Computer Graphics II – Deferred Shading

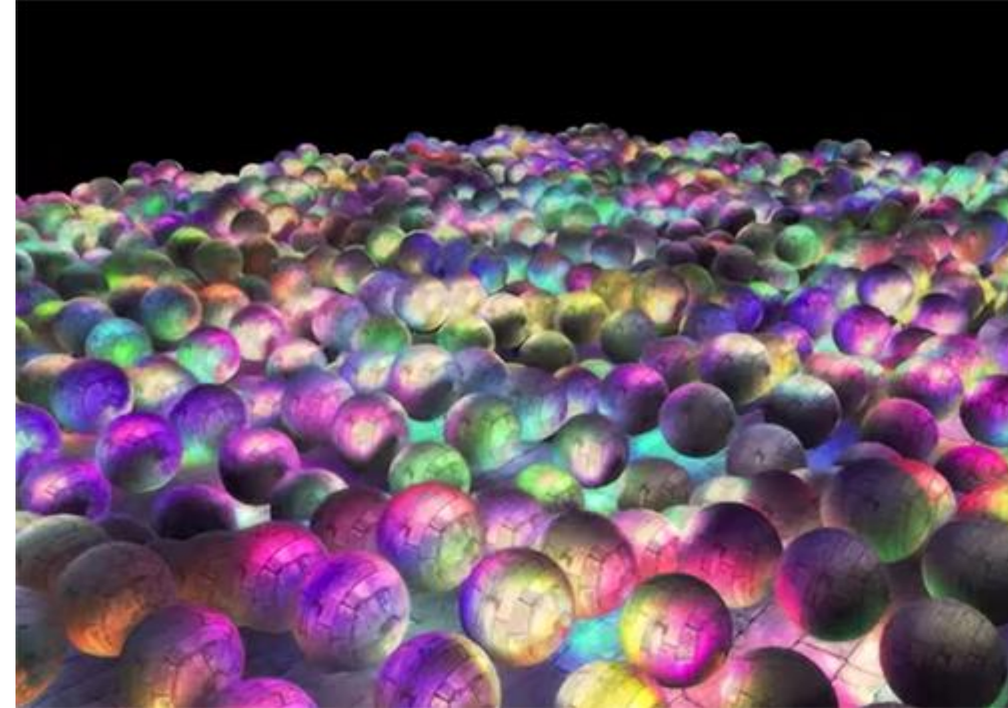
Kai Lawonn

Introduction

- The way we did lighting so far → called forward rendering/shading (straightforward to render an object, light it to all light sources in a scene, then render the next object, and so on for each object)
- Easy to understand and implement → quite heavy on performance (each rendered object iterate over each light source for every rendered fragment)
- Forward rendering wastes fragment shader runs in scenes with a high depth complexity (multiple objects cover the same screen pixel) as most fragment shader outputs are overwritten

Introduction

- Deferred shading/rendering tries to overcome these issues
- Significantly optimize scenes with large numbers of lights (render thousands of lights with an acceptable framerate)
- Image with 1847 point lights rendered with deferred shading → not be possible (framerate) with forward rendering



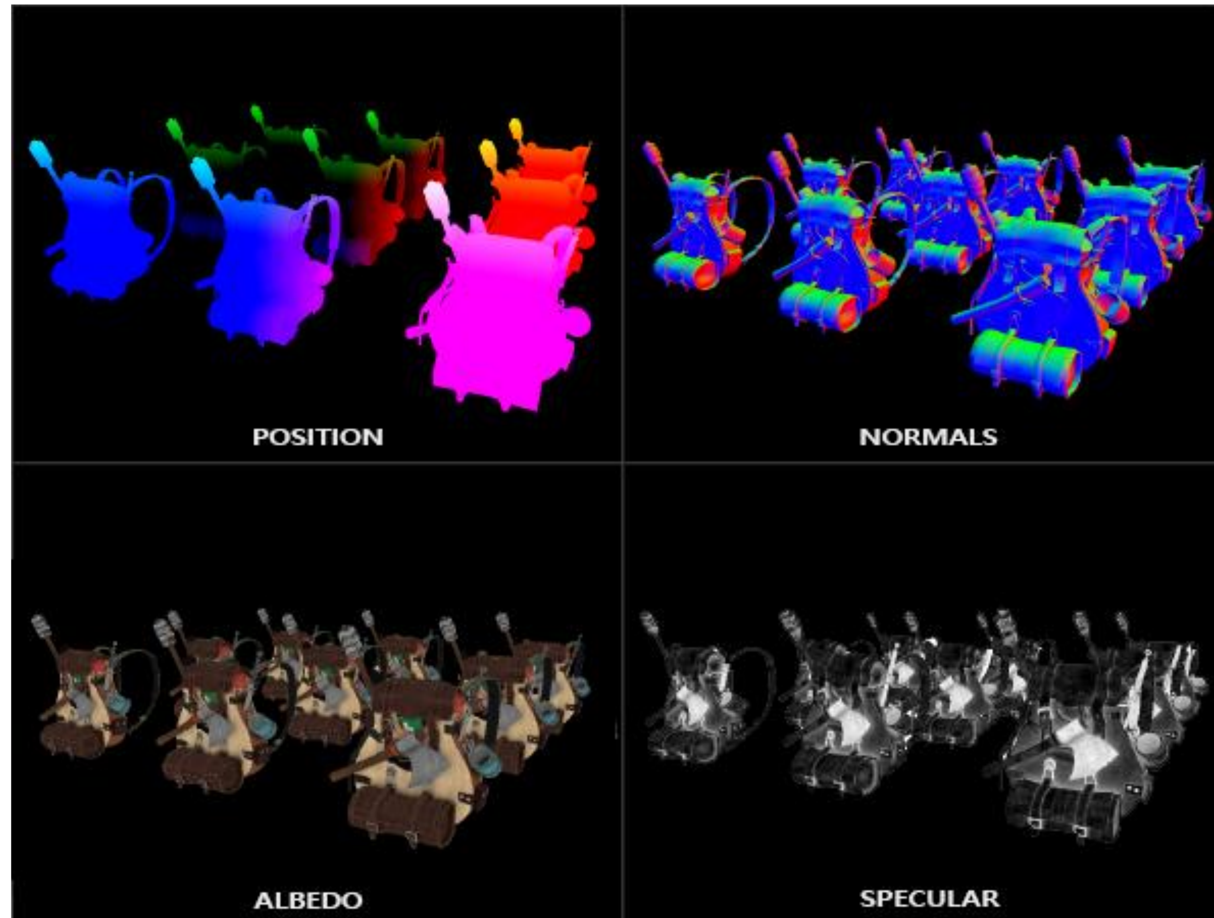
(Image by Hannes Nevalainen)

Introduction

- Deferred shading is based on the idea that we defer or postpone most of the heavy rendering (like lighting) to a later stage
- Consists of two passes:
- 1st (geometry) pass: render scene, retrieve geometrical information from objects → store in textures (G-buffer; like position, color, normals…)
- 2nd (light) pass: geometric information of a scene stored in the G-buffer is then later used for (more complex) lighting calculations

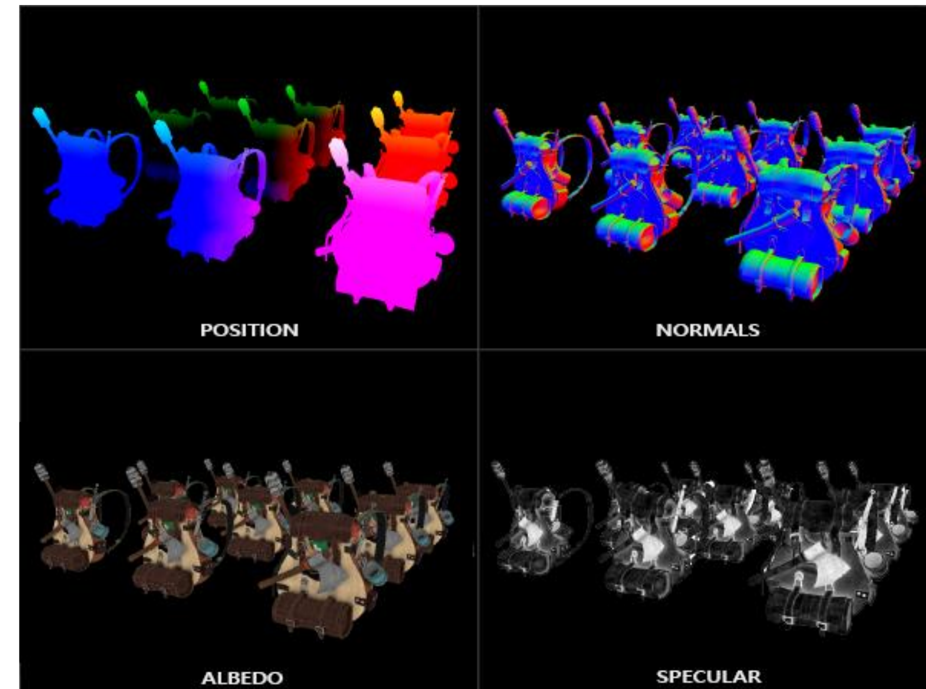
Introduction

- Below is the content of a G-buffer of a single frame:



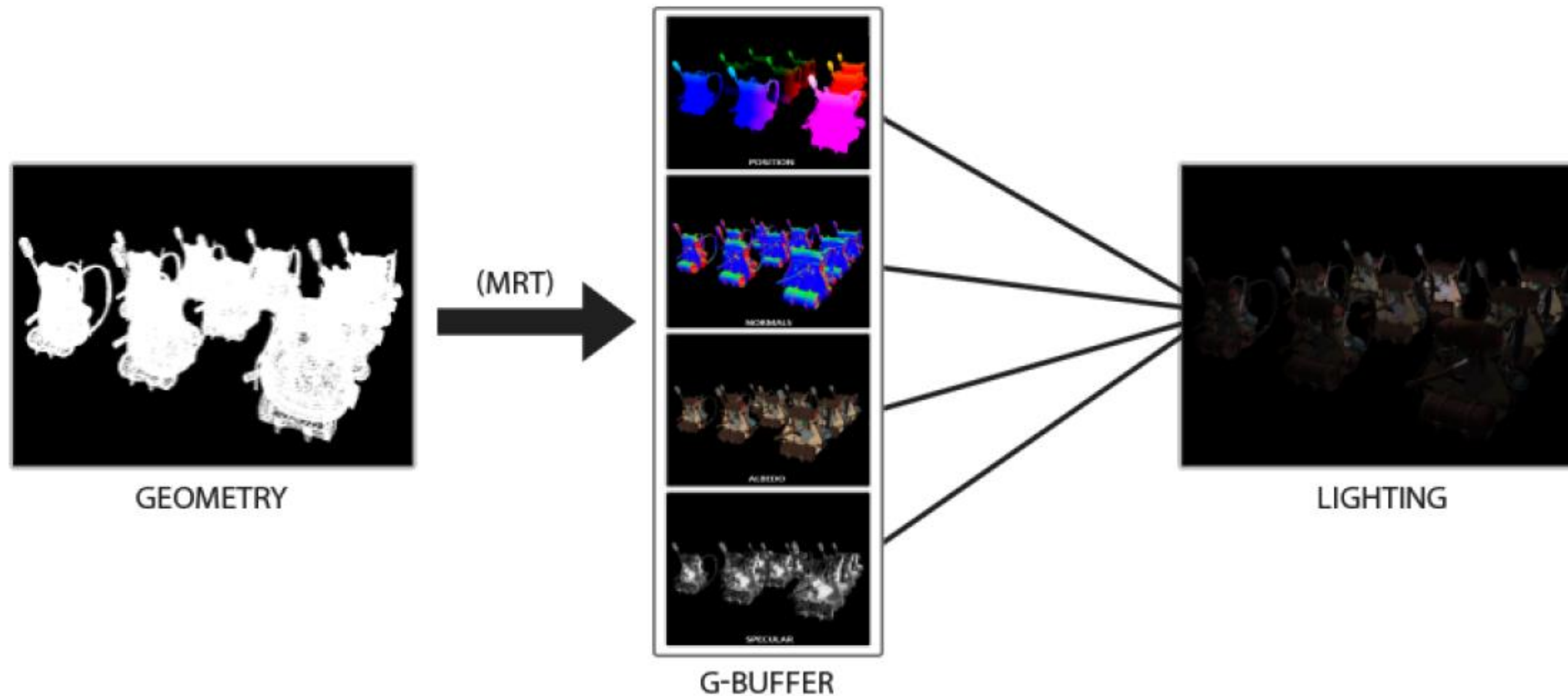
Introduction

- Use textures from the G-buffer in a 2nd (lighting) pass: render screen-filled quad and calculate the scene's lighting (with geometrical information stored in the G-buffer)
- Instead of taking each object from vertex to fragment shader → decouple its advanced fragment processes to a later stage
- Lighting calculations remain the same (but required input variables from G-buffer textures instead of the vertex shader)



Introduction

- Total process of deferred shading:



Introduction

- A few disadvantages: G-buffer requires to store a relatively large amount of scene data in its texture colorbuffers → memory (position vectors require a high precision)
- Does not support blending (only information of the topmost fragment) and MSAA does not work
- Several workarounds of these
- Filling G-buffer in geometry pass is quite efficient (store object information, i.e., position, color or normals into a framebuffer with small or zero amount of processing)
- Using multiple render targets (MRT) do this in a single render pass

The G-Buffer

The G-Buffer

- The G-buffer collective term of all textures storing lighting-relevant data for the final lighting pass
- Relevant the data to light a fragment with forward rendering:
  - 3D position vector to calculate the (interpolated) fragment position variable used for lightDir and viewDir
  - RGB diffuse color vector (aka albedo)
  - 3D normal vector (for surface’s slope)
  - Specular intensity float
  - All light source position and color vectors
  - The player or viewer’s position vector

The G-Buffer

- With these (per-fragment) variables are able to calculate the (Blinn-) Phong lighting
- Light source positions and colors, and player’s view position can be configured using uniform variables
- Other variables are all specific to each of an object’s fragments
- If exact same data passed to final deferred lighting pass, can calculate the same lighting effects (even rendering fragments of a 2D quad)

The G-Buffer

- No limit OpenGL what we can store in a texture → store all per-fragment data in one or multiple screen-filled textures (G-buffer) for later use later in the lighting pass
- G-buffer textures same size as the lighting pass's 2D quad → get exact fragment data like in forward rendering setting (but in the lighting pass; there is a one on one mapping)

The G-Buffer

- In pseudocode the entire process will look a bit like this:

```
while (...) // render loop
    {
        // 1. geometry pass: render all geometric/color data to g-buffer
        glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
        glClearColor(0.0, 0.0, 0.0, 1.0); // black so it won’t leak in g-buffer
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        gBufferShader.use();
        for (Object obj : Objects)
        {
            ConfigureShaderTransformsAndUniforms();
            obj.Draw();
        }
        // 2. lighting pass: use g-buffer to calculate the scene’s lighting
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
        lightingPassShader.use();
        BindAllGBufferTextures();
        SetLightingUniforms();
        RenderQuad();
    }
```

The G-Buffer

- Geometry pass: initialize a framebuffer object (gBuffer) that has multiple colorbuffers attached and a depth renderbuffer object (RBO)
- For position and normal texture use a high-precision texture (16 or 32-bit float per component), albedo and specular values default texture (8-bit precision per component)

The G-Buffer

```
unsigned int gBuffer;
glGenFramebuffers(1, &gBuffer);
glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
unsigned int gPosition, gNormal, gAlbedoSpec;
// position color buffer
glGenTextures(1, &gPosition);
glBindTexture(GL_TEXTURE_2D, gPosition);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA16F,SCR_WIDTH,SCR_HEIGHT,0,GL_RGBA,GL_FLOAT,NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER,GL_COLOR_ATTACHMENT0,GL_TEXTURE_2D,gPosition,0);
// normal color buffer
glGenTextures(1, &gNormal);
glBindTexture(GL_TEXTURE_2D, gNormal);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA16F,SCR_WIDTH,SCR_HEIGHT,0,GL_RGBA,GL_FLOAT,NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER,GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, gNormal, 0);
```

The G-Buffer

```
// color + specular color buffer
glGenTextures(1, &gAlbedoSpec);
glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,SCR_WIDTH,SCR_HEIGHT,0,GL_RGBA,GL_UNSIGNED_BYTE,NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER,GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D, gAlbedoSpec, 0);
// tell OpenGL which color attachments we'll use (of this framebuffer) for rendering
unsigned int attachments[3] =
                {GL_COLOR_ATTACHMENT0,GL_COLOR_ATTACHMENT1,GL_COLOR_ATTACHMENT2};
glDrawBuffers(3, attachments);
```

The G-Buffer

- Next, render into the G-buffer with following fragment shader:

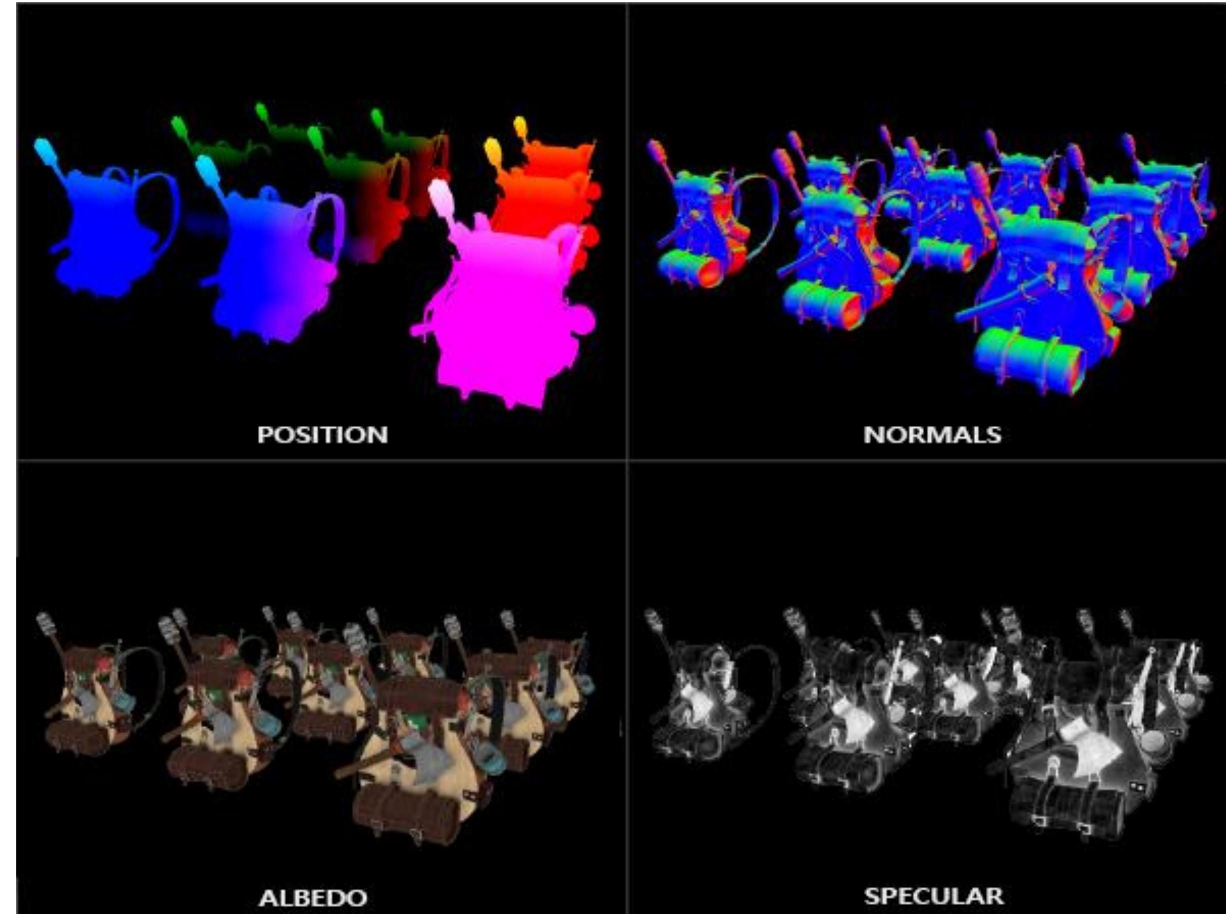
```
#version 330 core
layout (location = 0) out vec3 gPosition;
layout (location = 1) out vec3 gNormal;
layout (location = 2) out vec4 gAlbedoSpec;
in vec2 TexCoords;
in vec3 FragPos;
in vec3 Normal;
uniform sampler2D texture_diffuse1;
uniform sampler2D texture_specular1;

void main()
{
    // store the fragment position vector in the first gbuffer texture
    gPosition = FragPos;
    // also store the per-fragment normals into the gbuffer
    gNormal = normalize(Normal);
    // and the diffuse per-fragment color
    gAlbedoSpec.rgb = texture(texture_diffuse1, TexCoords).rgb;
    // store specular intensity in gAlbedoSpec's alpha component
    gAlbedoSpec.a = texture(texture_specular1, TexCoords).r;
}
```

The G-Buffer

Keep in mind that it is extremely important with lighting calculations to keep all variables in the same coordinate space (here, store and calculate all variables in world-space)

The G-Buffer

- Render backpack objects into the gBuffer framebuffer and visualize its content by projecting each color buffer one by one onto a screen-filled quad:



The Deferred Lighting Pass

The Deferred Lighting Pass

- With fragment data in the G-Buffer, can now calculate scene's final lighted colors by iterating over each pixel (G-Buffer textures) and use their content as input to the lighting algorithms
- G-buffer texture values represent final transformed fragment values, have to do the expensive lighting operations once per pixel → Deferred shading efficient (especially in complex scenes, normally multiple expensive fragment shader calls in a forward rendering)

The Deferred Lighting Pass

- Lighting pass: to render a 2D screen-filled quad and execute an expensive lighting fragment shader on each pixel:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, gPosition);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, gNormal);
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, gAlbedoSpec);
// also send light relevant uniforms
shaderLightingPass.use();
SendAllLightUniformsToShader(shaderLightingPass);
shaderLightingPass.setVec3("viewPos", camera.Position);
RenderQuad();
```

The Deferred Lighting Pass

- Fragment shader of the lighting pass similar to used lighting shaders
- New: method to obtain lighting's input variables (sample from the G-buffer):

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D gAlbedoSpec;

struct Light {
    vec3 Position;
    vec3 Color;
};
```

The Deferred Lighting Pass

```
const int NR_LIGHTS = 32;
uniform Light lights[NR_LIGHTS];
uniform vec3 viewPos;

void main()
{
    // retrieve data from gbuffer
    vec3 FragPos = texture(gPosition, TexCoords).rgb;
    vec3 Normal = texture(gNormal, TexCoords).rgb;
    vec3 Albedo = texture(gAlbedoSpec, TexCoords).rgb;
    float Specular = texture(gAlbedoSpec, TexCoords).a;

    // then calculate lighting as usual
    vec3 lighting  = Albedo * 0.1; // hard-coded ambient component
    vec3 viewDir  = normalize(viewPos - FragPos);
```

The Deferred Lighting Pass

```
for(int i = 0; i < NR_LIGHTS; ++i)
{
        // diffuse
        vec3 lightDir = normalize(lights[i].Position - FragPos);
        vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Albedo * lights[i].Color;
        lighting += diffuse;
    }
    FragColor = vec4(lighting, 1.0);
}
```

F5…

- … 32 small lights

The Deferred Lighting Pass

- Disadvantages of deferred shading:
  - Not possible to do blending (values from single fragments in the G-buffer, blending combination of multiple fragments)
  - Use the same lighting algorithm for most of scene's lighting (can alleviate this a bit by including more material-specific data in the G-buffer)
- Overcome these disadvantages (especially blending): split renderer into two parts:
  - Deferred rendering part
  - Forward rendering part
- Forward: specifically meant for blending or special shader effects not suited for a deferred rendering pipeline

Combining Deferred Rendering with Forward Rendering

Deferred and Forward Rendering

- Render lights as a 3D cube positioned at the light source's position emitting the color of the light alongside the deferred renderer
- First idea: forward render all the light sources on top of the deferred lighting quad at the end of the deferred shading pipeline
- Render the cubes, but only after finished the deferred rendering operations

Deferred and Forward Rendering

- In code this will look a bit like this:

```
 // deferred lighting pass
[...]
RenderQuad();
// now render all light cubes with forward rendering as we’d normally do
shaderLightBox.use();
shaderLightBox.setMat4("projection", projection);
shaderLightBox.setMat4("view", view);
for (unsigned int i = 0; i < lightPositions.size(); i++)
{
   model = glm::mat4(1.0f);
   model = glm::translate(model, lightPositions[i]);
   model = glm::scale(model, glm::vec3(0.25f));
   shaderLightBox.setMat4("model", model);
   shaderLightBox.setVec3("lightColor", lightColors[i]);
   RenderCube();
}
```

F5…

- … nope!

Deferred and Forward Rendering

- First copy depth information stored in the geometry pass into the framebuffer’s depth buffer and only then render the light cubes
- This way light cubes’ fragments rendered only on top of the previously rendered geometry
- Can copy the content of a framebuffer to the content of another framebuffer with the help of glBlitFramebuffer

Deferred and Forward Rendering

- Stored depth of objects rendered in deferred shading pass in the gBuffer FBO
- To copy the content of depth buffer to the depth buffer of the default framebuffer, the light cubes would then render as if all of the scene's geometry was rendered with forward rendering

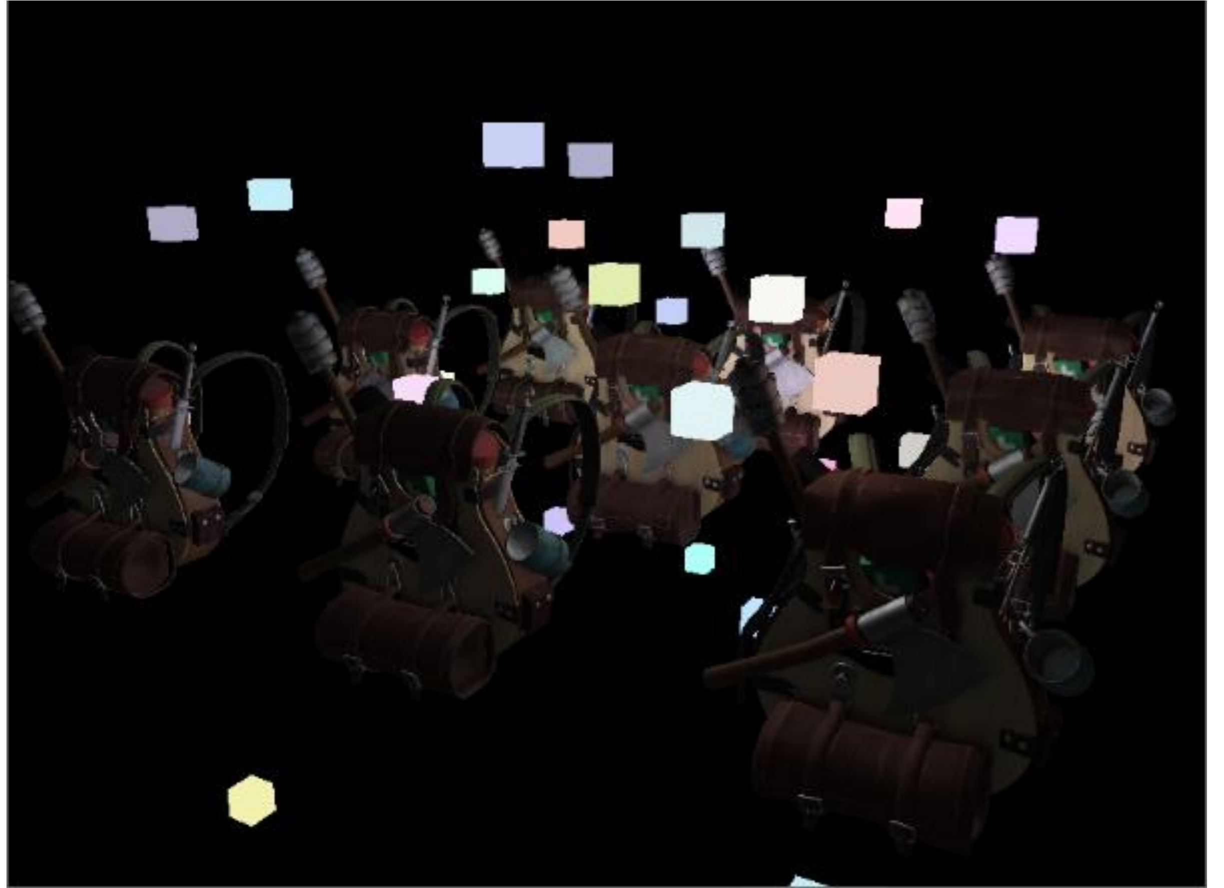
Deferred and Forward Rendering

- Have to specify a framebuffer as the read framebuffer and similarly specify a framebuffer as the write framebuffer:

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, gBuffer);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0); // write to default framebuffer
glBlitFramebuffer(0, 0, SCR_WIDTH, SCR_HEIGHT, 0, 0, SCR_WIDTH, SCR_HEIGHT,
GL_DEPTH_BUFFER_BIT, GL_NEAREST);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// now render light cubes as before
[...]
```

F5…

- … yes!



A Larger Number Of Lights

Introduction

- Deferred rendering can render an enormous amount of light sources without a heavy cost on performance
- Itself does not allow a large amount of light sources (still calculate each fragment's lighting component for each scene's light sources)
- But with a very neat optimization, large amount can apply to the deferred rendering pipeline: that of light volumes

Introduction

- Normally, calculate contribution of each light source in a scene, regardless of their distance to the fragment
- Large portion of lights never reach fragment → waste of computations
- Idea behind light volumes: calculate the radius or volume of a light source (area where light is able to reach fragments)
- Most light sources use attenuation → can use that to calculate the maximum distance or radius their light is able to reach
- Then only do lighting calculations if a fragment is inside one or more of these light volumes → saves considerable amount of computation
- Trick to this approach: figuring out size or radius of the light volume of a light source

Calculating Light’s Volume or Radius

- To obtain light’s volume or radius solve the attenuation equation for a brightness (dark can be 0.0; slightly more lit but still dark 0.03)
- To calculate a light’s volume/radius, use extensive attenuation functions (Lighting II lecture):

$$F_{att} = \frac{I_{max}}{K_c + K_l \cdot d + K_q \cdot d^2}$$

- Not a solution for $F_{att} = 0$, but close to

Calculating Light's Volume or Radius

- Acceptable brightness value is 5/256
- Divided by 256 as the default 8-bit framebuffer can display that many intensities per component

The attenuation function used is mostly dark in its visible range (having less than 5/256, light volume would become too large and thus less effective)

As long as a user cannot see a sudden cut-off of a light source at its volume borders we'll be fine.

This always depends on the type of scene; a higher brightness threshold results in smaller light volumes and thus a better efficiency, but can produce noticeable artifacts where lighting seems to break at a volume's borders.

Calculating Light's Volume or Radius

- The attenuation equation we have to solve becomes:

$$\frac{5}{256} = \frac{I_{max}}{K_c + K_l \cdot d + K_q \cdot d^2}$$

$$\frac{256}{5} = \frac{K_c + K_l \cdot d + K_q \cdot d^2}{I_{max}}$$

$$0 = K_q \cdot d^2 + K_l \cdot d + K_c - \frac{256}{5} \cdot I_{max}$$

Calculating Light's Volume or Radius

- Solved:

$$d_{1/2} = \frac{-K_1 \pm \sqrt{K_1^2 - 4K_q(K_c - \frac{256}{5} I_{max})}}{2K_q}$$

$$d = \frac{-K_1 + \sqrt{K_1^2 - 4K_q(K_c - \frac{256}{5} I_{max})}}{2K_q}$$

Calculating Light's Volume or Radius

- General equation allowing to calculate the light volume's radius for the light source given a constant, linear and quadratic parameter:

```
float constant = 1.0;
float linear = 0.7;
float quadratic = 1.8;
float lightMax = std::fmaxf(std::fmaxf(lightColor.r, lightColor.g),
                                                           lightColor.b);
float radius = (-linear + std::sqrtf(linear * linear - 4 * quadratic *
    (constant - (256.0 / 5.0) * lightMax))) / (2 * quadratic);
```

Calculating Light's Volume or Radius

- Calculate this radius for each light source of the scene and use it to only calculate lighting for that light source if a fragment is inside the light source's volume

Calculating Light's Volume or Radius

- Updated lighting pass fragment shader with the calculated light volumes (teaching purposes and not viable in a practical setting)

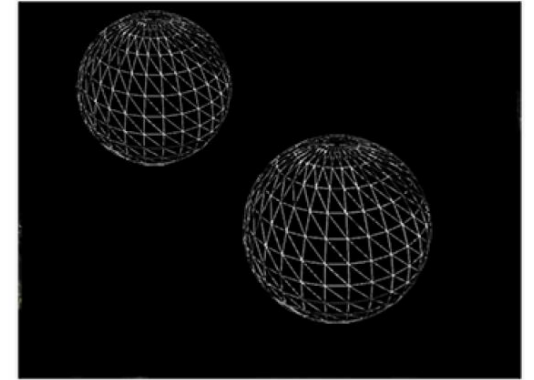
```
struct Light {
   [...]
   float Radius;
};
void main()
{
   [...]
   for(int i = 0; i < NR_LIGHTS; ++i)
   {
      // calculate distance between light source and current fragment
      float distance = length(lights[i].Position - FragPos);
      if(distance < lights[i].Radius)
      {
         // do expensive lighting
         [...]
      }
   }
}
```

How to Really Use Light Volumes

- Fragment shader does not really work in practice and only illustrates how we can sort of use a light's volume to reduce the lighting calculations
- GPU and GLSL are really bad at optimizing loops and branches
- The reason for this: shader execution on the GPU highly parallel and most architectures have a requirement that for large collection of threads they need to run the exact same shader code for it to be efficient
- Shader always executes all branches of an if statement (to ensure the shader runs are the same) → radius check optimization completely useless
- Still calculate lighting for all light sources!

How to Really Use Light Volumes

- Approach is to render actual spheres (scaled by light volume radius)
- Center of spheres are positioned at the light source's position and it encompasses the light's visible volume
- Trick: use largely the same deferred fragment shader for rendering the sphere
- Rendered sphere produces fragment shader invocations, that match pixels the light source affects → render the relevant pixels and skip all other pixels



How to Really Use Light Volumes

- Do this for each light source blend the resulting fragments
- Same scene as before, but render only relevant fragments per ligh source
- Reduces computations from $nr_objects\ *\ nr_lights$ to $nr_objects\ +\ nr_lights$ (efficient with a large number of lights)
- Makes deferred rendering suitable for rendering a large number of lights
- An issue: face culling should be enabled (otherwise render a light's effect twice), when enabled user might enter light source's volume after which the volume isn't rendered anymore (due to back-face culling) → removing the light source's influence
- Solved this with a stencil buffer trick

How to Really Use Light Volumes

- Rendering light volumes take a heavy toll on performance (generally faster than normal deferred shading but not the best optimization)
- Two other popular (efficient) extensions on top of deferred shading: deferred lighting and tile-based deferred shading
- Incredibly efficient at rendering large amounts of light and also allow for relatively efficient MSAA

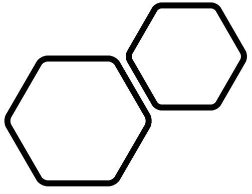
Deferred Rendering vs Forward Rendering

Deferred Rendering vs Forward Rendering

- By itself (without light volumes) deferred shading is already a large optimization (each pixel only runs a single fragment shader, compared to forward rendering)
- Deferred rendering a few disadvantages: a large memory overhead, no MSAA and blending still has to be done with forward rendering
- Having a small scene and not too many lights, deferred rendering is not necessarily faster (sometimes even slower as the overhead then outweighs the benefits of deferred rendering)
- In more complex scenes deferred rendering quickly becomes a significant optimization; especially with the more advanced optimization extensions

Deferred Rendering vs Forward Rendering

- Note, all effects that can be accomplished with forward rendering can also be implemented in a deferred rendering context (requires a small translation step)
- E.g., want to use normal mapping in a deferred renderer → change geometry pass shaders to output a world-space normal extracted from a normal map (TBN matrix) instead of the surface normal
- Lighting calculations do not to change
- E.g., parallax mapping, first displace the texture coordinates in the geometry pass before sampling an object’s diffuse, specular or normal textures



Questions???