SIGGRAPH2011 VANCOUVER

Filtering Approaches for Real-Time Anti-Aliasing

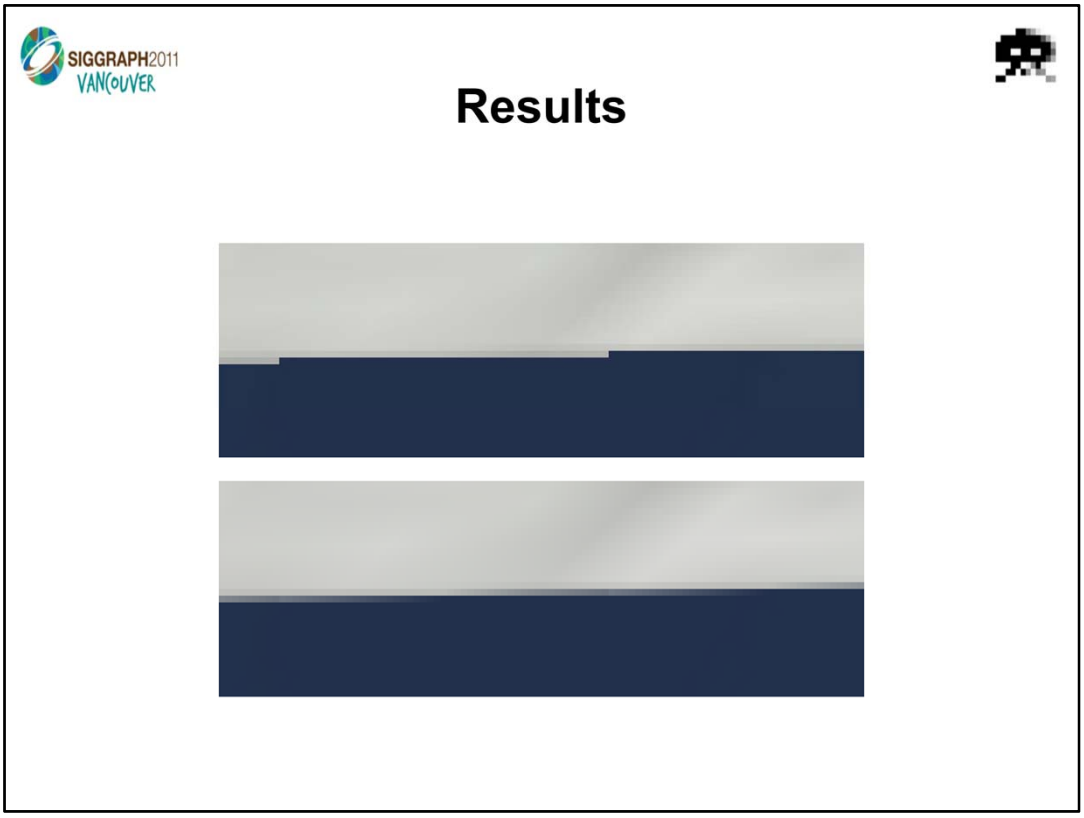
http://www.iryoku.com/aacourse/

Filtering Approaches for Real-Time Anti-Aliasing

Distance-to-edge AA (DEAA)

Hugh Malan
CCP Games

hmalan@ccpgames.com



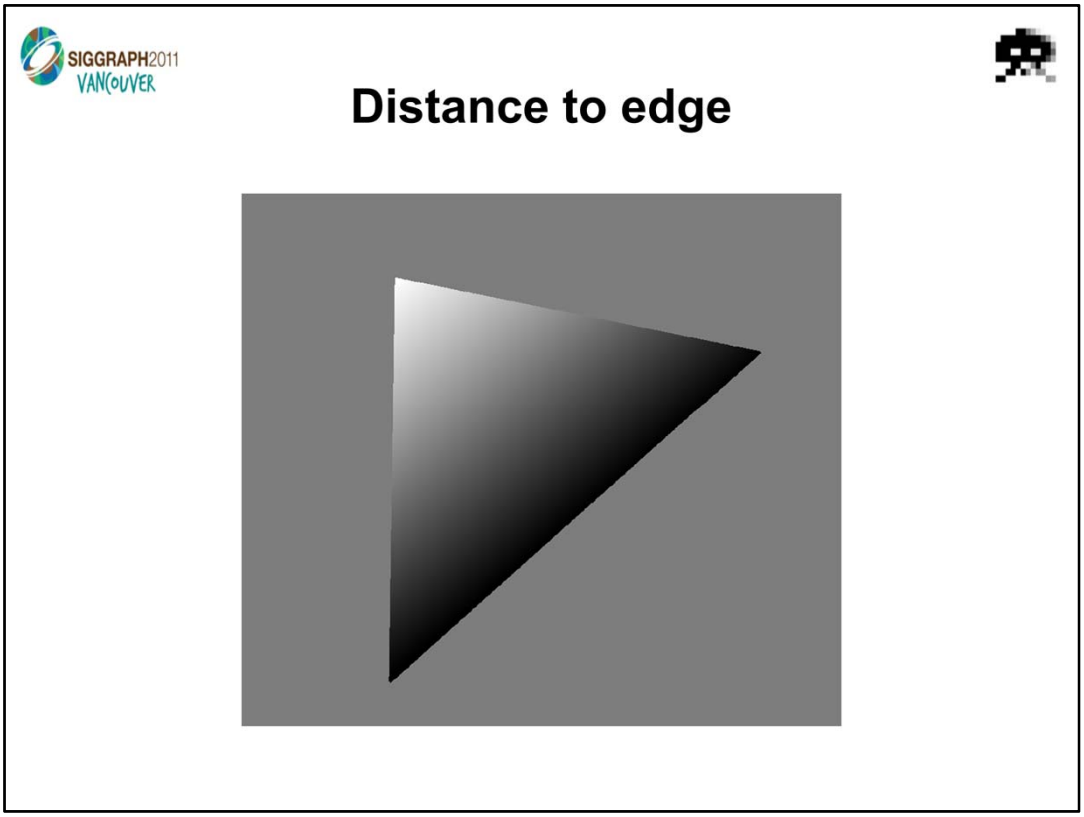
Top row is the original image; bottom row is the antialiased result.

This antialiasing method is part of the same family as MLAA and all the other antialiasing effects that operate as a postprocess. The approach we're taking is to compute distance to edge in the pixel shader during the beauty pass and write out blur hints alongside the color. The big advantage to this technique is that it has the capability to deliver very smooth edges, with up to 256 different levels of blur, and it avoids many of the temporal problems that MLAA techniques have like crawling along rotating edges.

It has quite different strengths and weaknesses to standard MLAA. Since the distance-to-edge is computed analytically there's no search for edges in the pixel's neighbourhood, so the expensive local-search step is skipped completely. The corresponding downside is that it needs extra space in the framebuffer to store the distance-to-edge information: 8 bits is OK, 32 bits would be ideal.

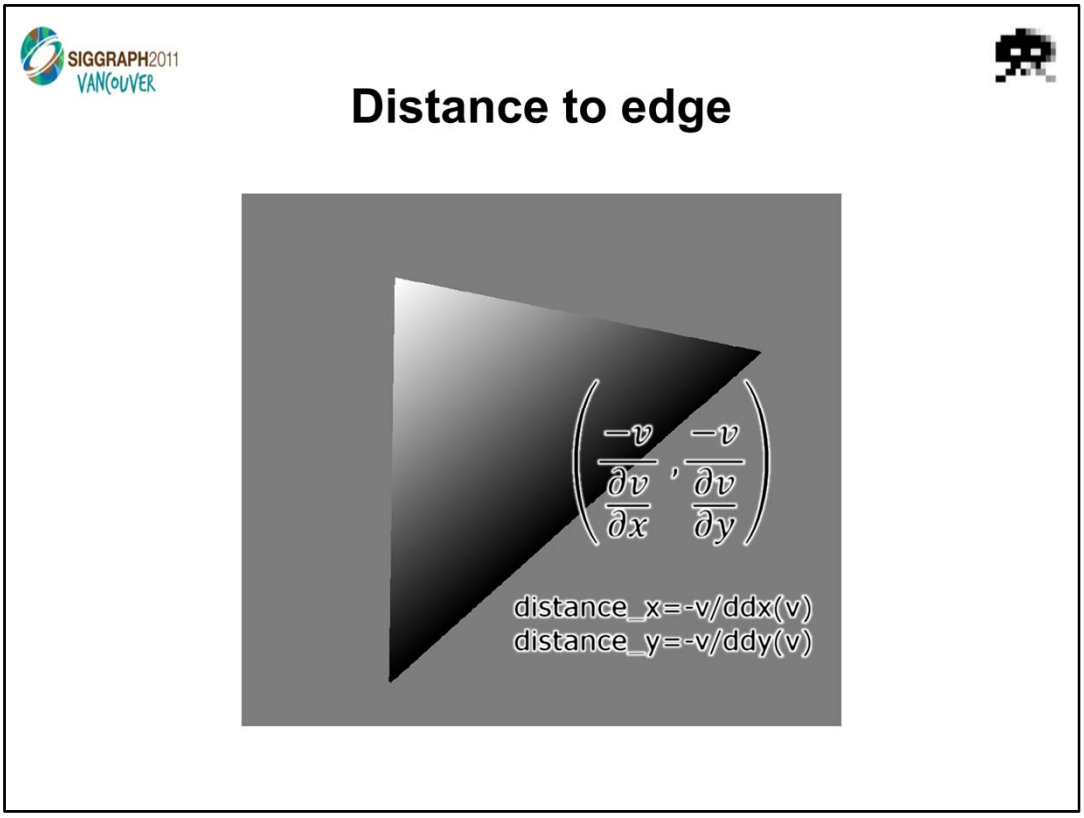
It does have a couple of other downsides though, which I'll cover.

We investigated applying it to APB, and I'll go over the problems that we hit. The obvious question is, why didn't this make it in? The reasons are pretty boring: this method had a couple of problems when we ran it with their content, we were still working on those problems and were not ready to start thinking about integrating it when they were locking down the assets for launch.



The most important part of this technique is the method for computing the distance to the nearest triangle edge in the pixel shader, so we'll start there.

For a given triangle, if the vertex program outputs the value v=1 on one vertex and v=0 on the other two verts/opposite edge, then the interpolated value v is roughly proportional to distance from that edge.

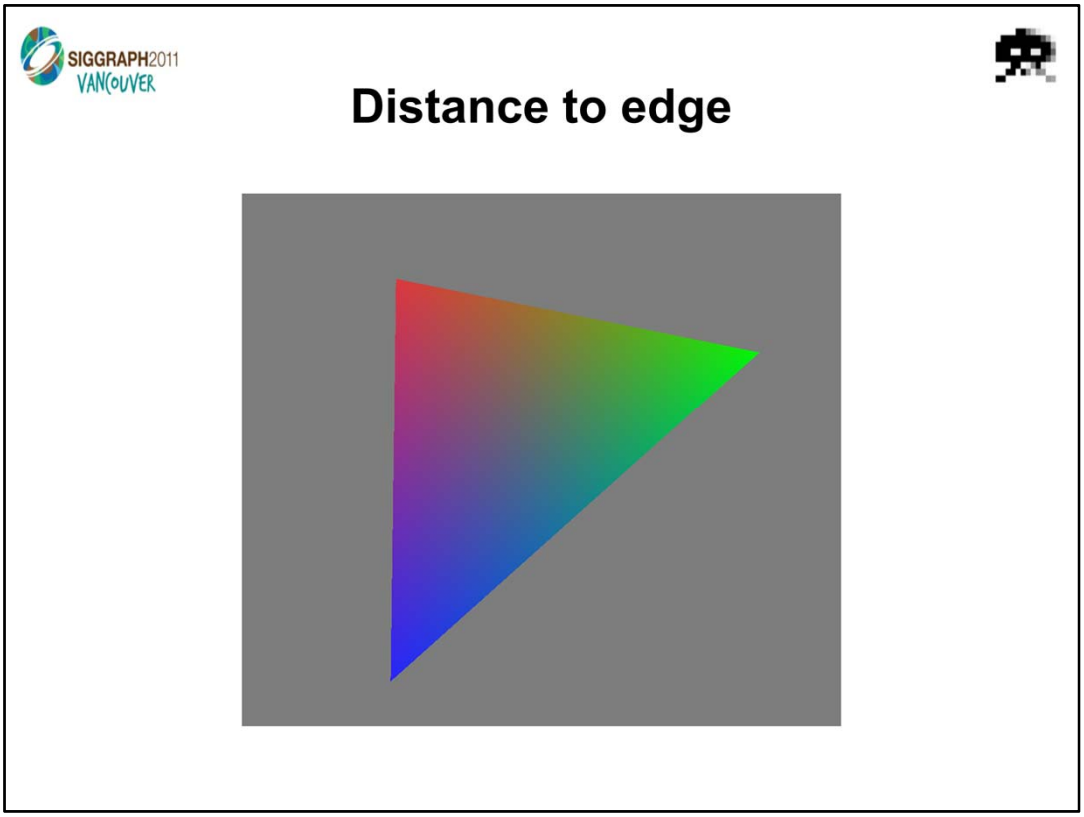


The values distance_x=-v/(dv/dx) and distance_y=-v/(dv/dy) give the signed distance to the edge of the triangle in screen pixels. If distance_x is positive, the triangle edge is to the right of the current pixel; if it's negative is to the left of the current pixel.

So this method allows the pixel shader to figure out the distance to one particular edge of the triangle, with pretty good precision. Due to perspective correction v is not linearly interpolated across the triangle, so the calculation gives an estimate rather than an exact value.

There's a lot of interesting uses for the basic ddx/ddy screen-space distance to edge maths, but we won't go there in this talk.

Anyway, the next step is to set up some fast, robust code in the vertex program to output the value 0 on silhouette verts and 1 everywhere else, that supports skinned and deforming meshes, procedurally generated geometry, and everything else that shows up in games these days.

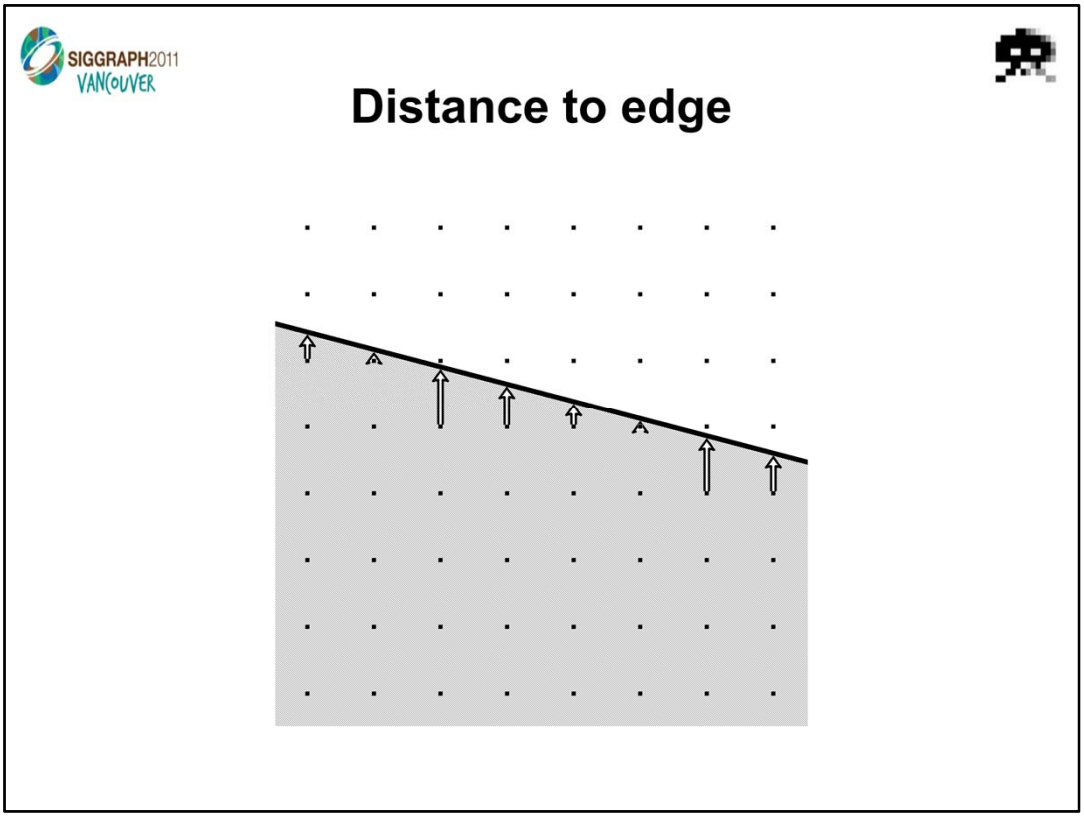


Unfortunately, I don't know of any such code. I experimented with a bunch of methods and couldn't find anything both robust and fast enough to be an option. I'd suggest taking a different approach altogether, which is to write out distance-to-edge hints for all potential aliased edges and let the postprocess decide whether to blur based on color difference.

Computing the distance-to-edge value for all three edges is straightforward. We do this by making v be a three-component vector instead of a scalar. This image shows the vector v as a color. At each of the three verts one component will be 1, and the others 0. To put it another way, along each edge of the triangle exactly one component will be zero – eg in the image the top edge is implied by the blue component, the left edge is implied by the green component.

We can then figure out the distance to the edge of the triangle in the four directions left, right, above and below the pixel we're currently shading. To be precise, for each direction we measure the distance to each of the three triangle edges, and take the minimum of the three values.

As you might guess, all the maths can be vectorized and made completely branch free.

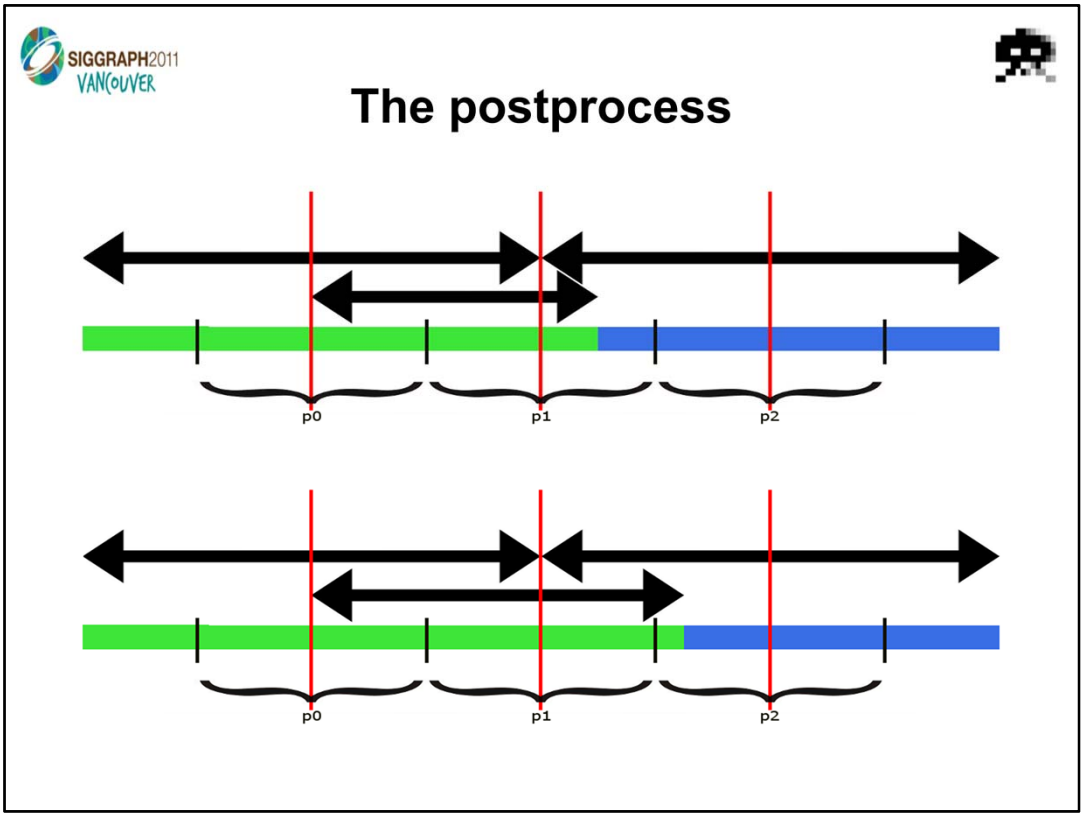


This diagram shows a large triangle being drawn. The black line is the edge of the triangle; pixels below the line are inside the triangle and are rasterized. The arrows indicate pixels where the distance upwards to a triangle edge is less than one pixel - we don't care about distances greater than one pixel, for reasons that'll become clear shortly.

We calculated distance values for four directions - left, right, up and down. Storing them in an RGBA rendertarget with 8 bits per component is the natural choice. This is a pretty expensive item – if it's a dealbreaker, there are some other options.

If this triangle was part of a surface, then we'd expect a complementary triangle to be rendered on the other side of the edge indicated by the black line, and to write out distance-to-edge values for the upward direction that were the exact inverse of the downward distances shown here. But if the black line is a silhouette edge then this won't happen, the distance values for the complementary pixels could be anything. These cases will come up again when we talk about the postprocess.

Also note that these distance hints are written no matter what, irrespective of whether the edge needs antialiasing or not.



To explain how the postprocess works let's start with the 1D case.

The image above shows two cases. The green and blue rectangles indicate two pieces of geometry that have been rasterized; they're slightly different in the two cases. Three pixels are involved: p0, p1 and p2. The long red vertical lines indicate the pixel center, corresponding to the positions where the rasterizer samples the geometry and the forward pass pixel shader is run. The short black vertical lines indicate midpoints between pixel centers. The arrows show the distance-to-edge stored for each pixel center; the maximum distance is clamped to 1 pixel.

In the diagrams, imagine the cube is the green region and the sky is the blue. Pixel p1 is part of the cube, pixel p2 is part of the sky. When the cube was rasterized it wrote to pixel p1 and set up the distance-to-edge values; but pixel p2 is part of a large sky triangle and has no relevant distance-to-edge hints.

In this case only one pixel of a neighbouring pair has a distance-to-edge value: this will happen along all silhouette edges, and near nonmanifold edges.

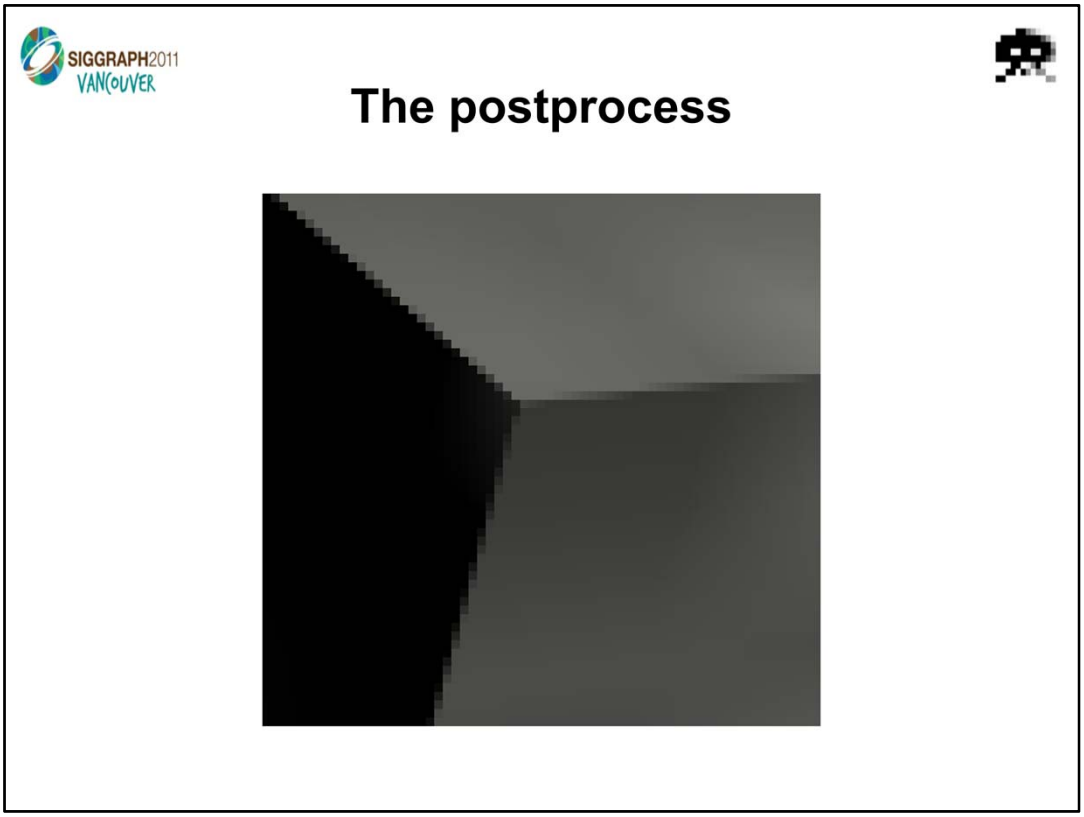
There are two other cases cases to consider. Two neighbouring pixels might have complementary distance-to-edge values - this will happen when the two pixels are in two neighbouring triangles that share an edge. Alternatively the distances might both be defined, but not be complementary: once case where this will happen if there are lots of subpixel triangles.

In the upper case in the diagram, the only pixel that needs to be blurred is pixel p1. The distance-to-edge values encoded for that pixel give us all the information required about pixel coverage. A bit of pixel p2 needs to be blended in.

The lower case is more complex: note pixel p2. This pixel should be a mixture of green and blue, even though the distance-to-edge values for pixel p2 suggest there's no edges nearby. To correctly handle this case, the postprocess has to look at the hint values for neighbouring pixels, and choose between the two competing values.

One straightforward way to support all these cases is for the postprocess to choose the minimum of the two competing distance-to-edge values to compute the blend.

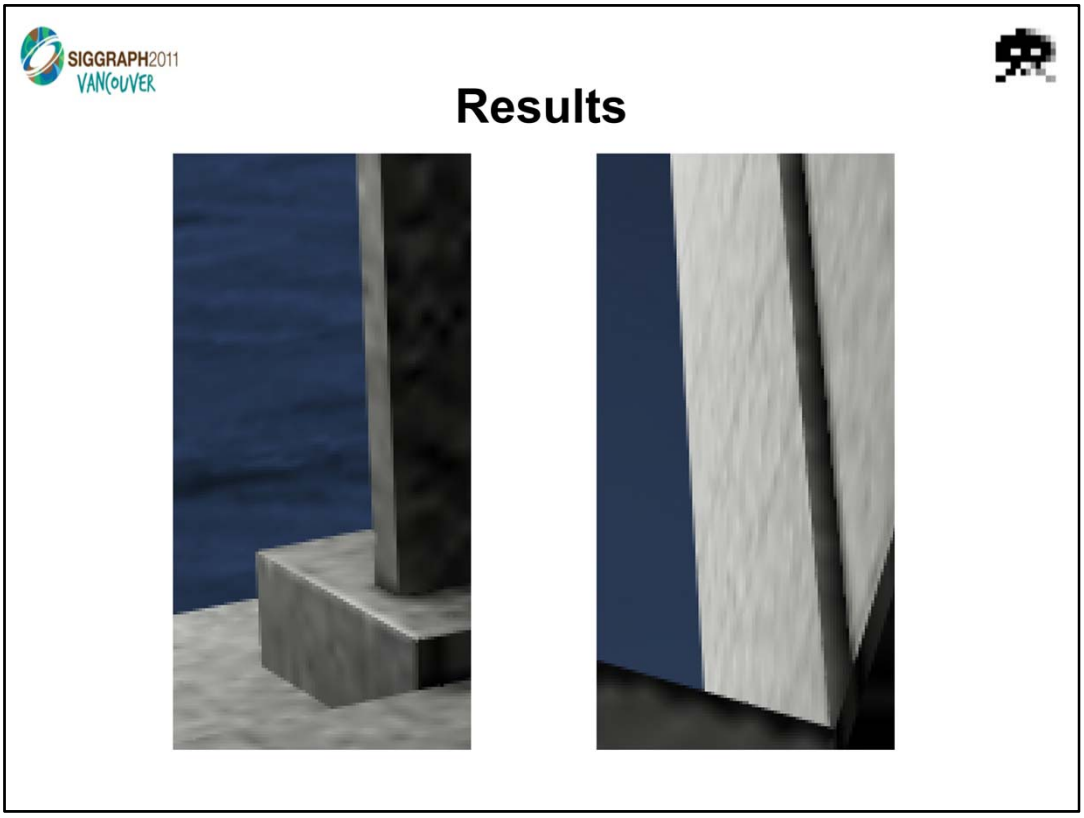
So the whole process is this. For each pixel, consider the region within half a pixel in each direction. Examine the two neighbours and compute the distance-to-edge value in each direction. If the distance-to-edge is less than half a pixel, then that neighbour will need to be blended in. The contribution of each of the three pixels is the fraction of the one-pixel area it covers.



Extending to 2D is simple: for each pixel, apply the 1D case either vertically or horizontally. Decide which by summing the distance-to-edge values involved in each approach, use the axis with the smaller sum.

A more natural option is to compute both horizontal and vertical blends and combine them. For instance: multiply the calculated two coverage values for the center pixel, and renormalize the resulting weights. All these options costs more and none of the options I tried improved the quality; they often introduced artifacts.

Lastly, the blur only needs to be applied to edges that are potentially aliased. This can be decided by comparing pixel colors and skipping the blend if the color difference across an edge isn't big enough to cause noticeable aliasing, in a similar spirit to MLAA.

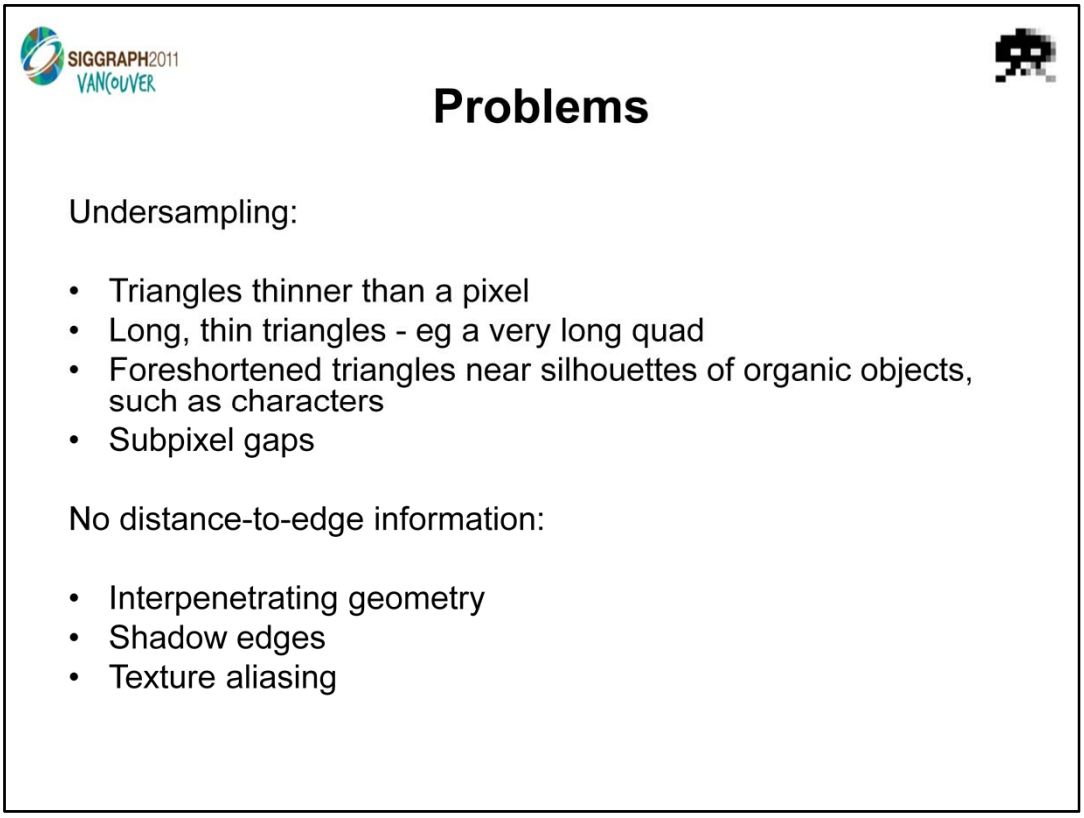


Demo here.

We get a very smooth gradient that correctly matches the steps of the edge. Duplicating the quality of the gradient with MLAA would need a very large neighbourhood search.

One situation where MLAA's local neighbourhood is especially noticeable is for an edge slowly rotating past an axis. With MLAA the local neighbourhood reveals itself with localized blurs on each step, which travel along the edge as it rotates. With DEAA we get a seamless transition as it copes with the very long edge steps.

One case that MLAA can't support is subpixel movement of an edge – for instance, where a perfectly vertical line moves sweeps across the screen. Methods that only use the color buffer can't infer the subpixel position of the edge; in comparison DEAA will correctly simulate new columns of pixels fading in as the line moves.



Any thin feature (such as a ledge, or doorframe) will become thinner than a pixel when viewed from side-on or from a distance.

Triangles that are thinner than a pixel lead to problems because the percieved edge won't have a consistent set of distance-to-edge values. A cluster of pixels along an edge with unnaturally low distance-to-edge values creates a section of edge that looks aliased. When all other edges are nicely antialiased these sections really stand out.

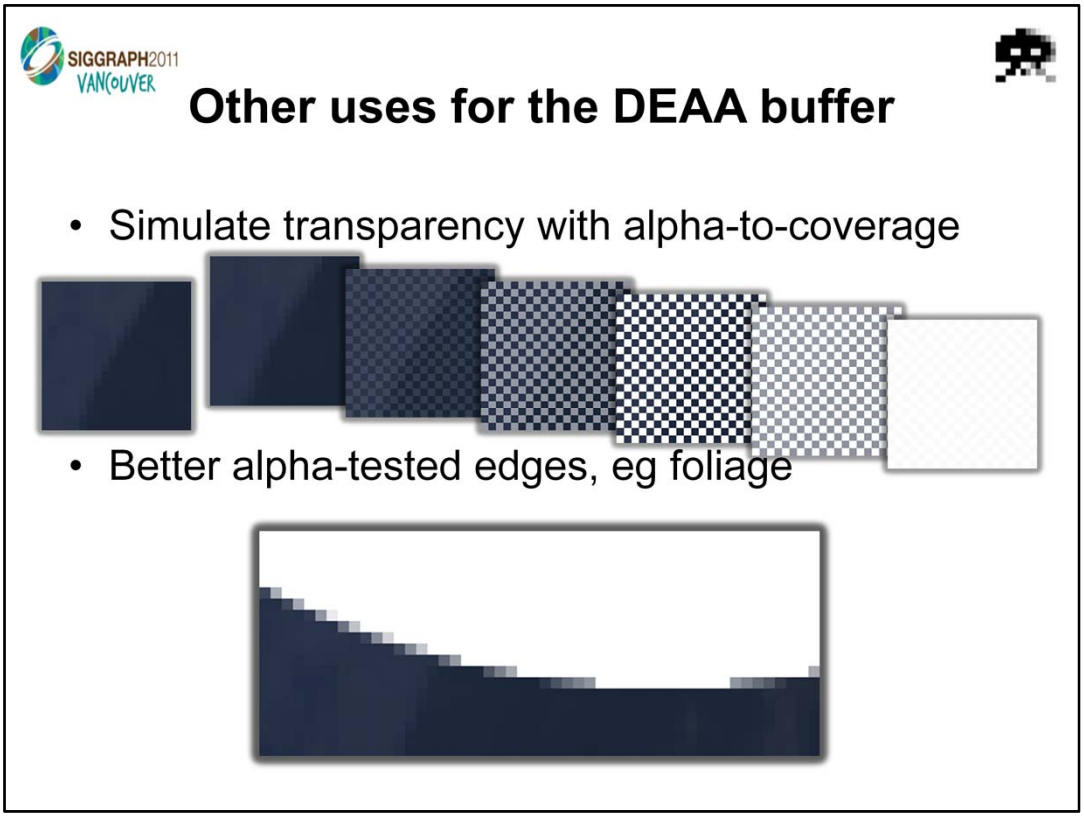
One way to minimize the problem is to make the tools that creates the vertex data not set up a varying silhouette parameter towards an edge that will never cause aliasing problems.

This isn't an option smoothly curving surfaces where neighbouring triangles have only a slight bend between them, such as character models. In this case subpixel triangles are unavoidable near the silhouette edge. In the GPU Pro article I suggested extruding backfacing verts outwards by a pixel or half a pixel to ensure that silhouette triangles are large enough to avoid the undersampling problems. This does work, but when used on characters it makes them look fat. In that article I described a two-pixel blend for the postprocess, and it led to really bad quality problems in these cases. The three-pixel blend that I described here copes much better.

Subpixel gaps between geometry leads to aliasing problems because the gap is undersampled. For example, a thin gap between two pillars showing sky. Although the distance-to-edge values might be completely consistent, the pixel being blended in

won't consistently be sky; sometimes it'll be from the other pillar.

Lastly, I've listed three cases where there is a visible edge but there's no distance-to-edge information, so this method won't help. In comparison, MLAA will provide some antialiasing in each of those cases.



Simulated alphablending.
Overwrite the distance-to-edge hint values to fade the new object in progressively.
The first image in the sequence is the original, the other six have a 50% alpha-to-coverage checkerboard rendered over the top. No alphablending is used. The distance-to-edge values vary from 0.0 to 1.0 over the sequence; the postprocess blur blends in neighbouring pixels by varying amounts to simulate the alpha blend.

With some limitations, this means you could apply deferred lighting to semitransparent surfaces. Alex Evans described how LittleBigPlanet fakes deferred lighting on 50% transparent surfaces in a similar way.
Another use is better quality alpha-to-coverage effects.

A second idea is for improving alpha-tested edges, like on foliage.
This is pretty simple: compute the alpha-test value, and kill the pixel if it's below zero as usual. If the pixel is not killed, feed the alpha-test value into the existing distance-to-edge evaluation function and framebuffer encode process, and you're done.