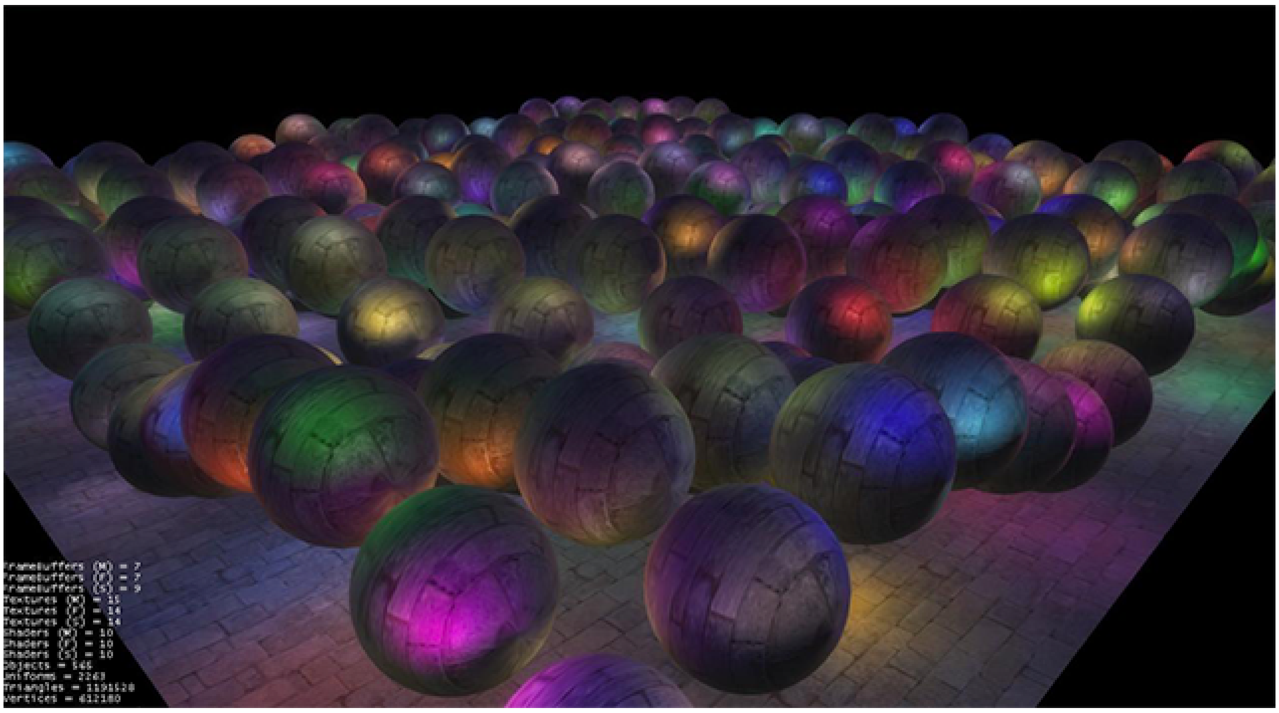
Forward Rendering vs. Deferred Rendering

gamedevelopment.tutsplus.com/forward-rendering-vs-deferred-rendering--gamedev-12342a

If you're a developer of 3D games, then you've probably come across the terms *forward rendering* and *deferred rendering* in your research of modern graphics engines. And, often, you'll have to choose one to use in your game. But what are they, how do they differ, and which one should you pick?



```
FrameBuffers (M) = 7
FrameBuffers (F) = 7
FrameBuffers (S) = 9
Textures (M) = 15
Textures (F) = 14
Textures (S) = 14
Shaders (M) = 10
Shaders (F) = 10
Shaders (S) = 10
Objects = 566
Uniforms = 2263
Triangles = 1191528
Vertices = 612180
```

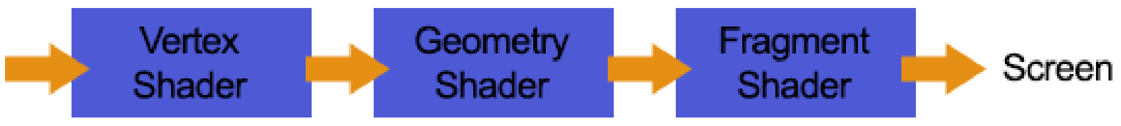
Deferred Rendering for many lights (Image courtesy of Hannes Nevalainen)

Modern Graphics Pipelines

To begin, we need to understand a little bit about modern, or programmable, graphics pipelines.

Back in the day, we were limited in what the video card graphics pipeline had. We couldn't change how it drew each pixel, aside from sending in a different texture, and we couldn't warp vertices once they were on the card. But times have changed, and we now have *programmable graphics pipelines*. We can now send code to the video card to change how the pixels look, giving them a bumpy appearance with normal maps, and adding reflection (and a great deal of realism).

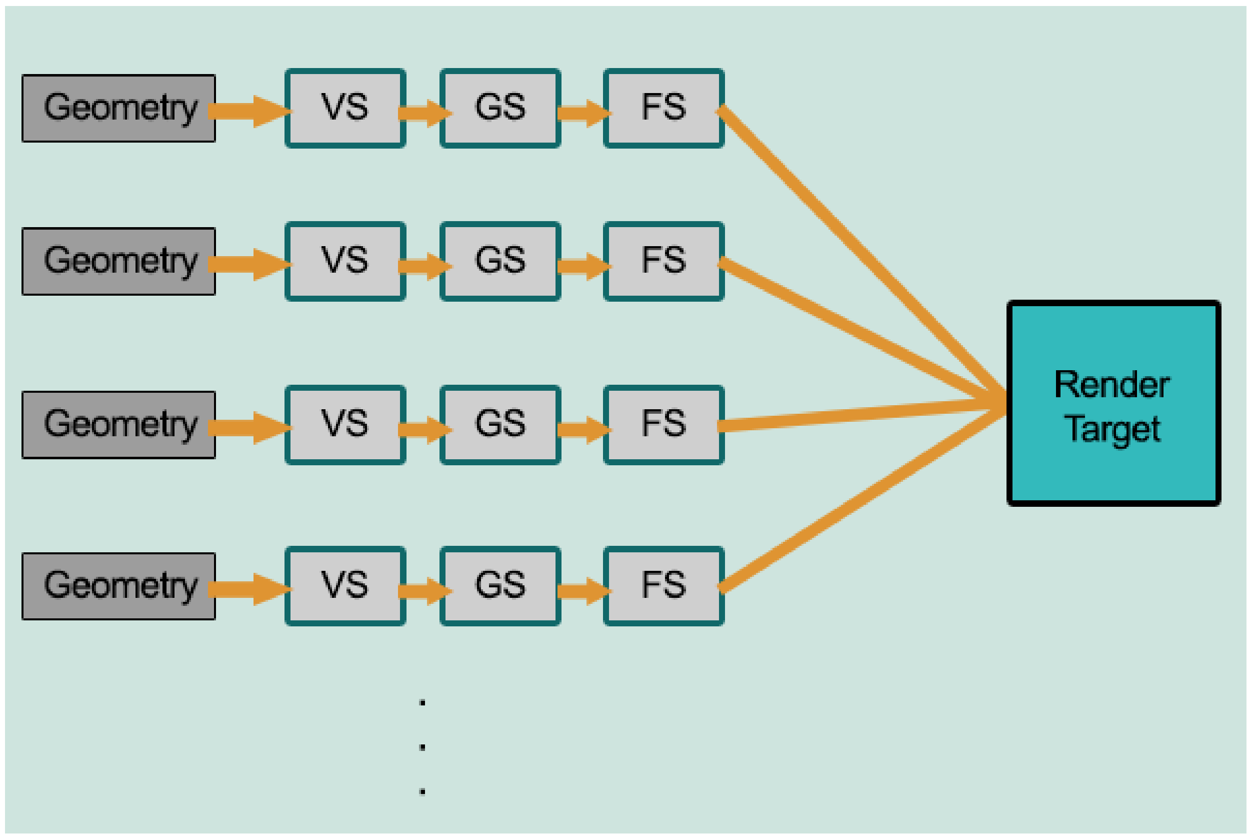
This code is in the form of *geometry*, *vertex*, and *fragment shaders*, and they essentially change how the video card renders your objects.



Simplified view of a programmable graphics pipeline

Forward Rendering

Forward rendering is the standard, out-of-the-box rendering technique that most engines use. You supply the graphics card the geometry, it projects it and breaks it down into vertices, and then those are transformed and split into fragments, or pixels, that get the final rendering treatment before they are passed onto the screen.



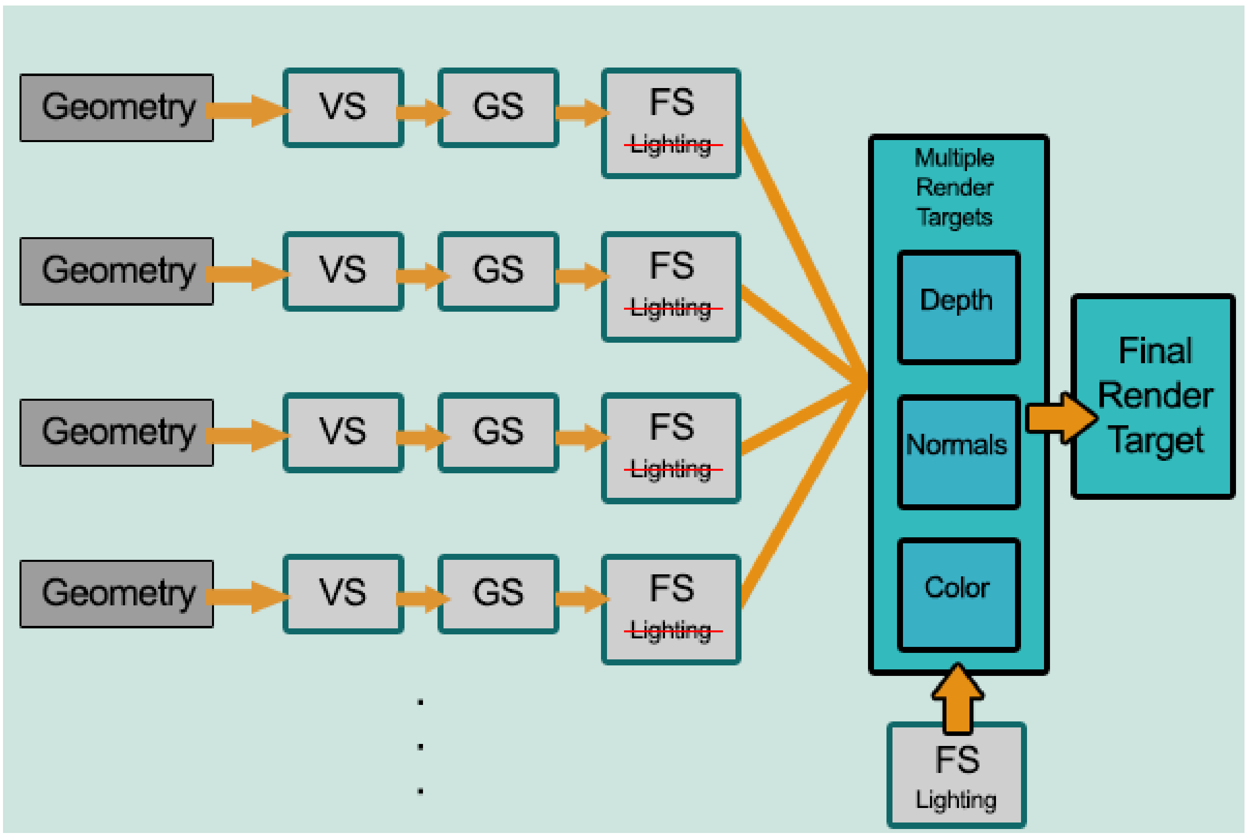
Forward rendering: Geometry shader to vertex shader to fragment Shader

It is fairly linear, and each geometry is passed down the pipe one at a time to produce the final image.

Deferred Rendering

In deferred rendering, as the name implies, the rendering is deferred a little bit until all of the geometries have passed down the pipe; the final image is then produced by applying shading at the end.

Now, why would we do that?



Deferred rendering: Geometry to vertex to fragment shaders. Passed to multiple render targets, then shaded with lighting.

Deferred lighting is a modification of deferred rendering that reduces the size of the G-buffer by using more passes on the scene.

Lighting Performance

Lighting is the main reason for going one route versus the other. In a standard forward rendering pipeline, the lighting calculations have to be performed on every vertex and on every fragment in the visible scene, for every light in the scene.

If you have a scene with 100 geometries, and each geometry has 1,000 vertices, then you might have around 100,000 polygons (a very rough estimate). Video cards can handle this pretty easily. But when those polygons get sent to the fragment shader, that's where the expensive lighting calculations happen and the real slowdown can occur.

Developers try to push as many lighting calculations into the Vertex shader as possible to reduce the amount of work that the fragment shader has to do.

The expensive lighting calculations have to execute for each visible fragment of every polygon on the screen, regardless if it overlaps or is hidden by another polygon's fragments. If your screen has a resolution of 1024x768 (which is, by all means, not very high-res) you have nearly 800,000 pixels that need to be rendered. You could easily reach a million fragment operations every frame. Also, many of the fragments will never make it to the screen because they were removed with depth testing, and thus the lighting calculation was wasted on them.

If you have a million of those fragments and suddenly you have to render that scene again for each light, you have jumped to `[num lights] x 1,000,000` fragment operations per frame! Imagine if you had a town full of street lights where each one is a point-light source...

The formula for estimating this forward rendering complexity can be written, in big O notation, as `O(num_geometry_fragments * num_lights)`. You can see here that the complexity is directly related to the number of geometries and number of lights.

Fragments are potential pixels that will end up on the screen if they do not get culled by the depth test.

Now, some engines optimize this, by cutting out lights that are far away, combining lights, or using light maps (very popular, but static). But if you want dynamic lights and a lot of them, we need a better solution.

Deferred Rendering to the Rescue

Deferred Rendering is a very interesting approach that reduces the object count, and in particular the total fragment count, and performs the lighting calculations on the pixels on the screen, thereby using the resolution size instead of the total fragment count.

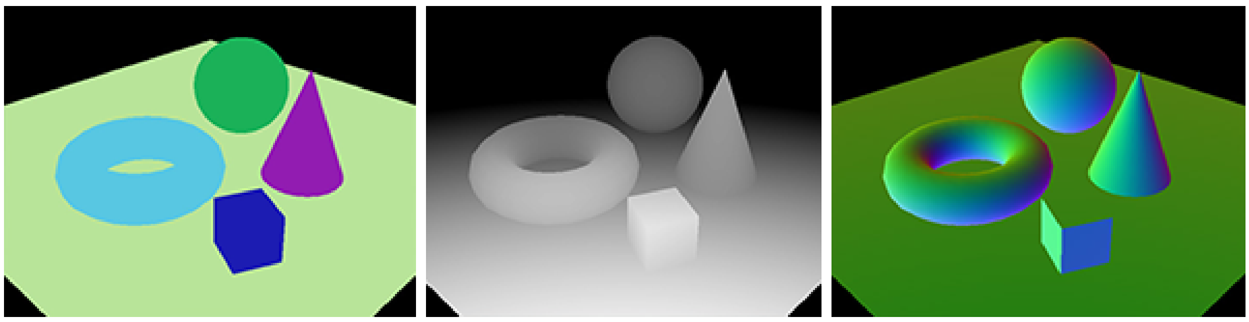
The complexity of deferred rendering, in big O notation, is: `O(screen_resolution * num_lights)`.

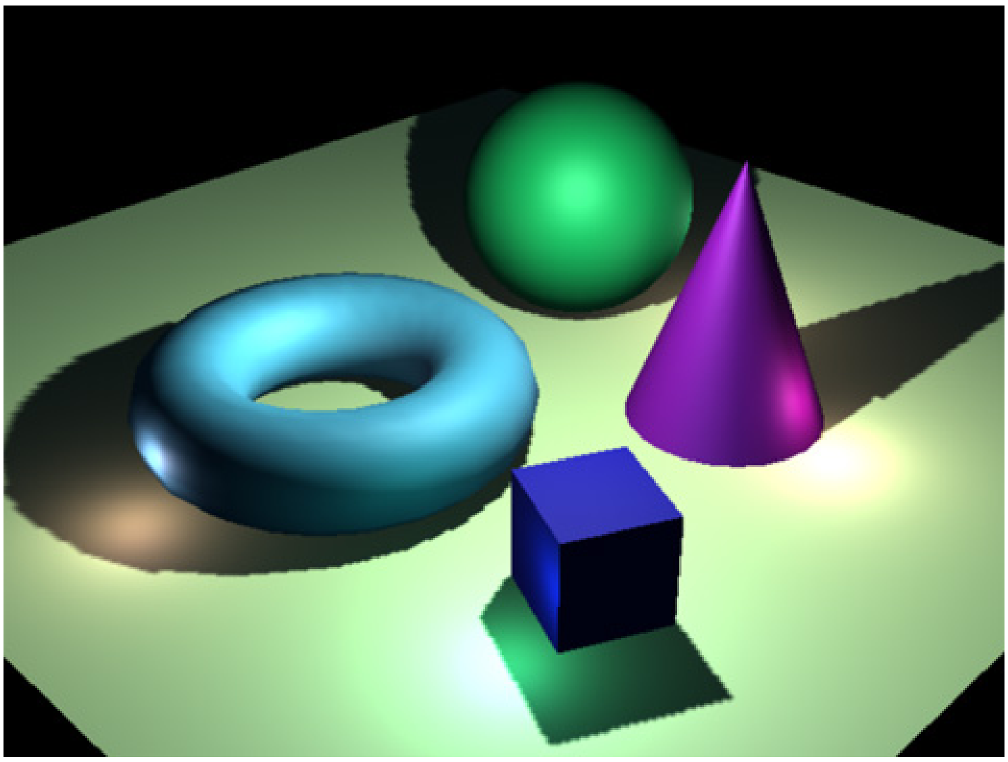
You can see that it now doesn't matter how many objects you have on the screen that determines how many lights you use, so you can happily increase your lighting count. (This doesn't mean you can have unlimited objects—they still have to be drawn to the buffers to produce the final rendering result.)

Let's see how it works.

The Guts of Deferred Rendering

Every geometry is rendered, but without light shading, to several screen space buffers using *multiple render targets*. In particular, the depth, the normals, and the color are all written to separate buffers (images). These buffers are then combined to provide enough information for each light to light the pixels.


Color, Depth, and Normal buffers. (Images by astrofa, via Wikimedia Commons.)


Final lighting (shading) result generated using the three buffers. (Image by astrofa, via Wikimedia Commons.)

By knowing how far away a pixel is, and its normal vector, we can combine the color of that pixel with the light to produce our final render.

Which to Pick?

The short answer is, if you are using many dynamic lights then you should use deferred rendering. However, there are some significant drawbacks:

- This process requires a video card with multiple render targets. Old video cards don't have this, so it won't work on them. There is no workaround for this.
- It requires high bandwidth. You're sending big buffers around and old video cards, again, might not be able to handle this. There is no workaround for this, either.

- You can't use transparent objects. (Unless you combine deferred rendering with Forward Rendering for just those transparent objects; then you can work around this issue.)
- There's no anti-aliasing. Well, some engines would have you believe that, but there are solutions to this problem: edge detection, FXAA.
- Only one type of material is allowed, unless you use a modification of deferred rendering called Deferred Lighting.
- Shadows are still dependent on the number of lights, and deferred rendering does not solve anything here.

If you don't have many lights or want to be able to run on older hardware, then you should stick with forward rendering and replace your many lights with static light maps. The results can still look amazing.