering engine in a flexible yet efficient manner.



User Implemented Classes

APEX interfaces with the game engine via several classes which are implemented by the user. The first class manages the creation and deletion of render resources. A user defined instance of NxUserRenderResourceManager is provided to the ApexSDK at startup, and is used globally by APEX to request/release render resources from the game.

```
class NxUserRenderResourceManager
{
public:
  virtual NxUserRenderVertexBuffer   *createVertexBuffer( const NxUserRenderVertexBufferDesc &desc );
  virtual void                        releaseVertexBuffer( NxUserRenderVertexBuffer &buffer );

  virtual NxUserRenderIndexBuffer    *createIndexBuffer( const NxUserRenderIndexBufferDesc &desc );
  virtual void                        releaseIndexBuffer( NxUserRenderIndexBuffer &buffer );

  virtual NxUserRenderBoneBuffer     *createBoneBuffer( const NxUserRenderBoneBufferDesc &desc );
  virtual void                        releaseBoneBuffer( NxUserRenderBoneBuffer &buffer );

  virtual NxUserRenderInstanceBuffer *createInstanceBuffer( const NxUserRenderInstanceBufferDesc &desc
);
  virtual void                        releaseInstanceBuffer( NxUserRenderInstanceBuffer &buffer );

  virtual NxUserRenderResource       *createResource( const NxUserRenderResourceDesc &desc );
  virtual void                        releaseResource( NxUserRenderResource &resource );
};
```

APEX only requests new resources from this interface in the context of calls to updateRenderResources(), so those functions generally do not need to be thread safe. However, any APEX API call that results in actor deletion can result in render resources being released to NxUserRenderResourceManager, so the release methods of this user implemented class must be thread safe.

Each of the NxUserRender*Buffer classes are also abstract and must be implemented by the user. The buffer descriptor passed to the create methods will describe the data format of each semantic (or member) of that buffer, so the user class has the ability to pack or interleave that data into as many or as few GPU buffers as necessary. The descriptor will also contain a hint from APEX on the usage of that buffer; whether it is static or dynamic. Each render buffer class has a user-implemented write buffer method that can be called by APEX (only from within the context of updateRenderResources()) to update the data in that buffer. The write methods are defined as follows:

```
void writeBuffer( NxRenderVertexSemantic::Enum semantic, void *srcData, NxU32 srcStride,
                  NxU32 firstDestElement, NxU32 numElements )
```

APEX provides a pointer to the new source data (srcData) and it's stride (srcStride), plus the ranges of values in the target buffer to be updated (firstDestElement, numElements). The user is expected to know the size of the APEX data from the descriptor used to create the buffer. If locking and unlocking of buffers is expensive for the game, they can cache the writeBuffer() calls made during updateRenderResources() and then play them back all in a single lock/write/unlock pass. Caching multiple writeBuffer() calls is guaranteed to generate coherent data so long as the actor remains locked until all of the writes are completed.

Next, the user must implement an NxUserRenderResource class for managing a renderable entity. The descriptor for the NxUserRenderResource contains pointer to all the various user buffers allocated for this renderable actor. Some of those buffers may be shared with other actors. The public interface of the NxUserRenderResource has methods to update indices in the various buffers, associate a material, and to retrieve those user buffers. APEX will only call these methods from within updateRenderResources() calls.

Note that when an NxUserRenderResource instance is released to the NxUserRenderResourceManager, the resource manager should not implicitly release the render buffers used by that render resource. Instead, APEX will explicitly release those resources as required. Only APEX understands all of the sharing semantics, so the the game should not call any of those release methods itself.

The last class that the user must implement is NxUserRenderer:

```
class NxUserRenderer
{
public:
  virtual void renderResource( const NxApexRenderContext &context ) = 0;
};
```

An instance of this class must be passed to dispatchRenderResources() (described below). From the context of dispatchRenderResources(), APEX will call this renderResource() method a number of times to pass NxApexRenderContexts back to the game engine. NxApexRenderContext is not user-defined, but it is a simple structure:

```
class NxApexRenderContext
{
public:
        NxUserRenderResource *renderResource;
        NxMat34           local2world;
        NxMat34           world2local;
};
```

It essentially informs the game engine that a particular NxUserRenderResource instance should be rendered at a particular world pose. These render contexts could be rendered immediately by your NxUserRenderer, or it may cache them for later use. The render contexts are guaranteed to be valid until the next updateRenderResources() call is made on that actor.

NxApexRenderable Actor Interface

All transactions with the "Interface to Rendering" are initiated by the game calling an APEX API. This interface is thread-safe, and can be called at any time during the frame, including during simulation, and while calls are being made from other game threads to other APEX API's (assuming the locking semantics described below are respected). These calls are all methods of the NxApexRenderable class, which all renderable APEX actors derive from:

```
const NxBounds3& getBounds() const;
```

```
void updateRenderResources();
void dispatchRenderResources( NxUserRenderer &renderer, NxReal lod );
```

Calling updateRenderResources() on an APEX actor allows APEX to issue callbacks to allocate any buffers that need to be allocated, and to update (fill-in) any buffers that need to be updated. By performing these tasks in callbacks from updateRenderResources(), APEX allows the game to control when buffers are allocated and written to, as well as the thread context in which this occurs. These buffers can therefore be directly allocated from the graphics subsystem (e.g.: from D3D), even when the graphics API is not thread-safe, thus avoiding additional buffer copies. updateRenderResources() must be called before dispatchRenderResources(), but can be skipped entirely if the actor will not be rendererd that frame.

Calling dispatchRenderResources() on an APEX actor triggers APEX to call the provided renderer's renderResource() method for each of the actor's NxUserRenderResources. It will not update or allocate any new rendering resources. It can be called as many times as the game wishes to generate the same set of NxApexRenderContexts.

APEX Scene Based Render Resource Updating

The ApexScene has an API for creating an iterator for the scene's renderable actors. It is used in the following manner:

```
MyCachedRenderer renderer;
NxApexRenderableIterator *iter = gApexScene->createRenderableIterator();
for( NxApexRenderable *r = iter->getFirst() ; r ; r = iter->getNext() )
{
   if( earlyCullCheck( r->getBounds() ) ) continue;
   r->updateRenderResources();
   r->dispatchRenderResources( renderer, 0 );
}
iter->release();

renderer.render_pass_1();
renderer.render_pass_2();
```

When the game is ready to begin rendering, it can acquire an iterator from the ApexScene and use it to call updateRenderResources() and dispatchRenderResources() on all renderables in the scene it wishes to render. The iterator implicitly handles all of the actor locking and will in fact temporarily skip over actors locked by other threads so that the loop can be completed as efficiently as possible. If the same iterator is used to make multiple passes through a scene's actors, it will also safely deal with actor deletions.

With this method, the game can store all of the NxApexRenderContexts passed to it's renderer and use them safely in as many rendering passes as it needs without any worry of conflicting with the ApexScene or other game threads making calls to the APEX API.

Ad-Hoc Render Resource Updating

Commonly, the game rendering thread will already have a scene graph for it's rendering purposes and this graph will contain pointers to all the NxApexRenderable actors. In this case, the game will not want to use an iterators to access the actors. Instead, it will want to access them in an ad-hoc method.

```
NxApexRenderable *r = getApexRenderable( gameActor );
MyCachedRenderer gRenderer;
if( earlyCullCheck( r->getBounds() ) ) return;
r->lockRenderResources();
r->updateRenderResources();
r->dispatchRenderResources( gRenderer, 0 );
r->unlockRenderResources();
```

With this method, APEX is unable to ensure the actor has not been deleted by another game thread, so the game engine must ensure the actor still exists. While the actor is locked, it cannot be deleted or modified in any way by APEX. It's rendering data is guaranteed to stay in a consistent state until it is unlocked.

Example Overview

From 1000 feet up, the situation looks like this:

1. The game allocates an NxUserRenderResourceManager instance and provides it to the NxApexSDK at creation
2. The main game thread calls NxApexScene::simulate() and NxApexScene::fetchResults() each frame (probably not in that order)
3. When NxApexScene::fetchResults() is called, the APEX actor states are updated to their new world positions and world bounds.
4. In the rendering thread, using either an NxApexScene iterator or the ad-hoc locking method, the game finds actors which are close enough to the view frustrum to be renderable and calls their updateRenderResources() methods
5. Inside updateRenderResources(), APEX (may) call back to the NxUserRenderResourceManager to allocate new render resources. APEX then calls buffer write methods to pass the latest ApexActor state to the game engine.
6. Subsequent calls to updateRenderResources() before the next invocation of NxApexScene::fetchResults() should equate to NOOPs
7. The rendering thread can now call dispatchRenderResources() on any of the ApexActors which have been updated, and it can make this call as many times as it wishes. Each time, dispatchRenderResources() will generate the same list of NxApexRenderContexts.

Materials

APEX does not "know" anything about render materials. The details of materials and shaders vary so much one renderer to the next, as well as from one game engine to the next. It would be very difficult, if not impossible, to abstract material handling in an useful way.

However, APEX must know something about materials, to the extent that it groups render vertices according to different materials, and informs the application as to which material is to be used with a vertex buffer when render resources are dispatched (see Render Resources Interface). It accomplishes this via the Named Resource Provider, allowing the application to register all materials (as void* pointers or integer handles) by their names. APEX render mesh assets (NxApexRenderMeshAsset) group vertices by material, and refer to the materials by name. Before the rendering resources are dispatched to the user for a draw call, those material names are matched to the user's material handles (void* or integer) using the Named Resource Provider. In this way APEX has an efficient and meaningful way to refer to the material at runtime.

This would be the end of the section on materials, except that APEX has cases where it is its "own customer," that is, with applications that need to render. Once case is for authoring tools, another is for sample or demo applications.

Default Material Library Implementation

As a utility for tools, samples, and demos, APEX has a common generic material library class, ApexDefaultMaterialLibrary. This class can create simple materials with texture maps (currently a diffuse, bump, and normal map are supported). The texture maps may be read in from a variety of file format (to support authoring), and stores the texture maps in its own generic internal format.

The ApexDefaultMaterialLibrary is used to store material information for fractured mesh authoring in the DestructionTool, and to load materials for rendering in the Destruction module sample applications. Our SimpleRenderer code, used by our tools and demos, interfaces with ApexDefaultMaterialLibrary.

A ApexDefaultMaterialLibrary stores ApexDefaultMaterials, which in turn can refer to ApexDefaultTextureMaps. These three classes are defined in samples/common/include, and are based upon abstract base classes.

Asynchronous Simulation Interface

Simulation is controlled by the APEX scene. The NxApexScene class implements simulate(), checkResults(), and fetchResults() methods that operate in similar fashion to the corresponding methods of the PhysX SDK NxScene class.

Just as with scene creation, these three simulation methods automatically call the corresponding PhysX SDK methods, eliminating the need for the game to make duplicate calls. Taking control of stepping the PhysX SDK simulations allows APEX modules to perform tasks before, during, and after the PhysX SDK simulation.

In order to do all this, each NxApexScene spawns a new simulation thread, the "APEX scene thread".

When the game thread calls NxApexScene::simulate(), a signal is sent which unblocks the APEX scene thread. The APEX scene thread then passes through three phases. In each phase, each APEX ModuleScene (see above) is allowed to perform any asynchronous processing that it desires. Furthermore, during the second phase, the PhysX SDK simulation is allowed to proceed, by means of a call to NxScene::simulate(), followed by a call to NxScene::checkResults(block=true).

The APEX scene in turns passes the execution context to each module in turn.

Buffered API

Changes to the simulation state (calls to "set" methods of the API) must be made before the call to the simulate() method for the timestep(s) in which the changes are to be effective. Changes that are made after the call to simulate() are buffered until simulation completes (i.e.: until fetchResults()). Changes are applied in the order in which the API calls were originally made. This functionality is similar to what is provided by the NxD wrapper for the PhysX SDK, except that the functionality is built-in to APEX rather than being implemented as a wrapper.

Deletion of scenes is also buffered. Deleting a scene during simulation will not take effect until the simulation completes. Even though the deletion is buffered, the application must not make any more API calls that reference the scene, e.g.: don't call fetchResults().

Calling fetchResults() before a corresponding simulate() call is allowed but does nothing. This avoids the need for the application to do a first-time check when calling fetchResults() for the previous frame immediately before simulate() for the next frame.

Scalability

The "Soul of APEX" is to facilitate the creation of scalable physics content by game developers.

For this reason, this section is, in many ways, the most important one of the document.

Scalability Requirement: Each APEX feature must implement 1 or more Scalable Parameters (See Definitions).

However, due to the hierarchical relationship of many Authorable Objects, not all Authorable Objects need to implement a scalable parameter.

By definition, a Scalable Parameter is the specification of a range of values that is authored and saved in asset. Scalable Parameters may be found in any kind of asset, including those that are only instantiated once per-module or per-module-scene. In other words, a module may define Scalable Parameters that are "global" to the module, or that are authored on a per-scene basis.

Scalable Parameters don't inherently have anything to do with Level of Detail (LoD) (see below), but they can be used to configure various LoD settings. For example, a simple LoD system might simulate the N most important widgets in the scene, while leaving the remaining widgets static. In such a scenario, N could be a Scalable Parameter.

Level of Detail

APEX Modules will frequently implement Level of Detail (LoD) systems. LoD systems are simply algorithms that determine the best way to distribute limited resources over a set of objects in a scene. They typically use criteria such as distance the object from the camera, size of the object, age of the object, or relevance to game-play, to determine the optimal distribution of resources.

An LoD system, however, does not guarantee scalability. For example, a simple LoD system might simulate the N most important widgets in the scene, while leaving the remaining widgets static. In such a system, if N is a simple parameter that is authored and saved in an Asset, then it is not scalable. If, however, N is a Scalable Parameter that is authored as a range of values (i.e.: min/max) and saved in an Asset, then the LoD system is scalable.

Ownership of PhysX SDK Objects

The APEX SDK provides a mechanism for the application to make queries about PhysX SDK objects, such as NxActors, that have been created by APEX. For example:

```
void myOnContactNotify( NxContactPair& pair, NxU32 events )
{
  const NxApexPhysXObjectDesc *desc0 = NxGetApexSDK()->getPhysXObjectInfo( pair[ 0 ] );
  const NxApexPhysXObjectDesc *desc1 = NxGetApexSDK()->getPhysXObjectInfo( pair[ 1 ] );

  if( desc0 )
  {
     if( desc0->getActor()->getOwner()->getObjectTypeID() ==
          gModuleDestructible->getModuleID() )
     {
         NxDestructibleActor* da = dynamic_cast<NxDestructibleActor*>
             ( desc0->getActor() );
         //...
     }
```

```
    }
}
```

Serialization

Serialization is the process by which APEX assets are saved (serialized) by an authoring tool and loaded (de-serialized) by the APEX run-time.

Every APEX Authorable Object defines a set of data that makes up the serialized asset. The serialization process is divided into two parts:

- The first part determines what data will be serialized, and in what order (the asset contents).
- The second part determines how the data is encoded and stored in the "stream" (the asset format).

Asset Contents (serialize and deserialize methods)

The first part of the serialization process, the determination of the serialized data and the order in which it is serialized, is specific to each Authorable Object, and is implemented in methods called serialize() and deserialize(), in the NxApexAssetAuthoring public interface class. The implementation of both methods, however, is actually in the ApexAsset implementation class. This is because the module's createXxxAsset() method needs to access the deserialize() functionality in order to create and load the asset.

Stream Format (NxApexStream class)

The second part of the serialization process, the determination of how the data is encoded and stored, is common to all Authorable Objects, and is implemented in a class derived from the NxApexStream public base class. A default implementation is provided in the NxUserStream class, but an alternate implementation may be provided by the game developer. The default implementation simply uses the standard C library to read and write the asset as a binary file, with little endien byte ordering, using API's such as fopen(), fread(), fwrite(), and fclose(). A user defined implementation of the NxApexStream interface can be used, for example, to wrap the asset in a game specific header, or to compress the data, or to combine multiple assets into a single file.

User defined stream format: "apexuser.dll"

In the APEX run-time, it is straightforward to use a user-defined stream implementation. The game developer simply implements a new class, derived from NxApexStream, and use it in all the calls NxModule::createXxxAsset(). This gives the game engine total control over how the APEX assets are stored. It allows the game to "wrap" APEX assets in any type of game specific wrapper, embed them in some game specific file(s), or encode them in some game specific encoding.

In the authoring pipeline however, the situation is a bit more complicated, because the authoring tools are typically provided as binary executables. In order to support user-defined stream implementations, the NxApexSDK class provides a createStream(const char *filename, bool load) method, for mandatory use by all authoring tools, that creates and returns a pointer to a class derived from NxApexStream. This method attempts to load a Windows DLL called "apexuser.dll", which must, in turn, export a "C" function NxApexStream*createStream(const char *filename, bool flags). If the DLL cannot be loaded, or does not export a function named createStream(), then NxApexSDK::createStream() returns a pointer to the APEX default implementation of NxApexStream.

Asset header

All APEX assets use a common preamble, or "header". It consists of the following fields:

1. APEX stream version (NxU32) (enum ApexStreamVersion)
2. Authorable object type (ApexSimpleString)
3. Authorable object version (NxU32)

Note: Headers for APEX stream versions prior to ApexStreamVersion_1_0 only contain the APEX stream version number. Subsequent data is asset specific, but there is no way to determine the type of asset from the contents of the stream. These assets rely on the game engine to ensure that a stream contains the correct type of asset for the deserialize method being invoked. This problem only occurs in alpha releases of APEX.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com