Game Engine Architecture

Week 2 -Lab

Bootsctrap.cpp

- This first tutorial we covered the basic elements of building a scene in **Ogre**. The primary focus was the **Ogre::SceneManager**, **Ogre::SceneNode**, and Entity(external link). An **Ogre::Entity** is anything represented by a mesh. A **Ogre::SceneNode** is what attaches an object to your scene. Finally, the SceneManager is the object that organizes everything. It keeps track of the entities and nodes in your scene and determines how to display them.

SimpleSample.cpp

- Today we are going to cover lights, but if you remember, we added a simple light to the scene as a teaser. New Light objects can also be requested from the **Ogre::SceneManager**. We give the Light a unique name when it is created.
- Starting from **Ogre** 1.10 camera and lights require to create separate scene node for them so that they need to be attached to scene nodes.
- Once the Light is created and attached to its SceneNode, we set its position. The three parameters are the x, y, and z coordinates of the location we want to place the Light.

Template.cpp

- Next step is to create a camera.
- We are going to cover more about camera today.
- The next thing we did, is to ask the SceneManager to create an Entity.
- The parameter given to this function must be a mesh that was loaded by **Ogre**'s resource manager. For now, resource loading is one of the many things that **OgreBites::ApplicationContext** is taking care of for us.
- Now that we have an Entity, we need to create a SceneNode so the Entity can be displayed in our scene. Every SceneManager has a root node. That node has a method called createChildSceneNode that will return a new SceneNode attached to the root.
- We save the SceneNode pointer that is returned by the method so that we can attach our Entity to it.
- We now have a basic scene set up. Compile and run your application. You should see an **Ogre**'s head on your screen.

Adding Another Entity

- With our first Entity, we did not specify the location we wanted anywhere. Many of the functions in [Ogre] have default parameters. The SceneNode::createChildSceneNode(external link) method can take three parameters, but we called it with none.
- The parameters are the name, position, and rotation of the SceneNode being created.
- In older versions of [Ogre], you were required to provide a unique name for your Entities and SceneNodes. This is now optional. [Ogre] will generate names for them if you do not provide one.
- It also uses (0, 0, 0) as a default position.
- `SceneNode* ogreNode2 = scnMgr->getRootSceneNode()->createChildSceneNode(Vector3(84, 48, 0));`
- This is the same thing we did the first time, except we are now providing a Vector3 to our createChildSceneNode method.

addAnotherEntity

- The Ogre::Entity class is very extensive. We will now introduce just a few more of its methods that will be useful. The Entity class has setVisible and isVisible methods. If you want an Entity to be hidden, but you still need it later, then you can use this function instead of destroying the Entity and rebuilding it later.
- Remember that Entities do not need to be pooled like they are in some graphics engines. Only one copy of each mesh and texture is ever loaded into memory, so there is not a big savings from trying to minimize the number of Entities.
- The getName method returns the name of an Entity, and the getParentSceneNode method returns the SceneNode that the Entity is attached to. In our case, this would be the root SceneNode.

sceneNode

- You can set the position after creating the node with setPosition. This is still relative to its parent node. You can move an objective relative to its current position by using translate

moreEntities

- SceneNodes are used to set a lot more than just position. They also manage the scale and rotation of objects. You can set the scale of an object with setScale. And you can use yaw, pitch, and roll to set the object's orientation. You can use resetRotation to return the object to its default orientation.
- You can use numAttachedObjects to return the number of children attached to your node. You can use one of the many versions of **Ogre::SceneNode::getAttachedObject** to retrieve one of the SceneNode's children. The method detachObject can be used to remove a specific child node, and detachAllObjects can be used to remove all.

camera

```
// create the camera
    Camera* cam = scnMgr->createCamera("myCam");

    cam->setNearClipDistance(5); // specific to this sample
    cam->setAutoAspectRatio(true);
    camNode->attachObject(cam);
    //camNode->setPosition(0, 0, 140);
    camNode->setPosition(0, 0, 222);

    // Step1: It rotates the SceneNode so that its line of sight focuses on the vector you give it. It makes the Camera "look at" the point.
    camNode->lookAt(Vector3(0, 0, 0), Node::TransformSpace::TS_WORLD);
```

- You can retrieve the Camera by name using the SceneManager's createCamera method.
- Next, we will position the Camera and use a method called lookAt to set its direction using camNode.

- Next, we will position the Camera and use a method called lookAt to set its direction using camNode.
- The last thing we'll do (apart of attachning camera to a SceneNode) is set the near clipping distance to 5 units. This is the distance at which the Camera will no longer render any mesh. If you get very close to a mesh, this will sometimes cut the mesh and allow you to see inside of it. The alternative is filling the entire screen with a tiny, highly magnified piece of the mesh's texture. It's up to you what you want in your scene.

Viewports

- When dealing with multiple Cameras in a scene, the concept of a Viewport becomes very useful. We will touch on it now, because it will help you understand more about how **Ogre** decides which Camera to use when rendering a scene. **Ogre** makes it possible to have multiple SceneManagers running at the same time. It also allows you to break up the screen and use separate Cameras to render different views of a scene. This would allow the creation of things like splitscreens and minimaps.

- There are three constructs that are crucial to understanding how **Ogre** renders a scene: the Camera, the SceneManager, and the RenderWindow. We have not yet covered the RenderWindow. It basically represents the whole window we are rendering to. The SceneManager will create Cameras to view the scene, and then we tell the RenderWindow where to display each Camera's view. The way we tell the RenderWindow which area of the screen to use is by giving it a **Ogre::Viewport**. For many circumstances, we will simply create one Camera and create a Viewport which represents the whole screen.

Viewports

- Now let's set the background color of the Viewport.
- We usually set it to black because we are going to add colored lighting later, and we don't want the background color affecting how we see the lighting.
- The last thing we are going to do is set the aspect ratio of our Camera. If you are using something other than a standard full-window viewport, then failing to set this can result in a distorted scene.
- We will set it here forour demo even though we are using the default aspect ratio.

```
 // and tell it to render into the main window
    Viewport* vp = getRenderWindow()->addViewport(cam);
    //let's set the background color of the Viewport.
    vp->setBackgroundColour(ColourValue(1, 0, 0));
    vp->setDimensions(0, 0, 0.5, 0.5);
    cam->setAspectRatio(Real(vp->getActualWidth()) / Real(vp->getActualHeight()));
```

Mesh

- Before we get to shadows and lighting, let's add some elements to our scene. Let's put a ninja right in the middle of things.
- This setting simply allows you to turn on/off shadows for a given object.
- ninjaEntity->setCastShadows(true);
- We will also create something for the ninja to be standing on. We can use the **Ogre::MeshManager** to create meshes from scratch. We will use it to generate a textured plane to use as the ground.

Plane

- The first thing we'll do is create an abstract Plane object. This is not the mesh, it is more of a blueprint.
- We create a plane by supplying a vector that is normal to our plane and its distance from the origin. So we have created a plane that is perpendicular to the y-axis and zero units from the origin.
- Plane plane(Vector3::UNIT_Y, 0);
- There are other overloads of the Plane constructor that let us pass a second vector instead of a distance from the origin. This allows us to build any plane in 3D space we want.

MeshManager

- Now we'll ask the MeshManager to create us a mesh using our Plane blueprint. The MeshManager is already keeping track of the resources we loaded when initializing our application. On top of this, it can create new meshes for us.
- Basically, we've created a new mesh called "ground" with a size of 1500x1500.

MeshManager

MeshPtr Ogre::MeshManager::createPlane	(	const String &	name,
		const String &	groupName,
		const Plane &	plane,
		Real	width,
		Real	height,
		int	xsegments = 1,
		int	ysegments = 1,
		bool	normals = true,
		unsigned short	numTexCoordSets = 1,
		Real	uTile = 1.0f,
		Real	vTile = 1.0f,
		const Vector3 &	upVector = Vector3::UNIT_Y,
		HardwareBuffer::Usage	vertexBufferUsage = HardwareBuffer::HBU_STATIC_WRITE_ONLY,
		HardwareBuffer::Usage	indexBufferUsage = HardwareBuffer::HBU_STATIC_WRITE_ONLY,
		bool	vertexShadowBuffer = false,
		bool	indexShadowBuffer = false
	)		

name	The name to give the resulting mesh
groupName	The name of the resource group to assign the mesh to
plane	The orientation of the plane and distance from the origin
width	The width of the plane in world coordinates
height	The height of the plane in world coordinates
xsegments	The number of segments to the plane in the x direction
ysegments	The number of segments to the plane in the y direction
normals	If true, normals are created perpendicular to the plane
numTexCoordSets	The number of 2D texture coordinate sets created - by default the corners are created to be the corner of the texture.
uTile	The number of times the texture should be repeated in the u direction
vTile	The number of times the texture should be repeated in the v direction
upVector	The 'Up' direction of the plane texture coordinates.
vertexBufferUsage	The usage flag with which the vertex buffer for this plane will be created
indexBufferUsage	The usage flag with which the index buffer for this plane will be created
vertexShadowBuffer	If this flag is set to true, the vertex buffer will be created with a system memory shadow buffer, allowing you to read it back more efficiently than if it is in hardware
indexShadowBuffer	If this flag is set to true, the index buffer will be created with a system memory shadow buffer, allowing you to read it back more efficiently than if it is in hardware

setMaterialName

- Now we will create a new Entity using this mesh.
- We want to tell our SceneManager not to cast shadows from our ground Entity.
- It would just be a waste.
- Don't get confused, this means the ground won't cast a shadow, it doesn't mean we can't cast shadows on to the ground.
- And finally we need to give our ground a material. For now, it will be easiest to use a material from the script that Ogre includes with its samples.
- You should have these resources in your SDK or the source directory you downloaded to build Ogre.
- Make sure you add the texture for the material and the Examples.material script to your resource loading path. In our case, the texture is called 'rockwall.tga'. You can find the name yourself by reading the entry in the material script.

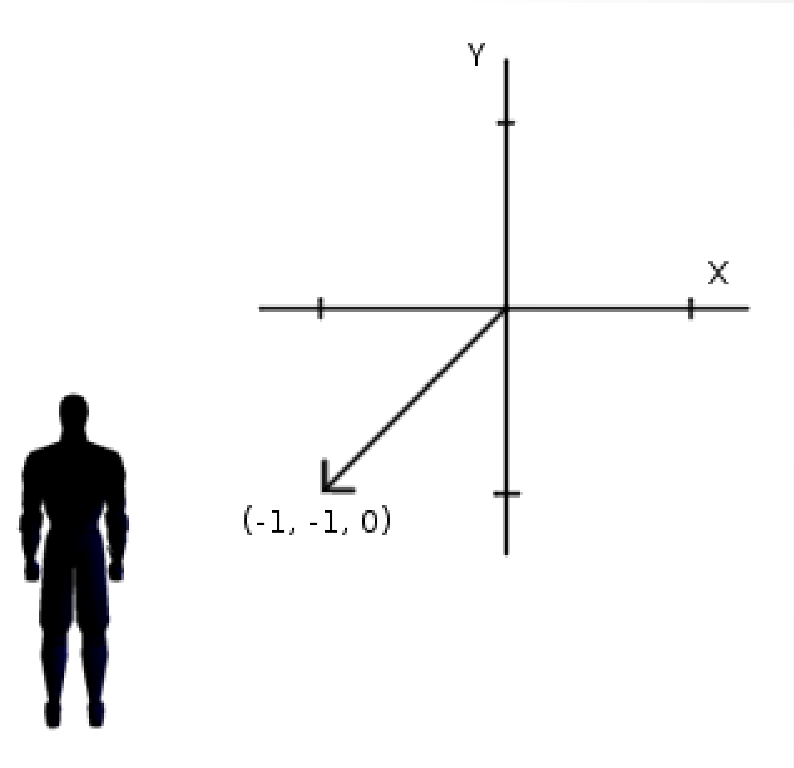
Shadow

- Enabling shadows in Ogre is easy. The SceneManager class has a Ogre::SceneManager::setShadowTechnique method we can use. Then whenever we create an Entity, we call Ogre::Entity::setCastShadows to choose which ones will cast shadows.
- setShadowTechinique method takes several of different techniques.
- Refer to Ogre::ShadowTechnique for more details.
- Let's turn off the ambient light so we can see the full effect of our lights. Add the following changes:

```
//scnMgr->setAmbientLight(ColourValue(0.5, 0.5, 0.5));
    scnMgr->setAmbientLight(ColourValue(0, 0, 0));
    scnMgr->setShadowTechnique(ShadowTechnique::SHADOWTYPE_STENCIL_ADDITIVE);
```

Creating a SpotLight

- Let's add a Light to our scene. We do this by calling the Ogre::SceneManager::createLight method. Add the following code right after we finish creating the groundEntity:
- We'll set the diffuse and specular colors to pure blue.
- Next we will set the type of the light to spotlight.
- The spotlight requires both a position and a direction - remember it acts like a flashlight. We'll place the spotlight above the right shoulder of the ninja shining down on him at a 45 degree angle.
- Finally, we set what is called the spotlight range. These are the angles that determine where the light fades from bright in the middle to dimmer on the outside edges.



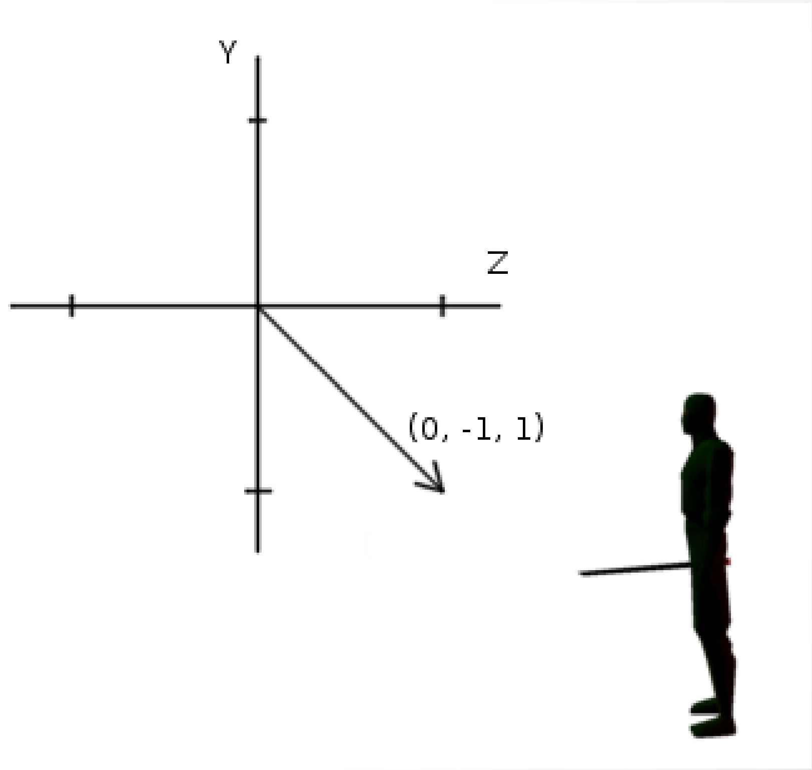
Spotlight

```
//! [newlight]
    Light* spotLight = scnMgr->createLight("SpotLight");
    spotLight->setDiffuseColour(0, 0, 1.0);
    spotLight->setSpecularColour(0, 0, 1.0);
    spotLight->setType(Light::LT_SPOTLIGHT);
    SceneNode* spotLightNode = scnMgr->getRootSceneNode()-
>createChildSceneNode();
    spotLightNode->attachObject(spotLight);
    //! [newlight]

    //! [lightpos]
    spotLightNode->setDirection(-1, -1, 0);
    spotLightNode->setPosition(Vector3(200, 200, 0));
    //Finally, we set what is called the spotlight range.These are the
angles that determine where the light fades from bright in the middle to
dimmer on the outside edges.
    spotLight->setSpotlightRange(Degree(35), Degree(50));
    //! [lightpos]
```

Directional Light

- Next we'll add a directional light to our scene. This type of light essentially simulates daylight or moonlight. The light is cast at the same angle across the entire scene equally. As before, we'll start by creating the Light and setting its type.
- Now we'll set the diffuse and specular colors to a dark red.
- Finally, we need to set the Light's direction. A directional light does not have a position because it is modeled as a point light that is infinitely far away.
- The Light class also defines a **Ogre::Light::setAttenuation** func tion which allows you to control how the light dissipates as you get farther away from it.



Directional Light

```
//! [directlight]
    Light* directionalLight = scnMgr-
>createLight("DirectionalLight");
    directionalLight->setType(Light::LT_DIRECTIONAL);
    //! [directlight]

    //! [directlightcolor]
    directionalLight->setDiffuseColour(ColourValue(0.4, 0, 0));
    directionalLight->setSpecularColour(ColourValue(0.4, 0, 0));
    //! [directlightcolor]

    //! [directlightdir]
    SceneNode* directionalLightNode = scnMgr->getRootSceneNode()-
>createChildSceneNode();
    directionalLightNode->attachObject(directionalLight);
    directionalLightNode->setDirection(Vector3(0, -1, 1));
    //! [directlightdir]
```

PointLight

- To complete the set, we will now add a point light to our scene.
- We'll set the the specular and diffuse colors to a dark gray.
- A point light has no direction. It only has a position. We will place our last light above and behind the ninja.

```
//! [pointlight]
    Light* pointLight = scnMgr-
>createLight("PointLight");
    pointLight->setType(Light::LT_POINT);
    //! [pointlight]

    //! [pointlightcolor]
    pointLight->setDiffuseColour(0.3, 0.3,
0.3);
    pointLight->setSpecularColour(0.3,
0.3, 0.3);
    //! [pointlightcolor]

    //! [pointlightpos]
    SceneNode* pointLightNode = scnMgr-
>getRootSceneNode()-
>createChildSceneNode();
    pointLightNode-
>attachObject(pointLight);
    pointLightNode->setPosition(Vector3(0,
150, 250));
    //! [pointlightpos]
```