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

A trip through the Graphics Pipeline 2011, part 5

July 5, 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/)**.*

After the last post about texture samplers, we're now back in the 3D frontend. We're done with vertex shading, so now we can start actually rendering stuff, right? Well, not quite. You see, there's a bunch still left to do before we actually start rasterizing primitives. So much so in fact that we're not going to see any rasterization in this post – that'll have to wait until next time.

Primitive Assembly

When we left the vertex pipeline, we had just gotten a block of shaded vertices back from the shader units, with the implicit promise that this block contains an integral number of primitives – i.e., we don't allow triangles, lines or patches to be split across multiple blocks. This is important, because it means we can truly treat each block independently and never need to buffer more than one block of shader output – we can, of course, but we don't have to.

The next step is to assemble all the vertices belonging to a single primitive (hence "primitive assembly"). If that primitive happens to be a point, this just reads exactly one vertex and passes it on. If it's lines, it reads two vertices. If it's triangles, three. And so on for patches with larger numbers of control points.

In short, all that happens here is that we gather vertices. We can either do this by reading the original index buffer and keeping a copy of our vertex index->cache position map around (as I described), or we can store the indices for the fully expanded primitives along with the shaded vertex data, which might take a bit more space for the output buffer but means we don't have to read the indices again here. Either way works fine.

And now we have expanded out all the vertices that make up a primitive. In other words, we now have complete triangles, not just a bunch of vertices. So can we rasterize them already? Not quite.

Viewport culling and clipping

Oh yeah, that. Yeah, I guess we'd better do that first, huh? This is one part of pipeline that really does exactly what you'd expect, pretty much the way you would expect it too (i.e. the way it's explained in the docs). So I'm not gonna explain polygon clipping in general here, you can look that up in any computer graphics textbook, although most make a terrible mess of it; if you want a good explanation, use Jim Blinn's (chapter 13 of **this book (http://www.amazon.com/Jim-Blinns-Corner-Graphics-Pipeline/dp/1558603875)**), although you probably want to pass on his alternative [0,w] clip space these days, to avoid confusion if nothing else.

Anyway, clipping. The short version is this: Your vertex shader returns vertex positions on homogeneous clip space. Clip space is chosen to make the equations that describe the view frustum as simple as possible; in the case of D3D, they are $-w \leq x \leq w$, $-w \leq y \leq w$, $0 \leq z \leq w$, and $0 < w$; note that all the last equation really does is exclude the homogeneous point (0,0,0,0), which is something of a degenerate case.

We first need to find out if the triangle is partially or even completely outside any of these clip planes. This can be done very efficiently using **Cohen-Sutherland (http://en.wikipedia.org/wiki/Cohen%E2%80%93Sutherland)**-style out-codes. You compute the clip out-code (or just clip-code) for each vertex (this can be done at vertex shading time and stored along with the positions, for example). Then, for each primitive, the bitwise AND of the clip-codes will tell you all the view-frustum planes that *all* vertices in the primitive are on the wrong side of (if there's any, that means the primitive is completely outside the view frustum and can be thrown away), and the bitwise OR of the clip-codes will tell you the planes that you need to clip the primitive against. Given the clipcodes, all this is just a few gates worth of hardware – simple stuff.

Additionally, the shaders can also generate a set of "cull distances" (a triangle will be discarded if any one cull distance for all vertices is less than zero), and a set of "clip distances" (which define additional clipping planes). These get considered for primitive rejection/clip testing too.

The actual clipping process, if invoked, can take one of two forms: we can either use an actual polygon clipping algorithm (which adds extra vertices and triangles), or we can add the clipping planes as extra edge equations to the rasterizer (if that sounds like gibberish to you, wait until the next part where I explain rasterization – it'll ask make sense eventually). The latter is more elegant and doesn't require an actual polygon clipper at all, but we need to be able to handle all normalized 32-bit floating point values as valid vertex coordinates; there might be a trick for building a fast HW rasterizer that does this, but it seems tricky to say the least. So I'm assuming there's an actual clipper, with all that involves (generation of extra triangles etc). This is a nuisance, but it's also very infrequent (more so than you think, I'll get to that in a second), so it's not a big deal. Not sure if that's special hardware either, or if that path grabs a shader unit to do the actual clipping; depends on whether dispatching a new vertex shading load at this stage is awkward or not, how big a dedicated clipping unit is, and how many of them you need. I don't know the answer to these questions, but at least from the performance side of things, it doesn't much matter: we don't really clip that often. That's because we can use guard-band clipping.

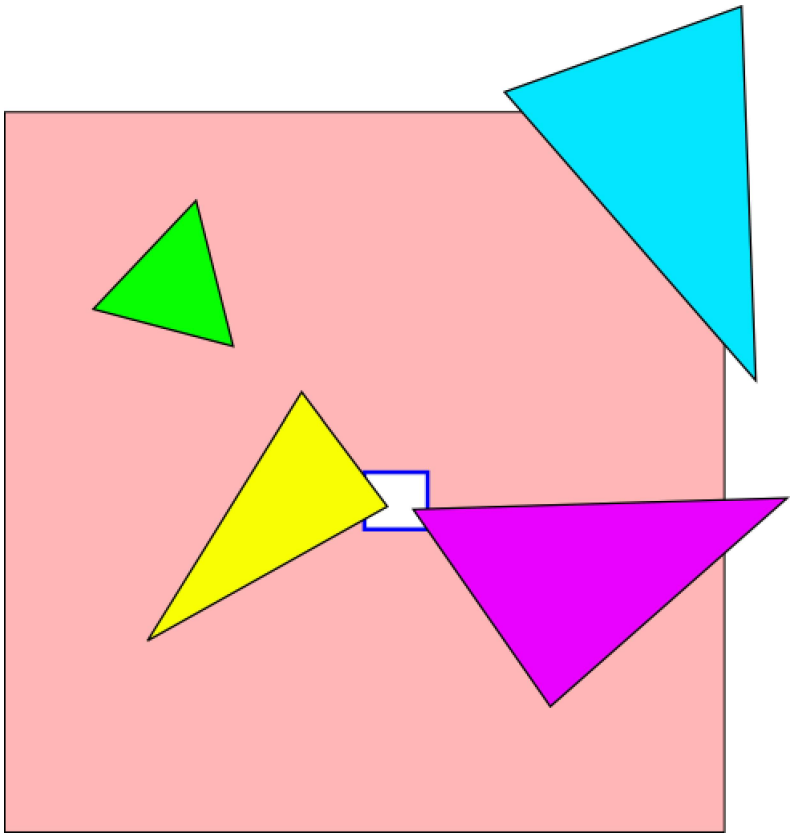
Guard-band clipping

The name is something of a misnomer; it's not a fancy way of doing clipping. In fact, it's quite the opposite: a straight-forward way of not doing clipping. :)

The underlying idea is very simple: Most primitives that are partially outside the left, right, top and bottom clip planes don't need to be clipped at all. Triangle rasterization on GPUs works by, in effect, scanning over the full screen area (or more precisely, the scissor rect) and asking for every pixel: "is this pixel covered by the current triangle?" (In reality it's a bit more complicated and way more efficient than that, but that's the general idea). And that works just as well for triangles completely within the viewport as it does for triangles that extend past, say, the right and top clipping planes. As long as our triangle coverage test is reliable, we don't need to clip against the left, right, top and bottom planes at all!

That test is usually done in integer arithmetic with some fixed precision. And eventually, as you move say one triangle vertex further and further out, you'll get integer overflows and wrong test results. I think we can all agree that the rasterizer producing pixels that aren't actually inside the triangle is, at the very least, extremely offensive behavior and should be illegal! Which it in fact is – hardware that does this is in violation of the spec.

There's two solutions for this problem: The first is to make sure that your triangle tests never, ever generate the wrong results, no matter how your input triangle looks. If you manage that, then you don't ever need to clip against the aforementioned four planes. This is called "infinite guard-band" because, well, the guard-band is effectively infinite. Solution two is to clip triangles eventually, just as they're about to go outside the safe range where the rasterizer calculations can't overflow. For example, say that your rasterizer has enough internal bits to deal with integer triangle coordinates that have $-32768 \leq X \leq 32767$, $-32768 \leq Y \leq 32767$ (note I'm using capital X and Y to denote screen-space positions; I'll stick with this convention). You still do your viewport cull test (i.e. "is this triangle outside the view frustum") with the regular view planes, but only actually clip against the guard-band clip planes which are chosen so that after the projection and viewport transforms, the resulting coordinates are in the safe range. I guess it's time for an image:



(https://fgiesen.files.wordpress.com/2011/07/guardband_clip.png)
Guard-band clipping

The small white rectangle with blue outline that's roughly in the middle represents our viewport, while the big salmon-colored area around it is our guard band. It looks like a small viewport in this image, but I actually picked a huge one so you can see anything! With our -32768 .. 32767 guard-band clip range, that viewport would be about 5500 pixels wide – yes, that's some huge triangles right there :). Anyway, the triangles show off some of the important cases. The yellow triangle is the most common case – a triangle that extends outside the viewport but not the guard band. This just gets passed straight through, no further processing necessary. The green triangle is fully within the guard band, but outside the viewport region, so it would never get here – it's been rejected above by the viewport cull. The blue triangle extends outside the guard-band clip region and would need to be clipped, but again it's fully outside the viewport region and gets rejected by the viewport cull. Finally, the purple triangle extends both inside the viewport and outside the guard band, and so actually needs to be clipped.

As you can see, the kinds of triangles you need to actually have to clip against the four side planes are pretty extreme. As said, it's infrequent – don't worry about it.

Aside: Getting clipping right

None of this should be terribly surprising; nor should it sound too difficult, at least if you're familiar with the algorithms. But the devil's in the details, always. Here's some of the non-obvious rules the triangle clipper has to obey in practice. If it ever breaks *any* of these rules, there's cases where it will produce cracks between adjacent triangles that share an edge. This isn't allowed.

Vertex positions that are inside the view frustum must be preserved, bit-exact, by the clipper.
Clipping an edge AB against a plane must produce the same results, bit-exact, as clipping the edge BA (orientation reversed) against that plane. (This can be ensured by either making the math completely symmetric, or always clipping an edge in the same direction, say from the outside in).
Primitives that are clipped against multiple planes must always clip against planes in the same order. (Either that or clip against all planes at once)
If you use a guard band, you *must* clip against the guard band planes; you can't use a guard band for some triangles but then clip against the original viewport planes if you actually need to clip. Again, failing to do this will cause cracks – and if I remember correctly there was actually a piece of graphics hardware in the bad old days that shipped with this bug enshrined in silicon. Oops. :)

Those pesky near and far planes

Okay, so we have a really nice quick solution for the 4 side planes, but what about near and far? Particularly the near plane is bothersome, since with all the stuff that's only slightly outside the viewport handled, that's the plane we do most of our clipping for. So what can we do? A z guard band? But how would that work – we're not actually rasterizing along the z axis at all! In fact, it's just some value we interpolate over the triangle, damn!

On the plus side, though, it's just some value we interpolate over the triangle. And in fact the z-near test ($Z < 0$) is *really easy* to do once you interpolate Z – it's just the sign bit. z-far ($Z > 1$) is an extra compare though (not I'm using Z not z here, i.e. these are "screen" or post-projection coordinates). But still, we're doing Z-compares per pixel anyway (Z test!), so it's not a big extra expense. It depends, but doing z-clip this way is definitely an option. And you need to be able to skip z-near/z-far clipping if you want to support things like NVidias 'depth clamp' OpenGL extension; in fact, I would argue the existence of that extension is a pretty good hint that they're doing this, or at least used to for a while.

So we're down to one of the regular clip planes: $0 < w$. Can we get rid of this one too? The answer is yes, with a rasterization algorithm that works in homogeneous coordinates, e.g. **this one (http://www.cs.unc.edu/~olano/papers/2dh-tri/)**. I'm not sure whether hardware uses that one though. It's nice an elegant, but it seems like it would be hard to obey the (very strict!) D3D11 rasterization rules to the letter using that algorithm. But maybe there's some cool tricks that I'm not aware of. Anyway, that's about it with clipping.

Projection and viewport transform

Projection just takes the x, y and z coordinates and divides them by w (unless you're using a homogeneous rasterizer which doesn't actually project – but I'll ignore that possibility in the following). This gives us normalized device coordinates, or NDCs, between -1 and 1. We then apply the viewport transform which maps the projected x and y to pixel coordinates (which I'll call X and Y) and the projected z into the range [0,1] (I'll call this value Z), such that at the z-near plane Z=0 and at the z-far plane Z=1.

At this point, we also snap pixels to fractional coordinates on the sub-pixel grid. As of D3D11, hardware is required to have exactly 8 bits of subpixel precision for triangle coordinates. This snapping turns some *very* thin slivers (which would otherwise cause problems) into degenerate triangles (which don't need to be rendered at all).

Back-face and other triangle culling

Once we have X and Y for all vertices, we can calculate the signed triangle area using a cross product of the edge vectors. If the area is negative, the triangle is wound counter-clockwise (here, negative areas correspond to counter-clockwise because we're now in the pixel coordinate space, and in D3D pixel space y increases downwards not upwards, so signs are inverted). If the area is positive, it's wound clockwise. If it's zero, it's degenerate and doesn't cover any pixels, so it can be safely culled. At this point, we know the triangle orientation so we can do back-face culling (if enabled).

And that's it! We're now ready for rasterization… almost. Actually we have to do triangle setup first. But doing that requires some knowledge of how rasterization will be performed, so I'll put that off until the next part… see you then!

Final remarks

Again, I skipped some parts and simplified others, so here's the usual reminder that things are a bit more complicated in reality: For example, I pretended that you just use the regular homogeneous clipping algorithm. Mostly, you do – but you can have some vertex shader attributes flagged as using screen-space linear instead of perspective-correct interpolation. Now, the regular homogeneous clip always does perspective-correct interpolation; in the case of screen-space linear attributes, you actually need to do some extra work to make it not perspective-correct. :)

I talk about primitives some of the time, but mostly I'm just focusing on triangles here. Points and lines aren't hard, but let's be honest, they're not what we're here for either. You can work out the details if you're interested. :)

There's tons of rasterization algorithms out there, some of which (like Olanos 2DH method that I cited) allow you to skip nearly all clipping, but as I mentioned, D3D11 has very strict requirements on the triangle rasterizer so there's not much wiggle room for HW implementations; I'm not sure if those methods can be tweaked to exactly follow the spec (there's a lot of subtle points that I'll cover next time). So here and in the following I'm assuming you can't do the ultra-sleek thing; then again, the not-quite-so-sleek approaches I'm running with have slightly less math per pixel in the rasterizer, so they might win for HW implementations anyway. And of course I might be missing the magic pixie dust right around the corner that solves all of these problems. That occurs surprisingly often in graphics. If you know an awesome solution, give me a shout in the comments!

Lastly, the triangle culling I'm describing here is the bare minimum; for example, the class of triangles that will generate zero pixels upon rasterization is much larger than just zero-area tris, and if you can find it out quickly enough (or with few enough gates), you can drop the triangle immediately and don't need to go through triangle setup. This is the last point where you can cull cheaply before going through triangle setup and at least some rasterization – finding other ways to early-reject tris pays off handsomely here.

From → Coding, Graphics Pipeline

17 Comments

1. **Ben permalink**
   Ahh.. those 4 clipper requirements warm my heart!

   I stumbled on those 4 exactly (except the first one, which was obvious before I started) when writing a rasterizer a couple years back.

   Anyway – great series. Learning a lot.
   Thanks

   Reply
2. **Jocelyn Houle permalink**
   "Again, failing to do this will cause cracks – and if I remember correctly there was actually a piece of graphics hardware in the bad old days that shipped with this bug enshrined in silicon. Oops. :)"

   Mmhhhh… Tasty bit of information… What GPU was that?

   "If you know an awesome solution, give me a shout in the comments!"

   Dachsbacher et al. wrote a 3D rasterizer (http://www.vis.uni-stuttgart.de/~dachsbcn/download/3dr_techreport.pdf) that failed to have academia's approval (i.e. only a tech report). But it does sport some interesting concepts that allows to rasterize triangles over arbitrary surfaces, and therefore bridges the gap between rasterization and ray tracing. Haven't heard of any IHV trying to accelerate such path, though… (kind of out of the loop, now) If only for robust ray-casting, this article is quite a gem…

   Reply
   - **fgiesen permalink**
     "What GPU was that?"
     Not naming any names – not because I don't want to offend, I just seriously don't remember :). This was maybe 2000/2001 or so. When 3dfx, S3, Matrox and 3Dlabs were still around in the 3D graphics space, and you really would run into driver/compatibility issues even with totally basic stuff on a regular basis.

     The 3D rasterizer paper looks sweet (as indeed most of Carstens papers do), but same as with Olanos paper, I don't see any obvious way to integrate things like subpixel grid snapping (required for D3D10/11 hardware!) into that kind of approach, especially since it needs to be absolutely, unconditionally robust. Another worry is precision – 32-bit float is definitely not enough to evaluate edge equations will full precision for large triangles; even doubles with their 53-bit mantissa (including hidden 1 bit) are uncomfortably tight. Plus any floating-point implementation needs to recompute the edge equations regularly using a multiply-accumulate style operation to avoid drift; in contrast, a fixed-point rasterizer only ever does integer adds after the initial triangle setup.

     Reply
3. **Kevin permalink**
   Very informative and detailed article :)

   I think you may have forgotten one thing, though. All polygons must be clipped to the near plane by generating new vertices, not by discarding pixels, otherwise you'll run into accuracy problems when one of a triangle's vertices passes behind the view point(which is when it's behind the w=0 plane, I believe).

Reply

- **fgiesen permalink**
  I actually link to Olano's paper that describes a fully clipless rasterizer. You do *not* need to clip, but you do need to add an extra edge function corresponding to the near plane.

  Nvidia used completely clipless rasterizers for quite a while. I'm not sure whether they still do. The D3D11 spec insists on some invariance properties that are quite tricky to obtain in a clipless rasterizer.

  Reply

4. **Sin permalink**
   "As of D3D11, hardware is required to have exactly 8 bits of subpixel precision for triangle coordinates."

   I am not familiar with subpixel precision. I assume it means that there are several subpixels (in case of 8 bits, 256) in a pixel, and is totally unrelated to sub-pixel resolution (https://www.grc.com/ctwhat.htm) ?

   But if that's the case, then what will happen if a triangle occupies some subpixels ? Will the pixel get colored using a certain weight, like with the ratio of occupied subpixels divided by total subpixels in a pixel ?

   Reply

   - **fgiesen permalink**
     No. Sub-pixel accuracy just means that the triangle coordinates aren't snapped to integer pixel positions (which produces very visible artifacts especially with slow movement). 8-bit subpixel precision means that coordinates are rounded to the next 1/256ths of a pixel in both the x and y directions. This has nothing to do with coverage computations or any kind of anti-aliasing (or sub-pixel rendering); I explain a bit more here.

     Reply

     - **Sin permalink**
       Thanks, I read your post and left another comment there.

       I thought a bit after reading your post last time, and I did think that it perhaps had something to do with coverage :p

       So, just to clear out my confusion, this subpixel precision only means that the coordinates aren't snapped to integer positions, which can further be used for things like alpha to coverage and MSAA, for example the technique explained in http://software.intel.com/en-us/articles/rendering-grass-with-instancing-in-directx-10

       Now that I think about it, these features do need non-integer positions. Otherwise, how can they know that a certain triangle occupies a number of samples out of 16 samples (MSAA4x) in a certain pixel.

     - **Sin permalink**
       err, I meant MSAA 16x. Something wrong with my head.

5. **zhebin permalink**
   Great articles, and help me a lot!
   But I have two questions in this topic:
   1.
   How to handle clip space coordinate (0, 0, 0, 0) in the Cohen-Sutherland algorithm?
   Considering the "w > 0" plane, The outcode of (0, 0, 0, 0) will be 1xxxx, but the other two vertices's outcodes may be 0xxxx, so it still needs clipping. However, after clipping, I think we still get the same three vertices – I assume clipping for open interval domain(w > 0) works the same way as closed interval(w >= 0).
   Another idea is that, as you mentioned, this is a degenerated triangle(a line), so we can throw away it. But this is beyond the power of Cohen-Sutherland outcodes and needs special handling.
   Unfortunately, I didn't see such kind of special handling in many open source software rasterizers.

   2.
   "If you use a guard band, you must clip against the guard band planes; you can't use a guard band for some triangles but then clip against the original viewport planes if you actually need to clip. Again, failing to do this will cause cracks."

   Could you explain why clipping to original viewport planes will cause cracks?
   It's difficult for me to imagine a case where cracks will happen, and AFAIK, there are still some software rasterizers doing things like this.

   Reply

   - **fgiesen permalink**
     First question:

Cohen-Sutherland can only clip for inclusive bounds. If you're gonna do a w-clip with Cohen-Sutherland, you end up having to pick an epsilon and clip to "eps ≤ w" instead. That works, but we can do better by looking at what happens with 0. The only clip space "point" the w > 0 rule actually catches is (0,0,0,0)—using quotes here because that's not technically a point as per the definition of homogeneous points (all-0 is explicitly excluded), but we'll run with it. And we can just look at cases involving that point directly, rather than trying to catch them with the clipper. Let's do triangles: a triangle is a convex polygon with 3 vertices and every point inside the triangle can be written as a convex combination of those three vertices

$$\lambda_0 v_0 + \lambda_1 v_1 + \lambda_2 v_2$$

where all the $\lambda_i \geq 0$ and $\lambda_0 + \lambda_1 + \lambda_2 = 1$. This works in a projective space with homogeneous coordinates as well, except we don't need the weights to sum to 1 anymore; homogeneous coordinates are only unique up to scale. So in a projective space, the equivalent of looking at convex combinations is looking at conical combinations – linear combinations where the weights are non-negative.

Now suppose one of the triangle vertices hits (0,0,0,0), say v_2. That means the last term of our conical combination is always zero and the whole expression reduces to

$$\lambda_0 v_0 + \lambda_1 v_1$$

with the $\lambda_i \geq 0$. That's a conical combination of two homogeneous vertices, which corresponds to a line. But a triangle that is actually a line is degenerate and doesn't generate any visible fragments according to our rasterization rules! So long story short, if one of your triangles' vertices is (0,0,0,0) (again, technically not a point), that guarantees the triangle is degenerate and you can drop it, same as any other zero-area triangle. The same argument applies to lines: a line with one of the vertices at (0,0,0,0) is actually degenerate and just a homogeneous point (whether that means it should produce visible fragments or not depends on your choice of line rasterization rules), and a homogeneous point at (0,0,0,0) is not actually a valid point and gets dropped.

Second question:

Consider a quad split into two triangles, with the interior diagonal going from the bottom left to the top right point.

The top right and bottom right points are entirely within the viewport. The bottom left point is *just* outside the left edge of the viewport (but well within the guard band). The top left point is *way* off to the left, outside the guard band.

The bottom right triangle is fully within the guard band and can get rasterized directly. The top left triangle is clipped.

Now if you clip the bottom left vertex to the actual viewport while processing the clipped triangle, the clipped vertex may not end up on exactly the same sub-pixel position, due to round-off. If you have a shared edge between two triangles and you slightly nudge one of its vertices in one triangle but not the other, you get cracks.

There's other ways to avoid this, but *by far* the easiest is to make sure that edges containing the same two vertices will always get handled the same way. In this case, that means you have one set of clip planes (corresponding to the guard band boundaries) that you use for clipping, and you don't switch from the "looser" guard band planes to the "tighter" viewport planes once you discover you actually need to clip.

Reply

- **fgiesen permalink**
  Just to make that clear: the point of the answer to the first question is that you can treat the w>0 clip purely as a symbolical operation. It does not actually have to be implemented as a "proper" plane clip at all, you just need to check for it and reject the primitive if necessary.

- **zhebin permalink**
  First question:
  Thanks! I got your point.
  And I came up with the approach to catch the (0,0,0,0) case with outcodes right after posting this question. That is, (0,0,0,0) is equivalent to the following conditions:
  -w <= x <= w (true)
  -w <= y <= w (true)
  0 <= z 0 (false)

  So if the outcode of one vertex is 0000001b(meaning is shown as follows), we can conclude it is (0,0,0,0) and this triangle can be dropped directly. This method may be not so efficient, but it can be integrated well into the outcodes algorithm.
  near | far | left | right | bottom | top | w

0 0 0 0 0 0 1

Second question:
Yes, this case opens my mind, and I think I need take care of such cases in my own rasterizer implementation. Thank you very much!

- **zhebin permalink**
  Sorry that the typesetting is not very good.
  Correct the typo:
  “-w <= x <= w" (true)
  "-w <= y <= w" (true)
  "0 <= z 0” (false)

6. **xdnoam permalink**
   “The answer is yes, with a rasterization algorithm that works in homogeneous coordinates, e.g. this one.”
   The link is broken, here’s an updated one – https://www.csee.umbc.edu/~olano/papers/2dh-tri/2dh-tri.pdf

   Reply

Trackbacks & Pingbacks

1. A trip through the Graphics Pipeline 2011: Index « The ryg blog
2. Triangle rasterization in practice « The ryg blog