Computer Graphics Lab 6

Assistant Prof. Georgios Papaioannou

Teaching Assistant Konstantinos Vardis

Dept. of Informatics, Athens University of Economics and Business

Introduction

This lab introduces texture filtering, wrapping, sampling and alpha testing

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

- The vertex and fragment shaders have been modified to pass the uv coordinates to the fragment shader and to sample a texture for texture mapping.
- The OBJ loading files have been moved to the OBJ filter in the Visual Studio Solution. Their location in the hard drive has not been changed.
- A Texture and a TGA class are included. The TGA class is responsible for the loading of tga textures from a physical location. The Texture class retrieves the loaded TGA file and loads the texture to the GPU. The function you are mostly interested in, is the **GenerateTexture** where you can directly modify the texture filtering that is applied later on.
- The SceneGraph has been further modified so that it can allow rendering geometry the BasicGeometry shader as well. Passing 2 in the root->draw function renders the geometry with no lighting applied. This is useful for placing lights and for the texturing example.
- The eye and light settings have been moved to each scene's init function.

- Since the **SceneExampleDraw**, **SceneExample2Draw** functions are exactly the same (the transformations are static), you need to change only the corresponding **SceneExampleInit** function to change scenes. **SceneExampleInit** loads the spheres and **SceneExample2Init** loads the pirate scene.

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 **TextureFilteringExampleInit** and the **TextureFilteringExampleDraw** functions and run the project. You will see 3 spheres with different textures and three quads. The noisy textured quad and the "OpenGL" quad top quad have their uv coordinates applied in the range 0-1. The "red texture" quad has its uv coordinates in the range 0-2. Since we are exceeding the 0-1 range you see that that texture is repeated.
3. Change the texture clamping so that parameter values outside the 0-1 range are clamped instead of repeated. This is in located in the **GenerateTexture** function in the Texture class. Change GL_REPEAT for S and T coordinates to GL_CLAMP_TO_BORDER. This clamps the coordinates outside the 0-1 range to a default border color which is black.
4. Change the clamping to GL_CLAMP_TO_EDGE. This clamps the coordinates outside the 0-1 range to the color at the edge of the texture.
5. Move **VERY** close to the top textured quad (by using the UP arrow key) so that the size of the quad is larger than the screen. In this case, each pixel is covered by more than one texel and the texture magnification filter is applied. Currently this is set to GL_NEAREST which chooses the value of the nearest texel for each sample (center of the pixel). You see that this results in bad quality. Change the magnification filter (GL_TEXTURE_MAG_FILTER) to GL_LINEAR from GL_NEAREST. GL_LINEAR performs bilinear interpolation based on the four neighboring samples. Do the same operation again (move close to the quads). The result is a bit better.
6. Now move **FAR** from the quads (by using the DOWN arrow key). In this case each pixel covers more than one texels and the minification filter is applied. Currently this is set to GL_NEAREST. Again, the quality is bad (the texture flickers a lot). Change the texture minification filter (GL_TEXTURE_MAG_FILTER) with no mipmaps from GL_NEAREST

to GL_LINEAR. Do the same operation again (move far from the quads). The result is a bit better.

7. A better minification filter is by enabling mipmaps. You can enable them in the **LoadOBJModels** function by setting the last parameter to true for the texturing meshes. Currently the GL_NEAREST_MIPMAP_NEAREST filter is used which chooses the mipmap which most closely matches the size of the pixel and then use nearest filtering for getting the texture value. Change this to the next best GL_LINEAR_MIPMAP_NEAREST. This chooses the mipmap that is nearest and then bilinear interpolation for the texture value.
8. Change the filter to GL_LINEAR_MIPMAP_LINEAR. This performs trilinear interpolation (both for choosing a mipmap and selecting a texture value).
9. The “OpenGL” quad has a texture with an alpha channel. The alpha value is set to 0 for the text “OpenGL”. Currently, alpha testing is disabled. By disabling rendering in alpha values less than 1, the OpenGL text will not be rendered. You can enable alpha testing in the BasicGeometry fragment shader. You will see that instead of the black “OpenGL” text you see the gray background.
10. You can directly manipulate the texture coordinates in the vertex shader. Comment out the **TextureFilteringExampleInit** and the **TextureFilteringExampleDraw** functions and uncomment th**e SceneGraphExampleInit** an**d SceneGraphExampleDraw**. Three textured spheres are rendered using an omnidirectional light. Look at the OmniLight.vert vertex shader at the part where the texture coordinates are passed to the fragment shader. Add a small offset (such as texcoord = texcoord0 + vec2(0.2, 0.2)). The textures have an offset of 0.2 in each s,t coordinate.
11. Similarly you can change the texture sampling location in the OmniLight fragment shader. Change the texture sampling location from texcoord.xy to texcoord.xy + vec2(0.2, 0.2).
12. Comment out **SceneGraphExampleInit** and uncomment th**e SceneGraphExample2Init** to see the effects in the classic pirate scene. You will also need to uncomment the scene 2 obj files in the **LoadObjModels** function.
13. It is easy to load any obj file and create a scene with a basic light. Comment out **SceneGraphExample2Init** and uncomment th**e SceneGraphExample3Init** to see the Knossos model. You will also need to uncomment the scene 3 obj files in the **LoadObjModels** function. Make sure that you have set GL_REPEAT as texture wrapping for this scene (change it to GL_CLAMP_TO_BORDER or GL_CLAMP_TO_EDGE if you are wondering why). All scenes look best if mipmapping and linear magnification filter are enabled. **Note: Since this scene loads a heavy obj file, loading will be faster if you run the project in Release instead of Debug mode.**