Computer Graphics Lab 7

Assistant Prof. Georgios Papaioannou

Teaching Assistant Konstantinos Vardis

Dept. of Informatics, Athens University of Economics and Business

Introduction

This lab introduces frame buffer objects and basic shadow mapping

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 directional and omni lights have been removed. Instead, a basic spotlight shader has been created to show how a basic spotlight is rendered with shadows.
- A shadow map shader has been created which renders the scene as viewed from the light.
- The root->draw now accepts 0 for rendering with the spotlight shader and 1 for rendering with the shadow map shader
- The **CreateShadowFBO** function creates a basic framebuffer object and a shadow texture which will store the shadow map of the spotlight. This is called in the **InitializeRenderer** function.
- The Render function has changed. A first pass renders the scene to the shadow map FBO in **DrawSceneToShadowFBO** and the second pass renders the scene as usual.

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. Run the project. The first scene is shown with shadows. This blockiness is the shadow texels projected on the geometry. They are visible because the shadow map resolution is too low. Change the shadow map resolution to 256, 512, 1024 respectively. In the end, keep it to 512 or 1024. This is located in the **CreateShadowFBO** function.
3. Modify the SpotLight fragment shader to support blurred shadows using Percentage Closer Filtering (PCF). PCF works by taking samples in the neighborhood of a point, testing these whether they are in shadow and averaging the result. To implement a basic PCF approach do the following:
    - First, implement PCF in the **is_point_in_shadow_pcf_5** function. Instead of doing a simple **if** check to test if a point is in shadow, you will need to do the same test for the central pixel (the one you are already checking in the **is_point_in_shadow**, the position_lcs) and 4 more samples, 2 horizontal and 2 vertical.

	X	
X	**X**	**X**
	X	

    - For each extra sample sample step (the green ones) you need to define a sample step. This step denotes how far from the central texel (the black one) you will move. Try two texels initially. One texel in uv space is 1.0 / texture_size. So, two texels is 2.0 / texture_size. Use **textureSize (uniform_sampler_shadow_map, 0)** to get the texture size in GLSL. The value of 0 is for the lod level. We have no mipmaps, so this is 0. (**Note: This is already implemented, but you can change the step once you are done**)
    - Change the constant bias to different values (constant_depth_bias variable in the **is_point_in_shadow** function). Try 0 initially, then increase the values up to 0.001. In the end, leave it to 0.0005 as before.

- The algorithm then (in pseudocode) is shown below:

```
define a shadow step s (it is already there)
define a variable total_sum to hold the final visibility value: total_sum = 0

// center
sample the shadow map at the central pixel
if it is not in shadow add 1 to total_sum
create the sample position for the top pixel

// top
top_pixel = central_pixel + (0, step)
sample the shadow map at the top pixel
if it is not in shadow add 1 to total_sum

// bottom
bottom_pixel = central_pixel + (0, -step)
sample the shadow map at the bottom pixel
if it is not in shadow add 1 to total_sum

// left
left_pixel = central_pixel + (-step, 0)
sample the shadow map at the left pixel
if it is not in shadow add 1 to total_sum

// right
right_pixel = central_pixel + (step, 0)
sample the shadow map at the right pixel
if it is not in shadow add 1 to total_sum

divide total_sum by 5 to normalize the result

return total_sum
```

- Set the shadow_factor in **main** to get the result of the **is_point_in_shadow_pcf_5** function.
- In a similar fashion, implement the **is_point_in_shadow_pcf_9** function. The only difference is that you need to sample horizontally, vertically and diagonally (green and red X's) and divide by 9 instead.

X	X	X
X	X	X
X	X	X

- Set the shadow_factor in **main** to get the result of the **is_point_in_shadow_pcf_9** function.
- Change the texel offsets and the shadow sizes to see how PCF modifies the shadow rendering. You can also uncomment the part that rotates the light in **Render** and in **DrawSportLightSource** to see how the shadows are dynamically recalculated.
- Uncomment the function **SceneGraphExampleInit** and uncomment the function **SceneGraphExample2Init** function to see the shadows in the pirate scene and **SceneGraphExample3Init** to see the Knossos model.