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

A trip through the Graphics Pipeline 2011, part 8

July 10, 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 part, I'll be dealing with the first half of pixel processing: dispatch and actual pixel shading. In fact, this is really what most graphics programmer think about when talking about pixel processing; the alpha blend and late Z stages we'll encounter in the next part seem like little more than an afterthought. In hardware, the story is a bit more complicated, as we'll see – there's a reason I'm splitting pixel processing into two parts. But I'm getting ahead of myself. At the point where we're entering this stage, the coordinates of pixels (or, actually, quads) to shade, plus associated coverage masks, arrive from the rasterizer/early-Z unit – with triangle in the exact same order as submitted by the application, as I pointed out last time. What we need to do here is to take that linear, sequential stream of work and farm it out to hundreds of shader units, then once the results are back, we need to merge it back into one linear stream of memory updates.

That's a textbook example of fork/join-parallelism. This part deals with the fork phase, where we go wide; the next part will explain the join phase, where we merge the hundreds of streams back into one. But first, I have a few more words to say about rasterization, because what I just told you about there being just one stream of quads coming in isn't quite true.

Going wide during rasterization

To my defense, what I told you *used* to be true for quite a long time, but it's a serial part of the pipeline, and once you throw in excess of 300 shader units at a problem, serial parts of the pipeline have the tendency to become bottlenecks. So GPU architects started using multiple rasterizers; as of 2010, **NVidia employs four rasterizers (http://www.highperformancegraphics.org/previous/www_2010/media/Hot3D/HPG2010_Hot3D_NVIDIA.pdf)** and **AMD uses two (http://www.highperformancegraphics.org/previous/www_2010/media/Hot3D/HPG2010_Hot3D_AMD.pdf)**. As a side note, the NV presentation also has a few notes on the requirement to keep stuff in API order. In particular, you really do need to sort primitives back into order prior to rasterization/early-Z, like I mentioned last time; doing it just before alpha blend (as you might be inclined to do) doesn't work.

The work distribution between rasterizers is based on the tiles we've already seen for early-Z and coarse rasterization. The frame buffer is divided into tile-sized regions, and each region is assigned to one of the rasterizers. After setup, the bounding box of the triangle is consulted to find out which triangles to hand over to which rasterizers; large triangles will always be sent to all rasterizers, but smaller ones can hit as little as one tile and will only be sent to the rasterizer that owns it.

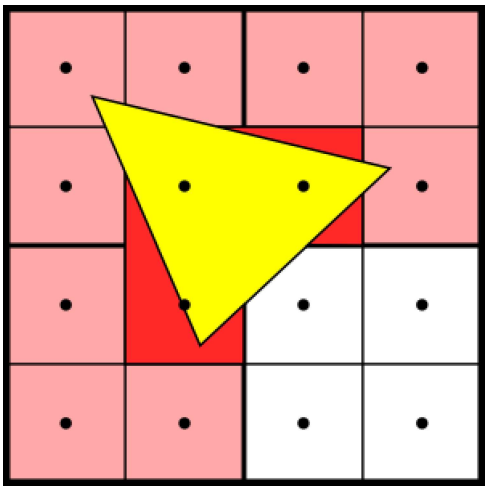
The beauty of this scheme is that it only requires changes to the work distribution and the coarse rasterizers (which traverse tiles); everything that only sees individual tiles or quads (that is, the pipeline from hierarchical Z down) doesn't need to be modified. The problem is that you're now dividing jobs based on screen locations; this can lead to a severe load imbalance between the rasterizers (think a few hundred tiny triangles all inside a single tile) that you can't really do anything about. But the nice thing is that everything that adds ordering constraints to the pipeline (Z-test/write order, blend order) comes attached to specific frame-buffer locations, so screen-space subdivision works without breaking API order – if this wasn't the case, tiled renderers wouldn't work.

You need to go wider!

Okay, so we don't get just one linear stream of quad coordinates plus coverage masks in, but between two and four. We still need to farm them out to hundreds of shader units. It's time for another dispatch unit! Which first means another buffer. But how big are the batches we send off to the shaders? Here I go with NVidia figures again, simply because they mention this number in **public white papers (http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf)**; AMD probably also states that information somewhere, but I'm not familiar with their terminology for it so I couldn't do a direct search

for it. Anyway, for NVidia, the unit of dispatch to shader units is 32 threads, which they call a "Warp". Each quad has 4 pixels (each of which in turn can be handled as one thread), so for each shading batch we issue, we need to grab 8 incoming quads from the rasterizer before we can send off a batch to the shader units (we might send less in case there's a shader switch or pipeline flush).

Also, this is a good point to explain why we're dealing with quads of 2×2 pixels and not individual pixels. The big reason is derivatives. Texture samplers depend on screen-space derivatives of texture coordinates to do their mip-map selection and filtering (as we saw back in **part 4 (https://fgiesen.wordpress.com/2011/07/04/a-trip-through-the-graphics-pipeline-2011-part-4/)**); and, as of shader model 3.0 and later, the same machinery is directly available to pixel shaders in the form of derivative instructions. In a quad, each pixel has both a horizontal and vertical neighbor within the same quad; this can be used to estimate the derivatives of parameters in the x and y directions using **finite differencing (http://en.wikipedia.org/wiki/Finite_difference)** (it boils down to a few subtractions). This gives you a very cheap way to get derivatives at the cost of always having to shade groups of 2×2 pixels at once. This is no problem in the interior of large triangles, but means that between 25-75% of the shading work for quads generated for triangle edges is wasted. That's because all pixels in a quad, even the masked ones, get shaded. This is necessary to produce correct derivatives for the pixels in the quad that are visible. The invisible but still-shaded pixels are called "helper pixels". Here's an illustration for a small triangle:



(https://fgiesen.files.wordpress.com/2011/07/quad_coverage.png)

The triangle intersects 4 quads, but only generates visible pixels in 3 of them. Furthermore, in each of the 3 quads, only one pixel is actually covered (the sampling points for each pixel region are depicted as black circles) – the pixels that are filled are depicted in red. The remaining pixels in each partially-covered quad are helper pixels, and drawn with a lighter color. This illustration should make it clear that for small triangles, a large fraction of the total number of pixels shaded are helper pixels, which has attracted some **research attention (http://graphics.stanford.edu/papers/fragmerging/shade_sig10.pdf)** on how to merge quads of adjacent triangles. However, while clever, such optimizations are not permissible by current API rules, and current hardware doesn't do them. Of course, if the HW vendors at some point decide that wasted shading work on quads is a significant enough problem to force the issue, this will likely change.

Attribute interpolation

Another unique feature of pixel shaders is attribute interpolation – all other shader types, both the ones we've seen so far (VS) and the ones we're still to talk about (GS, HS, DS, CS) get inputs directly from a preceding shader stage or memory, but pixel shaders have an additional interpolation step in front of them. I've already talked a bit about this in the **previous part (https://fgiesen.wordpress.com/2011/07/08/a-trip-through-the-graphics-pipeline-2011-part-7/)** when discussing Z, which was the first interpolated attribute we saw.

Other interpolated attributes work much the same way; a plane equation for them is computed during triangle setup (GPUs may choose to defer this computation somewhat, e.g. until it's known that at least one tile of the triangle passed the hierarchical Z-test, but that shall not concern us here), and then during pixel shading, there's a separate unit that performs attribute interpolation using the pixel positions of the quads and the plane equations we just computed.

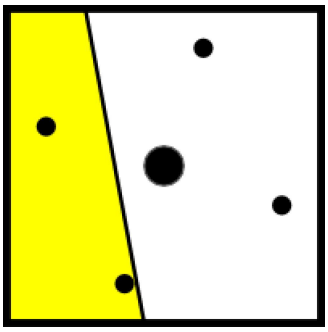
Update: Marco Salvi points out (in the comments below) that while there used to be dedicated interpolators, by now the trend is towards just having them return the barycentric coordinates to plug into the plane equations. The actual evaluation (two multiply-adds per attribute) can be done in the shader unit.

All of this shouldn't be surprising, but there's a few extra interpolation types to discuss. First, there's "constant" interpolators, which are (surprise!) constant across the primitive and take the value for each vertex attribute from the "leading vertex" (which vertex that is is determined during primitive setup). Hardware may either have a fast-path for this or just set up a corresponding plane equation; either way works fine.

Then there's no-perspective interpolation. This will usually set up the plane equations differently; the plane equations for perspective-correct interpolation are set up either for X, Y-based interpolation by dividing the attribute values at each vertex by the corresponding w, or for barycentric interpolation by building the triangle edge vectors. Non-perspective interpolated attributes, however, are cheapest to evaluate when their plane equation is set up for X, Y-based interpolation *without* dividing the values at each vertex by the corresponding w.

"Centroid" interpolation is tricky

Next, we have "centroid" interpolation. This is a flag, not a separate mode; it can be combined both with the perspective and no-perspective modes (but not with constant interpolation, because it would be pointless). It's also terribly named and a no-op unless multisampling is enabled. With multisampling on, it's a somewhat hacky solution to a real problem. The issue is that with multisampling, we're evaluating triangle *coverage* at multiple sample points in the rasterizer, but we're only doing the actual *shading* once per pixel. Attributes such as texture coordinates will be interpolated at the pixel center position, as if the whole pixel was covered by the primitive. This can lead to problems in situations such as this:



(https://fgiesen.files.wordpress.com/2011/07/msaa_samples.png)

Here, we have a pixel that's partially covered by a primitive; the four small circles depict the 4 sampling points (this is the default 4x MSAA pattern) while the big circle in the middle depicts the pixel center. Note that the big circle is outside the primitive, and any "interpolated" value for it will actually be linear extrapolation; this is a problem if the app uses texture atlases, for example. Depending on the triangle size, the value at the pixel center can be very far off indeed. Centroid sampling solves this problem. The original explanation was that the GPU takes all of the samples covered by the primitive, computes their centroid, and samples at that position (hence the name). This is usually followed by the addition that this is just a conceptual model, and GPUs are free to do it differently, so long as the point they pick for sampling is within the primitive.

If you think it somewhat unlikely that the hardware actually counts the covered samples, sums them up, then divides by the count, then join the club. Here's what *actually* happens:

> If all sample points cover the primitive, interpolation is done as usual, i.e. at the pixel center (which happens to be the centroid of all sample positions for all reasonable sampling patterns).
> If not all sample points cover the triangle, the hardware picks one of the sample points that do and evaluates there. All covered sample points are (by definition) inside the primitive so this works.

That picking used to be arbitrary (i.e. left to the hardware); I believe by now DX11 actually prescribes exactly how it's done, but this more a matter of getting consistent results between different pieces of hardware than it is something that API users will actually care about. As said, it's a bit hacky. It also tends to mess up derivative calculations for quads that have partially covered pixels – tough luck. What can I say, it may be industrial-strength duct tape, but it's still duct tape.

Finally (new in DX11!) there's "pull-model" attribute interpolation. Regular attribute interpolation is done automatically before the pixel shader starts; pull-model interpolation adds actual instructions that do the interpolation to the pixel shader. This allows the shader to compute its own position to sample values at, or to only interpolate attributes in some branches but not in others. What it boils down to is the pixel shader being able to send additional requests to the interpolation unit while the shader is running.

The actual shader body

Again, the general shader principles are well-explained in the API documentation, so I'm not going to talk about how individual instructions work; generally, the answer is "as you would expect them to". There are however some interesting bits about pixel shader execution that are worth talking about.

The first one is: texture sampling! Wait, didn't I wax on about texture samplers for quite some time in part 4 already? Yes, but that was the texture sampler side of things – and if you remember, there was that one bit about texture cache misses being so frequent that samplers are usually designed to sustain at least one miss to main memory per request (which is 16-32 pixels, remember!) without stalling. That's a lot of cycles – hundreds of them. And it would be a tremendous waste of perfectly good ALUs to keep them idle while all this is going on.

So what shader units actually do is switch to a different batch after they've issued a texture sample; then when that batch issues a texture sample (or completes), it switches back to one of the previous batches and checks if the texture samples are there yet. As long as each shader unit has a few batches it can work on at any given time, this makes good use of available resources. It does increase latency for completion of individual batches though – again, a latency-vs-throughput trade-off. By now you should know which side wins on GPUs: Throughput! *Always*. One thing to note here is that keeping multiple batches (or "Warps" on NVidia hardware, or "Wavefronts" for AMD) running at the same time requires more registers. If a shader needs a lot of registers, a shader unit can keep less warps around; and if there are less of them, the chance that at some point you'll run out of runnable batches that aren't waiting on texture results is higher. If there's no runnable batches, you're out of luck and have to stall until one of them gets its results back. That's unfortunate, but there's limited hardware resources for this kind of thing – if you're out of memory, you're out of memory, period.

Another point I haven't talked about yet: Dynamic branches in shaders (i.e. loops and conditionals). In shader units, work on all elements of each batch usually proceeds in lockstep. All "threads" run the same code, at the same time. That means that ifs are a bit tricky: If *any* of the threads want to execute the "then"-branch of an if, all of them have to – even though most of them may end up ignoring the results using a technique called **predication (http://en.wikipedia.org/wiki/Branch_predication)**, because they didn't want to descend down there in the first place.. Similarly for the "else" branch. This works great if conditionals tend to be coherent across elements, and not so great if they're more or less random. Worst case, you'll always execute both branches of every if. Ouch. Loops work similarly – as long as at least one thread wants to keep running a loop, all of the threads in that batch/Warp/Wavefront will.

Another pixel shader specific is the `discard` instruction. A pixel shader can decide to "kill" the current pixel, which means it won't get written. Again, if all pixels inside a batch get discarded, the shader unit can stop and go to another batch; but if there's at least one thread left standing, the rest will be dragged along. DX11 adds more fine-grained control here by way of writing the output pixel coverage from the pixel shader (this is always ANDed with the original triangle/Z-test coverage, to make sure that a shader can't write outside its primitive, for sanity). This allows the shader to discard individual samples instead of whole pixels; it can be used to implement Alpha-to-Coverage with a custom dithering algorithm in the shader, for example.

Pixel shaders can also write the output depth (this feature has been around for quite some time now). In my experience, this is an excellent way to shoot down early-Z, hierarchical Z and Z compression and in general get the slowest path possible. By now, you know enough about how these things work to see why. :)

Pixel shaders produce several outputs – in general, one 4-component vector for each render target, of which there can be (currently) up to 8. The shader then sends the results on down the pipeline towards what D3D calls the "Output Merger". This'll be our topic next time.

But before I sign off, there's one final thing that pixel shaders can do starting with D3D11: they can write to Unordered Access Views (UAVs) – something which only compute and pixel shaders can do. Generally speaking, UAVs take the place of render targets during compute shader execution; but unlike render targets, the shader can determine the position to write to itself, and there's no implicit API order guarantee (hence the "unordered access" part of the name). For now, I'll only mention that this functionality exists – I'll talk more about it when I get to Compute Shaders.

Update: In the comments, Steve gave me a heads-up about the correct AMD terminology (the first version of the post didn't have the "Wavefronts" name because I couldn't remember it) and also posted a link to **this great presentation by Kayvon Fatahalian (http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf)** that explains shader execution on GPUs, with a lot more pretty pictures that I can be bothered to make :). You should really check it out if you're interested in how shader cores work.

And… that's it! No big list of caveats this time. If there's something missing here, it's because I've genuinely forgotten about it, not because I decided it was too arcane or specific to write up here. Feel free to point out omissions in the comments and I'll see what I can do.

From → Coding, Graphics Pipeline

19 Comments

1. **Marco Salvi permalink**
   I am really enjoying your series. You should collect it in a single document once it's complete.

   A minor note on attributes interpolation:

   ".. then during pixel shading, there's a separate unit that performs attribute interpolation using the pixel positions of the quads and the plane equations we just computed."

   This used to be true. These days you are mostly likely to find a separate unit that "simply" generates barycentric coordinates (according the type of interpolation requested), while the rest of the per-attribute calculation is performed on the shader cores.

   Reply
   - **fgiesen permalink**
     "I am really enjoying your series. You should collect it in a single document once it's complete."
     Thanks! Once I'm done I'll at least go over the text again, revise it a bit and probably add a few more illustrations. I may collect it as a single document – not decided yet.

     "These days you are mostly likely to find a separate unit that "simply" generates barycentric coordinates (according the type of interpolation requested), while the rest of the per-attribute calculation is performed on the shader cores."
     Ah. I know that the full-blown interpolators *used* to be there, but it's generally not something that gets much attention in official presentations, so it's hard to pinpoint when exactly they switch :)

     Makes sense, though. I suspect that everything that involves fused multiply-adds will end up getting moved into the shader units, sooner or later. Especially as compute to texture ratio keeps going up steadily, there's no reason not to.

     Reply
2. **Rex Guo permalink**
   Thanks for another great write-up, ryg!

   While on the topic of pixel/fragment shaders, could you give some guidance on the ideal ratio of compute/memory instruction mix? Is it still ~20:1 ? Does NV also publish the cycle count for the math ops?

   Reply
   - **fgiesen permalink**
     "could you give some guidance on the ideal ratio of compute/memory instruction mix"
     That kind of information is outside the scope of this series; it depends too much on the underlying hardware. That said, as you get more transistors with lower power consumption, it's easier to add compute power than it is to add (DRAM) memory bandwidth. And memory latency is worse – lowering latency without harming bandwidth of power consumption is *seriously hard*.

     NV does publish the latency of their math ops – it's in the same document I got the memory access latency from (on the same page, too!). They use fully pipelined ALUs with very high latencies; 18+ cycles per operation. This means that they do need to switch warps every cycle – you need 18 warps running concurrently on a shader unit to not ever hit instruction dependency stalls, though if the shaders have several independent dependency chains, that number goes down. If you compare the ~20 cycles for ALU ops to the ~400 cycles for memory access, you can see where the 20:1 ratio comes from – while a warp is waiting on a texture fetch, it temporarily drops out of the rotation, so you need correspondingly more runnable warps to fill the gap.

     Reply
3. **Steve permalink**

Excellent posts, thanks!

> “Warps”, or whatever the AMD terminology is

AFAIK AMD calls them “wavefronts” (“Running Code at a Teraflop: How a GPU Shader Core Works”, http://bps10.idav.ucdavis.edu/ – a great companion course for this stuff)

Reply

- **fgiesen permalink**
  Ah, Kayvons presentation is one I’ve been looking for, but I couldn’t remember the title or the name of the author, which makes it hard to find :). I’ll update the article accordingly, thanks a bunch!

  Reply

4. **Barbie permalink**
   Do you have any insight about the derivative computation based on helper pixels (outside of the triangle) ? I’ve never really looked into the artifacts it can trigger, but I assume e.g. computing texture coordinates based on a previous texture lookup can end up looking really weird if the texture lookup ended up wrapping around.

   Reply

   - **fgiesen permalink**
     Extrapolation for helper pixels doesn’t cause problems with linear attributes: if you think of it in world space, the texture coordinates are interpolated linearly across the triangle, so their world-space derivatives are constants. In screen-space (post-projection), there’s an added perspective distortion, which is a projective transform that takes the plane of the triangle to the plane the screen quad lies in (i.e. viewing plane). Since it’s a transform between planes, it’s well-defined even outside the triangle, and approximating the derivatives with screen-space finite differences isn’t any more (or less) correct outside the tri than it is inside of it.

     If your quantities are not linear, or cross a seam, then yes, that causes visible artifacts. Usually you get some 2×2 quads that are a lot blurrier than others (because some quads see artificially high derivatives, which makes them pick small mip levels). Shader authors by now are usually aware of the problem, but if you look for this in games, you’ll find it :). Early- to mid-2000s shaders for things like Environment-Mapped Bump Mapping have an especially bad case of this (because they were written for 1.x or 2.x shader HW which doesn’t give you any means of specifying gradients directly – even if you had enough instruction slots to compute them…)

     Reply

5. **Egor Yusov permalink**
   “In a quad, each pixel has both a horizontal and vertical neighbor within the same quad; this can be used to estimate the derivatives of parameters in the x and y directions using finite differencing (it boils down to a few subtractions)”
   Does this mean that for instance both pixels in each row get the same horizontal derivative?

   It seems like computing just one difference should not be a problem for GPU, while different sources tell us that computing derivatives is quite expensive and suggest not computing them in a shader, when possible. There are even special instructions that compute coarse derivatives. Do you know how these instructions work?

   Reply

   - **fgiesen permalink**
     Yes. The x deltas and y deltas are the same for the 2 pixels in a row or column, respectively. And this is with “fine” derivatives – with “coarse” derivatives, both x and y derivatives are the same for all pixels in a quad.

     Microsoft have updated the online D3D11 docs to include the instruction specs, so here they are for reference:
     Fine derivatives
     Coarse derivatives

     The computation itself is cheap, but the necessary shuffling to get the difference between two values computed in different threads usually has some extra cost. While this might make it more expensive than other ALU instructions, it’s not particularly bad, and certainly much cheaper than e.g. texture fetches.

     While the name might suggest otherwise, there should be no appreciable speed difference between computing coarse and fine derivatives.

     Reply

     - **fgiesen permalink**
       For what it’s worth, on implementation of derivative instructions in hardware: By now there’s specs for both recent AMD and NV ISAs (or at least a reasonably close approximation thereof) online, so you can just look it up.

In NVidias PTX 3.1 ISA, you could implement a coarse x derivative like this: (hope I got this right – this is the right sequence of operations, but I might have screwed up the masks)

```
shfl.idx.b32  Rtl, Rsrc, 0x00, 0x1c00;
shfl.idx.b32  Rtr, Rsrc, 0x01, 0x1c00;
sub.f32       Rdsrc_dx, Rtr, Rtl;
```

Coarse y:

```
shfl.idx.b32  Rtl, Rsrc, 0x00, 0x1c00;
shfl.idx.b32  Rbl, Rsrc, 0x02, 0x1c00;
sub.f32       Rdsrc_dy, Rbl, Rtl;
```

Fine x:

```
    shfl.bfly.b32 Ropposite|p, Rsrc, 0x01, 0x1e00;
@p  sub.f32       Rdsrc_dx, Ropposite, Rsrc;
@!p sub.f32       Rdsrc_dx, Rsrc, Ropposite;
```

Fine y:

```
    shfl.bfly.b32 Ropposite|p, Rsrc, 0x02, 0x1d00;
@p  sub.f32       Rdsrc_dy, Ropposite, Rsrc;
@!p sub.f32       Rdsrc_dy, Rsrc, Ropposite;
```

With AMD's Southern Islands ISA (used for their recent GCN cores), you can perform the shuffling using the `DS_SWIZZLE_B32` instruction and the subtractions are simple. On both architectures works out to roughly three times the cost of a regular ALU operation (add, sub, mad, etc.). Not free, but not particularly expensive either.

6. **Niad permalink**
   Hi thank you for creating this series!

   I was wondering about the quad overshading.

   If the derivatives are only required for texturing and calls to ddx/ddy, why not disable it if a shader does not use these features?

   Are derivatives required for anything else? Does current hardware disable the quads if they aren't used?

   This might seem random, but I have a project that works exactly like this, no texturing, very small triangles. It seems be bottlenecked by this issue.

   Reply
   - **fgiesen permalink**
     All current 3D hardware that I know of uses quads always. It is not something you can just disable.

     This is not a software/driver thing; quad granularity is part of the design of most fixed-function blocks in the pipeline that are used at the per-pixel level. the rasterizer determines coverage in terms of quads, not pixels. Depth and stencil testing is done at quad (or larger) granularity at a time. Pixel/fragment shading is done on groups of pixels at a time (usually between 16 and 64), and the hardware that determines these groups works on quads, not individual pixels. Attributes are interpolated based on the triangle that a fragment came from. With quad-based shading and 64-wide wavefronts, that means that a wavefront can reference up to 16 unique triangles; without quad-based shading, it would be 64. There needs to be storage for the plane equations of attributes, and hardware to set up these plane equations for interpolation per triangle from vertex attributes before the shader runs. Getting rid of quads would mean you would need 4x the amount of storage for plane equations (to handle the worst case), 4x more (or 4x faster) hardware to set up these attributes, and so forth. And once shading is done, blending and writing to memory is usually done on a quad basis as well.

     It's not that it's impossible to design HW that doesn't use quads; but supporting true pixel-granularity shading essentially boils down to re-designing a bunch of hardware blocks to have one or more of: 4x the clock rate, 4x the operation width, 4x the amount of routing/control logic, 4x the amount of buffer space. It's not something you just "enable" – it's a major

change, costly, power-hungry, it doesn't benefit workloads that GPU vendors (currently) care about, and is thus unlikely to get implemented any time soon.

Reply

7. **nlguillemot permalink**
   minor typo: "With multisampling ob" should be "With multisampling on", I assume.

   Reply
   - **fgiesen permalink**
     Thanks, fixed!

     Reply

Trackbacks & Pingbacks

1. A trip through the Graphics Pipeline 2011: Index « The ryg blog
2. Viaje alucinante por un pipeline grafico « martin b.r.
3. A trip through the Graphics Pipeline 2011, part 13 « The ryg blog
4. Optimizing the basic rasterizer « The ryg blog