CMSC 498M: Chapter 4
Game Engines and Ogre 3D

Reading: The material for this lecture is adapted from
- Online Ogre tutorials:

Overview:
- Game engines
- The Ogre 3D game engine

Game Engine

Game Engine:
- A core software component ("middleware") of a computer video
game or other interactive application with real-time graphics.
- Provides the underlying technologies.
- Simplifies development.
- Facilitates implementation on multiple platforms such as game
  consoles and Microsoft Windows, Linux, Mac OS X.

Core Functionality:
- Rendering engine: 2D or 3D graphics. Support for higher-level
  functionality, such as scene graphs and animation.
- Physics engine: E.g., collision detection and collision response.
- Others: sound, scripting, networking, streaming, memory management,
  threading.
## Common Game Engines [Source: DevMaster.net]

<table>
<thead>
<tr>
<th>Commercial Engines</th>
<th>Open-Source Engines</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Torque Game Engine (GarageGames)</td>
<td>- Ogre</td>
</tr>
<tr>
<td>- TV3D SDK 6 (Truevision3D)</td>
<td>- Irrlicht</td>
</tr>
<tr>
<td>- 3DGameStudio (Contec Datasystems)</td>
<td>- Crystal Space</td>
</tr>
<tr>
<td>- C4 Engine (Terathon Software)</td>
<td>- jME</td>
</tr>
<tr>
<td>- Unity (Over the Edge)</td>
<td>- Panda3D</td>
</tr>
<tr>
<td>- 3Impact (3Impact)</td>
<td>- Reality Factory</td>
</tr>
<tr>
<td>- DX Studio (Worldweaver)</td>
<td>- The Nebula Device 2</td>
</tr>
<tr>
<td>- Beyond Virtual (Gekido Design Group)</td>
<td>- RealmForge</td>
</tr>
<tr>
<td>- LawMaker Game Engine (Darkroom Studios)</td>
<td>- Blender Game Engine</td>
</tr>
<tr>
<td>- Deep Creator (Right Hemisphere)</td>
<td>- OpenSceneGraph</td>
</tr>
</tbody>
</table>

### Case Study: Ogre 3D Game Engine

What is OGRE?
- Object-Oriented Graphics Rendering Engine.
- Scene-oriented 3D engine.
- Object-oriented, written in C++.
- Interfaces with both Direct3D and OpenGL.
- Windows, Linux, MacOS.
Ogre Game Engine

Overview of Features:
- **Object-oriented design.**
- **Scene Management:** Supports BSP, octrees, occlusion culling, LOD.
- **Basic Physics:** Collision detection, rigid body physics.
- **Lighting:** Supports per-vertex, per-pixel, light-mapping.
- **Shadows:** Shadow mapping, shadow volumes.
- **Texturing:** Basic, multi-texturing, bump-mapping, mip-mapping, ...
- **Animation:** Inverse kinematics, skeletal animation, blending.
- **Meshes/Surfaces:** Mesh loading, skinning, progressive, Bezier patches.
- **Special Effects:** Environment mapping, lens flares, billboardig,
  particle system, motion blur, sky, water, fog.
- **Plug-ins:** Configurable extensions without recompilation.
- **Archiving:** Support for Zip/PK3 formats.

Ogre Design Highlights

Design Highlights/Innovations:
- **Design Patterns:** Based on standard object-oriented design patterns.
- **Decoupled scene graph/scene contents:** Scene graph makes minimal
  assumptions about the nature of the objects stored within it.

---

Chapter 4, Slide 5
Copyright © David Mount and Amitabh Varshney

Chapter 4, Slide 6
Copyright © David Mount and Amitabh Varshney
Ogre Design Highlights

Design Highlights/Innovations:
- **Plug-in Architecture**: Libraries may be added for easy extension.
- **Hardware Acceleration Support**: Using either OpenGL or Direct3D.
- **Flexible Render Queue Architecture**: Parts of a scene are organized in levels, to control rendering order.

Robust Material System:
- Provides a scripting system for designing materials.
- Automatic fallback mechanism allows system to gracefully degrade if graphics hardware lacks some rendering capability.

Native Mesh and Skeleton Format: Does not support 3rd-party mesh formats. Uses its own format for maximum efficiency.

Multiple Animation Types: Skeletal, morph, pose.

Extensible Resource Management: Allocates/deallocates memory for resources (e.g., meshes, skeletons, materials). Access Zip archives.

Ogre Subsystem Overview

Root Object:
- Main access point to Ogre.
- Creating it starts Ogre. Destroying it shuts Ogre down.

Resource: Anything Ogre needs to render a scene.
- **Mesh**: Binary format optimized for fast loading. May contain animation data.
- **Skeleton**: Bone hierarchy and key-frame data for animation.
- **Material**: Surface material: color, shading, texture properties.
- **GPU Program**: Can process GPU programs for vertex/pixels shaders.
- **Texture**: Images for texture mapping. (See OpenIL image library.)
- **Compositor**: Script-based viewport post-processing (for special visual effects).
- **Font**: Font definition files.

Resource Managers: Locate, load, allocate memory for resources.
Ogre Subsystem Overview

Scene Management:
- Ogre’s encapsulation of the Scene Graph object.
- Composed of SceneNode objects.
- Transformations: Apply to node and its descendents.
- Entities:
  - Most content is stored in this object.
  - Can be attached to a SceneNode.
  - Entities implement the MoveableObject interface, for transformations.

Render System:
- Interface with graphics system (e.g., OpenGL).

Render Target:
- The window to which graphics is rendered.
- Typically your graphics viewport.
- May be a texture (for certain special effects, such as reflection).

Ogre Subsystem Overview

Ogre Managers: Single instance objects that manage various Ogre objects. Examples:
- LogManager: Logging messages.
- PlatformManager: Provides access to details of underlying hardware.
- ArchiveManager: Resource management for containers, such as zip files.
- ParticleSystemManager: Details and implementation of particle systems, emitters, and affectors.
- MaterialManager: Maintains loaded Material instances.
- SkeletonManager, MeshManager: Maintain the associated objects.
- TextureManager: Maintains access to textures.
- ... (many more)

Accessing managers: Each manager has one instance: getSingleton.
MeshManager::getSingleton().someMethod();
Ogre: Getting Started

Initialization: Create a Root object. Options:

\[ \text{Root* mRoot = new Root();} \]
\[ \text{Root* mRoot = new Root("plugins.cfg");} \]
\[ \text{Root* mRoot = new Root("plugins.cfg", "ogre.cfg");} \]
\[ \text{Root* mRoot = new Root("plugins.cfg", "ogre.cfg", "ogre.log");} \]

Configuration Files: Contain information for start-up purposes.

Plugins.cfg: List of Ogre library plug-ins. Example:

```
# Define plugin folder
PluginFolder=.

# Define plugins
Plugin=RenderSystem_Direct3D9
Plugin=RenderSystem_GL
Plugin=Plugin_ParticleFX
Plugin=Plugin_BSPSceneManager
Plugin=Plugin_OctreeSceneManager
Plugin=Plugin_CgProgramManager
```

Log file for diagnostic and exception logging.

Ogre: Getting Started

More Configuration Files:

Ogre.cfg: Basic rendering/window options. Ogre’s start-up screen allows you to adjust these options.

```
Render System=OpenGL Rendering Subsystem
[Direct3D9 Rendering Subsystem]
Allow NVPerfHUD=No
Anti aliasing=None
Floating-point mode=Fastest
Full Screen=Yes
Rendering Device=Mobile Intel(R) 945GM Express Chipset Family
VSync=No
Video Mode=800 x 600 @ 32-bit colour

[OpenGL Rendering Subsystem]
Colour Depth=32
Display Frequency=60
FSAA=0
Full Screen=Yes
RTT Preferred Mode=FBO
VSync=No
Video Mode=1280 x 1024
```
More Configuration Files:

resources.cfg: Provides locations of resources for resource manager.

```plaintext
[Bootstrap]
Zip=C:/OgreSDK/media/packs/OgreCore.zip

[General]
FileSystem=C:/OgreSDK/media
FileSystem=C:/OgreSDK/media/fonts
FileSystem=C:/OgreSDK/media/materials/programs
FileSystem=C:/OgreSDK/media/materials/scripts
FileSystem=C:/OgreSDK/media/materials/textures
FileSystem=C:/OgreSDK/media/models
FileSystem=C:/OgreSDK/media/overlays
FileSystem=C:/OgreSDK/media/particle
FileSystem=C:/OgreSDK/media/gui
FileSystem=C:/OgreSDK/media/DeferredShadingMedia
Zip=C:/OgreSDK/media/packs/cubemap.zip
Zip=C:/OgreSDK/media/packs/cubemapsFS.zip
Zip=C:/OgreSDK/media/packs/dragon.zip
Zip=C:/OgreSDK/media/packs/fresnelDemo.zip
Zip=C:/OgreSDK/media/packs/ogreTestmap.zip
Zip=C:/OgreSDK/media/packs/skybox.zip
```

This example is provided with the Ogre demo programs.

Note: You may need to modify the default contents of this file with the actual path names on your system.

Initialize Render Window and Configuration:

```cpp
if (mRoot->showConfigDialog()) {
    mRoot->initialise(true, "My Render Window");
    RenderWindow* mWindow = mRoot->getAutoCreatedWindow();
}
```
Create a Camera:

```
Camera* mCamera;
// Create the camera
mCamera = mSceneMgr->createCamera( "PlayerCam" );

// Position it at 500 in Z direction
mCamera->setPosition( Vector3( 0, 0, 500 ) );

// Look back along -Z
mCamera->lookAt( Vector3( 0, 0, -300 ) );
mCamera->setNearClipDistance( 5 );
```

- `setPosition(...)`: Specifies the camera’s position.
- `lookAt(...)`: Specifies what the camera is looking at.
- `setNearClipDistance(...)`: Sets distance to near clipping plane.
- You can set other things, such as far clipping plane and aspect ratio.

Create a Viewport: This is where graphics are sent to. We set aspect ratio to that of the viewport.

```
// Create one viewport, entire window
Viewport* vp = mWindow->addViewport( mCamera );
vp->setBackgroundColour( ColourValue( 0, 0, 0 ) );

// Alter the camera aspect ratio to match the viewport
mCamera->setAspectRatio(
    Real( vp->getActualWidth() ) / Real( vp->getActualHeight() )
);
```

- `addViewport(...)`: Add a viewport to the current window.
- `setBackgroundColour(...)`: Set the default color (to black).
- `setAspectRatio(...)`: Set the camera’s aspect ratio.
Frame Listener: Encapsulates callbacks for each render cycle.
- `frameStarted( ... )`: Invoked before frame is drawn.
- `frameEnded( ... )`: Invoked after frame is drawn.

```cpp
class myFrameListener : public FrameListener {
public:
    bool frameStarted ( ... );  // we'll discuss this later
    bool frameEnded ( ... );
};

bool myFrameListener::frameStarted ( ... ) {
    // ... do input and other per-frame processing
    return true;  // return false to end program
}

bool myFrameListener::frameStarted ( ... ) {
    // ... do post-frame processing
    return true;
}
```

Render Loop: Start the rendering process.

```cpp
Root mRoot = new Root ( ... );
MyFrameListener myListener;
mRoot->addFrameListener( myListener );
/* Start rendering process. Passes control to Ogre and returns
 only for callbacks. */
mRoot->startRendering();
```

- Frame listener must be added to the root before starting the rendering process.
Sample code to create 3 robot entities and transform them.

```c
void createScene( void ) {
    // set background lighting
    mSceneMgr->setAmbientLight( ColourValue( 1, 1, 1 ) );

    Entity* ent = mSceneMgr->createEntity( "Robot", "robot.mesh" );
    // create entity from mesh
    SceneNode* node = mSceneMgr->getRootSceneNode()->
        createChildSceneNode( "RobotNode", Vector3( -100, 0, 0 ) );
    node->attachObject( ent );
    node->yaw( Degree( -90 ) ); // rotate about y by -90 degrees

    ent = mSceneMgr->createEntity( "Robot2", "robot.mesh" );
    node = mSceneMgr->getRootSceneNode()->
        createChildSceneNode( "RobotNode2" );
    node->attachObject( ent );
    node->pitch( Degree( -90 ) ); // rotate about x by -90 degrees
    // Create a third robot. Code omitted.
}
```
Summary

Summary:
- Game engines
- The Ogre 3D game engine

What’s Next?
- More Ogre details to come