CMSC 498M: Chapter 4
Game/Graphics Engines

Reading: The material for this lecture is adapted from

Overview:
- Introduction to game and graphics engines
- The Ogre 3D game engine
- Game/graphics engine architecture

Game/Graphics Engines

Game/Graphics 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 Windows, Linux, Mac OS, xBox, PS3, Wii, iPhone.

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/Graphics Engines

<table>
<thead>
<tr>
<th>Commercial Engines:</th>
<th>Open Source Engines:</th>
</tr>
</thead>
<tbody>
<tr>
<td>C4 Engine</td>
<td>OGRE (C++/OGL, DX/Win, Mac, Linux)</td>
</tr>
<tr>
<td>(C++/OGL/Win, Mac, PS3)</td>
<td></td>
</tr>
<tr>
<td>Torque Game Engine</td>
<td>Irrlicht (C++/OGL, DX/Win, Mac, Linux, XBox)</td>
</tr>
<tr>
<td>(C++/DX/Win, Linux, Mac)</td>
<td></td>
</tr>
<tr>
<td>3DGameStudio</td>
<td>Panda3D (C++, Python/OGL, DX/Win, Mac, Linux)</td>
</tr>
<tr>
<td>(C++/DX/Win)</td>
<td></td>
</tr>
<tr>
<td>TV3D SDK 6.5</td>
<td>Crystal Space (C++/OGL/Win, Mac, Linux)</td>
</tr>
<tr>
<td>(C++, C#, VisualBasic/DX/Win)</td>
<td></td>
</tr>
<tr>
<td>Unity</td>
<td>jME (Java/OGL/Win, Mac, Linux)</td>
</tr>
<tr>
<td>(C#, Java/OGL, DX/Win)</td>
<td></td>
</tr>
<tr>
<td>Leadwerks Engine 2</td>
<td>Blender Game Engine (C++, Python/OGL/Win, Mac, Linux)</td>
</tr>
<tr>
<td>(Many!/OGL/Win)</td>
<td>Reality Factory (C++/OGL, DX/Win)</td>
</tr>
<tr>
<td>ShiVa Engine</td>
<td></td>
</tr>
<tr>
<td>(C++/OGL/Win, Linux, Mac, Wii, iPhone, Android)</td>
<td></td>
</tr>
<tr>
<td>Esenthel Engine</td>
<td></td>
</tr>
<tr>
<td>(C++/DX/Win)</td>
<td></td>
</tr>
</tbody>
</table>

Legend:
- OGL – OpenGL
- DX – DirectX
- Win – Windows
- Mac – MacOS

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.

Source: DevMaster.net, Feb 2011
Ogre Game Engine

Overview of Features:

**General:** Object-oriented, scene-graph based, flexible, open-source.

**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.

**Special Effects:** Environment mapping, lens flares, billboardin,
particle system, motion blur, sky, water, fog.

**Plug-ins:** Configurable extensions without recompilation.

**Archiving:** Support for Zip/PK3 formats.

---

Ogre High-Level Overview

**Scene Manager:** Contents of the scene, its structure, how it’s viewed from cameras, etc.

**Resource Manager:** Geometry, textures, fonts, whatever.

**Rendering:** Lower-level end of the rendering pipeline.
Ogre Scene Graph

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.

```
Ogre Scene Graph

Scene Graph
  (attached)
    Scene Node
      (attached)
        Moveable Object
          (implement)
            Entity
              (contains)
                Subentity
                  (implements)
                    Renderable
                      Used for rendering

Scene Graph
  (attached)
    Scene Node
      (attached)
        Moveable Object
          (implement)
            Entity
              (contains)
                Subentity
                  (implements)
                    Renderable
                      Used by application
```

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.

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:

```
Root* mRoot = new Root();
Root* mRoot = new Root("plugins.cfg");
Root* mRoot = new Root("plugins.cfg", "ogre.cfg");
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
```
**Ogre: Getting Started**

**More Configuration Files:**
- **Ogre.cfg**: Basic rendering/window options. Ogre's start-up screen allows you to adjust these options.

```
[OpenGL Rendering Subsystem]
Render System=OpenGL 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

[Direct3D9 Rendering Subsystem]
Colour Depth=32
Display Frequency=60
FSAA=0
Full Screen=Yes
RTT Preferred Mode=FBO
VSync=No
Video Mode=1280 x 1024
```

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

```
[Bootstrapping]
Zip=C:/OgreSDK/media/packs/OgreCore.zip
Zip=C:/OgreSDK/media/packs/cubemap.zip
Zip=C:/OgreSDK/media/packs/cubemapsJS.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.
Ogre: Getting Started

Initialize Render Window and Configuration:

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

Create a Camera:

```cpp
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.

```
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::frameEnded ( ... ){
        // ... do post-frame processing
        return true;
    }
```
Ogre: Getting Started

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.

---

Ogre: Getting Started

Sample code to create an Ogre Head with a light source.

```cpp
using namespace Ogre;
void TutorialApplication::createScene(void) {
    Entity* head = mSceneMgr->createEntity("Head", "ogrehead.mesh");
    SceneNode* headNode = mSceneMgr->getRootSceneNode()->createChildSceneNode();
    headNode->attachObject(head); // attach entity to this node
    mSceneMgr->setAmbientLight(Ogre::ColourValue(0.5, 0.5, 0.5));
    // add another light source
    Light* l = mSceneMgr->createLight("MainLight");
    l->setPosition(20, 80, 50); // ... at this location
}
```
Game Engine Architecture

- We have discussed what game and graphics engines do, and we have given an example, but how would you go about designing one of your own?

Components:
- What are the major components of a game engine?
- How to separate game-independent components from game-dependent components?

Organization:
- How are these components defined and organized?

Structure:
- Assuming an object-oriented approach, what class structure should be used for the various elements?
Game Engine Architecture - Components

Game Engine Architecture

**Substrate:**
- Hardware (PC, Xbox, PS3, …) and operating system
- Graphics API (OpenGL, DirectX)
- Third-party libraries (basic data structures, networking support)
- Math libraries (trigonometry, linear algebra, geometry)

**Core Systems:**
- Memory allocation
- Engine configuration (global engine configuration)
- Parsers (XML for reading configuration files)
- Debugging and performance (unit testing, profiling, error logging)
- Localization (language preferences, time-zone, cultural modifications)
- Start-up/shut-down (initializations and saving final state)

---

Game Engine Architecture - Components

**Game Engine Architecture - Major Components**

**Resource Manager:**
- 3D models, skeletons, animations
- Textures
- Text (help files and narration) and audio
- Loading and decompression

**Low-level Rendering System:**
- Materials and shaders
- Lighting (and static shadowing)
- Texture and mesh management (what must be loaded to GPU?)
- Cameras
- Viewports

**High-level Rendering System:**
- Spatial subdivision (BSP, octree) and occlusion culling
- Level of detail (LOD) - simplifying models for distant objects
- Special effects (particle systems, environment mapping, motion blur)
**Game Engine Architecture - Components**

**Game Engine Architecture - Major Components**

**Human Device Interface:**
- Input device configuration
- Input interface (mapping low-level inputs to game operations)

**Physics:**
- Object physical states (position, velocity, orientation, angular velocity)
- Bounding volumes and collision detection
- Forces and motion constraints (e.g., inverse kinematics)
- Rigid bodies and physical integration

**Animation:**
- Interpolation (linear and spherical, a.k.a., LERP and SLERP)
- Animation controllers
- Inverse kinematics (mapping contact constraints to motion)
- Animation decompression and playback

**Audio:**
- DSP and playback

**Online Multiplayer:**
- Authentication and registration
- Game-state replication
- Latency compensation (dealing with slow networks and fast games)

**Gameplay Foundations:**
- Static world elements
- Dynamic game object model
- Event/messaging system (communication between game components)

**Game-Specific Systems:**
- Player mechanics (movement, camera-relative control, state)
- Game cameras (fixed, player following)
- AI (goals and decision making, path finding, coordinated movement)
- Other game elements (power-ups, vehicles, weapons, ...)

Chapter 4, Slide 25
Copyright © David Mount and Amitabh Varshney

---

**Game Engine Architecture - Components**

**Game Engine Architecture - Major Components**

**Online Multiplayer:**
- Authentication and registration
- Game-state replication
- Latency compensation (dealing with slow networks and fast games)

**Gameplay Foundations:**
- Static world elements
- Dynamic game object model
- Event/messaging system (communication between game components)

**Game-Specific Systems:**
- Player mechanics (movement, camera-relative control, state)
- Game cameras (fixed, player following)
- AI (goals and decision making, path finding, coordinated movement)
- Other game elements (power-ups, vehicles, weapons, ...)

Chapter 4, Slide 26
Copyright © David Mount and Amitabh Varshney
Core Systems

How are these game-engine components designed and implemented?

Low-Level System:

Basic Data Structures:
- Arrays (extensible) - fast indexing, fast insertion/deletion at the end.
- Lists - slow indexing, fast insertion/deletion from any position
- Maps (hash tables - one entry per key) - fast searching and insertion
- Dictionaries (hash tables - multiple entries per key) - (same as map)
- Provided by standard libraries (STL for C++ and Java SE packages)

Platform-Specific Concepts:
- Endianness - Are numbers stored low-order byte first (little endian) or high-order first (big endian)?
- System time - Converting from OS’s internal format
- File system
- Memory allocation/deallocation

Source: 3D Game Engine Architecture by D. H. Eberly

Core Systems

Mathematics System:

Basic Math Functions: Trigonometry, square roots, ...
Math Constants: π, e, degrees-to-radians
Numeric Limits: max/min values for ints, floats, ...
Linear algebra:
- 2-d and 3-d vectors: (possibly homogeneous-coordinate versions)
- Affine geometry: Represent points and vectors as separate objects
- Matrices: multiply, transpose, invert
- C++: Operator overloading works very well here
- Conversions: Provide conversions to alternate formats (e.g., OpenGL)

Quaternions: For 3-d rotations about arbitrary axes
Interpolation: Linear (for points) and spherical (for rotations)
Basic Geometric Objects: Lines, planes, spheres, triangles
Colors: RGB or RGBA

Many of these are built-in or can be downloaded from the Web.
Core Systems

Object System:

Run-Time Type Information (RTTI):
• To keep track of polymorphic type information at run-time
• Object-oriented languages like Java and C++ do this automatically:

```cpp
class Weapon {
    virtual void shoot();
};
class AK47: public Weapon {
    virtual void shoot();
};
Weapon* p = new AK47();
p->shoot(); // invokes AK47::shoot()
```

• If you use C, you will need to implement this yourself (e.g., jump table)

Name Generation and Unique Identifiers:
• Useful for providing names for dynamically created objects

Smart Pointers:
• Pointers to copies of shared objects.
• Use reference counters for garbage collection.

Controllers:
• Most game objects can be altered over time (moved, reshaped)
• A controller is an object that encapsulates such a transformation
• Controllers can be associated with game objects:

```cpp
class Controller { // generic controller
    Object* obj; // associated game object
    double startTime; // controller start time
    void setObject(Object* o);
    void initialize(double start);
    double getControllerTime(double gameTime);
};

class ConcreteController: public Controller {
    Transformation getTransformation(double gameTime);
};
class Object { // game object
    ControllerList* ctls;
    void addController(Controller* c);
    void remController(Controller* c);
    void updateControllers(double gameTime);
};
```
Scene Graphs and Renderers

Core Classes:

Spatial:
- A generic class that refers to any object that can be transformed
- Spatial is a subclass of Object, and so it can be associated with a controlling transformation

Node:
- A node in the scene graph
- Node is a subclass of Spatial, and so can be assigned a controller
- Children may be attached and detached

```cpp
// transformable object
class Spatial : public Object {
    Spatial* getParent(); // get parent in scene graph
    void setParent(Spatial* p); // parent in scene graph
};
// node in a scene graph
class Node : public Spatial {
    int attachChild(Spatial* c);
    int detachChild(Spatial* c);
    Spatial* getChild(int i);
};
```

Geometric State:

- Transformation:
  - Encapsulates a transformation
  - Implemented as a 4×4 matrix
  - Methods: rotation, translation, …, composition, inverse, …
- Bounding volume:
  - Class encapsulating a bounding shape (e.g., box, sphere, ellipsoid)
  - Methods: visibility culling and collision detection
- Geometric updates:
  - Transformations are propagated down the scene graph
  - Bounding volumes are updated and propagated up the scene graph
Scene Graphs and Renderers

Core Classes:

Geometric Types:
- Types: Polylines, triangle meshes, particle sets, ...
- Common methods: e.g., loading, drawing, bounding volumes
- Particular methods: e.g., computing normals for a mesh

Render State:
- These encapsulate rendering information
- They can be attached to Node objects

Global State: Classes encapsulate basic rendering properties.
- Alpha blending - Used for blurring and transparency
- Material - Encodes surface material properties (diffuse, specular, ...)
- Fog - Encodes global fog parameters
- Culling - Encodes whether/how culling is performed (back/front face)
- Wireframe - Used to enable wireframe rendering of meshes (debugging)
- Shading - Encodes global shading model (smooth, flat)

Render State:

Light: Class encapsulating a light source
- Type - point source, spot light, directional light, ...
- Location - position, direction (for spot light)
- Intensity - light color and strength
- Attenuation - intensity decrease with distance
- Other parameters - spot-light exponent, spot-light angle, ...

Texture: Class encapsulating a texture object
- Image - stores image array
- Filters - how is texture to be filtered when minimizing/maximizing.
- Mip-mapping - compute and store mip-maps
- Repeat mode - repeat or clamped?
- Drawing mode - replace, modulate, blend, ...

Multi-Texturing: Applying multiple textures
- Used for shadow mapping
Scene Graphs and Renderers

Renderers and Cameras:

Camera: An object that encodes a single camera (view)
- Subclass of Spatial, so cameras can be transformed
- Camera position: location, view direction, up vector (gluLookAt)
- Projection properties: field-of-view, aspect ratio, near/far clipping plane distances (gluPerspective)
- Viewport: size and location on graphics window (glViewport)
- Culling: removing clearly invisible objects (based on bounding volumes)
- Picking:
  - Provides a method for mapping screen coordinates to an object
  - Done by doing a ray-shoot into the scene to find the first hit

Renderer:
- Provides a method for single-pass drawing of scene-graph objects
- Provides enable/disable options: lighting, colors, texturing, normals, ...
- Frees programmer from mess of dealing directly with graphics API

Advanced Issues in Scene Graphs

Advanced Scene Graph Entities:

Billboarding:
- Using 2-d textures to create 3-d illusion

Level-of-detail (LOD):
- Hierarchy of models at successively lower resolutions

Hidden-surface-removal:
- When z-buffering disabled
- Render from back to front
- Later objects overwrite earlier ones

Visibility culling:
- Based on a graph of cells (rooms) and portals (passages between rooms)

Curves and surfaces/Terrains:
- Loading, decompressing, smoothing, and rendering

Many more! Animation, skinning, GPU shaders, ...
## Summary

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

<table>
<thead>
<tr>
<th>Low-level System:</th>
<th>Render State:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Basic data structures</td>
<td>- Lights</td>
</tr>
<tr>
<td>- Platform/locale specific</td>
<td>- Textures</td>
</tr>
<tr>
<td>- Math system</td>
<td>- ...</td>
</tr>
<tr>
<td>- Resource manager</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object System:</th>
<th>Physics:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Name generation</td>
<td>- Physics state</td>
</tr>
<tr>
<td>- Smart pointers</td>
<td>- Collision detection/response</td>
</tr>
<tr>
<td>- Objects and Controllers</td>
<td>- Inverse kinematics</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core Classes:</th>
<th>AI:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Spatial (transformable)</td>
<td>- Agents</td>
</tr>
<tr>
<td>- Node (entry in scene graph)</td>
<td>- Finite-state automata</td>
</tr>
<tr>
<td>- Geometric state</td>
<td>- Flocking &amp; coordinated motion</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Renderers and Cameras:</th>
<th>Audio:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Camera</td>
<td>- Playback</td>
</tr>
<tr>
<td>- Renderer</td>
<td>- Digital signal processing</td>
</tr>
</tbody>
</table>

And many more...