DigiPen
INSTITUTE OF TECHNOLOGY

CS300
Deferred Shading

Lighting Review

- In computer graphics, the lighting and material (surface) model is divided into four independents components.
- Each component is calculated independently and added together at the end.
- They are:
  a) Ambient
  b) Diffuse
  c) Specular
  d) Emissive

Lighting Review

- Ambient component ($I_{ambient} = I_a K_a$)
- Diffuse component ($I_{diffuse} = I_d K_d Max(N.L, 0)$)
- Specular component ($I_{specular} = I_s K_s Max(R.V, 0)^{ns}$)
- Emissive component ($I_{emissive}$)
- Attenuation factor ($Att = \min(\frac{1}{c_1 + c_2 d_L + c_3 {d_L}^2}, 1.0)$)
- Spotlight Effect ($SpotlightEffect = \left(\frac{\cos(Alpha) - \cos(Phi/2)}{\cos(Theta/2) - \cos(Phi/2)} \right)^P$)

Lighting Review

- Total light intensity on the object surface

$$I_{tot} = I_{emissive} + I_{globalAmbient} K_a + Att * I_{ambient} + Att * SpotlightEffect * (I_{diffuse} + I_{specular})$$

- Total lights intensity for multiple lights:

$$I_{tot} = I_{emissive} + I_{globalAmbient} K_a + \sum_{i=0}^{n-1} Att_i * (I_{ambient})_i + Att_i * SpotlightEffect_i * (I_{diffuse} + I_{specular})_i$$

Lighting Review

- Alternative method to calculate the specular component is by using the normal and the half-vector.
- The half-vector is the normalized vector that bisect the angle between the light and view vector.

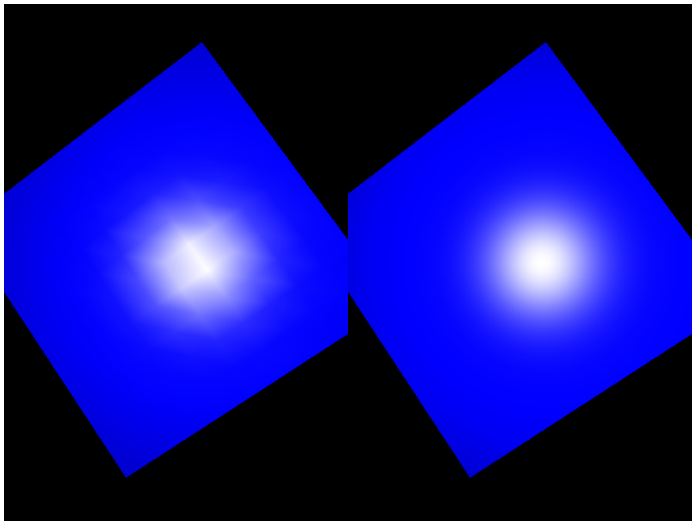
$$I_{specular} = I_s K_s Max(N.H, 0)^{ns}$$

- Where:
  - I_s is the light specular intensity
  - K_s is the material's specular reflection coefficient
  - N is the surface normal
  - H is the half vector
  - ns is the index that controls the shininess or the tightness of the specular highlight, determined by the surface properties.

Lighting Implementation

- Two methods to do the lighting calculation:
  - per-vertex: cheaper than per-pixel, but it requires a lot of vertices to have a good result.
  - per-pixel: more expensive, but the result is much better even when the polygon count is low.

The quad on the left is subdivided into 200 triangles and it uses per-vertex lighting.



The quad on the right has only 2 triangles, but it uses per-pixel lighting. Notice that the highlight is much smoother.

Lighting Implementation Steps

- Initialize the light(s) and material parameters:
  - Light position
  - Light intensity (ambient, diffuse, specular)
  - Light attenuation factor
  - Light spot light parameters (if light is a spot light)
  - Material properties (ambient, diffuse, specular, shininess, emmisive)

Lighting Implementation Steps

- Calculate required variables:
  - N – the surface normal at the vertex/pixel
  - P – the vertex/pixel position in **camera space**
  - L – the vector from the vertex/pixel to the light position in **camera space**
  - V – the vector from the vertex/pixel to the eye/camera position.
  - H – the half vector (H = (L+V) / |L+V|)
  - D – distance from the vertex/pixel to the light position (for the attenuation factor calculation)

Lighting Implementation Steps

- Use the lighting equation to calculate the final light color for that vertex/pixel.
- If atmospheric attenuation is used, interpolate the final vertex/pixel color to the atmosphere color.

Real-time Lighting

- Modern games use many lights on many objects covering many pixels.
- These could get expensive.
- Three techniques used in real-time lighting:
  - Single pass lighting
  - Multi pass lighting
  - Deferred shading

Single Pass Lighting

- Algorithm:

```
For each object
    For each light
        frameBuffer = lightModel(light,object)
```

Single Pass Lighting

- Good for scenes with small numbers of lights (eg. outdoor sunlight)
- Wasted calculations for hidden surfaces
- Difficult to organize if there are many lights
- Easy to overflow shader length limitations
- Difficulty in implementing shadows
  - Shadow maps can easily overflow VRAM since each light needs its own shadow map.

Multi Pass Lighting

- Algorithm:

```
For each light
    For each object affected by light
        frameBuffer += lightModel(light,object)
```

Multi Pass Lighting

- Worst case complexity is num_objects * num_lights
- Sorting by light or by object are mutually exclusive:
  - hard to maintain good batching
  - Like in single pass, you can have wasted calculations for hidden surfaces.
- Ideally the scene should be split exactly along light boundaries, but getting this right for dynamic lights can be a lot of CPU work

Deferred Shading

- Algorithm:

```
For each object
    G-Buffer = lighting properties of the object
For each light
    frameBuffer += lightModel(light,G-buffer)
```

Deferred Shading

- Improves engine management.
- Small complexity (good).
- Integration with popular shadow techniques (shadow mapping, PCF, etc).

Deferred Shading Pros

- Many small lights are just as cheap (or expensive) as a few big ones.
- Very easy and simple to make.
- Very tight boundaries on what pixels the light volumes affect (i.e. not many pixels, if any, are wasted).
- Scales very nicely as number of lights increases.
- Worst case complexity depends on number of object and light.
  - O(number_of_objects + number_of_lights)

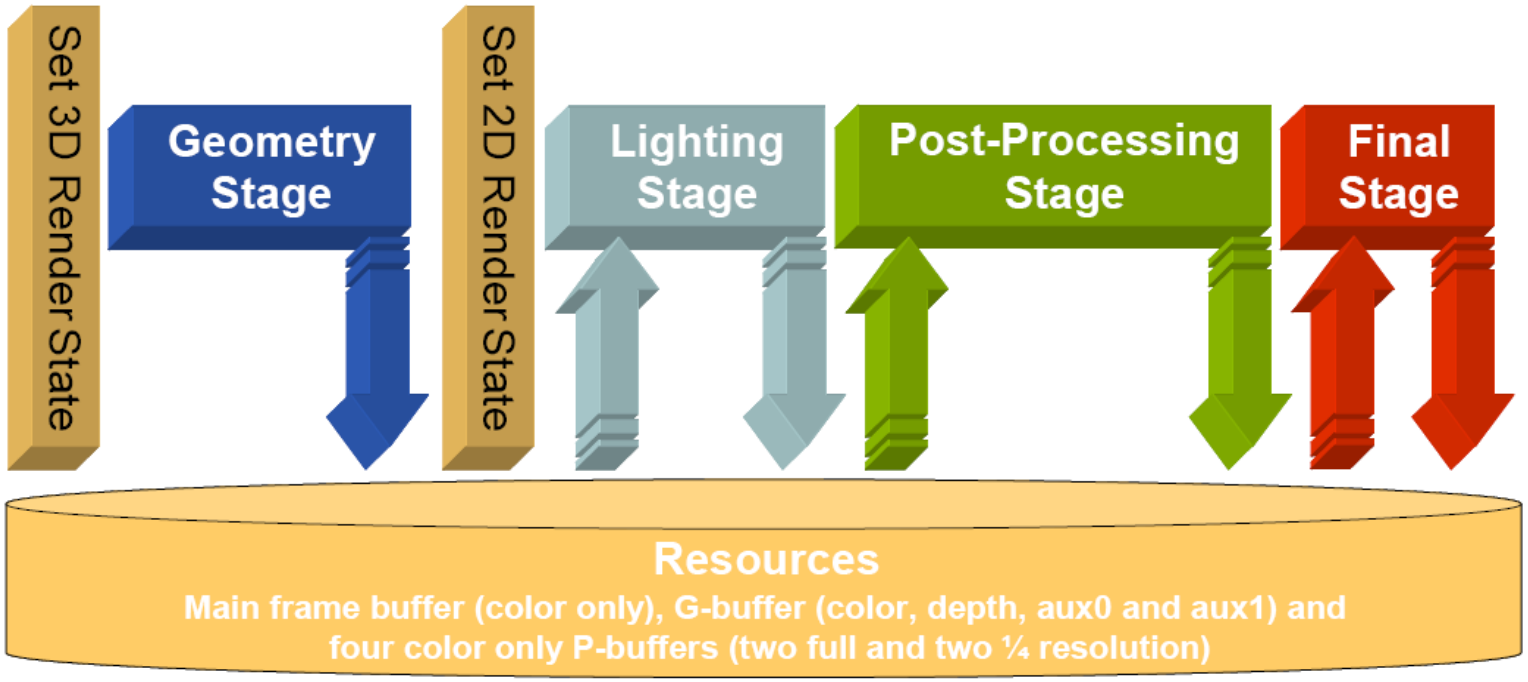
Deferred Shading Cons

- Difficult, but not impossible, to support a reasonably wide variety of materials.
  - Limited number of terms in G-buffer
- The material used in the scene need to behave the same.
- No hardware anti-aliasing.
- Requires a lot of memory, especially at higher screen resolution.
- Cannot handle translucent object.

Deferred Shading Pipeline

- Geometry stage
- Lighting stage
- Post-processing stage
- Final stage

Deferred Shading Pipeline



* G-Buffer = Geometry Buffer
  P-Buffer = Pixel Buffer

Pipeline Explanation

- Geometry stage: feeds the G-buffer with the information needed to be used in next stages
- Lighting stage: receives as input the contents of the G-buffer and accumulates lighting into a full resolution P-buffer
- Post-processing stage: passes are performed in order to enhance the image generated previously
- Final Stage: the enhanced image passes to the main frame buffer to be displayed on the screen

Pipeline Explanation

- Each stage communicates with the other ones through a shared memory area (the resources) which is located in the video memory of the graphic card.

- These resources are organized as buffers.

Resources

- Render targets

  When a stage writes information to the video memory, it writes them to the render targets.

- Textures

  When a stage reads information from the video memory, it reads them from the textures.

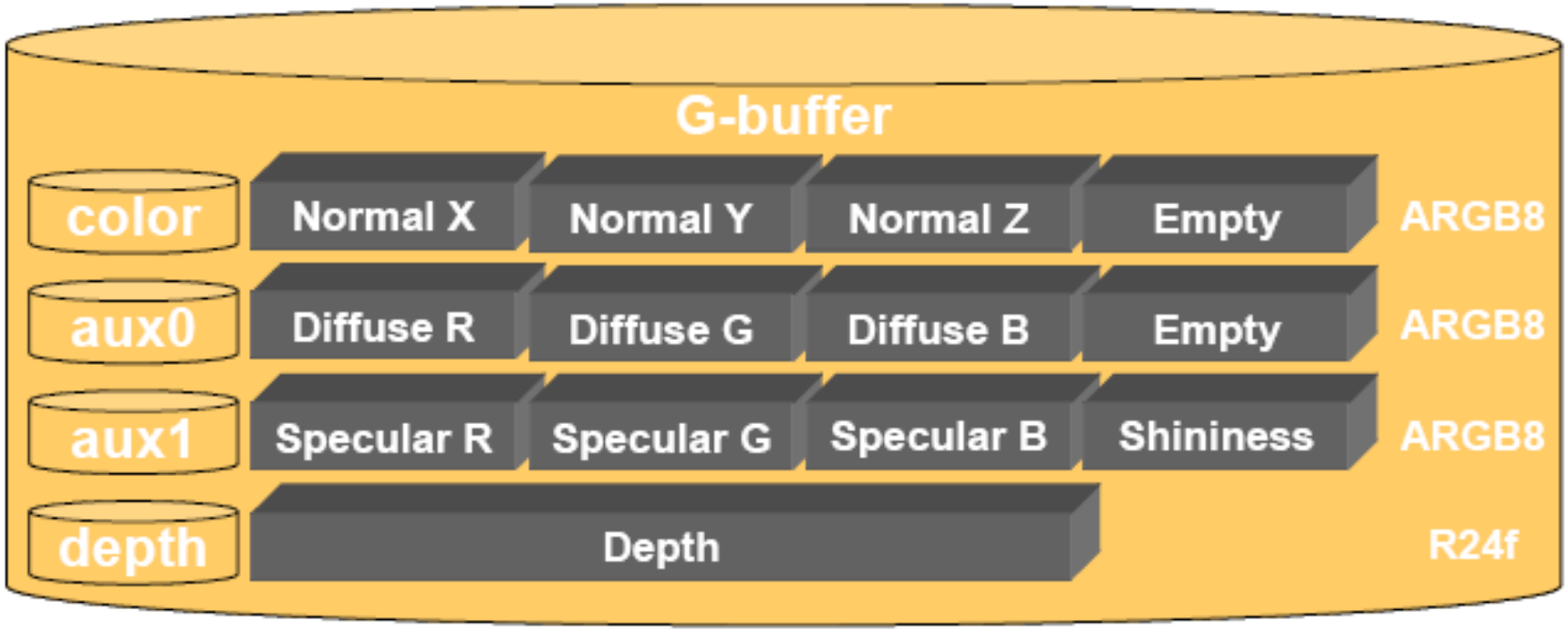
G-Buffer

- What is it?
  - A storage area where we store the scene geometric attributes. Due to the amount of data to be stored, it is composed of multiple render targets.
- What attributes does it store?
  - Position
  - Normal
  - Material parameter (ambient, diffuse, specular, emissive, shininess)

G-Buffer

- All render targets used in the G-Buffer will have same resolutions.
- Amount of memory needed to store information for just a pixel:
  - 4 (depth value) + 3 (normal) + 3 x 4 (ambient, diffuse, specular, emissive) + 1 (shininess) = 20 bytes
- Amount of memory needed for a 1280x768 screen resolution:
  - 1280 x 768 x 20 = 19,660,800 bytes (about 20 MB)

G-Buffer



Example - Killzone

G-Buffer Note

- The value stored in the depth buffer is a normalized depth value, i.e. it is not the one after the perspective divide.
  - [near, far] -> [0, 1]

- In our example, we need 4 render targets and 1 depth buffer to store all the G-Buffer values.

G-Buffer Note

- Geometry data can be shuffled around in the render targets.
- Make sure the hardware can support the number of render target needed.
- Need to balance the memory consumption and precision.

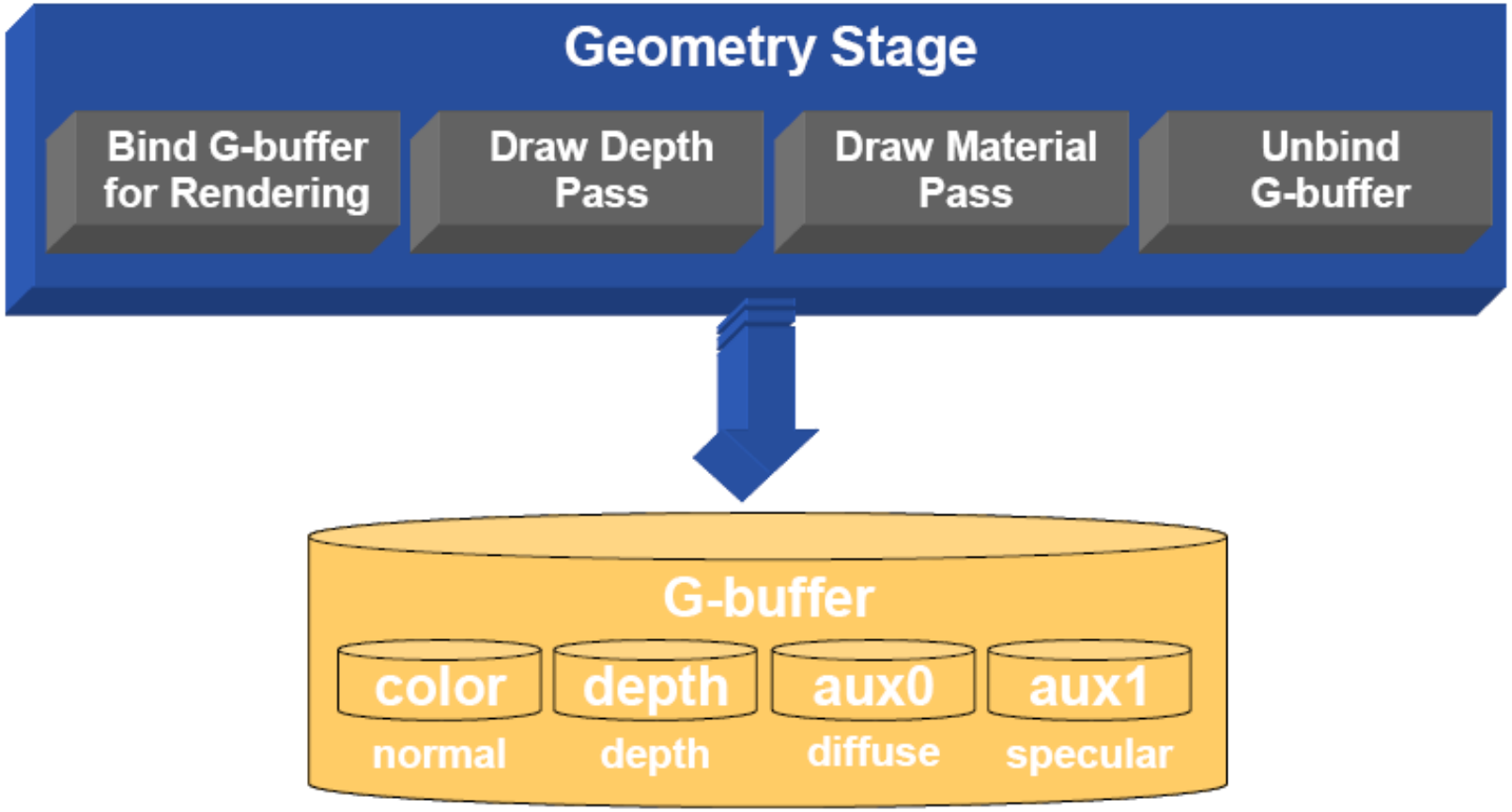
Geometry Stage

- Implement camera perspective projection in 3D space.
- Feed the G-Buffer with information required by the next stage (the lighting stage).
- Deals with the geometric data.

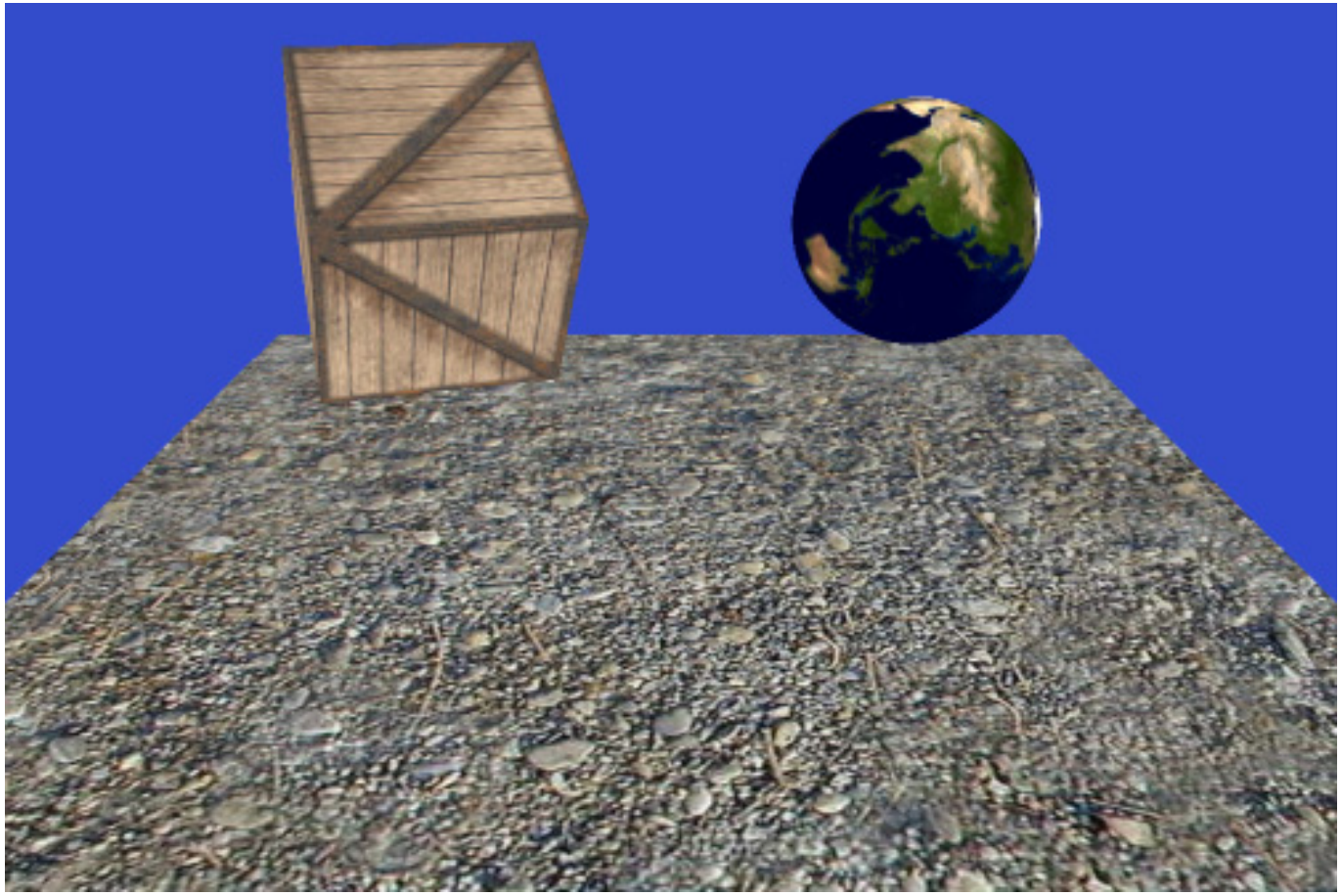
Geometry Stage

- Input:
  - The mesh of the scene to be render and its associated material property.
- Output:
  - The G-Buffer filled with information required to shade all pixels to generate the final image.

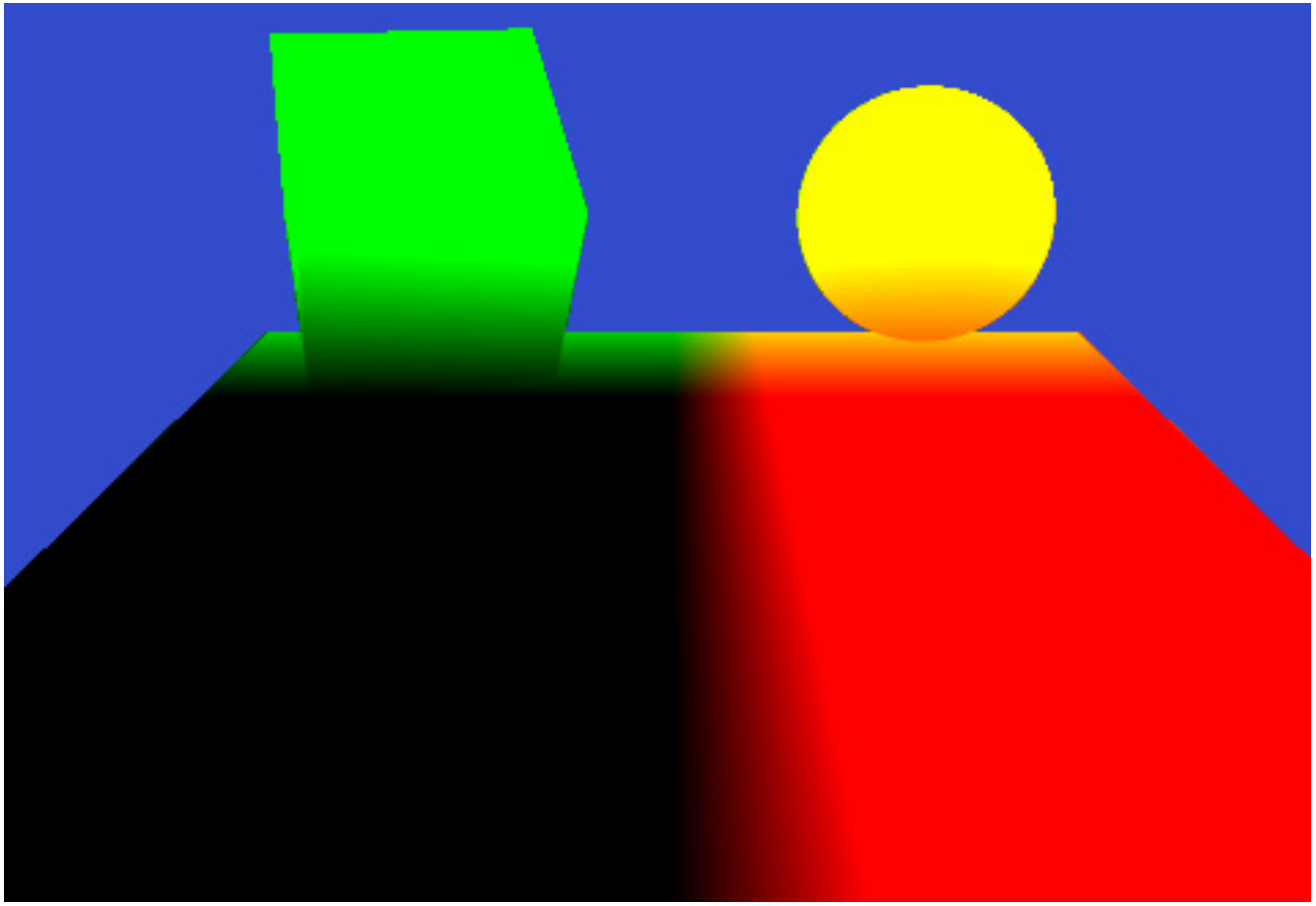
Geometry Stage



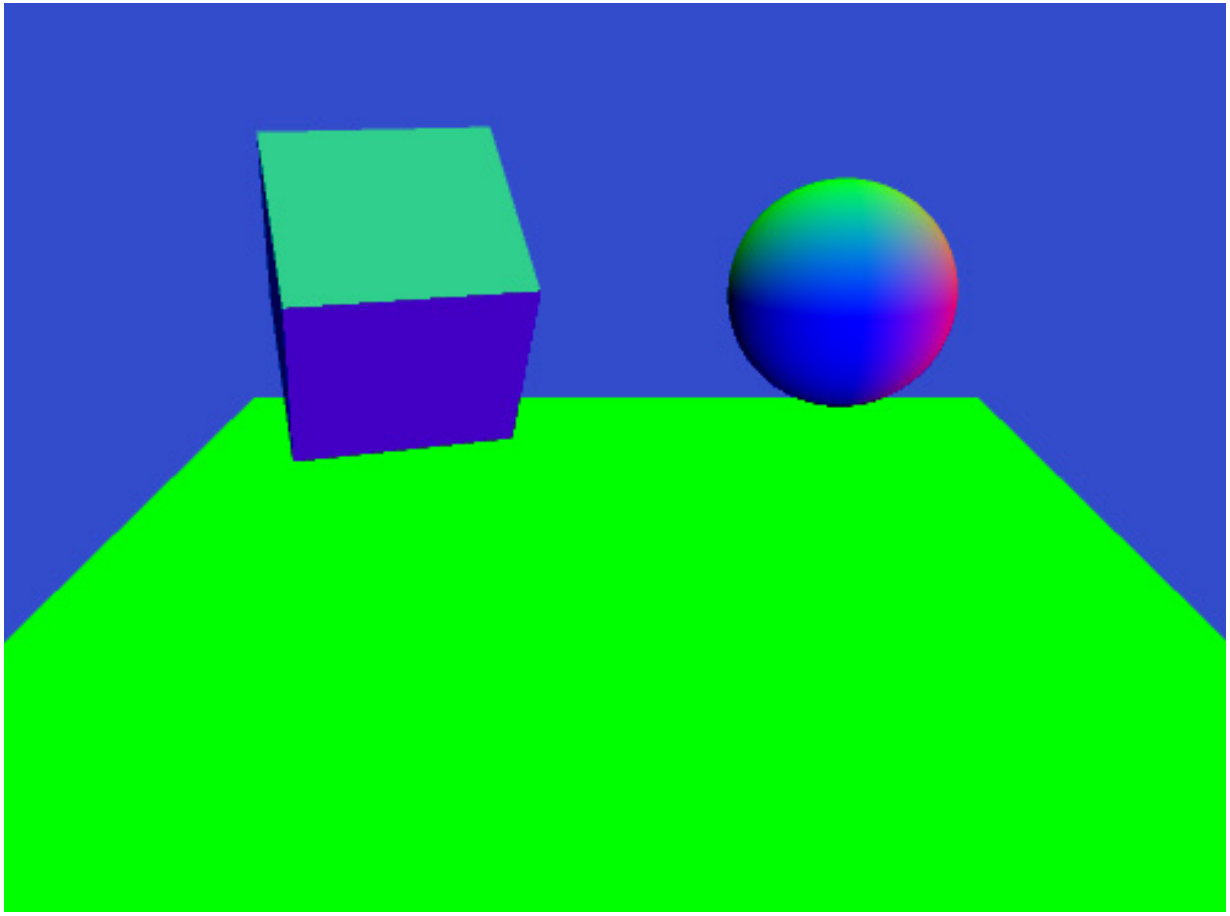
Target 0 - Diffuse



Target 1 (depth) – Eye Space Position



Target 2 – Normal (in eye space)



Geometry Stage

- Pseudo code:
  - Setup the render targets for the G-buffer.
  - Render the scene without lighting.
  - Copy the render targets to texture maps.
    - If the API supports rendering to texture map directly, this step can be avoided.

Lighting Stage

- Accumulate contribution from each light in the scene into the P-buffer (pixel buffer).
- Combine the result from the G-Buffer and the resulting P-Buffer to generate the semi-final image.
- If no post-processing is required, the result of this stage is the final image

Lighting Stage

- Input:
  - The content of G-buffer as texture maps.
  - The parameters for each light in the scene.

- Output:
  - The accumulated light intensity in P-buffer.
  - Each component of the lights (ambient, diffuse, specular) will be accumulated separately.

Lighting Stage

- Pseudo code:
  - Setup the G-buffer as input texture maps.
  - Loop through each light in the scene.
  - For each light,
    - Pass the light parameters (position, direction, color parameters, bounding sphere, etc) to the shader.
    - Render a screen aligned rectangle (the shader will accumulate the light intensity).
  - Copy the render targets to texture
  - Add the light accumulation textures to get the final (or semi-final) image.

```
/**
*    Start rendering to the texture
*    Both color and depth buffers are cleared.
*/
void FBORenderTexture::start(){
    // Bind our FBO and set the viewport to the proper size
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, m_fbo);
    glPushAttrib(GL_VIEWPORT_BIT);
    glViewport(0,0,m_width, m_height);

    // Clear the render targets
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glClearColor( 0.0f, 0.0f, 0.0f, 1.0f );

    glActiveTextureARB(GL_TEXTURE0_ARB);
    glEnable(GL_TEXTURE_2D);

    // Specify what to render an start acquiring
    GLenum buffers[] = { GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT,
                         GL_COLOR_ATTACHMENT2_EXT };
    glDrawBuffers(3, buffers);
}

/**
*    Stop rendering to this texture.
*/
void FBORenderTexture::stop(){
    // Stop acquiring and unbind the FBO
    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
    glPopAttrib();
}
```

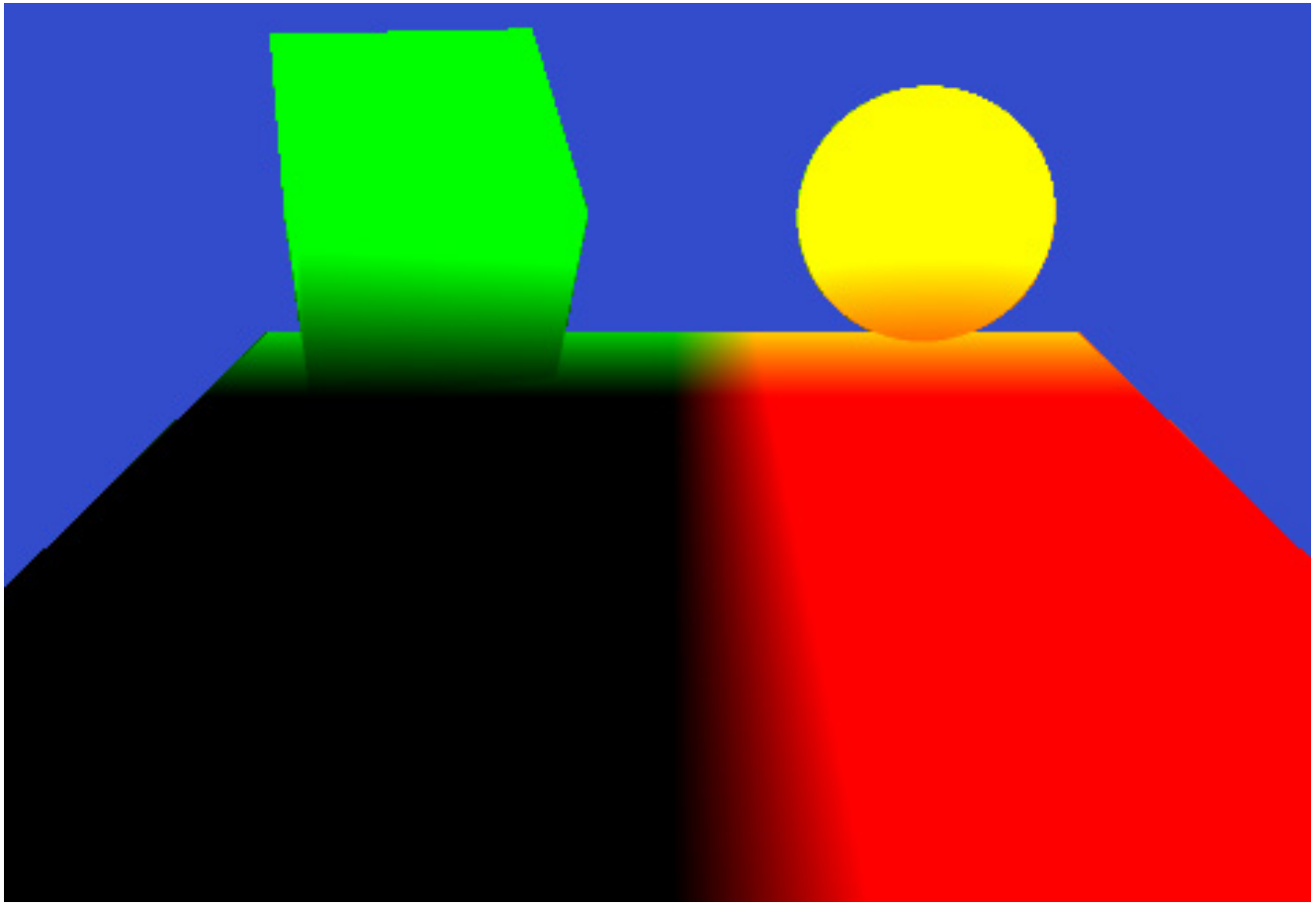
Geometry Stage Vertex Shader

```
#version 330

layout (location = 0) in vec3 Position;
layout (location = 1) in vec2 TexCoord;
layout (location = 2) in vec3 Normal;

uniform mat4 gWVP;
uniform mat4 gView;

out vec2 TexCoord0;
out vec3 Normal0;
out vec3 ViewPos0;

void main()
{
    gl_Position = gWVP * vec4(Position, 1.0);
    TexCoord0 = TexCoord;
    Normal0 = (gView * vec4(Normal, 0.0)).xyz;
    ViewPos0 = (gView * vec4(Position, 1.0)).xyz;
}
```

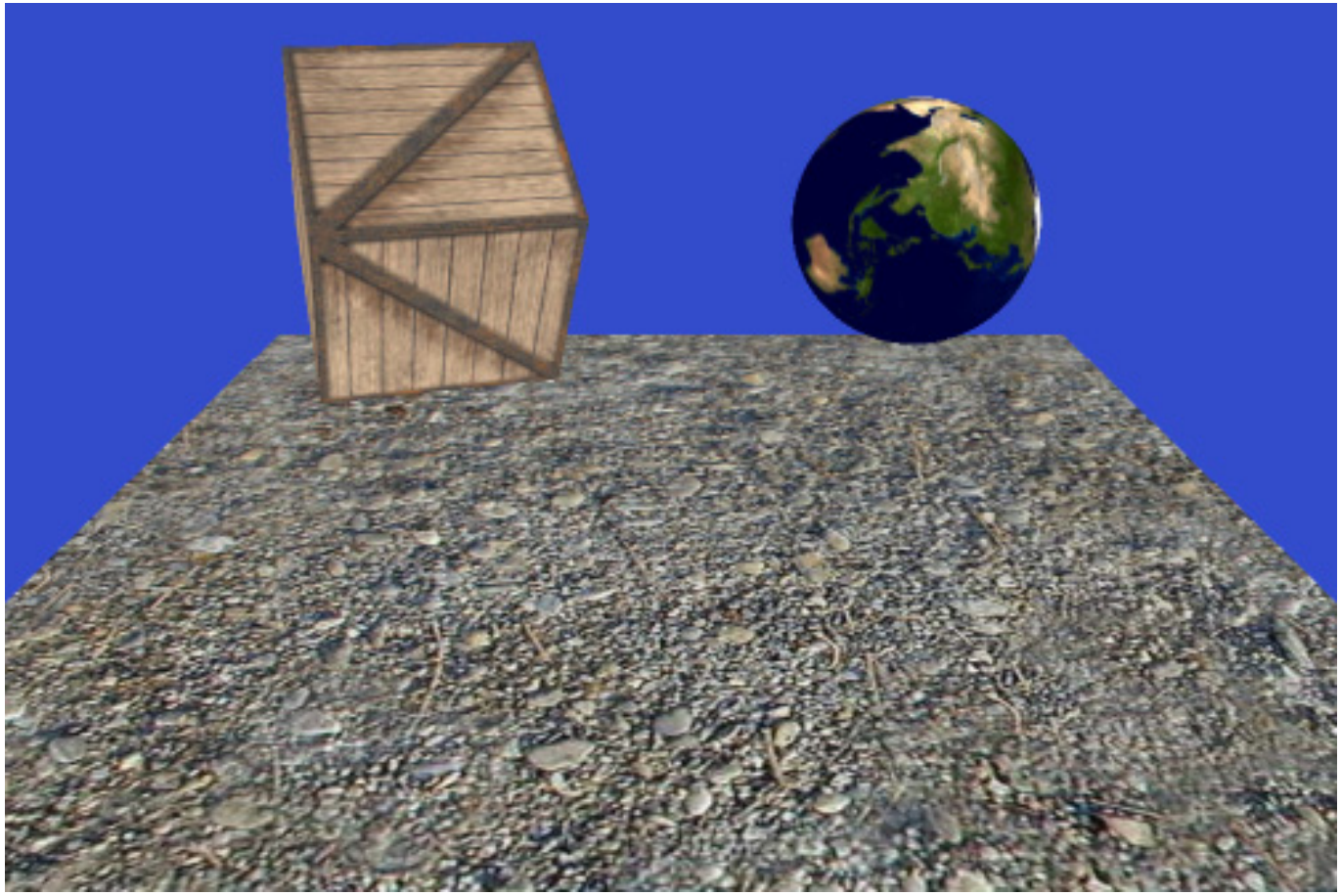
Geometry Stage Fragment Shader

```
#version 330

in vec2 TexCoord0;
in vec3 Normal0;
in vec3 ViewPos0;

layout (location = 0) out vec3 ViewPosOut;
layout (location = 1) out vec3 DiffuseOut;
layout (location = 2) out vec3 NormalOut;
layout (location = 3) out vec3 TexCoordOut;

uniform sampler2D gColorMap;

void main()
{
    ViewPosOut = ViewPos0;
    DiffuseOut = texture(gColorMap, TexCoord0).xyz;
    NormalOut = normalize(Normal0);
    TexCoordOut = vec3(TexCoord0, 0.0);
}
```

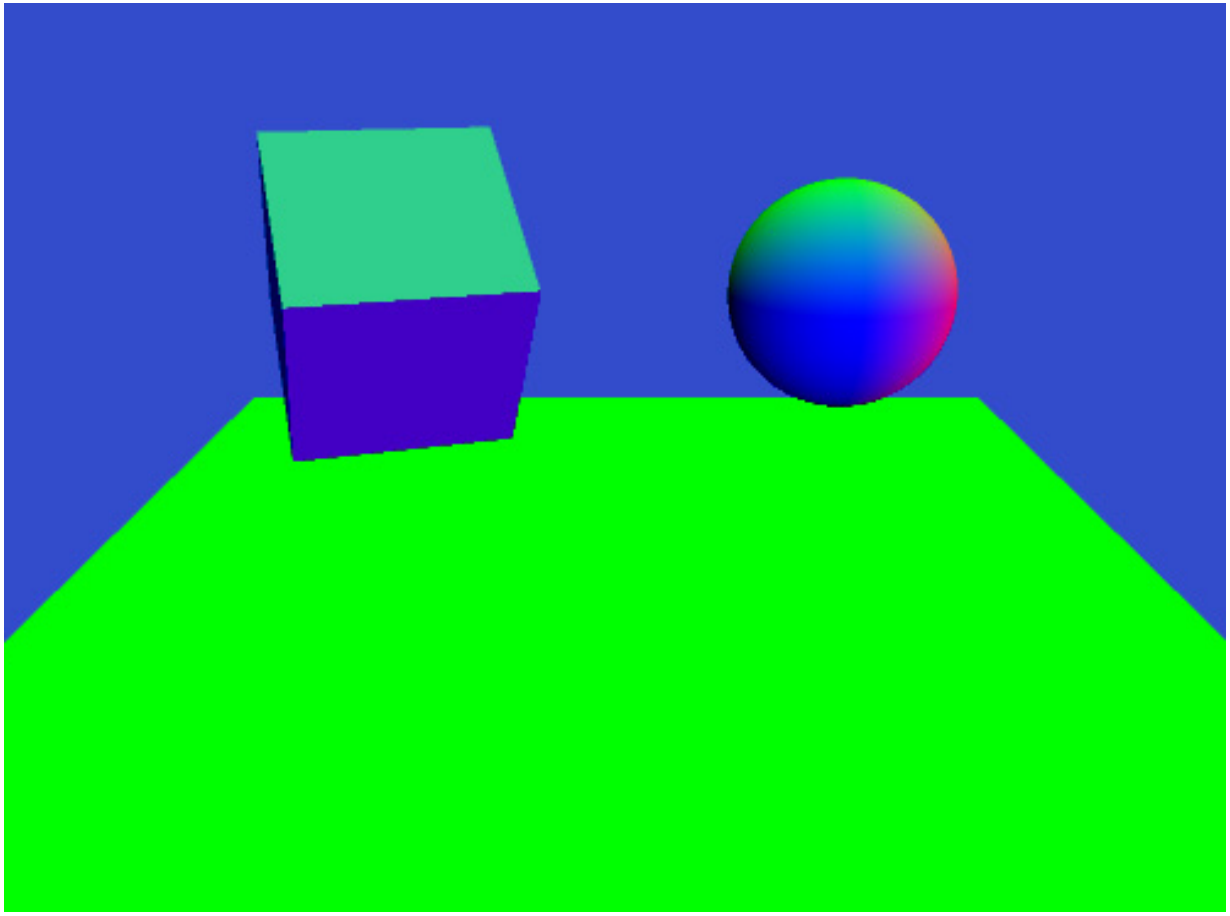
layout (location = 0) View Position



layout (location = 1) Diffuse color



layout(location = 2) Normal (in eye space)



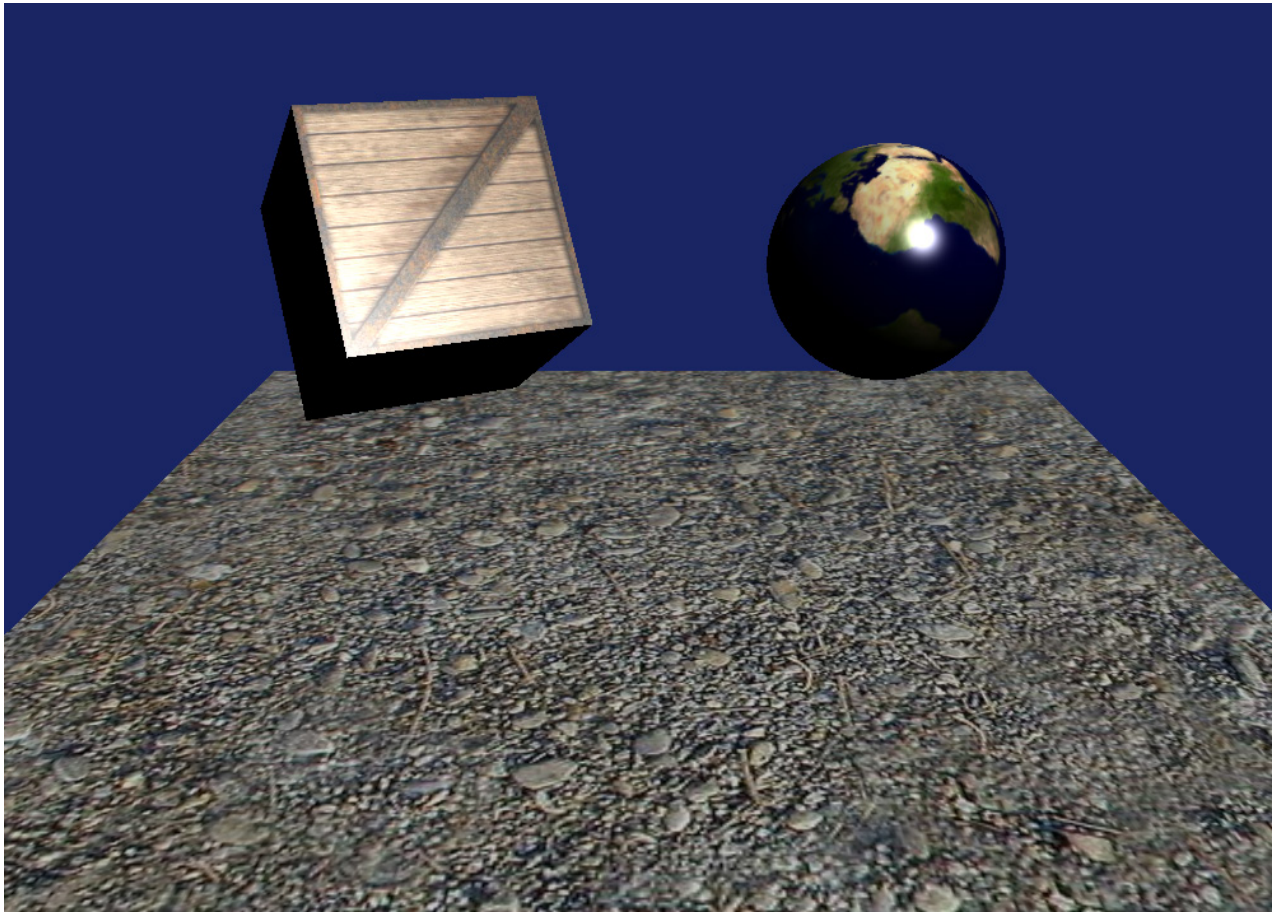
Deferred Rendering Vertex Shader

```
#version 330

layout (location = 0) in vec3 Position;

uniform mat4 gWVP;

void main()
{
        gl_Position = gWVP * vec4(Position, 1.0);
}
```

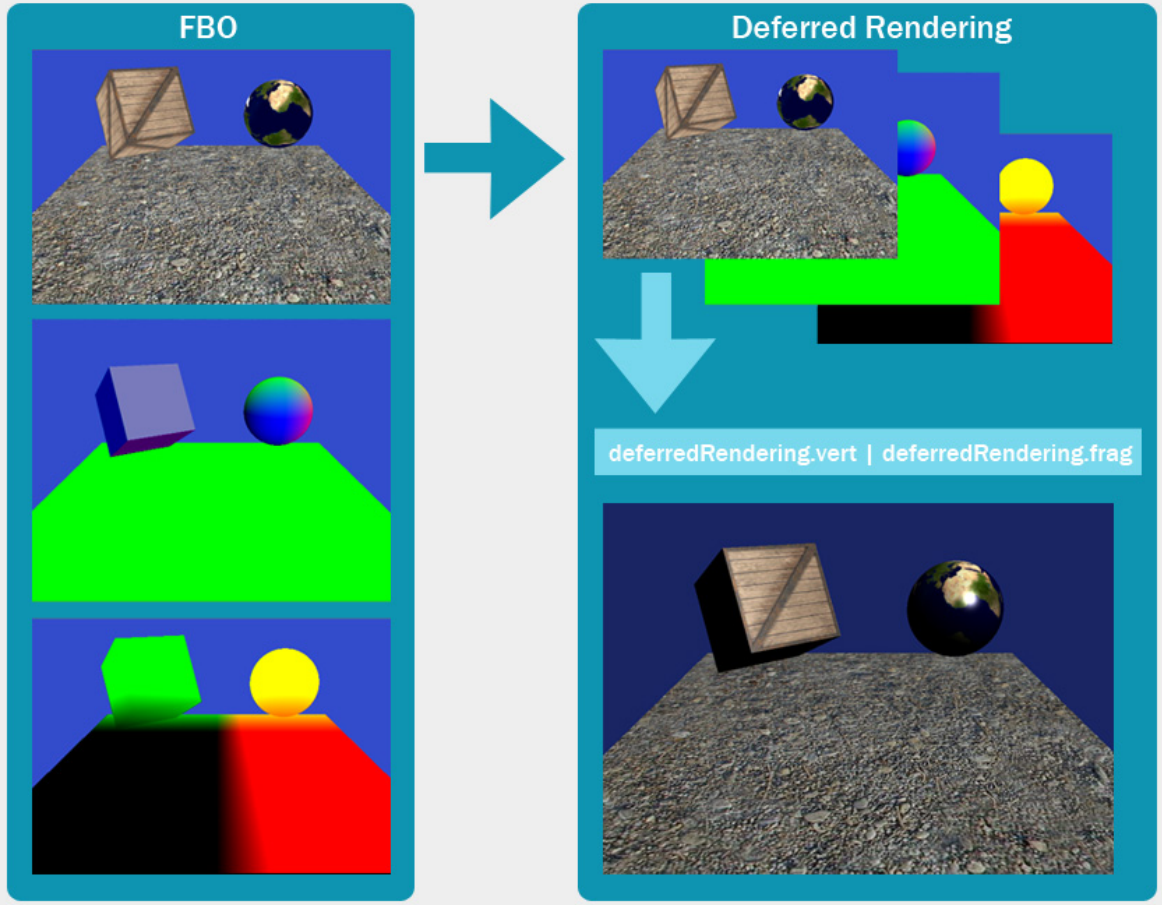
Deferred Rendering Fragment Shader

```
void main()
{
        vec2 TexCoord = CalcTexCoord();
        vec3 ViewPos = texture(gPositionMap, TexCoord).xyz;
        vec3 Color = texture(gColorMap, TexCoord).xyz;
        vec3 Normal = texture(gNormalMap, TexCoord).xyz;
        Normal = normalize(Normal);

        FragColor = vec4(Color, 1.0) * CalcPointLight(ViewPos, Normal);
}
```

Lighting Output



Overall Pipeline



Lighting Stage Optimization

- Rendering a screen size rectangle for each light in the scene could get really expensive.
- For example, if we have 8 lights in the scene and the screen in set to 1280x768, the pixel shader will be executed 1280x768x8 = 7864320 times per frame.
- Reducing the rectangle size will reduce the number of pixels for each light.

Lighting Stage Optimization

- How to reduce the rectangle size?

Lighting Stage Optimization

- How to reduce the rectangle size?
  - Calculate the bounding rectangle in the screen space of the light sphere of influence.
  - Enable the scissoring test and set it to the calculated bounding rectangle.

Lighting Stage Optimization

- Instead of using the scissoring test, you can just render the rectangle in screen space.

Lighting Stage Optimization

- How to calculate the bounding rectangle in the screen space?

Lighting Stage Optimization

- Method 1:
  - Calculate the 8 vertices of the bounding box of the light sphere.
  - Project these vertices to the screen.
  - Generate bounding rectangle for the projected vertices.

Lighting Stage Optimization

- Method 1:
  - Calculate the 8 vertices of the bounding box of the light sphere.
  - Project these vertices to the screen.
  - Generate bounding rectangle for the projected vertices.

- Some issues with method 1:
  - The resulting rectangle will be larger than necessary.
  - Might fail when some vertices are behind the camera.

Lighting Stage Optimization

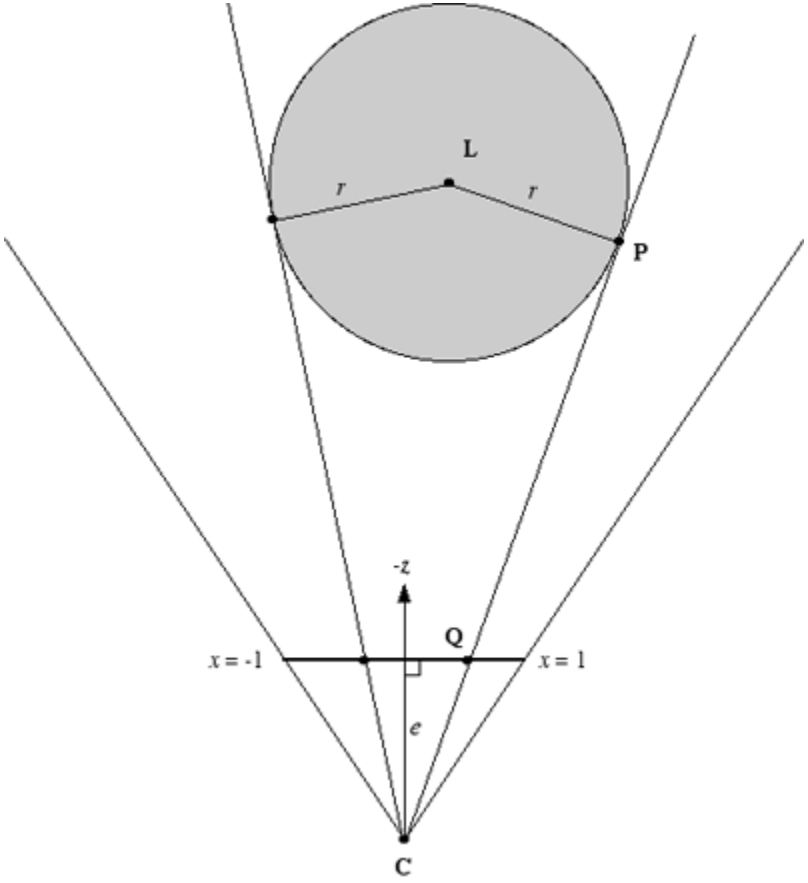
- Method 2:
  - Calculate 4 planes that pass the camera (the origin in the camera space) and are tangent to the sphere.
  - 2 of the planes are parallel with the x-axis. The other 2 are parallel with the y-axis.

- Before calculating the planes:
  - If the camera is within the light sphere then the light affects the whole viewport.
  - If the light sphere is outside the frustum then the light does not affect any pixels on the viewport.

Lighting Stage Optimization

- The diagram on the right shows the planes that are parallel with the y-axis.

- Where:
  - L is the light position
  - r is the light sphere radius
  - c is the camera position (the origin in camera space)



Lighting Stage Optimization

- For the 2 planes that parallel with the y-axis
  - General plane equation: $Ax + By + Cz + D = 0$
  - $D = 0$ because the plane pass the origin.
  - $B = 0$ because the plane parallel with the y-axis.
  - $A.L_x + C.L_z = r$ because the light position is a distance 'r' from the planes.
  - $A^2 + C^2 = 1$ because the normal length is 1.

- Solving for A and C will result in the 2 planes that are parallel with the y-axis.

Lighting Stage Optimization

- The equations of the 2 planes that parallel with the y-axis:

$$Ax + By + Cz + D = 0$$

Where:

$$A_0=(rL_x+\mathrm{sqrt}(r^2L_x^2-(L_x^2+L_z^2)(r^2-L_z^2)))/(L_x^2+L_z^2)$$
$$A_1=(rL_x-\mathrm{sqrt}(r^2L_x^2-(L_x^2+L_z^2)(r^2-L_z^2)))/(L_x^2+L_z^2)$$
$$B_{0,1}=0$$
$$C_{0,1}=(r-A_{0,1}L_x)/L_z$$
$$D_{0,1}=0$$

Lighting Stage Optimization

- Similarly, the planes that are parallel with the x-axis can be calculated.

- Once you have the planes equations, intersecting them with plane z = -1 and taking into account the camera field of view, aspect ratio and viewport setting will allow you to calculate the bounding rectangle in the screen space.

Lighting Stage Optimization

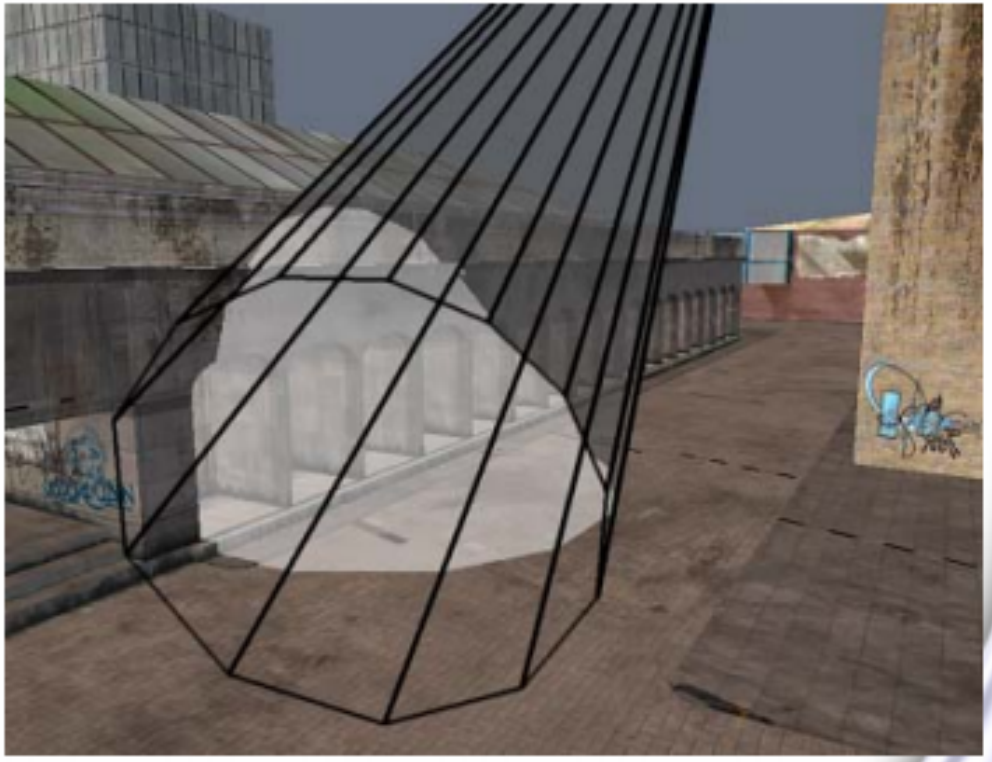
- Putting the steps together:
  1. Calculate the light position and the radius of light sphere of influence.
  2. Check if the camera is within the light sphere.
  3. Do frustum culling for the light sphere.
  4. Calculate the 2 planes equations that are parallel with the y-axis and determine the order (left/right).
  5. Intersect the tangent planes with plane z = -1 and take into account the field of view, aspect ratio and viewport setting to calculate the left and right of the bounding rectangle in screen space.
  6. Repeat 4 and 5 for the planes that are parallel with the x-axis.
  7. Set the scissor setting using the calculated bounding rectangle.
  8. Render the screen size quad.

Lighting Stage Optimization

- Alternative to rendering screen size quad (or a smaller one) for each light, render a geometry that bound the light volume:
  - Sphere for point light
  - Cone for spot light
  - Cylinder (or screen size quad) for directional light

- Make sure that geometry completely contains the actual light volume.

Lighting Volumes



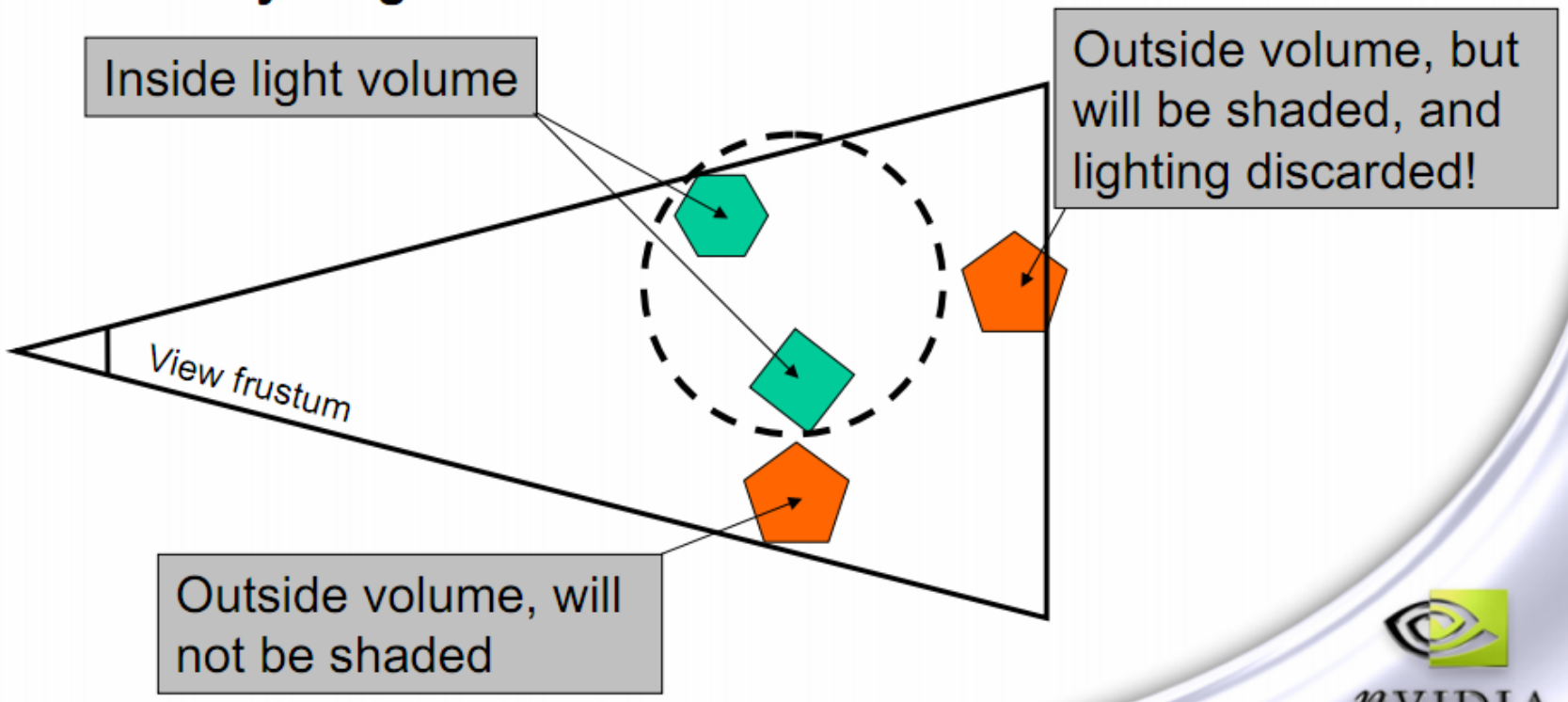
Courtesy of Shawn Hargreaves, GDC 2004

Lighting Stage Optimization

- The naïve method is to render the light volume geometry normally.
- Unfortunately, in most cases, we are rendering more pixels than necessary, i.e. a lot of wasted lighting calculation.
- Better methods?

Lighting Optimization

- Only want to shade surfaces inside light volume
  - Anything else is wasted work



NVIDIA.

Lighting Stage Optimization

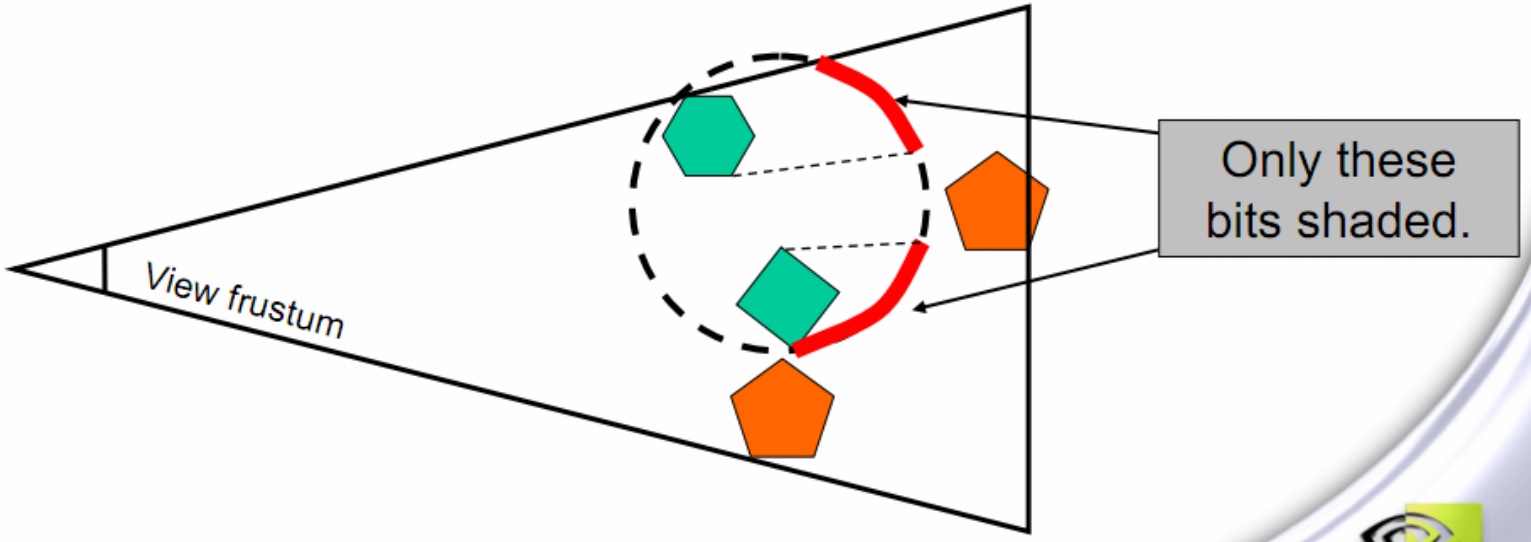
- A better method is to set depth test to 'greater' and render the back face of the light volume.
- Some wasted calculation still occurs if the light front side is:
  - intersecting the object
  - farther from the camera compared to the object
- An even better method?

Lighting Stage Optimization

- An even better method is to do 2 passes. Practically no wasted light calculation.
- Pass 1:
  - Color write: disable
  - Depth test: less
  - Stencil function: always
  - Stencil operation: replace with ‘X’
- Pass 2:
  - Color write: enable
  - Depth test: greater
  - Stencil function: equal to ‘X’
  - Stencil operation: nothing (set all to ‘keep’)

Stencil Cull

- Only regions that fail depth test represent objects within the light volume



6800 LEAGUES UNDER THE SEA

NVIDIA.

Lighting Stage Optimization

- Options on how to render the geometry:
  - Render normally with depth test
  - Render the back face with depth test set to 'greater'
  - 2 passes rendering:
    - Pass 1:
      - Color write: disable
      - Depth test: less
      - Stencil function: always
      - Stencil operation: replace with 'X'
    - Pass 2:
      - Color write: enable
      - Depth test: greater
      - Stencil function: equal to 'X'
      - Stencil operation: nothing (set all to 'keep')

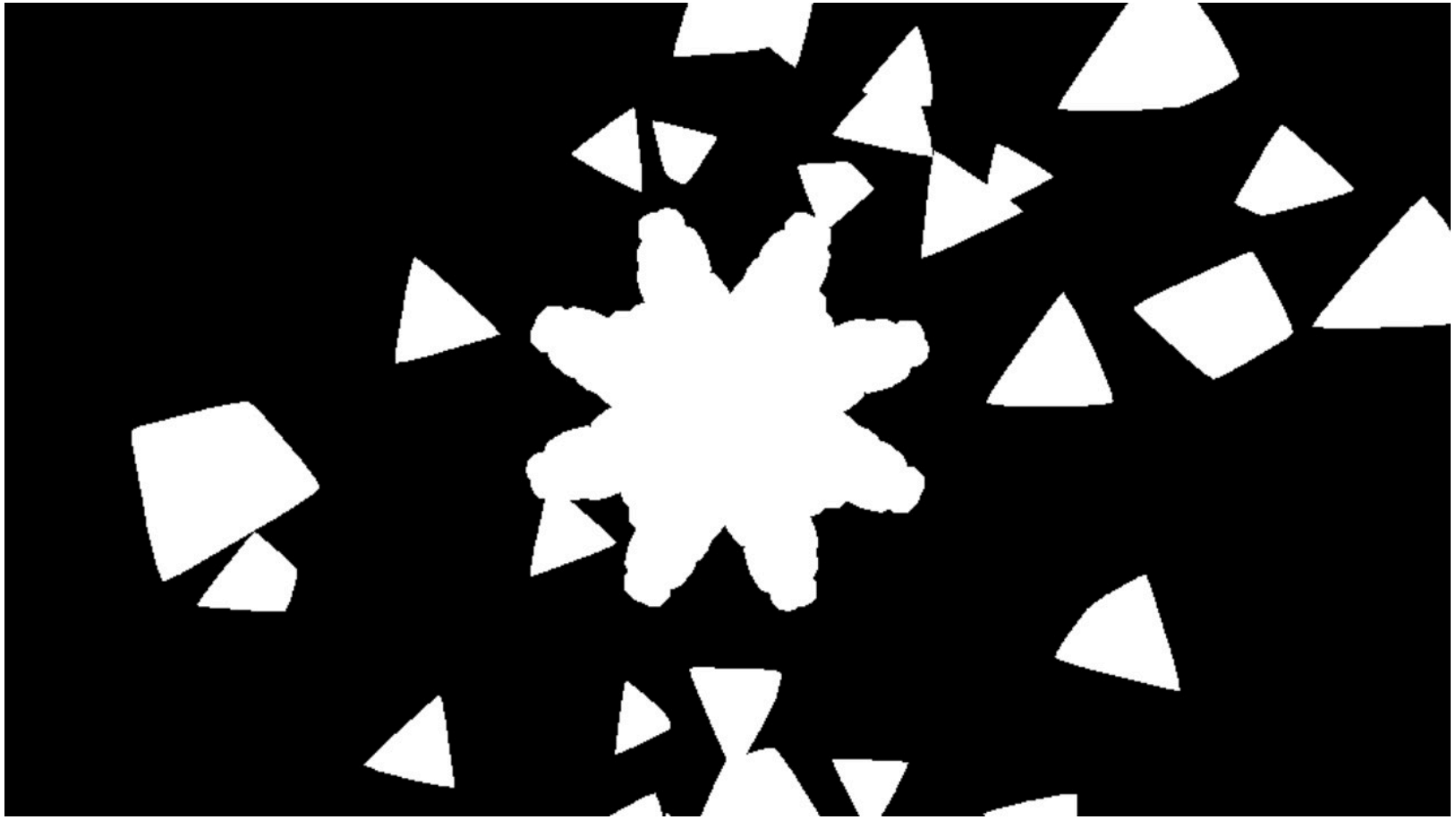
Lighting Stage Optimization

- Take care when the light volume intersect the front or the back clip plane.

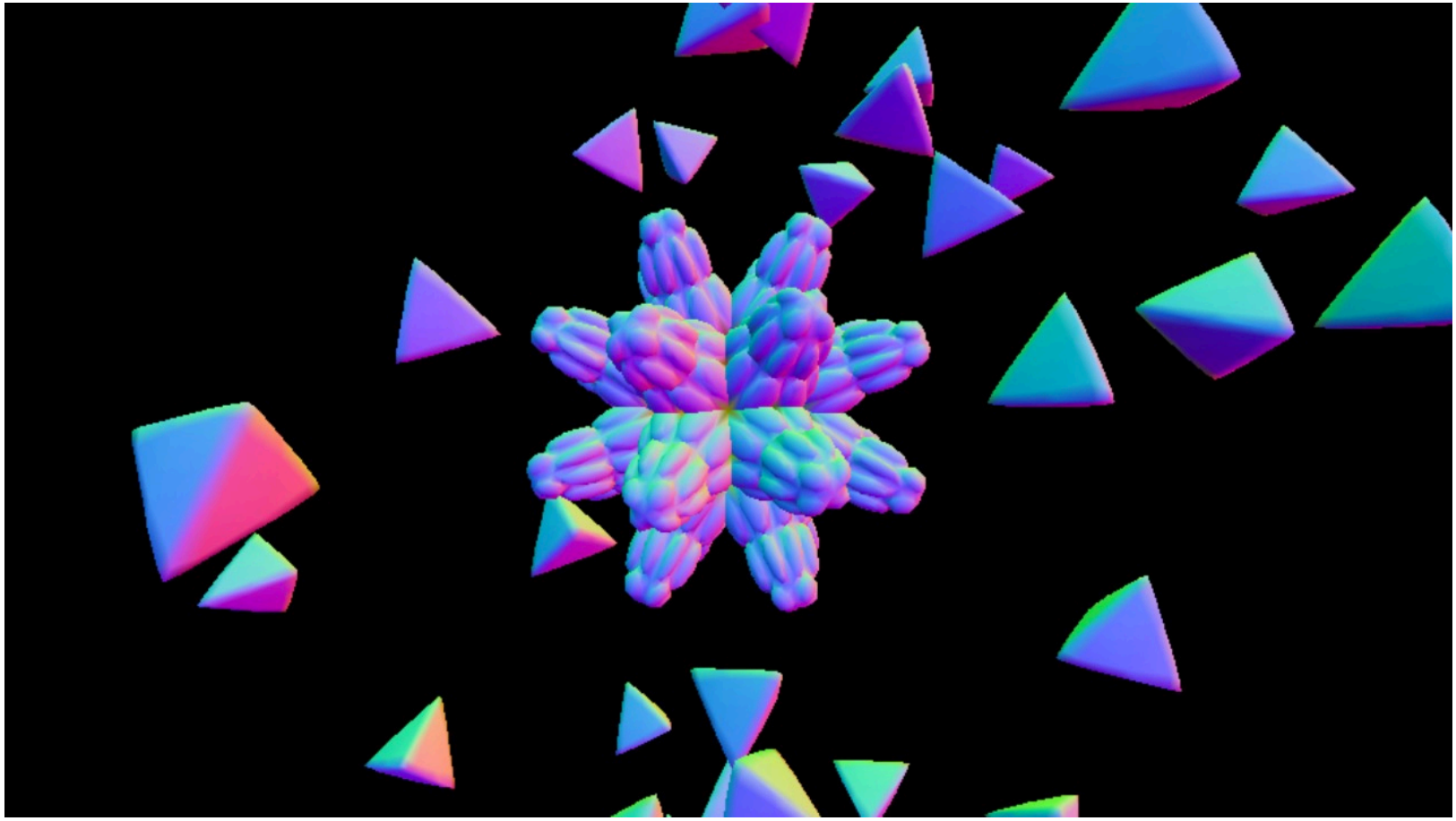
- Similar scheme like the shadow volume rendering can be used:
  - Use extension to ‘clamp’ instead of clip
  - Use shader to ‘clamp’ the depth value

Example (in pictures)

http://marcinignac.com/blog/deferred-rendering-explained/

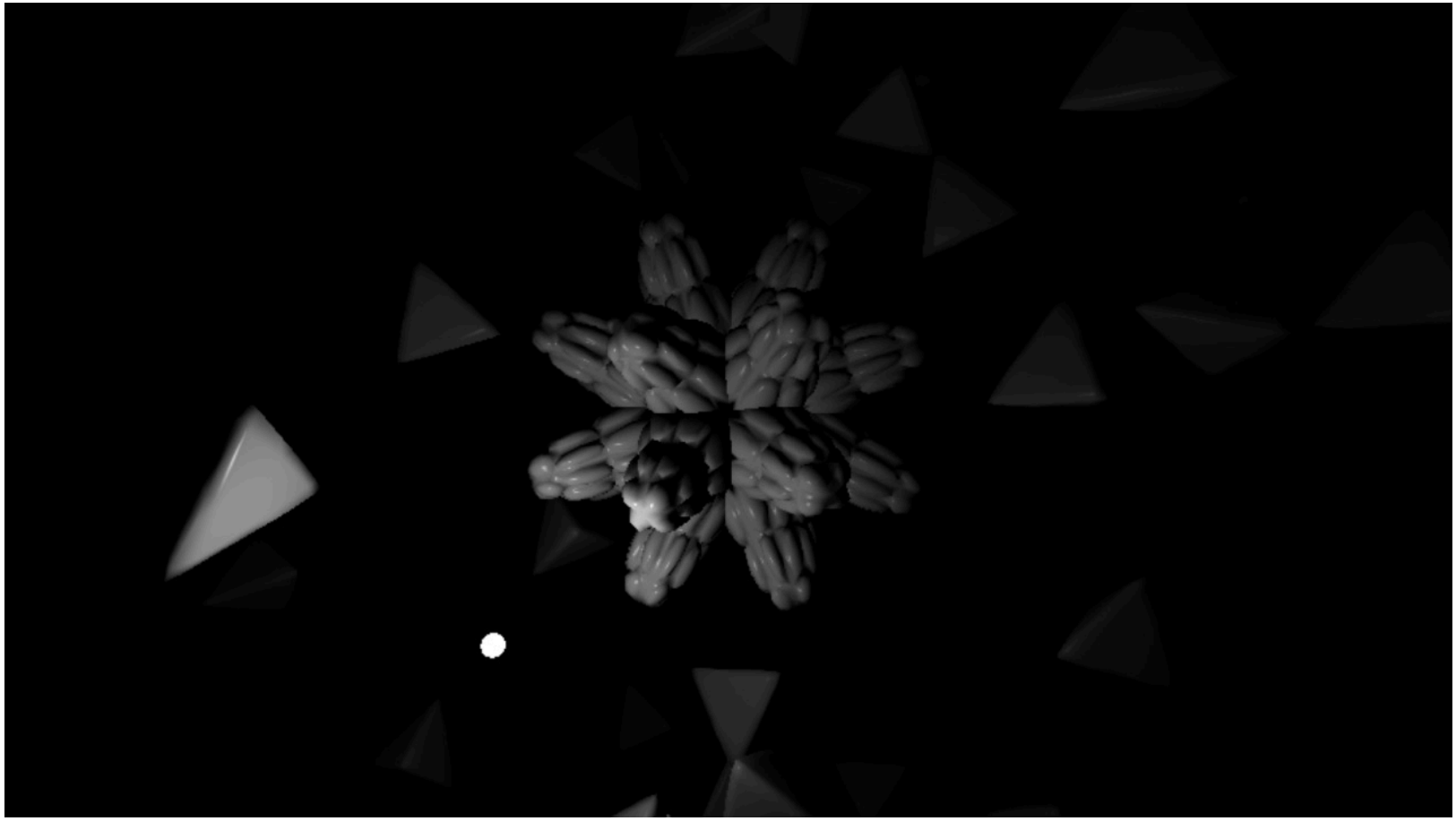


Normals

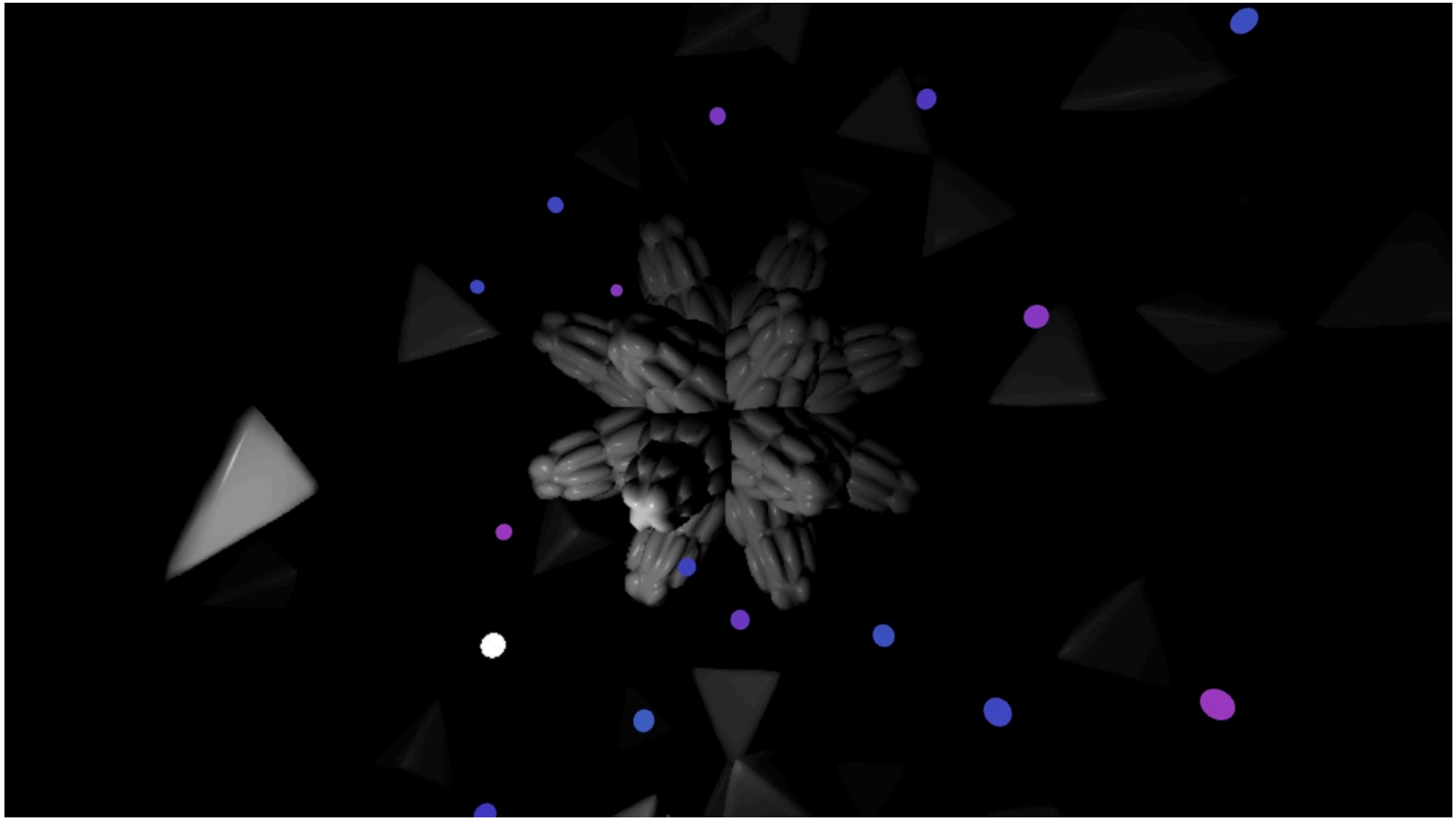


Depth from eye

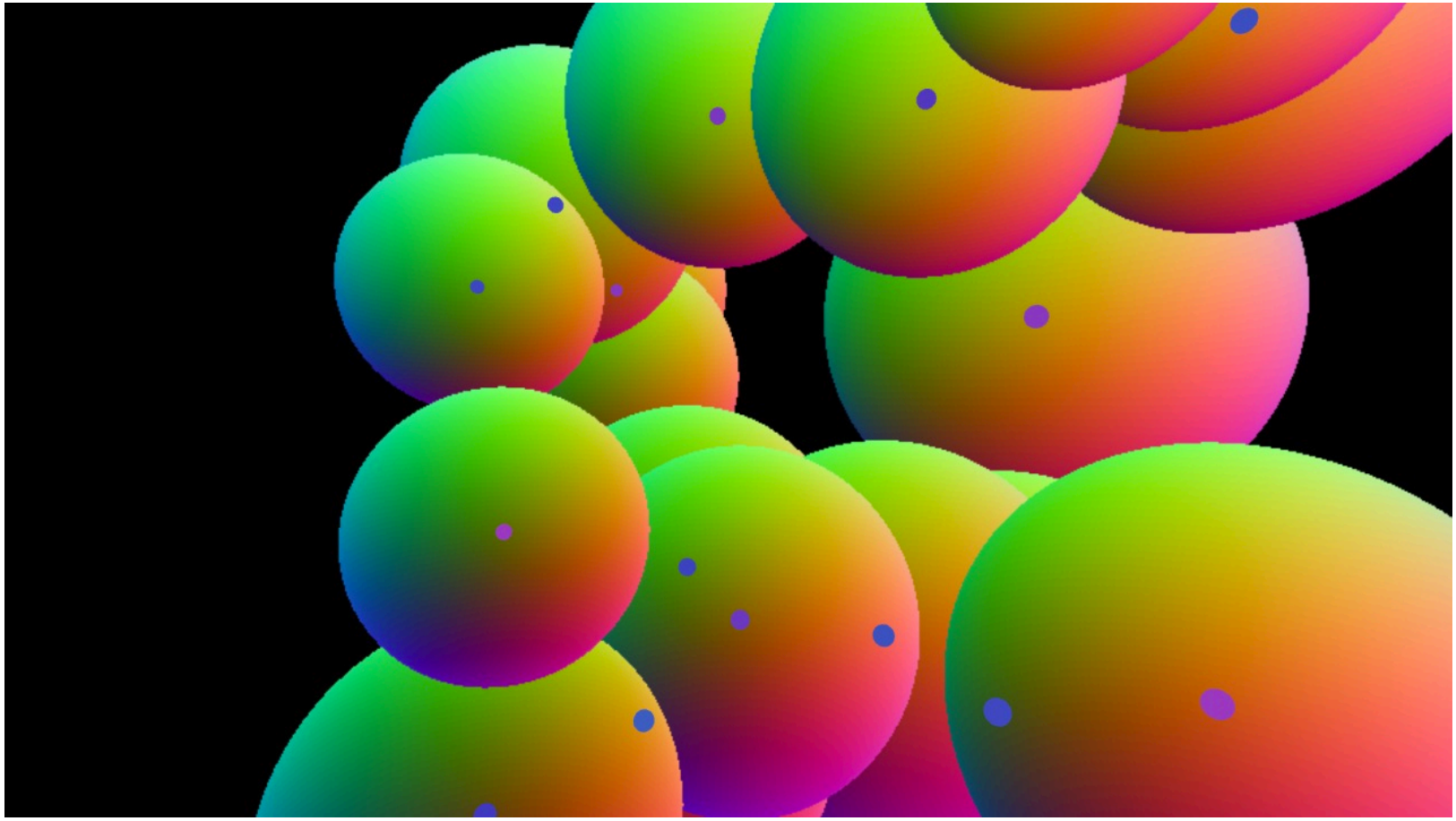
One light == one FSQ



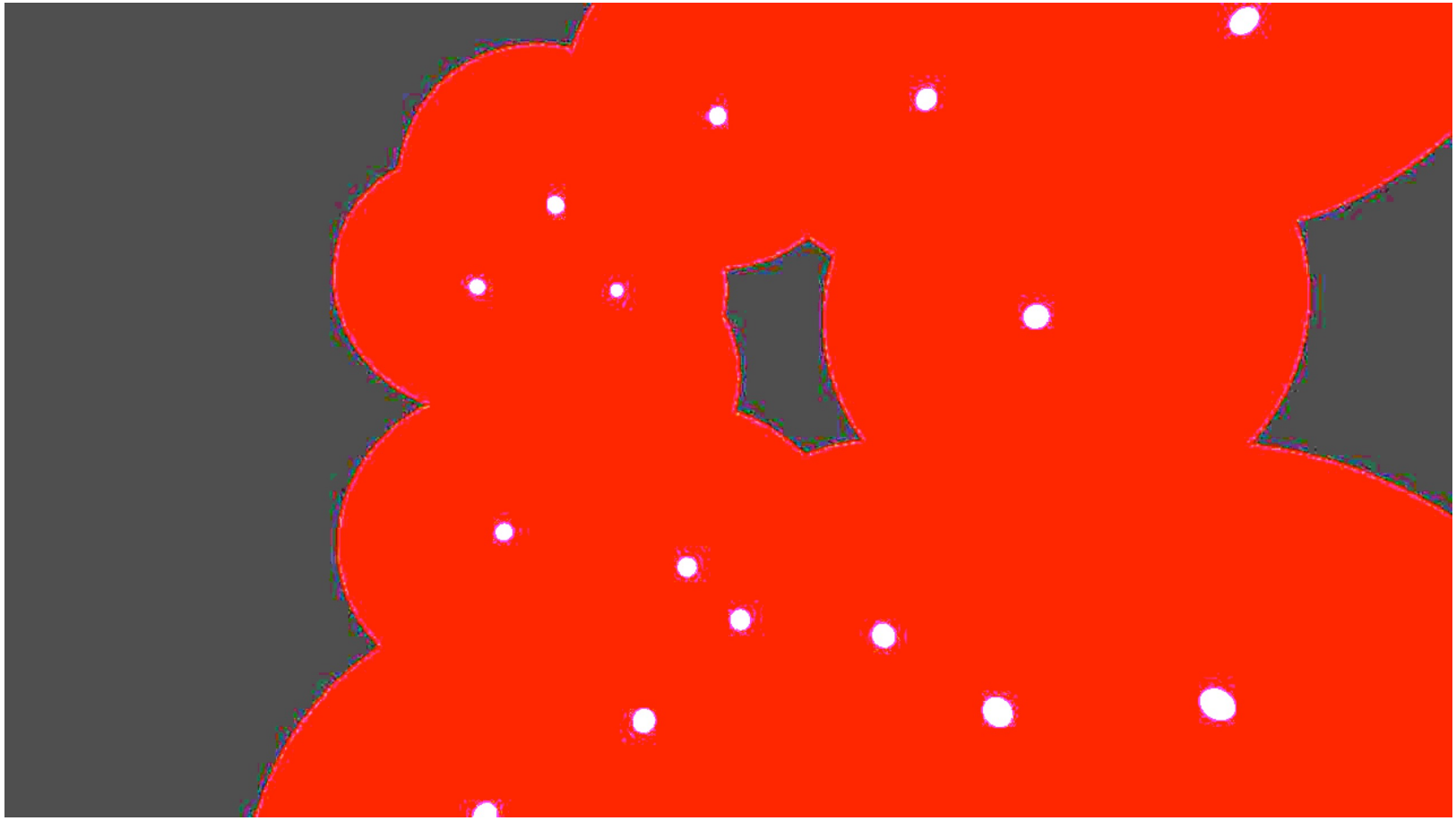
Many lights == ? FSQ



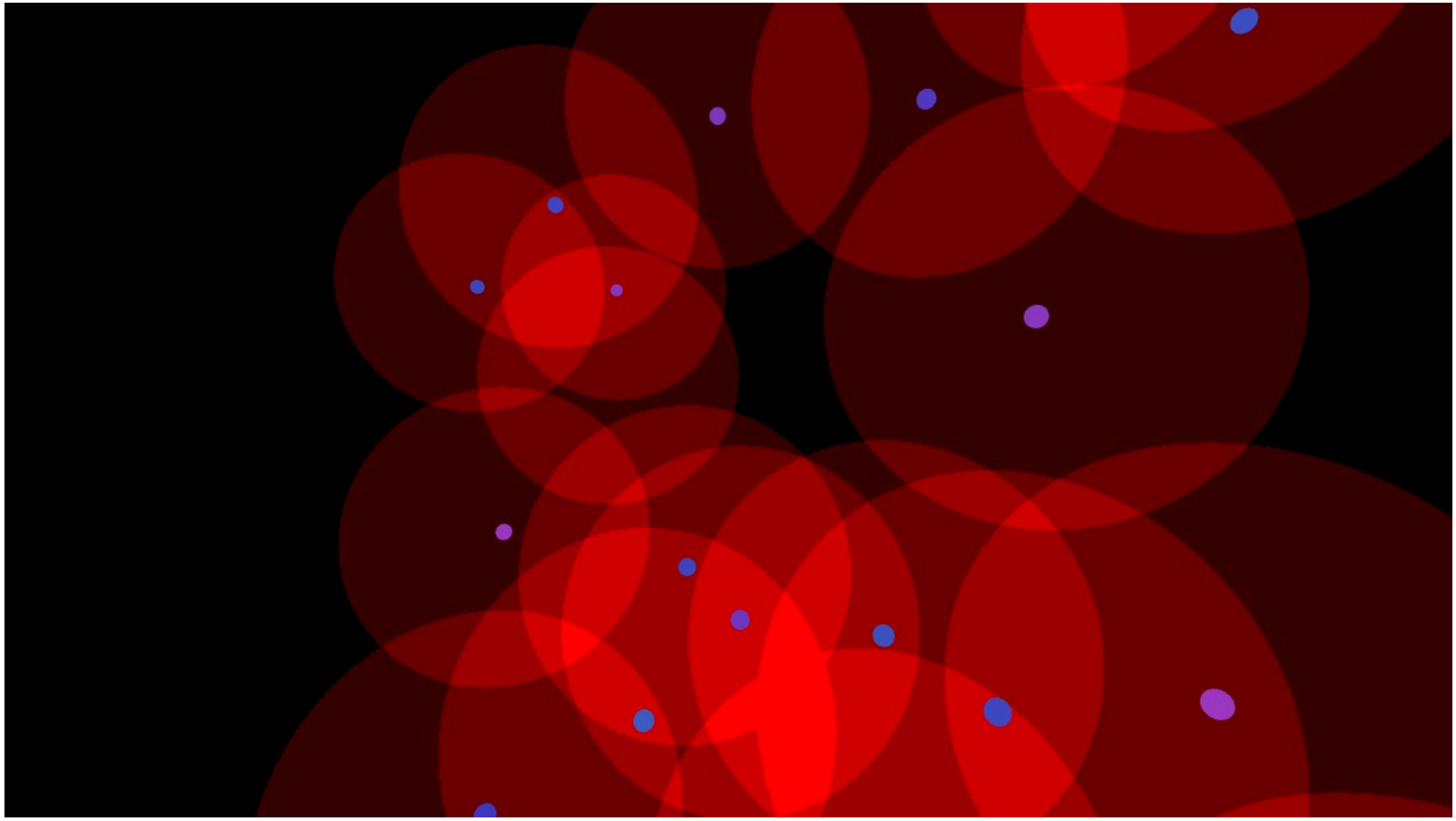
Light volume for point lights == Spheres!



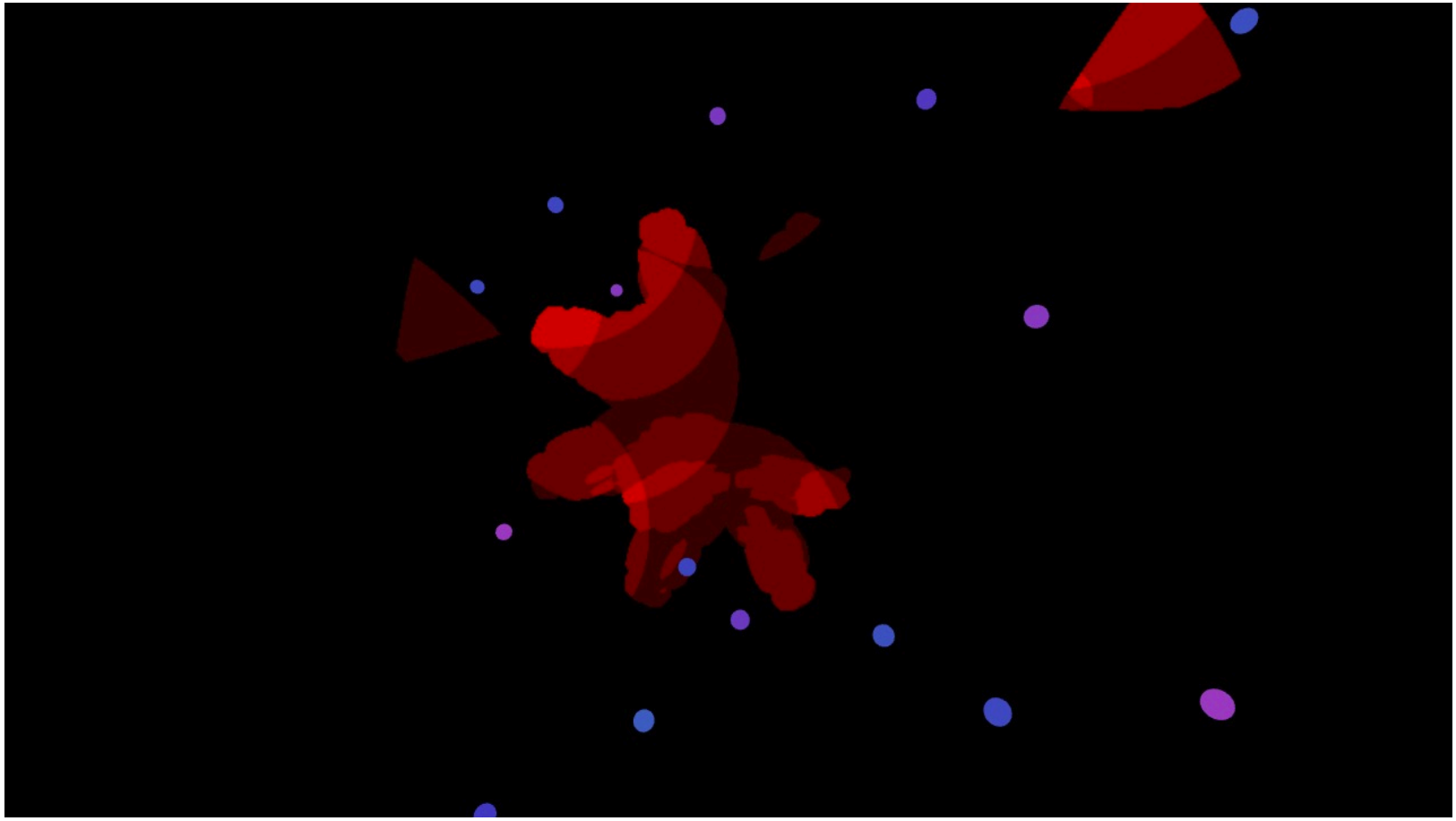
Debug Draw – pixels affected by the light



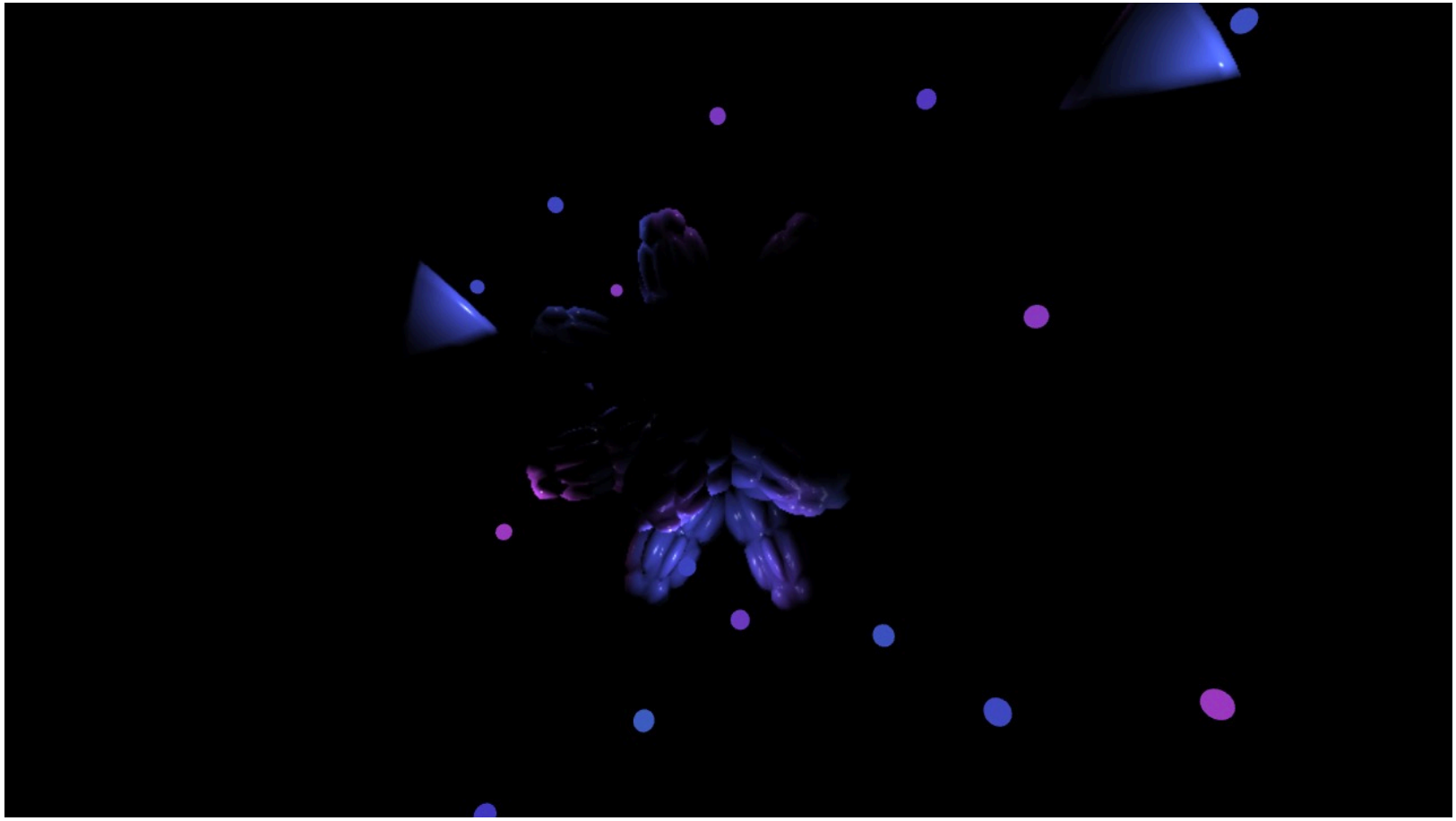
Pixel Overdraw?



Stencil Hack to draw objects in spheres



Final output



Post-processing Stage

Post-processing Stage

- The image generated after the lighting stage can be used as final image.
- However, since with all these information we have about the scene, we can do additional post-processing to enhance the image quality.

Post-processing Stage

- What can be done in the post-processing stage?
  - Anti-aliasing
  - Transparency
  - Other effects (glow, filter, blur, distortion, etc)
  - Tone mapping
  - Gamma correction

Anti-Aliasing

- To implement the anti-aliasing filter we use a shader to blur the final image only at the pixels with large discontinuities in normal and/or depth.

- To find those pixels, we use an edge detection filter applied to the normal and/or depth map.

Anti-Aliasing

- Can use Sobel operator for edge detection filter. Shown below are the Sobel-x and Sobel-y operator.

$$D_x = \frac{1}{8}\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \qquad D_y = \frac{1}{8}\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

- The magnitude of the gradient is calculated as:

$$\|\nabla_{sobel}\| = \sqrt{D_x^2 + D_y^2}$$

Anti-Aliasing

- Simpler method to detect the edge is by adding together the differences from all 8 neighboring pixels around the target pixel.
- 8 directions to detect discontinuity in all directions.
- We use the result as blur factor (clamped between 0 to 1).

Anti-Aliasing

- Blur factor calculation pseudo-code:
  - Generate a table with 8 entries containing the offset for neighboring fragments.
  - Reset the blur factor accumulator to 0
  - For each entry in the offset table
    - Calculate a sample position by adding the current fragment position and the offset.
    - Sample the normal/depth at the calculated sample position and subtract the normal/depth value of the current fragment from it.
    - add the result to the accumulator.
  - Clamp the blur factor between 0 and 1.

Anti-Aliasing

- Color blurring pseudo-code :
  - Generate a table with 8 entries containing the offset for neighboring fragments.
  - Reset the color accumulator to 0
  - For each entry in the offset table
    - Calculate a sample position by adding the current fragment position and the offset multiplied by the blur factor.
    - Sample the color and add it to the color accumulator.
  - Add the color for the current fragment position to the accumulator.
  - Average the color accumulator to get the final color for the fragment.

Anti-Aliasing

Anti-Aliasing

Anti-Aliasing

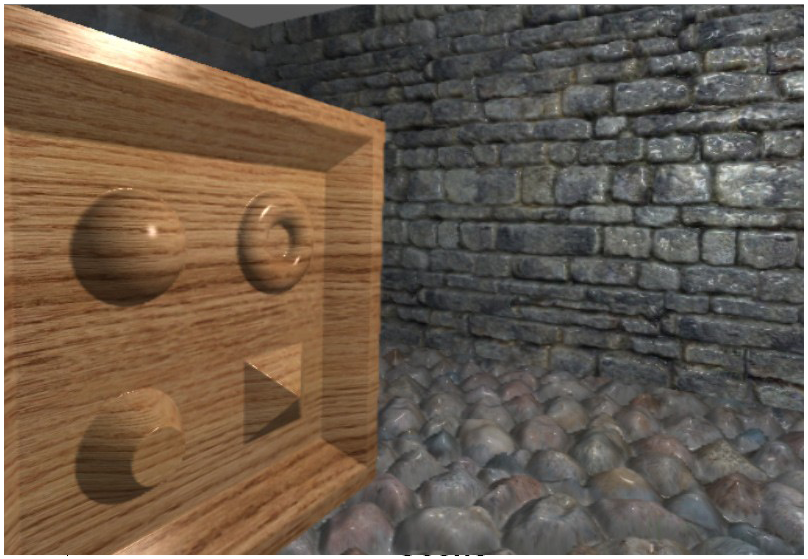
Comparison

Transparency

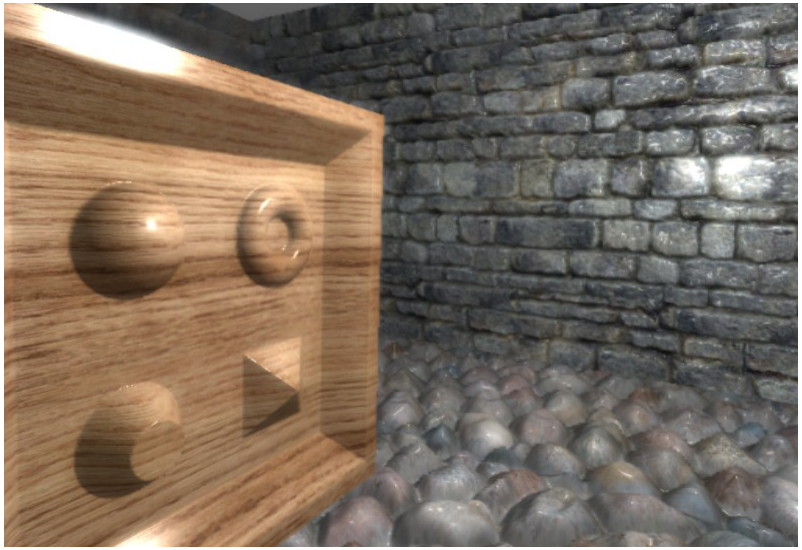
- Deferred shading does not support transparency. It only shades the nearest surface.
- So, for transparent polygons, just render them last after sorting them back to front.
- Make sure that the same depth buffer is used for both the deferred shading and the transparent triangles rendering.

Glow Effect

- Used to ‘glow’ around bright regions.



Original image



With glow effect

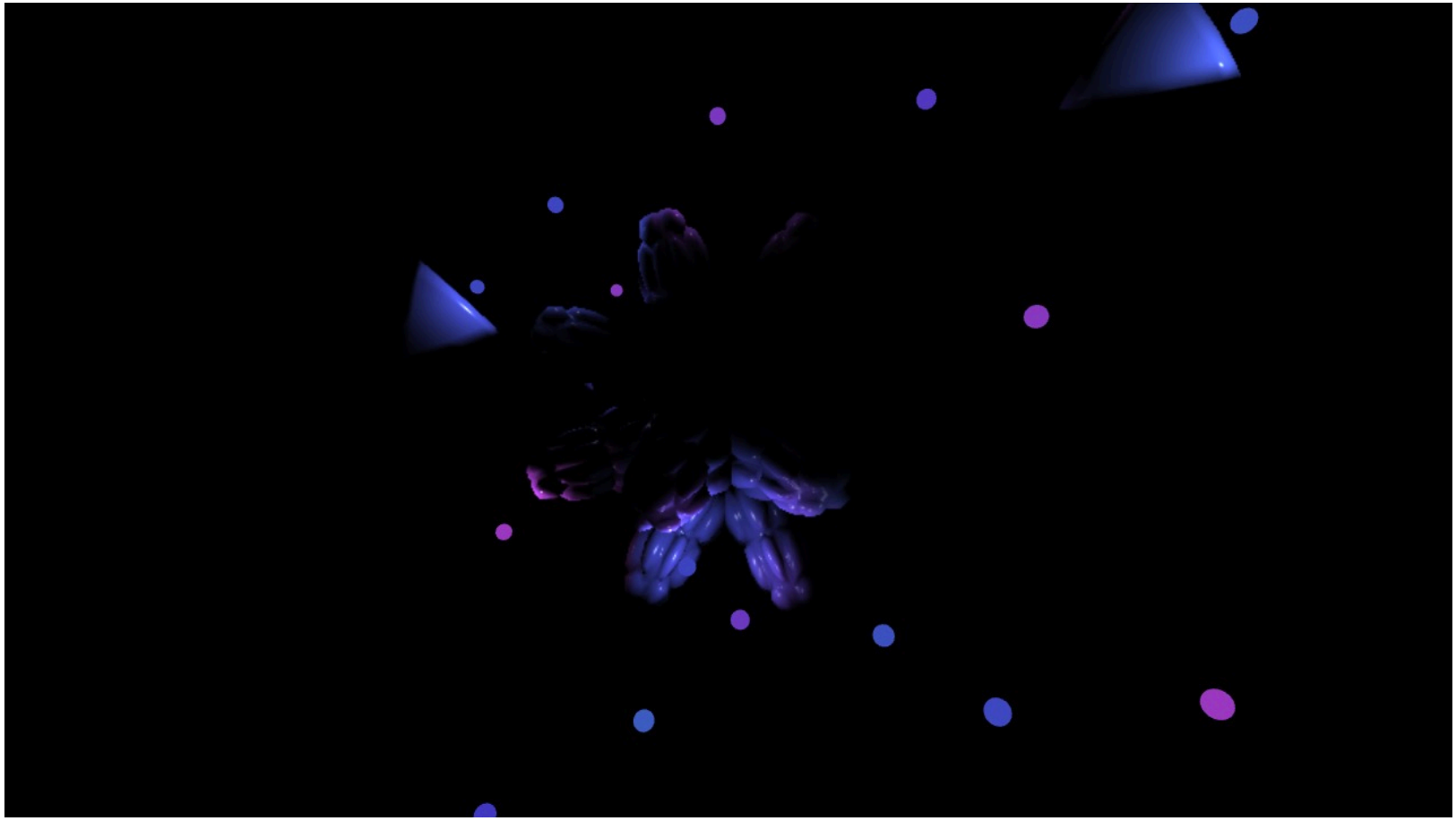
Glow Effect

- Glow effect uses two passes.
- It uses half or quarter of the picture original size
- Separate the process into an horizontal and vertical pass.
- The same shader is used for the horizontal and vertical passes as it receives the offsets as a shader parameter

Glow Effect

- Glow effect pseudo-code:
  - Render a lower resolution of the original image using a horizontal blur filter.
  - Copy the result out to texture.
  - Render the result at the same resolution using the vertical blur filter.
  - Copy the result out to texture.
  - Render the original image added with the result.

Tone Mapping – original output



Tone mapping - with Gamma correction

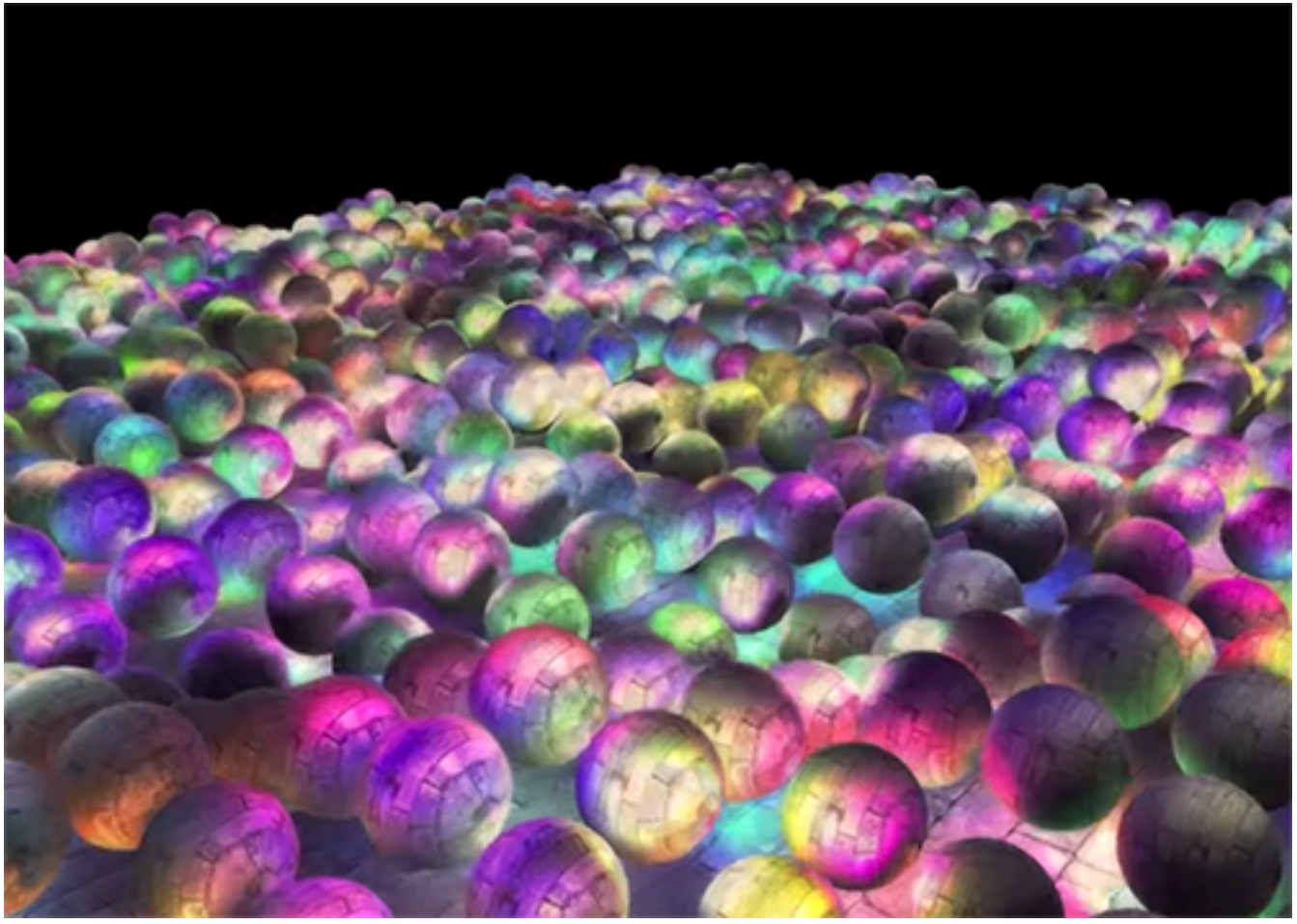
Demo

Final Phase

- The final stage is merely rendering the final image to be displayed on the screen.
- Draw a full screen quad
- Use the render target of the previous stage as a texture input
- For every fragment, lookup the texel value and pass it to the framebuffer

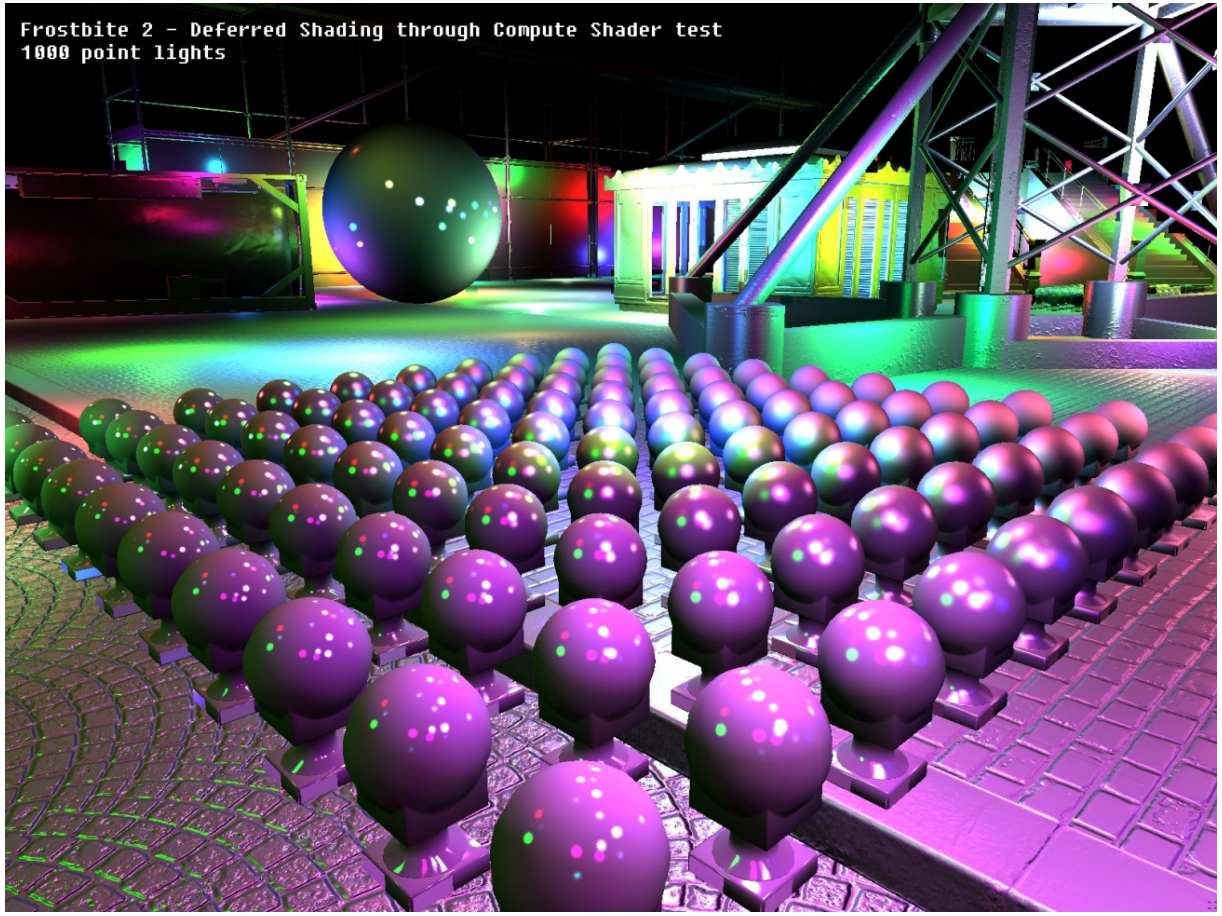
And, One More Thing

1847 point lights (Hannes Nevalainen)



1280x720, 8000 lights, 158 FPS

DX Compute Shader (1000 point lights)



References

- https://learnopengl.com/#!Advanced-Lighting/Deferred-Shading
- http://www.codinglabs.net/tutorial_simple_def_rendering.aspx
- http://diaryofagraphicsprogrammer.blogspot.com/2009/09/river-of-lights.html
- http://www.bungie.net/News/content.aspx?type=topnews&link=Siggraph_09