Assignment 2: Deferred Shading and Shadow Maps

Lund University Graphics Group

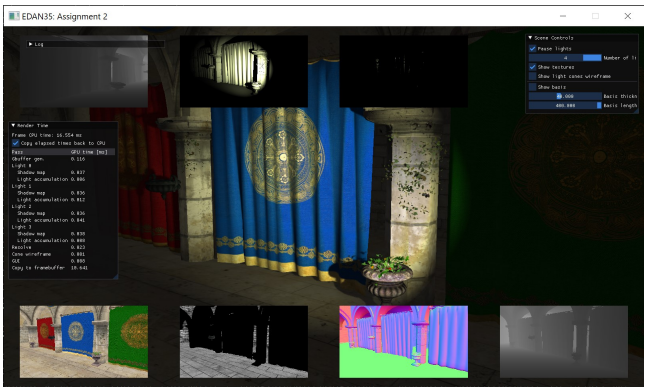

Figure 1: The main window using the Bonobo framework with debug windows for off-screen buffers, and performance timings.

1 Deferred shading

1.1 Getting started

Follow the instructions found in the BUILD.rst file on GitHub to setup everything. For the last steps of the setup, you should use `src ▸ EDAN35` instead of `src ▸ EDAF80`, and `EDAN35_Assignment2` instead of `EDAF80_Assignment1`. When you start the program, you will see the diffuse texture color of the Sponza atrium model. A recap of application and IDE shortcuts can be found in Section A and B at the end of this document.

All your changes should be in `src ▸ EDAN35 ▸ assignment2.cpp` and the shaders `shaders ▸ EDAN35 ▸ fill_gbuffer.frag` and `shaders ▸ EDAN35 ▸ accumulate_light.frag`.

1.2 Buffer viewing

In this assignment you will use debugging utilities built into the Bonobo framework. In particular, the buffers that are used are drawn in small windows inside the main window, as illustrated in Figure 1. This functionality allows you to look at the different geometry and light buffers that are being generated by the shaders. In the main window, from the top left is the current light's shadow map, then the accumulated light diffuse and light specular buffers. Across the bottom of the window, from left to right is, the diffuse texture, the specular component, the normal texture, and the depth buffer.

In addition to the showing the buffers, there is are three different GUI windows:

1. One for showing the application logs;
2. One for controlling some scene parameters like pausing the light animations or reducing the number of lights used;
3. And one for listing how long each pass took to perform, from the point of view of the GPU.

1.3 Render pass setup

There are three main rendering passes:

1. Rendering the geometry buffer;
2. Rendering the shadow maps and accumulating the lights;
3. Final resolve using GBuffer and Light Buffer.

At the beginning of each pass three things must be set correctly: framebuffers, shaders and clears. Framebuffers are set using `glBindFramebuffer()`, and shaders using `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 four buffers, with the same size as the window we are rendering into. These four buffers are grouped together under a framebuffer object, which can then be used as an alternative destination to the screen when rendering, i.e. instead of having the output of the fragment shaders being shown on screen, they will be saved into the framebuffer. As a framebuffer object can have multiple attached buffers, the fragment shader can output multiple values and use the `layout (location = X)` notation to match an output variable to a buffer.

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 four buffers in the geometry buffer, all stored in the `textures` variable at the following indices:

- `Texture::GBufferDiffuse` has the diffuse texture color, which is displayed in the first window from the left
- `Texture::GBufferSpecular` has the specular color, which is displayed in the second window from the left
- `Texture::GBufferWorldSpaceNormal` has the normal and is displayed in the third window, and is all black at startup
- `Texture::DepthBuffer` is in the window on the right.

Note that you can still move around and see how that effects the geometry buffer in real time.

Start by moving the camera around, and look towards the sky. **Which artefacts can you see? Why is it happening and how could you fix it?**

Open `assignment2.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 shaders `fill_gbuffer.vert` and `fill_gbuffer.frag`. However, this is rendered into the four 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 project the vertices positions to the screen and pass through the texture coordinates as well as the normal, tangent and binormal. 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 the diffuse texture, 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`. Remember that normals have their components in $[-1, 1]$, but for textures they are in $[0, 1]$. The normal part of the buffer should look similar to Figure 2.

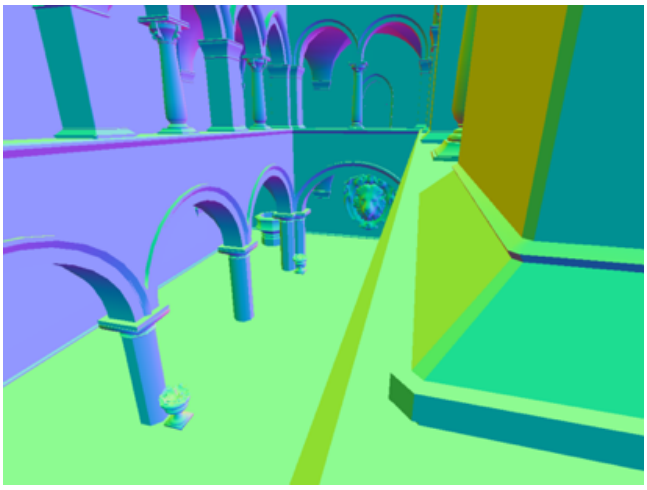

Figure 2: World space normals, encoded into RGB.

Figure 3: Lighting of one light source.

Render geometry buffer:

- Clear buffers that need to;
- Write normal to `geometry_normal`.

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 `constant::lights_nb` in `assignment2.cpp`.

Open `accumulate_lights.frag` up and look at it. Right now it only outputs a constant value of $\begin{pmatrix}0 & 0 & 0 & 1\end{pmatrix}$. Change the written value for the red channel to 0.2 for example; any value will do as long as the final color is not pure black. Now reload the shaders and have a look at the small window displaying the content of texture `Texture::LightDiffuseContribution`. You should see the region lit by a spotlight, but **what happens as the spotlight rotates**? You should probably go to `assignment2.cpp` and have a quick look at what happens during the second pass. Once that has been fixed, go back to `accumulate_lights.frag`. 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 do the X and Y component contain? We also provide you with `inverse_screen_resolution`, a **`vec2`** containing the inverse of the windows resolution. Remember that you moved and scaled the normal to fit the textures $[0, 1]$ range. This needs to be undone.

You will also need the world-space 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 screen-space position with the depth and perform the inverse projection using the `camera.view_projection_inverse` matrix supplied. Don't 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**, but without the ambient term as it will be added during the last pass. Similarly, k_d and k_s will also be added during the last pass. The remaining of the phong diffuse and specular terms should be placed into their respective buffers `light_diffuse_contribution` and `light_specular_contribution`.

Render light sources:

- Clear any buffers that need to;
- Calculate texture coordinates from screen position;
- Extract normal from geometry buffer;
- Extract depth from geometry buffer;
- Perform inverse projection to obtain world space position;
- 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 contribution is added by using an accumulative blend mode. This is done for you.

After this, the result should look similar to Figure 3.

Falloff and composition:

- Calculate distance falloff;
- Calculate angular falloff;
- Composite light using phong shading, falloffs and `light_intensity` and `light_color`.

2 Shadow Maps

Looking at the result, only one thing is missing, shadows. For this lab we are using the technique called shadow maps. Shadow maps work by rendering a depth map 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. Most of the work has already been done in `assignment2.cpp`, but still the content of the shadow map (displayed in the window in the top left corner) remains completely black. **What is possibly happening, and how can you fix it?** If all this is done correctly, you should now be able to see the result in the small debug at the top left. The shadow map is called `shadowmap_texture`. It should look similar to the depth buffer part of the geometry buffer, however it should have a continuous rotation, and not react to the controls.

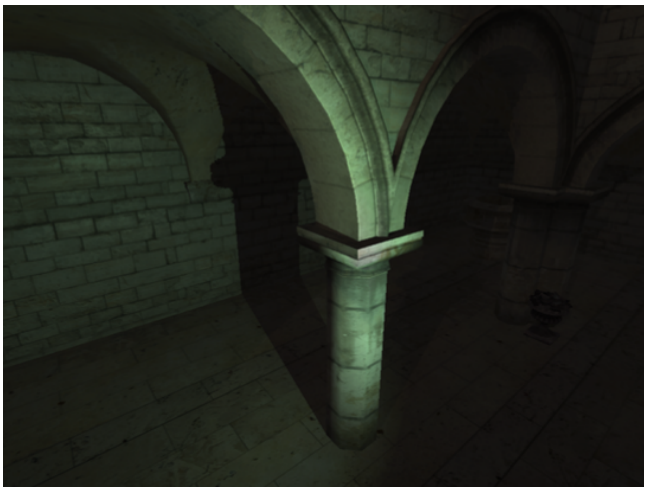

Figure 4: Lighting of one light source, with shadows.

Figure 5: Lighting of one light source, with close up of the shadow.

Rendering the shadow map:

- Find out why the shadow map is black.

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 `lights[light_index].view_projection` 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 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 $[-1, 1]$, the depth in the shadow map is between $[0, 1]$ and texture coordinates should be in $[0, 1]$.

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 4. However, if you look closer, it looks like Figure 5, which brings us to *Percentage Closer Filtering*.

Using the shadow map:

- Project world space position using the shadow camera's view-projection matrix;
- Calculate the depths;
- Compare and adjust the light.

Figure 6: Lighting of one light source, with PCF-filtered shadows.

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

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 6 and 7, the final scene should look like Figure 8.

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 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.

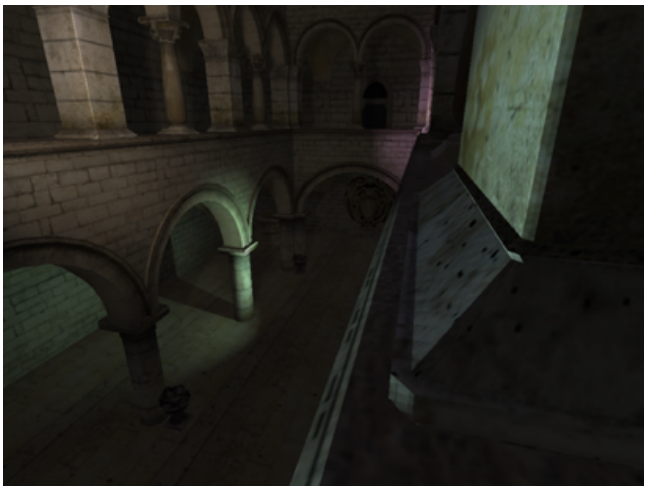

Figure 8: Final solution.

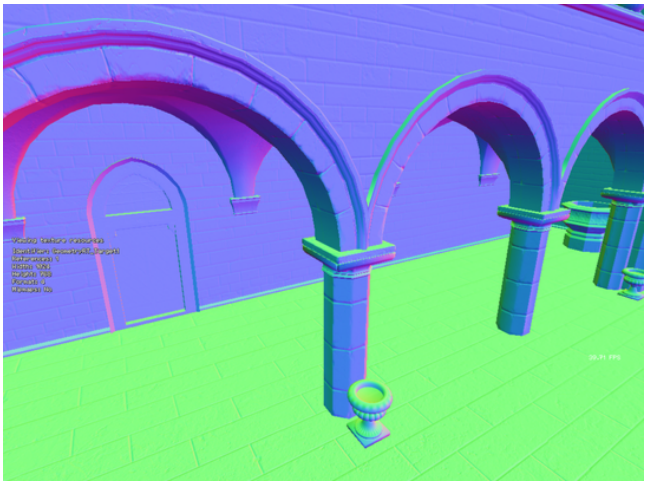

Figure 9: World-space normals, using normal maps, encoded into RGB.

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 3×3 matrix representing this space, you can transform the encoded normal to world space, 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 `fs_in.normal` and changing the range of it, use the supplied `fs_in.tangent`, `fs_in.binormal` and `fs_in.normal` to create a 3×3 matrix, transforming from tangent space to world space. Look at GLSL's **`mat3`** constructor taking three **`vec3`**s 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 2 to Figure 9.

The final result, with details added by the normal maps should look like Figure 10.

Normal maps:

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

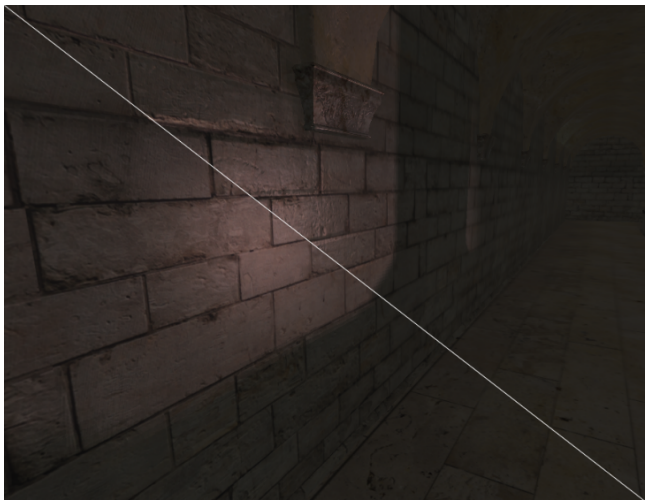

Figure 10: Lower left without normal maps. Upper right with normal maps.

Table 1: Various controls when running an assignment."Reload the shaders" is not available in assignments 1 and 2 of EDAF80, while "Toggle fullscreen mode" is missing from assignment 2 of EDAN35.

Action	Shortcut
Move forward	W
Move backward	S
Strafe to the left	A
Strafe to the right	D
Move downward	Q
Move upward	E
"Walk" modifier	⇧
"Sprint" modifier	Ctrl
Reload the shaders	R
Hide the whole UI	F2
Hide the log UI	F3
Toggle fullscreen mode	F11

A Framework controls

The framework uses standard key bindings for movement, such as W, A, S, and D. But there are also custom key bindings for moving up and down, as well as controlling the UI. All those key bindings are listed in Table 1.

There is only one action currently bound to the mouse, and that is rotating the camera. To do so, move the mouse while holding the left mouse button.

GUI elements can be toggled being a collapsed and expanded state by double clicking on their title bar. And they can be moved around the window by dragging their title bar wherever desired (within the window).

B IDE key bindings

To help with getting certain tasks done more efficiently, Table 2 lists key bindings of different IDEs for several common actions.

Table 2: Various keyboard shortcuts for Visual Studio 2019 and 2017, and Xcode.

Action	Shortcut	
	Visual Studio	Xcode
Build	Ctrl + B	⌘ + B
Run (with the debugger)	F5	⌘ + R
Run (without the debugger)	Ctrl + F5	
Toggle breakpoint at current line	F9	⌘ + \
Stop debugging	⇧ + F5	⌘ + .
Continue (while in break mode)	F5	ctrl + ⌘ + Y
Step Over (while in break mode)	F10	F6
Step Into (while in break mode)	F11	F7
Step Out (while in break mode)	⇧ + F11	F8
Comment selection	Ctrl + K, Ctrl + C	⌘ + /
Uncomment selection	Ctrl + K, Ctrl + U	⌘ + /
Delete entire row	Ctrl + X	