Computer Graphics Lab 2

Winter Semester 2015/2016

Date: 5/11/2015

Assistant Prof. Georgios Papaioannou

Teaching Assistant Konstantinos Vardis

Dept. of Informatics, Athens University of Economics and Business

Introduction

This lab introduces concepts regarding coordinate spaces and transformations, how to write a shader program and the OpenGL mathematics library (GLM)

NOTE: glm official webpage is http://glm.g-truc.net/.

The manual is also located in your documents folder in e-class.

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 solution produces both 32-bit and 64-bit executables.
7. **Make yourself comfortable with the code!**

Changes From Last Lab

- The Resize function contains now the viewport and projection transformations.
- The Render function shows how basic world and camera transformations are calculated.
- The InitializeRenderer functions shows how to load shader files and the Render*(Triangle, Quad, Planets) function show how to pass uniform variables to the shader.
- There is an idle function that is being called when no events occur. This function calls glutPostRedisplay to render again. So rendering never stops.

- There is a Mouse and a MouseMotion function for handling mouse events. Mouse is called upon mouse button press and MouseMotion upon mouse movement.

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

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

1. Look at the code and familiarize yourself with the changes.
2. Run the program. You should see the triangle from the last lab. We are now rendering the triangle in the whole window (of size 500 x 500 pixels). Look at the **Resize** function. Each time the window is resized, this function receives the new width and height and sets the viewport and projection transformations accordingly. Try to resize the window and see how this affects the viewing window.

   Change the Viewport Transformation (**glViewport**) to fill a different part of the screen (from XY=0 to 300 pixels) (0, 0, 300, 300). See how this affects rendering.

   Change:

   ```
   glViewport(0, 0, width, height);
   ```

   To:

   ```
   // set fixed parameters for width, height
   width = 300;
   height = 300;
   glViewport(0, 0, width, height);
   ```

   Ok, now change it back ☺. We want to render in the whole window.
3. Look at the **Render** function. We have set up a camera using glm::lookat and a perspective projection. Camera is located at (0,0,50) and target at the origin (0,0,0). Use W,S,A,D,R,F keys to move the camera eye and the left mouse click to move the camera target. This is the most basic camera system (and quite unintuitive). Change the camera parameters to see how the world is viewed from different directions. **NOTE: At the end of this exercise, it would be advisable to revert any camera changes you made, in order to proceed to the following steps.**
4. Look at the **DrawTriangle** and **DrawQuad** functions. For each object, a set of transformations is applied to place them on the screen. Change the object_to_world matrices. For example, rotate the triangle on the Z axis by 35 degrees, translate the quad by 2 units on the Y axis, scale the quad uniformly by 50%, etc. Check this before moving onto more complex transformations.
5. Now, let's leave the boring 2D shapes and look at a more complex transformation example. A planet with two satellites. Run the **DrawPlanets** function (yes the colors are boring ☹, but

we have still a long way to go before we apply materials, textures and lighting). Each planet is positioned differently around the world. Ok, now take it step by step. Do NOT proceed to the next steps, until you understand what is happening.

- Only Planet1 is uncommented so check this code first. Then, run the project. A planet is at the center. Press F1 to enter wireframe mode and see how it rotates around itself.
- Uncomment the part for Planet 2. Read how the transformations are applied. Run the project again (wireframe mode). A planet is positioned at the top right and rotates around itself.
- Uncomment the part for Planet 3. Read how the transformations are applied. Run the project again (wireframe mode). A planet is orbiting around planet 1.
- Uncomment the part for Planet 4. This is more complex. Read how the transformations are applied. Run the project again (wireframe mode). A planet is orbiting around planet 3 who is orbiting around planet 1.
- Add one more planet orbiting around Planet 1.
- Note that there is a **parent_transform** applied right before we move to the eye space coordinate system. If you look at the **Render** function, this **parent_transform** is defined as the identity matrix. Then, it is being applied hierarchically. For example, in the planetary system, Planet1 affects Planets 2 and 3. The way it affects them is specified in the **parent_transform** for each planet right before **DrawPlanet1** and **DrawPlanet2** are called. Since this planetary system is imaginary, the way these planets are affected by each other, is entirely fictional.
- The **parent_transform** is initially the identity matrix. If you change it to – for example – rotate around the Y axis, this will affect ALL the transformations in the scene (quads, triangles, planets). Use the variable **world_rotate_y** that is changing whenever the right button click is pressed (look at **MouseMotion** function). Modify the **parent_transform** to rotate the world on the Y axis using the **world_rotate_y** variable. Use **glm::rotate** for this. If you do it correctly, then you should be able to rotate the whole world each time you press the right click mouse button and move the mouse horizontally.

6. Now let's play with shaders. First, look at the **BasicGeometry** shaders. Again, spend some time to understand what is happening. The vertex shader is called for each vertex and the fragment shader for each fragment.
7. As you have seen, all planets are rendered with the same gray color. We can change the output color, regardless of what color values we pass at each vertex, by changing the outcome of the fragment shader. Print a yellow color for all spheres.

   Change:

   ```
   out_color = vertex_color;
   ```

   To:

   ```
   // yellow color
   vec4 yellow_color = vec4(1.0, 1.0, 0.0, 1.0);
   out_color = yellow_color;
   ```

8. Of course, we can do to fragments (pixels) whatever we want. For example, we can change the final color based on the vertical coordinates of the window. We can use **mix** for this

which does linear interpolation based on two values. Example:

```
// glFragCoord returns the relative coordinates of the window. (0, 0 bottom
// left -> width, height top right)
// get a value 0->1 based on the vertical coordinate (500, 500 is the size of
// the current window)
float lerpValue = gl_FragCoord.y / 500.0f;
// use mix to get the final color such as: color = mix(value_at_0, value_at_1,
// value_to_interpolate);
out_color = mix(vec4(1.0f, 0.0f, 0.0f, 1.0f), vec4(0.0f, 0.0f, 1.0f, 1.0f),
lerpValue);
```

9. All objects so far are rendered using the BasicGeometry shader. Create a new shader (BasicGeometry2) that affects the rendering of Planets 1 and 2. The BasicGeometry2 shader should output the color green for each fragment. To do this:
    - Create two new shader files (copy - paste the BasicGeometry.vert and .frag) and add them to the project.
    - Look at the shader creation and replicate it for the new variables which will hold the new shader object and uniforms.
    - Modify the shader code to write green color for each fragment
    - Change the shader that render Planets 1 and 2 (look at the **glUseProgram** function)