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

# A trip through the Graphics Pipeline 2011, part 10

July 20, 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/)**.*

Welcome back. Last time, we dove into bottom end of the pixel pipeline. This time, we'll switch back to the middle of the pipeline to look at what is probably the most visible addition that came with D3D10: Geometry Shaders. But first, some more words on how I decompose the graphics pipeline in this series, and how that's different from the view the APIs will present to you.

## There's multiple pipelines / anatomy of a pipeline stage

This goes back to **part 3 (https://fgiesen.wordpress.com/2011/07/03/a-trip-through-the-graphics-pipeline-2011-part-3/)**, but it's important enough to repeat it: if you look in, for example, the D3D10 documentation, you'll find a diagram of the "D3D10 pipeline" that includes all stages that might be active. The "D3D10 pipeline" includes Geometry Shading, even if you don't have a Geometry shader set, and the same for Stream-Out. In the purely functional model of D3D10, the Geometry Shading stage is always there; if you don't set a Geometry Shader, it just happens to be very simple (and boring): data is just passed through unmodified to the next pipeline stage(s) (Rasterization/Stream-Out).

That's the right way to specify the API, but it's the wrong way to think about it in this series, where we're concerned with how that functional model is actually implemented in hardware. So how do the two shader stages we've seen so far look? For VS, we went through the Input Assembler, which prepared a block of vertices for shading, then dispatched that batch to a shader unit (which chews on it for a while), and then some time later we get the results back, write them into a buffer (for Primitive Assembly), make sure they're in the right order, then send them down to the next pipeline stage (Culling/Clipping etc.). For PS, we receive to-be-shaded quads from the rasterizer, batch them up, buffer them for a while until a shader unit is free to accept a new batch, dispatch a batch to a shader unit (which chews on it for a while), and then some time later we get the results back, write them into a buffer (for the ROPs), make sure they're in the right order, then do blend/late Z and send the results on to memory. Sounds kind of familiar, doesn't it?

In fact, this is how it *always* looks when we want to get something done by the shader units: we need a buffer in the front, then some dispatching logic (which is in fact pretty universal for all shader types and can be shared), then we go wide and run a bunch of shaders in parallel, and finally we need another buffer and a unit that sorts the results (which we received potentially out-of-order from the shader units) back into API order.

We've seen shader units (and shader execution) and we've seen dispatch; and in fact, now that we've seen Pixel Shaders (which have some peculiarities like derivative computation, helper pixels, discard and attribute interpolation), we're not gonna see any big additions to shader unit functionality until we get to Compute Shaders, with their specialized buffer types and atomics. So for the next few parts, I won't be talking about the shader units; what's really different about the various shader types is the shape and interpretation of data that goes in and comes out. The shader parts that don't deal with IO (arithmetic, texture sampling) stay the same, so I won't be talking about them.

## The Shape of Tris to Shade

So let's have a look at how our IO buffers for Geometry Shaders look. Let's start with input. Well, that's reasonably easy – it's just what we wrote from the Vertex Shader! Or well, not quite; the Geometry Shader looks at primitives, not individual vertices, so what we really need is the output from Primitive Assembly (PA). Note that there's multiple ways to deal with this; PA could expand primitives out (duplicating vertices if they're referenced multiple times), or it could just hand us one block of vertices (I'll stick with the 32 vertices I used earlier) with an associated small "index buffer" (since we're indexing into a block of 32 vertices, we just need 5 bits per index). Either way works fine; the former is the natural input format for the clip/cull I discussed after PA, but the latter needs far less buffer space when running GS, so I'll use that model here.

One reason you need to worry about amount of buffer space with GS is that it can work on some pretty large primitives, because it doesn't just support plain lines or triangles (2 and 3 vertices per primitive respectively), but also lines/triangles with adjacency information (4/6 vertices per primitive). And D3D11 adds input primitives that are much fatter still – a GS can also consumes patches with up to 32 control points as input. Duplicating the vertices of e.g. a 16-control point patch, which could each have up to 16 vector attributes (32 with D3D11)? That'd be some serious memory waste. So I'm assuming non-duplicated, indexed vertices for this path. Which makes the input for a batch of primitives: the VS output, plus a (relatively small) index buffer.

Now, the geometry shader runs per primitive. For vertex shaders, we needed to gather a batch of vertices, and we chose our batch size with a simple greedy algorithm that tries to pack as many vertices into a batch as possible without splitting a primitive across multiple batches – fair enough. And for pixel shading, we get plenty of quads from the rasterizer and pack them all into batches. Geometry Shaders are a bit more inconvenient – our input block is guaranteed to contain at least one full primitive, and possibly several – but other than that, the number of primitives in that block completely depends on the vertex cache hit rate. If it's high and we're using triangles, we might get something like 40-43; if we're using triangles with adjacency information we could have as little as 5 if we're unlucky.

Of course, we could try to collect primitives from several input blocks here, but that's kind of awkward too. Now we need to keep multiple input blocks and index buffers around for a single GS batch, and if a single batch can refer to multiple index buffers that means each primitive in that batch now needs to know where to get the indices and vertex data from – more storage requirements, more management, more overhead. Also ugly. And of course even with two input blocks you're still at crappy utilization if you hit two input batches with low vertex cache hit rate. You can support more input blocks, but that eats away at memory – and remember, you need space for the output geometry too (I'll get to that in a bit).

So this is our first snag: with VS, we could basically pick our target batch size, and we chose to not always generate full batches so as to make our lives in PA (and here in the GS, and later in the HS too) a bit easier. With PS, we always shade quads, and even fairly small tris usually hit multiple quads so we get an okay ratio of number of quads to number of tris. But with GS, we don't have full control over either ends of the pipeline (since we're in the middle!), and we need multiple input vertices per primitive (as opposed to multiple quads per one input triangle), so buffering up a lot of input is expensive (both in terms of memory and in the amount of management overhead we get).

At this stage, you can basically pick how many input blocks you're willing to merge to get one block of primitives to geometry shade; that number is going to be low because of the memory requirements (I'd be very surprised to see more than 4), and depending on how important you judge GS to be, you might even pick 1, i.e. don't merge across input blocks at all and live with crappy utilization on GS shading blocks/Warps/Wavefronts! That's not great with triangles and really bad with the primitives that have even more vertices, but not much of an issue when your main use case for GS in practice is expanding points to quads (point sprites) and maybe rendering the occasional cube shadow map (using the Viewport Array Index/Rendertarget Index – I'll get to that in a bit).

## GS output: no rose garden over here, either

So how's it looking on the output side? Again, this is more complicated than the plain VS data flow. Much more complicated in fact; while a VS only outputs one thing (shaded vertices) with a 1:1 correspondence between unshaded and shaded vertices, a GS outputs a variable number of vertices (up to a maximum that's specified at compile time), and as of D3D11 it can also have multiple output streams – however, a maximum of one stream can be sent on down the rest of the pipeline, which is the path I'm talking about now. The other destination for GS data (Stream-Out) will be covered in the next part.

A GS produces variable-sized output, but it needs to run with bounded memory requirements (among other things, the amount of memory available for buffers determines how many primitives can be Geometry Shaded in parallel), which is why the maximum number of output vertices is fixed at compile-time. This (together with the number of written output attributes) determines how much buffer space is allocated, and thus indirectly the maximum number of parallel GS invocations; if that number is too low, latency can't be fully hidden, and the GS will stall for some percentage of the time.

Also note that the GS inputs *primitives* (e.g. points, lines, triangles or patches, optionally with adjacency information), but outputs *vertices* – even though we send primitives down to the rasterizer! If the output primitive type is points, this is trivial. For lines and triangles however, we need to reassemble those vertices back into primitives again. This is handled by making the output vertices form a line or triangle strip, respectively. This handles what are perhaps the 3 most important cases well: single lines, triangles, or quads. It's not so convenient if the GS tries to do some actual extrusion or generate otherwise "complicated" geometry, which often needs several "restart strip" markers (which boils down to a single bit per vertex that denotes whether the current strip is continued or a new strip is started). So why the limitation? At the API level, it seems fairly arbitrary – why can't the GS just output a vertex list together with a small index buffer?

The answer boils down to two words: Primitive Assembly. This is what we're doing here – taking a number of vertices and assembling them into a full primitive to send down the pipeline. But we already use that functional block in this data path, just in front of the GS. So for GS, we need a second primitive assembly stage, which we'd like to keep simple, and assembling triangle strips is very simple indeed: a triangle is always 3 vertices from the output buffer in sequential order, with only a bit of glue logic to keep track of the current winding order. In other words, strips are not significantly more complex to support than what is arguably the simplest primitive of all (non-indexed lines/triangles), but they still save output buffer space (and hence give us more potential for parallelism) for typical primitives like quads.

## API order again

There's a few problems here, however: in the regular vertex shading path, we know exactly how many primitives there are in a batch and where they are, even before the shaded vertices arrive at the PA buffer – all this is fixed from the point where we set up the batches to shade. If we, for example, have multiple units for cull/clip/triangle setup, they can all start in parallel; they know where to get their vertex data from, and they can know ahead of time which "sequence number" their triangle will have so it can all be put into order.

For GS, we don't generally know how many primitives we're gonna generate before we get the outputs back – in fact, we might not have produced any! But we still need to respect API order: it's first all primitives generated from GS invocation 0, then all primitives from invocation 1, and so on, through to the end of the batch (and of course the batches need to be processed in order too, same as with VS). So for GS, once we get results back, we first need to scan over the output data to determine the locations where complete primitives start. Only then can we start doing cull, clip and triangle setup (potentially in parallel). More extra work!

## VPAI and RTAI

These are two features added with GS that don't actually affect Geometry Shader execution, but do have some effect on the processing further downstream, so I thought I'd mention them here: The Viewport Array Index (here, VPAI for short) and Render-target Array Index (RTAI). RTAI first, since it's a bit easier to explain: as you hopefully know, D3D10 adds support for texture arrays. Well, the RTAI gives you render-to-texture-array support: you set a texture array as render target, and then in the GS you can select per-primitive to which array index the primitive should go. Note that because the GS is writing vertices not primitives, we need to pick a single vertex that selects the RTAI (and also VPAI) per primitive; this is always the "leading vertex", i.e. the first specified vertex that belongs to a primitive. One example use case for RTAI is rendering cubemaps in one pass: the GS decides per primitive to which of the cube faces it should be sent (potentially several of them). VPAI is an orthogonal feature which allows you to set multiple viewports and scissor rects (up to 15), and then decide per primitive which viewport to use. This can be used to render multiple cascades in a Cascaded Shadow Map in a single pass, for example, and it can also be combined with RTAI.

As said, both features don't affect GS processing significantly – they're just extra data that gets tacked onto the primitive and then used later: the VPAI gets consumed during the viewport transform, while the RTAI makes it all the way down to the pixel pipeline.

## Summary so far

Okay, so there's some amount of trouble on the input end – we don't fully get to pick our input data format, so we need extra buffering on the input data, and even then we have a variable amount of input primitives which we're not necessarily going to be able to partition into nice big batches. And on the output end, we're again assembling a variable number of primitives, don't necessarily know which GS will produce how many primitives in advance (though for some GSs we'll be able to determine this statically from the compiled code, for example because all vertex emits are outside of flow control or inside loops with a known iteration count and no early-outs), and have to spend some time parsing the output before we can send it on to triangle setup.

If that sounds more involved than what we had in the VS-only case, that's because it is. This is why I mentioned above that it's a mistake to think of the GS as something that always runs – even a very simple GS that does nothing except pass the current triangle through goes through two more buffering stages, an extra round of primitive assembly, and might execute on the shader units with poor utilization. All of this has a cost, and it tends to add up: I checked it when D3D10 hardware was fairly new, and on both AMD and NVidia hardware, even a pure pass-through GS was between 3x and 7x slower than no GS at all (in a geometry-

limited scenario, that is). I haven't re-run this experiment on more recent hardware; I would assume that it's gotten better by now (this was the first generation to implement GS, and features don't usually have good performance in the first GPU generation that implements them), but the point still stands: just sending something through the GS pipe, even if nothing at all happens there, has a very visible cost.

And it doesn't help that GSs produce primitives as strips, sequentially; for a Vertex Shader, we get one invocation per vertex, which reads one vertex and writes one vertex (nice). For a GS, though, we might end up having only a batch of 11 GSs running (because there wasn't enough primitives in the input buffer), with each of them running fairly long and producing something like 8 output vertices. That's a long time to be running at low utilization! (Remember we need somewhere between 16 and 64 independent jobs per batch we dispatch to the shader units). It's even more annoying if the GS mainly consists of a loop – for example, in the "render to cube map" case I mentioned for RTAI, we loop over the 6 faces in a cube, check if a triangle is visible on that face, and output a triangle if that's the case. The computations for the 6 faces are really independent; if possible, we'd like to run them in parallel!

## Bonus: GS Instancing

Well, enter GS Instancing, another feature new in D3D11 – poorly documented, sadly (and I'm not sure if there's any good examples for it in the SDK). It's fairly simple to explain, though: for each input primitive, the GS gets run not just once but multiple times (this is a static count selected at compile time). It's basically equivalent to wrapping the whole shader in a

```
for (int i = 0; i < N; i++)
{
    // ...
}
```

block, only the loop is handled outside the shader by actually generating multiple GS invocations per input primitive, which helps us get larger batch sizes and thus better utilization. The `i` is exported to the shader as a system-generated value (in D3D11, with Semantic `SV_GSInstanceID`). So if you have a GS like that, just get rid of the outer loop, add a `[instances(N)]` declaration and declare `i` as input with the right semantic and it'll probably run faster for very little work on your part – the magic of giving more independent jobs to a massively parallel machine!

Anyway, that's it on Geometry Shaders. I've skipped Stream-Out, but this post is already long enough, and besides SO is a big enough topic (and independent enough of GS!) to warrant its own post. Next post, to be more precise. Until then!

From → Coding, Graphics Pipeline

**2 Comments**

## Trackbacks & Pingbacks

1. A trip through the Graphics Pipeline 2011: Index « The ryg blog
2. Page not found « The ryg blog