Let's take a look at PhysXTerrainDemo:

The PhysX Terrain sample is an extension of the Terrain Demo that adds PhysX functionality. The user can shoot boxes which interact with the terrain based on gravity, restitution and friction forces. The simulation will make use of PhysX hardware if available. Otherwise, it falls back to software simulation. The scene was generated with Scene Designer, where the physics-related properties (friction and restitution) were specified as meta-data on the materials in the scene. The remainder of the application is standard Gamebryo-PhysX. The remaining sections give a Functional Overview, and a Code Overview.

There's a lot of code in TerrainDemo.cpp (1255 lines) and about six times less in NiTerrainSortProcessor.

Let's look at Terrain Demo: only 865 lines in TerrainDemo with NiTerrainSortProcessor unchanged.

The Terrain Demo demonstrates the terrain engine functionality in Gamebryo. It shows seamless integration with the NiEntity system and the ability to load GSA scene files containing terrain and various other entities. The demo allows one to walk about the terrain, view it in wire frame in order to observe the automatic level of detail management, and detach from the terrain to fly around the scene. Some extra features are also demonstrated in this demo, including a flexible meta-data system integrated into the terrain engine, and how to place decals on the terrain.

The NiEntity library implements a framework in which scenes can be loaded, saved, and manipulated. A scene contains a set of entities, which represent game objects, and optionally a set of selection sets. Entities can have any number of properties, and they provide an interface for accessing those properties. Selection sets are collections of entities and provide a way to logically group entities to facilitate easy access. This library is used by the Gamebryo Scene Designer to allow for entity placement and property editing. Scenes are saved out from the Scene Designer to files that can be loaded by the NiEntity library in a final application. A scene file contains all the information necessary to reconstruct all the entities in the scene with their saved property values. An application can then interact with entities in the scene and their properties by using the appropriate interface functions.
The following sections detail the code. This section assumes familiarity with NiApplication.

```cpp
#include "TerrainDemo.h"
#include <NiAnimation.h>
#include <NiMain.h>
#include <NiCollision.h>
#include <NiMeshLib.h>
#include <NiUIManager.h>
#include <NiUICheckBox.h>
#include <NiUISlider.h>
#include <NiMath.h>
#include <NiLicense.h>
#include <NiTerrainComponent.h>
#include <NiWaterComponent.h>
#include "NiTerrainSortProcessor.h"

NiEmbedGamebryoLicenseCode;

//----------------------------------------------------------------------------
NiApplication* NiApplication::Create()
{
    return NiNew TerrainDemo();
}

//----------------------------------------------------------------------------
TerrainDemo::TerrainDemo() :
    NiSample("Terrain Demo", DEFAULT_WIDTH, DEFAULT_HEIGHT, true)
{
    m_bUseNavSystem = true;
    m_bSnapToTerrain = true;
    m_spSkyDome = NULL;
    m_pkSceneView = NULL;
    m_kToggleWireframe.Initialize(this, &TerrainDemo::ToggleWireframe);
    m_kToggleTerrainSnap.Initialize(this, &TerrainDemo::ToggleSnapToTerrain);
    m_kCameraSpeed.Initialize(this, &TerrainDemo::SetCameraSpeed);
    m_kPlaceDecal.Initialize(this, &TerrainDemo::PlaceDecal);

    m_spWireframeProp = NiNew NiWireframeProperty();
    m_spWireframeProp->SetWireframe(false);
}

//----------------------------------------------------------------------------
bool TerrainDemo::CreateRenderer()
{
    NILOG("Creating the renderer ...\n");

    if (!NiApplication::CreateRenderer())
        return false;

    m_spRenderer->SetBackgroundColor(NiColor(1.0f, 1.0f, 1.0f));

    NILOG("Finished\n");
    return true;
}
```

Basic initialization starts here.
bool TerrainDemo::Initialize()
{
    m_spUpdateProcess = NiNew NiMeshUpdateProcess();
    m_spUpdateProcess->SetUpdateControllers(true);

    if (!NiSample::Initialize()) return false;
    return true;
}

bool TerrainDemo::CreateScene()
{
    NILOG("Creating the scene ...\n");

    NIASSERT(NiDataStream::GetFactory());                       // [1]
    NiDataStream::GetFactory()->SetCallback(                    // [2]
        NiDataStreamFactory::ForceCPUReadAccessCallback);

    m_spScene = NiNew NiNode();
    m_spScene->AttachProperty(m_spWireframeProp);

    m_spErrors = NiNew NiDefaultErrorHandler();
    m_spAssetManager = NiNew NiExternalAssetManager(NiFactories::GetAssetFactory());

    m_spSkyDome = CreateSkyDome();
    NIASSERT(m_spSkyDome);

    NiEntityStreamingAscii stream;

    // Attempt to load a scene containing our terrain
    if (!stream.Load(ConvertMediaFilename("terrain.gsa")))
    {
        NiMessageBox("Failed to load main scene file", "Error");
        return false;
    }

    m_spEntityScene = stream.GetSceneAt(0);
    NIASSERT(m_spEntityScene);

    // Update the scene to initialize the components
    m_spEntityScene->Update(0.0f, m_spErrors, m_spAssetManager);

    // Load the terrain after we have finished updating the components.
    NiTerrainComponent::LoadTerrain(m_spEntityScene, m_spAssetManager, m_spErrors);

    // Show the errors.
    NiUInt32 uiNumErrors = m_spErrors->GetErrorCount();
    if (uiNumErrors)
    {
        for (NiUInt32 ui = 0; ui < uiNumErrors; ++ui)
            NiMessageBox(m_spErrors->GetErrorMessage(ui), "Error!");

        return false;
    }
    // ... to be continued on the next page ...

Note, in order for the terrain geometry to be created, either the NiTerrainComponent::LoadTerrain or NiTerrain::Load function must be called. An advantage of using NiTerrainComponent::LoadTerrain is that it automatically loops over all the entities in the scene, gets a pointer to their scene graph hierarchy and applies the terrain global fog property. The global fog property is stored statically in the NiTerrain object,
thus allowing the same fog settings to be applied to all objects in the world. Note, a limitation with this release forces the user to apply and adjust this property in code and not from Scene Designer.

```cpp
// ... (continued from previous page) ...

NiTerrain* pkTerrain = NULL;
NiDirectionalLight* pkSunlight = NULL;
NiObject* pkObject = NULL;
NiUInt32 uiCount = m_spEntityScene->GetEntityCount();
for (NiUInt32 ui = 0; ui < uiCount; ui++)
{
    NiEntityInterface* pkEntity = m_spEntityScene->GetEntityAt(ui);
    if (!pkEntity->GetPropertyData("Scene Root Pointer", pkObject))
        continue;
    if (NiIsKindOf(NiCamera, pkObject))
        m_spCamera = NiDynamicCast(NiCamera, pkObject->Clone());
    else if (NiIsKindOf(NiTerrain, pkObject))
    {
        pkTerrain = NiDynamicCast(NiTerrain, pkObject);
        pkEntity->Update(pkEntity, 0, m_spErrors, m_spAssetManager);
    }
    NIVERIFY(pkTerrain);
    NIVERIFY(m_spCamera);
    // Create our sample decal texture.
    m_spDecalTexture = NiSourceTexture::Create(ConvertMediaFilename("Decal.tga"));
    AdjustCameraAspectRatio(m_spCamera);
    m_spCamera->Update(0.0f);
    m_spSkyDomeCamera = NiDynamicCast(NiCamera, m_spCamera->Clone());
    NIASSERT(m_spSkyDomeCamera);
    m_spSkyDomeCamera->SetTranslate(m_spSkyDomeCamera->GetTranslate());
    NILOG("Finished\n");
    return true;
}
CreateScene ends here.

UpdateFrame: This method is responsible dealing with terrain intersection queries.

```cpp
void TerrainDemo::UpdateFrame()
{
    NiSample::UpdateFrame();
    m_spErrors->ClearErrors();
    m_spEntityScene->Update(m_fAccumTime, m_spErrors, m_spAssetManager);
    m_spUpdateProcess->SetTime(m_fAccumTime);
    m_spScene->Update(*m_spUpdateProcess);
    // ... continued below ...
```

UpdateFrame: This method is responsible dealing with terrain intersection queries. Collision queries against the terrain are performed using NiTerrainInteractor objects. The interactor object is obtained
through a call to NiTerrain::GetInteractor() or a property on the NiTerrainComponent. The following code illustrates how one may retrieve these interactor objects and issue a ray cast against the terrain. The result is a "snap to terrain" or simple terrain following behavior.

```cpp
// ... (continued from previous page) ...

if (m_bSnapToTerrain)
{
    NiTerrainInteractor* pkInteractor = NULL;
    NiTListIterator kIter =
        NiTerrainComponent::GetTerrainComponents()->GetHeadPos();

    NiPoint3 kRayStart =
        m_spCamera->GetWorldTranslate() + NiPoint3(0, 0, FLT_MAX);
    NiRay kRay = NiRay(kRayStart, NiPoint3(0, 0, -1));

    while (kIter)
    {
        NiTerrainComponent* pkToCheck =
            NiTerrainComponent::GetTerrainComponents()->GetNext(kIter);

        size_t stSize;
        void* pvInteractor = NULL;
        pkToCheck->GetPropertyData("Interactor", pvInteractor, stSize);
        pkInteractor = (NiTerrainInteractor*)pvInteractor;

        NIASSERT(pkInteractor);
        if (pkInteractor->Collide(kRay))
        {
            NiPoint3 kColPoint, kColNorm;
            kRay.GetIntersection(kColPoint, kColNorm);

            m_spCamera->SetTranslate( kColPoint.x,
                                      kColPoint.y,
                                      kColPoint.z + 5.0f);

            DisplayMetaData(pkInteractor, kColPoint, kRay.GetCollidedLeaf());

            m_spCamera->Update(m_fAccumTime);
        }
    }
}
//----------------------------------------------------------------------
void TerrainDemo::EndUpdate()
{
    NiSample::EndUpdate();
    //... (to be continued) ...
```

End of UpdateFrame. Note, the code actually loops over all possible terrain components. This version of the terrain engine is limited to a single terrain component therefore making the loop unnecessary.
#ifndef TERRAINDEMO_H
#define TERRAINDEMO_H

#include <NiSample.h>
#include <NiEntity.h>
#include <NiTerrainLib.h>
#include <NiMesh.h>

class TerrainDemo : public NiSample
{
    public:
        TerrainDemo();
        virtual bool CreateRenderer();
        virtual bool CreateScene();
        virtual bool CreateFrame();
        virtual bool CreateUIElements();
        virtual bool Initialize();
        virtual void UpdateFrame();
        virtual void EndUpdate();
        virtual void Terminate();

    protected:
        void ToggleWireframe(bool bEnable);
        void ToggleSnapToTerrain(bool bEnable);
        void SetCameraSpeed(float fSpeed);
        void PlaceDecal();
        void DisplayMetaData(NiTerrainInteractor* pkInteractor, const NiPoint3 & kIntersectionPt, const NiTerrainDataLeaf * pkLeaf);

        NiMesh* CreateSkyDome();

    private:
        NiUIMemberSlot1<TerrainDemo, bool> m_kToggleWireframe;
        NiUIMemberSlot1<TerrainDemo, bool> m_kToggleTerrainSnap;
        NiUIMemberSlot1<TerrainDemo, float> m_kCameraSpeed;
        NiUIMemberSlot0<TerrainDemo> m_kPlaceDecal;

        NiDefaultErrorHandlerPtr m_spErrors;
        NiExternalAssetManagerPtr m_spAssetManager;
        NiScenePtr m_spEntityScene;
        NiMeshPtr m_spSkyDome;
        NiCameraPtr m_spSkyDomeCamera;
        NiTexturePtr m_spDecalTexture;
        NiUILabel* m_pkMetadataLabel;
        NiSceneRenderView* m_pkSceneView;
        NiVisibleArray m_kVisibleSet;

        Ni3DRenderView* m_pkGrassView;
        NiMeshUpdateProcessPtr m_spUpdateProcess;

        NiWireframePropertyPtr m_spWireframeProp;
        float m_fCameraSpeed;
        bool m_bSnapToTerrain;
};
#endif

Briefly looking up what we have seen thus far in TerrainDemo.h (highlighted). About 1/4th of the code.
Time to step back and mention some basic facts:

NiMemObject

Description: NiMemObject implements interfaces overriding the global new and delete operators. All derived classes will now funnel their allocations/deallocations through the NiMemManager object. External code should never call "new" or "delete" on NiMemObjects directly. Rather, they should use "NiNew" and "NiDelete".

Note that the class implementation of new varies depending on whether the preprocessor definition NI_USE_MEMORY_MANAGEMENT is defined. "NiNew" changes implementation depending on the preprocessor symbol to either pass in the file, line, and function information to the NiMemObject "new" or only use the size. Also note that deriving from this class provides zero per-instance overhead in size.

NiApplication : public NiMemObject

Description: NiApplication is a platform-independent wrapper for Gamebryo sample applications. It is designed to wrap platform-dependent functions so that an application can be written for one platform and run on the other platforms that Gamebryo supports. NiApplication handles tasks such as window creation, renderer creation, creation of an input system, and providing access to the command line.

Constructors

NiApplication( const char* pcWindowCaption,
               unsigned int uiWidth = DEFAULT_WIDTH,
               unsigned int uiHeight = DEFAULT_HEIGHT,
               bool bUseFrameSystem = false,
               unsigned int uiMenuID = 0,
               unsigned int uiNumStatusPanes = 0,
               unsigned int uiBitDepth = 0)

This is the default constructor. The window caption, menu pane, and number of status panes are all Win32-specific. The width, height, and bit depth are all passed along to the renderer creation function. The width and height have default values that are appropriate for the target platform being compiled against. See the appropriate NiApplication source file for any particular platform to see what these default values are. bUseFrameSystem indicates whether or not to use the Gamebryo frame rendering system to render the scene or to render the frame directly.

Static Member Functions

    static NiApplication* Create()

This function must be implemented in any derived class. This is the singleton constructor method and must construct and return a pointer to the NiApplication object.

In this example we call Create() but we extend NiSample, not NiApplication, so let’s take a look.
NiSample : public NiApplication

Description: This class provides consistent functionality among all our samples. It allows the user to add a consistent UI system, camera navigation controls, and setup the shader system. This class is intended for Emergent use only.

Constructors

NiSample( const char* pcWindowCaption,
          unsigned int uiWidth = DEFAULT_WIDTH,
          unsigned int uiHeight = DEFAULT_HEIGHT,
          bool bUseFrameSystem = false,
          unsigned int uiMenuID = 0,
          unsigned int uiNumStatusPanes = 0,
          unsigned int uiBitDepth = 0 )

This constructor supplies all of its arguments to NiApplication and initializes the NiSample slots.

Note that in our example some of the default values listed above are being relied upon.

NiApplication : public NiMemObject

Virtual Member Functions: This section describes the set of virtual functions available in NiApplication. You may also wish to see the documentation on commonly overridden methods (see below) to get an overview of the more important of these functions.

- virtual bool Initialize()

Called when the application is being created, but before the first OnIdle loop. This function creates the camera, renderer, input system, scene, and visual trackers. It also updates the scene to time zero. Classes that derive from NiApplication may wish to not override this function and instead override the sub-functions that it calls, such as CreateScene.

- virtual bool CreateScene()
- virtual bool CreateCamera()
- virtual bool CreateFrame()
- virtual bool CreateRenderer()
- virtual bool CreateInputSystem()
- virtual bool CreateVisualTrackers()

This set of functions is called from Initialize.

They set up default versions of whatever their function is designed to handle. Classes that derive from NiApplication will probably wish to override some or all of these functions to set up their application in a custom manner. CreateFrame is only called if bUseFrameSystem in the constructor is true.
Commonly Overridden Methods in NiApplication:

Three NiApplication virtual functions, Initialize, Terminate, and OnIdle, can be overridden by the application to provide specific behavior or to prevent default behavior.

Initialization

virtual bool Initialize()

The return value indicates success (true) or failure (false). Common causes of failure include failure to create a renderer and failure to create a scene because a file needed by the streaming system cannot be found. The default version of Initialize handles tasks such as renderer creation, scene creation, etc. Applications that override Initialize should call NiApplication::Initialize first and check to see if that call is successful before proceeding. More complicated applications might not call the base class Initialize at all. Such applications should consult the source code for NiApplication::Initialize to make sure they are handling all tasks handled by the base-class version.

virtual bool CreateScene()

In simple cases, overloading only CreateScene is sufficient. Initialize calls CreateScene as one of several steps.

Termination

virtual void Terminate()

The default implementation assigns NULL to the smart pointers held by NiApplication to force the final decrement on reference counts so that the objects are deleted. To allow objects to be deleted in the appropriate order, applications that override Terminate should delete their own objects before calling NiApplication::Terminate.

NiApplication and the Frame Rendering System

Classes derived from NiApplication can optionally make use of the Gamebryo frame rendering system. This feature can be turned on or off only in the NiApplication constructor via the bUseFrameSystem parameter. This parameter is stored internally as a const Boolean member and is used to switch between different modes of operation in different functions.

Note that Gamebryo's automatic shadowing system is only enabled in NiApplication if the bUseFrameSystem variable is true. With this parameter set as false, which is the default value, the frame rendering system is disabled. In this situation, the application will need to use the m_spCamera, m_spScene, and m_spRenderer variables directly in order to render a frame.

Any NiMeshScreenElements objects or NiVisualTracker objects that the application is using can be obtained through the GetScreenElements and GetVisualTrackers protected functions.
NiApplication and the Frame Rendering System (... continued)

If the bUseFrameSystem parameter is set to true, applications will need to perform the following steps to use the frame rendering system to perform rendering (abridged, from the documentation):

- The CreateFrame function is called from Initialize.
- The NiApplication::BeginFrame function does not do anything.
- The RenderFrame function does not call CullFrame to perform culling.
- The RenderFrame function does not call RenderScreenItems or RenderVisualTrackers.
- The NiApplication::EndFrame function does not do anything.
- The NiApplication::DisplayFrame function calls Display on m_spFrame

It is also worthwhile to note that NiApplication will perform some convenience functions automatically when using the frame rendering system [snip].

Commonly Overridden Methods in NiApplication (... continued)

Idle Loop

```cpp
// MeasureTime returns false if the frame rate is over the pre-set limit
if (!MeasureTime())
    return;
ResetFrameTimings();

BeginUpdate();
UpdateFrame();
EndUpdate();

if (m_bUseFrameSystem || m_spRenderer != NULL)
{
    BeginFrame();
    RenderFrame();
    EndFrame();
    DisplayFrame();
    UpdateVisualTrackers();
}

UpdateMetrics();
m_iClicks++;
if ((m_fAccumTime >= m_fAppEndTime) && (m_fAppEndTime != NI_INFINITY))
{
    QuitApplication();
}
```

The OnIdle method allows the application to do real-time work. The default implementation keeps track of system time, calls the virtual functions that handle user input, measures the frame rate, clears the backbuffer and z-buffer, and renders the scene. Most applications do not override OnIdle directly but instead override select sub-functions. Most often, the NiApplication version of the function should be called by the derived implementation to maintain default functionality.
The functionality of various methods may be different depending on whether or not the frame rendering system is being used (i.e., whether or not the bUseFrameSystem constructor parameter is true). See the NiApplication and the Frame Rendering System section for more information about what is different when the frame rendering system is enabled.

### Commonly Overridden Methods in NiApplication (... continued)

```cpp
virtual void BeginUpdate()
```

Begins any update timing. Almost never overridden.

```cpp
virtual void UpdateFrame()
```

Updates input and measures frame rate. This is the appropriate place to update scene graph nodes and perform any other time-varying logic an application requires.

```cpp
virtual void EndUpdate()
```

Stores the results of update timing. Almost never overridden.

At about this time it seems reasonable to assume we have a basic idea. Now let’s look at another demo:

The CharacterAnimationDemo application demonstrates many of the advanced features of the Gamebryo animation system, including the NiActorManager animation controller and the NiSkinningLODController character level-of-detail controller. The NiActorManager class allows the application to easily control the many artist generated animation sequences available in Momma. The application simply needs to provide the actor manager with a single "event code" that tells the manager what sequence to use during a given animation. The NiActorManager class handles all the complexities of animation blending and transitioning.

NiActorManager also provides a callback system for applications to manage animation sequence events. In the CharacterAnimationDemo, the walk and run animation sequences were tagged with "text keys" to identify when in those sequences Momma's feet strike the ground. CharacterAnimationDemo implements a very basic footprint mechanism to demonstrate this functionality — a left or right footprint becomes visible when these events occur. When one footprint is made visible, the other disappears. (Note: we recommend something more advanced and artistic in terms of application behavior for games that want to have their characters leave footprints behind them).

NiSkinningLODController gives the application control over the number of bones that are used to deform Momma's skin. BoneLOD allows an application to dynamically adjust how many bones in a skeleton affect its skin. It allows both the polygon count of the mesh and the number of bones to be swapped out at run-time by switching between levels of detail. [...]
// include "CharacterAnimationDemo.h"
#include "LM_Anim.h"
#include <NiUIManager.h>
#include <NiCommonSemantics.h>
#include <NiMeshUtilities.h>
#include <NiLicense.h>
NiEmbedGamebryoLicenseCode;

NiApplication* NiApplication::Create()
{
    return NiNew CharacterAnimationDemo;
}

CharacterAnimationDemo::CharacterAnimationDemo() :
    NiSample("CharacterAnimationDemo", DEFAULT_WIDTH, DEFAULT_HEIGHT, true),
    m_pkBoneLOD(NULL), m_uiLOD(0), m_bIsIdling(true), m_pkMommaCallback(0)
{
    m_kLODNextSlot.Initialize(this, &CharacterAnimationDemo::NextLODLevel);
    m_kNextAnimStateSlot.Initialize(this, &CharacterAnimationDemo::SetNextAnimState);
    m_kNavUpAxis = NiPoint3(0.0f, 1.0f, 0.0f);
    m_fNavDefaultScale = 400.f;

    // Set media path.
    #if defined(_XENON)
        SetMediaPath("D:\DATA\WIN32\");
    #elif defined(WIN32)
        SetMediaPath("/../DATA/WIN32/");
    #elif defined(_PS3)
        SetMediaPath("/../DATA/PS3/");
    #elif defined(_WII)
        SetMediaPath("/Demos/CharacterAnimationDemo/Data/Win32/");
    #endif

    m_kSpeed = NiPoint2::ZERO;
    m_uiMaxOutputLogLines += 2;
}

void CharacterAnimationDemo::Terminate()
{
    NiDelete m_pkMommaCallback;
    m_spAM = 0;
    m_spWireframe = 0;
    NiSample::Terminate();
}

void CharacterAnimationDemo::QuitCharacterAnimationDemo()
{
    QuitApplication();
}

bool CharacterAnimationDemo::CreateScene()
{
    m_spScene = NiNew NiNode;
    NIASSERT(m_spScene);

    // to be continued ...
// Load the KFM
m_spAM = NiActorManager::Create(ConvertMediaFilename("LM.kfm"));
if (!m_spAM)
{
    NiOutputDebugString("Momma KFM Load Failed\n");
    return false;
}

// Manage the root node of Momma's character
NiNode* pkMomma = NiDynamicCast(NiNode, m_spAM->GetNIFRoot());
NIASSERT(pkMomma);
NiMatrix3 kXRot, kYRot, kZRot;
kXRot.MakeXRotation(NI_PI * 0.5f);
kYRot.MakeYRotation(NI_PI * 0.35f);
kZRot.MakeZRotation(NI_PI * -0.1f);
pkMomma->SetRotate(kZRot * kYRot * kXRot);
m_spScene->AttachChild(pkMomma);

// Initialize an array of actor manager event codes for the random idle animations, then choose the base idle as the initial default
InitializeRandomIdle();
m_spAM->SetTargetAnimation(LM_Anim::IDLE01BASE);
m_spAM->Update(0.0f);

// Initialize text-key callbacks from the actor manager
InitializeCallbacks();

// Light Momma with a directional light
NiDirectionalLight* pkDirLight = NiNew NiDirectionalLight;
pkDirLight->SetAmbientColor(NiColor::BLACK);
pkDirLight->SetDiffuseColor(NiColor(0.7f, 0.7f, 0.85f));
pkDirLight->AttachAffectedNode(pkMomma);
m_spScene->AttachChild(pkDirLight);

// Attach a wireframe property
m_spWireframe = NiNew NiWireframeProperty;
m_spScene->AttachProperty(m_spWireframe);

// Locate BoneLOD controller
m_pkBoneLOD = m_spAM->GetSkinningLODController();
NIASSERT(m_pkBoneLOD);
m_pkBoneLOD->SetBoneLOD(m_uiLOD);

// Center camera on scene
m_spCamera->SetTranslate(NiPoint3(-400.0f, 50.0f, 0.0f));

// Attach the footprint quads used during animation callbacks
pkMomma->AttachChild(CreateFootprintQuads());

// Perform initial updates -- required for UpdateSelected.
m_spScene->Update(0.0f);
m_spScene->UpdateNodeBound();
m_spScene->UpdateProperties();
m_spScene->UpdateEffects();
return true;

// to be continued ...

This should give us an idea where the study of Gamebryo should start.
void CharacterAnimationDemo::InitializeRandomIdle()
{
    // Initialize an array of event codes that can be chosen randomly
    m_kRandomIdle.SetSize(10);
    m_kRandomIdle.SetAt(0, LM_Anim::IDLE01BASE);
    m_kRandomIdle.SetAt(1, LM_Anim::IDLE01BASE);
    m_kRandomIdle.SetAt(2, LM_Anim::IDLE01BASE);
    m_kRandomIdle.SetAt(3, LM_Anim::IDLE01BASE);
    m_kRandomIdle.SetAt(4, LM_Anim::IDLE01BASE);
    m_kRandomIdle.SetAt(5, LM_Anim::IDLE01BASE);
    m_kRandomIdle.SetAt(6, LM_Anim::IDLE02LOOKBACK);
    m_kRandomIdle.SetAt(7, LM_Anim::IDLE03SHIVER);
    m_kRandomIdle.SetAt(8, LM_Anim::IDLE04ITCH);
    m_kRandomIdle.SetAt(9, LM_Anim::IDLE05TURNAROUND);
}

void CharacterAnimationDemo::InitializeCallbacks()
{
    // Setup callbacks for footsteps -- the callback object is derived
    // from NiActorManager::CallbackObject
    // [...]
}

Note the definitions. Where are they coming from?

// This file was automatically generated.
// It contains definitions for all the
// animations stored in the associated
// KFM file. Include this file in your
// final application to easily refer to
// animation sequences.

#ifndef LM_ANIM_H
#define LM_ANIM_H

namespace LM_Anim {

enum {
    IDLE01BASE = 0,
    IDLE02LOOKBACK = 1,
    IDLE03SHIVER = 2,
    IDLE04ITCH = 3,
    IDLE05TURNAROUND = 4,
    WALKFORWARD = 5,
    WALKBACK = 6,
    WALKLEFT = 7,
    WALKRIGHT = 8,
    ICEBREAK01 = 9,
    RUN_V2 = 10,
    SKID_TO_STOP = 11,
    INPLACE_LEFT_TO_RIGHT = 12,
    INPLACE_RIGHT_TO_LEFT = 13,
    JUMP01 = 100,
    JUMPLEFT = 101,
    JUMPRIGHT = 102
};
}
#endif // #ifndef LM_ANIM_H

This is LM_Anim.h

Bone LOD

Using Bone LOD to improve your scene performance for characters.

Bone LOD allows you to dynamically adjust how many bones in a skeleton work with the standard LOD system. Instead, you can reduce bone influence for different LODs by changing either the Mesh LOD or the Bone LOD. Bone influence can be set to a lower value to reduce performance and ensure the meshing of the skin.

For animation, when the character occupies more than 1000 tips, hair and war add influence to the animation mesh. The result is that the character can be rendered at one time, which makes the scene rendering more robust for the artist to set up bone LODs. Inside of 3ds Max or Max, the documentation on Bone LOD can be used to

Momma's and the ends of the bone.

First bone of the bone.

Momma's distance.

choose M.

First, assi.

Momma's is depress.

Momma's the.

M is the.

this root a

be used a
#ifndef CHARACTERANIMATIONDEMO_H
#define CHARACTERANIMATIONDEMO_H

#include <NiSample.h>
#include <NiAnimation.h>

class CharacterAnimationDemo : public NiSample {
public:
    CharacterAnimationDemo();
    virtual void UpdateFrame();
    virtual void Terminate();

protected:
    virtual bool CreateScene();
    virtual bool CreateUIElements();
    virtual bool CreateNavigationControllers();

    void QuitCharacterAnimationDemo();
    void InitializeRandomIdle();
    void InitializeCallbacks();
    NiNode* CreateFootprintQuads();
    NiActorManager::EventCode InterpretInput(const NiPoint2& kDir, float fTime);

    void NextLODLevel();
    void SetNextAnimState();

class MommaCallbackObject : public NiMemObject,
                           public NiActorManager::CallbackObject {
public:
    virtual void AnimActivated( NiActorManager*,
                                 NiActorManager::SequenceID, 
                                 float, 
                                 float) { }
    virtual void AnimDeactivated( NiActorManager*,
                                  NiActorManager::SequenceID, 
                                  float, 
                                  float) { }
    virtual void AnimCompleted( NiActorManager*,
                                NiActorManager::SequenceID, 
                                float, 
                                float) { }
    virtual void TextKeyEvent( NiActorManager* pkManager, 
                            NiActorManager::SequenceID eSequenceID, 
                            const NiFixedString& kTextKey, 
                            const NiTextKeyMatch* pkMatchObject, 
                            float fCurrentTime, 
                            float fEventTime);
    virtual void EndOfSequence( NiActorManager*, 
                               NiActorManager::SequenceID, 
                               float, 
                               float) { }
    virtual ~MommaCallbackObject();
    // ... to be continued ...

About two thirds of CharacterAnimationDemo.h is presented here.
// ... continued from previous page ...
NiMeshPtr m_spLeftFootprint;
NiMeshPtr m_spRightFootprint;
}

NiUIMemberSlot0<CharacterAnimationDemo> m_kLODNextSlot;
NiUIMemberSlot0<CharacterAnimationDemo> m_kNextAnimStateSlot;

NiTPrimitiveArray<NiActorManager::EventCode> m_kRandomIdle;
NiWireframePropertyPtr m_spWireframe;
NiActorManagerPtr m_spAM;
NiSkinningLODController* m_pkBoneLOD;
unsigned int m_uiLOD;
bool m_bIsIdling;
NiPoint2 m_kSpeed;
MommaCallbackObject* m_pkMommaCallback;
unsigned int m_uiLODLogEntry;
unsigned int m_uiAnimStateLogEntry;
};
#endif

The rest of the CharacterAnimationDemo.h file.

Dodgeball Technology Highlights

**Rag-doll Physics Integrated with Animation:** The Slim character in Dodgeball demonstrates the complete rag-doll pipeline in Gamebryo. There are two major components to this: Gamebryo's Animation Tool allows the preview of rag-doll characters and the complete specification of transitions between rag-doll and animation. For example, in Dodgeball the transition from animation to physics is done through a cross-fade that transfers the characters momentum to the physics system. The transition back is a simple blend from the resting physics pose to the beginning of a "getting-up" animation.

The model was authored in 3dsMax and rigged with a PhysX rag-doll skeleton. A series of animations were created in Max and split into sequences using Gamebryo Max plug-in. The same process could have been done in Maya. The Gamebryo Animation Tool was then used to preview the physical behavior of the character. Animation Tool also enabled the artist to specify and preview all of the transitions between key-framed animation and physical control, and vice versa. The resulting animation data is loaded into the Dodgeball application and only a few lines of code are required to manage the rag-doll behavior.

Dodgeball is also the only major Gamebryo sample to use animation accumulation. This is particularly useful while physics is in control because it allows the root of the character model to automatically follow the physical behavior. The root is them in the correct position when the character needs to be smoothly transitioned back to animation.

**Shadows:** Dodgeball uses Gamebryo's integrated shadow system for dynamic shadows. Shadows are set up in 3dsMax or Maya, exported through the Gamebryo tool chain and appear in the application with almost no additional effort. Dodgeball also demonstrates how to add shadow casters and receivers at run-time. Shadows are very helpful in tracking where the balls are moving in 3D space.
Some questions at this point:

- How is the kid imported in Dodgeball? Can you add him to TerrainDemo?
- How is SimpleApp put together. Does it offer a clearer perspective on basic Gamebryo?
- In this regard, what are the relevant chapters from the Programmer’s Manual?
- Do people write their own Application Frameworks (NiSample, NiApplication, etc.)?
- What can we change in the various demos and to what extent?
- What does the workbook has to say in terms of learning basic Gamebryo?

**Dodgeball Code Overview:**

In this section we review the important classes that make up the Dodgeball sample.

---

The **DodgeBall Class**: The top-level application class is derived from Gamebryo's NiSample class. It is primarily responsible for three areas of functionality:

**Initialization**: The Dodgeball class initializes the basic application and all of the other classes. It creates the custom UI system and initializes the PhysX SDK object. Initialization is done in this class because it is the central starting point for the application.

**UI and Game State Management**: The Dodgeball class manages the current game and UI state, such as which splash screen is up or whether the game is in play. It also manages the custom UI created for the splash screens and handles input. We put UI functionality in this class because it has top-level control of the application and contains the access to inherited NiSample functions.

**Update**: The basic per-frame update code is contained in the Dodgeball class, although virtually all the work of update is inherited from the NiSample class or done by the PlaygroundManager object. Update is done here because this class is derived from the NiSample class and can inherit functionality.

**UI Manager Classes**: Dodgeball implements a custom UI built on top of NiUserInterface classes. The custom UI is designed to display the splash screens and buttons (the CustomUIGroup and CustomButton classes). It is also responsible for the camera navigation system (the NiNavCustomController class) and hence the interpretation of all user input.

**Playground Manager**: The PlaygroundManager class performs most of the work in dealing with the game environment, and it also manages the character and cannons (to be continued below).
Playground Manager: (continued from previous page)

The PlaygroundManager class performs most of the work in dealing with the game environment, and it also manages the character and cannons. Its functionality can be broken into 3 areas:

- **Playground management:** The PlaygroundManager class is responsible for loading the playground art assets and providing information about them, such as the playable area and the locations of cannon mount points.

- **PhysX Scene Management:** The PlaygroundManager object creates and owns the NiPhysXScene object and is hence the primary contact point for the PhysX scene. In a step that comes post initialization (when all PhysX content has been created) it sets up a contact reporter so that the application knows about collisions. This information is restricted to collisions between the ball and the character using the actor group information.

- **Update:** The update functionality is broken into two pieces in order to facilitate asynchronous physics simulation and to enforce the requirement that no input be given to the physics system while it is simulating. The pre-input update fetches simulation results that were begun in the previous frame and pushes any results to the Gamebryo scene graph. Note the use of the previous frame time as the time for which results are requested. User input and processing is done now, while simulation is known to be stopped. The post-update function updates all the other objects with the current time before re-starting simulation.

So far only Initialization seems to be relevant. Skip over the next capsule, on to Character Management.

Cannon and Ball Management

The cannons and balls are managed by the CannonManager class and the Cannon class. The latter deals with targeting the cannon - setting it's pose to follow the character while it is targeting to fire. It's Update function returns true when the cannon is in a position to fire.

The CannonManager is responsible for loading the art assets for both the cannon and the ball. The cannons are cloned to create one for each mount point. Twenty ball clones are created at initialization so that they are available immediately to avoid frame stutter when a cannon fires. We know the number of balls we need because the application resets after a fixed number.

Update for the CannonManager class updates each cannon in turn. If the cannon indicates that it should fire, and we don't reset, the ShootBall function is called to take a ball from the list of inactive balls, set it's pose to match the cannon's orientation, and set it in motion. Here we make use of the ability to add and remove NiPhysXProp objects to and from an NiPhysXScene object. Creation and deletion of the prop in the PhysX scene is automatically handled. The ball is also added to the scene graph.
Character Management

The character in Dodgeball, Slim, is both an animated character and a physics rag-doll. Most of the work in setting Slim up to use both animation and physics was done in the art tools (see Dodgeball Art Assets), but some run-time set-up is also required. All the code for managing the character is in the TargetManager class.

Slim has a physics sequence assigned. When Slim is loaded (note the ordering of function calls in TargetManager::LoadActor to ensure correct loading) we create a NiPhysXSequenceManager object to manage transitions between physics and animations. And we specify the physical sequence:

```cpp
m_spActorPhysX = NiNew NiPhysXSequenceManager;
m_spActorPhysX->SetManager(pkCtrlMgr, m_spActorProp);
// Set the sequence to be physical
m_spActorPhysX->MakeSequencePhysical(m_spActorManager->GetSequenceData(SLIM_Anim::PHYSX01));
```

With the NiPhysXSequenceManager in place, the actor manager can be used to transition to and from physics without any special handling. However, the exact time when physics is activated depends on the transition settings, so we set up callbacks so that we can track when physics is activated. The information is important for force application: forces should only be applied to a dynamic actor, but the character is kinematic when it is controlled by a key-sequence. The callbacks on the "get-up" animations are used to prevent the cannons from firing when the character is unable to run away.

The character can be in any one of three states:

**Demo Mode:** If there is no user input, Slim randomly chooses a point in the playground and uses greedy search to try to run there. When Slim gets near the goal, he pauses and then chooses a new goal. Demo mode ends when Slim is hit by a ball (in which case he goes physical) or when the player provides some input, in which case Slim becomes player controlled.

**Player Controlled:** In this mode Slim is controlled by the input system. The code determines the reference frame for input (the camera frame) and the direction Slim is facing. Based on the input, a new target animation is selected to move Slim in the desired direction. There is special handling of situations when Slim gets too close to a boundary of the playground and his shocked stop animation is played.

**Physics controlled:** When Slim is under physics control the application must manage the forces applied to Slim and keep track of when Slim should get up again. When applying forces from the ball impact we must buffer the forces. There are three reasons for this: PhysX requires that information obtained from a contact report not be used to modify the scene until after the simulation step has completed; when Slim is first hit he is kinematic (animation controlled) and hence does not respond to the impact force; and we wish to modify the forces to exaggerate the hit. Hence, forces are queued when the contact report function is called (via TargetManager::ProcessHit) and removed from the queue in the update function for physics mode.

The TargetManager::Update function keeps track of which sequence is active and calls the appropriate update procedure.

Let’s take a look at how the character is loaded in Dodgeball:
bool TargetManager::LoadActor()
{
    // Create the Actor Manager. Tell it not to load the NIF file.
m_spActorManager = NiActorManager::Create(
    NiApplication::ConvertMediaFilename("SLIM.kfm"), true, false);
    if (!m_spActorManager)
    {
        NIASSERT(0 && "Couldn't create Actor Manager\n");
        NiMessageBox(
            "Could not create Actor manager. Aborting\n",  
            "Actor Manager Failure.");
        return false;
    }

    // Load the NIF file. We need to do it now, in a separate pass, because we
    // need to pull the PhysX info out of the stream.
    NiStream* pkStream = NiNew NiStream();
    if (!pkStream->Load(NiApplication::ConvertMediaFilename("SLIM.nif")))
    {
        assert(0 && "Couldn't load SLIM.nif file\n");
        NiMessageBox("Could not load SLIM.nif. Aborting\n",  
            "Missing nif file.");
        NiDelete pkStream;
        return false;
    }

    // Look for a camera and the PhysX content.
    for (unsigned int ui = 0; ui < pkStream->GetObjectCount(); ui++)
    {
        if (NiIsKindOf(NiPhysXProp, pkStream->GetObjectAt(ui)))
        {
            // We have found the PhysX content in the NIF.
            m_spActorProp = (NiPhysXProp*)pkStream->GetObjectAt(ui);
        }
    }
    NIASSERT(m_spActorProp != 0);

    // Set up the actor manager with the loaded file.
    if (!m_spActorManager->ReloadNIFFile(pkStream, false))
    {
        NIASSERT(0 && "Couldn't create AM from NIF\n");
        NiMessageBox("Could not set-up Actor Manager. Aborting\n",  
            "Missing nif file.");
        NiDelete pkStream;
        return false;
    }

    // Load all the sequences.
    if (!m_spActorManager->LoadAllSequenceData(pkStream))
    {
        assert(0 && "Couldn't load sequences\n");
        NiMessageBox("Could not load sequences. Aborting\n",  
            "Broken/Missing KF file.");
        NiDelete pkStream;
        return false;
    }

    NiDelete pkStream;

    // Get the controller manager for the character
    NiControllerManager* pkCtrlMgr = m_spActorManager->GetControllerManager();
    NIASSERT(pkCtrlMgr->GetCumulativeAnimations());

    // Set the initial location
    m_spActorManager->GetNIFRoot()->SetTranslate(ms_kStartLocation);

    // to be continued on the next page
// continued from previous page

// Update the actor manager.
m_spActorManager->SetTargetAnimation(SLIM_Anim::IDLE);
m_spActorManager->Update(0.0f);

// Attach the root node of our character to the scene, and get the
// walkable area.
PlaygroundManager* pkPlayground = PlaygroundManager::Instance();

m_spActorRoot = NiDynamicCast(NiNode, m_spActorManager->GetActorRoot());
NIASSERT(m_spActorRoot);

m_spActorLocator = m_spActorRoot->GetObjectByName("Bip01 NonAccum");
NIASSERT(m_spActorLocator);

pkPlayground->GetScene()->AttachChild(m_spActorManager->GetNIFRoot());

// Attach slim as an affected node to the dynamic shadow casting light.
NiNode* pkDynLightParent = NiDynamicCast(NiNode, pkPlayground->GetScene()->GetObjectByName("direct01_dynamic"));
NIASSERT(pkDynLightParent);

NiLight* pkDynLight = NiDynamicCast(NiLight, pkDynLightParent->GetAt(0));
NIASSERT(pkDynLight);

NiNode* pkRoot = NiDynamicCast(NiNode, m_spActorManager->GetActorRoot());
pkDynLight->AttachAffectedNode(pkRoot->GetParent());
pkPlayground->GetWalkableArea(m_kWalkArea.m_kMin.x, m_kWalkArea.m_kMin.y,

m_kWalkArea.m_kMax.x, m_kWalkArea.m_kMax.y, m_kWalkArea.m_fZ);

// Instantiate physics content
pkPlayground->GetPhysXScene()->AddProp(m_spActorProp);

// Set up physical control of one sequence.
m_spActorPhysX = NiNew NiPhysXSequenceManager;
m_spActorPhysX->SetManager(pkCtrlMgr, m_spActorProp);
m_spActorPhysX->MakeSequencePhysical(m_spActorManager->GetSequenceData(SLIM_Anim::PHYSX01));

// Set a callback for activation of the PhysX sequence. We need this to
// know when we can start applying forces and start testing for sleeping.
// We also use it to keep track of when it is fair to hit Slim.
NISequenceData* pkSeqData;
pkSeqData = m_spActorManager->GetSequenceData(SLIM_Anim::PHYSX01);
pkSeqData->AddDefaultActivationCallback(this, pkCtrlMgr);
pkSeqData->AddDefaultActivationCallback(this, pkCtrlMgr);
pkSeqData->AddDefaultActivationCallback(this, pkCtrlMgr);
pkSeqData->AddDefaultActivationCallback(this, pkCtrlMgr);

// Find the actors we will check for sleeping
AssignSleepActors();

return true;
}

Loading Slim in Dodgeball. Not just a few lines of code.

Overview of the SimpleApp Demo

This demonstration shows how to create a simple Gamebryo-based application and load and display a NIF file without using the NiApplication layer. This is an important concept, since the NiApplication library exists mainly to allow simple cross-platform demos and rapid prototyping of NI-based applications. Developers should feel free to create their own application framework if NiApplication
Reconsidering SimpleApp (Part I).

does not suit them. This demo shows the absolute minimum that is required to create a Gamebryo application that renders a scene and handles input.

```cpp
#include <NiAnimation.h>
#include <NiDebug.h>
#include <NiMain.h>
#include <NiMeshLib.h>
#include <NiMeshCullingProcess.h>
#include <windows.h>
#include <NiLicense.h>
NiEmbedGamebryoLicenseCode;
#include <NiDX9Renderer.h>
#pragma comment(lib, "NiDX9Renderer.lib")
#pragma comment(lib, "dxguid.lib")
#include <NiD3D10Renderer.h>
#pragma comment(lib, "NiD3D10Renderer.lib")

static NiWireframePropertyPtr gs_spWireframe;

bool SimpleApp(HWND hWnd, bool bD3D10);
bool CreateRenderer(HWND hWnd, bool bD3D10, NiRendererPtr& spRenderer);

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wparam, LPARAM lparam)
{
    switch (msg)
    {
    case WM_CHAR:
        {
            switch ((unsigned char)wparam)
            {
            case 'w': // toggle wireframe mode
                case 'W':
                    gs_spWireframe->SetWireframe(!gs_spWireframe->GetWireframe());
                    break;
            case 'q': // terminate the application (ALT+F4 works also)
                case 'Q':
                case VK_ESCAPE:
                    PostMessage(hWnd, WM_DESTROY, 0, 0);
                    break;
            }
        break;
    case WM_SYSKEYUP:
        {
            // Trap the ALT key, allowing the application to continue
            // operation, without bringing up the main menu
            switch ((unsigned char)wparam)
            {
            case VK_MENU:
                case VK_F10:
                    return 1;
                    break;
            }
        break;
    case WM_DESTROY:
        {
            PostQuitMessage(0);
            break;
        }
    }
    return DefWindowProc(hWnd, msg, wparam, lparam);
    }
```
int WINAPI WinMain(HINSTANCE hI, HINSTANCE, LPSTR cmdline, int iWinMode)
{
    //***** basic window setup *****
    static char lpszName[] = "Simple Application";

    WNDCLASS wc;
    wc.style         = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = WndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hI;
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName  = 0;
    wc.lpszClassName = lpszName;
    RegisterClass(&wc);

    // create a window with client area 1024x768
    RECT rect = {0, 0, 1024, 768};
    DWORD dwStyle = WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX;
    AdjustWindowRect(&rect, dwStyle, false);
    HWND hWnd = CreateWindow(lpszName, lpszName, dwStyle, 0, 0,
                              rect.right - rect.left, rect.bottom - rect.top, NULL, NULL, hI, NULL);
    ShowWindow(hWnd, iWinMode);
    UpdateWindow(hWnd);

    // Check for command-line parameters requesting a D3D10 renderer
    bool bD3D10 = false;
    if (strstr(cmdline, "d3d10"))
        bD3D10 = true;

    //***** Gamebryo initialization *****
    NiInit();
    int iSuccess = 0;
    if (!SimpleApp(hWnd, bD3D10))
        iSuccess = -1;
    NiShutdown();

    return 0;
}

SimpleApp reconsidered (Part II)

SimpleApp Demo Code Description

Implementation varies from platform to platform, so it is not useful to describe the code in depth here. The source code can be referred to for specific details, but the general methodology of the sample is easy to understand.

First, the platform-dependent initializations occur. This can include initializing the run-time environment, preparing the message loop, and creating a suitable window. Command-line arguments or initialization files should be consulted in this part of the code to determine how the initial state of the program should be created.

Then, the Gamebryo initializations occur. First, an NiCamera is created, and then an NiRenderer object to act as the main renderer. Once this is complete, the initial scene can be built, whether programmatically, through NIF files, or using both methods. The parts then need to be connected so the
NiCamera knows where the NiRenderer object and scene graph are. At this point, it is usually a good idea to do an initial Update call on the scene. After the Gamebryo components and scene are initialized, the message loop of the program can begin. This section is usually of the form:

```cpp
while (bProgramNotDone)
{
    HandleUserInteraction();
    UpdateSceneBasedOnInteraction();
    spRenderer->BeginFrame();
    spRenderer->BeginUsingDefaultRenderTargetGroup(NiRenderer::CLEAR_ALL);
    NiDrawScene(spCamera, spScene, kCuller);
    spRenderer->EndUsingRenderTargetGroup();
    spRenderer->EndFrame();
    spRenderer->DisplayFrame();
}
```

Once the application has completed, it must begin cleaning up after itself. Gamebryo objects should be deleted, including the scene graph, NiRenderer object, and NiCamera. Then the program can free any system resources it is using, and exit the program entirely.

Note that a call to NiInit() must preceed any calls to Gamebryo interfaces, and that NiShutdown() must be called prior to application exit, but after the last call to any Gamebryo function. All Gamebryo objects must be deleted prior to the call to NiShutdown().

```cpp
bool SimpleApp(HWND hWnd, bool bD3D10)
{
    NiImageConverter::SetImageConverter(NiNew NiDevImageConverter);
    NiTexture::SetMipmapByDefault(true);
    NiSourceTexture::SetUseMipmapping(true);

    // Setup the material shader generation folder prior to creating the // renderer.
    NiMaterial::SetDefaultWorkingDirectory("../../../Data/Shaders/Generated");

    // create a renderer
    NiRendererPtr spRenderer;
    bool bSuccess = CreateRenderer(hWnd, bD3D10, spRenderer);
    if (!bSuccess || !spRenderer)
    {
        NiOutputDebugString("Direct3D renderer creation error: ");
        NiOutputDebugString(NiRenderer::GetLastErrorString());
        NiOutputDebugString("\n");
        if (bD3D10)
        {
            NiMessageBox("Unable to create a D3D10 Renderer.\n" "This symptom may indicate you are not running on a " "D3D10-capable graphics card.\n", "D3D10 Renderer Creation Failed");
        }
        else
        {
            NiMessageBox("Unable to create a DX9 Renderer.\n", "DX9 Renderer Creation Failed");
        }
        return false;
    }

    spRenderer->SetBackgroundColor(NiColor(0.5f, 0.5f, 0.5f));
    // continues on next page
```
// create a scene
NiStream kStream;
bool bLoaded = kStream.Load("../../Data/TheClockAndTheOcularis.nif");
NI_UNUSED_ARG(bLoaded);
NIASSERT(bLoaded);
NiNodePtr spScene = NiDynamicCast(NiNode, kStream.GetObjectAt(0));
NIASSERT(spScene != NULL);
NiCameraPtr spCamera = NiDynamicCast(NiCamera, kStream.GetObjectAt(1));
NIASSERT(spCamera != NULL);
Ni2DBufferPtr pkBackBuffer = spRenderer->GetDefaultBackBuffer();
if(pkBackBuffer)
{
NiFrustum kFrustum = spCamera->GetViewFrustum();
float fAspectRatio = (float)pkBackBuffer->GetWidth()/(float)pkBackBuffer->GetHeight();
kFrustum.m_fLeft = fAspectRatio * kFrustum.m_fBottom;
kFrustum.m_fRight = fAspectRatio * kFrustum.m_fTop;
spCamera->SetViewFrustum(kFrustum);
}
kStream.RemoveAllObjects();
gs_spWireframe = NiNew NiWireframeProperty;
spScene->AttachProperty(gs_spWireframe);
//RecursiveMunge(spScene);
// connect the parts and do initial update
spScene->Update(0.0f);
spScene->UpdateProperties();
spScene->UpdateEffects();
spCamera->Update(0.0f);
// create a culling process
NiVisibleArray kVisible(32, 32);
NiMeshCullingProcess kCuller(&kVisible, NULL);
//***** application idle loop *****
for( ; ; )
{
// message pump
MSG msg;
if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
{
if (msg.message == WM_QUIT)
break;
if (!TranslateAccelerator(hWnd, NULL, &msg))
{
TranslateMessage(&msg);
DispatchMessage(&msg);
}
}
else {
// Update frame
spScene->Update(NiGetCurrentTimeInSec());

// Render frame
spRenderer->BeginFrame();
spRenderer->BeginUsingDefaultRenderTargetGroup( NiRenderer::CLEAR_ALL);
NiDrawScene(spCamera, spScene, kCuller);
spRenderer->EndUsingRenderTargetGroup();
spRenderer->EndFrame();
spRenderer->DisplayFrame();
}
// Release the global smart pointer reference prior to NiShutdown.
gs_spWireframe = NULL;
// All other smart pointers declared in this scope
// (spRenderer, spScene, spCamera) will be deleted
return true;
} // ends on next page
bool CreateRenderer(HWND hWnd, bool bD3D10, NiRendererPtr& spRenderer)
{
    if (bD3D10)
    {
        NiD3D10Renderer::CreationParameters kParameters(hWnd);
        NiD3D10RendererPtr spD3D10Renderer;
        bool bResult = NiD3D10Renderer::Create(kParameters, spD3D10Renderer);
        if (bResult)
            spRenderer = spD3D10Renderer;
        return bResult;
    }
    else
    {
        spRenderer = NiDX9Renderer::Create(0, 0, hWnd, NULL);
        return (spRenderer != NULL);
    }
}

Reconsidering SimpleApp (Parts II and IV). A lot of platform specific code.

Regarding our previous questions 3-5 the answers are: yes, sometimes (perhaps always) developers write their own application framework. Reading the code above with the programmer’s manual in front of us is likely to take us deep into where Gamebryo learning starts. Same thing for the demos and likely one can modify them effectively. This may take a couple of months more. Finally, the workbook:

This very basic introduction to the concept of state machines should convince you that a typical game - really, any game that gives the player any options - can be built around a high level finite state machine. With a little bit of planning, it isn’t difficult to build these types of states into even the earliest stages of a game, from the beginning.

Implementing a Simple Game Application using Gamebryo and a Simple Finite State Machine

The remainder of this chapter will assemble a simple game application framework, implementing a subset of the states described above.

Implementing Gameplay Elements Using Gamebryo

This very basic introduction to the concept of state machines should convince you that a typical game - really, any game that gives the player any options - can be built around a high level finite state machine. With a little bit of planning, it isn’t difficult to build these types of states into even the earliest stages of a game, from the beginning.

To be continued and finished in Lecture Notes Fourteen.