Computer Graphics Lab 4

Assistant Prof. Georgios Papaioannou

Teaching Assistant Konstantinos Vardis

Dept. of Informatics, Athens University of Economics and Business

Introduction

This lab introduces an object loader and scene management using a scene graph structure

Getting started

1. Start Visual Studio and load the solution file (.sln). You should now have a window on the right (or left) hand side of your display that indicates that solution ‘lab1’ has loaded.
2. In the Solution Explorer (on your right or left) locate the Headers and Source filter. The first one contains the file Renderer.h which simply contains the definitions of the functions of Renderer.cpp we wish to expose. The Source filter contains the Main.cpp which is the main file of the application and the Renderer.cpp which contains the implementations of basic OpenGL functions. Double clicking on these files will display their contents the source code editor window.
3. The Shaders filter contains all the shader files necessary for each lab.
4. The ShaderGLSL class provides basic shader functionality (loading and compiling shaders)
5. The HelpLib files contain basic functions and includes in order to make your life easier :).
6. The SceneGraph filter contain the scene graph files
7. The solution produces both 32-bit and 64-bit executables.
8. **Make yourself comfortable with the code!**

Changes From Last Lab

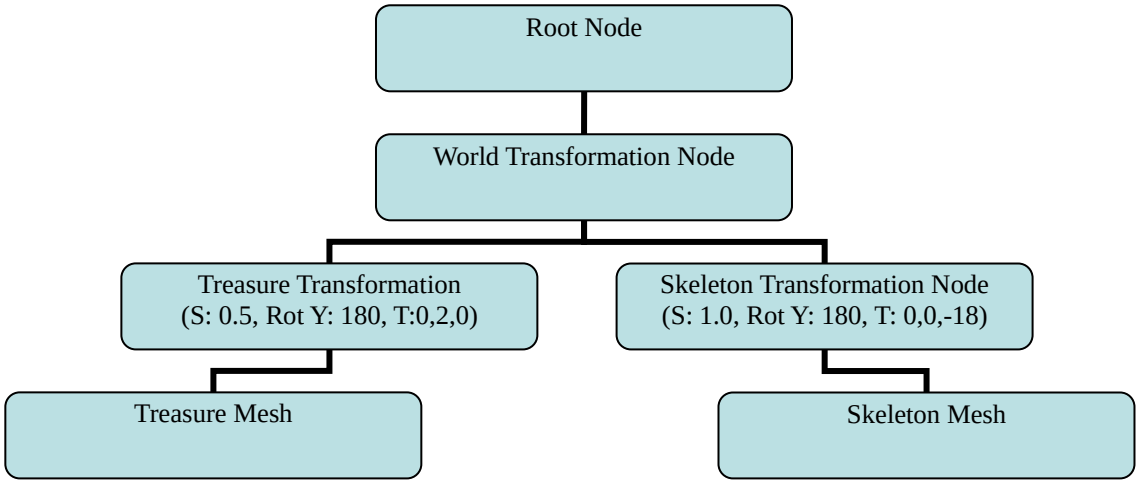
- ObjLoader, ObjMaterial and OGLMesh classes. The first class parses the obj file, the second class is a data structure for holding material information and the third class passes the loaded information in the GPU (compatible with OpenGL 3.3)
- The SceneGraph filter includes several classes for the scene graph. The main is the Root node (all nodes are children of the root node), the Node is the basic node class, the GroupNode is the class that holds one or more children nodes, the TransformNode is the basic transformation group node and the GeometryNode renders an OGLMesh.
- A BasicLighting shader is included which creates a basic lighting effect, otherwise the models will look flat and unrealistic.
- The Data folder contains the obj models and their textures

Key Input

- **ESC**: exit the application
- **F1**: change between rendering modes (fill, wireframe and point rendering)
- **W/S**: camera eye Z translation
- **A/S**: camera eye X translation
- **R/F**: camera eye Y translation
- **UP/DOWN**: world Z translation
- **LEFT/RIGHT**: world X translation
- **PGUP/PGDOWN**: world Y translation

Exercises (Follow these in the order given!!!!!)

1. Look at the code and familiarize yourself with the changes.
2. Uncomment the **SceneGraphExampleInit** and **SceneGraphExampleDraw** functions and run the projects. You will see a familiar scene of the planetary system. Look at how the graph is constructed and how it is rendered. Add a third planet which rotates around planet 2 on the Y axis. Note that in order to apply a certain scale, you have to undo all previous scaling transformations.
3. The OGLMesh class uses the BasicGeometry shader. Change it to use the BasicLighting shader (look at the **OGLMesh::init** function). Change it back to BasicGeometry shader.
4. Create a second scene using the scenegraph (create different Init and Draw functions). The structure should be like this:

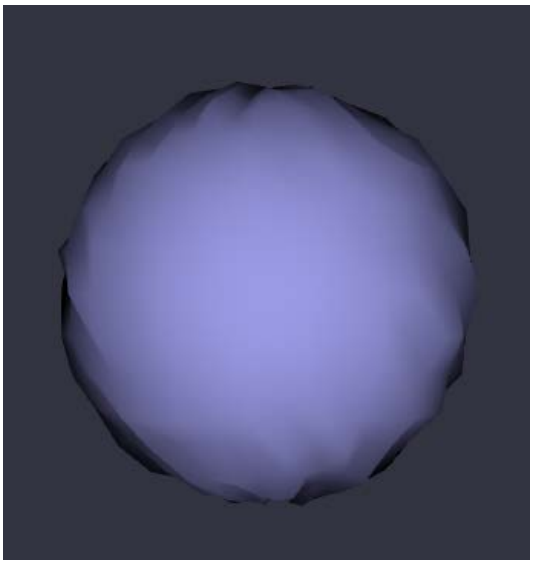


The camera should be altered to fit the scene. Use something like eye: 80.0f, 60.0f, 100.0f.

The obj files for the treasure and the skeleton are located in the Pirates folder. You need to load them as well. Look at how the loading works in the **LoadObjModels** function.

Once you load the scene, change the shader back to BasicLighting. The objects look three dimensional now.

5. A small exercise to test the power of shaders. What we want to do is get the static sphere and slightly modify the vertex positions in the vertex shader to give it a more rough look. Something like this:



We can do this just by playing with the vertex shader! The idea is:

- Use the BasicLighting shader (so that the result will still look shaded)
- Get the incoming vertex position in the vertex shader (we already have this)
- Modify the vertex position in the direction of the normal by a random offset so that the new position is the current position plus the normal multiplied by the offset (position_extruded = position + normal * offset)
- Use this new position to create the gl_Position

A very simple random function is one that accepts a vec3 and returns a number between -0.05 and 0.05. This function should be in the vertex shader (before main)

```
// get a random number between -0.05,0.05
float rand(vec3 n)
{
// get a random number between 0,1
float random_number = fract(dot(n.xyz, vec3(17.532, 59.124, 67.354) *
12345.893));
// get it in the range -0.05,0.05
return 0.05 * (2.0 * random_number - 1.0);
}
```

The variable that we can pass to this function to get a random number for each vertex should be something that also changes per vertex. The normal is a good candidate for this.

6. This can be further improved. The object has now a rough surface, but we can also animate it based on a time parameter. This time parameter should give a value between -1,1 and can be passed as a uniform in the vertex shader (using glUniform1f).

   The steps are:

   On the C++ side:

   - Declare a uniform for the BasicLighting shader. The uniforms are created in the OGLMesh.cpp in the init function.
   - During rendering, get a time value which returns the time that has passed since the application has started in milliseconds. Use glutGet(GLUT_ELAPSED_TIME) for this.
   - To convert it to a range -1,1 we first round it in a 0-360 offset (so that it represents an angle) and then use it as an argument in the sine function (in the vertex shader) that returns values in the range between -1,1. Something like this:

     ```
     // pass the timer parameter
     float timer_elapsed = glutGet(GLUT_ELAPSED_TIME) * 0.05f;
     // get it to round between 0 and 360
     float timer_elapsed_360 = int(timer_elapsed) % 360;
     // convert it to radians
     timer_elapsed_360 = glm::radians(timer_elapsed);
     ```

   - You can do this in the Draw function in the GeometryNode. Once you get the time, pass it to the shaders using glUniform1f.

   On the GPU side:

   - Declare the uniform you created in the vertex shader
   - Multiply the random offset you already get from the rand function by the uniform time parameter.