1 The Software Stack

This section assumes a reasonably recent Windows version (Vista or later), which use the WDDM [2] driver model. Older driver models (and other platforms) are somewhat different, but that's outside the scope of this text—I just picked WDDM because it's probably the most relevant model on PCs right now. The basic structure looks like this:

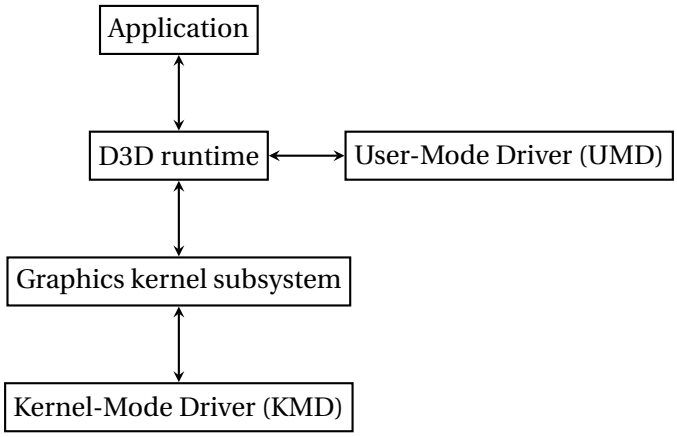

Figure 1.1: The software stack (D3D on WDDM).

1.1 Application and API

It all starts with the application. On PC, all communication between an app and the GPU is mediated by the graphics API; apps may occasionally get direct access to memory that's GPU-addressable (such as Vertex Buffers or Textures), but on PC they can't directly generate native GPU commands[1]—all that has to go through the API and the driver.

The API is the recipient of the app's resource creation, state-setting, and draw calls. The API runtime keeps track of the current state your app has set, validates parameters and does other error and consistency checking, manages user-visible resources, and may or may not validate shader code and shader linkage (it does in D3D, while in OpenGL this is handled at the driver level). It can also merge batches if possible and remove redundant state changes. It then packages it all up nicely and hands it over to the graphics driver—more precisely, the user-mode driver.

[1] Not officially, anyway; since the UMD writes command buffers and runs in user mode, an app could conceivably figure out where the UMD stores its current write pointer and insert GPU commands manually, but that's not exactly supported or recommended behavior.

1.2 The User-Mode Driver (UMD)

This is where most of the "magic" on the CPU side happens. As the name suggests, it is user-mode code; it's running in the same context and address space as the app (and the API runtime) and has no elevated privileges whatsoever. It contains most of the complicated bits of a graphics driver, which also means that if an app crashes the driver, it will probably happen in there. That's a good thing: since it's all user-mode code running in the app's address space, it may take down the running program, but it's unlikely to affect other processes—contrast previous models, where a driver problem could (and frequently would) cause a blue screen and crash the whole system. The UMD is just a normal DLL. It's called "nvd3dum.dll" (Nvidia) or "atiumd*.dll" (AMD). It implements a lower-level API, the DDI [1], that is called by D3D; this API is fairly similar to the one you're seeing on the surface, but a bit more explicit about things like memory and state management.

This module is where things like shader compilation happen. D3D passes a pre-validated shader token stream[2] to the UMD—i.e. it's already checked that the code is valid in the sense of being syntactically correct and obeying D3D constraints (using the right types, not using more textures/samplers than available, not exceeding the number of available constant buffers, stuff like that). This is compiled from HLSL code and usually has quite a number of high-level optimizations (various loop optimizations, dead-code elimination, constant propagation, predicating ifs etc.) applied to it—this is good news since it means the driver benefits from all these relatively costly optimizations that have been performed at compile time. However, it also has a bunch of lower-level optimizations (such as register allocation and loop unrolling) applied that drivers would rather do themselves; long story short, this usually just gets immediately turned into a intermediate representation (IR) and then compiled some more; shader hardware is close enough to D3D bytecode that compilation doesn't need to work wonders to give good results (and the HLSL compiler having done some of the high-yield and high-cost optimizations already definitely helps), but there's still lots of low-level details (such as HW resource limits and scheduling constraints) that D3D neither knows nor cares about, so this is not a trivial process.

For games and other well-known apps, programmers at Nvidia/AMD tend to look at the shader code and write hand-optimized versions for whatever graphics architecture is current at the time, because driver compilers aren't perfect. This well-known app detection and shader substitution happens in the UMD too. Drivers also used to do somewhat sleazier things like using "optimized" shaders in benchmarks that used cheaper texture filtering or using simpler shaders for far-away objects, all to get better benchmark scores. Luckily we seem to have moved past that now (or at least the cheating has gotten less obvious).

To make matters more interesting, some of the API state may actually end up being compiled into the shader. To give an example, relatively exotic (or at least infrequently used) features such as texture borders are probably not implemented in full generality the texture sampler, but emulated with extra code in the shader (or just not supported at all). This means that there's sometimes multiple versions of the same shader floating around, for different

[2] For the curious, the format of D3D9-level Shader bytecode, i.e. versions up to 3.0, is described in the regular WDDM docs that are a part of the WDK; documentation for later bytecode versions is only in the (non-public) WGF specs.

combinations of API states. And while D3D presents the fiction of completely independent Vertex, Pixel, Geometry etc. shaders, the driver may decide to compile e.g. Vertex and Geometry shaders together to minimize the amount of glue code between them.

Incidentally, this is also the reason why you'll often see a delay the first time you use a new shader or resource; a lot of the creation/compilation work is deferred by the driver and only executed when it's actually necessary (you wouldn't believe how much unused crap some apps create!). Graphics programmers know the other side of the story—if you want to make sure something is actually created (as opposed to just having memory reserved), you need to issue a dummy draw call that uses it to "warm it up". That's the price you have to pay for this kind of lazy evaluation.

The UMD also gets to deal with fun things like all the D3D9 "legacy" shader versions and the fixed function pipeline—yes, all of that will get faithfully passed through by D3D. The 3.0 shader profile is fairly reasonable, but 2.0 is crufty and the various 1.x shader versions are seriously weird—remember 1.3 pixel shaders? Or, for that matter, the fixed-function vertex pipeline with vertex lighting and such? Support for all that's still there in D3D and the guts of every modern graphics driver, though of course everything just gets translated into regular shader code at this point.

Then there's things like memory management. The UMD will get things like texture creation commands and need to provide space for them. Now, the UMD does not actually own or directly manage video memory; the OS manages video memory and the KMD (Kernel-Mode Driver) is responsible for whatever magic is necessary to actually allocate memory in the Aperture (a system-memory region accessible to the graphics card) or in Video Memory. But it is the UMD that needs to initiate such allocation requests, and since they're not cheap, it might also decide to bundle small resources into larger memory blocks and suballocate from them.

The UMD can, however, write command buffers.[3] A command buffer contains rendering commands in whatever format the hardware understands. The UMD will convert all the state-changing and rendering commands received from the API into this form, and also insert some new ones that are requisite glue (for example, shaders might need to be uploaded into a special memory area before they can be used for the first time).

There's a problem here, though; as I mentioned, the UMD does not actually "own" the graphics card, nor does it get to manage video memory. So while the UMD is putting together the command buffer, it does not actually know where textures, vertex buffers etc. will be in memory by the time the graphics card gets to see the commands. So it can't put the right offsets in just yet; instead, each command buffer comes with an associated "allocation list" that lists all blocks of memory that need to be in memory for the command buffer to run and a "patch list" that gives the locations in the command buffer where their eventual addresses have to be written to. These patches are applied later by the KMD.

In general, drivers will try to put as much of the actual processing into the UMD as possible; the UMD is user-mode code, so anything that runs in it doesn't need any costly kernel-mode

[3] In the original (blog) version of this text, I used the terms "command buffer" and "DMA buffer" interchangably; this version goes into more detail on WDDM, where the two names refer to separate concepts, so I'll have to be more precise with the terminology; apologies for any confusion this might cause.

transitions, it can freely allocate system memory, farm work out to multiple threads, and so on: it's just a regular DLL (even though it's loaded by the API, not directly by the app). This has advantages for driver development too: if the UMD crashes, the app might crash with it, but not the whole system; it can just be replaced while the system is running (it's just a DLL!); it can be debugged with a regular debugger; and so on. So it's not only efficient, it's also convenient.

However, there's a problem: there's one UMD instance per app that uses the 3D pipeline, but all of them are trying to access a single shared resource, the GPU (even in a dual-GPU configuration, only one of them is actually the main display). In a multi-tasking OS, there's two possible solutions to this problem. Either make sure to only grant exclusive access to the GPU to one app at a time (this was how D3D fullscreen mode originally worked), or introduce some layer that arbitrates access to the GPU and hands all processes their time-slices. And since by now even regular window management uses 3D functionality for rendering, the exclusive model isn't workable anymore. So we need something that acts as the gatekeeper for access to the GPU. In the case of WDDM, that something is the DirectX graphics kernel subsystem (aka `Dxgkrnl.sys`), an OS component which contains—among other things—the GPU scheduler and the video memory manager. In WDDM, the UMD doesn't talk to the kernel layer directly, but has to go through the D3D runtime again; however, the runtime basically just forwards the calls to the right kernel-mode entry points.

1.3 GPU scheduler and Video Memory Manager

The GPU scheduler is exactly that: it arbitrates access to the 3D pipeline using time-slicing, the same way that threads are multiplexed onto a (usually much smaller) number of logical processors. A context switch incurs, at the very least, some state switching on the GPU (which requires extra commands). And of course, different apps likely use very different sets of resources, so things might need to get swapped into (or out of) video memory. At any given point in time, only one process actually has its command buffers forwarded to the KMD; the remaining command buffers just get queued up.

Except it's not actually command buffers (in WDDM lingo) that get sent to the GPU at all; WDDM calls the buffers that actually get sent to the hardware "DMA buffers". The UMD calls the "Render" entry point with a command buffer, allocation list and patch list; all this eventually gets forwarded to the KMD, which has to validate that command buffer and turn it into a (hardware-executable) DMA buffer with a new allocation list and patch list. In theory, because there's this translation step, command buffers could be completely different from hardware DMA buffers; but in practice it's far more efficient (and also simpler) to keep the two very similar, if not identical.

But before the DMA buffer can actually be submitted, the scheduler needs to ensure that all required resources are available. If everything mentioned in the allocation list is already in video memory, the DMA buffer is good to go. If not, however, the allocation list is sent to the video memory manager first.

The video memory manager keeps track of the current contents of video memory. When presented with an allocation list, it has to figure out how to get all resources into GPU-accessible

memory (either video memory or the Aperture). This will involve some uploads from system to video memory, but some resources that aren't used for a given DMA buffer might also get moved from video memory back to system memory. Or resource might be shuffled around in video memory (defragmented) to make sure there's a large enough hole for a large resource. Finally, the video memory manager might also find out that there's just no way to fit all allocations required by a given DMA buffer into memory at once, and request that it be split into several smaller pieces.[4] If all went well, the video memory manager ends up with a list of resource movement commands: copy texture A from system memory to video memory, move vertex buffer B in video memory to compact free space, and so forth. It then calls a special KMD entry point to turn these commands into hardware-consumable DMA buffers that perform memory transfer operations (so-called "paging buffers"). A rendering DMA buffer might take no paging buffers at all (if everything is already in GPU-accessible memory), or it might need several.

Either way, after the video memory manager is done, the final locations for all resources are known, so the scheduler calls the KMD to patch the right offsets into the DMA buffer. And then, finally, the DMA buffers are actually ready to be submitted to the GPU: first the paging buffers, then the rendering command buffer. If the buffer was split, this page-then-render sequence is repeated several times. The KMD will then actually submit the DMA buffer to the GPU.

In the other direction, once the GPU is finished with a DMA buffer, it's supposed to report back to the driver, which in turn pings the graphics kernel subsystem. The video memory manager needs this information to know when it's safe to actually free/recycle memory (dynamic vertex buffers, for example); allocations can't be freed or reused while the GPU might still reference them!

1.4 The Kernel-Mode Driver (KMD)

This is the component that actually deals with the hardware (WDDM-speak for this is "Display miniport driver", by the way, but I prefer KMD). There may be multiple UMD instances running at any one time, but there's only ever one KMD, and if that crashes, the graphics subsystem is in trouble; it used to be "blue screen"-magnitude trouble, but starting with WDDM, Windows actually knows how to kill a crashed driver and reload it—progress! But since the video memory manager tries to keep resources in video memory (which might've been partially overwritten by the GPU reset that typically happens during KMD initialization), apps still need to re-create everything that wasn't cached in system memory. And of course, if the KMD didn't just crash, but went on scribbling all over kernel memory, all bets are off.

From the discussion above, we know a lot of things that the KMD does by now: it handles resource allocation requests from the UMD (but not their physical placement, which is done by the Video Memory Manager). It can render command buffers into DMA buffers, build paging buffers, patch DMA buffers, and submit them to the GPU. It hands a list of different

[4] To facilitate DMA buffer splitting, patch lists contain not just a patch locations, but also split locations so the video memory manager can determine split points without being able to read the (hardware-dependent) DMA buffer format.

memory types (and their physical memory locations) to the video memory manager, and tells it what things can (or should) go where. It also keeps track of different contexts, which contain state information; this is necessary so the correct state can be restored on a context switch.

There's also a lot of things we haven't seen yet. There's functions related to video mode switching, page flipping, the hardware mouse cursor, power-saving functionality, the hardware watchdog timer (resets the GPU if it doesn't respond within a set time interval), deal with hardware interrupts, and so forth. There's some functions that deal with swizzling ranges; we'll see texture swizzling later in section **??**. And of course there's video overlays, which are special because they go through the DRM-infested "protected media path", to ensure that say a Blu-Ray player can push its precious decoded video pixels to the GPU without other user-mode processes getting their hands on the decompressed, decrypted data.

But back to our DMA buffers: what does the KMD do with them once they've been submitted? The answer is one we'll be seeing a lot through this text: put them in another buffer. Or, more precisely, put their addresses into a buffer. Another DMA buffer, to be precise, though this one is usually quite small and a ring buffer. Instead of rendering commands, the ring buffer just contains calls into the actual DMA buffers we just prepared. It's still just a buffer in (GPU-accessible) memory, though, and for anything to happen, the GPU must be told about it. There's usually two (GPU) hardware registers in play here: one is a read pointer, which is the current read position on the GPU side (this is how far it has consumed commands so far), and the other one is the write pointer, which is the location up to which the KMD has written commands to the ring buffer. Whenever the GPU is done with a DMA buffer, it'll return to look at the main ring buffer. If its current read pointer is different from the write pointer, there's more commands to process; if not, there's been no new DMA buffers submitted in the meantime, so the GPU will be idle for a while (until the write pointer changes). Communication with the GPU will be explained in more detail in chapter **??**.

1.5 Aside: OpenGL and other platforms

OpenGL is fairly similar to what I just described, except there's not as sharp a distinction between the API and UMD layer. And unlike D3D, the (GLSL) shader compilation is not handled by the API at all, it's all done by the driver. An unfortunate side effect is that there are as many GLSL frontends as there are 3D hardware vendors, all of them basically implementing the same spec, but with their own bugs and idiosyncrasies—everyone who's ever shipped a product using GL knows how much pain that causes to app authors. And it also means that the drivers have to do all the optimizations themselves whenever they get to see the shaders—including expensive optimizations. The D3D bytecode format is really a cleaner solution for this problem: there's only one compiler (so no slightly incompatible dialects between different vendors!) and it allows for some costlier data-flow analysis than you would normally do.

Open Source implementations of GL tend to use either Mesa or Gallium3D, both of which have a single shared GLSL frontend that generates a device-independent IR and supports multiple pluggable backends for actual hardware. In other words, that space is fairly similar

to the D3D model of sharing the front-end for different implementations and only specializing the low-level codegen that's actually different for every piece of hardware.

1.6 Further Reading

This section is just a coarse overview of the D3D10+/WDDM graphics stack; other implementations work somewhat differently, although the basic entities (driver, API, video memory manager, scheduler, command/DMA buffer dispatch) exist everywhere in some form or another. More details can be found, for example, in the official WDDM documentation [2].

Bibliography

[1] Microsoft. *User-Mode Display Driver Functions*, 2006. URL http://msdn.microsoft.com/en-us/library/windows/hardware/ff570118(v=VS.85).aspx.

[2] Microsoft. *Windows Vista Display Driver Model (WDDM) Reference*, 2006. URL http://msdn.microsoft.com/en-us/library/windows/hardware/ff570595(v=vs.85).aspx.