Computer Graphics Lab 3

Assistant Prof. Georgios Papaioannou

Teaching Assistant Konstantinos Vardis

Dept. of Informatics, Athens University of Economics and Business

Introduction

This lab introduces vertex array objects, vertex buffer objects and rendering using basic primitive types

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!**

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. Uncomment the **DrawPrimitives** function and run the project. You should see some 2D elements such as points, lines and triangles. The triangle is built in the **BuildTriangleVAO** function, the circle in **BuildTriangleFanVAO** and the rest in **BuildQuadsVAO**. The quads are rendered using triangle strips (without an index buffer) and triangles (using an index buffer). Look at the **Build** functions to see how the VAOs are being constructed and at the **Draw** functions to see how they are rendered. The draw code is similar, the only things that change are the VAO objects and the glDraw* calls.
3. Note that pressing the right mouse button and moving horizontally you perform a world rotation on the X axis. Rotate more than 90 degrees and see that the 2D objects are also visible. This is because back-face culling is disabled. Alternatively, press W to move in the camera Z direction until you pass through the objects, so that you look at them from the back side.
4. Enable the part that changes culling in the **Render** function. Run the project again and rotate more than 90 degrees. You should see that the back faces of the triangle primitives are not rendered.
5. Now change glCullFace to GL_FRONT and run the code. Now front faces are culled. Now change it back to GL_BACK.
6. Now change glFrontFace command to GL_CW and run again to see how it affects rendering. Now change it back to GL_CCW. This way you can control which primitives are being shown.
7. Comment the **DrawPrimitives** function and uncomment the **DrawPlanets** function. Also, set the variable **uniform_color** in the fragment shader to be the output color of the fragment shader (instead of the vertex color). Run the project. Look how the planets behave when the blue planet goes behind the red planet. Enable depth testing and run the project again.
8. **Exercise**: Do some basic lighting. You do not need to know yet why the lighting calculations work, but you need to modify the VAO constructions in order to get them to work. The basic idea is this. We can place a "fake light" in the location of the camera looking along the negative z axis in eye space, think of it as a miner's light. The lighting calculation is simply a dot product between the ECS normal of the object and the ECS direction of the camera in the fragment shader. Modulate the final color by the result of the dot product.

   The steps for completing this exercise are:

   - Create a new VAO for the sphere where the positions are stored in a vertex structure similar to the **VertexStruct** struct. This struct should hold a vec3 for the position and a vec3 for the normal. You can use the same logic of the **BuildSphereVAO** to generate the vertices (just store them in a **VertexStruct** object instead). The normal for the sphere is the normalized position. Example for the top left vertex:

     ```
     SphereVertexStruct top_left;
     top_left.position.x = sphere_radius * sintheta * cosphi;
     top_left.position.z = sphere_radius * sintheta * sinphi;
     top_left.position.y = sphere_radius * costheta;
     top_left.normal = glm::normalize(top_left.position);
     ```

   - Create a new shader called **BasicLighting** similar to the **BasicGeometry** where the vertex input is the position and the normal of the **SphereVertexStruct.** Each vertex shader should output besides the **gl_Position**, the normal (in OCS) to the fragment shader.

- Besides the uniforms for the world_view_projection matrix and the color of the sphere you also need to pass the normal transformation matrix which will transform the normals from OCS->ECS. The transformation matrix for the normals is the inverse of the transpose of the world_view matrix:

```
// the variable to pass to the uniform_mv variable
glm::mat4x4 normal_world_view_matrix =
glm::transpose(glm::inverse(world_to_camera_matrix * object_to_world_matrix));
```

- In the fragment shader, transform the incoming normal from object space to eye space using the above matrix and do a dot product between this and the direction of the camera. The fragment shader code should be like this:

```
// uniform for color and for worldview matrix (OCS->ECS)
uniform vec4 uniform_color;
uniform mat4 uniform_mv;
// incoming normal in OCS from vertex shader
in vec3 normal_ocs;

void main(void) {
// Transform normal from OCS to ECS (note the 0 in the w coordinate)
vec4 normal_ecs = uniform_mv * vec4(normal_ocs, 0);
// Normalize it
normal_ecs.xyz = normalize(normal_ecs.xyz);
// Calculate dot product between 0,0,-1 and normal
float light = max(0.0, dot(normal_ecs.xyz, vec3(0,0,-1)));
// Multiply this with the final color
out_color = uniform_color * light;
}
```

Note: These lighting calculations are not entirely correct, but since we have not done any lighting yet, it is enough for now.

- The final result should look like this: