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

A trip through the Graphics Pipeline 2011, part 12

September 6, 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, we'll look into what is perhaps the "poster boy" feature introduced with the D3D11 / Shader 5.x hardware generation: Tessellation. This one is interesting both because it's a fun topic, and because it marks the first time in a long while that a significant user-visible component has been added to the graphics pipeline that's not programmable.

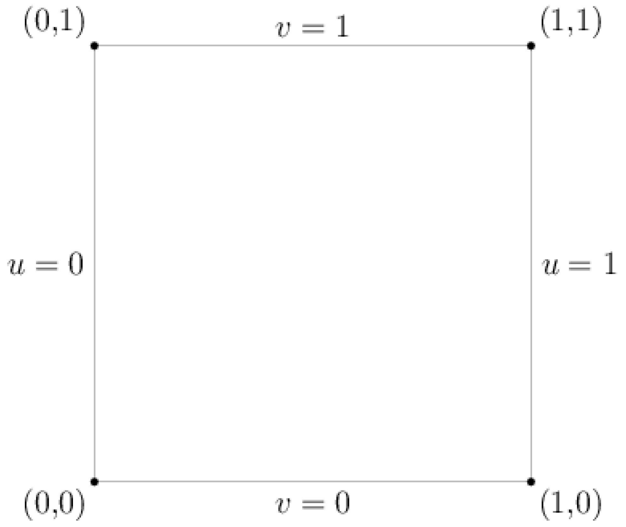
Unlike Geometry Shaders, which are conceptually quite easy (it's just a shader that sees whole primitives as opposed to individual vertices), the topic of "Tessellation" requires some more explanation. There's tons of ways to tessellate geometry – to name just the most popular ones, there's Spline Patches in dozens of flavors, various types of Subdivision Surfaces, and Displacement Mapping – so from the bullet point "Tessellation" alone it's not at all obvious what services the GPU provides us with, and how they are implemented.

To describe how hardware tessellation works, it's probably easiest to start in the middle – with the actual primitive tessellation step, and the various requirements that apply to it. I'll get to the new shader types (Hull Shaders and Domain Shaders in D3D11 parlance, Tessellation Control Shader and Tessellation Evaluation Shader in OpenGL 4.0 lingo) later.

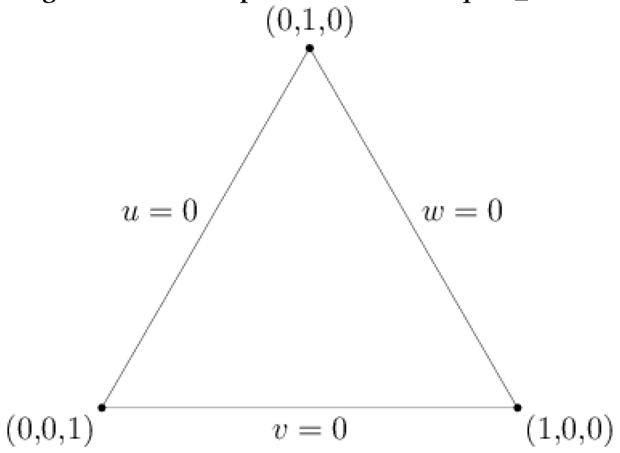
Tessellation – not quite like you'd expect

Tessellation as implemented by Shader 5.x class HW is of the "patch-based" variety. Patch types in the CG literature are mostly named by what kind of function is used to construct the tessellated points from the control points (B-spline patches, Bézier triangles, etc.). But we'll ignore that part for now, since it's handled in the new shader types. The actual fixed-function tessellation unit deals only with the *topology* of the output mesh (i.e. how many vertices there are and how they're connected to each other); and it turns out that from this perspective, there's basically only two different types of patches: quad-based patches, which are defined on a parameter domain with two orthogonal coordinate axes (which I'll call u and v here, both are in [0,1]) and usually constructed as a tensor product of two one-parameter basis functions, and triangle-based patches, which use a redundant representation with three coordinates (u, v, w) based on barycentric coordinates (i.e. $u, v, w \geq 0, u + v + w = 1$). In D3D11 parlance, these are the "quad" and "tri" domains, respectively. There's also an "isoline" domain which instead of a 2D surface produces one or multiple 1D curves; I'll treat it the same way as I did lines and point primitives throughout this series: I acknowledge its existence but won't go into further detail.

Tessellated primitives can be drawn naturally in their respective domain coordinate systems. For quads, the obvious choice of drawing the domain is as a unit square, so that's what I'll use; for triangles, I'll use an equilateral triangle to visualize things. Here's the coordinate systems I'll be using in this post with both the vertices and edges labeled:

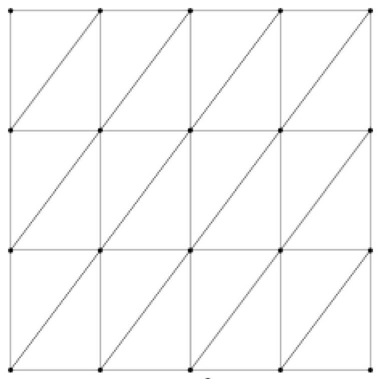


(https://fgiesen.files.wordpress.com/2011/09/quad_coords2.png)

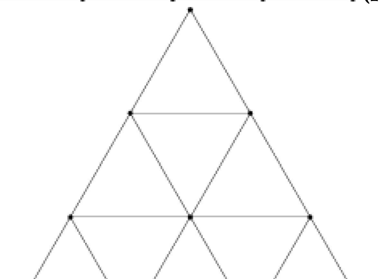


(https://fgiesen.files.wordpress.com/2011/09/tri_coords.png)

Anyway, both triangles and quads have what I would consider a "natural" way to tessellate them, depicted below. But it turns out that's not actually the mesh topology you get.

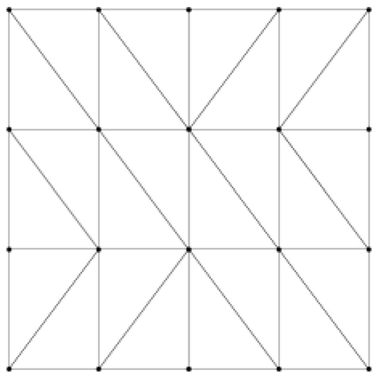
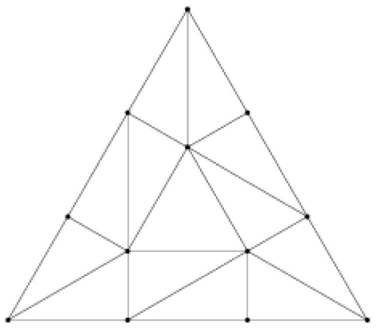


(https://fgiesen.files.wordpress.com/2011/09/quad_tess_simple1.png)



(https://fgiesen.files.wordpress.com/2011/09/tri_tess_simple1.png)

Here's the *actual* meshes that the tessellator will produce for the given input parameters:


(https://fgiesen.files.wordpress.com/2011/09/quad_tess4x3.png)

(https://fgiesen.files.wordpress.com/2011/09/tri_tess3.png)

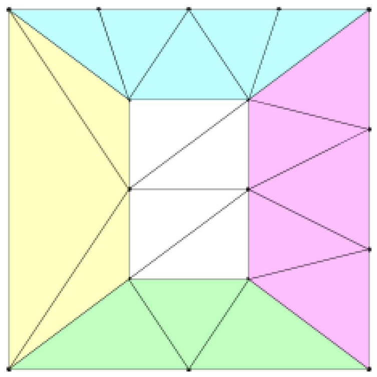
For quads, this is (roughly) what we're expecting – except for some flipped diagonals, which I'll get to in a minute. But the triangle is a completely different beast. It's got a very different topology from the "natural" tessellation I showed above, including a different number of vertices (12 instead of 10). Clearly, there's something funny going on here – and that something happens to be related to the way transitions between different tessellation levels are handled.

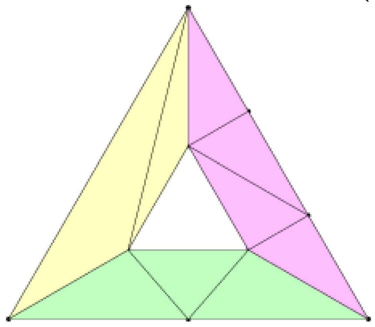
Making ends meet

The elephant in the room is handling transitions between patches. Tessellating a single triangle (or quad) is easy, but we want to be able to determine tessellation factors per-patch, because we only want to spend triangles where we need them – and not waste tons of triangles on some distant (and possibly backface-culled) parts of the mesh. Additionally, we want to be able to do this quickly and ideally without extra memory usage; that means a global fix-up post-pass or something of that caliber is out of the question.

The solution – which you've already encountered if you've written a Hull or Domain shader – is to make all of the actual tessellation work purely local and push the burden of ensuring watertightness for the resulting mesh down to the shaders. This is a topic all by itself and requires, among other things, **great care in the Domain Shader code (http://www.ludicon.com/castano/blog/2010/09/precise/)**; I'll skip all the details about expression evaluation in shaders and stick with the basics. The basic mechanism is that each patch has multiple tessellation factors (TFs), which are computed in the Hull Shader: one or two for the actual inside of the patch, plus one for each edge. The TFs for the inside of the patch can be chosen freely; but if two patches share an edge, they'd better compute the exact same TFs along that edge, or there will be cracks. The hardware doesn't care – it will process each patch by itself. If you do everything correctly, you'll get a nice watertight mesh, otherwise – well, that's your problem. All the HW needs to make sure is that it's *possible* to get watertight meshes, preferably with reasonable efficiency. That by itself turns out to be tricky in some places; I'll get to that later.

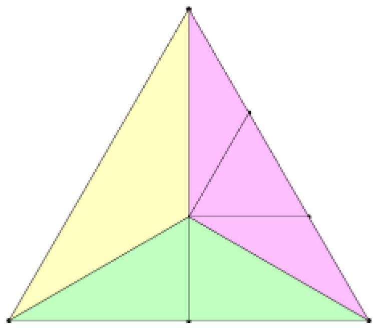
So, here are some new reference patches – this time with different TFs along each edge so we can see how that works:


(https://fgiesen.files.wordpress.com/2011/09/quad_tess_asym.png)


(https://fgiesen.files.wordpress.com/2011/09/tri_tess_asym.png)

I've colored the areas influenced by the different edge tessellation factors; the uncolored center part in the middle only depends on the inside TFs. In these images, the u=0 (yellow) edge has a TF of 2, the v=0 (green) edge has a TF of 3, the u=1 / w=0 (pink) edge has a TF of 4, and the v=1 (quad only, cyan) edge has a TF of 5 – exactly the number of vertices along the corresponding outer edge. As should be obvious from these two images, the basic building block for topology is just a nice way to stitch two subdivided edges with different number of vertices to each other. The details of this are somewhat tricky, but not particularly interesting, so I won't go into it.

As for the inside TFs, quads are fairly easy: The quad above has an inside TF of 3 along u and 4 along v. The geometry is basically that of a regular grid of that size, except with the first and last rows/columns replaced by the respective stitching triangles (if any edge has a TF of 1, the resulting mesh will have the same structure as if the inside TFs for u/v were both 2, even if they're smaller than that). Triangles are a bit more complicated. Odd TFs we've already seen – for a TF of N, they produce a mesh consisting of $\frac{N+1}{2}$ concentric rings, the innermost of which is a single triangle. For even TFs, we get $\frac{N}{2}$ concentric rings with a center vertex instead of a center triangle. Below is an image of the simplest even case, $N = 2$, which consists just of edge stitches plus the center vertex.


(https://fgiesen.files.wordpress.com/2011/09/tri_tess_asym_even.png)

Finally, when triangulating quads, the diagonal is generally chosen to point away from the center of the patch (in the domain coordinate space), with a consistent tie-breaking rule. This is simply to ensure maximum rotational symmetry of the resulting meshes – if there's extra degrees of freedom, might as well use them!

Fractional tessellation factors and overall pipeline flow

So far, I've only talked about integer TFs. In two of the so-called "partitioning types", namely "Integer" and "Pow2", that's all the Tessellator sees. If the shader generates a non-integer (or, respectively, non-power-of-2) TF, it will simply get rounded up to the next acceptable value. More interesting are the remaining two partitioning types: Fractional-odd and Fractional-even tessellation.

Instead of jumping from tessellation factor to tessellation factor (which would cause visible pops), new vertices start out at the same position as an existing vertex in the mesh and then gradually move to their new positions as the TF increases.

For example, with fractional-odd tessellation, if you were to use an inner TF of 3.001 for the above triangle, the resulting mesh would look very much like the mesh for a TF of 3 – but topologically, it'd be the same as if the TF was 5, i.e. it's a patch with 3 concentric rings, even though the middle ring is very narrow. Then as the TF gets closer to 5, the middle ring expands until it is eventually at its final position for TF 5. Once you raise the TF past 5, the mesh will be topologically the same as is the TF was 7, but again with a number of almost-degenerate triangles in the middle, and so forth. Fractional-even tessellation uses the same principle, just with even TFs.

The output of the tessellator then consists of two things: First, the positions of the tessellated vertices in domain coordinates, and second, the corresponding connectivity information – basically an index buffer.

Now, with the basic function of the fixed-function tessellator unit explained, let's step back and see what we need to do to actually churn out primitives: First, we need to input a bunch of input control points comprising a patch into the Hull Shader. The HS then computes output control points and "patch constants" (both of which get passed down to the Domain Shader), plus all Tessellation Factors (which are essentially just more patch constants). Then we run the fixed-function tessellator, which gives us a bunch of Domain Positions to run the Domain Shader at, plus the associated indices. After we've run the DS, we then do another round of primitive assembly, and then send the primitives either down to the GS pipeline (if it's active) or Viewport transform, Clip and Cull (if not).

So let's look a bit into the HS stage.

Hull Shader execution

Like **Geometry Shaders (https://fgiesen.wordpress.com/2011/07/20/a-trip-through-the-graphics-pipeline-2011-part-10/)**, Hull Shaders work on full (patch) primitives as input – with all the input buffering headaches that causes. How much of a headache entirely depends on the type of input patch. If the patch type is something like a cubic Bézier patch, we need 4×4 = 16 input points *per patch* and might just produce a single quad of output (or even none at all, if the patch is culled); clearly, that's a somewhat awkward amount of data to work with, and doesn't lend itself to very efficient shading. On the other hand, if tessellation takes plain triangles as input (which a lot of people do), input buffering is pretty tame and not likely to be a source of problems or bottlenecks.

More importantly, unlike Geometry Shaders (which run for every primitive), Hull Shaders don't run all that often – they run once *per patch*, and as long as there's any actual tessellation going on (even at modest TFs), we have way less patches than we have output triangles. In other words, even when HS input is somewhat inefficient, it's less of an issue than in the GS case simply because we don't hit it that often.

The other nice attribute of Hull Shaders is that, unlike Geometry Shaders, they don't have a variable amount of output data; they produce a fixed amount of control points, each which a fixed amount of associated attributes, plus a fixed amount of patch constants. All of this is statically known at compile time; no dynamic run-time buffer management necessary. If we Hull Shade 16 hulls at a time, we know exactly where the data for each hull will end up before we even start executing the shader. That's definitely an advantage over Geometry Shaders; for lots of Geometry Shaders, it's possible to know statically how many output vertices will be generated (for example because all the control flow leading to `emit / cut` instructions can be statically evaluated at compile time), and for all of them, there's a guaranteed maximum number of output vertices, but for HS, we have a guaranteed fixed amount of output data, no additional analysis required. In short, there's no problems with output buffer management, other than the fact that, again depending on the primitive type, we might need lots of output buffer space which limits the amount of parallelism we can achieve (due to memory/register constraints).

Finally, Hull Shaders are somewhat special in the way they are compiled in D3D11; all other shader types basically consist of one block of code (with some subroutines maybe), but Hull Shaders are generated factored into multiple phases, each of which can consist of multiple (independent) threads of execution. The details are mainly of interest to driver and shader compiler programmers, but suffice it to say that your average HS comes packaged in a form that exposes lots of latent parallelism, if there is any. It certainly seems like Microsoft was really keen to avoid the bottlenecks that plague Geometry Shaders this time around.

Anyway, Hull Shaders produce a bunch of output per patch; most of it is just kept around until the corresponding Domain Shaders run, except for the TFs, which get sent to the tessellator unit. If any of the TFs are less than or equal to zero (or NaN), the patch is culled, and the corresponding control points and patch constants silently get thrown away. Otherwise, the Tessellator (which implements the functionality described above) kicks in, reads the just-shaded patches, and starts churning out domain point positions and triangle indices, and we need to get ready for DS execution.

Domain Shaders

Just like for **Vertex Shading (https://fgiesen.wordpress.com/2011/07/03/a-trip-through-the-graphics-pipeline-2011-part-3/)** way back, we want to gather multiple domain vertices into one batch that we shade together and then pass on the PA. The fixed-function tessellator can take care of this: "just" handle it along with producing vertex positions and indices (I put the "just" in quotes here because this does involve some amount of bookkeeping).

In terms of input and output, Domain Shaders are very simple indeed: the only input they get that actually varies per vertex is the domain point u and v coordinates (w, when used, doesn't need to be computed or passed in by the tesselator; since $u + v + w = 1$, it can be computed as $w = 1 - u - v$). Everything else is either patch constants, control points (all of which are the same across a patch) or constant buffers. And output is basically the same as for Vertex Shaders.

In short, once we get to the DS, life is good; the data flow is almost as simple as for VS, which is a path we know how to run efficiently. This is perhaps the biggest advantage of the D3D11 tessellation pipeline over Geometry Shaders: the actual triangle amplification doesn't happen in a shader, where we waste precious ALU cycles and need to keep buffer space for a worst-case estimate of vertices, but in a localized element (the tessellator) that is basically a state machine, gets very little input (a few TFs) and produces very compact output (effectively an index buffer, plus a 2D coordinate per output vertex). Because of this, we need way less memory for buffering, and can keep our Shader Units busy with actual shading work instead of housekeeping.

And that's it for this post – next up: Compute Shaders, aka the final part in my original outline for this series! Until then.

Final remarks

As usual, I cut a few corners. There's the "isoline" patch type, which I didn't go into at all (if there's any demand for this, I can write it up). The Tessellator has all kinds of symmetry and precision requirements; as far as vertex domain positions are concerned, you can basically expect bit-exact results between the different HW vendors, because the D3D11 spec really nails this bit down. What's intentionally not nailed down is the order in which vertices or triangles are produced – an implementation can do what it wants there, provided it does so consistently (i.e. the same input has to produce the same output, always). There's a bunch of subtle constraints that go into this too – for example, all domain positions written by the Tessellator need to have both u and 1-u (and also v and 1-v) exactly representable as float; there's a bunch of necessary conditions like this so that Domain Shaders can then produce watertight meshes (this rule in particular is important so that a shared edge AB between two patches, which is AB to one patch and BA to the other, can get tessellated the same way for both patches).

Writing Domain Shaders so they actually can't produce cracks is tricky and requires great care; I intentionally sidestep the topic because it's outside the scope of this series. Another much more trivial issue that I didn't mention is the winding order of triangles generated by the Tessellator (answer: it's up to the App – both clockwise and counterclockwise are supported).

The description of Input/Output buffering for Hull and Domain shaders is somewhat terse, but it's very similar to stages we've already seen, so I'd rather keep it short and avoid extra clutter; re-read the posts on Vertex Shaders and Geometry Shaders if this was too fast.

Finally, because the Tesselation pipeline can feed into the GS, there's the question of whether it can generate adjacency information. For the "inside" of patches this would be conceivable (just more indices for the Tessellator unit to write), but it gets ugly fast once you reach patch edges, since cross-patch adjacency needs exactly the kind of global "mesh awareness" that the Tessellation pipeline design tries so hard to avoid. So, long story short, no, the tessellator will not produce adjacency information for the GS, just plain triangles.

From → Coding, Graphics Pipeline

12 Comments

1. **Naoki permalink**
   Awesome article, I learnt a lot. I have a question though. Could you elaborate a bit on how you compute inside tessellation factors? For example in the image of the square above, why is the inside TF equal to 3 along u and 4 along v?

   Thanks.

   Reply
   - **fgiesen permalink**

D3D inside TFs for quad domains count all rows and columns of the grid, including the ones covered by edge regions. If the inside TF were less than 2 (but still >0) in either u or v, the white "inside" region would just disappear completely. I mention this in the article – the "(if any edge has a TF of 1, the resulting mesh will have the same structure as if the inside TFs for u/v were both 2, even if they're smaller than that)" bit. By far the easiest way to understand how the factors interact is to write a small app that renders a single primitive in wireframe mode and then play with the values for a bit. :)

Reply

- **Naoki permalink**
  Yes, I'll probably do that :) Thanks for the explanation.

2. **Thanh Nguyen permalink**
   Another awesome article! Your "trip through the graphics pipeline" series is very informative and helpful. I have a question though: your image for quad primitive vertices/edges ordering is in ccw order. D3D11 doc is saying they should be in cw order: http://msdn.microsoft.com/en-us/library/windows/desktop/ff471574(v=vs.85).aspx
   So which one is the correct order?

   Reply

   - **fgiesen permalink**
     Sorry for taking a while to reply. I show the coordinate system I use at the top of the article. I use the standard mathematical convention with (0,0) being the origin, the positive x axis pointing right and the positive y axis pointing up. In that coordinate system, the D3D edge ordering (u=0, then v=0, then u=1, then v=1) is a counter-clockwise sweep. The D3D docs put the v=0 axis at the "top of the patch", which means they refer to a patch coordinate system where (0,0) is in the top-left corner and positive y points downwards. That corresponds to a y-flip of the coordinate system, which turns counter-clockwise sweeps into clockwise sweeps (and vice versa). If you prefer to draw it that way, just mirror all my figures about the x axis :)

     It's fairly arbitrary either way, since the patch uv coordinate system is something that's entirely up to the user; you get to pick whatever you like. The only thing you need to be careful about is the winding order of the output triangles (the "output topology"): D3D supports both "triangle_cw" and "triangle_ccw", but they refer to clockwise or counterclockwise in their UV space with (0,0) being the top left corner.

     Reply

     - **Thanh Nguyen permalink**
       Ah, I see. Thanks for the clarification.

3. **Sven Kiesser permalink**
   Great article and well explained, i have one question though:
   Why are there two different types of fractional_spacing (odd and even)? Is one type not enough?

   Reply

   - **fgiesen permalink**
     I don't know what the rationale was for including both. For what it's worth, fractional_odd seems to be far more popular in practice.

     Reply

4. **Paul Frischknecht permalink**
   Thanks. The sample "AdaptiveTessellationCS40" that was included with some dx sdks shows the formulas/algorithms for the tessellation strategies for those interested.

   Reply

Trackbacks & Pingbacks

1. A trip through the Graphics Pipeline 2011: Index « The ryg blog
2. » Quad Patch Tessellation in Unity Defective Studios Devblog
3. Good resource on tessellation subdivison pattern | Technology & Programming Answers

Blog at WordPress.com.