MTAT.03.015 Computer Graphics (Fall 2013)
Exercise session XIV: OGRE

Ilya Kuzovkin, Konstantin Tretyakov

December 9, 2013

In this exercise session we will have a look at the high-level graphics engine OGRE[1]. We will see how the concepts we know about are included into the OGRE engine, making our life easier. Lighting, materials, shadows, environment mapping and other techniques are accessible by adding few lines of code, without the need to implement the annoying details on our own.

As before, the solutions will have to be submitted as a zipped project directory. Please, keep Windows libraries and project files in the archive, even if you work on Linux or Mac.

You can always seek for additional information and help in the OGRE tutorials [2] and reference pages[3].

Before proceeding, check out the `README.txt` file in the practice files archive for some important notes. In particular, if you work on Linux, you will need to install OGRE before proceeding (for Windows, all the necessary libraries are provided). In Debian-based systems this can be done using a command like the following:

```
$ sudo apt-get install libogre-1.8-dev libois-dev
```

1 Structure of the application

We start by comparing the structure of the application to the familiar structure we have been using so far. Open the project `1_OgreTriangle` and read through the code in the `triangle.cpp` file. Compare it to GLUT-based applications we have seen before.

Exercise 1 (0.5pt). Add a small square which will fly around the triangle and rotate around its own center. For that you will need to:

1. Create a new `Ogre::ManualObject` object using `createManualObject()` method of the scene manager.

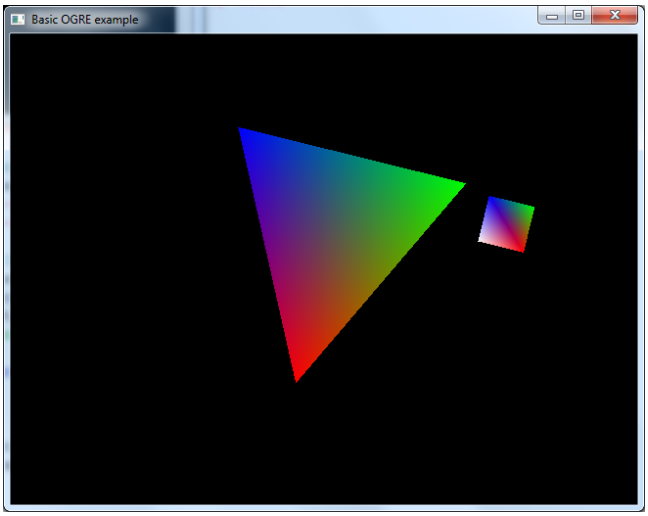
[1] `http://www.ogre3d.org/`
[2] `http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Tutorials`
[3] `http://www.ogre3d.org/docs/api/html/`

2. Describe the vertices of your square. Note that, in contrast to OpenGL, in OGRE we *first* write the `position` command to create the vertex itself, and then describe its attributes, such as color. The order in which the vertices are specified will depend on which type of `Ogre::RenderOperation`[4] (triangle list, triangle strip or triangle fan) you decide to use.

3. Create an `Ogre::SceneNode`, and attach the new object to it.

4. Use this `SceneNode` to animate the object (update its position) in the `frameRenderingQueued()` method (which is an analog of the `idleFunc()` we were using in GLUT).

The existing code for creating the triangle will serve you as an example. The result should look something like this:



2 Lighting, materials and textures

Open the project `2_OgreLighting`. The general structure is the same as before. Study the code in `createLitSphereScene()`. Here we configure our scene: create a sphere, add material specification to it, enable lighting and shadows.

In Exercise Session 9, which was about different types of shadows, we implemented three different shadow techniques. You might remember that for implementing *stencil shadows*, for example, you first needed to create shadow volume objects and after that implement the logic by carefully playing with stencil, color and depth buffers. OGRE allows you to enable shadows with literally one line of code[5]. This line is already written for you in the `createLitSphereScene()` function, but is commented. Find and uncomment it.

[4] `http://www.ogre3d.org/docs/api/html/classOgre_1_1RenderOperation.html`
[5] See more `http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Basic+Tutorial+2`

Now pay attention to these lines in the middle of the `run()` function:

```
Ogre::ResourceGroupManager::getSingleton().
                    addResourceLocation("../data", "FileSystem");
Ogre::ResourceGroupManager::getSingleton().
                    initialiseAllResourceGroups();
```

In reality graphics-heavy applications would often use many external *resource files*, to specify meshes, textures, materials, and other things. The first line tells OGRE where to look for the resources: data files (textures, meshes, etc.) and scripts. The second line instructs it to read in resource descriptions and initialise *resource groups*.

Have a look at the `Examples.material` file[6] in the `data` folder and try to apply some of materials described there to our sphere. For example, try

```
sphere->setMaterialName("Examples/WaterStream");
```

If you would like to try other example materials you should download the full OGRE SDK[7] and copy the resources (e.g. textures) needed for each particular example from the SDK to our `data` folder.

Exercise 2 (1pt). Create two new files `data/Sphere.material` and `data/Plane.material`. Those will be used to describe materials of our objects.

1. Use examples in `data/Examples.material` and the relevant OGRE manual pages[8] to create a material script for the sphere, which exactly reproduces material parameters we have specified in the code. The result should look exactly the same: red sphere with white specular spot on it.

Next, let us see how texture mapping is done within `.material` scripts.

2. Find material examples in `data/Examples.material`, which have a `texture_unit` section. Note that textures can also be animated (e.g. `Examples/WaterStream`).

3. Describe the material with a texture in the `Plane.material` file. Pick some favourite picture of yours for the texture.

Note that you can use `WASD` keys and the mouse to control the camera in this project. This is enabled by the invocation of the `controlCamera` utility method of the `SimpleMouseAndKeyboardListener` class, provided in `input_util.h`.

[6] This is a file from the OGRE SDK, used in many examples and tutorials.
[7] `http://www.ogre3d.org/download/sdk`
[8] `http://www.ogre3d.org/docs/manual/manual_16.html`

Exercise 3* (0.5pt). A realistic scene requires some background. A simple way to provide such background is by enabling a "sky box", which is simply a cube-mapped texture, applied to the background around the scene. To specify a cube-mapped texture you need 6 image files, which have common prefix `mytexture_` and six suffixes, one for each side of the cube: `bk`, `dn`, `fr`, `lf`, `rt`, `up`. See the images `stevecube_*.jpg` in our `data` folder. Have a look at the tutorial[9] and add a sky box to our scene. You can use the provided images or make up your own.

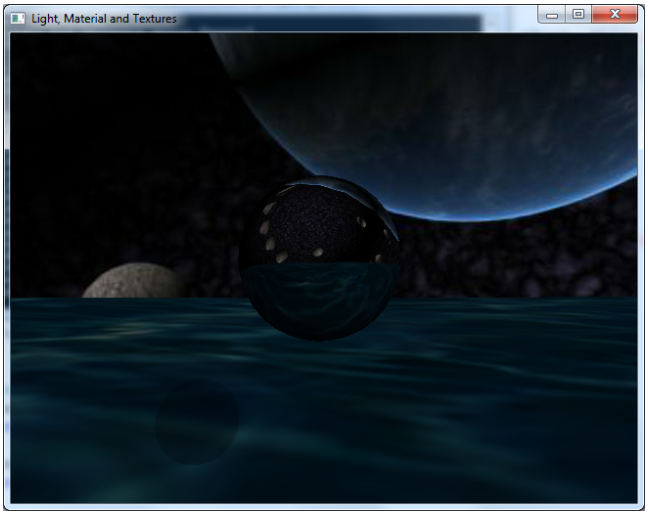
Exercise 4* (0.5pt). Look at the scene now. You might feel the urge to make the sphere reflective. The simplest way to do it would be to apply the cube map texture from the previous exercise as a `cubic_reflection` type of environment map to the sphere material. This would not be completely correct, however, as the reflection would not take into account the "floor" nor any other objects that might appear in the scene. To make proper reflections the cube map must be generated *dynamically* by actually rendering the scene from the point of view of the sphere. Study the code in `examples/CubeMapping/include/CubeMapping.h` and use it to add a dynamically changing cubic reflection map onto the sphere. Here is an approximate description of the steps you need to do to implement dynamical cube mapping:

1. Make our `Application` class inherit `Ogre::RenderTargetListener`
2. Make use of `preRenderTargetUpdate()` method
3. Add some global variables
4. Create cube map texture in the same way as it is done in `createCubeMap()`
5. Use `Examples/DynamicCubeMap` to create a new material in our `Sphere.material` script
6. Enable this material using `setMaterialName()` method

Note how we use an additional camera to update textures used for the cube. Since textures are updated on each frame you will get actual dynamic reflections.

[9] `http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Basic+Tutorial+3`

After completing exercises 2, 3 and 4 your scene might look something like this.



3 Meshes and character animation

Now let us study how OGRE works with meshes. Open the project `3_OgreMesh`. Meshes are handled as any other object we have seen before: they are entities, they are attached to a `SceneNode` and you can specify materials, textures and shadows in the exactly same way as we have seen before. Most importantly, however, a complex mesh can be loaded from a resource file. Have a look at `createOgreScene()` routine to see how mesh is loaded and attached to the scene.

Exercise 5 (0.5pt). Let us try to animate the mesh using skeletal animation. In the `data/Character` directory (which we use as resource location for this project) among other things we have a `Sinbad.skeleton` file. It describes the armature (set of bones) for our character and 13 different keyframe animations for that armature[10]. The animations are named `IdleBase`, `IdleTop`, `RunBase`, `RunTop`, `HandsClosed`, `HandsRelaxed`, `DrawSwords`, `SliceVertical`, `SliceHorizontal`, `Dance`, `JumpStart`, `JumpLoop`, `JumpEnd`. Let us use the `RunBase` animation to move the mesh.

1. Create a variable that will be used to keep track of the animation

```
AnimationState* mRunningAnimation;
```

[10] `http://www.ogre3d.org/docs/manual/manual_75.html`

2. In the `createOgreScene()` routine request pointer to the animation sequence named `"RunBase"`

```
mRunningAnimation = mBodyEnt->getAnimationState("RunBase");
```

3. Control phase of the animation with `addTime()` method of `AnimationState`. Use it in the `animateObjects()` routine.

4. Use `setEnable()` method to enable and disable the animation in `keyPressed()` and `keyReleased()` routines.

In total this animation can be configured with 4-5 lines of code (not counting a few `if` statements, which you will probably need).

Exercise 6* (0.5pt). Next let us make our character move when we press `WASD` keys. Do the following:

1. Add new `Vector3` variable, which will be used to store current direction where the ogre is facing.
2. Update this vector in the `keyPressed` and `keyReleased` functions according to the user input.
3. Update character position in `animateObjects`.

We would also like our character to turn his face to the same direction where he is moving. One way to do that is:

1. Use `getOrientation().zAxis().angleBetween()` of object's `SceneNode` to calculate angle between current object orientation and requested direction.
2. Rotate the object using that angle.

Congratulations! You now have a running ogre at your disposal.

The code you see in this project is actually a simplified version of an OGRE `Character` sample application. You can find the full version with all 13 animations, smooth transitions between animations and floating camera in the OGRE SDK[11].

Complex meshes like the one we use here, their skeletons and animations can be created with modeling tools like Blender and then exported to OGRE format using exporters[12] provided by OGRE. The "Sinbad" character was created in Blender, so you can open `data/Character/Sinbad.blend` and do with it whatever you want[13]. To access the list of animations open "Dope Sheet" in one of the panels, choose "Action Editor"[14] and then you will see a drop-down list with the available animations.

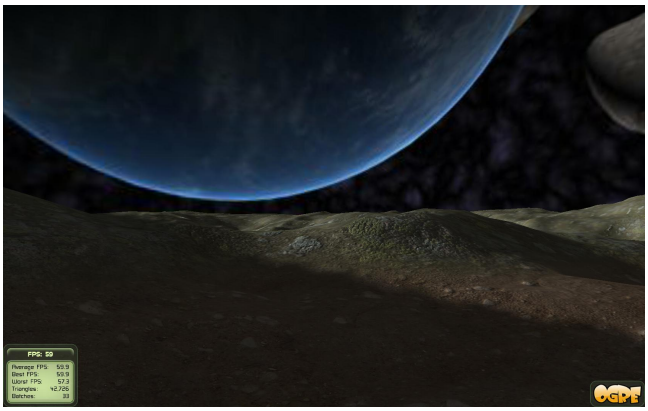
Exercise 7* (2pt). This bonus exercise offers you a variety of interesting things you can do. Implement all of them, any of them or any combination of them. Depending on the amount of effort you can get 1-2 bonus points.

More animations
Study the code in OGRE SDK `Samples/Character` and add more animations to our application: idle animation, jump, smooth transitions, sword movements, etc. Add camera behaviour if you like. This exercise relies heavily on your coding skills, if you are friends with C++, this exercise should be mostly copy-pasting to correct places.

Export from Blender
Create a simple custom mesh with an armature and an animation in Blender (e.g. use the one you made in the previous practice session), export it to OGRE using the BlenderExporter[15] and use it in the application.

Terrain
In OGRE official Basic Tutorial 3: "Terrain, Sky, and Fog"[16] you can learn how to create terrains, sky (as we already did in Exercise 3) and fog resulting in a scene like this:

[11] Check out the file `Samples/Character/include/SinbadCharacterController.h`
[12] `http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Blender+Exporter`
[13] The "Sinbad" model is licensed under CC-BY-SA by Zi Ye, i.e. you can use it freely given that you give appropriate credit. Check out the appropriate README file.
[14] `http://wiki.blender.org/index.php/Doc:2.6/Manual/Animation/Editors/DopeSheet/Action`
[15] `http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Blender+2.5+Exporter`
[16] `http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Basic+Tutorial+3`



Take your animated character and place him into such a world. Choose details to your own liking: terrain, sky, light, shadows, textures, reflections, army of ogres, other creatures[17] and perhaps a mysterious reflective torus knot in the middle of everything...

4 Particle systems

Specialized functionality in OGRE is implemented via separate modules called "plugins". This way you are free to only use the parts of the library that you need in your project. Open project `4_OgreParticles`. Have a look at the following line in the very beginning of the `run()` function:

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

It tells OGRE the name of a file in `bin` directory (or whatever directory your app is launched from), listing which plugins must be loaded. Inside that file you will see that a plugin called ParticleFX is enabled. This plugin lets you add *particle system* objects to your scene. A particle system is typically described using a *particle script* in the resource files, which resembles `.material` files we have seen before. Open `data/Examples.particle` and study it.

Exercise 8 (0.5pt). To complete this exercise do the following:

1. Make particle source follow mouse movements. For that you will need to add mouse listener in the way analogous to the way how keyboard listener is added. See comments in the code.

2. Play with different parameters of the particle system description. Modify the particle script so that the particles would fly out of the source in random directions, somewhat like fireworks or Bengal lights.

[17] Look inside `media/models` folder of OGRE SDK for more meshes and characters