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

A trip through the Graphics Pipeline 2011, part 7

July 8, 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/)**.*

In this installment, I'll be talking about the (early) Z pipeline and how it interacts with rasterization. Like the last part, the text won't proceed in actual pipeline order; again, I'll describe the underlying algorithms first, and then fill in the pipeline stages (in reverse order, because that's the easiest way to explain it) after the fact.

Interpolated values

Z is interpolated across the triangle, as are all the attributes output by the vertex shader. So let me take a minute to explain how that works. At this point I originally had a section on how the math behind interpolation is derived, and why perspective interpolation works the way it works. I struggled with that for hours, because I was trying to limit it to maybe one or two paragraphs (since it's an aside), and what I can say now is that if I want to explain it properly, I need more space than that, and at least one or two pictures; a picture may say more than thousand words, but a nice diagram takes me about as long to prepare as a thousand words of text, so that's not necessarily a win from my perspective :). Anyway, this is something of a tangent anyway, so I'm adding it to my pile of "graphics-related things to write up properly at some point". For now, I'm giving you the executive summary:

Just linearly interpolating attributes (colors, texture coordinates etc.) across the screen-space triangle does not produce the right results (unless the interpolation mode is one of the "no perspective" ones, in which case ignore what I just wrote). However, say we want to interpolate a 2D texture coordinate pair (s, t). It turns out you do get the right results if you linearly interpolate $\frac{1}{w}$, $\frac{s}{w}$ and $\frac{t}{w}$ in screen-space (w here is the homogeneous clip-space w from the vertex position), then per-pixel take the reciprocal of $\frac{1}{w}$ to get w, and finally multiply the other two interpolated fractions by w to get s and t. The actual linear interpolation boils down to setting up a plane equation and then plugging the screen-space coordinates in. And if you're writing a software perspective texture mapper, that's the end of it. But if you're interpolating more than two values, a better approach is to compute (using perspective interpolation) **barycentric coordinates (http://en.wikipedia.org/wiki/Barycentric_coordinate_system_(mathematics))** – let's call them λ_0 and λ_1 – for the current pixel in the original clip-space triangle, after which you can interpolate the actual vertex attributes using regular linear interpolation without having to multiply everything by w afterwards.

So how much work does that add to triangle setup? Setting up the $\frac{\lambda_0}{w}$ and $\frac{\lambda_1}{w}$ for the triangle requires 4 reciprocals, the triangle area (which we already computed for back-face culling!), and a few subtractions, multiplies and adds. Setting up the vertex attributes for interpolation is really cheap with the barycentric approach – two subtractions per attribute (if you don't use barycentric, you get some more multiply-add action here). Follow me? Probably not, unless you've implemented this before. Sorry about that – but it's fairly safe to ignore all this if you don't understand it.

Let's get back to why we're here: the one value we want to interpolate *right now* is Z, and because we computed Z as $\frac{z}{w}$ at the vertex level as part of projection (see previous part), so it's already divided by w and we can just interpolate it linearly in screen space. Nice. What we end up with is a plane equation for $Z = aX + bY + c$ that we can just plug X and Y into to get a value. So, here's the punchline of my furious hand-waving in the last few paragraphs: Interpolating Z at any given point boils down to two multiply-adds. (Starting to see why GPUs have fast multiply-accumulate units? This stuff is absolutely everywhere!).

Early Z/Stencil

Now, if you believe the place that graphics APIs traditionally put Z/Stencil processing into – right before alpha blend, way at the bottom of the pixel pipeline – you might be confused a bit. Why am I even discussing Z at the point in the pipeline where we are right now? We haven't even started shading pixels! The answer is simple: the Z and stencil tests reject pixels. Potentially the majority of them. You really, *really* don't want to completely shade a detailed mesh with complicated materials, to then throw away

95% of the work you just did because that mesh happens to be mostly hidden behind a wall. That's just a really stupid waste of bandwidth, processing power and energy. And in most cases, it's completely unnecessary: most shaders don't do anything that would influence the results of the Z test, or the values written back to the Z/stencil buffers.

So what GPUs actually do when they can is called "early Z" (as opposed to late Z, which is actually at the late stage in the pipeline that traditional API models generally display it at). This does exactly what it sounds like – execute the Z/stencil tests and writes early, right after the triangle has been rasterized, and before we start sending off pixels to the shaders. That way, we notice all the rejected pixels early, without wasting a lot of computation on them. However, we can't always do this: the pixel shader may ignore the interpolated depth value, and instead provide its own depth to be written to the Z-buffer (e.g. depth sprites); or it might use discard, alpha test, or alpha-to-coverage, all of which "kill" pixels/samples during pixel shader execution and mean that we can't update the Z-buffer or stencil buffer early because we might be updating depth values for samples that later get discarded in the shader!

So GPUs actually have two copies of the Z/stencil logic; one right after the rasterizer and in front of the pixel shader (which does early Z) and one after the shader (which does late Z). Note that we can still, in principle, do the depth testing in the early-Z stage even if the shader uses some of the sample-killing mechanism. It's only writes that we have to be careful with. The only case that really precludes us from doing any early Z-testing at all is when we write the output depth in the pixel shader – in that case the early Z unit simply has nothing to work with.

Traditionally, APIs just pretended none of this early-out logic existed; Z/Stencil was in a late stage in the original API model, and any optimizations such as early-Z had to be done in a way that was 100% functionally consistent with that model; i.e. drivers had to detect when early-Z was applicable, and could only turn it on when there were no observable differences. By now APIs have closed that gap; as of DX11, shaders can be declared as "force early-Z", which means they run with full early-Z processing even when the shader uses primitives that aren't necessarily "safe" for early-Z, and shaders that write depth can declare that the interpolated Z value is conservative (i.e. early Z reject can still happen).

Z/stencil writes: the full truth

Okay, wait. As I've described it, we now have two parts in the pipeline – early Z and late Z – that can both write to the Z/stencil buffers. For any given shader/render state combination that we look at, this will work – in the steady state. But that's not how it works in practice. What actually happens is that we render a few hundred to a few thousand batches per frame, switching shaders and render state regularly. Most of these shaders will allow early Z, but some won't. Switching from a shader that does early Z to one that does late Z is no problem. But going back from late Z to early Z is, if early Z does any writes: early Z is, well, earlier in the pipeline than late Z – that's the whole point! So we may start early-Z processing for one shader, merrily writing to the depth buffer while there's still stuff down in the pipeline for our old shader that's running late-Z and may be trying to write the same location at the same time – classic race condition. So how do we fix this? There's a bunch of options:

Once you go from early-Z to late-Z processing within a frame (or at least a sequence of operations for the same render target), you stay at late-Z until the next point where you flush the pipeline anyway. This works but potentially wastes lots of shader cycles while early-Z is unnecessarily off.
Trigger a (pixel) pipeline flush when going from a late-Z shader to an early-Z shader – also works, also not exactly subtle. This time, we don't waste shader cycles (or memory bandwidth) but stall instead – not much of an improvement.
But in practice, having Z-writes in two places is just bad news. Another option is to not ever write Z in the early-Z phase; always do the Z-writes in late-Z. Note that you need to be careful to make conservative Z-testing decisions during early Z if you do this! This avoids the race condition but means the early Z-test results may be stale because the Z-write for the currently-dispatched pixels won't happen until a while later.
Use a separate unit that keeps track of Z-writes for us and enforces the correct ordering; both early-Z and late-Z must go through this unit.

All of these methods work, and all have their own advantages and drawbacks. Again I'm not sure what current hardware does in these cases, but I have strong reason to believe that it's one of the last two options. In particular, we'll meet a functional unit later down the road (and the pipeline) that would be a good place to implement the last option.

But we're still doing all this testing per pixel. Can't we do better?

Hierarchical Z/Stencil

The idea here is that we can use our tile trick from rasterization again, and try to Z-reject whole tiles at a time, before we even descend down to the pixel level! What we do here is a strictly conservative test; it may tell us that "there might be pixels that pass the Z/stencil-test in this tile" when there are none, but it will never claim that all pixels are rejected when in fact they weren't.

Assume here that we're using "less", "less-equal", or "equal" as Z-compare mode. Then we need to store the *maximum* Z-value we've written for that tile, per tile. When rasterizing a triangle, we calculate the *minimum* Z-value the active triangle is going to write to the current tile (one easy conservative approximation is to take the min of the interpolated Z-values at the four corners of the current tile). If our triangle minimum-Z is larger than the stored maximum-Z for the current tile, the triangle is guaranteed to be completely occluded. That means we now need to track maximum-Z per-tile, and keep that value up to date as we write new pixels – though again, it's fine if that information isn't completely up to date; since our Z-test is of the "less" variety, values in the Z buffer will only get smaller over time. If we use a per-tile maximum-Z that's a bit out of date, it just means we'll get slightly worse early rejection rates than we could; it doesn't cause any other problems.

The same thing works (with min/max and compare directions swapped) if we're using one of the "greater", "greater-equal" or "equal" Z-tests. What we can't easily do is change from one of the "less"-based tests to a "greater"-based tests in the middle of the frame, because that would make the information we've been tracking useless (for less-based tests we need maximum-Z per tile, for greater-based tests we need minimum-Z per tile). We'd need to loop over the whole depth buffer to recompute min/max for all tiles, but what GPUs actually do is turn hierarchical-Z off once you do this (up until the next Clear). So: don't do that.

Similar to the hierarchical-Z logic I've described, current GPUs also have hierarchical stencil processing. However, unlike hierarchical-Z, I haven't seen much in the way of published literature on the subject (meaning, I haven't run into it – there might be papers on it, but I'm not aware of them); as a game console developer you get access to low-level GPU docs which include a description of the underlying algorithms, but frankly, I'm definitely not comfortable writing about something here where really the only good sources I have are various GPU docs that came with a thick stack of NDAs. Instead I'll just nebulously note that there's magic pixie dust that can do certain kinds of stencil testing very efficiently under controlled circumstances, and leave you to ponder what that might be and how it might work, in the unlikely case that you deeply care about this – presumably because your father was killed by a hierarchical stencil unit and you're now collecting information on its weak points for your revenge, or something like that.

Putting it all together

Okay, we now have all the algorithms and theory we need – let's see how we can take our new set of toys and wire it up with what we already have!

First off, we now need to do some extra triangle setup for Z/attribute interpolation. Not much to be done about it – more work for triangle setup; that's how it goes. After that's coarse rasterization, which I've discussed in the previous part.

Then there's hierarchical Z (I'm assuming less-style comparisons here). We want to run this between coarse and fine rasterization. First, we need the logic to compute the minimum Z estimates for each tile. We also need to store the per-tile maximum Zs, which don't need to be exact: we can shave bits as long as we always round up! As usual, there's a trade-off here between space used and early-rejection efficiency. In theory, you could put the Z-max info into regular memory. In practice, I don't think anyone does this, because you want to make the hierarchical-Z decision without a ton of extra latency. The other option is to put dedicated memory for hierarchical Z onto the chip – usually as SRAM, the kind of memory you also make caches out of. For 24-bit Z, you probably need something like 10-14 bits per tile to store a reasonable-accuracy Z-max in a compact encoding. Assuming 8×8 tiles, that means less than 1MBit (128k) of SRAM to support resolutions up to 2048×2048 – sounds like a plausible order of magnitude to me. Note that these things are fixed size and shared for the whole chip; if you do a context switch, you lose. If you allocate the wrong depth buffers to this memory, you can't use hierarchical Z on the depth buffers that actually matter, and you lose. That's just how it goes. This kind of things is why hardware vendors regularly tell you to create your most important render targets and depth buffers first; they have a limited supply of this type of memory (there's more like it, as you'll see), and when it runs out, you're out of luck. Note they don't necessarily need to do this all-or-nothing; for example, if you have a really large depth buffer, you might only get hierarchical Z in the top left 2048×1536 pixels, because that's how much fits into the Z-max memory. It's not ideal, but still much better than disabling hierarchical-Z outright.

And by the way, "Real-Time Rendering" mentions at this point that "it is likely that GPUs are using hierarchical Z-buffers with more than two levels". I doubt this is true, for the same reason that I doubt they use a multilevel hierarchical rasterizer: adding more levels makes the easy cases (large triangles) even faster while adding latency and useless work for small triangles: if you're drawing a triangle that fits inside a single 8×8 tile, any coarser hierarchy level is pure overhead, because even at the 8×8 level, you'd just do one test to trivial-reject the triangle (or not). And again, for hardware, it's not that big a performance issue; as long as you're not consuming extra bandwidth or other scarce resources, doing more compute work than strictly necessary isn't a big problem, as long as it's within reasonable limits,

Hierarchical stencil is also there and should also happen prior to fine rast, most likely in parallel with hierarchical Z. We've established that this runs on air, love and magic pixie dust, so it doesn't need any actual hardware and is probably always exactly right in its predictions. Ahem. Moving on.

After that is fine rasterization, followed in turn by early Z. And for early Z, there's two more important points I need to make.

Revenge of the API order

For the past few parts, I've been playing fast and loose with the order that primitives are submitted in. So far, it didn't matter; not for vertex shading, nor primitive assembly, triangle setup or rasterization. But Z is different. For Z-compare modes like "less" or "lessequal", it's very important what order the pixels arrive in; if we mess with that, we risk changing the results and introducing nondeterministic behavior. More importantly, as per the spec, we're free to execute operations in any order *so long as it isn't visible to the app*; well, as I just said, for Z processing, order is important, so we need to make sure that triangles arrive at Z processing in the right order (this goes for both early and late Z).

What we do in cases like this is go back in the pipeline and look for a reasonable spot to sort things into order again. In our current path, the best candidate location seems to be primitive assembly; so when we start assembling primitives from shaded vertex blocks, we make sure to assemble them strictly in the original order as submitted by the app to the API. This means we might stall a bit more (if the PA buffer holds an output vertex block, but it's not the correct one, we need to wait and can't start setting up primitives yet), but that's the price of correctness.

Memory bandwidth and Z compression

The second big point is that Z/Stencil is a serious bandwidth hog. This has a couple of reasons. For one, this is the one thing we really run for all samples generated by the rasterizer (assuming Z/Stencil isn't off, of course). Shaders, blending etc. all benefit from the early rejection we do; but even Z-rejected pixels do a Z-buffer read first (unless they were killed by hierarchical Z). That's just how it works. The other big reason is that, when multisampling is enabled, the Z/stencil buffer is per sample; so 4x MSAA means 4x the memory bandwidth cost of Z? For something that takes a substantial amount of memory bandwidth even at no MSAA, that's seriously bad news.

So what GPUs do is Z compression. There's various approaches, but the general idea is always the same: assuming reasonably-sized triangles, we expect a lot of tiles to just contain one or maybe two triangles. If that happens, then instead of storing Z-values for the whole tile, we just store the plane equation of the triangle that filled up this tile. That plane equation is (hopefully) smaller than the actual Z data. Without MSAA, one tile covers 8×8 actual pixels, so triangles need to be relatively big to cover a full tile; but with 4x MSAA, a tile effectively shrinks to 4×4 pixels, and covering full tiles gets easier. There's also extensions that can support 2 triangles etc., but for reasonably-sized tiles, you can't go much larger than 2-3 tris and still actually save bandwidth: the extra plane equations and coverage masks aren't free!

Anyway, point is: this compression, when it works, is fully lossless, but it's not applicable to all tiles. So we need some extra space to denote whether a tile is compressed or not. We could store this in regular memory, but that would mean we now need to wait two full memory round-trips latencies to do a Z-read. That's bad. So again, we add some dedicated SRAM that allows us to store a few (1-3) bits per tile. At its simplest, it's just a "compressed" or "not compressed" flag, but you can get fancy and add multiple compression modes and such. A nice side effect of Z-compression is that it allows us to do fast Z-clears: e.g. when clearing to Z=1, we just set all tiles to "compressed" and store the plane equation for a constant Z=1 triangle.

All of the Z-compression thing, much like texture compression in the texture samplers, can be folded into memory access/caching logic, and made completely transparent to everyone else. If you don't want to send the plane equations (or add the interpolator logic) to the Z memory access block, it can just infer them from the Z data and use some integer delta-coding scheme. This kind of

approach usually needs extra bits per sample to actually allow lossless reconstruction, but it can lead to simpler data paths and nicer interface between units, which hardware guys love.

And that's it for today! Next up: Pixel shading and what happens around it.

Postscript

As I said earlier, the topic of setting up interpolated attributes would actually make for a nice article on its own. I'm skipping that for now – might decide to fill this gap later, who knows.

Z processing has been in the 3D pipeline for ages, and a serious bandwidth issue for most of the time; people have thought long and hard about this problem, and there's a zillion tricks that go into doing "production-quality" Z-buffering for GPUs, some big, some small. Again, I'm just scratching the surface here; I tried to limit myself to the bits that are useful to know for a graphics programmer. That's why I don't spend much time on the details of hierarchical Z computations or Z compression and the like; all of this is very specific on hardware details that change slightly in every generation, and ultimately, mostly there's just no practical way you get to exploit any of this usefully: If a given Z-compression scheme works well for your scene, that's some memory bandwidth you can spend on other things. If not, what are you gonna do? Change your geometry and camera position so that Z-compression is more efficient? Not very likely. To a hardware designer, these are all algorithms to be improved on in every generation, but to a programmer, they're just facts of life to deal with.

This time, I'm not going into much detail on how memory accesses work in this stage of the pipeline. That's intentional. There's a key to high-throughput pixel shading and other per-pixel or per-sample processing, but it's later in the pipeline, and we're not there yet. Everything will be revealed in due time :)

From → Coding, Graphics Pipeline

13 Comments

1. **KeyJ permalink**
   Why is the order of the primitives so important and needs to be preserved? As long as only fully opaque triangles are rendered (i.e. blending is off, which should be the case for the largest part of typical scenes), the rendering order shouldn't make any difference (except maybe some additional overdraw if the application sorted primitives by depth), or am I mistaken here?

   Reply
   - **fgiesen permalink**
     Suppose you're rendering a batch that has 3 triangles, all with the exact same vertex coordinates but different colors (this example is a bit contrived, but nevertheless completely valid). Let's say the first triangle is red, the second green, and the third blue.

     If you render this with "less" as compare mode onto a freshly cleared render target and depth buffer, you will only see the red triangle, If you use "less_equal", you will only see the blue triangle. That's what the functional API model (that is fully sequential) requires.

     Allowing reordering of triangles in the pipeline, even just for opaque rendering, breaks this: for both compare modes, you can make any of the 3 triangles come out "on top" using an appropriate ordering.

     This is a bit tricky, since you *can* allow triangle reordering within a batch if you only care about the contents of the Z-buffer (e.g. for shadow map rendering with a NULL pixel shader). For both compare functions, the Z-buffer at the end of the frame will contain at each location the minimum of all depth values that have been written there.

     However, the output of our Z-processing isn't just the Z-buffer, it's also an updated coverage mask that tells us which pixels/samples to shade. And for the coverage mask computation, order matters.

     Reply
2. **TomF permalink**
   I believe it's common in HW to have all real Z done late. That result is then fed back "upstream" into the coarse Z unit, and the only sort of early Z is coarse Z. This is conservative and can be "late" – a triangle occluded by the previous triangle will still be shaded – but it is always safe and requires no mode-switching shenanigans.

   Reply
   - **fgiesen permalink**

I'm not sure which type of implementation is more common these days, but I've definitely worked with chips that support both pre- and post-shading Z (including ones that don't do any coarse Z). But I've never seen anyone use anything but the obvious brute-force solution for the late Z->early Z transition: just flush the pixel pipeline when you're doing that switch.

Reply

3. **piyush permalink**
   contrary to popular belief, computer scholars do have a sense of humor.
   and you, sir are a living example

   Reply
4. **Martin Wardener permalink**
   "early Z is, well, earlier in the pipeline than early Z".. I assume you mean "..than late Z"..?

   Reply
   - **fgiesen permalink**
     Indeed! Thank you, fixed.

     Reply
5. **PixInverse permalink**
   I'm little confused by your explanation of perspective correct interpolation. Once we have performed the perspective divide x/w, y/w, z/w, then Z (z/w), the value we want to interpolate, doesn't vary linearly anymore across the surface of the 2D triangle. What now varies linearly is 1 / Z.
   To interpolate a vertex attribute correctly, we first need to divide by the vertex attribute value by Z of the vertex it is defined to, then linearly interpolate them, and then finally multiply the result by Z-Interpolated, which is the depth of the point on the triangle, that the pixel overlaps.

   Zcorrect = 1 / (Interpolated 1/ Z)
   Ucorrect = (Interpolated u / Z) * Zcorrect
   Vcorrect = (Interpolated v / Z) * Zcorrect

   In your post above, you says u/w, v/w and z / w can be interpolated linearly for perspective correct interpolation. Could you please explain if I'm missing anything here?

   Reply
   - **fgiesen permalink**
     Z=z/w ends up being an affine function of screen-space X and Y and is what's used for depth buffering.

     It's _never_ used for interpolation. Remember Z is set up by the projection matrix to reach (depending on the convention) either 0 and 1 or -1 and 1 at znear and zfar, respectively.

     What you use for interpolation is 1/w (which is also linear in screen space). Normally you set up perspective-corrected barycentric coordinates (usually called I and J) and then you interpolate I/w, J/w and 1/w in screen space, and then for every pixel you solve I=(I/w) / (1/w), J=(J/w) / (1/w) and use I and J to interpolate attributes. (As many as you want, and without having to individually divide them all through by the vertex w's).

     Reply

Trackbacks & Pingbacks

1. A trip through the Graphics Pipeline 2011: Index « The ryg blog
2. A trip through the Graphics Pipeline 2011, part 8 « The ryg blog
3. A trip through the Graphics Pipeline 2011, part 9 « The ryg blog
4. Linear Depth | The Devil In The Details