The ryg blog
When I grow up I'll be an inventor.

A trip through the Graphics Pipeline 2011, part 1

July 1, 2011
*This post is part of the series **"A trip through the Graphics Pipeline 2011" (https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/)**.*

It's been awhile since I posted something here, and I figured I might use this spot to explain some general points about graphics hardware and software as of 2011; you can find functional descriptions of what the graphics stack in your PC does, but usually not the "how" or "why"; I'll try to fill in the blanks without getting too specific about any particular piece of hardware. I'm going to be mostly talking about DX11-class hardware running D3D9/10/11 on Windows, because that happens to be the (PC) stack I'm most familiar with – not that the API details etc. will matter much past this first part; once we're actually on the GPU it's all native commands.

The application

This is your code. These are also your bugs. Really. Yes, the API runtime and the driver have bugs, but this is not one of them. Now go fix it already.

The API runtime

You make your resource creation / state setting / draw calls to the API. 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, may or may not validate shader code and shader linkage (or at least D3D does, in OpenGL this is handled at the driver level) maybe batches work some more, and then hands it all over to the graphics driver – more precisely, the user-mode driver.

The user-mode graphics driver (or UMD)

This is where most of the "magic" on the CPU side happens. If your app crashes because of some API call you did, it will usually be in here :). It's called "nvd3dum.dll" (NVidia) or "atiumd*.dll" (AMD). As the name suggests, this is user-mode code; it's running in the same context and address space as your app (and the API runtime) and has no elevated privileges whatsoever. It implements a lower-level API (the DDI) 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 management and such.

This module is where things like shader compilation happen. D3D passes a pre-validated shader token stream 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.

And of course, if your app is a well-known game, programmers at NV/AMD have probably looked at your shaders and wrote hand-optimized replacements for their hardware – though they better produce the same results lest there be a scandal :). These shaders get detected and substituted by the UMD too. You're welcome.

More fun: 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 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 combinations of API states.

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". Ugly and annoying, but this has been the case since I first started using 3D hardware in 1999 – meaning, it's pretty much a fact of life by this point, so get used to it. :)

Anyway, moving on. The UMD also gets to deal with fun stuff 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 ain't that bad (it's quite reasonable in fact), but 2.0 is crufty and the various 1.x shader versions are seriously whack – remember 1.3 pixel shaders? Or, for that matter, the fixed-function vertex pipeline with vertex lighting and such? Yeah, support for all that's still there in D3D and the guts of every modern graphics driver, though of course they just translate it to newer shader versions by now (and have been doing so for quite some time).

Then there's things like memory management. The UMD will get things like texture creation commands and need to provide space for them. Actually, the UMD just suballocates some larger memory blocks it gets from the KMD (kernel-mode driver); actually mapping and unmapping pages (and managing which part of video memory the UMD can see, and conversely which parts of system memory the GPU may access) is a kernel-mode privilege and can't be done by the UMD.

But the UMD can do things like **swizzling textures (https://fgiesen.wordpress.com/2011/01/17/texture-tiling-and-swizzling/)** (unless the GPU can do this in hardware, usually using 2D blitting units not the real 3D pipeline) and schedule transfers between system memory and (mapped) video memory and the like. Most importantly, it can also write command buffers (or "DMA buffers" – I'll be using these two names interchangeably) once the KMD has allocated them and handed them over. A command buffer contains, well, commands :). All your state-changing and drawing operations will be converted by the UMD into commands that the hardware understands. As will a lot of things you don't trigger manually – such as uploading textures and shaders to video memory.

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 transitions, it can freely allocate 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 your app). This has advantages for driver development too – if the UMD crashes, the app crashes 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.

But there's a big elephant in the room that I haven't mentioned yet.

Did I say "user-mode driver"? I meant "user-mode drivers".

As said, the UMD is just a DLL. Okay, one that happens to have the blessing of D3D and a direct pipe to the KMD, but it's still a regular DLL, and in runs in the address space of its calling process.

But we're using multi-tasking OSes nowadays. In fact, we have been for some time.

This "GPU" thing I keep talking about? That's a shared resource. There's only one that drives your main display (even if you use SLI/Crossfire). Yet we have multiple apps that try to access it (and pretend they're the only ones doing it). This doesn't just work automatically; back in The Olden Days, the solution was to only give 3D to one app at a time, and while that app was active, all others wouldn't have access. But that doesn't really cut it if you're trying to have your windowing system use the GPU for rendering. Which is why you need some component that arbitrates access to the GPU and allocates time-slices and such.

Enter the scheduler.

This is a system component – note the "the" is somewhat misleading; I'm talking about the graphics scheduler here, not the CPU or IO schedulers. This does exactly what you think it does – it arbitrates access to the 3D pipeline by time-slicing it between different apps that want to use it. A context switch incurs, at the very least, some state switching on the GPU (which generates extra commands for the command buffer) and possibly also swapping some resources in and out of video memory. And of course only one process gets to actually submit commands to the 3D pipe at any given time.

You'll often find console programmers complaining about the fairly high-level, hands-off nature of PC 3D APIs, and the performance cost this incurs. But the thing is that 3D APIs/drivers on PC really have a more complex problem to solve than console games – they really do need to keep track of the full current state for example, since someone may pull the metaphorical rug from under them at any moment! They also work around broken apps and try to fix performance problems behind their backs; this is a rather annoying practice that no-one's happy with, certainly including the driver authors themselves, but the fact is that the business perspective wins here; people expect stuff that runs to continue running (and doing so smoothly). You just won't win any friends by yelling "BUT IT'S WRONG!" at the app and then sulking and going through an ultra-slow path.

Anyway, on with the pipeline. Next stop: Kernel mode!

The kernel-mode driver (KMD)

This is the part that actually deals with the hardware. There may be multiple UMD instances running at any one time, but there's only ever one KMD, and if that crashes, then boom you're dead – used to be "blue screen" dead, but by now Windows actually knows how to kill a crashed driver and reload it (progress!). As long as it happens to be just a crash and not some kernel memory corruption at least – if that happens, all bets are off.

The KMD deals with all the things that are just there once. There's only one GPU memory, even though there's multiple apps fighting over it. Someone needs to call the shots and actually allocate (and map) physical memory. Similarly, someone must initialize the GPU at startup, set display modes (and get mode information from displays), manage the hardware mouse cursor (yes, there's HW handling for this, and yes, you really only get one! :), program the HW watchdog timer so the GPU gets reset if it stays unresponsive for a certain time, respond to interrupts, and so on. This is what the KMD does.

There's also this whole content protection/DRM bit about setting up a protected/DRM'ed path between a video player and the GPU so no the actual precious decoded video pixels aren't visible to any dirty user-mode code that might do awful forbidden things like dump them to disk (…whatever). The KMD has some involvement in that too.

Most importantly for us, the KMD manages the *actual* command buffer. You know, the one that the hardware actually consumes. The command buffers that the UMD produces aren't the real deal – as a matter of fact, they're just random slices of GPU-addressable memory. What actually happens with them is that the UMD finishes them, submits them to the scheduler, which then waits until that process is up and then passes the UMD command buffer on to the KMD. The KMD then writes a call to command buffer into the main command buffer, and depending on whether the GPU command processor can read from main memory or not, it may also need to DMA it to video memory first. The main command buffer is usually a (quite small) **ring buffer (https://fgiesen.wordpress.com/2010/12/14/ring-buffers-and-queues/)** – the only thing that ever gets written there is system/initialization commands and calls to the "real", meaty 3D command buffers.

But this is still just a buffer in memory right now. Its position is known to the graphics card – there's usually a read pointer, which is where the GPU is in the main command buffer, and a write pointer, which is how far the KMD has written the buffer yet (or more precisely, how far it has *told* the GPU it has written yet). These are hardware registers, and they are memory-mapped – the KMD updates them periodically (usually whenever it submits a new chunk of work)…

The bus

…but of course that write doesn't go directly to the graphics card (at least unless it's integrated on the CPU die!), since it needs to go through the bus first – usually PCI Express these days. DMA transfers etc. take the same route. This doesn't take very long, but it's yet another stage in our journey. Until finally…

The command processor!

This is the frontend of the GPU – the part that actually reads the commands the KMD writes. I'll continue from here in the next installment, since this post is long enough already :)

Small aside: OpenGL

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. Not fun. 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.

Omissions and simplifcations

This is just an overview; there's tons of subtleties that I glossed over. For example, there's not just one scheduler, there's multiple implementations (the driver can choose); there's the whole issue of how synchronization between CPU and GPU is handled that I didn't explain at all so far. And so on. And I might have forgotten something important – if so, please tell me and I'll fix it! But now, bye and hopefully see you next time.

From → Coding, Graphics Pipeline

39 Comments

1. **Bitouo permalink**
   Thank you for this great article! Will you write a little bit about how synchronization between CPU and GPU is handled. I have been curious about it for a long while. Or maybe point out some nice articles I can read. :)

   Reply
   - **fgiesen permalink**
     Yes, I'll be filling in the blanks about the details of memory/resource lifetime management and CPU/GPU synchronization as I go along.This will be one of our recurring themes in fact, since it affects virtually all parts of the pipeline in some way. I'll explain it as soon as I get there :)

     Reply
2. **Compulsive Dabbler permalink**
   Thanks a ton for this, I haven't found a single other resource that explains these details so concisely!

   Reply
3. **3dfx permalink**
   Impressive article. Can't wait for the next installment!

   Reply
4. **Christophe Riccio permalink**
   "And it also means that the drivers have to do all the optimizations themselves whenever they get to see the shaders – including expensive optimizations."

   This is actually a drawback of the D3D approach because each GPU architechture is really different, the first tack of the compiler is to "un-optimized" the bytecode before running the GPU optimizations.

   Reply
   - **fgiesen permalink**
     I'm actually working on a shader compiler for actual hardware right now (albeit a R&D one, not one intended for production) and I can tell you that – at least as far as my experience goes – this is an urban myth, or at least blown way out of proportion.

Yes, D3D bytecode is an abstraction, and I'd rather not have the HLSL compiler do "information-destroying" transforms like loop unrolling, function inlining or deciding whether to use ifs or predication. Detecting and undoing this does mean extra work for the compiler and is annoying and a waste of time, but in the grand scheme of things it's not that big a deal.

But the bulk of it is simply that the HLSL compiler does some optimizations which are "information-neutral" (such as register packing or scheduling) that are just a waste of time on the HLSL compiler side; if I compile a shader, the first thing I do is convert it to some optimizer-friendly IR (nowadays, that usually means Static Single Assignment form or its descendants), and that simply implicitly destroys these optimizations. There's no extra work involved in "un-optimizing" them – it just happens as a side effect of IR construction. So it's just a waste of time for HLSL to be doing this, but not actually a problem.

And it's definitely nice to have a shared frontend that does all the cool "code cleanup" transformations for me; the HLSL compiler does constant propagation, global value numbering, partial redundancy elimination and loop-invariant code motion, and has fairly sophisticated mechanisms for algebraic simplifications.

All of these things are costly both in terms of implementation complexity and run time, and they're more or less completely device independent. Any regular compiler does them before the IR reaches the device-dependent backend code generator (which is what the driver is); it makes perfect sense for D3D to be doing it too. It pulls expensive work out of the runtime compilation loop, and it's a lot of tricky work that drivers now don't need to worry about. In the OpenGL/GLSL world, they do, and if you've used GLSL particularly in its first few years, you got to see the (awful) results. I think farming this off into a separate stage was a very good call on MS's part.

Reply

5. **KeyJ permalink**
   Great article. I have one question though: Where is the scheduler located? Is it part of the UMD, of D3D, the KMD or is is some completely different library or process?

   Reply
   - **fgiesen permalink**
     The scheduler is part of the OS/driver model.

     The total sequence of stages up to this point is App (you) -> API (OS/driver) -> UMD (driver) -> Scheduler (OS) -> KMD (driver) -> GPU.

     This also means that the notion of command/DMA buffer is part of the driver model, not just some implementation detail, since the scheduler needs to know about it (after all, that's the things it's scheduling!). It doesn't touch the data (or know what it means), but it does need to know that such a thing exists to pass it around.

     Reply
     - **Marek permalink**
       Is this sequence of stages applies to Linux also? I don't understand UMD idea in Linux.

     - **fgiesen permalink**
       The structure is similar but not quite the same. The equivalent to the KMD is the kernel DRM driver, UMD is the GL driver, state tracker and everything up to and including libdrm/libdri. The details are different though. In particular, I'm not aware of a kernel-land central video memory manager, graphics scheduler or video memory paging / command buffer patching mechanism.

     - **x4da permalink**
       In radeon queue scheduling and submission are done on kernel driver side. Same with memory management: userspace request kernel driver to set up VM address space and requests a Buffer Object to be created and mapped to VM AS, which kernel driver (using TTM mechanism) does and adds to its private BO pool.

     - **fgiesen permalink**
       That's just the point though, this is part of the per-device drivers, not a central graphics subsystem service like on Windows.

     - **x4da permalink**
       Yep, on linux thing could be different depending on vendor: proprietary nvidia drivers don't use kernel TTM buffer management and DRI/DRM interface.

6. **Corbin Simpson permalink**
   A good overview.

Two notes: First, GL drivers generally are built on very reusable code these days, which means that the general pattern of GLSL->IR->driver-specific bytecode is actually very viable and happens in production drivers. Gallium takes this to an extreme: GLSL is compiled to a GLSL IR, which is optimized at the high level and passed to Gallium as TGSI, which is GPU-neutral and serializeable. Then the specific driver backend turns TGSI into actual GPU bytecode. The entire process works pretty well, actually!

Second, I figured I'd explain what's different on Linux, as far as the kernel goes. Not much changes; the big things are in what's shared and how robust the drivers are. GPU scheduling's per-driver and usually very basic; the predominant system involves lockless blocking ioctl()s and a master process (X server, Wayland, etc.) which controls who is allowed to submit commands. On-GPU memory management is shared through a couple of kernel libraries, GEM and TTM, which allow everybody to enjoy common benefits in improvements to memory management. Finally, drivers can't be auto-evicted if they crash, but they do generally understand (at least for modern GPUs) how to reboot the GPU if it wedges or hangs.

Reply

- **fgiesen permalink**
  "the general pattern of GLSL->IR->driver-specific bytecode is actually very viable and happens in production drivers"
  Yes, certainly; this general architecture is really the only sane way to build optimizing compilers when your backend architecture goes through significant changes every 1-2 years :). But D3D standardizing on a frontend/IR (even if it's not ideal) helped them get significantly less problems in "portability" of shader code across different vendors at a critical point in time.

  Interesting to see that Linux is doing something very similar these days. Good stuff!

  Reply

  - **przemo_li permalink**
    Gallium 3D is idea of common front put even further. You can get others state trackers (eg. DX10, OpenVG, OpenGL ES, etc.) too. And it is evolving even more (eg. some devs want to use LLVM in the middle between GLSL and specific bytecode). Current DX model can not beat it. Since all common optimizations for hwd, must be implemented separately anyway. Gallium 3D is more of template you driver can fill with really unique stuff, while HLSL compiler is just top common stuff with some cleaning. (However, still its better than any non-gallium3d driver, but do not know how MacOSX handle GLSL, maybe there Apple will copy HLSL approach?)

  - **fgiesen permalink**
    As I've said in the article, the main reason I'm using the Win32/D3D graphics stack for this series (which is about GPUs not 3D APIs) is that it's the one I'm most familiar with, and I need to decide on *some* set of terminology to use. Beyond that, I'm not interested in talking about API differences. If you are, good for you, but please take it somewhere else.

7. **Nico permalink**
   And don't forget about the additional cost of having the cpu flushing it's cache and draining the write buffers to memory first and the PCIe-Hostcontroller which has to read it back from there. This also increases latency.

   Reply

   - **fgiesen permalink**
     Command buffers are customarily in write-combined memory which is uncached (but store buffered). So no cache-draining involved; on x86, it's just a `sfence` (which flushes the store buffers). This isn't free, but considering you went through the OS scheduler and a user->kernel mode transition just to get the command buffers to the KMD in the first place, this is small fry. And actually, all of this is pipelined to the point where graphics drivers are *up to 3 frames ahead* of the GPU (drivers used to try for more, but that was totally screwing with input latency and such, so it's now capped at 3). So at that point (if the CPU/driver is fast enough to get that far ahead!), assuming 60Hz full-framerate rendering, the latency between an app issuing a draw call and the GPU actually processing is about 50ms, and can be south of that if we're not rendering at full framerate. Point being, there's so much intentional latency introduced by the SW stack just to keep the GPU fed at all times, that additional (comparatively small) latencies added by such low-level effects are all but unnoticeable.

     Reply

8. **Raja permalink**
   This is absolutely fantastic. Just what I needed! I've always struggled to see the big picture of interactions, and your blog is just what the doctor ordered.
   Thanks a lot for taking the time to do this. I can't wait to read all your posts now!

   Reply

9. **dca permalink**
   A quick add: modern GPUs have somewhat called GPUVM – separate process address spaces with their own sets of page tables each.

Reply

10. **HY permalink**
    What is the mechanism in which the UMD communicates with the KMD? Does the KMD create special syscalls which the UMD invokes? Or exposes a \GLOBAL?? device node to be accessed by the UMD?

    Reply

    - **fgiesen permalink**
      The D3D runtime provides an API that UMDs are using to talk to the KMD. Just refer to the official docs if you're interested in the details. I'm not sure how the implementation looks under the hood – graphics driver authors (both KMD and UMD) don't need to worry, this is mediated by the D3D runtime and the DirectX Graphics Kernel Subsystem (dxgkrnl).

      Reply

11. **James Bedford permalink**
    Great article – can't wait to keep reading more!

    Reply

12. **Yogesh permalink**
    Thanks for the great pipeline series. You should add one article on locks.(locking cpu or gpu memory).

    Reply

13. **Steex permalink**
    Thanks for the excellent series! I'm not a professional graphics programmer, still as a game developer I have to know at least the facade of the pipeline. But only the fasade. So naturally while reading your series I was surprised… well, many times. Amazing algorithms! Ang pretty good explanation, I have to add.
    It's interesting though, what is changed to the modern time (end of 2015). Are the articles still relevant? For example, I read that AMD Mantle and DirectX 12 provide direct access to command buffers. Did this changed something in GPU architecture? Sorry for somewhat newbie questions. :)

    Reply

    - **fgiesen permalink**
      Neither Mantle, D3D12, nor Vulkan provide direct access to command buffers. They remove some of the intermediate layers in the software stack (by pushing the work to the application side), but that all happens before the stuff I talk about in this series.

      There have been no fundamental changes to desktop GPU architecture since I wrote this series, not at the (still relatively abstract) level of this series anyway. A lot of the details have changed – for example, asynchronous compute support means that the GPU can process multiple command streams at the same time (by having multiple command processors, time-slicing a single command processor in either SW or HW, or some combination thereof). That means that from a user point of view, there's now multiple command processors; but that doesn't change the way they work, it just means there's more of them (complicating internal synchronization, but again, that's below the level of abstraction of this series). Another big-ticket item would be what D3D12 calls "Rasterizer Ordered Views"; but I already talked about several ways to handle blend ordering. Basically, ROV support means the GPU can optionally track in-flight quads that are trying to write to the same location, and make sure they run in order.

      But for the most part, D3D12/Mantle/Vulkan are not about any of these things. The biggest change in these APIs is that they replace the "state machine" model of GL and older D3D versions with a model where all kinds of state is pushed by the application beforehand and compiled into a format the hardware can understand directly, and resource residency/memory/dependency management is pushed (to a significant extent) to the app. This saves a lot of work on the driver side, which otherwise has to do the state translation every time somebody changes any piece of state, and has to track which resources are in use by whom at what time to make sure the correct synchronization happens. That's where the speed-ups come from. The underlying HW didn't change at all.

      Reply

14. **cyang permalink**
    Hello fgiesen, thank you very much for this great article! I am a Ph.D student in EE and have been digging into Graphic cards as a side hobby project. Recently, I am thinking about making an open source PC graphic card project. Though this is a tough and unrealistic side project, it is still a good goal to move towards and learn stuff during the process. With that being said, I would appreciate it if you could allow me to translate this article into another language (in my case it is Chinese) and share these interesting series of article with some of my friends who are working with me on the side hobby project. Thank you very much and I look forward to hearing your permission. Have a nice day:)

    Reply

    - **fgiesen permalink**

The series as a whole is CC-0 licensed, you are allowed to do whatever you want with it!

Reply

- **cyang permalink**
  Thank you very much for your kind reply. Sorry I didn't check the series cover page before:)

15. **Connor A. Haskins permalink**
    When you say the API runtime "manages user-visible resources", the user in this context is the application, right? What are examples of these resources?

    Also, thank you so much for writing this.

    Reply

    - **fgiesen permalink**
      Textures, buffers (and constant buffers), render targets are the major ones. "Resource" is D3D-speak (note I'm using D3D terminology the whole way through the series) for all of those.

      Reply

Trackbacks & Pingbacks

1. Geeks3D Programming Links – July 01, 2011 - 3D Tech News, Pixel Hacking, Data Visualization and 3D Programming - Geeks3D.com
2. 3D Graphics Pipeline Explained - 3D Tech News, Pixel Hacking, Data Visualization and 3D Programming - Geeks3D.com
3. A trip through the Graphics Pipeline 2011: Index « The ryg blog
4. Understanding Modern GPUs (I): Introduction « TraxNet – Blog
5. Viaje alucinante por un pipeline grafico « martin b.r.
6. BreakTryCatch » Getting Started With DirectX 11
7. How do graphics processing units (GPUs) work? - Quora