# EDAN35: High Performance Computer Graphics
# Assignment 2: Deferred shading and shadow maps

## 1 Deferred shading

The purpose of this exercise is to get a higher understanding of how advanced graphics can be done using shaders.

### 1.1 Getting started

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

Check in EDAN35. There is deferred.cpp and a folder with shaders. All your changes should be in these. When you start the program, a dim image of the Sponza atrium appears. You can move with WASD and the left mousebutton.

### 1.2 Resource viewer

In this assignment you will be using something called the resource viewer extensively. The resource viewer is functionality in RenderChimp that allows you to look at the geometry and the textures that are loaded in the graphics engine. It also allows you to inspect render targets in realtime.

The resource viewer has to modes, one for textures and one for geometry. The texture mode is entered by pressing $F2$ on windows and 2 on macos x. Geometry mode is entered by pressing $F3$ of 3.

In each mode you can flip through the resources using the arrowkeys, left and right for one resource forward and backwards, and up and down for ten resources at a time.

In texture mode you can also separate the channels and see each channel by itself, as seen in Figure 2. This is done by pressing $space$.

Resource viewer controls:

- Enter and exit texture mode: $F2$(win) or 2(mac)
- Enter and exit geometry mode: $F3$(win) or 3(mac)
- Flip forward: right arrow
- Flip backwards: left arrow
- Flip 10 forward: up arrow
- Flip 10 backwards: down arrow
- Switch between composed and separated channels: $space$

### 1.3 Rendering the geometry buffer

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

Figure 1: Diffuse texture

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

Start by looking at the different textures in the geometry buffer in the resource viewer.

There are three textures in the geometry buffer, GeometryRT_Target0 to GeometryRT_Target2. Target0 is fully white, Target1 is fully black and Target2 contains the diffuse texture of the geometry, see Figure 1. 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 GeometryRT_Target0. The default rendering should result in white. Open deferred.cpp and find the function RCUpdate. 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 GeometryBuffer.vs and GeometryBuffer.fs shaders. However, this is rendered into the three textures of the geometry buffer, instead of rendered to the screen.

Look in the shader GeometryBuffer.vs. This is similar to vertex shaders that you have seen before. You calculate worldNormal, project the vertices positions to the screen and pass through the texture coordinats. You do not have to change anything in this shader.

Now look at GeometryBuffer.fs. Here you can see where you write the values to the different textures of the geometry buffer. gl_FragData[0] corresponds to GeometryRT_Target0 and so forth. If you look at gl_FragData[2], 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 gl_FragData[0] and gl_FragData[1]. Start with gl_FragData[0]. This should contain the depth. Here there are two problems to solve.

First, what is the depth? Look at gl_FragCoord in the OpenGL GLSL specifications. Second, the depth is a 32 bit float. The texture

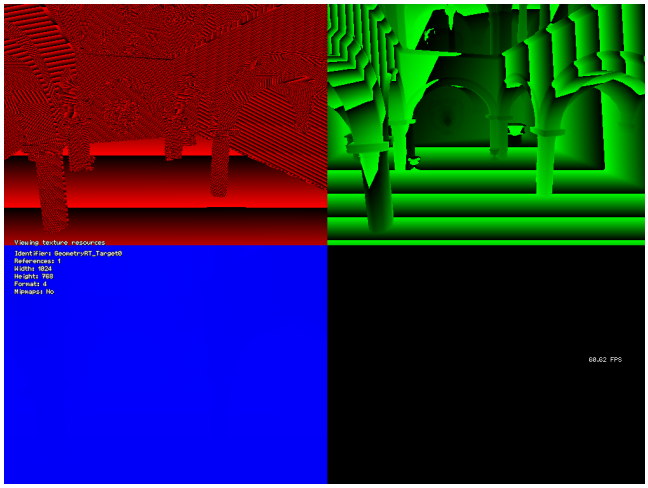

Figure 2: Depth buffer, split into RGBA. To see the channel separately, press *space*

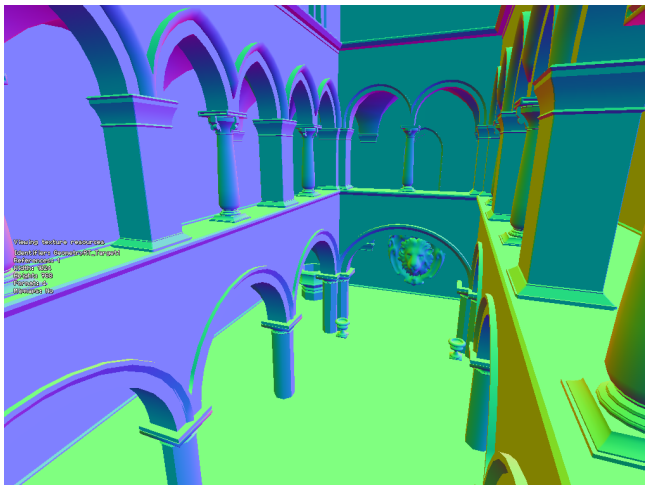

Figure 3: Worldspace normals, encoded into RGB

we are trying to save it in is RGBA, with each channel containing 8-bit values between 0.0 and 1.0. Methods for converting 8-bit RGBA to 32-bit float and vice versa is supplied in the shaders. Look at encodeDepth in GeometryBuffer.fs and EXTRACT_DEPTH in DeferredSpotLight.fs. Tip, look at the result using the resource viewer, and see how it changes when you move around. It should look similar to Figure 2.

You then have two more values to fill in. In gl_FragData[1], you should put the normal in x, y and z. Remember that the values in the worldNormal has a range of -1.0 to 1.0, but the textures has a range of 0.0 to 1.0. Also, in gl_FragData[1].a you should put the specular strength, read from the specular texture. The normal part of the buffer should look similar to Figure 3.

Render geometrybuffer:

- Write depth to gl_FragData[0].rgba
- Write normal to gl_FragData[1].rgb
- Write specular strength to gl_FragData[1].a

Figure 4: The initial light buffer

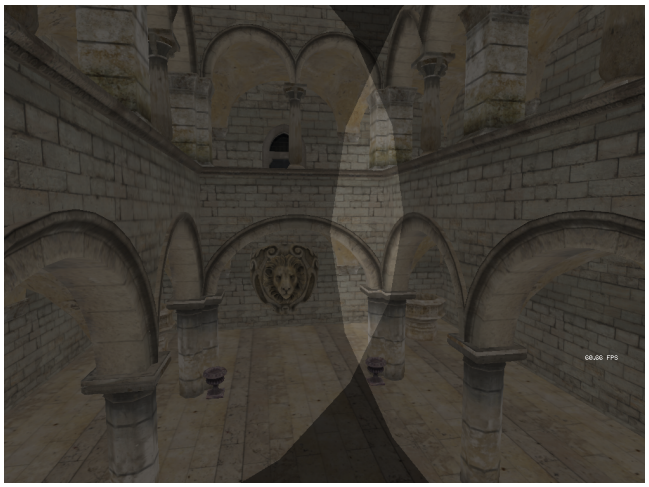

Figure 5: The initial lighting, with diffuse texture

### 1.4 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 N_LIGHTS in deferred.cpp. Start by changing in Deferred.cpp, where the comments say Pass 2. Here is a loop over all the light sources.

First of, before the loop, you need to make sure you render to the right context. You should render to the render target called *lightBuffer*. So, set the render target and clear the colorbuffer of it to $0.0, 0.0, 0.0, 0.0$. Important to note, do NOT clear the depthbuffer. This should all be done outside and before the loop.

Inside the loop, you need to render the bounding volume of the light source, a geometry called *lightBounds* in the code. To do this, you need to set the camera matrices, in the same way as it is done before rendering to the geometry buffer in Pass 1. Use the same camera. Then call *Renderer::render()*, but send in a pointer to *lightBounds* and use the appropriate shader, *spotLightShader*. After this is done, you should be able to find the texture LightRT_Target0 in the resource viewer, looking similar to Figure 4. The backbuffer should look similar to Figure 5.

Now you need to implement the light calculations. These should be in DeferredSpotLight.fs. Open it up and look at it. Right now it only outputs a constant value of $0.2, 0.2, 0.2, 1.0$. 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. Don't forget to divide by $w$.

You will also need the world position of the light source. This can be extracted from the light source object in Deferred.cpp and sent into the shader by calling *setValue("LightPosition")* on *lightBounds*. The variable is already declared in the shader. You should also send in the lights direction in the same way, extracted by calling *getWorldFront()* on the light source.

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
- Set camera resources
- Render light sources' bounding volume
- calculate texture coordinates from screen position
- Extract normal from geometry buffer
- Extract depth from geometry buffer
- Perform inverse projection to obtain world space position
- Send in light position and light direction
- Extract specular strength from geometry buffer
- Calculate phong shading

### 1.5 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 $45^\circ$.

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

Figure 6: Lighting of one light source

After this, the result should look similar to Figure 6.

Fallof and composition:

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

## 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 ligh-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. Inside the loop, after the position and rotation of the light source has been updated we need to change to the shadow map render target, *shadowMap*. This must be done before setting up and rendering the lights bounding volume to the light buffer. This is done in the same way as the you previously set the light buffer render target before the loop. So, set the render target and 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 the render target is setup and cleared you need to render to it. Start off by setting up the camera. Here you need to take the shadow camera and attach it to the light source wiht *attachChild()*. After that you need to use the camera to set the correct camera matrices to the renderer.

After that, all you need to do is to render all the geometry in the geometrylist to the shadow map, similar to when you rendered it to the geometry buffer. However, you should render it with a special shader called *sBufferShader*.

Then comes an important part. You need to extract the viewprojectionmatrix from the shadow camera, to be able to later perform the same projection within the spotlight shader. The camera has a

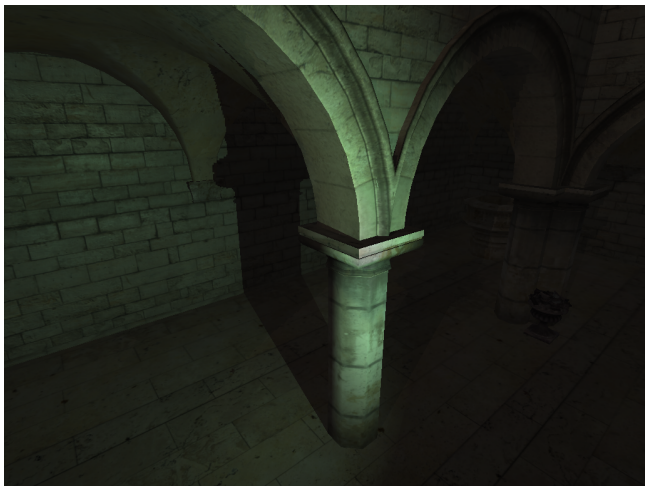

Figure 7: Lighting of one light source, with shadows

Figure 8: Lighting of one light source, with closup of the shadows

method call for this. This matrix needs to be sent into the spotlight shader. A suitable variable is declared in DeferredSpotLight.fs.

After this you need to detach the shadowCamera to be able to attach it to another light source. This must all be done before rebinding the light buffer render target and rendering the bounding volume of the light.

Now open the fragmentshader for writing the shadow map. It is ShadowBuffer.fs. In this, the final color is set to $1.0, 1.0, 1.0, 1.0$. It should contain the depth, encoded in exactly the same way as in the geometry buffer.

If all this is done correctly, you should be able to look at the result using the resource viewer. The shadow map, called ShadewRT_Target0. It should look similar to the depthbuffer part of the geometry buffer, in Figure 2, however it should have a continous rotation, and not react to the controls.

Rendering the shadow map:

- Bind and clear shadow rendertarget
- Attach shadow camera to lightsource

Figure 9: Lighting of one light source, with PCF-filtered shadows

- Set camera matrices to renderer
- Render geometry to shadow map
- Extraxt viewprojection matrix from shadow camera
- Detach the shadow camera from light source
- Bind, but do not clear, the light buffer render target
- Encode and write depth in fragment shader

### 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. Don't forget to divide by $w$. By doing the projection, you also calculate the depth of the fragment, in the projectionspace 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 is 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 Figure7. However, if you look closer, it looks like Figure 8. Which brings us to Percentage Closer Filtering.

Using the shadow map:

- Project world space position using the shadow cameras view-projection matrix
- Calculate the depths
- Compare and adjust 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

Figure 10: Lighting of one light source, with closup of the PCF-filtered shadows

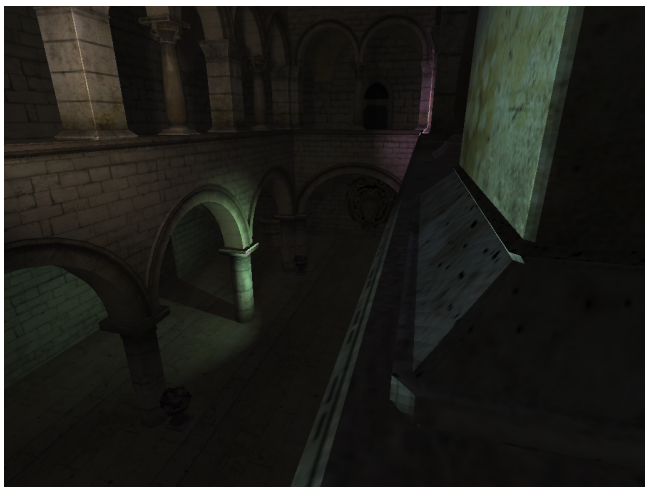

Figure 11: Final solution

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 as Figures 9 and 10, the final scene should look like Figure 11.

Percentage Closer Filtering:

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

## 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 technic for adding details by using a texture to alter the normal of the surface. If you flip through the textures you 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.

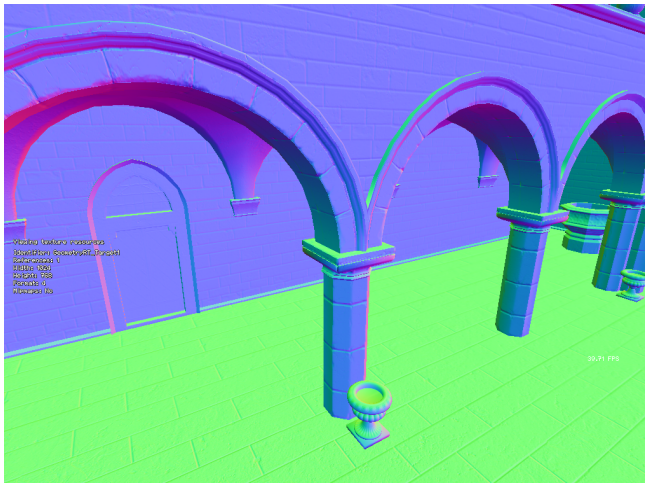

Figure 12: Worldspace normals, using normal maps, encoded into RGB

This is done before writing the normal to the geometry buffer in GeometryBuffer.fs. Instead of just normalizing the wordNormal and changing the range of it, use the supplied worldTangent, worldBinormal and worldNormal to create a $3x3$ matrix, transforming from tangent space to world space. Look at glsls' mat3 constructor taking three vec3s as arguments. Use the created matrix to transform the normal read from the normal map. Don't 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 12.

The final result, with details added by the normal maps should look like Figure 13.

Normal maps:

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

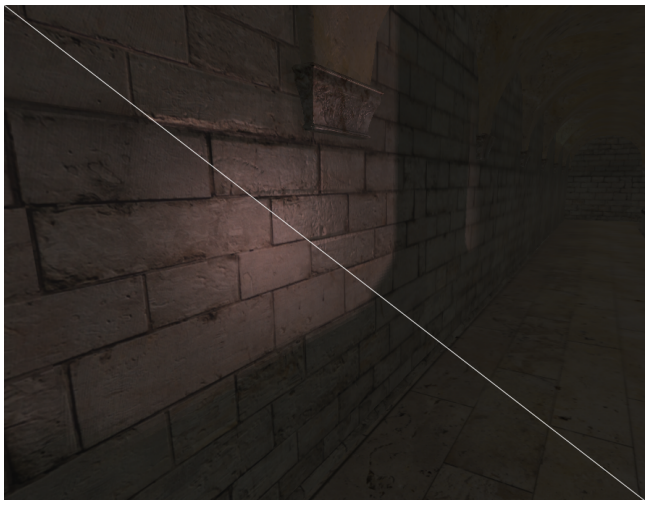

Figure 13: Final solution. Lower left without normal maps. Upper right with normal maps.