EDAN35 High Performance Computer Graphics
Assignment 2 : Deferred Shading and Shadow Maps

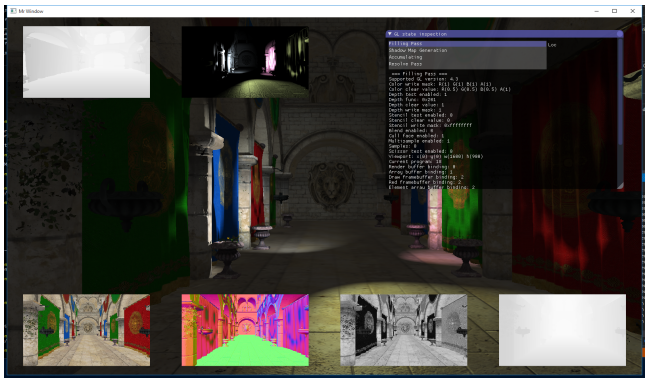

Figure 1: The main window using the Bonobo framework with debug windows for off screen buffers, and OpenGL state.

Figure 2: Diffuse texture.

1 Deferred Shading

1.1 Getting Started

Download the code from the assignments website, and unpack the zip-file. Start Microsoft Visual Studio by clicking on the EDAN35_Assignment2.sln file.

Look in the EDAN_Assignment2 project, in the Source Files folder, then Assignment Folder, and you should see the Deferred.cpp file. Also in the Resource Files Folder, then the Shaders folder, and then the Assignment folder, you can see the vertex (.vert) and fragment (.frag) shaders for the assignment.

All your changes should be in Deferred.cpp and the shaders. When you start the program, you will see the diffuse texture color of the sponza atrium model. You can move with WASD and the left mouse button.

1.2 Buffer and State Viewing

In this assignment you will use debugging utilities built into the Bonobo framework. In particular the buffers that are used are draw in small windows inside the main window. See Figure 1. This functionality allows you to look at the different geometry and light buffers that are being generated by the shaders. It allows you to inspect the render targets in real-time. In the main window, from the top left is the current light's shadow map, then the accumulated light buffer. Across the bottom of the window, from left to right is, the diffuse texture, the normal texture, the specular component, and the depth buffer. There is also a text window with the current OpenGL state for the current render passes. At startup this is the Filling Pass and the Resolve Pass. You can move, resize and minimize this window with the right mouse button.

1.3 Render Pass setup

There are three main rendering passes : Rendering the geometry buffer; Rendering the shadow maps and accumulating the lights; Final resolve using GBuffer and Light Buffer. At the beginning of each pass three things must be set correctly, render targets, shaders and clears. Render targets are set with bonobo::setRenderTargets. Shaders are set with glUseProgram(). Clears are performed by setting the clear value with either glClearDepth() or glClearColor, and then issuing the glClear() command with the correct bits set, either GL_DEPTH_BUFFER_BIT or GL_COLOR_BUFFER_BIT.

1.4 Rendering the geometry buffer

First off you need to render a geometry buffer. The geometry buffer is, in our case, a collection of three buffers, with the same size as the window we are rendering into. By binding these buffers as a render target, we set the graphics pipeline to put the resulting pixel color into these buffers instead of writing it to the screen. Also, as we have three buffers bound as render targets, we need to select which one to write to in the pixel shader.

Into these buffers, we write values making it possible for us to recreate the geometric information we need at each pixel to perform the lighting calculations.

There are three buffers in the geometry buffer, rtDiffuse, rtNormalSpecular, and rtDepthTexture. rtDiffuse has the diffuse texture color, which is what is displayed in the main window at startup, see Figure 2, rtNormalSpecular has the normal and specular values and is the normal is displayed in the second from the left window, which is all green at startup, and the specular is displayed in the third window, and is all white at startup. The rtDepthTexture is in the window on the right, and is black at startup. Note that you can still move around and see how that effects the geometry buffer in real time.

Start by looking at the depth buffer part of the geometry buffer, which is rtDepthTexture. Open deferred.cpp and find the function run(). Scroll down to where it says Pass 1, look at the code that renders the geometry to the geometry buffer. This loops through all the geometry in the scene, and renders it using the fill_gbuffer.vert and fill_gbuffer.frag shaders. However, this is rendered into the three buffers of the geometry buffer, instead of rendered to the screen.

Look in the shader fill_gbuffer.vert. This is similar to vertex shaders that you have seen before. You calculate model_to_world_normal_matrix, project the vertices positions to the

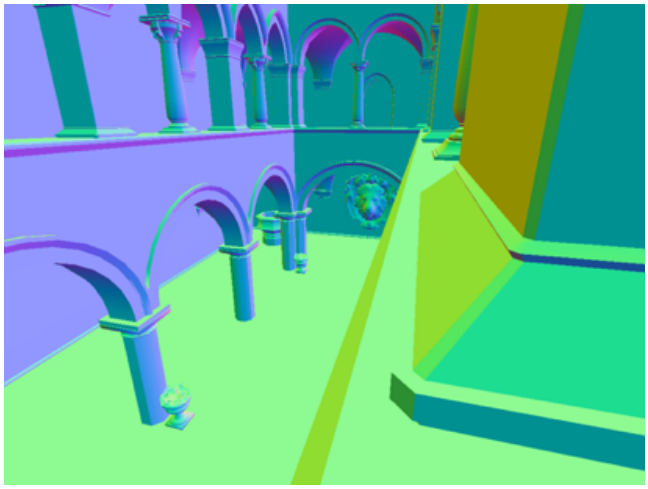

Figure 3: World space normals, encoded into RGB.

Figure 4: Lighting of one light source.

screen and pass through the texture coordinates. You do not have to change anything in this shader.

Now look at fill_gbuffer.frag. Here you can see where you write the values to the different textures of the geometry buffer. geometry_diffuse corresponds to rtDiffuse and so forth. If you look at geometry_diffuse, you can see how we write the diffuse texture. This is finished and you do not have to change this.

However, you do need to fix what you write to geometry_normal_and_specular. You have two more values to fill in. Remember that the values in the worldspace_normal has a range of -1.0 to 1.0, but the textures have a range of 0.0 to 1.0. Also, in geometry_norma_and_specular.a you should put the specular strength, which is read from the specular texture. The normal part of the buffer should look similar to Figure 3.

Render geometry buffer:

- Write normal to geometry_normal_and_specular.rgb
- Write specular strength to geometry_normal_and_specular.a

1.5 Adding light sources and shading

Next step is adding light sources. Four light sources are added automatically in the beginning. It is easy to add more, just set a higher amount to LIGHTS_NB in deferred.cpp. Start by making changes in deferred.cpp, where the comments say Pass 2, which is a loop over all the light sources. Uncomment all the commented out code inside Pass 2.2.

Now you need to implement the light calculations. These should be in accumulate_lights.frag. Open it up and look at it. Right now it only outputs a constant value of 0.0, 0.0, 0.0, 1.0. Here you should implement Phong shading, as you did in the introductory course. To do that you will need the position of the light source, and the position and normal of the geometry.

The normal of the geometry is available to you, as you wrote it to the geometry buffer. Now you need to retrieve it. As said, the geometry buffer is a collection of fullscreen textures. We need to locate and read the texel corresponding to the current pixel. To calculate the texture coordinates, look at gl_FragCoord again. What does the X and Y components contain? We also provide you with invRes, a vec2 containing the inverse of the windows resolution. Remember that you moved and scaled the normal to fit the textures 0.0 - 1.0 range. This needs to be undone.

You will also need the worldspace position of the pixel to shade it. Similar to the normal, you should be able to read and extract the depth from the depth part of the geometry buffer. You should then be able to compose a screenspace position with the depth and perform the inverse projection using the ViewProjectionInverse matrix supplied. Dont forget to divide by w.

Now, having the positions of both the light source and the geometry to be shaded, and the normal. You should do a simple phong calculation. Remember that you also have a specular term saved in the alpha channel of the normal and specular texture. Multiply this with the specular term of the lighting to gain visual richness.

Render light sources:

- Bind and clear render target
- Calculate texture coordinates from screen position
- Extract normal from geometry buffer
- Extract depth from geometry buffer
- Perform inverse projection to obtain world space position
- Extract specular strength from geometry buffer
- Calculate phong shading

1.6 Falloff and composition

To get a nice, correct lighting the light should have a falloff. The distance falloff is based on the square of the distance between the geometry and the light source. Similarly, as we are using a spotlight, it should have an angular falloff, depending upon the angle between direction from the light source to the geometry and the direction of the light. This is dependant upon the type of spotlight (reflector behind the bulb, etc.) so there is no right way of doing it. Implement a solution and make sure it reaches zero before 45○.

As the loop iterates over the light sources, their individual attibution is added by using an accumulative blend mode. This is done for you.

After this, the result should look similar to Figure 4.

Falloff and composition:

- Calculate distance falloff
- Calculate angular falloff
- Composite light using phong shading, falloffs and LightIntensity and LightColor

Figure 5: Lighting of one light source, with shadows.

Figure 6: Lighting of one light source, with close up of the shadow.

2 Shadow Maps

Looking at the result, only one thing is missing, shadows. For this lab we are using the technique called shadow maps. Shadowmaps work by rendering a depthmap from the point of view of the light source, e.g. the depth values of the surfaces hit by light from that light source.

2.1 Rendering the shadow map

First off, you need to render the shadow map. Start of by working in Deferred.cpp and uncommenting the code in Pass 2.1. Clear both the color and the depth. Color should be 0.0, 0.0, 0.0, 0.0 and the depth should be 1.0.

After that we need to render all the geometry to the shadow map. Now it is rendered with the shader called fill_shadowmap.frag.

If all this is done correctly, you should be able to see the result in the small debug at the top left. The shadow map is called rtShadowMap. It should look similar to the depth buffer part of the geometry buffer, however it should have a continous rotation, and not react to the controls.

Rendering the shadow map:

- Bind and clear shadow render target
- Bind, but do not clear, the light buffer render target

2.2 Using the shadow map

Now, while rendering the light from a light source, we can use the shadow map to check if a certain pixel is in shadow. Go back to the spotlight shader. To determine if a pixel is in shadow, use the matrix called shadowViewProjection to see what texel of the shadow map the pixel projects into by performing the projection. Dont forget to divide by w. By doing the projection, you also calculate the depth of the fragment, in the projection space of the shadow map camera. This should be compared to the depth read from the shadow map at the projected position to determine if the surface is hit by light from the light source at this depth. Remember that the projected values are in the range -1.0 - 1.0, the depth in the shadow map is between 0.0 - 1.0 and texture coordinates should be between 0.0 and 1.0.

The result of this comparison should be used to determine if any light should be added at all. The result should look similar to Figure 5. However, if you look closer, it looks like Figure 6 Which brings us to Percentage Closer Filtering.

Using the shadow map:

Figure 7: Lighting of one light source, with PCF-filtered shadows.

- Project world space position using the shadow camera's view-projection matrix
- Calculate the depths
- Compare anda djust the light

2.3 Percentage Closer Filtering

To reduce the blockiness of the shadows, we are going to implement a technique called Percentage Closer Filtering or PCF. This is done by basically doing more lookups in the shadow map, and weighting the results together. To do this, implement a sampling scheme around the projected position of the fragment. The sampling can be as simple as a double for-loop doing regular grid sampling. Compare each read and extracted depth with the same calculated depth and weigh the results together. The result should look like Figures 7 and 8, the final scene should look like Figure 9.

Percentage Closer Filtering:

- Sample the shadow map at several positions
- Compare the depths
- Weight the results together and adjust the light

Figure 8: Lighting of one light source, with close up of PCF-filtered shadows.

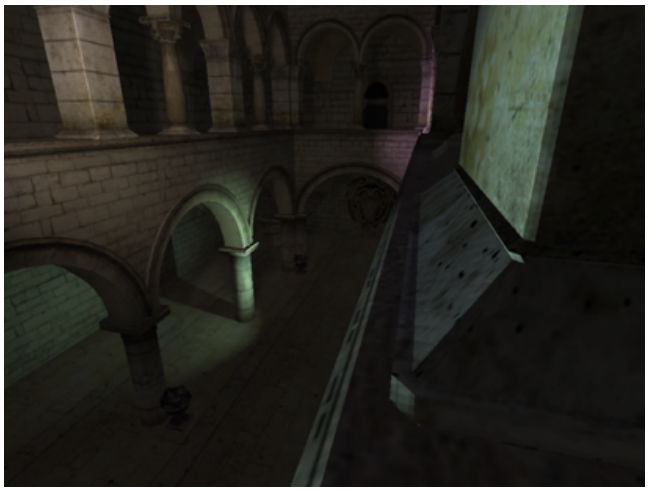

Figure 9: Final solution.

3 Voluntary part: Normal maps

Now, to make the rendering even better looking. This part is voluntary, and not required to pass the assignment.

In the introductory course, you implemented a method called bump mapping. This is a similar technique for adding details by using a texture to alter the normal of the surface. If you look in the res/crysponza/textures directory, you can see several textures that are mostly purple, with green and red streaks. These are normal maps, where each pixel is a normal, encoded in a similar way to how you encoded your normals in the geometry buffer. However, different from your normals, these normals are not expressed in world space, but in a space called tangent space. This is a 3D-space aligned with the normal and the directions of the texture coordinate space. Using a 3x3 matrix representing this space, you can transform the encoded normal to worldSpace, which adds a lot of detail to the shading of the scene.

This is done before writing the normal to the geometry buffer in fill_gbuffer.frag. Instead of just normalizing the worldspace_normal and changing the range of it, use the supplied worldspace_tangent, worldspace_binormal and worldspace_normal to create a 3x3 matrix, transforming from tangent space to world space. Look at

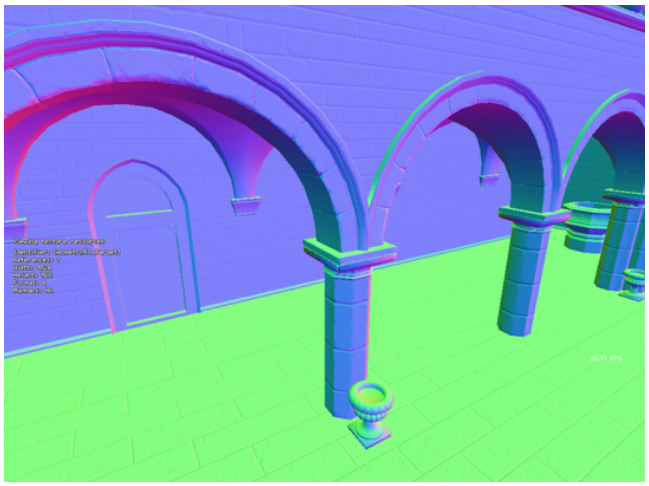

Figure 10: Worldspace normals, using normal maps, encoded into RGB.

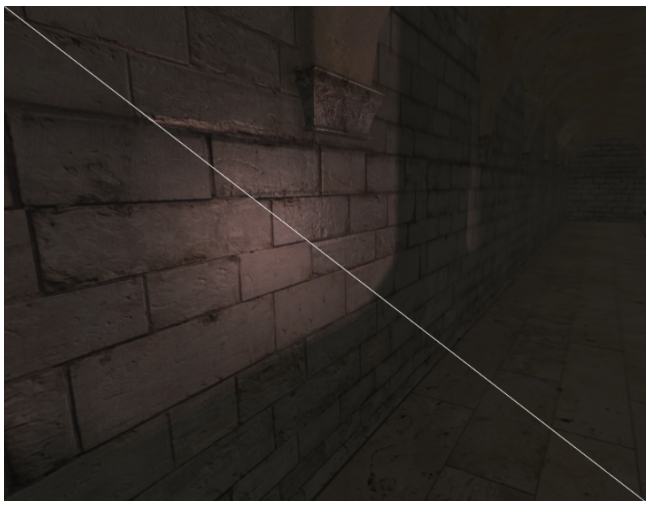

Figure 11: Lower left without normal maps. Upper right with normal maps.

GLSLs mat3 constructor taking three vec3s as arguments. Use the created matrix to transform the normal read from the normal map. Dont forget to normalize and encode it in a range suitable for writing to the normal texture.

The normal buffer should go from looking like Figure 3 to Figure 10.

The final result, with details added by the normal maps should look like Figure 11.

Normal maps:

- Setup tangent space
- Read normal from normal map
- Transform normal to world space
- Write the transformed normal to geometry buffer