

**SOFTWARE DESIGN DOCUMENT (SDD)
Delta3D Game and Simulation Engine**

**Delta3D AGEIA PhysX™ Integration
(dtAgeiaPhysX)**

**Version 0.3
14 November 2007**

Prepared by:

Alion Science and Technology

BMH Operation

5365 Robin Hood Road, Suite 100

Norfolk, VA 23513

(757) 857-5670

Table of Contents

1	Overview	3
2	Installation	3
3	Physics System.....	4
3.1	<i>The World Component.....</i>	4
3.1.1	Physics Scenes	4
3.1.2	Physics Materials.....	5
3.1.3	Physics Helpers.....	6
3.1.4	Other component functionality	6
3.2	<i>The Game Actor Interface</i>	8
3.2.1	Initialization.....	9
3.2.2	Syncing up between Physics and Models	10
3.2.3	Actor Types provided for you.....	11
3.3	<i>The Physics Helper.....</i>	11
3.3.1	Initialization and making your primitives.....	11
3.3.2	The callbacks.....	14
3.4	<i>Additional Physics Helpers</i>	15

Table of Figures

Figure 1	Physics in Action.....	4
Figure 2	Using the NxAgeiaDebugDraw Class.....	7
Figure 3	The Visual Remote Debugging Utility	8

1 Overview

This document defines the high level design of the Delta3D physics library, dtAgeiaPhysX. This library is the basic integration of the AGEIA PhysX engine. This design is intended as an overview of some of the basic features of the dtAgeiaPhysX library. This library should be used as part of a Delta3D application and can be accessed via the Delta3D-Extras repository using Subversion at the following location:

<https://delta3d-extras.svn.sourceforge.net/svnroot/delta3d-extras/dtAgeiaPhysX/trunk>

Note – this design document is intended to provide a basic overview of the dtAgeiaPhysX library. It is not a complete explanation of classes, methods, or functionality. Before reading this design document, it is recommended that you consult the Game Manager tutorial, the Dynamic Actor Layer (DAL) tutorial, and the STAGE tutorial to gain a better understanding of the underlying technology. These tutorials can be found at <http://www.delta3d.org/article.php?story=20050720155458456&topic=tutorials>. In addition, it will be helpful to consult the AGEIA API documentation to understand the basic concepts used by the PhysX™ engine. Finally, to use this software, you will need to run the AGEIA system drivers, SDK/API, and tool installers provided in the AGEIA directory of the SVN repository.

2 Installation

After downloading from SVN, you will notice a folder titled ‘AgeiaInstallers’. This folder contains the sdk installations from Ageia needed to get you going. As of the time of this writing the release of 2.7 is in there. Before you go clicking on every file you see WAIT, and please read all of this first! There are 3 files to install, a SystemSoftware, which lets software emulation run without the hardware, and enables hardware if you have a PhysX card; SDK Core, that allows you to have the include lib bin folders you will link to during development; and the Tools, which includes Rocket, Remote Debugger, and other knickknacks.

The SystemSoftware should only be installed if you do not already have your physics card installed and up to date with the latest drivers. If you already have drivers that are up to date, you do not need to install this, and all of your projects will run fine without going back to this revision.

After all of those 3(or 2) installations are complete, you made need to make an AGEIA_INC and AGEIA_LIB environment variable for ease of development use. These point to the SDK’s root several include directories, the sdk’s lib directory respectively.

3 Physics System

The physics system described by this document is the AGEIA™ PhysX™ engine. AGEIA is a commercial grade physics API used by top-tier console platforms and game development companies. For this implementation we use a combination of three pieces: the physics Component, the physics actor helper, and the Game Actor.

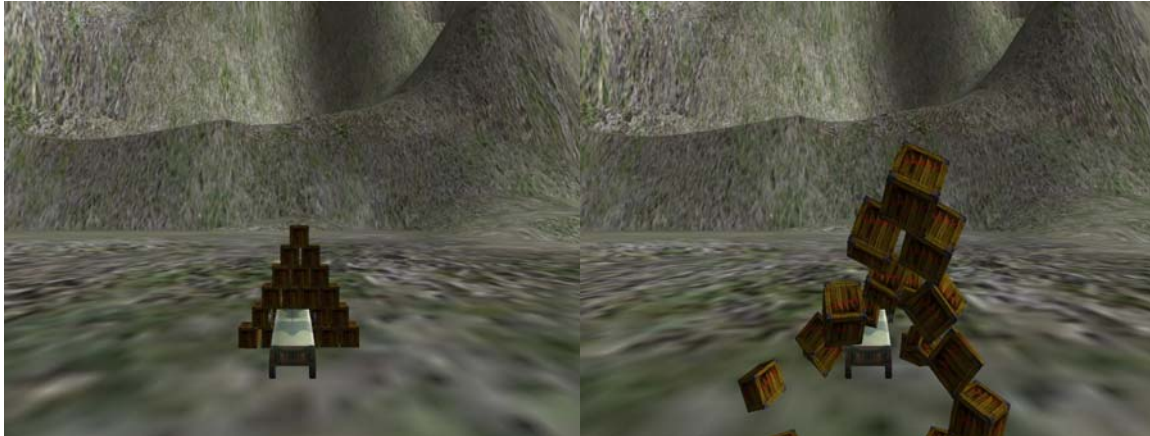


Figure 1 Physics in Action

3.1 The World Component

The NxAgeiaWorldComponent starts up the engine and holds onto all the physics objects we use to define our visual scene. Cached meshes and multiple scenes are also stored here, and updates are done per frame for the scenes, and the character controller manager.

3.1.1 Physics Scenes

These scenes include NxActors, NxJoints, and other Ageia specific modules. We only use one “default” scene. However, as needed, an application can create additional scenes to prevent physics geometries from interacting with physics geometries in other scenes. The default scene is made upon initialization with the name “Default”. You can make a new scene by calling CreateScene(). You can get a physics scene by calling GetPhysicsScene() and passing the name of the scene you want. Specifying a scene that doesn’t exist will return the “Default” scene. The physics Helper uses these scenes directly to load its primitives.

Functions on the component that interact with the scenes:

```
/**
 * Purpose: Create a scene with a scenedesc allowing multiple scenes.
 *   Not needed unless you want more than one scene (default provided).
 */
void CreateScene(const std::string& Name, NxSceneDesc& SceneDesc,
                bool DebugInfo, NxVec3 gravity = NxVec3(0,0,-9.81));

/**
 * Purpose: Delete a scene, not needed unless you manage multiple scenes.
```

```
    */
void DeleteScene(const std::string& Name);

/**
 * Purpose: Delete all scenes that have been made.
 */
void DeleteScenes();

/**
 * Purpose : Gets a scene with a specific name (default is "Default").
 * @return The physics scene.
 */
NxScene& GetPhysicsScene(const std::string& Name);

/**
 * Purpose : Gets the NxAgeiaScene struct for a specific scene name.
 * @return The physics scene struct
 */
NxAgeiaScene& GetPhysicsSceneStruct(const std::string& Name);
```

3.1.2 Physics Materials

Along with scene creation and management the World Component holds onto our Ageia specific materials that many actors share. These materials are actors themselves and are linked to our physics actors during runtime. The values they hold are Dynamic Friction (how easy for something to continue to slide once its already sliding), Static Friction (how easy for something to start sliding from rest), and Restitution (bounciness of an object).

For reference the functions that interact with the materials are as follows:

```
/// Register a material
bool RegisterMaterial(const dtCore::UniqueId& uniqueID);

/// Will return true if loaded in, false if couldnt load material in.
bool RegisterMaterial(const std::string& NameToUse, float restitution,
                    float dynamicFriction, float staticFriction,
                    const std::string& SceneToLoadIn = "Default");

/// Return the material if it exists, else gives the default (0) material.
NxMaterial& GetMaterial(const std::string& NameToRetrieve,
                    const std::string& SceneToLoadIn = "Default",
                    bool ambiguityFlag = true);

/// Return the material if it exists, else gives the default (0) material
NxMaterial& GetMaterial(const dtCore::UniqueId& uniqueID,
                    const std::string& SceneToLoadIn= "Default");

/// Get the material index for a specific material.
int GetMaterialIndex(const std::string& NameToRetrieve,
                    const std::string& SceneToLoadIn = "Default",
                    bool ambiguityFlag = true);

/// Get the material index for a specific material
int GetMaterialIndex(const dtCore::UniqueId& uniqueID,
```

```
const std::string& SceneToLoadIn = "Default");
```

3.1.3 Physics Helpers

The component manages all objects that have physics by tracking the Physics Helpers. Each actor that wants to have physics will create one of these and then register it with the Component when they enter the world. This section describes the basic helper methods on the component. See the Actor section below for more information on how to use helpers.

```
/**
 * Purpose: Registers an actor with the component. The
 *          component will give the geometry to PhysX and begin ticking it.
 * @param   toAdd - the helper that will get updated
 */
void RegisterAgeiaHelper(NxAgeiaPhysicsHelper &toAdd);

/**
 * Purpose: Used for removing an actor from the component
 * @param   toRemove - the helper to remove from being updated
 */
void UnRegisterAgeiaHelper(NxAgeiaPhysicsHelper &toRemove);

/**
 * Purpose: Used for removing all actors out of the list
 */
void ReleaseAllAgeiaHelpers();
```

3.1.4 Other component functionality

The primary job of the physics component is to call the appropriate physics behavior in the PhysX engine. This work is done from the component's tick local. Each tick, the component goes through all helpers, validates the current helper / actor objects, updates the objects with PRE_UPDATE condition flags (usually kinematic remote objects), updates the physics engine, and finally to gives the post-update so the helpers can visually move actors (transforms) to match the physics engine.

Although it is not inherently obvious, physics behaviors are not performed directly on your real scene. Instead, the PhysX engine builds its own instance of your geometries and moves them in its own 'scene' space. Objects in this space are always in 'world' coordinates and are optimized in hundreds of ways to handle collision detection, linear and angular velocity, and kinematic interactions. In general, the PhysX engine does not care about most of the things that you are rendering. In order to make this work, you need to tell the component enough information to recreate the physics geometry. This is done via the helper object (see below). Once you have a helper object with 'cooked' geometry, the component can give it to the physics engine and let it do its magic. Each frame, the component gives PhysX a chance to move objects and compute collisions. The resulting collisions and transformations are handled by the helper. Note that

neither Ageia, nor the component will directly move your objects in the visual scene (ie, Open Scene Graph). That is the helper's job and is usually done automatically for you.

There are also a number of tools that have been brought over for use with the physics engine. These include:

- NxAgeiaDebugDraw class – draws with osg modeled physics geometry with lines. Turn true flag on in TickLocal.



Figure 2 Using the NxAgeiaDebugDraw Class

- RemoteDebugger Access – on application startup if the remote debugger is running, your Delta3D can try to connect to the remote debugger. The PhysX remote debugger is fantastically powerful debugging tool. Although the interface is a bit tricky to get used, the tool allows you to capture and analyze all physics behaviors in your scene during runtime.

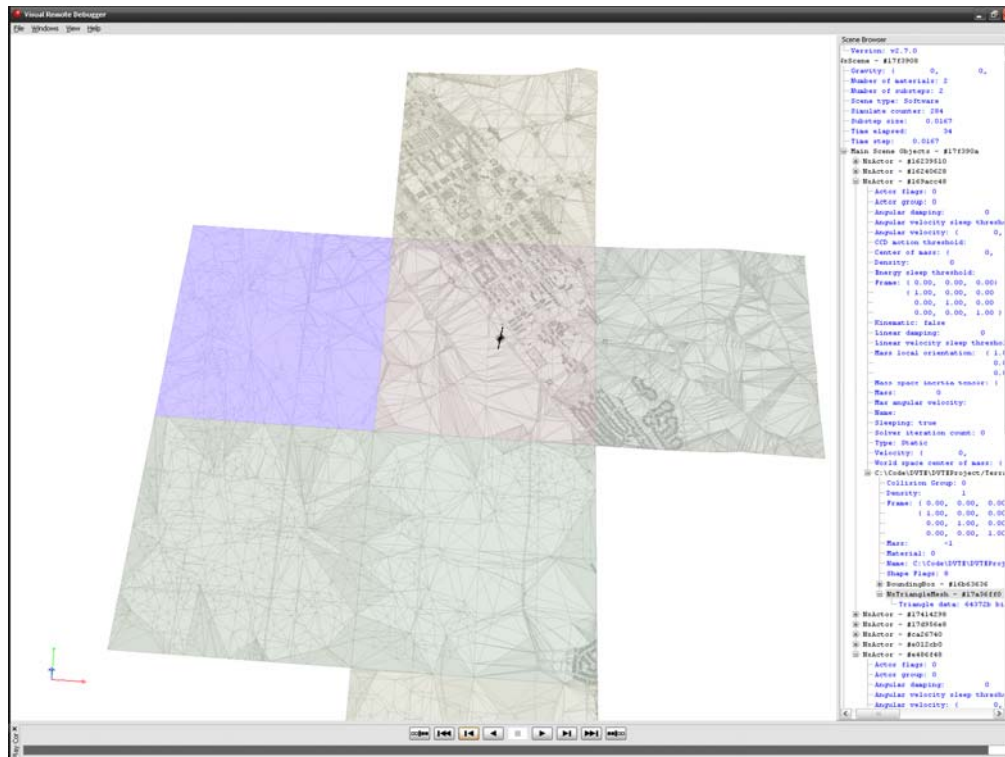


Figure 3 The Visual Remote Debugging Utility

- `NxAgeiaErrorReport` class – gives us AGEIA specific error reports or our use of AGEIA specific error reports during runtime.
- `NxAgeiaContactReport` class – reports back to our actors which actors were hit
- `NxAgeiaRaycastReport` class – allows you to do raycasts in your world to see if something will intersect one of the physics bodies. This behavior is extremely efficient and is much more optimized than the Delta3D `ISector` behavior. This behavior is also useful if you are doing extremely high velocity objects (such as bullets) that do not perform well with rigid-body-collisions.

3.2 The Game Actor Interface

This design is structured to specifically avoid creating large class hierarchies. Specifically, the goal is to allow developers to use whatever classes they want. Therefore, there is no base Ageia Game Actor that you need to inherit from. Instead, to add physics behavior to one of your game actors, you will need to create a member Physics Helper (see below) and add the 3 simple methods defined in `NxAgeiaPhysicsInterface`. Note that all physics objects derive from this interface class.

```
/**
```

```
* /brief Purpose: Expose the base physics methods for actors.
*/
class DELTA_AGEIA_PHYSX_EXPORT NxAgeiaPhysicsInterface
{
public:
    friend class NxAgeiaPhysicsHelper;

public:
    /// Corresponds to the AGEIA_FLAGS_PRE_UPDATE flag
    virtual void AgeiaPrePhysicsUpdate() = 0;

    /// Corresponds to the AGEIA_FLAGS_POST_UPDATE
    virtual void AgeiaPostPhysicsUpdate() = 0;

    /// Corresponds to the AGEIA_FLAGS_GET_COLLISION_REPORT
    virtual void AgeiaCollisionReport(ContactReport& contactReport,
        NxActor& ourSelf, NxActor& whatWeHit) = 0;

    // You would have to make a new raycast to get this report,
    // so no flag associated with it.
    virtual void AgeiaRaycastReport(const NxRaycastHit& hit,
        const NxActor& ourSelf, const NxActor& whatWeHit) = 0;
};
```

By implementing this interface, the component is able to check the correct flags the actor sets on its helper and call the necessary functions. The most commonly used behaviors are the `PrePhysicsUpdate()`, `PostPhysicsUpdate()`, and `CollisionReport()` functions.

3.2.1 Initialization

Although you do not need to follow any particular class hierarchy, there are several things you need to do to initialize your game actor to be used by the `NxAgeiaWorldComponent`. The most important thing is to create a Physics Helper. The physics helper is the link between the actor and the component. You will also need to set a material property and the AGEIA Flags. These settings will allow your objects to sync up in certain states.

```
/**
 * Purpose: AGEIA_FLAGS are used to give physics behavior to an object
 * without having a huge class heirarchy.
 */
enum AGEIA_FLAGS
{
    /// defaulted to this on constructor
    AGEIA_FLAGS_DEFAULT_NO_FLAGS = 0,

    /// user pre update is called
    AGEIA_FLAGS_PRE_UPDATE = 1,

    /// user post update is called
    AGEIA_FLAGS_POST_UPDATE = 2,

    /// So we call the callback
    AGEIA_FLAGS_GET_COLLISION_REPORT = 4,
```

```
    /// radius check but no bounce back - not implemented
    AGEIA_FLAGS_GET_TRIGGER_REPORT = 8,

    /// run through shapes - not implemented
    AGEIA_FLAGS_GET_ENTITY_REPORT = 16,

    /// this is for characters
    AGEIA_FLAGS_GET_SHAPE_REPORT = 32,

    /// when a character touches another character
    AGEIA_FLAGS_GET_CHARACTER_REPORT = 64,

    /// get a collision report and respond to movements
    AGEIA_NORMAL = //AGEIA_FLAGS_DEFAULT_MOVE_ON_COLLISION |
                  AGEIA_FLAGS_GET_COLLISION_REPORT,

    /// remote objects you're not moving
    AGEIA_REMOTE = AGEIA_FLAGS_GET_COLLISION_REPORT |
                  AGEIA_FLAGS_PRE_UPDATE,

    /// defaults for a character
    AGEIA_CHARACTER = AGEIA_FLAGS_GET_SHAPE_REPORT |
                     AGEIA_FLAGS_GET_CHARACTER_REPORT,

    /// on collision move, pre, post, and collision notice
    AGEIA_ALL =     AGEIA_FLAGS_PRE_UPDATE |
                   AGEIA_FLAGS_POST_UPDATE |
                   AGEIA_FLAGS_GET_COLLISION_REPORT,

    /// no more
    AGEIA_FLAGS_MAX_FLAGS
};
```

After this is done you need to call a `SetCollision****()` method on the helper (we'll get to more of this later). Then you have to call `RegisterAgeiaHelper` on the component and pass in your helper. Once that's done your object will be in the physics scene and you are all set.

3.2.2 Syncing up between Physics and Models

Having a physics model in the scene is fine and dandy, but we also want to see results! If you set your flags correctly (`AGEIA_COLLIDABLE`), every frame the physics engine updates your collision primitive. It's not until the `POST_UPDATE` is called on the helper (from the component) that your object is synced up. You should not have to do this manually, but if you want to the flag `AGEIA_FLAGS_POST_UPDATE` needs to be set.

In addition to the basic geometry, the helper can also help you sync up a model with multiple Degree of Freedom's (DOFs) or pieces. You don't need to create a new actor for every DOF. Instead, the physics helper can load onto each piece of geometry separately and updates them all for you. For instance, you could have a single actor that is tracking many physics particles that could all be handled by a single physics helper.

3.2.3 Actor Types provided for you.

Many different types of physics behaviors are available in the basic NxAgeiaPhysicsHelper classes. In addition, several types of physics behaviors that are more complex have been wrapped up into actors. These actors are provided as examples or can be used as they are. Note, some of these classes are provided in the SimulationCore repository under Delta3D-extras (SVN directory is <https://delta3d-extras.svn.sourceforge.net/svnroot/delta3d-extras/SimulationCore/trunk>).

- NxAgeiaFourWheelVehicleActor – implements a basic 4-wheel vehicle (like a truck or HMMWV). Provides many properties that you can tweak to make all sorts of vehicles. This class is available as part of the SimCore project.
- NxAgeiaParticleSystemActor – spawns 3d and 2d particle systems with physics enabled. This actor creates new physics objects based on direction, force values, timers, and such. You can create a massive amount of interaction in the scene this way. This class is available as part of the SimCore project.
- NxAgeiaMunitionsPSysActor – similar to the above actor, but is more targeted toward weapons fire such as firing a bullet or grenade.
- NxAgeiaRemoteKinematicActor – This class is used for objects that are controlled remotely – ie, they need to be collided with but cannot be directly moved by our physics engine. This is usually used in simulations where we can't move the object, but we need to collide with it. This class is available as part of the SimCore project.
- NxAgeiaTerraPageLandActor – advanced loader of terrain that makes heightfields from data we get in when paging in from a large Terrapage database. Heightfields off the base and buildings are made into triangle meshes.

3.3 The Physics Helper

So we have a component and we have actors, but how do we make them interact? To do this, you need the physics helper class that goes between the two. The helper is the connection between your game actor and the physics component and provides a LOT of behavior to help you make physics work. The physics helper is responsible to set up your data, link actors together, expose callbacks, and so on. To make this work, you need to create one and hold onto it as a member variable on your actor. The next step is to initialize the physics.

3.3.1 Initialization and making your primitives.

To want to make a primitive in your scene you use the following basic functions:

```
/**
 * Purpose   : To build a land heightfield from a mesh,
 *   Outs    : NxAgeiaBaseActor& p - filled in
 * @param    osg::Node* node - the drawing node object
 * @param    SceneName - name of the scene you want to add to
 */
NxActor* SetCollisionMeshHeightField(osg::Node* node,
    NxCollisionGroup collisionGroup = 0, const std::string& SceneName =
    "Default", const std::string& ActorName = "Default",
    int HeightFieldValue = -100) ;
```

```
/**
 * Purpose : To build a bounding box around an object,
 *   Outs : NxAgeiaBaseActor& p - filled in
 * @param  NxVec3* coord- where to place
 * @param  NxVec3* dimensions- on each side
 * @param  float Density - weight of the object
 * @param  SceneName - name of the scene you want to add to
 */
NxActor* SetCollisionBox(const NxVec3& coord, const NxVec3& dimensions,
    float Density, float Mass, NxCollisionGroup collisionGroup = 0,
    const std::string& SceneName = "Default",
    const std::string& ActorName = "Default", bool enableCCD = false);

/**
 * Purpose : To make a flat surface actor with infinite direction,
 *   Outs : NxAgeiaBaseActor& p - filled in
 * @param  NxVec3* coord- where to position the obj (doesnt really matter)
 * @param  Up - up vector of the ground you want to have
 * @param  SceneName - name of the scene you want to add to
 */
NxActor* SetCollisionFlatSurface(const NxVec3& coord, const NxVec3& up,
    NxCollisionGroup collisionGroup = 0,
    const std::string& SceneName= "Default",
    const std::string& ActorName = "Default");

/**
 * Purpose : To add a sphere in the world,
 *   Outs : NxAgeiaBaseActor& p - filled in
 * @param  NxVec3* coord - where to place the sphere
 * @param  float radius - radius the sphere has
 * @param  float Density - density the sphere has
 * @param  SceneName - name of the scene you want to add to
 */
NxActor* SetCollisionSphere(const NxVec3& coord, float radius,
    float Density, float Mass, NxCollisionGroup collisionGroup = 0,
    const std::string& SceneName= "Default",
    const std::string& ActorName = "Default", bool enableCCD = false);

/**
 * Purpose : To set a cylinder up to collide with,
 *   Outs : NxAgeiaBaseActor& p - filled in
 * @param  NxVec3* coord - coord that will be the center location
 * @param  float height - basic cylinder height
 * @param  float radius - basic cylinder radius
 * @param  float density - mass of object
 * @param  NxScene& m_Scene - scene the actor goes in
 * @param  SceneName - name of the scene you want to add to
 */
NxActor* SetCollisionCapsule(const NxVec3& coord, float height,
    float radius, float density, float Mass,
    NxCollisionGroup collisionGroup = 0,
    const std::string& SceneName= "Default",
    const std::string& ActorName = "Default");

/**
 * Purpose : To load in a mesh object and have the triangles
```

```
*      collidable without using a primitive,
* @param  osg::Node* node - the osg node for triangle data
* @param  NxVec3* coord - where the xyz are
* @param  collisiongroup - can only collide or not collide/callbacks etc,
* @param  useCaching models that will need to be cached
* @param  resource name - used for caching
* @param  SceneName - name of the scene you want to add to
* @param  ActorName - name of the actor for reference / physics model.
*/
NxActor* SetCollisionStaticMesh( osg::Node* node, const NxVec3& coord,
    bool useCaching = false, const std::string& resourceName = "",
    const std::string& SceneName= "Default", const std::string&
    ActorName = "Default", NxCollisionGroup collisionGroup = 0);

/**
* Purpose : To load in a mesh object and have the triangles
*      collidable without using a primitive,
* @param  osg::Node* node - the osg node for triangle data
* @param  NxMat34 - matrix offset of the item
* @param  float density - weight of the object
* @param  collisiongroup - can only collide or not collide/callbacks etc,
* @param  useCaching models that will need to be cached
* @param  resource name - used for caching
* @param  SceneName - name of the scene you want to add to
* @param  ActorName - name of the actor for reference / physics model.
*/
NxActor* SetCollisionConvexMesh( osg::Node* node, const NxMat34& mat,
    float density, float Mass, bool useCaching = false,
    const std::string& resourceName = "", const std::string& SceneName=
    "Default", const std::string& ActorName = "Default",
    NxCollisionGroup collisionGroup = 0);
```

Everything in the helpers is done by names of objects. Since you can have -n- number amount of objects on any helper the naming of objects is up to the user. However once something is loaded to an NxActor through one of the SetCollision Methods, you may then reference it by that name. GetPhysXObject(name)..

Why might you want to do this however you might ask? Well simply enough once you have the NxActor you are able to do all of ageia's functions on that actor. For example that is internally called in the helper when you want to bind two or more of your primitives together using joints. The joints are also in the helper and can provide you with flexibility and uniqueness in your simulation. They are as follows:

```
///  
//brief Purpose: Make a joint, puts onto list  
void CreateFixedJoint(NxActor* a0, NxActor* a1,  
    const NxVec3& globalAnchor, const NxVec3& globalAxis,  
    const std::string& SceneName = "Default");  
  
///  
//brief Purpose: Make a joint, puts onto list  
void CreateRevoluteJoint(NxActor* a0, NxActor* a1,  
    const NxVec3& globalAnchor, const NxVec3& globalAxis,  
    const std::string& SceneName = "Default");  
  
///  
//brief Purpose: Make a joint, puts onto list
```

```
void CreateSphericalJoint(NxActor* a0, NxActor* a1,
    const NxVec3& globalAnchor, const NxVec3& globalAxis,
    const std::string& SceneName = "Default");

///  
//brief Purpose: Make a joint, puts onto list
void CreatePrismaticJoint(NxActor* a0, NxActor* a1,
    const NxVec3& globalAnchor, const NxVec3& globalAxis,
    const std::string& SceneName = "Default");

///  
//brief Purpose: Make a joint, puts onto list
void CreateCylindricalJoint(NxActor* a0, NxActor* a1,
    const NxVec3& globalAnchor, const NxVec3& globalAxis,
    const std::string& SceneName = "Default");

///  
//brief Purpose: Make a joint, puts onto list
void CreatePointOnLineJoint(NxActor* a0, NxActor* a1,
    const NxVec3& globalAnchor, const NxVec3& globalAxis,
    const std::string& SceneName = "Default");

///  
//brief Purpose: Make a joint, puts onto list
void CreatePointInPlaneJoint(NxActor* a0, NxActor* a1,
    const NxVec3& globalAnchor, const NxVec3& globalAxis,
    const std::string& SceneName = "Default");

///  
//brief Purpose: Make a joint, puts onto list
void CreatePulleyJoint(NxActor* a0, NxActor* a1, const NxVec3& pulley0,
    const NxVec3& pulley1, const NxVec3& globalAxis,
    NxReal distance, NxReal ratio, const NxMotorDesc& motorDesc,
    const std::string& SceneName = "Default");

///  
//brief Purpose: Make a joint, puts onto list
void CreateDistanceJoint(NxActor* a0, NxActor* a1, const NxVec3& anchor0,
    const NxVec3& anchor1, const NxVec3& globalAxis,
    const std::string& SceneName = "Default");

///  
//brief Purpose: Make a joint, puts onto list
void CreateRopeSphericalJoint(NxActor* a0, NxActor* a1,
    const NxVec3& globalAnchor, const NxVec3& globalAxis,
    const std::string& SceneName = "Default");

///  
//brief Purpose: Make a joint, puts onto list
void CreateClothSphericalJoint(NxActor* a0, NxActor* a1,
    const NxVec3& globalAnchor, const NxVec3& globalAxis,
    const std::string& SceneName = "Default");

///  
//brief Purpose: Make a joint, puts onto list
void CreateBodySphericalJoint(NxActor* a0, NxActor* a1,
    const NxVec3& globalAnchor, const NxVec3& globalAxis,
    const std::string& SceneName = "Default");
```

3.3.2 The callbacks.

As mentioned above you have 3 callbacks that are called through the component to helper to actor. The functions are CallCollisionCallback(), CallPreUpdate(), and CallPostUpdate(). These should never have to be changed.

3.4 Additional Physics Helpers

You will notice whilst browsing through the code base of dtAgeiaPhysX there are also several other helper classes. These other helper classes provide their own hierarchy of physics objects. For example: the NxAgeiaPrimitivePhysicsHelper derives from the base, and has basic default parameters to just load collision volumes for one object; a place you might want to use this is to load one vehicle in as its on physics helper.

There are also the class NxAgeiaFourWheelVehicleHelper and NxAgeiaCharacterHelper. These classes respectively do what their names imply, as basic driving functionality is done through the vehicle, and the character helper is a place where your character can reside its physics.

Note: There are also several other callback functionalities for the character. For any class you will need to call the SetBaseInterfaceClass for the appropriate callbacks, and for characters SetCharacterInterfaceClass will get you the methods you need.