Deferred shading with view volume rendering

Suhail Mohamed

The above image is from *Resident Evil Village (2021)* this scene contains a multitude of lights all interacting with the environment around it, as you can imagine an efficient way of rendering these lights is required to keep the fps of the game reasonable. One technique to allow for this is deferred shading. Deferred shading is a type of shading technique that allows us to render multiple light sources and objects in a scene in an efficient manner. Normally a scene is lit in a 'forward shading' manner, this is a naive approach where we determine the lighting of an object based on all lights in the scene. For example, if a scene contained multiple light sources and multiple objects, each object would have its lighting calculated on it like so:

```
sphere_t sphere_list[NUM_SPHERES];
light_t  light_list [NUM_LIGHTS];
...
for (sphere in sphere_list)
    sphere.render(light_list) ← loop through each light
                                and apply to the sphere
```

Now in cases where the number of objects in the scene is small this method of calculating lighting is fine, but when a scene has many objects, many of which are overlapping, this method is inefficient, as many of the objects in which you calculated this expensive lighting on will end up being partially covered.

What deferred shading does is 'defers' the lighting calculation that we do, instead of calculating the lighting for all objects that get to the vertex shader we only calculate lighting for objects in our final scene, we do this by using off-screen rendering which allows us to attain the data required to light our final scene, from there we only calculate lighting on objects that will be viewable from the camera and nothing else.

View volume rendering is an extra layer of efficiency we can add. When rendering graphics we call the fragment shader many times, if you are drawing any shape you call the fragment shader once for each pixel contained in the area of the shape, this can add up. Let us say we have a point light, for example, a candle and our final non-lit scene, we calculate the effect of this one candle on all the pixels in our final scene…but why? We all know a candle doesn't have a large radius which it affects, like all point lights it attenuates. View volume rendering exploits this by only calculating lighting for all objects that are some radius away from the point light, thus reducing calls to the fragment shader.

Deferred shading with view volume rendering

Suhail Mohamed

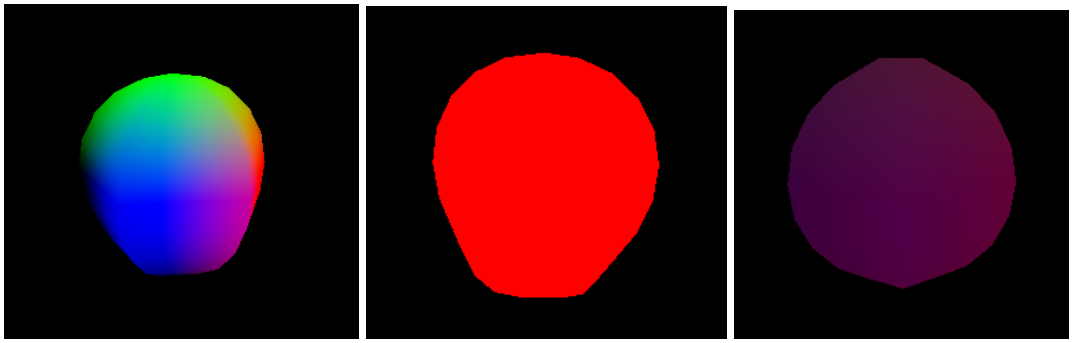
Project Overview:

This project can be broken down into 3 main stages:

1) Creating a framebuffer and loading data into it, this is called the *geometry pass*
2) Doing a stencil test to determine what areas should be rendered, this is called the *stencil test*
3) Rendering the lights using the data from the framebuffer, by this point, the scene will be stencil tested so we will only render objects that should be lit, this is called the *lighting pass*

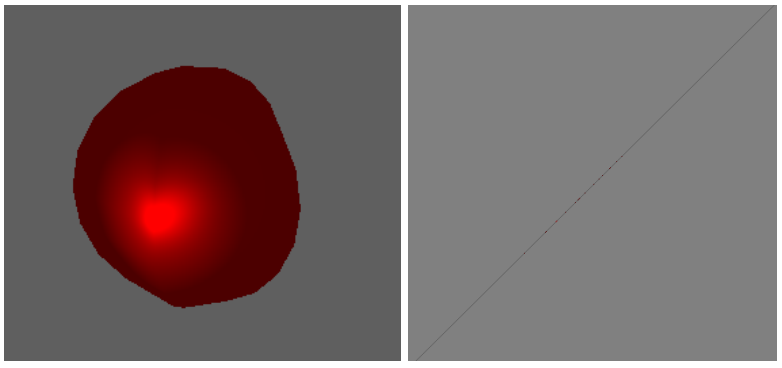
Creating the framebuffer & loading data (Geometry pass):

A framebuffer object (or an FBO) is a structure that holds various textures and a depth/stencil buffer (the stencil buffer isn't required). The key thing about a framebuffer is it allows for off-screen rendering, meaning we can compute where data about a scene would be if we had run it through our graphics pipeline, we can then hold this information in various textures which can be used later on like normal textures, for example in other shaders. This framebuffer is pivotal for deferred shading as it allows us to calculate the normals, colour and positions of the fragments in our final scene after they have been depth tested and interpolated, we can then use these components later on when calculating lighting within our view volumes.



Figure#1: Texture data from the framebuffer being rendered, from left to right, normals, colour and world position / 500

The above textures were all rendered onto a screen-sized quad, this is made apparent when we draw in openGL line mode.



Figure#2: Rendering in line mode we see we are just rendering a texture onto a quad

To create a framebuffer you use `glGenFramebuffers(...)` and bind the different texture buffers to the framebuffer, this is similar to how we bind VBOs to VAOs except with textures. To load data into the textures we have to tell OpenGL what textures we are going to be rendering to in our fragment shader

using the `glDrawBuffers(len(attachments), attachments)`, where the attachments are the different textures you want this data to go into. In my case, I ran `glDrawBuffers(3,f_buffer.attachments)` where `f_buffer.attachments = {COLOR_ATTACH0(position texture), COLOR_ATTACH1(normals texture), COLOR_ATTACH2 (colour texture)}` so my fragment shader loaded the data into the corresponding textures like so:

frame_buffer.frag:

```
in vec3 world_pos;
in vec3 normal;
in vec4 colour;

out vec3 pos_tex;    /* to attachment0 */
out vec3 normal_tex; /* to attachment1 */
out vec4 colour_tex; /* to attachment2 */

void main()
{
  pos_tex    = world_pos;
  normal_tex = normalize(normal);
  colour_tex = colour;
}
```

Determining what we should light (Stencil pass):

A point light releases light in all directions, it also attenuates. This attenuation is often estimated in computer graphics using the equation: $attenuation = 1 / (c + b * distance + a * distance^2)$ where c, b, a are chosen constants, this attenuation factor is then multiplied onto the calculated light to determine how much of the light the object gets. Now with view volumes, we want to have this attenuation be represented by a sphere. To do this though we need to solve for the *distance* component in the previous equation, this will act as the radius for the sphere. First, we will need to set a cut-off value, this value will be the smallest *attenuation* value we will consider, in this project, we will use 1/256:

Note we use I_{max} as it solves for the best radius size, and intensity is the `specularPower` constant defined in `pointLight` struct, this seemed to work well enough.

$$\frac{1}{256} = \frac{I_{max} * intensity}{attenuation}$$

$$\frac{attenuation}{256} = I_{max} * intensity$$

$$attenuation = 256 * (I_{max} * intensity)$$

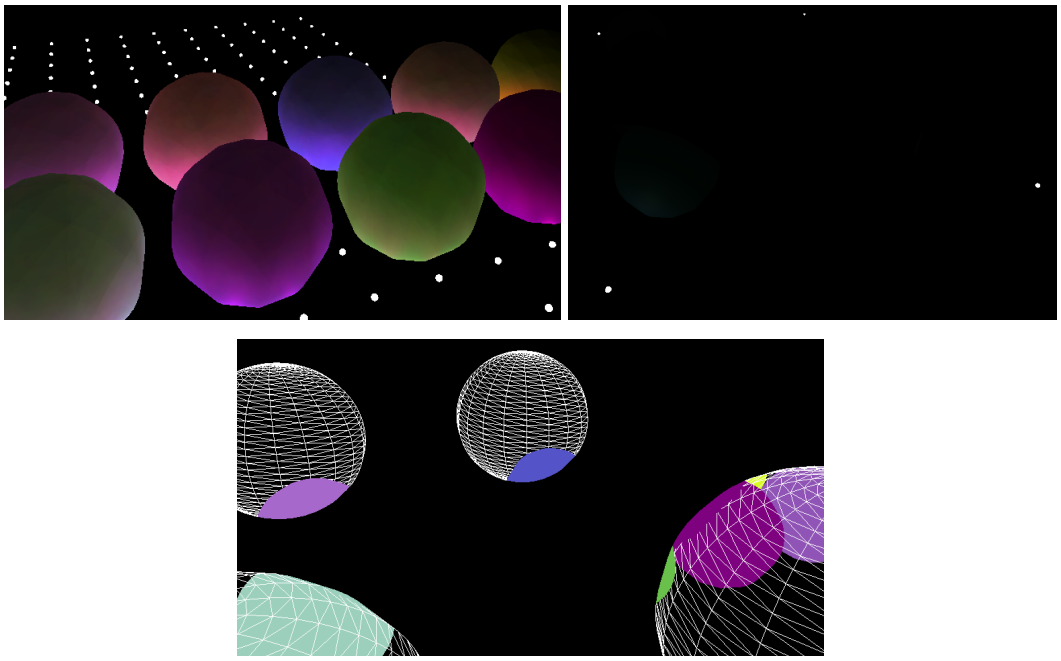
$(c + b * d + a * d^2) - 256 * (I_{max} * intensity) = 0 \leftarrow$ this is a quadratic we can solve

Using the quadratic formula we get:

$$d = \frac{-b + \sqrt{b^2 - 4*a*(c - 256 * c * intensity)}}{2*a}$$

Thus we have a radius we can scale our view volume by, d.

Now we can create spheres whose sizes reflect their attenuation, we can then render these spheres and in their fragment shaders calculate the lighting of the objects by using the textures in the framebuffer.



Figure#3: The first image (from the top left) shows us how it looks when we render lighting calculations from only our view volume spheres, the image after shows us how it looks when we only have corner lights on, and the last image is a debug view of the corner light image, the colours in the view volumes are the sphere objects that reach the fragment shader

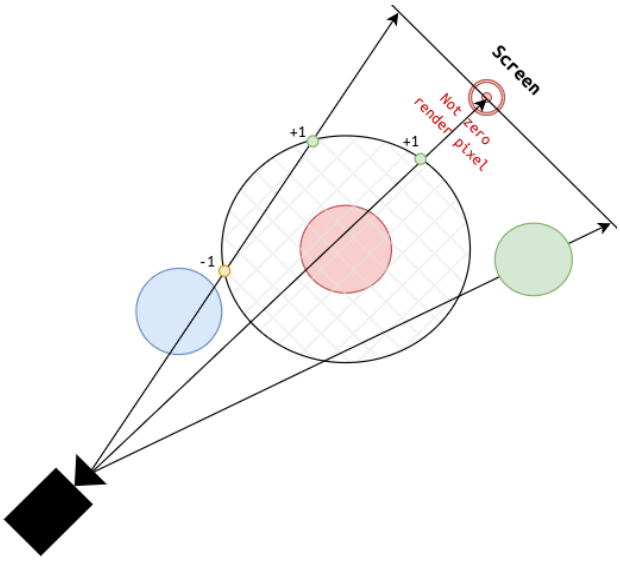
…wait one second why are we rendering the spheres on the corner view volumes, as you can see above there is a large gap between the corner view volumes and the spheres, this is wasted time in the fragment shader as these spheres should have no light interacting with them. We need to find a way to fix this, this is where the stencil test comes in. A stencil in OpenGL is essentially a buffer that determines what will be drawn, like a stencil in real life, it blocks out areas from our screen from being rendered to. It does this by using a stencil operation, a stencil operation is a simple calculation we apply to each screen pixel. Pixels that after this operation has been applied meet some criteria will be rendered. We can use this to only render what is contained within a view volume sphere.

There is a key insight which allows us to create a functioning stencil test, if there isn't a direct line from the front-face → front-face, back-face → back-face or front-face → back-face of the view volume then some object must be in between this area, if this is the case we ought to calculate the lights affect on that area as there is an object contained within the view volume (refer to *Figure#4* to see a visual of this). We can proceed with creating the stencil test using this information, here is how we set it up:

1) Ensure our framebuffers depth buffer has had *scene objects* loaded into it
2) Disable back-face culling, our stencil test relies on doing operations at the front and back faces of our view volume
3) Enable depth testing using the framebuffers depth data and render the view volumes using a null shader, ie: a shader that does not draw anything, we do this so we can initiate depth and stencil testing

4) For stencil operations we will do the following, note we run the stencil operation on each pixel on our screen:

- if our view volume fails the depth test for its *front face*, meaning some scene object is in front of it, we *decrement* our stencil value for that pixel
- If our view volume fails the depth test for its *back face*, we *increment* our stencil value for that pixel, this is needed incase a scene object is completely in front of our view volume we need to undue the previous action we had done (see blue sphere in *Figure#4*)
- We render any areas in the screen where the stencil pixel value is non-zero

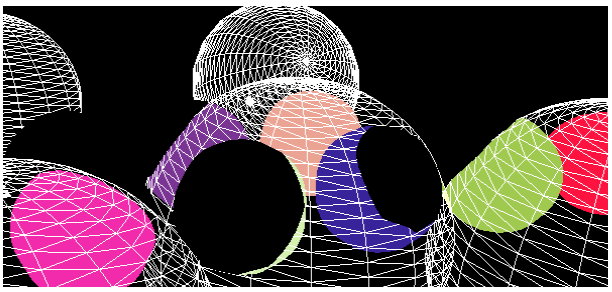


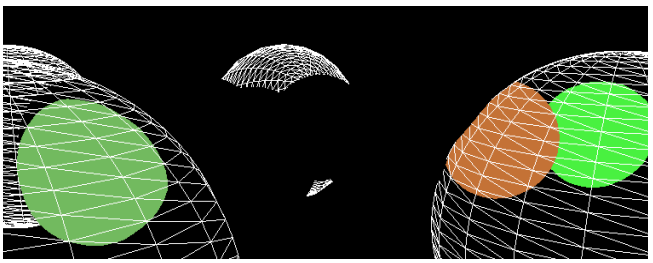
Solution.cpp:

```
shader_null.useProgram(true);

glEnable(GL_DEPTH_TEST);
glDisable(GL_CULL_FACE);
glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS, 0, 0);

/* back faces we increment when depth
test fails (GL_INCR_WRAP) */
glStencilOpSeparate(GL_BACK, GL_KEEP,
GL_INCR_WRAP, GL_KEEP);

/* front faces we decrement when depth
test fails (GL_DECR_WRAP) */
glStencilOpSeparate(GL_FRONT, GL_KEEP,
GL_DECR_WRAP, GL_KEEP);

/* rendering view volume */
light_list[idx].light_view.render(shader
_null);
```

Figure#4: Diagram describing how the stencil operation implemented works, each ray from the camera represents a pixel that we draw to our screen. The code to implement the stencil operation is on the right

Now with this implemented we render only the objects that are contained within view volumes which looks like so:





Figure#5: Showing rendering with the view volume optimization as we can see the only fragments that can get to the fragment shader are ones that are contained within a view volume

Rendering the lights (lighting pass):

The lighting pass for the program was just a normal Blinn-Phong lighting calculation, the only difference about our lighting pass was it was done using textures and we didn't do it while rendering the actual objects, but rather whilst rendering the view volume lighting was calculated. This creates the problem of determining how to get the proper texture samples as our texture values go from (0,0) → (1,1), but our display screen goes from (0,0) → (1024,1024). OpenGL provides the `gl_FragCoord` built-in, which allows us to determine the fragment's position in screen space by accessing `gl_FragCoord.xy`. To convert this to the (0,0) → (1,1) range we simply divide by our screen size.

phong_texture_shading.frag:

```
uniform sampler2D pos_tex;    /*GL_TEXTURE0*/
uniform sampler2D normal_tex; /*GL_TEXTURE1*/
uniform sampler2D colour_tex; /*GL_TEXTURE2*/

...

vec2 frag_tex_coord = calc_tex_coord();

/* getting values from textures */
frag.worldPos = texture(pos_tex,
                        frag_tex_coord).xyz;
frag.normal   = texture(normal_tex,
                        frag_tex_coord).xyz;
frag.colour   = texture(colour_tex,
                        frag_tex_coord);

frag_colour = calcPointLight(gMaterial,
                             gPointLight,
                             frag);
```

phong_texture_shading.frag:

```
/* determining texture coordinates */
vec2 calc_tex_coord()
{
      return gl_FragCoord.xy / g_wind_size;
}
```

Deferred shading with view volume rendering

Suhail Mohamed

Special Instructions:

w, s	Move forward/backward
a,d	Move up/down
Arrow keys	Change what the camera is looking at
r	Rotate camera
v	See view volumes
c	Light corners only
u	See sphere colours, no lighting
p,o, i	In order: toggle ambient, diffuse & specular light
q	Quit

References:

Research:

https://learnopengl.com/Advanced-Lighting/Deferred-Shading
https://learnopengl.com/Advanced-OpenGL/Framebuffers
https://learnopengl.com/Advanced-OpenGL/Stencil-testing
https://ogldev.org/www/tutorial37/tutorial37.html
https://www.youtube.com/watch?v=9_v8cvd-BSQ&t=3442s

Figures:

Image on page #1 from: Resident Evil Village (2021)
Figure#7: https://www.slideshare.net/guerrillagames/deferred-rendering-in-killzone-2-9691589 (slide 18)
Figure#8: http://c0de517e.blogspot.com/2020/12/hallucinations-re-rendering-of.html
Figure#9: https://github.com/austinEng/WebGL-Deferred-Shading