Jason's Mostly-Useless Ogre-view

Ogre 1.10 Cheat-Sheet
Last Updated: 10/5/2017

I'm putting this Ogre-view (aka Overview; reference: Trenton Daugherty) together in the hopes that tackling Ogre in ETGG3801/3802 will be a bit easier. Ogre is a rather large project, and sometimes hard to navigate for a newcomer. This document isn't meant to replace the wealth of information online, just to be a "quick-start" guide. Let me know if there's anything you'd like to have added here…(the sooner the better)

1. Where to go when this document fails you

…and it will. I'd suggest these resources:

- The Ogre 1.10 API (lists all classes and methods). Generated using doxygen.
  https://ogrecave.github.io/ogre/api/1.10/
- The Ogre Manual. This is a great overview of the major objects. It is especially useful if writing any ogre scripts (e.g. materials, overlays, etc.)
  https://ogrecave.github.io/ogre/api/1.10/manual.html
- The Ogre Wiki. Has a number of articles and code snippets (beware, though, some articles refer to the experimental ogre (2.x) and others refer to anchient-Ogre (<=1.8)
  http://www.ogre3d.org/tikiwiki/tiki-index.php
- The Forums. The community is very active and (usually) helpful to n00bs.
  http://www.ogre3d.org/forums/

2. Why Ogre, and Why Ogre 1.10?

I've stuck with Ogre over the years because:

- It does many of the things Jim did in ETGG2801/2802 (e.g. shadows, shader-support, scene graphs, etc.). So…it is a nice base for our engine, without us having to use code from ETGG2802.
- It's a large software package not written by us. In many ways, adapting to another person's coding style is much harder than writing code yourself from scratch, but chances are, that's what you'll be doing in the real world.
- It's in C++. This course sequence will likely be your sole experience (in class) to C++ at SSU.
- The active community encourages you (hopefully) to go above and beyond what we do in class.
- Using Ogre allows us to focus on the more interesting aspects of game engine development (without having to re-do many things from 2801/2802). If there's something particularly interesting (e.g. Singletons), I'll have you implement it in class (instead of relying on Ogre's Singleton class).

Why did I choose to use this version of Ogre? It is admittedly a bit dated – some of the techniques (e.g. tessellation) you saw in 2802 probably aren't possible because we're mainly targeting DX9/11 and OpenGL3.0/4.0. There is a newer Ogre 2.x version, but it's still a bit unstable and there's very little documentation. The new version looks amazing (they're targeting Vulcan, Metal, DX12) and supports physically based materials. I spent nearly 3 weeks attempting to use Ogre 2.1 in this class, but I felt it was going to cause more headaches than warranted.

3. Creating a minimal ogre application, with explanations of each component

…don't skip the descriptions – you'll regret it later!

3.1. Creating the root

The Ogre root is the one and only Ogre object that you should allocate / deallocate with new / delete. Everything else Ogre / related will be cascade-deleted when the root is destroyed. When the root is created, many of the top-level manager are also created. This includes (but isn't limited to):

- **TextureManager**: Manages all textures
- **HighLevelProgramManager**: Manages all shader programs
- **ResourceGroupManager**: Groups resource by user-defined names (e.g. Level1)
- **CompositorManager**: manages full-screen effects (like HDR)

There are a few managers that are housed in external libraries (lib + dll). This includes:

- **OverlayManager**: creates and manages 2d "stuff" on the screen (e.g. HUDs / text)
- **ParticleEffectManager**: creates and manages…particle effects.

Normally, when creating the root, you pass it the filename (relative to our exe) of a plugins.cfg script. Those should have been copied from the ogre_sdk/bin folder, but you may want to modify it if you don't need all the plugins being loaded (your ogre startup time could be significantly faster if you don't load a plugin). One catch: we have different plugins for Debug and Release builds, so you must use some kind of preprocessor macro to do it different (hint: use the _DEBUG VS2017 symbol that is only defined in Debug configurations)

The code to create the root is pretty simple:

```
Ogre::Root * mOgreRoot = new Ogre::Root("plugins.cfg");
```

And to delete it:

```
delete mOgreRoot;
```

3.2 Setting the RenderSystem

One of the cool things about Ogre is it is thin wrapper around OpenGL, DirectX, Vulcan (Ogre 2.x), Metal (Ogre 2.x), iOS, Android, etc. You generally have one or more RenderSystem dll's that are loaded after creating the root (we'll discuss one of about 3 methods to load this render system).

Ogre has some routines to simplify this task (including "latching" onto a Win32 dialog box), but I prefer the manual method of initializing the rendersystem. This way, we can completely control things like the rendering library (OpenGL / DirectX), resolution, vsync, etc. We could even make our own in-game configuration system!

You can get the list of available renders with this code snippet. Note: If plugins(_d).cfg wasn't found, this list will be empty! Usually this is because the file isn't there, or your IDE isn't set to use your exe-file location as its working directory.

```
Ogre::RenderSystemList rsl = mRoot->getAvailableRenderers();
```

This acts like a std::vector and *should* be populated with at least 2 render systems:

- OpenGL Rendering Subsystem
- Direct3d11 Rendering Subsystem
- (there are many more, by the way, these are the ones default-built by cmake)

If you feel lucky, pick the first one, or if you want to be a little smarter about it, you could search for the one that your graphics card supports best.

After you have a render system from rsl (each is a RenderSystem*), you can change options that are passed to the render system when it is initialized, by calling getConfigOption on a particular render system. This returns a std::map of String => ConfigOption, where ConfigOption includes both the current value and allowable values for a particular option (both the current and allowed values are strings). Once you know the option you want to change and the value you want to set it to, you can call sestConfigOption on the render system, passing it a modified ConfigOptionMap. Or, more conveninently, you can call the setConfigOption method of the render system.

Finally, set the renderSystem with:

```
mRoot->setRenderSystem(RenderSystem * chosen_render_system);
```

3.3. Creating the window

The Window is just that – a Win32 / OSX window. In Ogre, a texture is stored internally that is synced with the window (likely with some kind of double-buffering). We have a choice: Ogre will create a window for us. In our case, though, we would like it to render to the Win32 win we've already created. This is very platform-specific, so it would be wise to surround this with appropriate #if checks.

Create a parameter map (this is the platform specific part – hWnd is the window handle from an existing win32 window)

```
Ogre::String winHandle = Ogre::StringConverter::toString((size_t)hWnd);
Ogre::NameValuePairList params;
params["externalWindowHandle"] = winHandle;
```

The rest of the process is the same for any platform. The next step "cements" our choice of render system and creates an Ogre window.

```
mRoot->initialise(false);

// Note: The title and dimensions are ignored since we're giving it a window
// handle in our params map.
Ogre::RenderWindow * mWindow = mRoot->createRenderWindow("SSUGE",
           1024, 768, false, &params);
```

A note about saving Ogre pointers. Ogre stores references to these in a tree-like fashion (for example, the root contains a pointer to the scene manager; the scene manager contains pointers to entities, cameras, etc.). Saving pointers (perhaps to instance attributes) are optional – my rule of thumb: if I find myself accessing the same thing more than 1 time per run, it's probably more efficient (and more readable) to save them away. But…don't delete the memory pointed to by Ogre! Only the root should be deleted – everything else will be cleaned up in Ogre destructors.

3.4. Creating the scene manager

The scene manager is one of the most frequently used objects in Ogre. It is used to:

- create all scene components (lights, mesh instances [entities], cameras, particle effects, etc.)
- give us a fast means to access any of those components (by unique name)
- store the root of the scene graph.
- trigger rendering of the scene graph (in some kind of depth-first traversal of nodes)

We'll see more about SceneNodes and the various scene components later, but for now we'll focus the discussion on the SceneManager.

There are several types of Scene Managers in Ogre, each optimized for a certain type of scene. You can (relatively) easily create other types of scene mangers by making a dll with specific functions. You can also have more than one scene

manager in existence (for example, your game could switch between indoor and outdoor scenes using two different scene managers, each optimized for that type of geometry).

Here are the two methods to create a SceneManager:

```
Ogre::SceneManager * Ogre::Root::createSceneManager(Ogre::SceneTypeMask m,
            const std::string name = "");

// or

Ogre::SceneManager * Ogre::Root::createSceneManager(std::string typeName,
            const std::string name = "");
```

The second parameter in both is an optional name for the scene manager (useful if you're not storing a pointer to the scene manager, and later want to ask the root for a particular manager). The first parameter can be a SceneTypeMask, which has these allowable values (there are a few others – these are the main ones:

- Ogre::ST_GENERIC
    - if you're loading Plugin_OctreeSceneManager.dll [through plugin.cfg], this is an Octree manager – good all-around optimizations
    - There's also a non-standard DotSceneManager, which is what this loads.
- Ogre::ST_INTERIOR: Uses the Plugin_BSPSceneManager.dll; Good for interior areas.
- Ogre::ST_EXTERIOR_REAL_FAR: Uses the paging plugin (which I told you not to build b/c is uses boost)

The second variant of creating SceneManagers is useful if you have created your own plugin (one of the functions you define in your dll gives Ogre the name to use here).

3.5. Creating the Viewport(s)

A viewport is a rectangular area marked as a RenderTarget. Currently, Ogre creates a 1-to-1 connection between a camera and a viewport – the camera tells us what to render, the viewport tells us where to render. Although, if needed, you can switch cameras that are attached to a viewport (e.g. when toggling between squad-mates in a tactical shooter). You can have more than one viewport (e.g. one for mini-map), and you usually add / remove viewports relatively infrequently (in fact, most of the time, you'll just make one main viewport and that's it).

Section 4.7.3 shows how to create a Camer using the Scene Manager.

To create a viewport, use:

```
Ogre::Viewport * Ogre::RenderWindow::addViewport(Ogre::Camera *,
            int zorder = 0, float left = 0.0f, float top = 0.0f,
            float width = 1.0f, float height = 1.0f)
```

You can always get a viewport from the render window (either by order that it was added or by zorder)
The optional zorder is useful for layering (lower numbers render first). The left, top, width, and height parameters tell what proportion of the screen to render to (so the viewport scales when / if the render window changes size). A left and top of 0.0 indicates the upper-left. The screen is 1.0 width and height wide / tall.

To modify the camera attached to an viewport, use:

```
Ogre::Viewport::setCamera(Ogre::Camera*);
```

You can optinally tell the viewport what color to clear with using:

```
Ogre::Viewport::setBackgrounColour(Ogre::ColourValue);
```

3.6 Enumerating resource locations and loading resources

This should be done before you actually attempt to load a resource (e.g. an instance of a mesh). The first step merely tells ogre one or more places to look for resources. You would call this once for each folder or zip file you want to add to your resource path.

```
Ogre::ResourceGroupManager::getSingleton().addResourceLocation(
            "path-relative-to-exe", "FileSystem",
            "Resource-group-name");
```

The first parameter is a dos-style relative path, relative to the exe (e.g. "..\\media\\textures"). The second parameter can either be "Zip" or "FileSystem" (watch the capitalization). The third parameter is the name of an existing Resource Group. The name "General" is auto-created for you. If you want to add other resource groups, do so with:

```
Ogre:ResourceGroupManager::getSingleton().createResourceGroup(std::string);
```

Breaking your resources into groups is handy because you can batch-load and un-load and re-load a group to manage memory (Ogre can also be configured to auto-manage your resource memory).
The next step is to actually load your resources into memory. This is the concise way. You can alternatively load / unload only select groups or entries.

```
Ogre::ResourceGroupManager::getSingleton().initialiseAllResourceGroups();
```

You hopefully raised an eyebrow at the getSingleton method. This is a common design pattern that we'll discuss shortly (probably in the next lab).

3.7. Creating "Stuff" in your Scene

As we've mentioned, the SceneManager holds the root SceneNode in our hierarchy of nodes. In a later lab / section, we'll explore how the parent-child relationship between nodes can be used to easily create and animate a scene. For now, suffice it to say for something to render, it must be attached to an ancestor SceneNode of the SceneManager's root.

To create a new SceneNode that is an immediate child of the root, you can use:

```
mSceneManager->getRootSceneNode()
```

which returns a SceneNode pointer. You can then create a new child scene node of that node by calling

`Ogre::SceneNode::createChildSceneNode` (for now, pass no parameters).

All things that are renderable in Ogre are derived from a common Ogre::MoveableObject class. The SceneNode class contains a single method called attachObject that is passed a pointer to an abstract class called MoveableObject. Because of C++'s polymorphic nature, we'll actually pass pointers to Entities, Lights, etc. To actually create these objects, you need to go through the SceneManager (it is actually managing the memory; we're just given a pointer). Creating the various objects follows the same pattern. Important Note: all objects (of the same type) must have a unique name:

```
Ogre::Light * SceneManager::createLight(std::string name);
Ogre::Entity * SceneManager::createEntity(std::string name,
            std::string mesh_fname);
Ogre::Camera * SceneManager::createCamera(std::string name)
```

To attach a MoveableObject to a SceneNode, use:

```
Ogre::SceneNode::attachObject(Ogre::MoveableObject *);
```

You can remove a particular object (or all objects in the second one) using:

```
Ogre::SceneNode::detachObject(Ogre::MoveableObject*);
Ogre::SceneNode::detachAllObjects();
```

The scene manager can, if requested, permanently delete a scene node or MoveableObject. You don't *have* to do this – when the root is destroyed, the scene manager is destroyed. That will in turn trigger all scene-related objects to be destroyed. However, if you think you might use the name again (or just want more control), you can destroy the various objects like this:

```
mSceneManager->destroyEntity(Ogre::Entity*);
mSceneManager->destroyCamera(Ogre::Camera*);
mSceneManager->destroySceneNode(Ogre::SceneNode*); // See the note below
```

…

SceneNodes require a bit more care before deleting. You must ensure that the scene node is detached from the scene graph; if not, the parent node will hold a garbage pointer and ogre will try to delete it when the scene graph is destroyed (bad things will happen in this case…). To detach a scene node, you can use something like this:

```
mChildNode->getParentSceneNode()->removeChild(mChildNode);
```

Once a mesh has been created, you modify its position, orientation, and scale through the scene node. This is just a handful of the transform-related methods. Note: these transforms are relative to the parent scene node (which for now is the root, so effectively we're not worrying about inherited transformations…yet). We'll look at this in more detail in a later section

```
Ogre::SceneNode::setPosition(Ogre::Vector3);
Ogre::SceneNode::setScale(Ogre::Vector3);
Ogre::SceneNode::setOrientation(Ogre::Quaternion);
```

Vectors are pretty self-explanatory. You could scale 50% like this:

```
snode->setScale(Ogre::Vector3(0.5f, 0.5f, 0.5f));
```

Quaternions are always a bit more troubling. I think the easiest way for a graphics programmer to approach them is as a rotation of x degrees around axis y. In fact, this is one way to create a Quaternion. For example to rotate 45 degrees about the axis (0, 1, 0), do this:

```
snode->setOrientation(Ogre::Quaternion(Ogre::Degree(45.0f),
           Ogre::Vector3(0, 1, 0));
```

Note: The Degree object above is telling Ogre that the 45.0f is a degree value (and not a radian). We're not converting to degrees.

3.7.1. Entities

Entities are instances of Meshes. Ogre only stores the actual vertex and index buffer once in memory. It then references that memory when issuing GL/DX draw calls. Each instance of the mesh can have separate materials (useful if you use re-skinned models) and separate animation states (if this is a rigged / animated [kinematic] mesh). You rarely interact directly with the raw mesh data. Instead, you'll create a mesh (as in the previous section) and attach it to a SceneNode to draw that mesh.

Ogre uses a binary .mesh format to store mesh files. There are a number of exporters for various modeling programs (Maya, Blender, 3DStudio, etc.). Each generally exports a (text) xml file. There is another tool "OgreXMLConverter" (it's built in the SDK if you use the TinyXML [or other] xml library) – or the 1.7 version (which generates 1.9-compatible meshes) is available online. The mesh files usually contain a link to the contents of a .material file (which will be explored in a later section) and possibly a .skeleton file (for kinematic animations).

3.7.2. Lights

Ogre supports the 3 traditional fixed-pipeline light sources (you can do others, but it would be through a shader instead). You indicate which you want after creating the light by passing Ogre::Light::LT_POINT, Ogre::Light::LT_SPOTLIGHT, or Ogre::Light::LT_DIRECTIONAL to this method.

```
Ogre::Light::setType(Ogre::Light::LightType t); // LightType is an enumeration
```

For point lights (you can figure out the other light types yourself), you can optionally use these methods to modify the light:

```
Ogre::Light::setCastShadows(bool);
Ogre::Light::setDiffuseColor(Ogre::ColourValue); // The rgba values are in
                                                 // the 0-1 range
Ogre::Light::setDirection(Ogre::Vector3); // Can also be set in scene node.

// The rest are pretty self-explanatory.
```

There is a setPosition, but I'd recommend you attach it to a scene node and set the position of the scene node (the effect is the same; doing it this way just feels more consistent).

3.7.3. Cameras

A few interesting methods of cameras

```
Ogre::Camera::setNearClipDistance(float);
Ogre::Camera::setAspectRatio(float);
Ogre::Camera::setAutoAspectRatio(bool);
```

This one is interesting. SceneNode's also have this method (and it works exactly the same). If you imagine a scene node as having a local coordinate system (x, y, and z axes, like in ETGG1803), ogre rotates this scene node such that the negative z axis is pointing towards this point. Warning: this doesn't unambiguously set the orientation. Often, the z axis is aligned properly, but the x and y axes are rotated at a weird angle. We'll likely run into this when looking at more sophisticated character controller schemes.

```
Ogre::Camera::lookAt(Ogre::Vector3);
```

3.7.4. Other "Stuff"

Later😊

3.8 Shadows

Ogre contains a handful of crude shadow systems (this is one of the major improvements in Ogre 2.x). You can create your own shadow system, but it's not for the faint of heart (one part is writing a shader, other parts are wrapping that inside ogre's render system). In the samples (contained in the Ogre SDK), there is one sample that showcases the various shadow techniques. Briefly, though:

- Stencil shadows: crisp edges that block the light. Uses the stencil buffer on the GPU.

- Texture shadows: renders shadows from the lights perspective (like stencil shadows), but instead of using the shadow buffer, uses a texture (clear = no shadows, color = shadows). You get a bit more control (including the ability to set shadow color), but you can usually the shadows look a little "pixely" unless you're very careful about the settings used and the relationship between the lights and objects.

There are two variants of each type: additive and modulative. They have subtle differences in how they mix the shadow color and the color of the object being rendered.
In my test, I used Stencil-Modulative using:

```
mSceneManager->setShadowTechnique(Ogre::SHADOWTYPE_STENCIL_MODULATIVE);
```

(sort of) related: You can set the ambient color for the scene

```
mSceneManager->setAmbientLight(Ogre::ColourValue);
```

Set the other variants in a similar manner. Make sure at least one light is configured as a shadow-caster. Beware: some light types work better with some shadow techniques (experiment to get the effect you like)

3.9. Rendering the world

For now, just call:

```
Ogre::Root::renderOneFrame();
```

This traverses the scene graph, rendering each object when it visits the scene node. Behind the scenes, lots of cool matrix concatenations, GL/DX state-changes, etc. are happening.

3.10 If something goes wrong at run-time

Ogre generates copious amounts of debug information that is stored in the /bin/$(Configuration)/ogre.log file. Check that out if weird things (like missing materials, run-time ogre crashes, etc.) occur.

4. Ogre Overlays

Ogre overlays are (usually) 2d graphical elements (images, text, textures, etc.) placed on top of the contents of a viewport. They can be used to created HUD's, menu's, etc.

I used these references when setting up overlays in my project. If something isn't covered here, you may wish to consult these references as well:

- http://www.ogre3d.org/tikiwiki/Creating+Overlays+via+Code
-

4.1. Adding Overlay support to your application

Since Ogre 1.9, Overlays aren't a "core" part of Ogre, so you must

- include:
  - OgreOverlay.h
  - OgreOverlayManager.h
  - OgreOverlayContainer.h
  - OgreTextAreaOverlayElement.h
  - OgreFontManager.h
  - (probably put these in stdafx.h if using pre-compiled headers)
- link: OgreOverlay[_d].lib
- Also make sure OgreOverlay[_d].dll is in the same folder as your executable.

If using the OgreBites framework, the overlay system has already been created for you. If not, you'll need to manually create it like so:

```
mOverlaySystem = new Ogre::OverlaySystem();
```

In either case, you need to tell your scene manager to include it as part of its rendering process by doing this:

```
mSceneManager->addRenderQueueListener(getOverlaySystem());        // Ogre Bites
mSceneManager->addRenderQueueListener(mOverlaySystem);            // No Bites
```

One key part of the Overlay System is the Overlay Manager: this creates and destroys overlays and overlay elements. It is created automatically (?) when creating the overy system. You can retrieve a pointer to it using it's singleton method:

```
Ogre::OverlayManager * mgr = Ogre::OverlayManager::getSingletonPtr();
```

4.2 Creating overlays and elements

You can have any number of overlays in your application. Each can be made visible or invisible at any time.

To create an overlay:

```
Ogre::Overlay * over = mgr->create("MyName");      // The name has to be unique!
```

To make an overlay visible or inivisible:

```
over->show();
over->hide();
```

To add content to an overlay, you must make at least one panel. You are able to add additional panels in a tree-like fashion if you wish to set up a complex GUI hierarchy. To create a panel:

```
Ogre::OverlayContainer * panel = (Ogre::OverlayContainer*)(mgr->createOverlayElement("Panel", "MyPanelName"));
```

Note: the static cast is necessary because createOverlayElement returns an OverlayElement* (which is the base-class for Panels, Text and Image-based elements. As long as you pass "Panel" as the first argument, this cast is safe.

You can specify dimensions for panels (and any other overlay element using pixel coordinates or normalized screen coordinates (0.0 – 1.0 in the x and y direction). If this panel or element is a child of another element, relative coordinates are relative to the parent's dimensions.

To add a panel to an overlay:

```
over->add2D(panel);
```

To create a TextElement, which is a part of a panel:

```
Ogre::TextAreaOverlayElement * text = (Ogre::TextAreaOverlayElement*)mgr->createOverlayElement("TextArea", "MyTextName");
```

And to add this text element to a panel:

```
panel->addChild(text);
```

To modify settings for any overlay element (text or panels):

```
element->setMetricMode(Ogre:GMM_PIXELS);      // Or GMM_RELATIVE
element->setPosition(x, y);                   // the interpretation is different if using pixels / relative
elements->setDiemsions(w, h);                 // ditto.
```

These methods only apply to panels:

```
element->setMaterial(material_name);          // More on materials later…
```

These methods only apply to text-areas:

```
element->setCaption(text);
element->setCharHeight(float amt);
element->setColourBottom(Ogre::ColourValue);
element->getColourTop(Ogre::ColourValue);
element->setFontName(std::string font_name);// Defined in a fontdef file (see
                                            // Media/packs/SdkTrays.zip/SdkTrays.fontdef for an example).
                                            // I used "SdkTrays/Value"
```

A side-note: The OverlayManager has a method to get the width and height of the viewport (in pixels) – you might find this useful if doing pixels-mode.

Note: There's a bug (I think) in Ogre: setting the dimensions of text doesn't seem to work until the window is rendered once. One fix is to call mRoot->renderOneFrame before constructing a GUI. The other is to update your text captions / colors / etc every frame.