Game Engineering

CS420-2016S-02

Introduction to OGRE

David Galles

Department of Computer Science
University of San Francisco
02-0: Anatomy of a Game Program

- The basics

  Initialize Rendering System
  Load Assets (models, animations, etc)
  Initialize non-graphics systems
  While (true)
    Update world
    Render next frame
**Anatomy of a Game Program**

- Slightly more complicated

Initialize Rendering System
Load Assets (models, animations, etc)
Initialize non-graphics systems
While (true)
  For each game system
    (Camera, AI, Player, world(physics))
    Call systems Think method, to do one frame’s worth of processing
Update Rendering system’s data structures
  (model locations, camera locations, etc)
Render next frame
We will be using Ogre in this class
  • Object-oriented Graphics Rendering Engine
Ogre is not a game engine, but a rendering engine
No physics, or anything else – just rendering
Intentional design decision – do one thing well, use other libraries & plugins to do other things
Ogre Idiosyncrasies

- Ogre does things a little differently

Initialize Rendering System
Load Assets (models, animations, etc)
Initialize non-graphics systems
Register Callbacks with graphics engine
Run Renderer
  (makes callbacks every frame)
You can register as many callbacks as you like
   • Camera system, AI, Player, World (physics manager)

However, you have no guarantee what order they will be called each frame

For more control, can register a single callback, that manages your other callbacks
Side note: you do not need to let Ogre have control of the rendering loop

Ogre has a “startRendering” method, that runs the main loop:

```java
while(true)
    // RenderOneFrame makes all callbacks
    // Does the rendering
    // returns false if any callback returns
    if (!RenderOneFrame())
        break
```

You can call lower-level Ogre methods yourself, if you wish
02-6: Setup

- Setup resources (filesystem stuff)
- Configure the renderer
  - various settings – resolution, OpenGL vs. DirectX, etc
- Create the SceneManager
  - Manages the rendering of the scene
  - Gives us access to models, cameras, etc.
  - Use the SceneManager to change the scene (can query it, too)
• Create cameras, viewports
  • For pong, single viewport (entire screen), single camera that controls the viewport
  • Could have multiple cameras, multiple viewports (split-screen, rear-view mirror, etc)
• Create “Scene”
  • Loading all of the required models
  • Setting up game systems (non-Ogre)
• Create FrameListener
  • Handles callbacks from the rendering loop
• When we create the scene, we need to add all of our models, lights, etc. to the scene

• While individual light sources will look much better, ambient light will allow us to see everything

```cpp
mSceneManager->setAmbientLight(ColourValue(1,1,1));
```

• The whole scene will be bathed in white light
02-9: Entities

• Adding the models
  • Create an entity, which contains
    • Mesh (all we care about for now)
    • Animation data, other information (we won’t use this just yet)
  • Entities are for small, movable geometry
  • Static geometry (mountains, buildings, etc) will be done in a different way
We create entities from the scene manager (pretty much everything happens through the scene manager)

createEntity takes a name (optional) and a filename

• The name is so that we can look up the entity later through the scene manager, without needing to hold on to the pointer

• Notice no path in the filename – we need to have our file structure set up beforehand so that the system can find everything
Scene Node

- We have the entity
  - Internal representation of the mesh (model)
- We need to position the entity in the world
- Attach it to a SceneNode
The “Scene” is a collection of objects to be rendered

- Represented as a graph (tree, really)
- Single root, has many children (each child can have children)
- Position of each scene node is an offset from its parent's position
  - Move the parent, everyone else goes along for the ride
02-13: Scene Graph
Create a child scene node, as a child of the root, at offset (0,0,0) from the root

Name can be used to get the scene node from the scene manager (optional!)

Position defaults to (0,0,0) (so this parameter is unnecessary)

Can pass in a rotation as well (as a Quaternion, more on those later)
Scene Node

- The scene node lets us define a position and orientation in the scene
- Then attach our entity to this scene node, and it is placed in the scene

mExampleSceneNode->attachObject(ent1);
You could hard-code in the addition of entities & scene nodes into your setup code.

Works OK for something simple like Pong.

For something complicated, you should read in the “level” from an outside text file.

Setup code parses this text file, sets up entities and scene nodes.
02-18: Onward!

- We have:
  - Initialized the graphics system
  - Set up a viewport & camera
  - Created all the necessary entities
  - Build the SceneGraph
- We’re ready to create game logic!
Main Loop

- Every frame, our frame listener will get a callback before any rendering starts:

  ```
  bool frameStarted(const FrameEvent &evt)
  ```

- the FrameEvent tells us how much time has passed since the last frame
  - Useful, since we don’t want the action in our world to be dependent upon framerate

- Return true to keep rendering, return false to stop render loop
bool PongFrameListener::frameStarted(const FrameEvent &evt)
{
    mInputHandler->Think(evt.timeSinceLastFrame);
    mAIManager->Think(evt.timeSinceLastFrame);
    mWorld->Think(evt.timeSinceLastFrame);
    mPongCamera->Think(evt.timeSinceLastFrame);

    bool keepGoing = true;

    if (mInputHandler->IsKeyDown(IOS::KC_ESCAPE) || mRenderWindow->isClosed())
    {
        keepGoing = false;
    }

    return keepGoing;
}
02-21: Unbuffered Input

• Setup the keyboard manager

```cpp
OIS::ParamList pl;
size_t windowHnd = 0;
std::ostringstream windowHndStr;

renderWindow->getCustomAttribute("WINDOW", &windowHnd);
windowHndStr << windowHnd;
pl.insert(std::make_pair(std::string("WINDOW"), windowHndStr.str()));
mInputManager = OIS::InputManager::createInputSystem(pl);
mKeyboard = static_cast<OIS::Keyboard*>(mInputManager->createInputObject(OIS::OISKeyboard,
false /* not buffered */ ));
```
02-22: Unbuffered Input

- Every frame, poll what keys are currently being depressed
- (Can also save the previous state, to see what was just pressed)

```c
void InputHandler::Think(float time)
{
    // Save old state
    mCurrentKeyboard->copyKeyStates(mOldKeys);

    // Capture input for this frame
    mKeyboard->capture();
}
```
You’ve created a basic pong game, and you want a nice way to show scores

Several Options
- Create 3D models of various numbers, use them for scoring
- Create a 3D scoreboard, change texture(s) to denote score
- Overlays
Overlays allow us to add 2D elements “in front of” (or overlayed over) our 3D scene

- Somewhat complicated, because the overlay system has a large amount of flexibility
- One or two bugs in overlays with the latest version of Ogre
- Can be script controlled
02-25: Overlays

- Overlays are containers for 2D elements
- Can have several overlays active at once
- Can turn overlays on / off
- Great for showing text (like scores), HUD elements (health bars, etc), debugging info, etc
02-26: Overlays

- Overlay is the top level container
- Can have one or more sub-containers (called panels) nested within the main overlay
- Can have further elements (more panels or text elements) nested within panels
02-27: Overlays

• Can do all sorts of cool things with overlays
  • HUDs
    • Health bars
    • Scores
    • Various other status information
  • 3D Cockpits
You want to create an overlay that has a background texture and some lettering. You need to create a material for the background, and a font for the lettering:

- test.material
- sample.fontdef
In Ogre 1.8 (and before), overlays were part of the main engine, and initialized by default.

Starting with Ogre 1.9, overlays are a separate system, and need to be initialized if you want to use them.

- Register the overlay system so that it gets a callback every frame to manage and display overlays properly

```cpp
mOverlaySystem = new Ogre::OverlaySystem();
mSceneMgr->addRenderQueueListener(mOverlaySystem);
```
Overlays in Code

- Overlays can be modified in code

```cpp
Ogre::OverlayManager& om = Ogre::OverlayManager::getSingleton();
Ogre::TextAreaOverlayElement *textArea =
    (Ogre::TextAreaOverlayElement *) om.getOverlayElement("Sample/Panel/Text");
textArea->setCaption("New Caption!")
```

- “Sample/Panel/Text” is the name given to this particular element in the overlay script
- Note typecasting
- Overlays can also be created in code if preferred.