Integration Overview

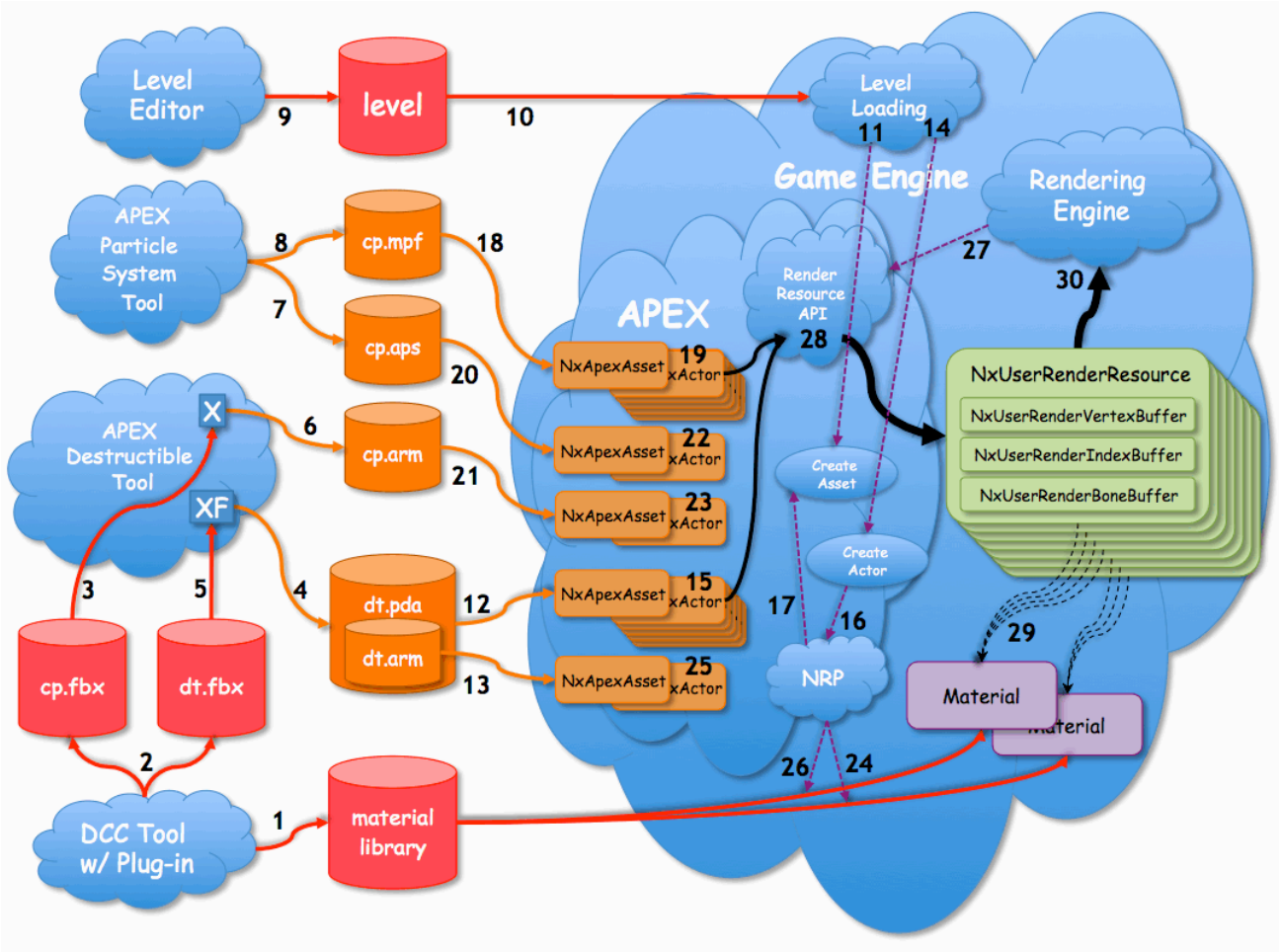
APEX™

November 2008

Table of Contents

Game Engine Integration

Diagram



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.
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.

Notice

Trademarks

Copyright

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