CMSC 23700 **Introduction to Computer Graphics** **Project 4**
Autumn 2015 **November 3, 2015**

Deferred Rendering

Due: Monday November 16 at 10pm

1 Summary

This assignment uses the same application architecture as the earlier projects, but it replaces the *forward rendering* approach with *deferred rendering*.

2 Description

The rendering techniques that you have implemented in the previous projects is sometimes called *forward rendering*. While it is very effective for many applications, it does not handle large numbers of light sources well. In a scene with many lights (*e.g.*, street lights going down a highway), we might have to run a forward-rendering pass per light (or fixed-size group of lights) resulting in substantial extra work. Furthermore, we may have to compute lighting for many fragments that are later overdrawn by other, closer, fragments. (Note: some hardware uses an early Z-buffer test to avoid running the fragment shader on fragments that will be discarded later, but this optimization only helps with fragments that are further away from the ones already drawn).

2.1 Basic technique

The idea of deferred shading (or deferred rendering) is to do one pass of geometry rendering, storing the resulting per-fragment geometric information in the *G-buffer*. A G-buffer is essentially a frame buffer with multiple attachments or *Multiple Render Targets* (MTR). These include

- fragment positions (*i.e.*, world space coordinates)
- the diffuse surface color without lighting
- the specular surface color and exponent without lighting
- world-space surface normals
- depth values

We then use additional passes (typically one per light) to compute lighting information and to blend the results into the final render target. Figure 1 shows the G-buffer contents and final result for a G-buffer with four components (color, normal, and depth).

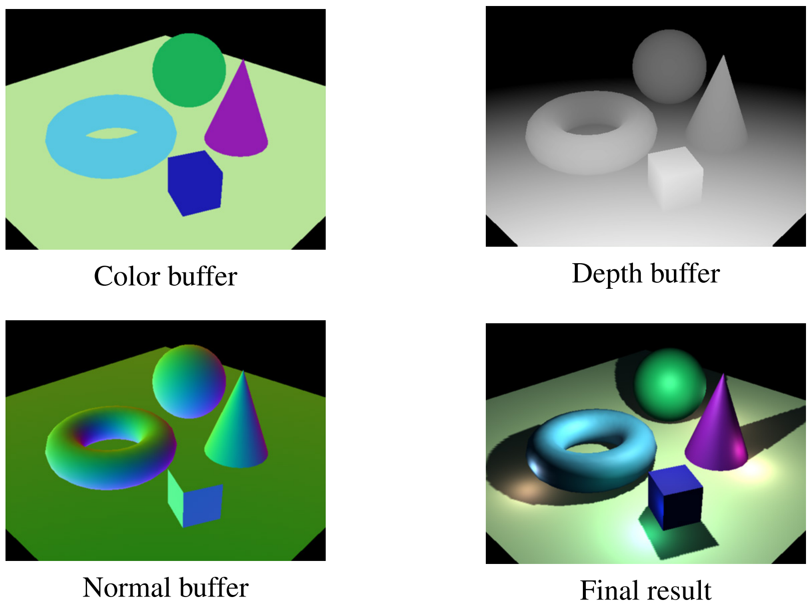
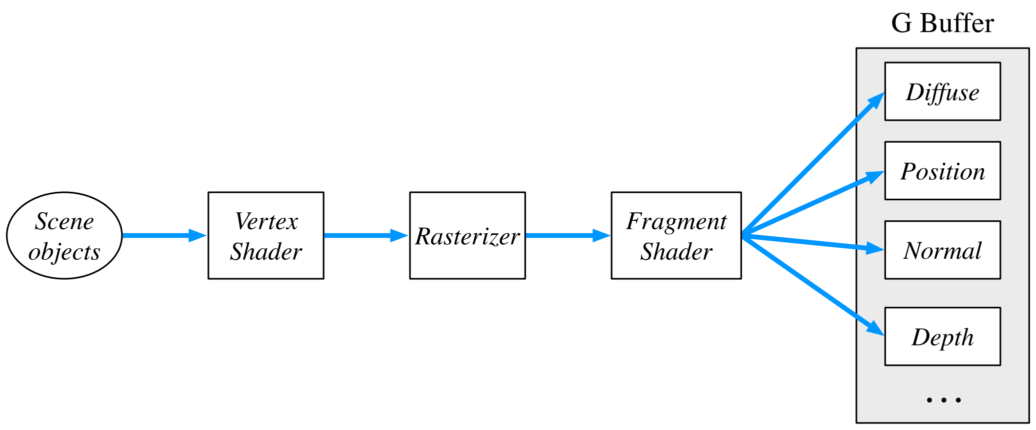


Figure 1: The G Buffer contents and final result

In the remainder of this section, we describe deferred rendering in more detail. You can also find additional information in Chapter 13 of the OpenGL SuperBible (pp. 613–624).

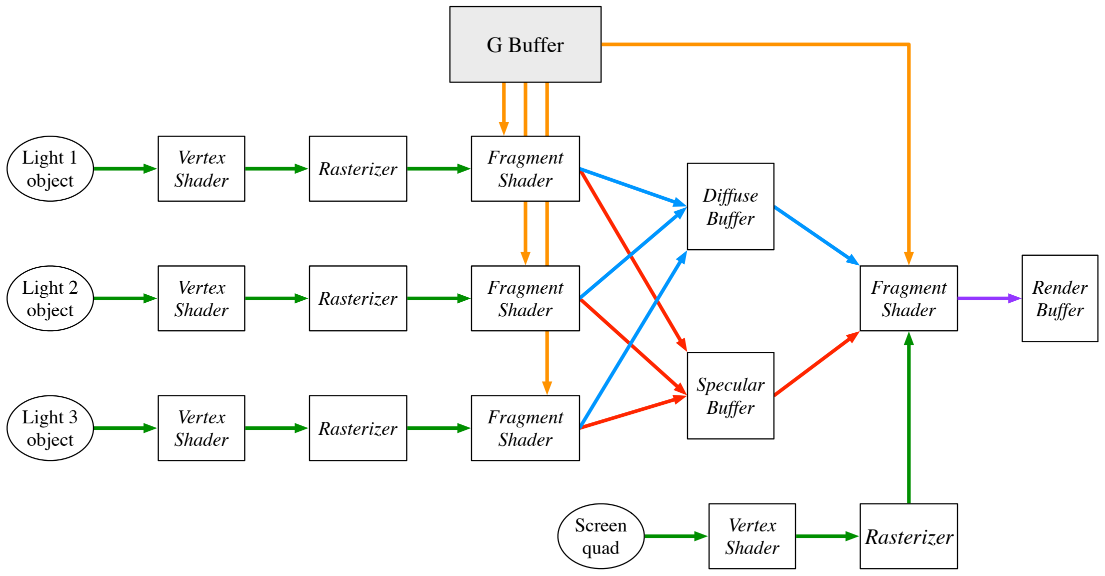
2.1.1 The Geometry Pass

The first pass is called the *geometry pass* and consists of a vertex shader that transforms the verticesas usual, but also outputs the world-space coordinates, and a fragment shader that writes the various pieces of information that we need to compute lighting to the G-buffer.



2.1.2 Lighting passes

The second phase of the technique is to run a rendering pass for each light in the scene. This pass renders a mesh that encompasses the light volume (in the case of directional lights, we can render the screen quad). The vertex shader for the lighting pass is trivial; it just transforms the vertex into clip-space coordinates, which are used to generate fragments that provide the iteration structure for doing the lighting calculation. We blend the output of each lighting pass into buffers for the diffuse and specular illumination components.



At this point, we might also compute screen-space ambient occlusion in a separate pass (see Section 4 below).

2.1.3 Final blending

The final phase is to combine the diffuse and specular buffers from the lighting passes and write them to the default framebuffer for display on the screen. We do this by drawing a screen-sized quad (*i.e.*, two triangles) and using the generated fragment coordinates to drive the blending of the lighting information with the diffuse color information in the G-buffer.

2.2 Implementation details

There is also the problem of computing lighting effects that cannot have any impact (because of attenuation). We need limit the lighting calculations to those fragments that are actually affected by the light. Furthermore, since the only fragments represented by the G-buffer are visible, we would like to avoid computing lighting information for occluded fragments.

The basic idea is to render point lights and spot lights as geometric objects representing the light volume (*i.e.*, spheres and cones), where we use the light's attenuation factor to determine the size of the volume. In the case of point lights, we can determine the sphere's radius as follows. Let

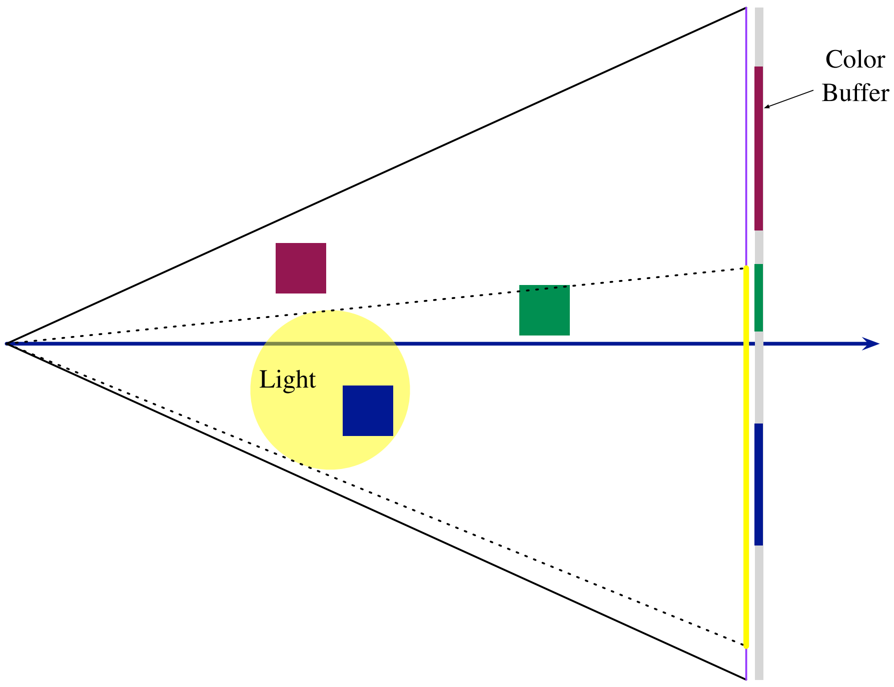
$$\begin{array}{rcll} I_{max} & = & \max(I_r, I_g, I_b) & \text{maximum per-channel intensity for the light} \\ \text{At}(d) & = & \frac{1}{ad^2+bd+c} & \text{attenuation as function of distance } d \end{array} \quad (1)$$

Assuming that we have 8-bits per channel, want to solve for d (assuming $a \neq 0$)

$$\begin{aligned}
\mathrm{At}(d)I_{max} &= \frac{1}{255} \\
255I_{max} &= ad^2 + bd + c \\
0 &= ad^2 + br + (c - 255I_{max}) \\
d &= \frac{-b + \sqrt{b^2 - 4a(c - 255I_{max})}}{2a}
\end{aligned}$$

Say $a = 0.3$, $b = 0$, and $c = 0$, then we get $d \approx 29$. The lights in the project are spot lights, so they will be represented as cones (there is a type `cs237::Cone` in the common code that can be used to generate the mesh representation of a spot-light volume).

When rendering the light-volume triangles, we have a situation like the following figure:



In this example, we would compute lighting information for all of the fragments in the yellow region of the screen. In this case, only the region covered by the blue object would have non-zero lighting, since all of the other fragments are outside the light volume and thus the light does not affect them.

This technique reduces the number of fragments for which we must compute lighting information, but it can still result in unnecessary lighting computations. For example, the green object is outside the light volume, but we would still spend time computing lighting for it. We can use a technique similar to stencil shadows to further reduce the lighting work. The basic idea is to do two passes for the light. The first puts non-zero values into the stencil buffer for those fragments whose world-space position is inside the light volume. The second pass uses this stencil information to discard fragments before lighting.

Specifically, we do the following for each point light or spot light in the scene:[1]

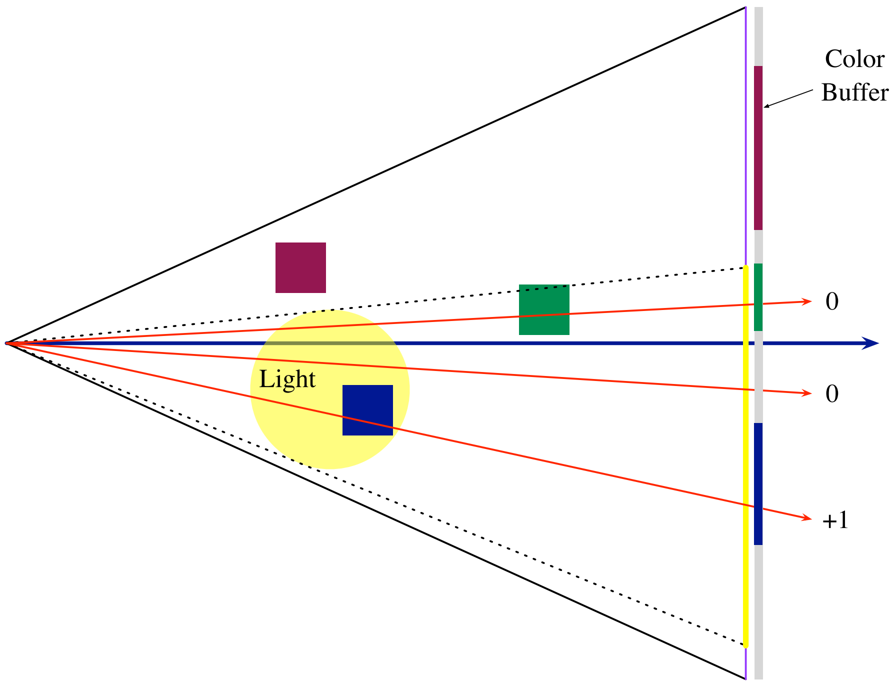
[1] Since the directional light affects the entire scene, there is little benefit in this technique for it.

1. Clear the stencil buffer.
2. Enable stencil test (`GL_ALWAYS`) and depth tests (`GL_LESS`) against the depth buffer computed in the geometry pass. We also disable culling so that both front and back-facing triangles are rendered.
3. Set the stencil operations as follows:

Face	**Stencil-test Fail**	**Depth-test Fail**	**Depth-test Pass**
Front	Keep	Keep	$+1$
Back	Keep	Keep	-1

4. Render the light-volume mesh using a null fragment shader.
5. Disable the depth test and enable front face culling[2]
6. Set the stencil test to $\neq 0$.
7. Render the light-volume polygons and compute lighting effects in the fragment shader, blending them into the result. The stencil test will cause fragments to be discarded after rasterization, but prior to the fragment shader, which will save unnecessary computation.

The following figure illustrates how this technique works. Light rays that pass through the light volume without hitting an object produce a stencil value of 0, whereas light rays that hit an object inside the light volume produce a stencil value greater than 0.



For a discussion of the stencil buffer operations, see Chapter 9 of the OpenGL SuperBible (pp. 372–376).

[2]We need to render the back face polygons to handle the case where the eye is in the light sphere.

Deferred rendering consumes a lot of memory bandwidth (especially during the geometry pass, where we are writing out multiple screen-sized render buffers). To reduce that cost, one can pack data into fewer targets using half floats, combining multiple logical buffers into a single render buffer, *etc.* See the SuperBible discussion for an example.

3 Project details

For this project, you will be implementing a variation of deferred rendering in a viewer that is similar to that of Projects 1–3. We will provide code for the viewer UI and and a forward-renderer that supports wireframe and texture rendering.

3.1 Scenes

The structure of scenes is similar to that of projects 1–3, but we have added *spotlight* objects to the scene, in addition to the directional and ambient light sources. The objects are in an array inside the `lighting` object in the scene descriptiom./ They have the following fields:

1. The `name` field provides a unique name for the light for debugging purposes.
2. The `pos` field specifies the world-space coordinates of the light.
3. The `direction` field is a vector that specifies the direction of the light that the light is pointing.
4. The `cutoff` field specifies the angle between the light's direction vector and the edge of the light cone (in degrees).
5. The `exponent` field specifies the fall off in intensity from the center of the light's beam.
6. The `intensity` field is an RGB triple that specifies the intensity of the light.
7. The `attenuation` field is a JSON object with three fields — `constant`, `linear`, and `quadratic` — that specifies the attenuation characteristics of the light. These values correspond (respectively) to the parameters c, b, and a from Equation 1 above.

3.2 Rendering

For this project, your renderer should support the following lighting mechanisms:

- Ambient lighting based on a global ambient light level, which is given in the scene description.
- Diffuse lighting based on the diffuse color of surfaces and the lights in the scene.
- Specular lighting based on the specular coefficient and exponent of surfaces and the lights in the scene. You should use the Phong-Blinn model for computing the specular illumination.
- Normal mapping for those surfaces that have a normal map.

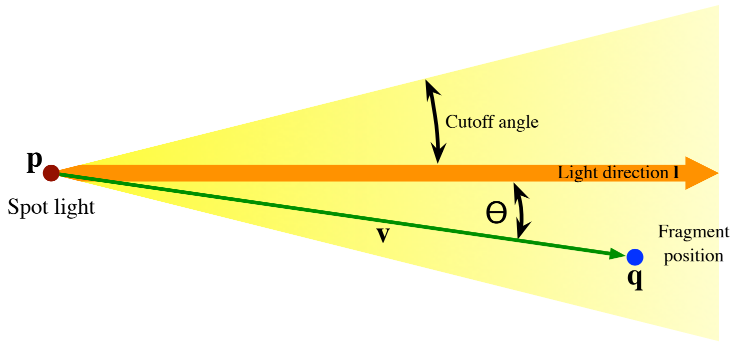
- Texture mapping for those surfaces that have a diffuse or specular texture.

The lights of the scene include a single directional light (as before), plus additional spot lights.

Note that unlike the previous projects, objects vary in their materials, with some objects having textures and others having uniform color. In fact, some objects use different materials for different groups in their model. The sample code supports this variety, but you will need to accommodate it in your shader code.

3.3 Spot lights

The semantics of spotlights is based on the classic OpenGL model. Each spotlight has a number of properties (listed above) that controls how it contributes to the illumination of a fragment. Consider a spotlight at position $\mathbf{p}$ and a fragment at position $\mathbf{q}$. Let $\mathbf{l}$ be the light's direction vector and let $\mathbf{v} = \mathbf{q} - \mathbf{p}$ be the vector from the light to the fragment position as illustrated in the following figure:



The intensity of the light on the fragment is given by the following equation:

$$I = \begin{cases} \mathrm{At}(\|\mathbf{v}\|)(\cos\theta)^e & \text{if } \theta \leq \mathit{cutoff} \\ 0 & \text{if } \theta > \mathit{cutoff} \end{cases}$$

(Recall that $\cos\theta = \frac{\mathbf{v}\cdot\mathbf{l}}{\|\mathbf{v}\|}$.) In the case where θ is greater than the light's cutoff, there is no illumination. Otherwise the illumination is scaled by distance-based attenuation (as defined above) and the cosine of θ raised to the power e. Note also that we can avoid having to compute $\cos^{-1}\left(\frac{\mathbf{v}\cdot\mathbf{l}}{\|\mathbf{v}\|}\right)$ in the shader by precomputing $\cos(\mathit{cutoff})$ for the light.

3.4 User interface

The sample code implements the following controls:

`a A`	toggle the display of the world-space axes
`d D`	switch to deferred-rendering mode
`q Q`	quit the viewer.
`t T`	switch to textured mode
`w W`	switch to wireframe mode
`-`	move the view toward the look-at point
`+`	move the view away from the look-at point
left arrow	rotate the view to the left
right arrow	rotate the view to the right
up arrow	rotate the view up
down arrow	rotate the view down

Deferred-rendering mode is mapped to texturing mode in the sample code, you should modify the `View` object initialization to hook in your rendering code for this mode.

4 Screen-space ambient occlusion

Students who are enrolled in the graduate track (CMSC 33700) should include support for screen-space ambient occlusion. Students who are enrolled in the undergraduate track (CMSC 23700) may add this feature to their projects for extra credit. A separate document will be posted to the project website with information about how to implement this effect, but we will be using a variation of the algorithm described in Chapter 13 of the OpenGL SuperBible (pp. 624–631).

5 Shadow mapping

It is also possible to combine deferred rendering with the shadow mapping technique that you implemented in Project 3. The basic idea is that for each light pass, you add a pass to compute the shadow map from the point of view of the light, which is then used during the lighting pass for the light. One can speed up this process by culling objects outside the light volume when constructing the shadow map. For extra credit, you may add shadow mapping to your implementation (this includes students who are enrolled in CMSC 33700).

6 Sample code

To get you started, we will seed your repository with sample code. The code will be organized into a directory called `proj4` with four subdirectories:

- `build`, which contains a `Makefile` for compiling your project.
- `shaders`, which is where you should put your shader source files.
- `src`, which will hold the C++ source code for your project. We have seeded it with code for loading scenes and for rendering scenes in wireframe and textured modes (*i.e.*, using forward rendering).
- `scenes`, which contains a simple scene for testing purposes. More complex scenes will be posted to the project web page.

7 Hints

- As a preliminary step, you may want to implement standalone renderers to test the shader code for each of the G-buffer components. These renderers would display their output to the screen (as in Figure 1). They will let you verify that your individual computations for the geometry pass are correct before you implement the lighting phases.
- You may also want to add code to render the light volumes in either the wireframe or texture rendering modes as a way to visualize what parts of the scene are affected by which lights.

8 Submission

We have set up an **svn** repository for each student on the `phoenixforge.cs.uchicago.edu` server and we will populate your repository with the sample code. You should commit the final version of your project by 10:00pm on Monday November 16. Remember to make sure that your final version **compiles** before committing!

History

2015-11-05 Fixed a couple of typos in the equations and reduced the size of a couple of the figures to waste less space.

2015-11-04 Original version.