Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS
Lecture #15

Game Engine Standards
How they did it?

• Engines
  – Ogre 3D
  – XNA platform
  – Unreal, Quake and CryEngine

• Components
  – Global architecture
  – Scene management
  – Input management
  – Resource management
Ogre 3D

• Object-oriented Graphics Rendering Engine
  – a graphics engine, not a game engine...
  – easy plugin of features (python script, ode physics engine, etc.)
  – http://www.ogre3d.org

Torchlight
Runic Games

Alien Dominion
Black Fire Games
Ogre features

• Programming
  – OO interface in C++
  – Extensible framework
  – Stable and high performance engine

• Platform
  – Multi-platform
  – Direct3D and OpenGL support

• Content
  – Scene manager
  – Resource manager
    • Material, meshes
  – Animation
  – Renderer
    • Special effects, shader
  – Plugins
Ogre architecture overview
Ogre

• The ‘Root’ object is the entry point
  – must be the first created object
  – must be the last deleted object
  – enables the configuration of the system
  – has a continuous rendering loop
• **SceneManager**
  – Contains everything that appears on the screen
  – Different managers for terrain (heightmap), exterior and interior scenes

• **Entity**
  – Type of object you can render in the scene
  – Anything that is being represented by a mesh (player, ground, …)
  – Not an entity: Lights, Billboards, Particles, Cameras, *etc.*
SceneNode

- Scene nodes keep track of location and orientation for all of the objects attached to it
- An Entity is only rendered on the screen if it is attached to a SceneNode object
- A scene node's position is always relative to its parent node
- A scene manager contains one root node to which all other scene nodes are attached

The final structure is the scene graph
// Create Root
Ogre::Root* mRoot = new Ogre::Root();

// Parses resources.cfg
setupResources();

// Shows the Ogre config GUI which configures the render system
// and constructs a render window
configure();

// The scene manager decides what to render
chooseSceneManager();

// We need a camera to render from
createCamera();

// and at least one viewport to render to
createViewports();
// Create any resource listeners (for loading screens)
createResourceListener();

// Now we can load the resources: all systems are on-line
loadResources();

// Now that the system is up and running: create a scene to render
createScene();

// Create any frame listeners (input manager: keyboard, mouse...)
createFrameListener();

// Kick off Ogre loop
mRoot->startRendering();

// Clean up
destroyScene();

// Delete root
delete mRoot;
Ogre application design

• Ogre is using a Frame Listener in the game loop to receive notification from the system
  – The game inherits from FrameListener

```cpp
class Game : public Ogre::FrameListener {
  // ...
}
```

– And register itself to listen to the notifications

```cpp
mRoot->addFrameListener(this);
```
Ogre game loop

• The Root::startRendering method starts the rendering cycle
  – It begins the automatic rendering of the scene
  – It will not return until the rendering cycle is halted

• During rendering, any FrameListener registered will be called back for each frame that is to be rendered
  – These classes can tell Ogre to halt the rendering if required, which will cause this method to return
Ogre game loop

- Ogre notifies the listeners at different time of the game loop

```cpp
// called just before a frame is rendered
virtual bool frameStarted(const FrameEvent& evt);

// called after all render targets have had their rendering commands issued, but before the render windows have been asked to swap buffers
virtual bool frameRenderingQueued(const FrameEvent& evt);

// called just after a frame has been rendered (buffers swapped)
virtual bool frameEnded(const FrameEvent& evt);
```

- if return value is false, program exits
- `evt.timeSinceLastFrame` contains how long is has been since the last call
**Ogre game loop**

```cpp
void Root::startRendering(void) {
    // ... Initialization ...
    mQueuedEnd = false;
    while( !mQueuedEnd ) {
        // Pump messages in all registered RenderWindow windows
        WindowEventUtilities::messagePump();
        if (!renderOneFrame()) break;
    }
}

bool Root::renderOneFrame(void) {
    if(!_fireFrameStarted()) return false;

    if (!_updateAllRenderTargets()) // includes _fireFrameRenderingQueued()
        return false;

    return _fireFrameEnded();
}
```
Input management in Ogre

- Ogre allows for both HID managements
  - polling (called unbuffered)
  - interruption (called buffered)
HID unbuffered in Ogre

- Update the user inputs in `frameRenderingQueued`

```cpp
bool Game::frameRenderingQueued(const Ogre::FrameEvent& evt) {
    // ...
    mMMouse->capture();   // to read mouse state
    mKeyboard->capture(); // to read keyboard state
    return processUnbufferedInput(evt);
}
```

- where `mMouse` and `mKeyboard` are defined using the OIS library included in Ogre

- `processUnbufferedInput` pass the event to user functions according to the updated keyboard and mouse states

```cpp
bool processUnbufferedInput(const Ogre::FrameEvent& evt);
```
HID unbuffered in Ogre

- Example

```cpp
static bool Game::prevLeftMouseDown = false;  // if a mouse button was pressed
static Ogre::Real Game::mMove = 0.2;         // the movement increment

bool Game::processUnbufferedInput(const Ogre::FrameEvent& evt) {
    // check if current left mouse button is pressed
    bool leftMouseDown = mMouse->getMouseState().buttonDown(OIS::MB_Left);
    if (leftMouseDown && !prevLeftMouseDown) {  // if not pressed before
        // do something when left mouse button pressed, e.g. shoot();
        prevLeftMouseDown = true;
    }
    // check if user is pressing up arrow
    if (mKeyboard->isKeyDown(OIS::KC_NUMPAD8) ||
        mKeyboard->isKeyDown(OIS::KC_UP)) {
        moveForward(mMove);
    }
    // update scene ...
}
```
HID buffered in Ogre

- Mouse and keyboard events are handled immediately instead of once per game loop
- Ogre uses an event mechanism (DP), the game class needs to inherit from

OIS::KeyListener for keyboard

```cpp
#include <OISEvents.h>
#include <OISInputManager.h>
#include <OISKeyboard.h>

class Game : public OIS::KeyListener
```

OIS::MouseListener for mouse

```cpp
#include <OISEvents.h>
#include <OISInputManager.h>
#include <OISMouse.h>

class Game : public OIS::MouseListener
```
HID buffered in Ogre

• The following member functions are inherited

```cpp
// OIS::KeyListener
virtual bool keyPressed( const OIS::KeyEvent& evt );
virtual bool keyReleased( const OIS::KeyEvent& evt );

// OIS::MouseListener
virtual bool mouseMoved( const OIS::MouseEvent& evt );
virtual bool mousePressed( const OIS::MouseEvent& evt, OIS::MouseButtonID id );
virtual bool mouseReleased( const OIS::MouseEvent& evt, OIS::MouseButtonID id );
```

– when a key is pressed, the keyPressed function is fired
– when the mouse moves, the mouseMoved function is fired
– etc.
HID buffered in Ogre

- The listening registrations are done during the application setup, typically in a createFrameListener function

```cpp
void Game::createFrameListener () {
    // ...
    mMouse->setEventCallBack(this);
    mKeyboard->setEventCallBack(this);
    // ...
}
```
Resource management in Ogre

- **Unknown**: Ogre is not aware of the resource. Its filename is stored but Ogre has no idea what to do with it
- **Declared**: Flagged for creation. Ogre knows what type of resource it is, and what to do with it when the time comes to create it
- **Created**: Ogre has created an empty instance of the resource, and added it to the relevant manager
- **Loaded**: Created instance has been fully loaded, stage at which the resource's file is accessed
Resource management in Ogre

1. Ogre's native ResourceManagers are created in Root::Root

2. Specify resource locations by calling

   ```cpp
   ResourceGroupManager::addResourceLocation("name", "locType")
   ```

3. Manually declare resources
   - **Declared** state for declared resources
   - **Unknown** otherwise
4. Script parsing to automatically declare resources
   - Set these resources as **Declared**
   - Creates the declared resources, now **Created**

5. Resources are loaded when
   - an entity ask for a unloaded resource
   - explicit call to load a resource
   - explicit call to load the declared resources
   ➢ loaded resources put in **Loaded** state
Resource management in Ogre

- `ResourceManager::unload` reverts a resource from **Loaded** to **Created**
- `ResourceManager::remove` removes a resource
  - back to **Unknown** state
- You can get a pointer to the resource with `ResourceManager::getByName` and unload or remove it manually
- Any existing resources are removed when the resource manager is destroyed
Resource management in Ogre

• Reloading resources is a very useful feature
  – resource is unloaded, and then loaded again
  – moves from *Loaded* to *Created* and then back to *Loaded* again

• ResourceManager::reloadAll reloads all resources of one type

• Resources can be individually reloaded with Resource::reload
Resource management in Ogre

• To extend the resource types

```cpp
class MyResource : public Ogre::Resource {
  protected:
    void loadImpl(); // load resource (e.g. from file)
    void unloadImpl(); // unload it
    size_t calculateSize() const; // get its size

  // ...
}

class MyResourceManager : public Ogre::ResourceManager {
  protected:
    Ogre::Resource * createImpl(const Ogre::String &name,
                                Ogre::ResourceHandle handle, const Ogre::String &group, bool isManual,
                                Ogre::ManualResourceLoader *loader, const Ogre::NameValuePairList *createParams); // creates the MyResource instance

  public:
    virtual MyResource * load(const Ogre::String &name, const Ogre::String &group); // load the resource (and create it if needed)
    // ...
}
```
Resource management in Ogre

• To extend the resource types

```cpp
MyResourceManager * mRM = new MyResourceManager();

ResourceGroupManager::getSingleton().declareResource("resourceName", "MyResource");

MyResource* _resource = mRM->load("resourceName", ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);

_resource->aFunction(); // you can now use the resource

// ...
_resource->reload();
// ...

mRM->unload("resourceName");

mRM->remove("resourceName");
// ...
```
Ogre

- Ogre uses many different design patterns
  - Factory in
    - MoveableObjectFactory, ParticleEmitterFactory, ...
  - Iterator in
    - ParticleIterator, ...
  - Singleton in
    - Root, OverlayManager, MaterialManager, ...
  - Listener in
    - FrameListener, ResourceGroupListerner, ...
- Other commonly appearing structures
  - Events, Buffers, Plugins, Serializers
Microsoft XNA Platform

- C# game engine for PC and Xbox 360
  - Easy programming of DirectX based games
- Documentation on MSDN Library
  - Development Tools and Languages
  - XNA Game Studio
- Two sets of libraries
  - XNA Framework
  - Content Pipeline
XNA Framework architecture

- Library of classes, interfaces and value types
  - Framework
    - commonly used game classes, *e.g.* timer and game loop
  - Framework.Audio
    - audio management
  - Framework.Graphics
    - 2D/3D graphics
  - Framework.Input
    - keyboard, mouse and Xbox 360 controller
  - Framework.Net
    - networking
  - Framework.Storage
    - file manipulation
  - ...
Microsoft.Xna.Framework
using Microsoft.Xna.Framework;

class BasicXNAGame : Game { // Inherits from XNA Game functionalities
    GraphicsDeviceManager graphics; // The graphics manager
    SpriteBatch spriteBatch; // The sprite batch

    static void Main() {
        BasicXNAGame game = new BasicXNAGame(); // Main program creates a new game ...
        game.Run(); // ... and runs it
    }

    public BasicXNAGame() {
        Content.RootDirectory = "Content"; // Setup of content directory
        graphics = new GraphicsDeviceManager(this); // Create the graphics manager
    }

    protected override void LoadContent() { // Create the sprite batch
        spriteBatch = new SpriteBatch(GraphicsDevice);
    }

    protected override void Update(GameTime gameTime) { // update code }

    protected override void Draw(GameTime gameTime) { // draw code }
}
XNA game loop

• The game loop is started by the function run of the class Game

```csharp
public class MyGame : Microsoft.Xna.Framework.Game { // ... }

static class Program {
    static void Main(string[] args) {
        MyGame game = new MyGame();
        game.Run();
    }
}
```

• The run method calls the virtual functions to initialize the game, to update and draw the game, and to process events
XNA game loop

- The game loop is made of calls to the update and draw functions of the game
  - gameTime is the time elapsed since the last game loop call

```csharp
protected override void Update(GameTime gameTime) { // ... }
protected override void Draw(GameTime gameTime) { // ... }
```
Scene management XNA

- The scene management (e.g. scene graph) is up to the user
- The graphics library contains low-level API methods that take advantage of hardware acceleration capabilities to display 2D/3D objects
  - Basically an interface for Direct3D
  - With classes such as Texture2D, ModelMesh and Effect

- Models and Sprites
Input management XNA

• On PC, XNA can manage GamePad, Keyboard, Mouse and Microphone
• XNA provides only polling functions

```csharp
KeyboardState ks = Keyboard.GetState();
if (ks.IsKeyDown(Keys.Space)) { // ... }

MouseState ms = Mouse.GetState();
if (ms.LeftButton == ButtonState.Pressed) { // ... }
int curMousePos.X = ms.X;
int curMousePos.Y = ms.Y;
```
Input management XNA

• You can simulate events by manually checking changes in the state

```csharp
KeyboardState _oldState; // data member

// ...

KeyboardState newState = Keyboard.GetState();
if (newState.IsKeyDown(Keys.Space)) {
    if (!_oldState.IsKeyDown(Keys.Space)) { // Key just pressed }
} else if (_oldState.IsKeyDown(Keys.Space)) { // Key just released }
_oldState = newState; // Update state
```
Resource management XNA

• Game assets are managed by the XNA Framework Content Pipeline
• It transfers the run-time native loading process to compile time (implemented in Visual Studio)
  – Each asset is imported from its original file format and processed into a managed code object
  – Those objects are then serialized to a file that is included in the game’s executable
  – At run time, the game reads the serialized data from the file directly into a managed code object
Resource management XNA

• Default asset importers in XNA
  – Autodesk model: .fbx
  – DirectX effect: .fx
  – Sprite fonts: .spritefonts
  – Texture: .bmp, .jpg, .png, .tga, ...
  – DirectX file: .x
  – Microsoft Audio file: .xap
  – XML file: .xml

• Automatically detected (dedicated project) and added to resource file
Resource management XNA

- To load a resource

```csharp
SpriteBatch spriteBatch;
Texture2D myTexture; // This is a texture to render

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    myTexture = Content.Load<Texture2D>("mytexture");
}

protected override void UnloadContent()
{
    // ...
}
```
Resource management XNA

- Custom Content Pipelines can be added to support additional art assets and formats.
- Or to derive special-purpose content from another piece of content at the time the game is built.
- The asset is added in XNA project and its properties specify the appropriate importer.
  - At build time the assigned importer is invoked.
  - The asset is built into the game in a form that can be loaded at run time.
Resource management XNA

- To manage new asset files
  - A custom importer is required that outputs a CustomContent object
  - A custom content processor is also needed
  - The ContentManager.Load method must be extended to support the custom data object
Unreal Engine

• Unreal Engine 3 technology is available through UDK: the Unreal Development Kit
  – Main page: http://udk.com/
  – Documentation: http://udn.epicgames.com/Three/

• Features
  – Own editing environment (UnrealEd)
  – Highly dependent on scripts (UnrealScript)
  – Animation manager (AnimTrees)
  – Interface with PhysX engine (Unreal PhAT)
  – Networking, audio, particle, shader, AI managers
  – and more
Unreal Game

• UnrealScript is used to create custom classes to form the gameplay for the game
  – Located in a dedicated folder and pointed by a configuration file

• Content is stored within packages stored in a Content directory of the Unreal installation
  – including sub-folders for characters, maps, environments, sounds, etc.
Unreal Game

• The scripts are compiled into packages usable by the engine

• Default packages are
  – Core, Engine, GFxUI, GameFramework, UnrealEd, GFxUIEditor, IpDrv, OnlineSubsystemPC, OnlineSubsystemSteamworks, UDKBase, and UTEditor
  – Plus your own MyGame package
UDK Gameplay

• Player's viewpoint is handled in the GetPlayerViewPoint function of the PlayerController class

class MyGamePlayerCamera extends Camera;
function UpdateViewTarget(out TViewTarget OutVT, float DeltaTime) { // ... }

• Input from the player are handled and translated into controlling the game
  – the class responsible for determining how the player controls the game is PlayerController

class MyGamePlayerController extends GamePlayerController;
defaultproperties { CameraClass=class'MyGame.MyGamePlayerCamera' }
UDK Gameplay

- The visual representation of the player and the logic for determining how it interacts with the physical world is encapsulated in the Pawn class

```cpp
class MyGamePawn extends Pawn;
defaultproperties { // ... }
```

- The HUD class is responsible for displaying information about the game to the player

```cpp
class MyGameHUD extends MobileHUD;
defaultproperties { // ... }
```
UDK Gameplay

• The gametype determines the rules of the game and the conditions under which the game progresses and ends
• The gametype is also responsible for telling the engine which classes to use for PlayerControllers, Pawns, the HUD, etc.

```cpp
class MyGame extends FrameworkGame;

defaultproperties
{
    PlayerControllerClass=class'MyGame.MyGamePlayerController'
    DefaultPawnClass=class'MyGame.MyGamePawn'
    HUDType=class'MyGame.MyGameHUD'
    bDelayedStart=false
}
```
Unreal Game

• The UnrealFrontend application finally provides the ability to build scripts, either as a single operation or as part of a pipeline for building and packaging the game for testing or distribution.
Quake engine: Id Tech

- Current version is Id Tech 5
  - Used in Rage and Doom 4
- Id Tech 4 SDK download and documentation
- Source code released in November 2011
  - Used in Doom 3, Quake 4, Wolfenstein, Brink
An Id Tech 4 game

• Q4Radiant is the editor used to create the maps
  – To create a game you start by modeling the virtual world (objects, lights, shadows *etc.*)

• Q4Script system is then used to implement the game logic
  – the scripts will be called from the game with triggers activated by conditions defined in the editor
An Id Tech 4 game

• Script to spawn a monster at a location defined in the editor (targetMonster)

```c
void spawnMonster() {
    // create a variable to hold the entity handle
    entity newMonster;

    // spawn the monster and store his handle in the variable
    newMonster = sys.spawn("monster_strogg_marine");

    // move it to where that new target lives in the edited map
    newMonster.setWorldOrigin( $targetMonster.getWorldOrigin() );
}
```
And more...

• Another very good free SDK: CryEngine 3
  – SDK download: [http://www.crydev.net](http://www.crydev.net)
  – Source released in August 2011
  – Used in Crysis 2, also level design oriented

• Architecture
  – Engine
    • Config, Fonts, Shaders
  – Game
    • Animations, Entities, Levels, Music, Scripts, etc.
    • Libs
      – Dialogs, Particles, Sky, SmartObjects, UI, etc.
  • Scripts
    – AI, Entities, GameRules, Network, Utils, etc.
End of lecture #15

Next lecture

Final lecture