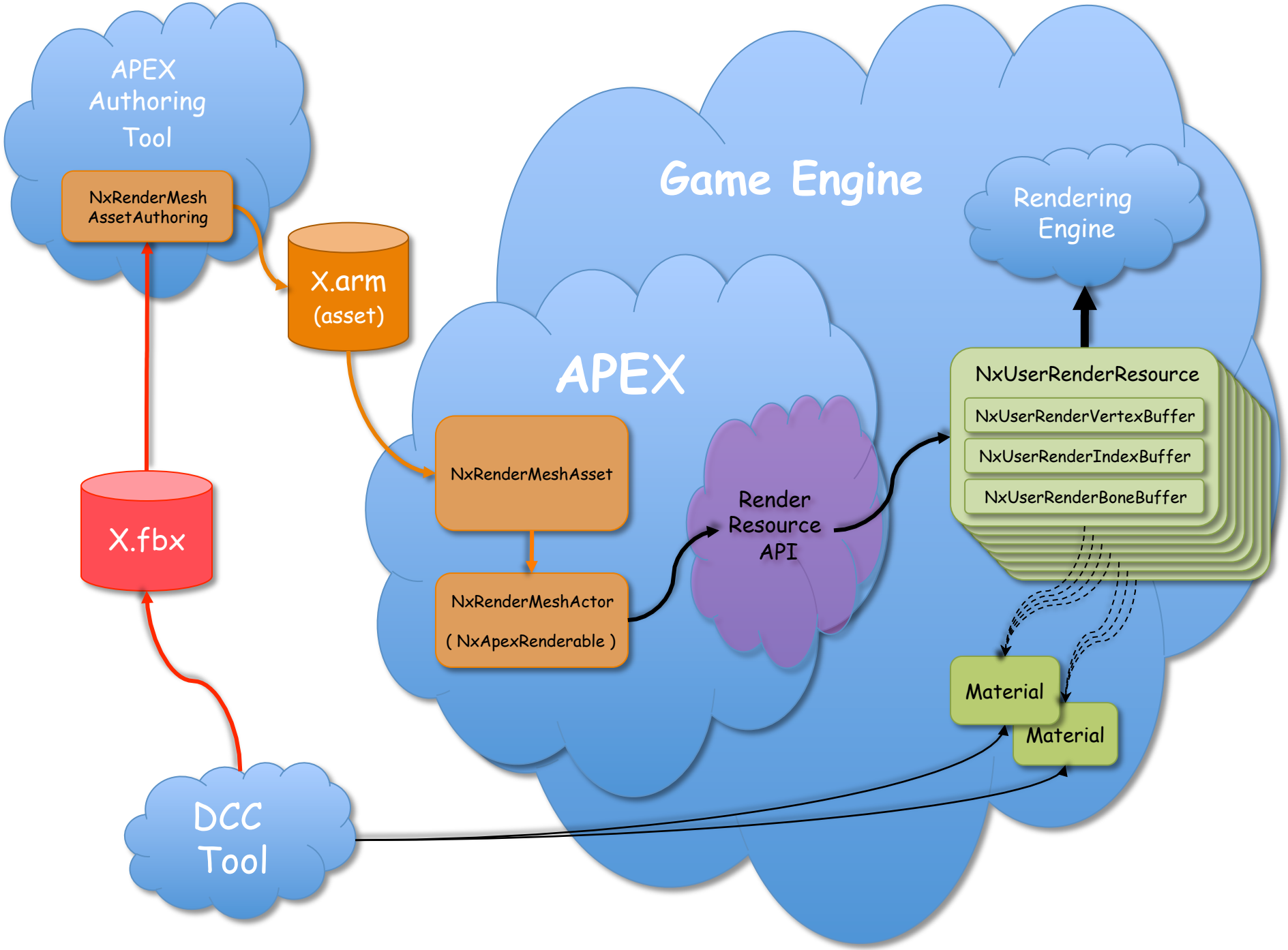
The APEX Render Mesh Asset and the Render Resources Interface

Jean Pierre Bordes

NVIDIA

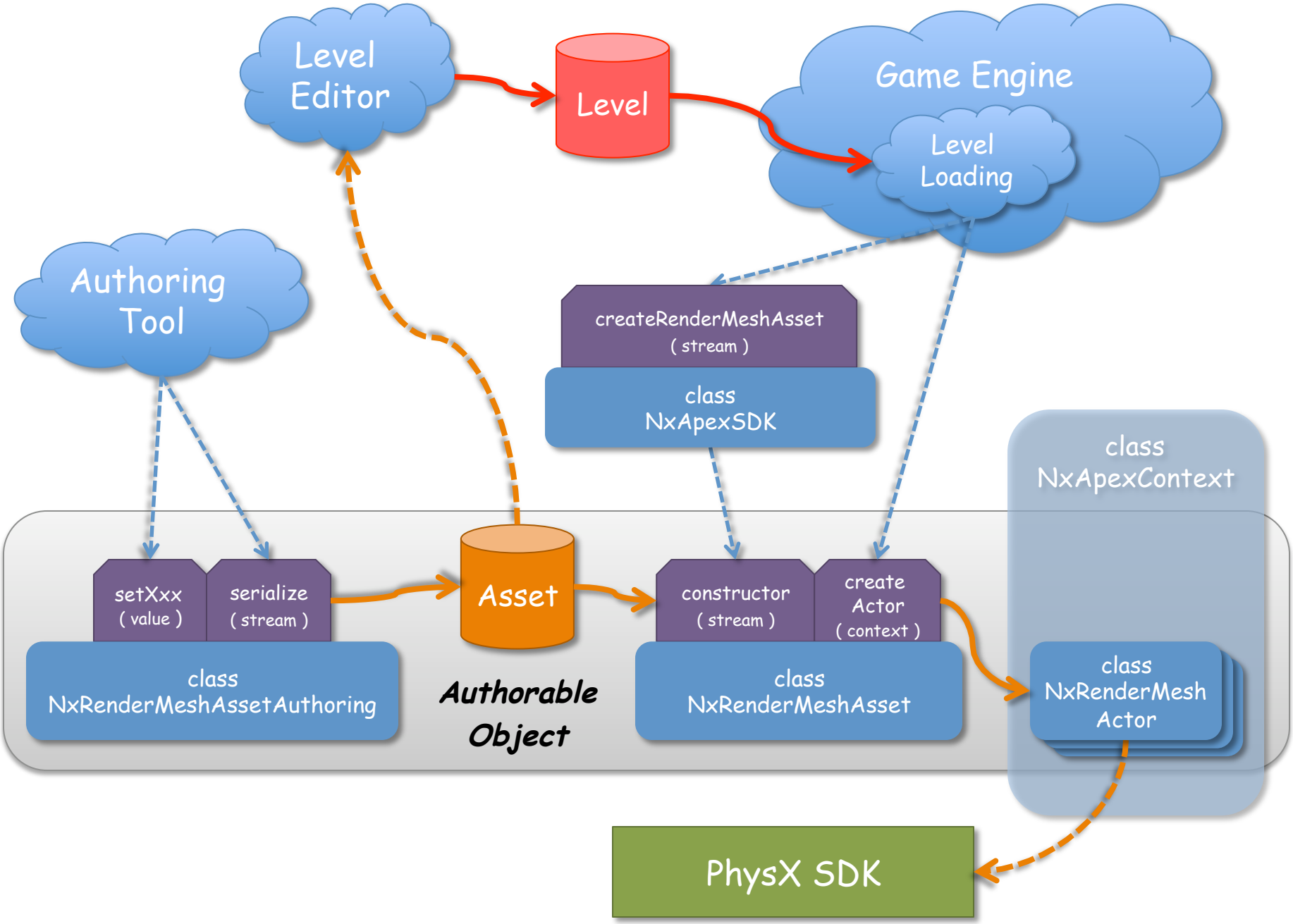
Introduction

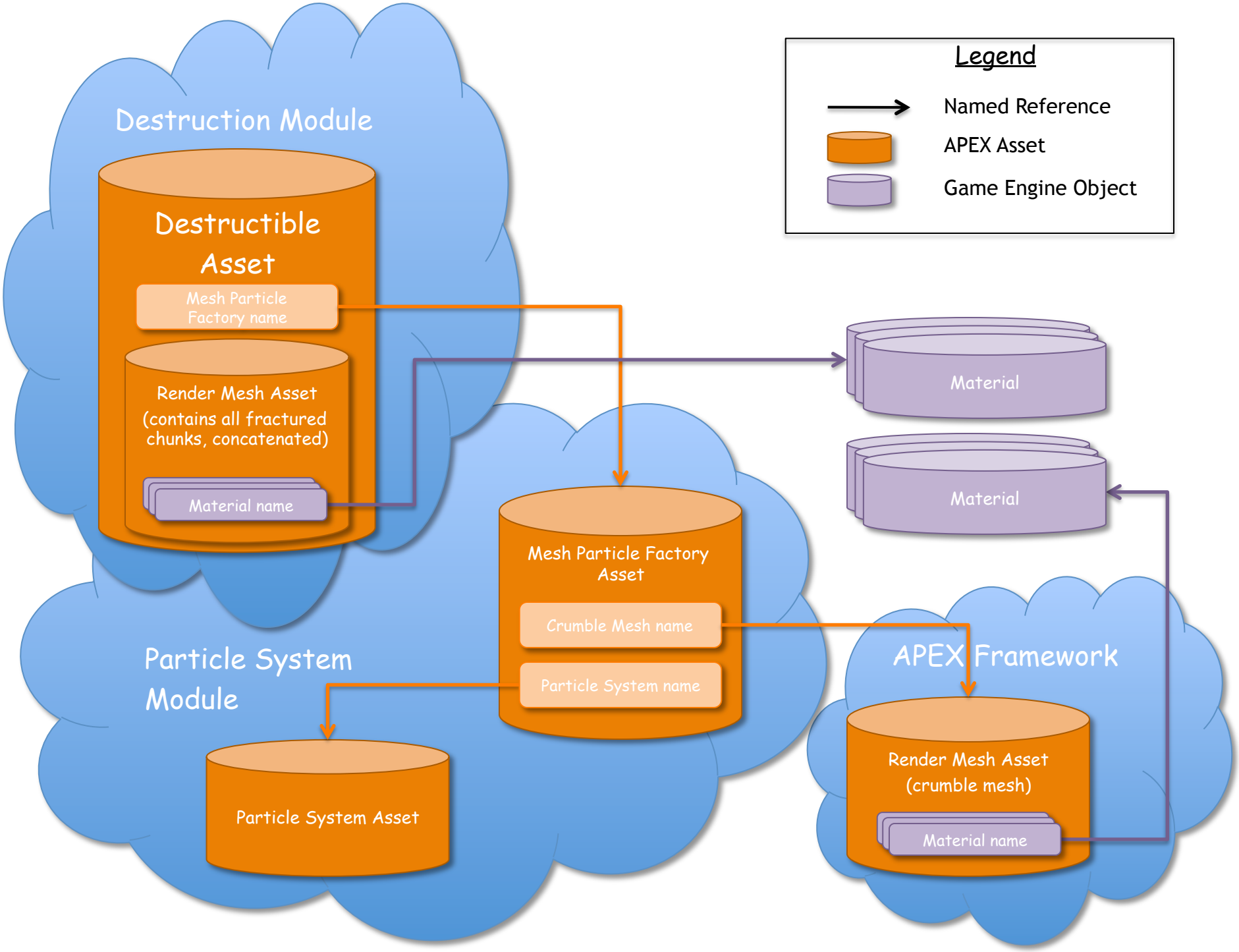
- Render Mesh Asset
  - Authoring and storage of meshes
- Render Resources Interface
  - A way for APEX to output graphics data

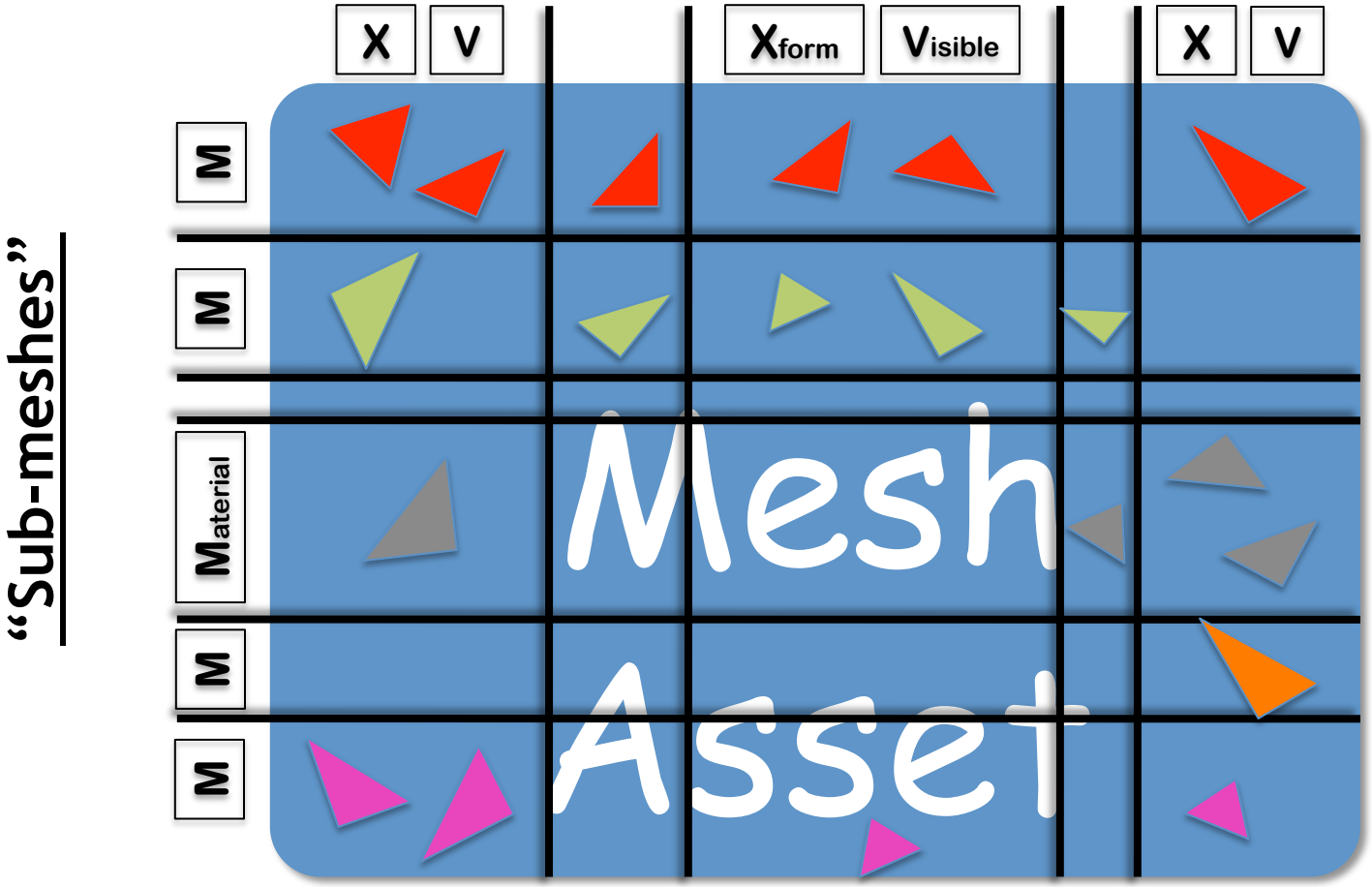

APEX
Authoring
Tool
NxRenderMesh
AssetAuthoring
X.arm
(asset)
X.fbx
DCC
Tool
Game Engine
APEX
NxRenderMeshAsset
NxRenderMeshActor
(NxApexRenderable)
Render
Resource
API
Rendering
Engine
NxUserRenderResource
NxUserRenderVertexBuffer
NxUserRenderIndexBuffer
NxUserRenderBoneBuffer
Material
Material

APEX Render Mesh

- Primary purpose: Generic Mesh Asset
  - Authoring and storage of arbitrary renderable meshes
- Three classes in APEX framework API:
  - NxRenderMeshAsset
    - The asset, at run-time
  - NxRenderMeshAssetAuthoring
    - For creating the asset, in an authoring tool
  - NxRenerMeshActor
    - An instance of the asset in a game level

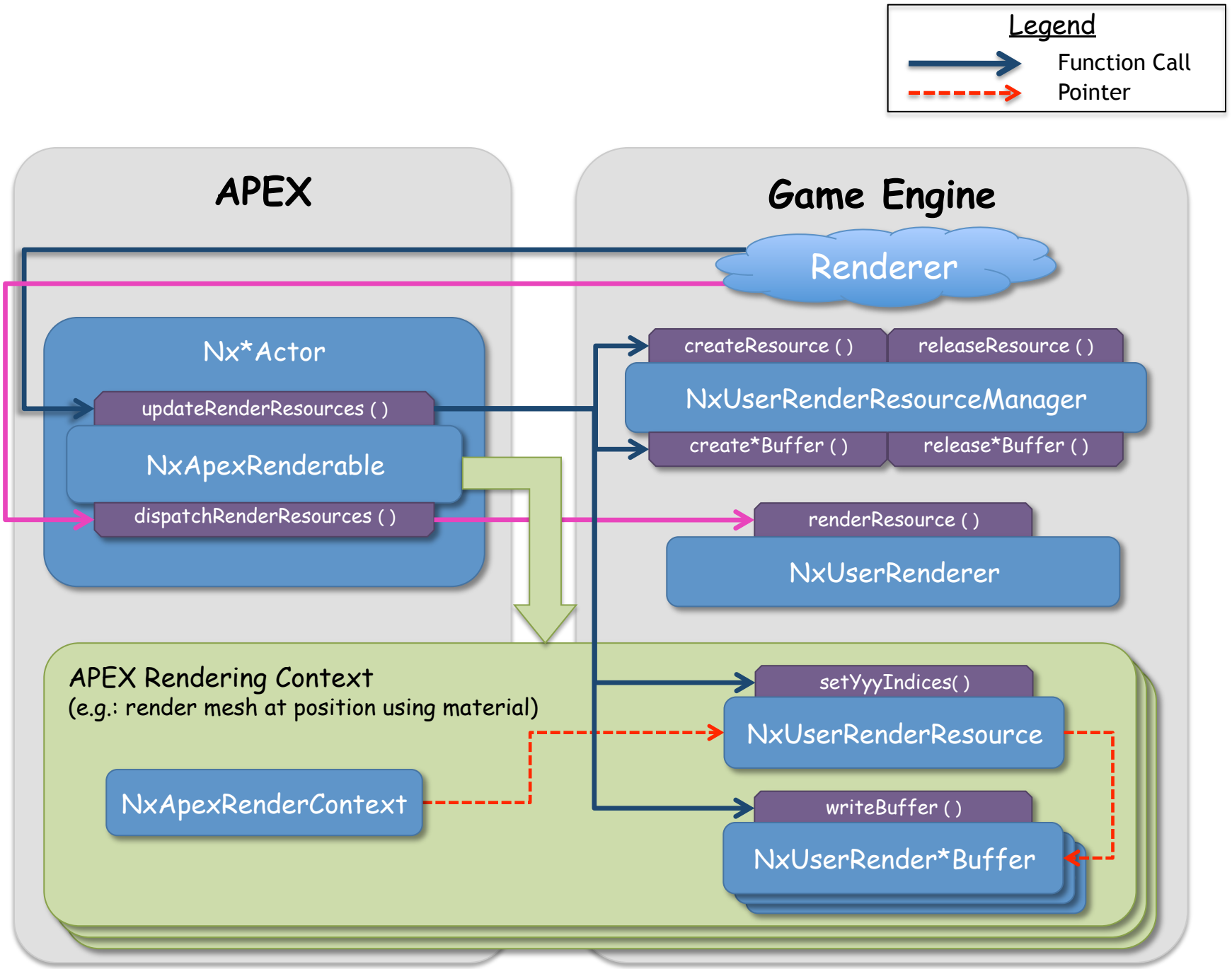

Level Editor
Level
Game Engine
Level Loading
Authoring Tool
createRenderMeshAsset (stream)
class NxApexSDK
class NxApexContext
setXxx (value)
serialize (stream)
Asset
constructor (stream)
create Actor (context)
class NxRenderMeshAssetAuthoring
Authorable Object
class NxRenderMeshAsset
class NxRenderMesh Actor
PhysX SDK


Legend
Named Reference
APEX Asset
Game Engine Object
Destruction Module
Destructible Asset
Mesh Particle Factory name
Render Mesh Asset (contains all fractured chunks, concatenated)
Material name
Material
Material
Mesh Particle Factory Asset
Crumble Mesh name
Particle System name
Particle System Module
Particle System Asset
APEX Framework
Render Mesh Asset (crumble mesh)
Material name


"Parts"
"Sub-meshes"
X
V
Xform
Visible
X
V
M
M
Material
M
M
Mesh
Asset

APEX Render Resources API

- Interface to game's rendering engine
  - A way for APEX to output graphics data
  - Portable and efficient interface
- Renderable APEX objects derive from
  - NxApexRenderable
- Game must implement several classes
  - NxUserRenderResourceManager
  - NxUserRenderer
  - NxUserRenderResource
  - NxUserRender*Buffer


Legend
Function Call
Pointer
APEX
Game Engine
Renderer
Nx*Actor
updateRenderResources ( )
NxApexRenderable
dispatchRenderResources ( )
createResource ( )
releaseResource ( )
NxUserRenderResourceManager
create*Buffer ( )
release*Buffer ( )
renderResource ( )
NxUserRenderer
APEX Rendering Context
(e.g.: render mesh at position using material)
NxApexRenderContext
setYyyIndices( )
NxUserRenderResource
writeBuffer ( )
NxUserRender*Buffer

NxUserRenderResourceManager

```
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 );
};
```

NxUserRender*

```
class NxUserRenderXxxBuffer
{
public:
virtual void writeBuffer( NxRenderVertexSemantic::Enum semantic, void *srcData,
    NxU32 srcStride,
                NxU32 firstDestElement, NxU32 numElements )

};
```

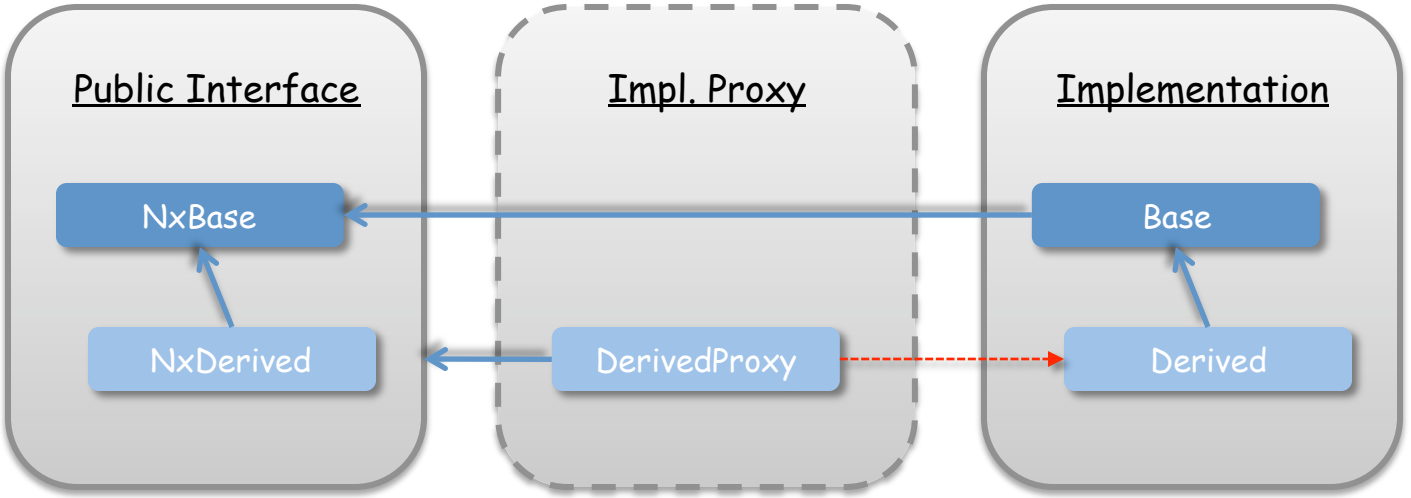
```
class NxUserRenderer
{
public:
virtual void renderResource( const NxApexRenderContext &context ) = 0;
};
```

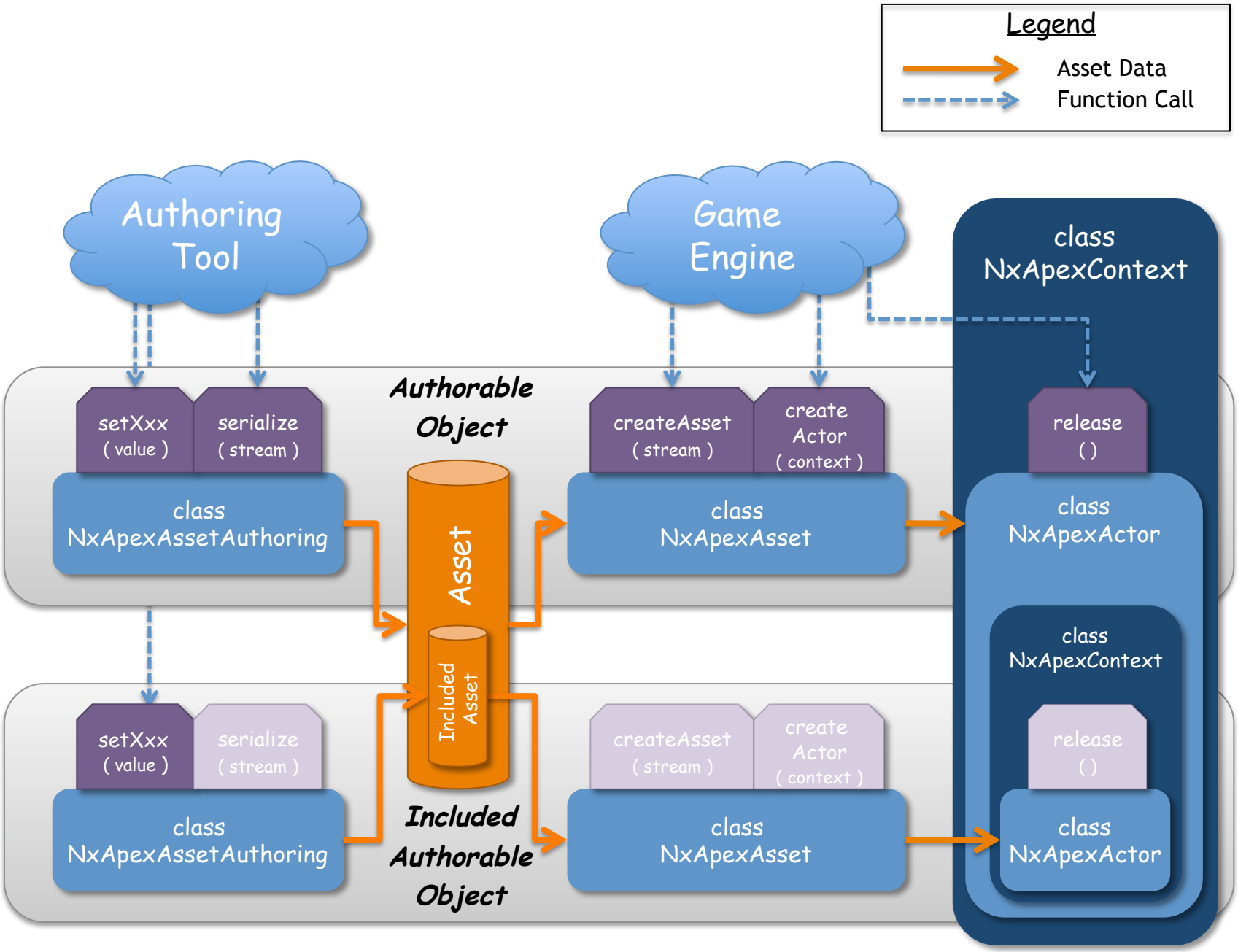
Game Rendering Engine

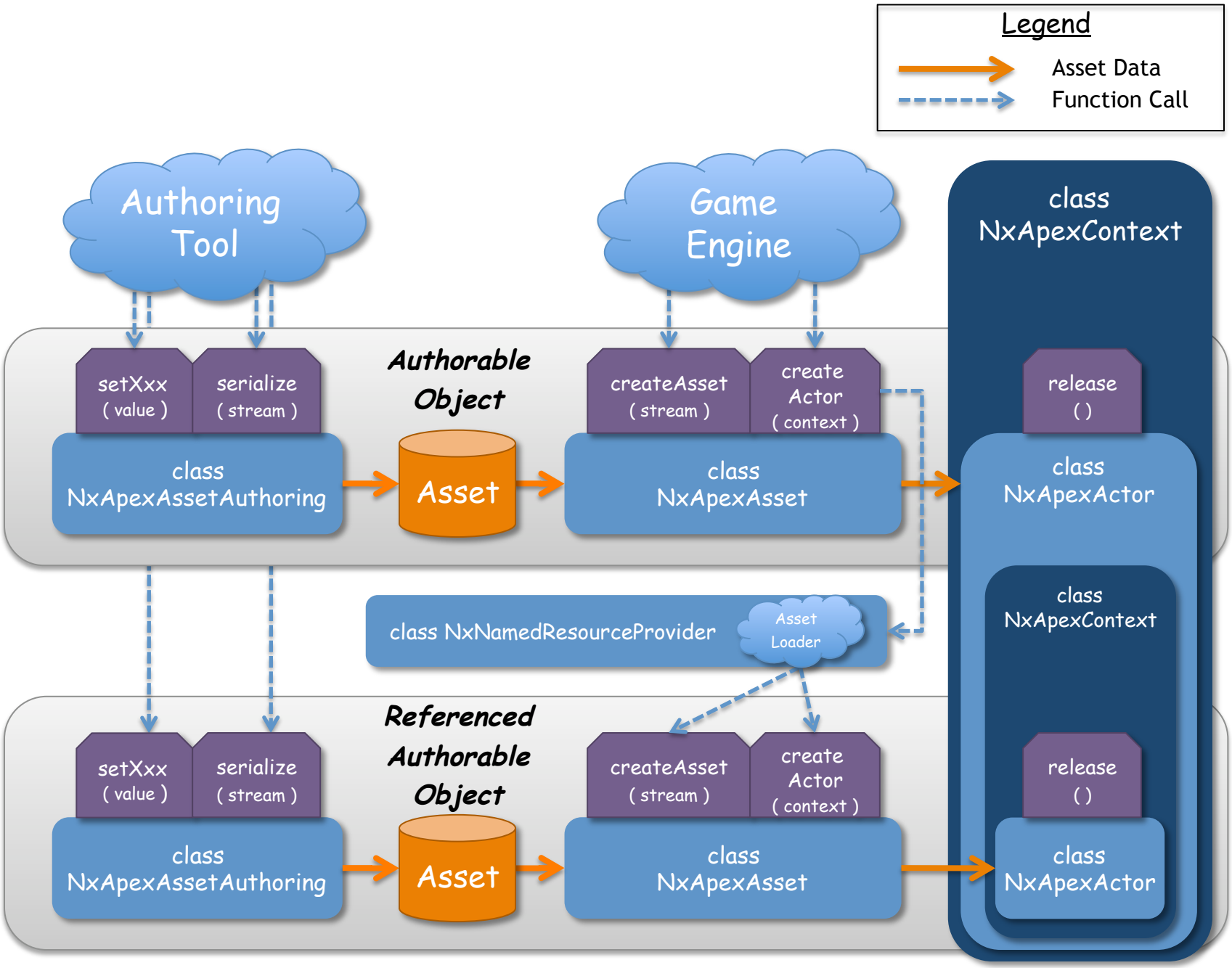
```
MyCachedRenderer renderer;

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();
```

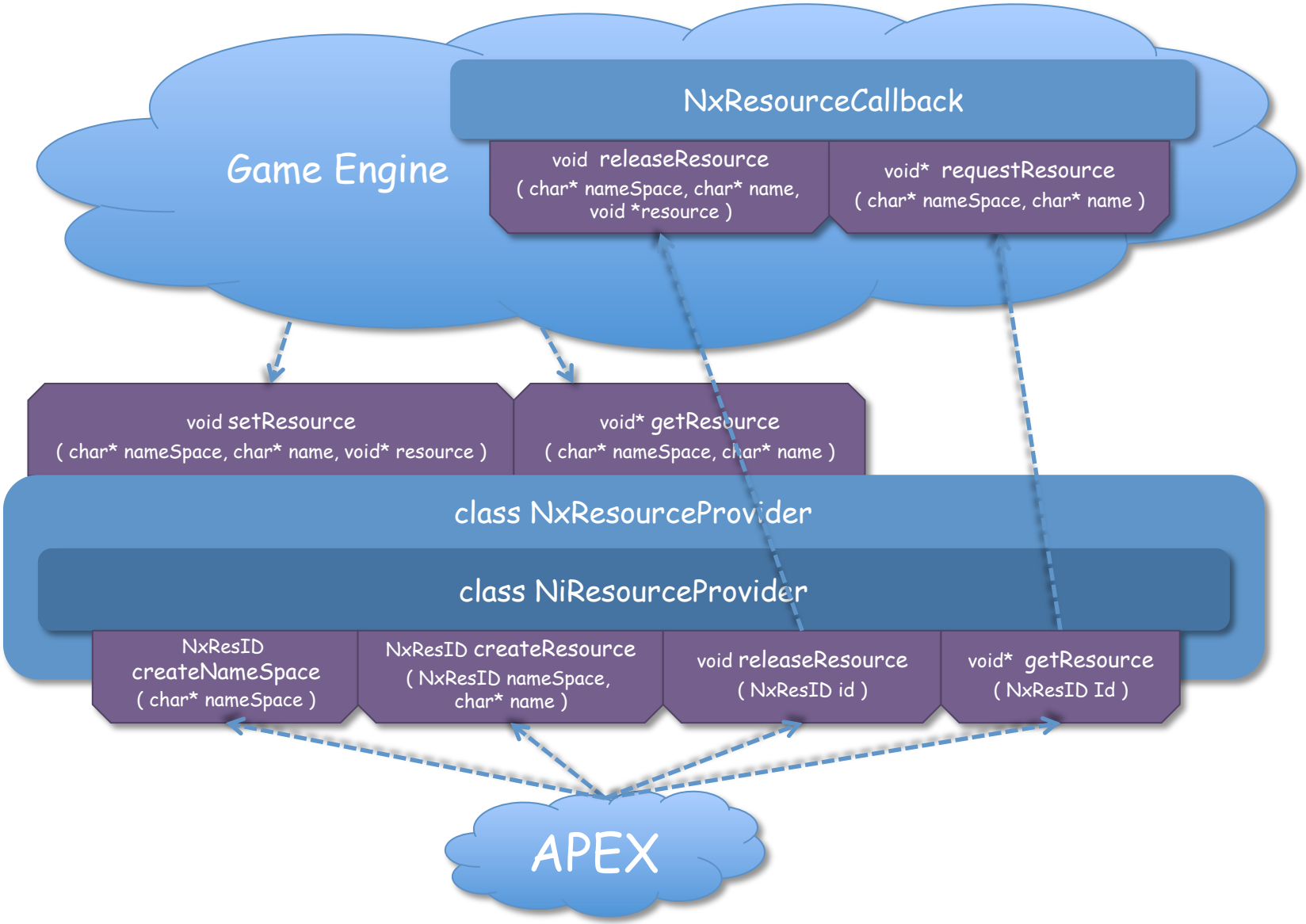
Backup Slides

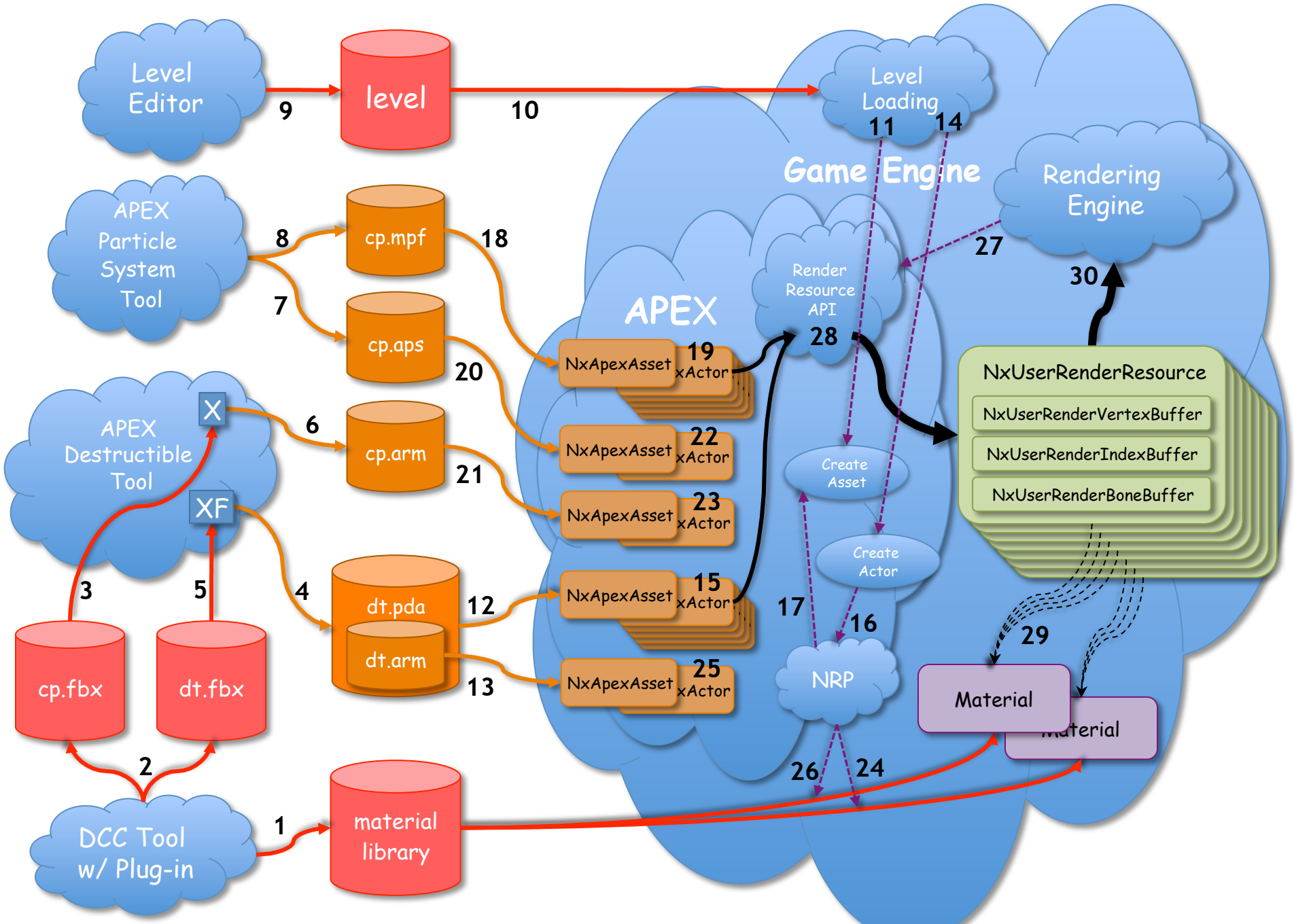
PIMPL - Pointer IMPLementation




Legend
Asset Data
Function Call
Authoring Tool
Game Engine
class NxApexContext
Authorable Object
setXxx (value)
serialize (stream)
createAsset (stream)
create Actor (context)
release ( )
class NxApexAssetAuthoring
Asset
Included Asset
class NxApexAsset
class NxApexActor
class NxApexContext
setXxx (value)
serialize (stream)
createAsset (stream)
create Actor (context)
release ( )
class NxApexAssetAuthoring
Included Authorable Object
class NxApexAsset
class NxApexActor


Legend
Asset Data
Function Call
Authoring Tool
Game Engine
class NxApexContext
Authorable Object
setXxx (value)
serialize (stream)
createAsset (stream)
create Actor (context)
release ( )
class NxApexAssetAuthoring
Asset
class NxApexAsset
class NxApexActor
class NxApexContext
class NxNamedResourceProvider
Asset Loader
Referenced Authorable Object
setXxx (value)
serialize (stream)
createAsset (stream)
create Actor (context)
release ( )
class NxApexAssetAuthoring
Asset
class NxApexAsset
class NxApexActor


Game Engine
NxResourceCallback
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


Level Editor
level
9
10
Level Loading
11
14
Game Engine
Rendering Engine
APEX Particle System Tool
8
7
cp.mpf
cp.aps
18
20
APEX
Render Resource API
27
30
28
APEX Destructible Tool
X
XF
6
cp.arm
21
NxApexAsset
19
22
23
15
25
xActor
NxUserRenderResource
NxUserRenderVertexBuffer
NxUserRenderIndexBuffer
NxUserRenderBoneBuffer
Create Asset
Create Actor
3
5
4
dt.pda
dt.arm
12
13
17
16
29
cp.fbx
dt.fbx
NRP
Material
2
26
24
DCC Tool w/ Plug-in
1
material library

Authoring

1. An artist uses the DCC tool to author a material library (textures, shaders, etc…) for use with the destructible object: outside faces, inside faces, and crumble particles.
2. An artist authors a mesh for the destructible object (“dt.fbx”) and a mesh for the crumble particles (“cp.fbx”), and export these meshes into FBX format files
3. The artist uses the APEX Destructible Tool to import the destructible object mesh from the FBX file, then authors the fracture parameters, and fractures the mesh.
4. The artist exports the fractured mesh as an APEX Destructible Asset (“dt.pda”). All APEX Assets generated by the APEX tools can optionally be encapsulated in a game specific wrapper file by implementing a user defined “stream” class (derived from NxStream) and providing it in “apexuser.dll”.
5. The artist uses the APEX Destructible Tool to import the crumble particle mesh from the FBX file.
6. The artist exports the crumble particle mesh as an APEX Render Mesh Asset (“cp.arm”).
7. The artist uses the APEX Particle System Tool (currently command-line only) to create and save an APEX Particle System Asset (cp.aps). This particle system will be used to simulate the crumble mesh (debris) particles.
8. The artist uses the APEX Particle System Tool to create and save an APEX Mesh Particle Factory Asset (cp.aps). This asset will be used to link the crumble Render Mesh Asset and the Particle System Asset to the Destructible Asset (Figure 2).
9. The level designer uses the level editor to create multiple instances of the destructible object in the level. The level editor stores the position and orientation of each instance in the level, as well as the name of the Destructible Asset (“dt”).

Runtime – Level Load

10. The game engine's level loading code reads the level file.
11. For every unique destructible object name referenced in the level, the level loader calls NxModuleDestructible::createDestructibleAsset(..).
12. This causes the asset file ("dt.pda") to be loaded, and a NxDestructibleAsset to be created.
13. The Destructible Asset contains a Render Mesh Asset, which is also loaded, and a NxRenderMeshAsset is created.
14. Next, the level loader calls NxDestructibleAsset::createDestructibleActor(..) for each instance of a destructible object in the level.
15. Each such call creates a NxDestructibleActor.
16. Since the DestructibleAsset references, by name, a Mesh Particle Factory Asset, the Named Resource Provider (NRP) is called to resolve the name to a pointer.
17. Since the Mesh Particle Factory Asset has not yet been loaded, the NRP user callback calls NxModuleParticles::createMeshParticleFactoryAsset(..) to load the asset.
18. This causes the asset file ("cp.mpf") to be loaded, and a NxMeshParticleFactoryAsset to be created.
19. Next, a Mesh Particle Factory Actor is created for each Destructible Actor that is created.
20. Since the Mesh Particle Factory Asset references, by name, a Particle System Asset, NRP is called to resolve the name to a pointer. Since the Particle System Asset has not yet been loaded, the NRP user callback calls NxModuleParticles::createApexParticleSystemAsset(..) to load the asset.
21. Since the Mesh Particle Factory Asset also references, by name, a Render Mesh Asset, the NRP is called to resolve the name to a pointer. Since the Render Mesh Asset has not yet been loaded, the NRP user callback calls NxApexSDK:: createRenderMeshAsset(..) to load the asset.
22. Next, a Particle System Actor is created from the newly loaded Particle System Asset.
23. Then, a Render Mesh Actor is created from the newly loaded Render Mesh Asset.
24. Since the Render Mesh Asset references, by name, a material, the NRP is called to resolve the name to a pointer. Since the material has not yet been loaded, the NRP user callback loads the material.
25. Then, a Render Mesh Actor is created from the Render Mesh Asset that was loaded back in step 13.
26. Since that Render Mesh Asset references, by name, a material, the NRP is called to resolve the name to a pointer. Since the material has not yet been loaded, the NRP user callback loads the material.

Runtime - Rendering

27. The rendering engine calls the APEX Render Resources API to render each NxDestructibleActor, using it's base class: NxApexRenderable. First, it calls NxApexRenderable::updateRenderResources(..), then it calls NxApexRenderable::displatchRenderResources(..).
28. The initial call to updateRenderResources(..) results in callbacks to the rendering engine to create instances of user-defined classes derived from the pure virtual base classes NxUserRenderXxxBuffer and NxUserRenderResource. Then, updateRenderResources(..) generates callbacks to the rendering engine to write data (e.g.: vertex buffers, index buffers) to the user-defined buffer classes (derived from NxUserRenderXxxBuffer). These calls to the writeBuffer(..) method copy the geometry data from APEX internal data structures into the user defined buffers in the NxUserRenderResource data structure.
29. When the NxUserRenderResource's are created, a void pointer to a "material" is provided, which can be stored in the user-defined class derived from NxUserRenderResource.
30. Finally, the rendering engine renders using the geometry and material reference now contained in the NiMesh/NxUserRenderResource data structure.