Matt DiGianfilippo
www.dgdesigned.net

In order to begin talking about DirectX12, an introduction to some of the major architectural design changes from DirectX11 is required. Having knowledge of using DirectX11 is a pre-requisite to begin using DirectX12, however, even with this knowledge it does not guarantee an easy transition. If you were to read any tutorial or textbook about DirectX12, they will make the same warning to the reader: Before you use DirectX12, make sure you have previous knowledge and feel comfortable in DirectX11. Previously, the DirectX11 driver handled most of the memory management and resource states which was likely taken for granted by the developers. Behind the scenes, extra allocations were created when a constant buffer was updated, shader switches within a pipeline were tracked, and transitions between readable and writeable textures was handled all by the driver. The driver had no knowledge of your engine's use cases and must handle each of these cases with a broad stroke. This is no longer the case with DirectX12. In order to make the graphics library multi-threaded friendly and use as many threads as the developer wishes, much of this work was pushed to the engine developer.

Previously in DirectX11, any sub-section of a graphics pipeline could be switched around "on the fly", although this was not a free switch. This allowed for developers to change the vertex and pixel shaders, as well as any other programmable stage without care. Other modifiable sections include the blend states (for cases of alpha blending), the stencil description (stencil/depth parameters), and raster states (face windings and fill/wire mode). This is no longer the case with how DirectX12 handles the graphics pipeline. Unknown to the developer, in DirectX11, any of these changes would require the driver to make a new pipeline object with the modified values. This responsibility has been pushed onto the developer. For any small change in the pipeline, a new PipelineState object must be created and used. Just as it was a requirement in DirectX11 to compile a shader, an input assembly layout is required to create a pipeline state. One new requirement is a root signature is also needed.

Graphics memory is a key new area that the engine developer has been tasked with the responsibility of managing. The first introduction to this idea a developer will be faced with is the creation of a root signature, as it is required to create a pipeline. A root signature is a table of DWORDs (32 bits) that either contain data or tell the GPU where to find the data needed by a shader. Even in the most basic of pipelines (one that does not use a constant buffer at any stage), an empty root signature of zero entries must still be created. Each root parameter can be visible to either a single, multiple, or all shader stages and is defined by the developer at creation of the root signature. This allows the GPU and DirectX12 to efficiently prioritize how the data is being used. Microsoft set an artificial max size of 64 DWORDs for a root signature in order to limit the abuse of large tables. While this is an artificial limit, some hardware devices may have a smaller max size causing the table to be moved from faster memory into slower memory. Samplers, to sample textures, must also be defined in the root signature. If the samplers are static, they do not add to the DWORD table size.

The size of each entry varies depending on what is stored. A constant value (of 32-bits in size) costs 1 DWORD. A descriptor table entry is also 1 DWORD. This is because the heap must be set active and the descriptor table entry is merely an offset into the heap. Finally, there is a constant buffer view; which costs 2 DWORDs. This is a full 64-bit address into any constant buffer heap, hence the cost 2

Root Slot	Root Parameter Type	DWORD size	Visible To Stage(s)
0	Constant Buffer View	2	Vertex
1	Constant Buffer View	2	Pixel
2	Descriptor Table	1	Pixel
3	Descriptor Table	1	Pixel
4	Descriptor Table	1	Pixel
5	Descriptor Table	1	Pixel
6	Constant Buffer View	2	Domain
7	Constant Buffer View	2	Vertex
8	Constant Buffer View	2	Pixel
	Total size (DWORD):	14	

DWORDs. There is also a redirection cost for each of these entries that one might want to be aware of. Since the constant value is stored within the root signature, there is no redirection. A descriptor table has two redirections, one to the start of the heap and a second to get to the offset. A constant buffer view has one redirection, to the address stored in the table. I am including a diagram of the root signature I ended up using for my entire engine. Microsoft notes that the root parameter entries should be ordered from most modified to least. Microsoft also recommends using one root signature for the entire program, use case may vary, as changing the root signature will unset any previously set root parameters. Setting active a different pipeline that uses the same root signature does not unset the entries.

Now that root signatures have been introduced, although likely not enough to get a full understanding, we can begin to talk about the idea of heaps. DirectX12 has made the developer responsible for any memory allocations, as well as ensure that the data is not modified until after it has been used by the graphics card. The DirectXTK team at Microsoft has created a great page allocator within their DirectXTK library, which I have borrowed and modified. The main idea is that the graphics memory is handled by a singleton that manages pools of linear allocators. Due to alignment requirements on the GPU, the linear allocators each have their own maximum size per allocation. Each allocation request searches for the correct allocator based on the requested memory size. The allocator then finds a used page with enough space left in it to fit the requested memory or a clean new page (which is a heap) is created to fit the allocation. Each allocation from these pages have reference counting built in. Since I am only using them temporarily (either for uploading data to a permanent resource or using it for a single frame as a constant buffer), the reference count will be zero after the command list has been executed. Once the reference count hits zero, a memory fence is pushed to the graphics card and stored in the linear page. Each frame, we will check every pending page to see if the graphics card has finished using the page or not. If it has, we can re-use this page by adding it back to the allocator's pool and pretend it is a clean and empty page.

This previous section was one of the hardest parts for me to wrap my head around with DirectX12. Even if my engine is single-threaded, the introduction of DirectX12 has made it somewhat multi-threaded and I must be aware of asynchronous events and not overwrite data prematurely. Thankfully, this linear page allocator does an amazing job. Without it, every allocation per object would have to be created in an array based on the number of frames in the swap chain. This would be a major mess as each object would have to be aware of how many frames we are buffering.

I touched upon it briefly, but DirectX11's single ImmediateContext singleton is no longer and is instead replaced by DirectX12's command lists. Command lists themselves are not thread safe and so DirectX12 recommends one command list per thread (if there is more than one thread within the engine). These command lists contain the usual commands you might expect: drawing a model, setting a pipeline active, changing the viewport or render target(s), updating the root signature table with a new entry, and more. One key difference is that until the command list is closed and executed, none of those commands will travel to the graphics card. Upon execution of a command list, it is added to a command queue which *is* thread-safe and where the graphics card pulls its orders from. Every command must be allocated from a command allocator (because we are managing the memory now!). Once a command

list has been executed, the command list can be reset at any time and must be before adding new commands to the list. Caution must be taken to ensure not to reset the allocator until all commands have been completed in the allocator. Notification that the allocator has finished can be achieved with memory fences. There are multiple types of commands lists but the most used are direct commands (for graphics-related commands) and compute commands (for compute shaders). These commands cannot be added to a single combined command list and must have their own command list. If synchronization is needed, then fences must be used.

Now that the absolute minimum about DirectX12 has been mentioned, we can continue with the actual layout of my engine. I will only highlight the major changes between my OpenGL engine that only uses forward rendering and the new deferred/forward-rendering DirectX12 engine. Previous architecture ideas such as my Text2D class, archive system, time classes, and input observers have been ported over and possibly slightly tweaked.

As previously mentioned, pipeline states are very important and are basically an entire pipeline configuration created and compiled all at once. This allows for DirectX12 to be faster as it doesn't need to compile a new pipeline state on the fly. Each of my graphics objects no longer references a shader object, but instead references a pipeline state. This state is set to active upon drawing the graphics object. In the future once I introduce shadows, I may have to adjust this pattern (or store the previous pipeline state in a temporary pointer) as special pipeline states for shadow mapping must be used instead of the graphic object's default pipeline.

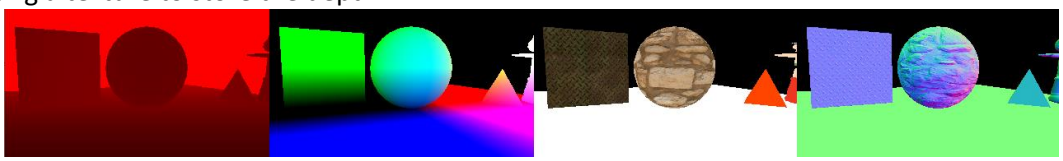
With the necessity of pipeline objects, a Pipeline Manager was created to manage all of these objects. One subtle change with the shader manager is that the Shader Manager now stores shaders at any individual stage. Since any small change to the pipeline required creating a new pipeline, many of my pipelines share similar vertex stages. This is similar to how the Texture Manager works. The Shader Manager stores all shaders to be used by the entire engine and not just a single scene. My pipelines derive from a single base pipeline. Derived from the base class are compute, deferred, forward, and sprite pipelines. Each of these pipelines required setting different configurations (such as multiple render targets for the deferred pipeline) and allowed me to compile any combination of shaders with the derived classes. In order to create a pipeline object, a PipelineStageDef struct is required. This struct contains pointers to any and all programmable stages that a user may want to define. By using a single struct, I reduced the pipeline constructor from requiring many parameters to a single temporary struct. Optional parameters include face windings and whether to render as a solid or wireframe.

The PCS Tree data storage pattern returns for the GameObject Manager. There is one major tweak. Rather than having a single null object at the root, there is now a second level of null objects. At this level is a deferred null object and a forward null object. Based on if the game object should be rendered using forward rendering or deferred, the game object will be attached to the respective parent. Using my PCS iterator, I will either iterator only on the deferred subtree or the forward subtree. This was a great way to manage my game objects all within a single tree.

The reason for the need to separate the two types of rendering is the steps to create a single frame to present to the screen are completely different. Within my engine, I have two types of renders: deferred and forward. The engine supports the ability for both deferred and forward rendered objects to co-exist. Generally, all of the heavy operations occur in the pixel shader stage. With forward rendering, for any pixel that reaches the pixel shader stage, we must perform all of those operations.

After the pixel shader stage is the output merger stage. All of those calculations we did in the pixel shader might be immediately tossed out (if the pixel is behind the previously written pixel based on z-depth) or overwritten in the future (if a future pixel is closer). That means there are a lot of operations being calculated that we may never even get to see the result.

Deferred rendering enters the scene. With deferred rendering, costly operations are not done at first; they are *deferred* until a later pass. What is done in the first pass is all of the information possibly needed is saved to what is called a geometry buffer (or G-Buffer). The G-Buffer is a series of textures that store information for the front-most (or winning) pixel at the end of this pass. While there's no standard for what textures (or how many) are within a G-Buffer object, the most basic versions of a G-Buffer generally include position (such as world position), normal, and specular values. I have also included an NDC depth buffer to be used for a depth-of-field post processing effect. Future iterations, if wanting to reduce memory footprint, could re-use the depth values stored in the depth buffer instead of creating a texture to store the depth.



Using the information stored within the G-Buffer, we can now perform any of our costly operations on the one pixel that won that position. In addition, we are able to now perform post-processing effects as we have in essence taken a screenshot of the current state of that frame. Using the data from the G-Buffer (or from a previous stage in post processing), we can add special effects to the final image such as edge detection, depth-of-field, and bloom.

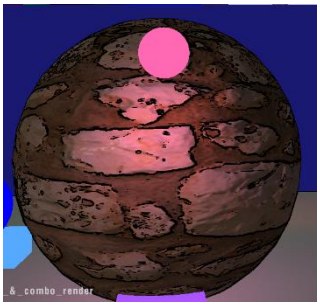
Deferred rendering isn't the end-all solution for rendering. It does have its own drawbacks. The biggest drawback is transparency can no longer be rendered without workarounds. This is because we only store information about the front-most object for that pixel. One workaround is to use compute shaders and keep a history of objects at that pixel. This brings us to the next argument against deferred rendering: memory footprint. Each texture within a G-Buffer is the same size as the window. If we have four textures, which my engine's G-Buffer contains, we are now quadrupling the memory requirement to render a single frame. If any post processing effects are added to the scene, there will also be temporary textures to store those results as well.

When handling lighting with forward rendering, all lights must be stored in a constant buffer (or multiple constant buffers) to perform the lighting calculations. Unless bottomless arrays are used (a new feature in DirectX12), a pixel shader must be created to be able to handle the correct amount of lights (as well as the light types). Even with bottomless arrays, each light must be iterated upon in case the pixel is within range of the light source. With deferred rendering, lighting can still be applied the same way (all in a single draw call), but there are now more options.

Light volumes are an idea that a light source's reach and impact on the scene can be thought of as a bounding volume. For a point light, light emits in all directions based upon a range, so think of it as a sphere whose size changes upon its range. A spot light is a cone with a curved based. A directional light impacts all objects in a scene and so we render a quad that fills the entire viewport. By using these light volumes, we only apply our lighting calculations to the pixels that the light source would actually reach. This reduces our branching cases and automatically creates an early out by aborting in a stage before

the pixel shader. A special pipeline for light volumes is required as we no longer will be rendering our volumes from the outside but from the inside. The idea behind this is that if an object is affected by the light source, it will be *inside* the object. This solves the edge case of if the camera is inside of a light volume, as normally the light source would not be rendered since we would only be seeing the inside of the object. One issue to watch out for is the camera's far plane. If a light volume clips the back plane, some light calculations will not be added to the scene. To fix this, depth clipping within the raster state is an important toggle to make sure to disable.

Within my engine demo, I have included two post-processing effects. These effects are made possible due to deferred rendering and its G-Buffer. After the final lighting values have been calculated and stored in a full-resolution texture, the post processing stages are applied. Each stage has their own temporary texture that it will write to. In future iterations, these temporary textures would be passed along to the following stage. Finally, once all post-processing has finished, a final screen quad is rendered where the final result is then copied to the back buffer of the swap chain and presented.



The first effect is edge-detection. Using the Sobel operator, edges are programmatically detected based upon a matrix kernel and a threshold. The Sobel operator acts upon a grayscale color value and checks the surrounding pixels (horizontally and vertically). If the magnitude of the difference between the luminance is above the threshold, a line is detected and a black pixel is filled in. If not, I have modified the pixel shader to have the pixel be discarded. The reason it is discarded instead of drawing a pixel with zero opacity is that I have modified this stage to not handle any alpha blending. The black pixels for when an edge is detected is drawn to the texture where all of the lighting calculations have been written to.

The second effect I have included in my demo is depth-of-field. The idea behind this effect is that things further away are no longer in focus, just like a human eye or a camera. Previously, without this effect, all objects are rendered in focus no matter how far away from the camera it may be. The magic behind this effect is that it is a two-stage process. The first stage is the final full-resolution texture is down-sampled to a texture that is one-quarter the size of the original window. On the second pass, we now want to draw our final image with the depth-of-field. Using the NDC z-depth texture from the G-Buffer, we convert the value to a linear scale based upon the camera's near and far plane. We also use user-defined blur-start and blur-range values to modify and tweak how quickly any blurring occurs. Based upon those values, we then lerp between the focused image's pixel and the blurred image's pixel based upon the blur delta.

This master's research study scratches just the surface of what DirectX12 and deferred rendering can do. In the future, I would like to figure out more optimized ways to handle constant buffer, unordered access views, and shader resource views as there are some subtle restrictions with each of these data structures I have not mentioned. I would also like to change my depth of field effect's down-sampling to instead use a compute shader as nothing is being rendered to the screen. In addition, shadow mapping will greatly improve all of my lighting and post processing effects to make an even more dramatic scene.