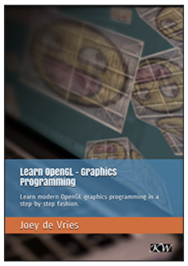
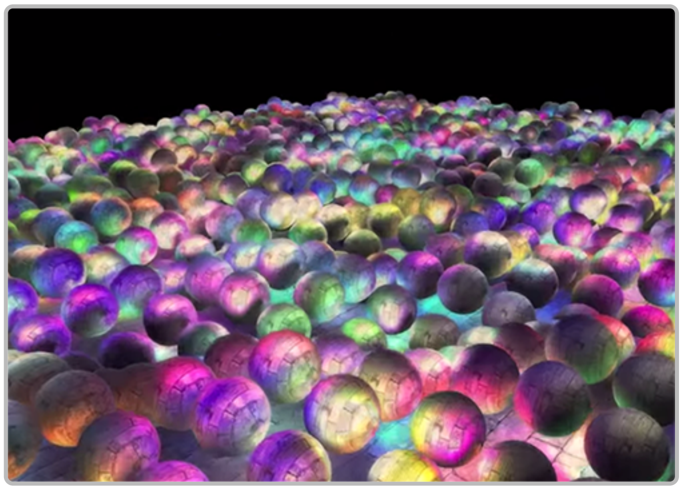

Learn OpenGL - Graphics Programming
Learn modern OpenGL graphics programming in a step-by-step fashion.
Joey de Vries

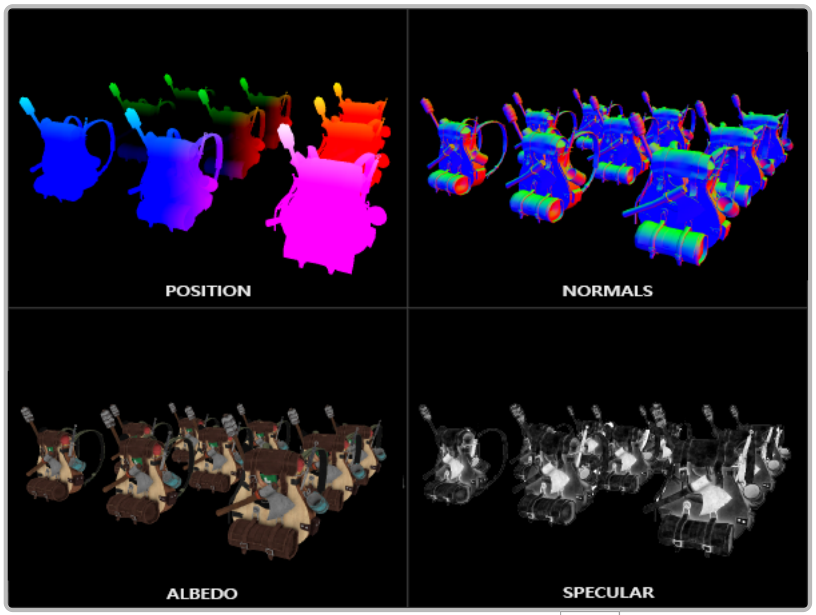
Deferred Shading

The way we did lighting so far was called forward rendering or forward shading. A straightforward approach where
an object and light it according to all light sources in a scene. We do this for every object individually for each objec
scene. While quite easy to understand and implement it is also quite heavy on performance as each rendered objec
iterate over each light source for every rendered fragment, which is a lot! Forward rendering also tends to waste a
fragment shader runs in scenes with a high depth complexity (multiple objects cover the same screen pixel) as frag
shader outputs are overwritten.

Deferred shading or deferred rendering aims to overcome these issues by drastically changing the way we render c
This gives us several new options to significantly optimize scenes with large numbers of lights, allowing us to rende
hundreds (or even thousands) of lights with an acceptable framerate. The following image is a scene with 1847 poi
rendered with deferred shading (image courtesy of Hannes Nevalainen); something that wouldn't be possible with
rendering.



Deferred shading is based on the idea that we *defer* or *postpone* most of the heavy rendering (like lighting) to a lat
Deferred shading consists of two passes: in the first pass, called the geometry pass, we render the scene once and
kinds of geometrical information from the objects that we store in a collection of textures called the G-buffer; think
position vectors, color vectors, normal vectors, and/or specular values. The geometric information of a scene store
buffer is then later used for (more complex) lighting calculations. Below is the content of a G-buffer of a single fram



Close X

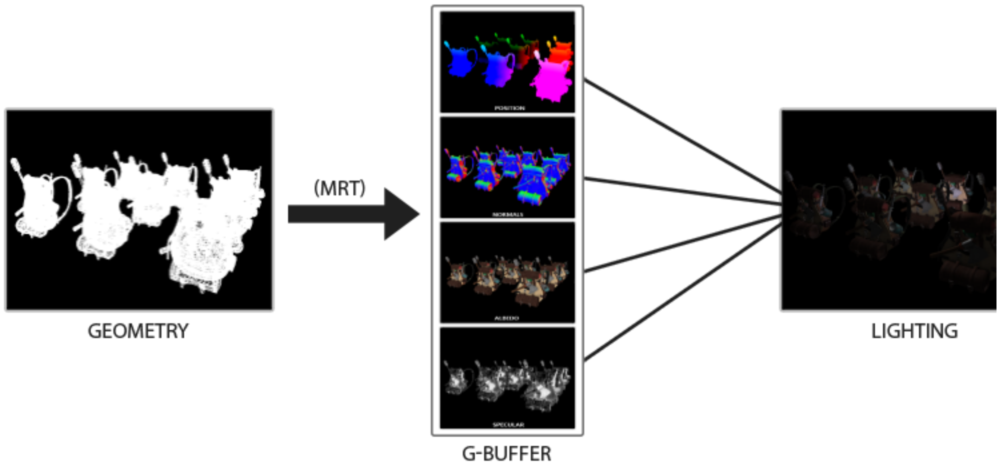
[illegible] we render a screen-filled qua
red in the G-buffer; pixel by p
ler to the fragment shader, v
e exactly the

Privacy

0x1de59bd9e52521a
46309474f83725315
33bd7c43

take all required input variables from the corresponding G-buffer textures, instead of the vertex shader (plus some
variables).

The image below nicely illustrates the process of deferred shading.



A major advantage of this approach is that whatever fragment ends up in the G-buffer is the actual fragment infor
that ends up as a screen pixel. The depth test already concluded this fragment to be the last and top-most fragme
ensures that for each pixel we process in the lighting pass, we only calculate lighting once. Furthermore, deferred r
opens up the possibility for further optimizations that allow us to render a much larger amount of light sources co
forward rendering.

It also comes with some disadvantages though as the G-buffer requires us to store a relatively large amount of sce
its texture color buffers. This eats memory, especially since scene data like position vectors require a high precision
disadvantage is that it doesn't support blending (as we only have information of the top-most fragment) and MSAA
works. There are several workarounds for this that we'll get to at the end of the chapter.

Filling the G-buffer (in the geometry pass) isn't too expensive as we directly store object information like position, c
normals into a framebuffer with a small or zero amount of processing. By using multiple render targets (MRT) we c
all of this in a single render pass.

The G-buffer

The G-buffer is the collective term of all textures used to store lighting-relevant data for the final lighting pass. Let'
moment to briefly review all the data we need to light a fragment with forward rendering:

- A 3D world-space **position** vector to calculate the (interpolated) fragment position variable used for `lightDi`
`viewDir`.
- An RGB diffuse **color** vector also known as albedo.
- A 3D **normal** vector for determining a surface's slope.
- A **specular intensity** float.
- All light source position and color vectors.
- The player or viewer's position vector.

With these (per-fragment) variables at our disposal we are able to calculate the (Blinn-)Phong lighting we're accust
The light source positions and colors, and the player's view position, can be configured using uniform variables, bu
variables are all fragment specific. If we can somehow pass the exact same data to the final deferred lighting pass
calculate the same lighting effects, even though we're rendering fragments of a 2D quad.

There is no limit in OpenGL to what we can store in a texture so it makes sense to store all per-fragment data in on
multiple screen-filled textures of the G-buffer and use these later in the lighting pass. As the G-buffer textures will
same size as the lighting pass's 2D quad, we get the exact same fragment data we'd had in a forward rendering se
this time in the lighting pass; there is a one on one mapping.

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); // keep it black so it doesn't leak into g-buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    gBufferShader.use();
```

Close X

Privacy

```
    // 2. lighting pass: use g-buffer to calculate the scene's lighting
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    lightingPassShader.use();
    BindAllGBufferTextures();
    SetLightingUniforms();
    RenderQuad();
}
```

The data we'll need to store of each fragment is a **position** vector, a **normal** vector, a **color** vector, and a **specular in**
value. In the geometry pass we need to render all objects of the scene and store these data components in the G-bu
can again use multiple render targets to render to multiple color buffers in a single render pass; this was briefly dis
the Bloom chapter.

For the geometry pass we'll need to initialize a framebuffer object that we'll call `gBuffer` that has multiple color b
attached and a single depth renderbuffer object. For the position and normal texture we'd preferably use a high-pre
texture (16 or 32-bit float per component). For the albedo and specular values we'll be fine with the default texture
(8-bit precision per component). Note that we use `GL_RGBA16F` over `GL_RGB16F` as GPUs generally prefer 4-com
formats over 3-component formats due to byte alignment; some drivers may fail to complete the framebuffer othe

```
unsigned int gBuffer;
glGenFramebuffers(1, &gBuffer);
glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
unsigned int gPosition, gNormal, gColorSpec;

// - 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);

// - 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,
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_ATTACHM
glDrawBuffers(3, attachments);

// then also add render buffer object as depth buffer and check for completeness.
[...]
```

Since we use multiple render targets, we have to explicitly tell OpenGL which of the color buffers associated with GI
we'd like to render to with `glDrawBuffers`. Also interesting to note here is we combine the color and specular in
data in a single **RGBA** texture; this saves us from having to declare an additional color buffer texture. As your deferr
shading pipeline gets more complex and needs more data you'll quickly find new ways to combine data in individua

Next we need to render into the G-buffer. Assuming each object has a diffuse, normal, and specular texture we'd us
something like the following fragment shader to render into the G-buffer:

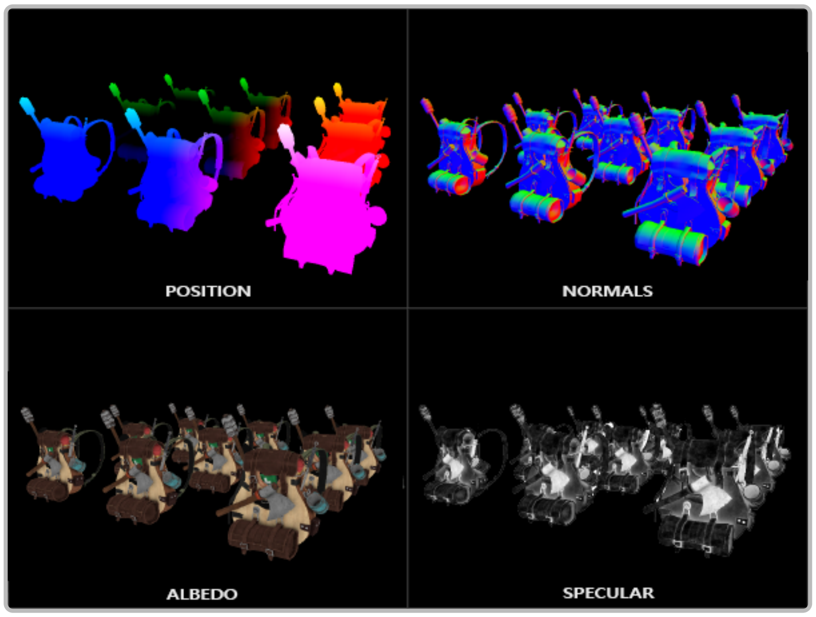
```
#version 330 core
layout (location = 0) out vec3 gPosition;
layout (location = 1) out vec3 gNormal;
layout (location = 2) out vec4 gAlbedoSpec;

in vec2 TexCoords;
in vec3 FragPos;
in vec3 Normal;
```

```
{
    // 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;
}
```

As we use multiple render targets, the layout specifier tells OpenGL to which color buffer of the active framebuffer
to. Note that we do not store the specular intensity into a single color buffer texture as we can store its single float
the alpha component of one of the other color buffer textures.

Keep in mind that with lighting calculations it is extremely important to keep all relevant variables in the sam
coordinate space. In this case we store (and calculate) all variables in world-space.

If we'd now were to render a large collection of backpack objects into the `gBuffer` framebuffer and visualize its co
projecting each color buffer one by one onto a screen-filled quad we'd see something like this:



Try to visualize that the world-space position and normal vectors are indeed correct. For instance, the normal vecto
to the right would be more aligned to a red color, similarly for position vectors that point from the scene's origin to
As soon as you're satisfied with the content of the G-buffer it's time to move to the next step: the lighting pass.

The deferred lighting pass

With a large collection of fragment data in the G-Buffer at our disposal we have the option to completely calculate
final lit colors. We do this by iterating over each of the G-Buffer textures pixel by pixel and use their content as inpu
lighting algorithms. Because the G-buffer texture values all represent the final transformed fragment values we on
do the expensive lighting operations once per pixel. This is especially useful in complex scenes where we'd easily inv
multiple expensive fragment shader calls per pixel in a forward rendering setting.

For the lighting pass we're going to render a 2D screen-filled quad (a bit like a post-processing effect) and execute
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();
```

Close X

Privacy

We bind all relevant textures of the G-buffer before rendering and also send the lighting-relevant uniform variables
shader.

The fragment shader of the lighting pass is largely similar to the lighting chapter shaders we've used so far. What i
the method in which we obtain the lighting's input variables, which we now directly 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;
};
const int NR_LIGHTS = 32;
uniform Light lights[NR_LIGHTS];
uniform vec3 viewPos;

void main()
{
    // retrieve data from G-buffer
    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);
    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);
}
```

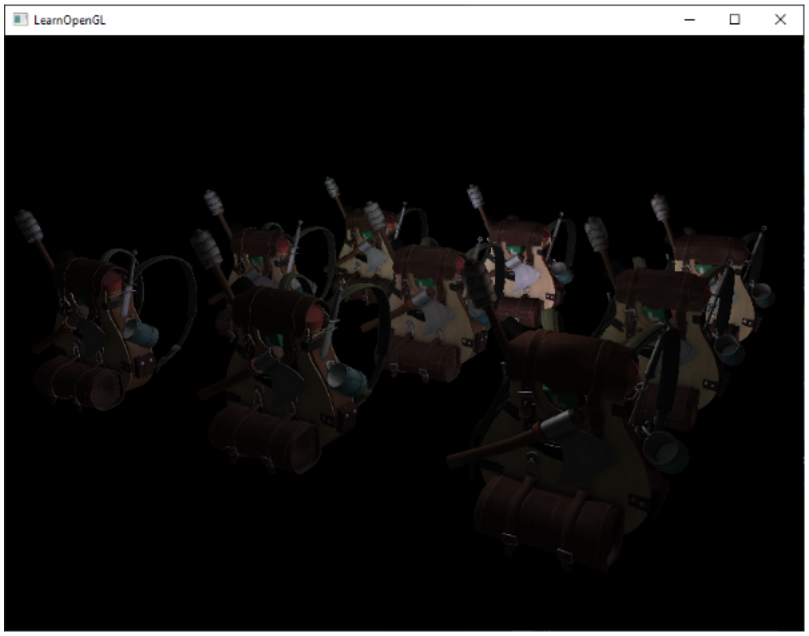
The lighting pass shader accepts 3 uniform textures that represent the G-buffer and hold all the data we've stored
geometry pass. If we were to sample these with the current fragment's texture coordinates we'd get the exact same
values as if we were rendering the geometry directly. Note that we retrieve both the `Albedo` color and the `Specu`
intensity from the single `gAlbedoSpec` texture.

As we now have the per-fragment variables (and the relevant uniform variables) necessary to calculate Blinn-Phong
we don't have to make any changes to the lighting code. The only thing we change in deferred shading here is the
obtaining lighting input variables.

Running a simple demo with a total of 32 small lights looks a bit like this:

Close X

Privacy



One of the disadvantages of deferred shading is that it is not possible to do blending as all values in the G-buffer ar single fragments, and blending operates on the combination of multiple fragments. Another disadvantage is that c shading forces you to use the same lighting algorithm for most of your scene's lighting; you can somehow alleviate by including more material-specific data in the G-buffer.

To overcome these disadvantages (especially blending) we often split the renderer into two parts: one deferred renc part, and the other a forward rendering part specifically meant for blending or special shader effects not suited for rendering pipeline. To illustrate how this works, we'll render the light sources as small cubes using a forward rende light cubes require a special shader (simply output a single light color).

Combining deferred rendering with forward rendering

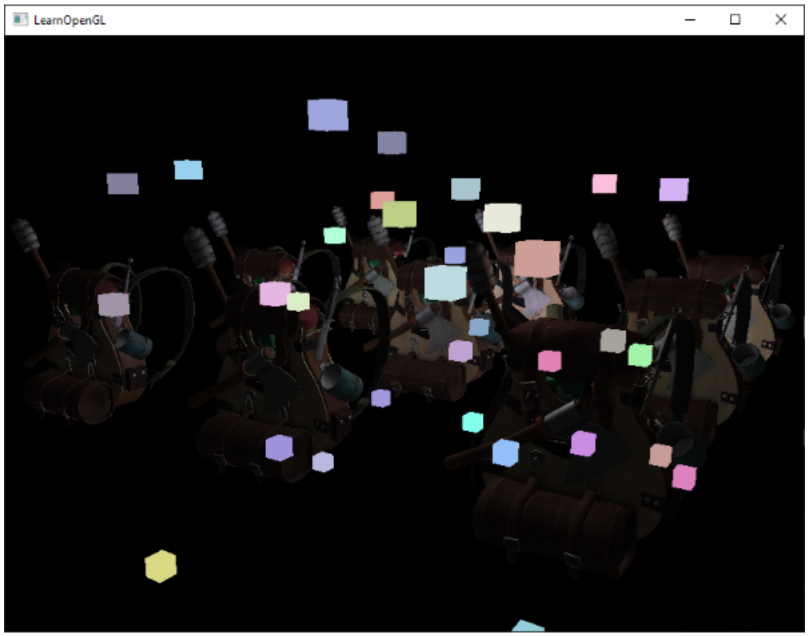
Say we want to render each of the light sources as a 3D cube positioned at the light source's position emitting the c light. A first idea that comes to mind is to simply forward render all the light sources on top of the deferred lighting the end of the deferred shading pipeline. So basically render the cubes as we'd normally do, but only after we've fin deferred rendering operations. 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();
}
```

However, these rendered cubes do not take any of the stored geometry depth of the deferred renderer into accoun as a result, always rendered on top of the previously rendered objects; this isn't the result we were looking for.

Close X

Privacy



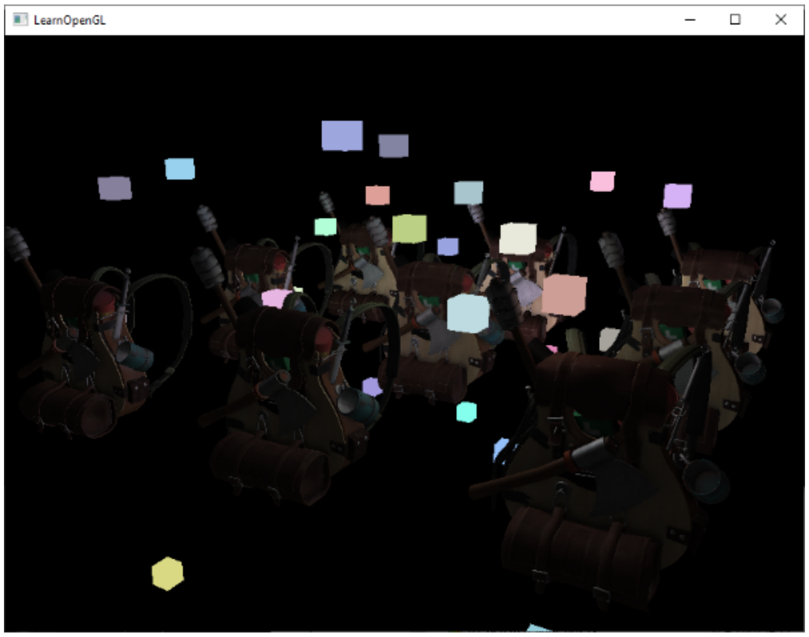
What we need to do, is first copy the depth information stored in the geometry pass into the default framebuffer's d
buffer and only then render the light cubes. This way the light cubes' fragments are only rendered when on top of t
previously rendered geometry.

We can copy the content of a framebuffer to the content of another framebuffer with the help of glBlitFramebu
function we also used in the anti-aliasing chapter to resolve multisampled framebuffers. The glBlitFramebuffe
function allows us to copy a user-defined region of a framebuffer to a user-defined region of another framebuffer.

We stored the depth of all the objects rendered in the deferred geometry pass in the gBuffer FBO. If we were to c
content of its depth buffer to the depth buffer of the default framebuffer, the light cubes would then render as if al
scene's geometry was rendered with forward rendering. As briefly explained in the anti-aliasing chapter, we have to
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
[...]
```

Here we copy the entire read framebuffer's depth buffer content to the default framebuffer's depth buffer; this can
be done for color buffers and stencil buffers. If we then render the light cubes, the cubes indeed render correctly ov
scene's geometry:



You can find the full source code of the demo here.

With this approach we can easily combine deferred shading with forward shading. This is great as we can now still a
blending and render objects that require special shader effects, something that isn't possible in a pure deferred ren
context.

A larger number of lights

What deferred rendering is often praised for, is its ability to render an enormous amount of light sources without a
on performance. Deferred rendering by itself doesn't allow for a very large amount of light sources as we'd still hav
calculate each fragment's lighting component for each of the scene's light sources. What makes a large amount of
sources possible is a very neat optimization we can apply to the deferred rendering pipeline: that of light volumes.

Normally when we render a fragment in a large lit scene we'd calculate the contribution of **each** light source in a sc
regardless of their distance to the fragment. A large portion of these light sources will never reach the fragment, so
waste all these lighting computations?

The idea behind light volumes is to calculate the radius, or volume, of a light source i.e. the area where its light is a
reach fragments. As most light sources use some form of attenuation, we can use that to calculate the maximum d
radius their light is able to reach. We then only do the expensive lighting calculations if a fragment is inside one or
these light volumes. This can save us a considerable amount of computation as we now only calculate lighting whe
necessary.

The trick to this approach is mostly figuring out the size or radius of the light volume of a light source.

Calculating a light's volume or radius

To obtain a light's volume radius we have to solve the attenuation equation for when its light contribution becomes
the attenuation function we'll use the function introduced in the light casters chapter:

$$F_{light} = \frac{I}{K_c + K_l * d + K_q * d^2}$$

What we want to do is solve this equation for when F_{light} is `0.0`. However, this equation will never exactly reach th
`0.0`, so there won't be a solution. What we can do however, is not solve the equation for `0.0`, but solve it for a brigh
value that is close to `0.0` but still perceived as dark. The brightness value of $5/256$ would be acceptable for this ch
demo scene; divided by 256 as the default 8-bit framebuffer can only display that many intensities per component.

> The attenuation function used is mostly dark in its visible range. If we were to limit it to an even darker bright
> than $5/256$, the 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. Of course this always depends on the type
> scene; a higher brightness threshold results in smaller light volumes and thus a better efficiency, but can pro
> noticeable artifacts where lighting seems to break at a volume's borders.

Close X

The attenuation equation we have to solve becomes:

Privacy

Here I_{max} is the light source's brightest color component. We use a light source's brightest color component as so
equation for a light's brightest intensity value best reflects the ideal light volume radius.

From here on we continue solving the equation:

$$\frac{5}{256} * Attenuation = I_{max}$$

$$5 * Attenuation = I_{max} * 256$$

$$Attenuation = I_{max} * \frac{256}{5}$$

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

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

The last equation is an equation of the form $ax^2 + bx + c = 0$, which we can solve using the quadratic equation:

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

This gives us a general equation that allows us to calculate x i.e. the light volume's radius for the light source giver
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) * light
  / (2 * quadratic);
```

We calculate this radius for each light source of the scene and use it to only calculate lighting for that light source i
fragment is inside the light source's volume. Below is the updated lighting pass fragment shader that takes the cal
light volumes into account. Note that this approach is merely done for teaching purposes and not viable in a practi
as we'll soon discuss:

```
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
            [...]
        }
    }
}
```

The results are exactly the same as before, but this time each light only calculates lighting for the light sources in v
volume it resides.

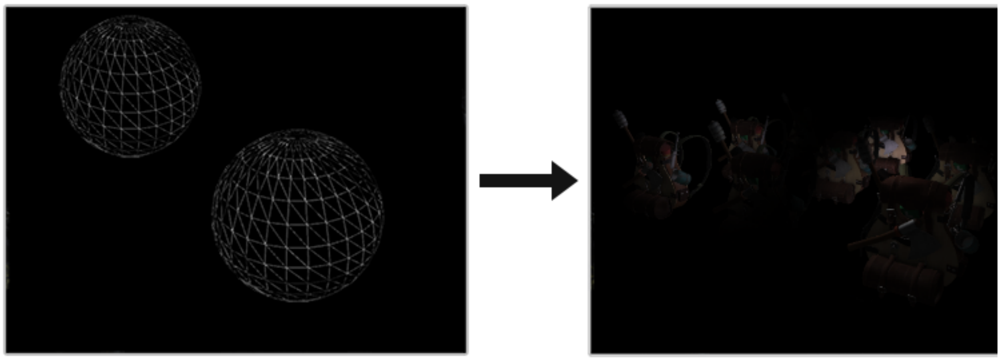
You can find the final source code of the demo here.

How we really use light volumes

The fragment shader shown above doesn't really work in practice and only illustrates how we can *sort of* use a ligh
to reduce lighting calculations. The reality is that your GPU and GLSL are pretty bad at optimizing loops and branc
reason for this is that shader execution on the GPU is highly parallel and most a Close X tures have a requirement th
large collection of threads they need to run the exact same shader code for it to [illegible]cient. This often means that
ame for that group of thread
for all light sources!

Privacy

The appropriate approach to using light volumes is to render actual spheres, scaled by the light volume radius. The
these spheres are positioned at the light source's position, and as it is scaled by the light volume radius the sphere
encompasses the light's visible volume. This is where the trick comes in: we use the deferred lighting shader for ren
spheres. As a rendered sphere produces fragment shader invocations that exactly match the pixels the light source
we only render the relevant pixels and skip all other pixels. The image below illustrates this:



This is done for each light source in the scene, and the resulting fragments are additively blended together. The res
the exact same scene as before, but this time rendering only the relevant fragments per light source. This effectivel
the computations from `nr_objects * nr_lights` to `nr_objects + nr_lights`, which makes it incredibly effic
scenes with a large number of lights. This approach is what makes deferred rendering so suitable for rendering a la
number of lights.

There is still an issue with this approach: face culling should be enabled (otherwise we'd render a light's effect twice
when it is enabled the user may enter a light source's volume after which the volume isn't rendered anymore (due t
face culling), removing the light source's influence; we can solve that by only rendering the spheres' back faces.

Rendering light volumes does take its toll on performance, and while it is generally much faster than normal defer
shading for rendering a large number of lights, there's still more we can optimize. Two other popular (and more eff
extensions on top of deferred shading exist called deferred lighting and tile-based deferred shading. These are ever
efficient at rendering large amounts of light and also allow for relatively efficient MSAA.

Deferred rendering vs forward rendering

By itself (without light volumes), deferred shading is a nice optimization as each pixel only runs a single fragment s
compared to forward rendering where we'd often run the fragment shader multiple times per pixel. Deferred rende
come with a few disadvantages though: a large memory overhead, no MSAA, and blending still has to be done with
rendering.

When you have a small scene and not too many lights, deferred rendering is not necessarily faster and sometimes
slower as the overhead then outweighs the benefits of deferred rendering. In more complex scenes, deferred rende
quickly becomes a significant optimization; especially with the more advanced optimization extensions. In addition,
render effects (especially post-processing effects) become cheaper on a deferred render pipeline as a lot of scene in
already available from the g-buffer.

As a final note I'd like to mention that basically all effects that can be accomplished with forward rendering can als
implemented in a deferred rendering context; this often only requires a small translation step. For instance, if we w
normal mapping in a deferred renderer, we'd change the geometry pass shaders to output a world-space normal e
from a normal map (using a TBN matrix) instead of the surface normal; the lighting calculations in the lighting pas
need to change at all. And if you want parallax mapping to work, you'd want to first displace the texture coordinate
geometry pass before sampling an object's diffuse, specular, and normal textures. Once you understand the idea be
deferred rendering, it's not too difficult to get creative.

Additional resources

- Tutorial 35: Deferred Shading - Part 1: a three-part deferred shading tutorial by OGLDev.
- Deferred Rendering for Current and Future Rendering Pipelines: slides by Andrew Lauritzen discussing high-l
  based deferred shading and deferred lighting.

18 Share Best Ne

tony perez
6 years ago
I want to implement this technique with bloom and hdr but I realized you do the lighting calculations in tl
instead of the first one so I can extract the bright color, can I divide the deferred_shading.fs shader in 2 s
calculate the lighting, extract the bright, do the bloom and then render everything with 5 color attachmen
I'm not sure what method would be the efficient one to achieve this. Thanks!

63 0 • Reply • Share ›

anossov
7 years ago edited
I think it would be useful to note that you can't use any `glClearColor` other than black during the geometry
up in all attachments and screws with everything. So if you want a custom color, you need to either `glCle`
times every frame and add `if (Normal == vec3(0.0, 0.0, 0.0){ discard; }` to the lighting fragment shader,
once to black and return early with a custom color if normal is black. Normals can't be black naturally sir
normalized.

And another thing — do NOT forget to disable GL_BLEND.

14 0 • Reply • Share ›

nikku ➦ anossov
4 years ago
3 years comment and it helped me a lot. Thank you!

0 0 • Reply • Share ›

Quinton Gordon ➦ nikku
a year ago
3 years later and it helps me now

1 0 • Reply • Share ›

xaviboom ➦ Quinton Gordon
a year ago
....and me too! XD

0 0 • Reply • Share ›

Sandor Muranyi ➦ anossov
5 years ago
If I were find this comment a week ago, that would save a LOT of debugging

0 1 • Reply • Share ›

Nabil
5 years ago edited
Hey Joey! I came across this little gem: https://www.3dgep.com/forwa... . It compares regular old Forwa
Deferred Shading, and this new technique call Forward+. Forward+ is literally an upgraded Forward rende
culling lights on the GPU. From what I've read, it's a very compelling algorithm that solves most of the iss
Deferred (primarily allowing transparency and multiple materials), while still allowing hundreds to thousa
Here is the original paper (it's very short): https://takahiroharada.file...

If you want to see it in action, here is a basic example that I found, https://github.com/bcrusco/... (the co
look a little familiar :P). I just implemented it myself, and I have to say, it's really easy to understand. The
one needs to read up on Compute Shaders :)

Anyway, my point is, for those who like your tutorial format, I think doing an article on Forward+ would be
well-received (since there is no other tutorial online atm), and may not require as much time to complete

10 0 • Reply • Share ›

Sein Lee
2 years ago
Always thank you, Joey! I've researched light pre-pass rendering since read this tutorial, and finally imple
few days ago. It is definitely boost my engine's rendering performance, it does not give enough satisfa

Close X

Privacy

View — uploads.disquscdn.com
View — uploads.disquscdn.com
View — uploads.disquscdn.com
View — uploads.disquscdn.com

7 0 • Reply • Share ›

JoeyDeVries Sensei ➜ Sein Lee
2 years ago
Wow, looks amazing; well done!

3 0 • Reply • Share ›

anossov
7 years ago edited
Another terrible story: blitting depth to a framebuffer that has `GL_FRAMEBUFFER_SRGB` enabled gamma-correc
which is obviously wrong. Maybe it's a driver bug. This took me a *long* time to figure out.

Edit: apparently, blitting to sRGB framebuffers was only described in the 4.4 specification, so I guess eve
implementation still does whatever it wants.

7 0 • Reply • Share ›

Lucy Dalzell ➜ anossov
3 months ago
THANK YOU SO MUCH
i was so confused for SO long omfg

0 0 • Reply • Share ›

Eugen Bondarev
2 years ago
Thanks for your tutorials, everything works nicely!
View — uploads.disquscdn.com

4 0 • Reply • Share ›

oddstoneGames ➜ Eugen Bondarev
a year ago
wow amazing!

0 0 • Reply • Share ›

Abdulla
7 years ago
Another great tutorial.
But I can not recognize how you retrieve a correct fragment position from the position texture using this
vec3 FragPos = texture(gPosition, TexCoords).rgb;
the rgb values ranges from 0 to 1 which means if we have an object at [-5,-1,-3] the corresponding rgb va
[0,0,0] and if we have another object at [-2,-4,0] it will have the same rgb value [0,0,0] also the objects at [
[5,3,2] will be decoded to same rgb values [1,1,1]

4 0 • Reply • Share ›

JoeyDeVries Sensei ➜ Abdulla
7 years ago
You're right, normally it will all get clamped between `0.0` and `1.0`. However, for storing the colo
information we use something called a **floating point** colorbuffer as we set the colorbuffer's i
format to `GL_RGBA16F` (note the F). If we use a floating point format it removes the clamping of
shader outputs and we can directly store their values.

Keep in mind that if you do not use a floating point format, you'd indeed need to map all value
`0.0` and `1.0` first and later re-transform them back to their original values once you need them
similar to normal maps as we store values ranging between [-1.0,1.0] to color values between
later re-map them back to their original normal vectors when we need them)

While not exactly related to deferred rendering, I go a bit more in depth about this in the HDR t

3 0 • Reply • Share ›

Close X

utorials.

Privacy

3 0 • Reply • Share ›

Wada Don
3 years ago
Granted I'm implementing this in D3D11 but this helped give me a general overview of how to implement
own renderer - I currently have 4 rendertargets including a deferred lightspace texture for a global directi
and results are pretty good, if I'm not using msaa (which causes glowing artifacts when objects are in sh
how the samples are resolved). Great work as always though!
View — uploads.disquscdn.com

2 0 • Reply • Share ›

Tommy Chen
6 years ago edited
Hi! First I'm so appreciate of your excellent tutorial!

I found that the GL_RGB16F is not as good as GL_RGBA16F in terms of compatibility. I have 3 GPUs at m
the GL_RGB16F format works on only 2 of them:

1) Intel HD 520 (Mobile Skylake) + Ubuntu 16.04: Works
2) Intel HD 4600 (Desktop Haswell) + Ubuntu 15.04: Does not work; binding it to FBO triggers error 1286
Framebuffer Operation.)
3) GeForce GTX 750 Ti + Ubuntu 15.04: Works

For Intel HD 4600, GL_RGB16F must be changed to GL_RGBA16F to be used as an output from the Fram
(vec3 in the fragment shader should be changed to vec4 accordingly.)

A similar question is also asked about Haswell here: (https://www.opengl.org/disc...)

2 0 • Reply • Share ›

bittersweet → Tommy Chen
3 years ago
I can't seem to make `GL_RGB16F` work on my GeForce GTX 750 Ti, but `GL_RGBA16F` does its job. Ol

0 0 • Reply • Share ›

KenLee → Tommy Chen
6 years ago
Hi, Tommy ~ Have you done the AdvancedOpenGL - AntiAliasing Chapter? I found on my Ubu
Intel HD 4600 environment, when I call glBlitFramebuffer(...) function with GL_READ_FRAMEB
MSAA FB and GL_DRAW_FRAMEBUFFER as default buffer(0), I get GL_INVALID_OPERATION。
same codes work just fine on my GT650M + Windows7 and Intel HD4000+ Windows. Is this a
driver on Intel driver on Linux? Thanks for your question anyway!

0 0 • Reply • Share ›

Ferruh Ali Senan → Tommy Chen
6 years ago
Do you know if a similar issue occurs with GL_RGBA32F?

0 0 • Reply • Share ›

Tommy Chen → Ferruh Ali Senan
6 years ago
I tested it on Intel HD 4000 (that's 1 generation older than HD 4600) and yes, there
issue.
GL_RGB16F and GL_RGB32F gives error 1286 while GL_RGBA16F and GL_RGBA32

20 0 • Reply • Share ›

Jon Kinsey
6 years ago
I modified this tutorial so I could compare a normal single pass render of the scene with the deferred ren
enabled back face culling and zoomed in slightly (z = 4.0f) which meant the front man obscures about 4
6 men in frame. The object order is important as if the men are rendered front->back then the obscured t
get discarded by the depth test. Here are the results:

Close X

Privacy

Single pass - front to back 13.8 ms

Intel on board card:
Deferred - back to front 27.1 ms
Deferred - front to back 26.8 ms
Single pass - back to front 43.4 ms
Single pass - front to back 25.7 ms

As you would probably expect the single pass with front->back objects is fastest. The deferred rendering
between the back->front tests though.

2 0 • Reply • Share ›

KenLee ➦ Jon Kinsey
6 years ago
Could you tell me how many lights you were rendering in your scene? And are these result wit
light volumes?

0 0 • Reply • Share ›

Jon Kinsey ➦ KenLee
6 years ago
Can't remember, but likely the first example - 32 lights without volumes.

0 0 • Reply • Share ›

Delicious Points
6 months ago
I hate to sound like a Debbie downer but another disadvantage to deferred rendering, is that we need to
own MSAA instead of simply hinting that. And that is another whole set of headaches to deal with as cu
to halt my PBR and SSAO because this entire time the framebuffer is displaying jaggies galore!!

1 0 • Reply • Share ›

Delicious Points ➦ Delicious Points
6 months ago
Well after sleepless nights I came to the conclusion that Multi-sample anti aliasing is not pos
deferred rendering. The reason being is that all lighting calculations are all after the textures '
the intermediary framebuffer. The solution here which I'm not enthusiastic about is the newer
approximate anti aliasing. Yeah I know it's done post processing but it just doesn't look as sat
Multi-sampling.

0 0 • Reply • Share ›

Botondar ➦ Delicious Points
6 months ago
There is a solution to doing MSAA with deferred rendering described by Nvidia. Th
also output the fragments that only have partial coverage to the G-buffer, which yo
the lighting pass to determine whether to do the lighting calculations per-fragment
sample (essentially doing selective supersampling). It's probably not practical thou
know of any products that actually shipped with this technique.

FXAA is also a pretty old technique at this point, the industry has moved onto temp
aliasing techniques which are the new standard (for better or for worse).

0 0 • Reply • Share ›

Delicious Points ➦ Botondar
6 months ago
Yeah I read that article. I still think it's not worth it as it sounds like a tor
do for a little improvement. I also heard of another method where you t
cases and outlines to apply MSAA to it only. And I still don't think FXAA
cracked up to be. Maybe I need to look into TXAA.

0 0 • Reply • Share ›

MahanGM
4 years ago edited
It was really straight forward to implement thanks to your explanatio Close X ould have been nice if you cou

Privacy

Final Boss

4 years ago edited

Not sure if this has been mentioned yet and I sure you already know this but a common optimization for distance thresholds is to dispense with the square root and just compare numbers in squared space, if y

if (distance2 <= radius2) // Store the squared radius instead

I know looping through the lights and checking distance isn't part of the final implementation but it may l worth mentioning since the optimization is so common.

1 0 • Reply • Share ›

Avatar This comment was deleted.

Avatar This comment was deleted.

JoeyDeVries Sensei ➜ Guest

4 years ago

Hmm not sure, it may be that the light volume sphere mesh isn't properly tessellate of its edges don't fully connect (or over-connect) to the other edges in which case overdraw in these edge regions, causing the lighting values to double there as they rendered?

1 0 • Reply • Share ›

Avatar This comment was deleted.

JoeyDeVries Sensei ➜ Guest

4 years ago

Alright, great to hear! It's probably the first and last circle used to genera sphere where you have an extra set of vertices on the start/last circle ed may involve some x-1,y-1 statements. And if it isn't, you can indeed alwa sphere from a 3D modeling package :)

1 0 • Reply • Share ›

Avatar This comment was deleted.

JoeyDeVries Sensei ➜ Guest

4 years ago

That's part of any OpenGL application lifecycle:

View — uploads.disquscdn.com

Best of luck! :D

8 0 • Reply • Share ›

Avatar This comment was deleted.

JoeyDeVries Sensei ➜ Guest

4 years ago

Hahah, it's always the little things!

1 0 • Reply • Share ›

Avatar This comment was deleted.

JoeyDeVries Sensei ➜ Guest

5 years ago

In theory the color of specular reflections comes from the light only, and not the object. So we multiplier for the specular reflection as we already have the light's color available. Thus a sing value is sufficient.

Close X

Privacy

How to combine deferred shading with HDR?

1 0 • Reply • Share ›

JoeyDeVries Sensei ➜ zchars
6 years ago
Make sure the output buffer of the lighting stage (where you calculate all the lighting) and/or
stage is using a floating point framebuffer, s.t. it can handle all intensities of light. Then you d
on this buffer after you've rendered each light for color (and gamma) correction.

0 0 • Reply • Share ›

zchars ➜ JoeyDeVries
6 years ago
My current problem is that I want to draw skybox in forward pass after deferred sh
glblitframebuffer. How can I apply HDR after both deferred and forward pass? Do I
a depth texture to my hdr FBO?

0 0 • Reply • Share ›

JoeyDeVries Sensei ➜ zchars
6 years ago
Another approach is to simply directly do the HDR tonemapping in the d
lighting shader (together with gamma correction after the tonemapping
You then do the same in your skybox shader such that the end result is
mapped (if your skybox is also HDR, otherwise it won't help you much);
both shaders are then in LDR as you're used to and you can use those h
like. Either output both passes to another framebuffer texture for post-p
render/blit them directly to the default framebuffer.

If you do want to still use HDR in the post-processing pipeline (for bloor
instance) or for other reasons after the deferred and forward pass you'c
render all passes you want to keep in HDR to (another) floating point fra
Then you do the tonemapping at the stage before you render to a LDR fı
It's usually a bit of juggling around with which framebuffers are HDR anc
aren't, and then properly tonemap them when transitioning to a LDR frar

0 0 • Reply • Share ›

zchars ➜ JoeyDeVries
6 years ago
Thanks for the quick reply!
My current approach is perfomed as follow:
===init===
setup gbuffer fbo
attach depth renderbuffer and other texture
setup hdr fbo
attach float point color texture and depth renderbuffer
===draw===
bind gbuffer
draw gbuffer
bind hdr fbo
use glblitframebuffer to copy depth from gbuffer to hdr fbo
set gldepthfunc to gl_less
set gldepthrande to specify the range to draw skybox
draw skybox in forwad pass to hdr fbo
bind default framebuffer
attach hdr color texture, then draw to screen

Now the HDR is working with deferred shading, but I couldn't render my
hdr fbo. Which step may go wrong?

0 0 • Reply • Share ›

Rupesh
7 years ago
Now if I have deferred rendering working as expected, to enable bloc Close X ct using deferred rendering, I
ne more FBO. then use that to
: will be kind of three pass ap

Privacy

Second pass : deferred lighting pass
Third pass : any postprocessing (bloom, color correction etc etc)

would love to hear from you ... :)

1 0 • Reply • Share ›

JoeyDeVries Sensei ➜ Rupesh
7 years ago
Yep :)

1 0 • Reply • Share ›

John
7 years ago
I've been following a few of these tutorials, really like the bloom/HDR/SSAO ones and I've been merging
own deferred shader. Now I've been trying to add a skybox at the end of the rendering, after the result of
has been written to a final texture. And I've been trying to blit the depth buffer of my geometry FBO as sh
tutorial, however it just throws GL_INVALID_OPERATION because it thinks the main FBO and my FBO has
formats, I use a depth24_stencil8 attachment and I've used glfw window hints to try and get the same bi
default FBO but with no luck. I was wondering if there's something I'm missing here since I never see you
formats for framebuffers in your tutorials and it just works, once again great tutorials!

1 0 • Reply • Share ›

Load more comments