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

A trip through the Graphics Pipeline 2011, part 11

August 14, 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! This time, the focus is going to be on Stream-Out (SO). This is a facility for storing the Output of the Geometry Shader stage to memory, instead of sending it down the rest of the pipeline. This can be used to e.g. cache skinned vertex data, or as a sort of poor man's Compute Shader on D3D10-level hardware using the D3D10 API (note that with D3D11, you can just use CS 4.0, even on D3D10 hardware). And just like the GS Instancing I mentioned last time, some of this is very poorly described in the API docs, so I'll have a few comments about API usage even though it's technically out of the intended scope of this series.

Vertex Shader Stream-Out (i.e. SO with NULL GS)

This is one of the features that's not properly explained in the D3D10 (or D3D11, for that matter) docs; in fact, it's not mentioned there at all except for a small throwaway remark in "Getting Started with the Stream-Output Stage (Direct3D 10)". You're supposed to figure it out from the examples – which themselves don't exactly go out of their way to make it clear what's going on. That's a pity – VS Stream-Out is easier than GS SO, and has some pretty useful applications by itself (e.g. caching skinned vertices).

So here's how it's done in D3D10 and 11: You simply pass Vertex Shader bytecode (instead of GS bytecode) to `CreateGeometryShaderWithStreamOutput`. Yes, the docs mention something about "Size of the compiled geometry shader" here – ignore it. What you get back is a Geometry Shader object that you can then pass to `GSSetShader`. This is, in effect, a NULL Geometry Shader – it doesn't actually go through GS processing. It's just some wrapper (more like duct tape really) to make it fit into the API model, where all rendering passes through the GS stage and SO comes right after GS – though as I've explained last time, actual HW tends to skip the GS stage completely when there's no GS set.

So the shaded vertices get assembled into primitives as before, but instead of getting sent down the rest of the pipeline as already described, they get forwarded to Stream-Out, where they arrive – as always – in a buffer. What exactly happens with them then depends on the Stream-Out declaration (which is passed at creation time). In the Stream-Out declaration, the app gets to specify where it wants each output vector to end up in the Stream-Out targets (or SO targets for short). If the SO declaration "matches" the Vertex Shader Output Declaration (i.e. the same attributes in the same order), data from the input buffers can be streamed more or less unprocessed into memory. If it doesn't match the declaration exactly – it might skip some attributes written by the shader, or write them in a different order – either way, there's some extra reordering involved. This might involve a dedicated reordering unit (which basically implements a gather-type operation from the SO input buffers), or it might involve generating lots of small memory writes instead of large burst writes, or something similar. Either way, it's extra effort and generally slower; the details of what exactly triggers a slow path depend on the hardware specifics, but really, it doesn't matter that much. If you want optimal SO performance, just make sure the SO declaration and Output declarations agree.

Another point is that SO usually doesn't have access to a very high-performance path to the memory subsystem. Unlike e.g. the ROPs, SO isn't really (yet?) a full citizen in current GPU designs, so it often only has access to one memory channel or something of the sort. That's something to keep in mind if you're producing a lot of data via SO. This is compounded by SO outputs always being full floats, so there's no way to conserve bandwidth by using one of the packed vertex data types.

Final remark on VS SO: As I mentioned earlier, SO operates on assembled *primitives*, not individual vertices. Note that Primitive Assembly discards adjacency information if it makes it that far down the pipeline, and since this happens before SO, vertices corresponding to adjacency info won't make it into SO buffers either. SO working on primitives not individual vertices is relevant for use cases like instancing a single skinned mesh (in a single pose) several times. If you were to draw your triangle mesh as you usually would and then use SO on that, this results in a data explosion – you get 3 unpacked, unshared vertices per input primitive. This works, but isn't exactly an efficient use of bandwidth, both on the SO and the later vertex input side. Instead, you should draw your triangle mesh as a (non-indexed) point list in the first pass, thereby shading each vertex exactly once. The SO buffer then ends up in 1:1 correspondence to your original vertex buffer, only with skinned instead of non-skinned vertices. You can then use that vertex buffer with your original primitive topology and index buffer.

Geometry Shader SO: Multiple streams

This basically works like SO with a NULL GS, except there's a Geometry Shader involved, which adds some new capabilities (and complications). In the VS case, we just had one output stream (note that streams are a D3D11+ feature – they don't exist on D3D10-level HW). That stream could be sent to SO or not, and it could also be sent to down the pipeline to viewport/clip/cull or not, but that's it. But Geometry Shaders allow multiple streams, which makes output routing a bit more difficult.

Basically, every GS can write to (as of D3D11) up to 4 streams. Each stream may be sent on to SO targets – yes, plural: a single stream can write to multiple SO targets, but a single SO target can receive values from only one stream, i.e. this is a one-to-many relationship, not a fully general many-to-many one. The presence of streams has some implications for SO buffering – instead of a single input buffer like I described in the NULL GS case, we now may have multiple input buffers, one per stream. In addition to SO targets, up to one stream may be sent down the pipe – i.e. the regular rendering pipeline and SO may be used simultaneously.

As in the NULL GS case, SO works on primitives, not individual vertices – that is, the strips you output in the GS get expanded out to full lines or triangles before they get into SO.

Tracking output size

There's another issue here: we don't necessarily know how much output data is going to be produced from SO. For GS, this comes about because each GS invocation may produce a variable number of output primitives; but even in the simpler VS case, as soon as indexed primitives are involved, the app might slip some "primitive cut" indices in there that influence how many primitives actually get written. This is a problem if we then want to draw from that SO buffer later, because we don't know how many vertices are actually in there! We do have an upper bound – the maximum capacity of the buffer as created – but that's it. Now, this could be resolved using some kind of query mechanism, but once you think it through, that seems fairly backwards: at the point we're using the SO buffer for drawing, we obviously do know how many primitives we actually wrote – the SO unit needs to keep track of its current output position, after all! If we employed some query mechanism, we would end up transporting that single 32-bit value back over the bus to the driver, which passes it on to the API, which passes it on to the app – which then immediately dispatches another draw, going through all the layers again in the opposite direction.

So that's not how it's solved. Instead, there's `DrawAuto`. The idea is very simple – the GPU already knows how many valid vertices it actually wrote to the output buffer; the SO unit keeps track of that while it's writing, and the final counter is also kept in memory (along with the buffer) since the app may render to a SO buffer in multiple passes. This counter is then used for DrawAuto, instead of having the app submit an explicit count itself – simplifying things considerably and avoiding the costly round-trip completely. Note that this query mechanism does exist – both for checking the number of vertices written and to determine whether an overflow occurred. But it's not on the critical path for rendering from SO buffers, which makes things a lot simpler for driver developers.

And that's it for SO, really. Not really a lot of HW info in this one, and not really a super-interesting topic from a pipeline perspective, which is why it took me so long to finish; sorry about that. Next up is Tessellation – this should be a lot quicker, since it's a fun topic :)

From → Coding, Graphics Pipeline

One Comment

Trackbacks & Pingbacks

1. A trip through the Graphics Pipeline 2011: Index « The ryg blog