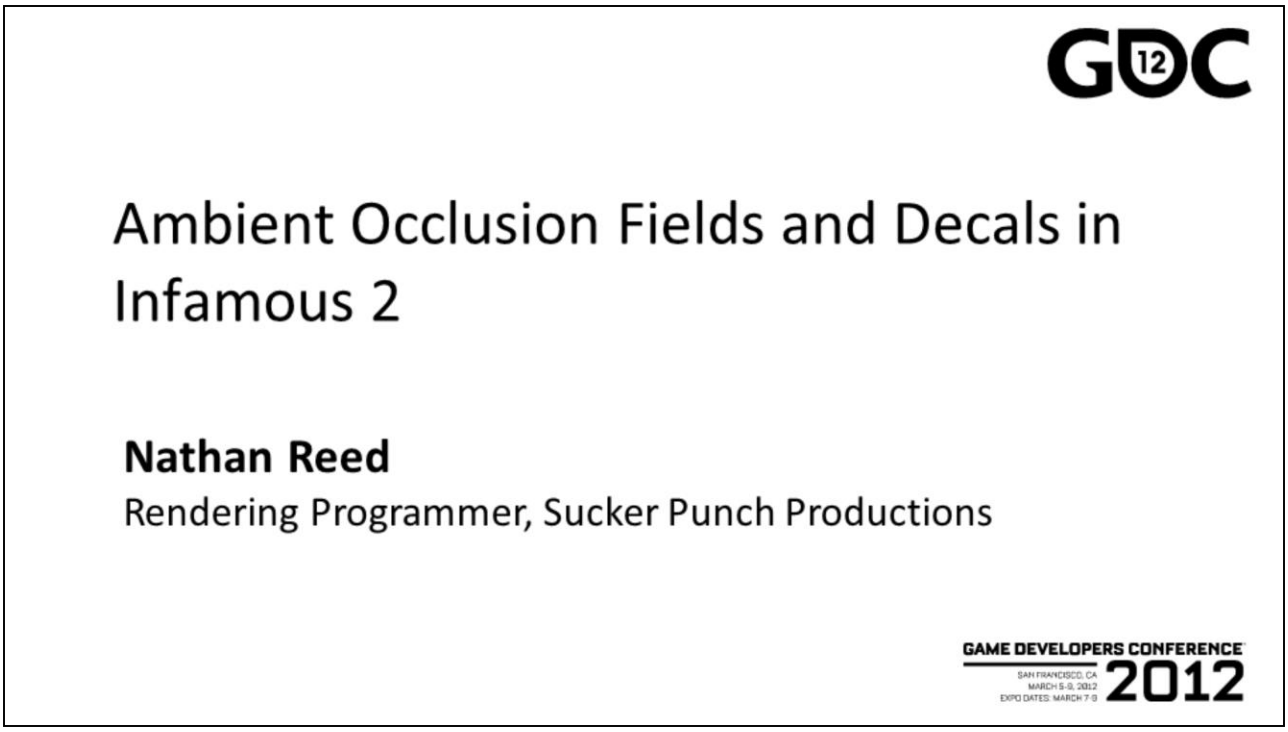


I'm Nathan Reed, a rendering programmer at Sucker Punch Productions in Bellevue, WA, and I'm going to speak today about a couple of new ambient occlusion techniques we used in our recent game, Infamous 2.

Background: Infamous 2

- PS3 exclusive
- Open-world, urban environment
- Deferred-shading renderer
- Supports per-vertex baked AO, and SSAO

First of all, some background on our game: Infamous 2 is a PS3 exclusive, open-world game set in an urban environment. We have a deferred-shading renderer, and like many game engines, it supports two main ambient occlusion (AO) technologies: static, per-vertex baked ambient occlusion, and screen-space ambient occlusion (SSAO).

AO – large or small scale?

- Baked AO is great, but…
  - Per-vertex needs tessellation for fine detail
  - Lightmaps need a lot of memory for fine detail
  - Can't move things around at runtime
- Best for large-scale, static objects

Static, baked AO is great when it works, but it has some drawbacks. In order to get smaller-scale details in your AO, you may need to tessellate your meshes more than you'd like if you're baking AO per-vertex; or if you're store it in textures, they need a lot of memory to get enough resolution for fine detail, especially for a big, open-world environment. And of course, with any baked approach you can't move or change anything in real-time.

Therefore, baked AO is best-suited for very large-scale occlusion where both source and target are likely to be static, such as from a building onto the streets, alleys, and other buildings around it. It's not well-suited for smaller-scale occlusion or for things that may move around.

AO – large or small scale?

- SSAO is great, but…
  - Limited radius in screen space
  - Missing data due to screen edges, occlusion
  - Inconsistent from one camera position to another
- Best for very fine details

On the other hand, SSAO is completely dynamic, so it can adapt to anything moving or changing. But it typically has a limited radius in screen space for performance reasons, so if you get up close to an object the shadows will seem to contract, since they can't get larger than a certain number of pixels. And you have no information about anything that's offscreen, or behind something else. Because of both of these effects, SSAO can give different-looking shadows at different camera positions.

As a result, SSAO is a good fit for very fine details of ambient occlusion, but not for larger scales.

Our hybrid approach

- Can complement baked AO and SSAO
- Medium-scale, partly static
- Work in world space: precompute AO from an object onto the space around it, store in a texture.

There's a gap between baked AO and SSAO, where neither approach is very well-suited for occlusion on the medium scale, bigger than the SSAO radius but smaller than mesh tessellation. So in our engine we've added a hybrid approach that can supplement baked AO and SSAO by handling occlusion on the medium scale.

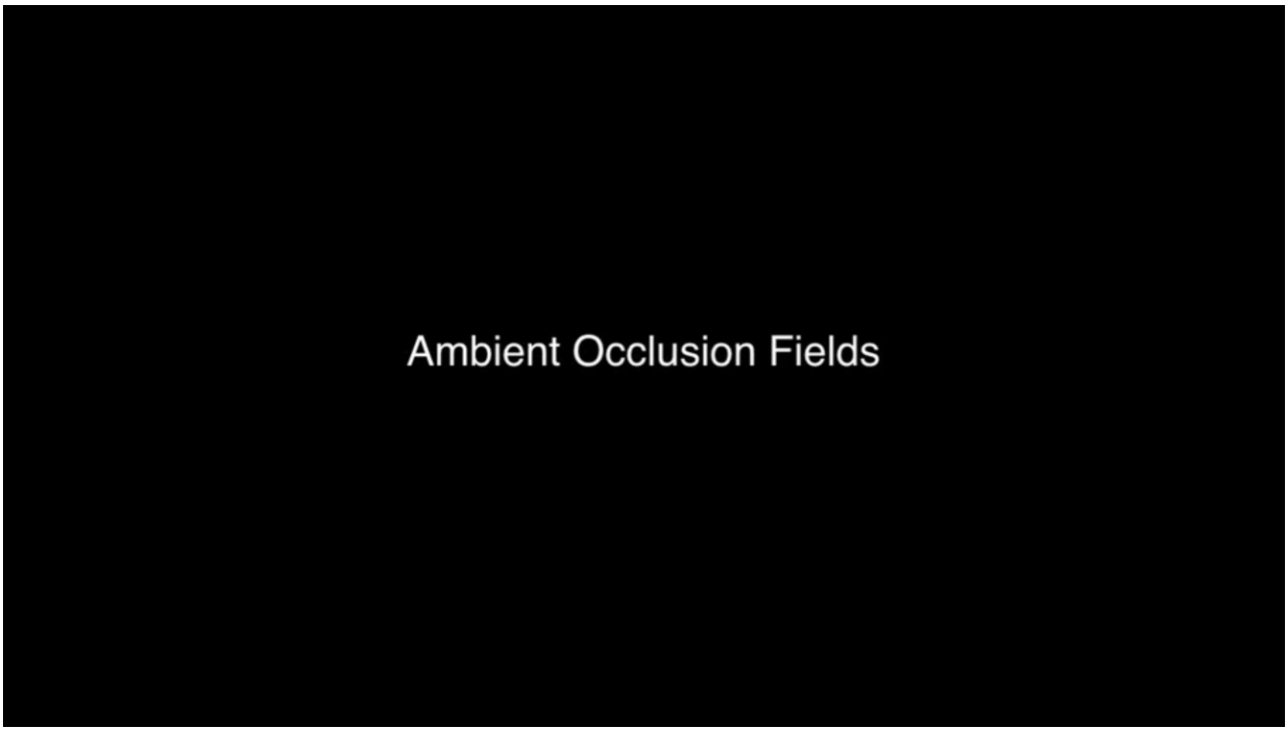
The basic idea is to precompute a representation of the AO that an object casts onto the space around it, and store that data in a texture. This is done in world space, so it has a consistent appearance from all camera positions.

Our hybrid approach

- Precompute based on source geometry only, not target. Can be moved in real-time.
- Apply like a light in deferred shading: evaluate AO per pixel, within region of effect.
- Two variants: AO Fields & AO Decals

And the precompute is based only on the source geometry, not on the target, so it can be moved around in real-time. It's not completely dynamic; it does require the source geometry to be rigid. It gets applied very much like a light in deferred shading: we draw a box around the object and use a pixel shader to evaluate AO at each shaded point within the box.

There are two variants of this, which we call AO Fields and AO Decals, and I'll talk about each in turn.



Let’s start with AO Fields. Here’s a video to demonstrate the technique. (The video is at http://reedbeta.com/gdc)

SSAO is disabled in this video, so the contact shadows you’re seeing around these objects is all due to the AO fields. We use it on many smaller objects like the mailbox and potted plants, but also on a few larger ones, such as the cars. As you can see, it gives quite plausible results for objects in motion.

AO Fields

- Similar to previously reported techniques
  - Kontkanen and Laine, "Ambient Occlusion Fields", SIGGRAPH '05
  - Malmer et al. "Fast Precomputed Ambient Occlusion for Proximity Shadows", Journal of Graphics Tools, vol. 12 no. 2 (2007)
  - Hill, "Rendering with Conviction", GDC '10

AO fields are similar to a few previously reported techniques, and here's my list of references.

AO Fields: Precomputing

- Put a volume texture around the source object
- Each voxel is an occlusion cone:
  - RGB = average direction toward occluder
  - A = width, as fraction of hemisphere occluded

So how does this work? First of all, we put a box around the car, and put a volume texture in the box. Each voxel in that texture stores an occlusion cone representing how the car looks from that point. The RGB components are a unit vector in the average direction of occlusion, and the alpha component stores the width of that cone, as a fraction of the hemisphere occluded.

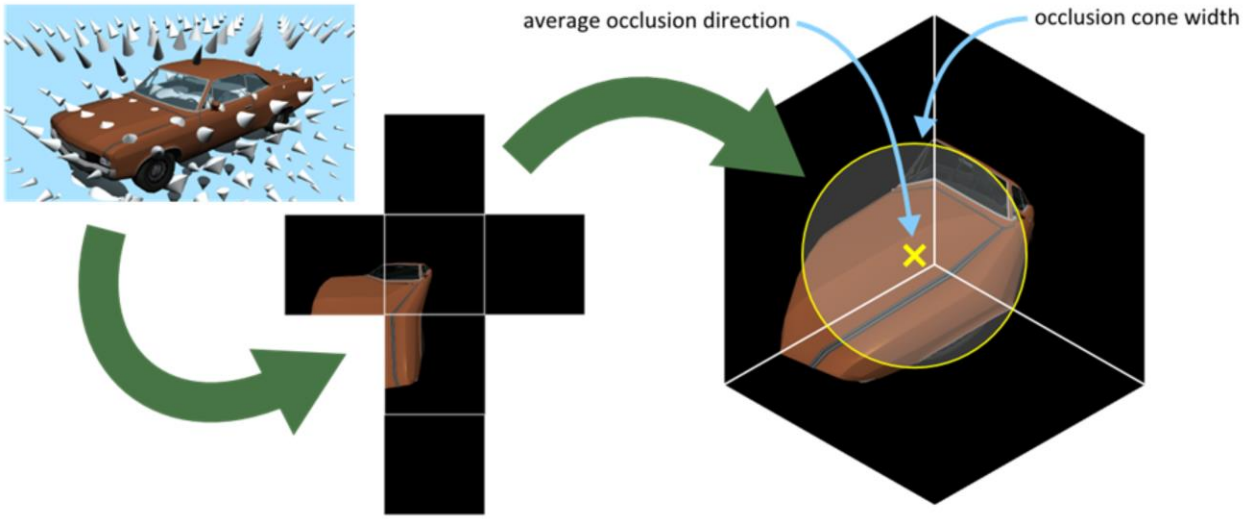
Here's a diagram of the occlusion samples surrounding the car. Each cone represents one voxel of the texture, and as you can see, the cones points toward the car, getting wider the closer they are.

AO Fields: Precomputing

- Iterate over volume texture voxels
  - Render geometry into a 32×32 cubemap centered on each voxel
  - Read-back and compute average direction of drawn pixels (weighted by solid angle)
  - Compute occluded fraction of hemisphere around that direction

All of this gets built offline by our tools in a pretty straightforward way. For each voxel, we put the camera at the voxel center and render the car into a small cubemap. Then we pull that cubemap back and work out the centroid of the drawn pixels, in 3D, with solid angle weighting. Then we count how many pixels were drawn, again with solid angle weighting, to get the occluded fraction of the hemisphere.

AO Fields: Precomputing



Here's this process schematically. There's an example of the cubemap as seen from one particular voxel. We pull back that cubemap, use the centroid of the drawn pixels to get the cone axis, and count the number of drawn pixels to get its width, as a fraction of the hemisphere.

AO Fields: Applying

- Draw the bounding box; pixel shader retrieves world pos and normal of shaded point
  - Just like a light in deferred shading – same tricks & optimizations apply
- Sample texture, decode occlusion vector and width
  - Transform world pos to field local space
  - Transform occlusion vector back to world space

That was the precomputed part of it. Now in real-time we need to apply this. It's exactly like a light in deferred shading: we draw the bounding box of the field and in the pixel shader, we sample the G-buffer to get the world-space position and normal vector of the shaded point. All the usual deferred-shading optimizations can be used, such as stencil masking or depth bounds tests.

Once we have the world position, we transform that into the local space of the field, sample the volume texture to get the occlusion vector and cone width, and transform the occlusion vector back into world space.

AO Fields: Applying

- Estimate occlusion using equation:

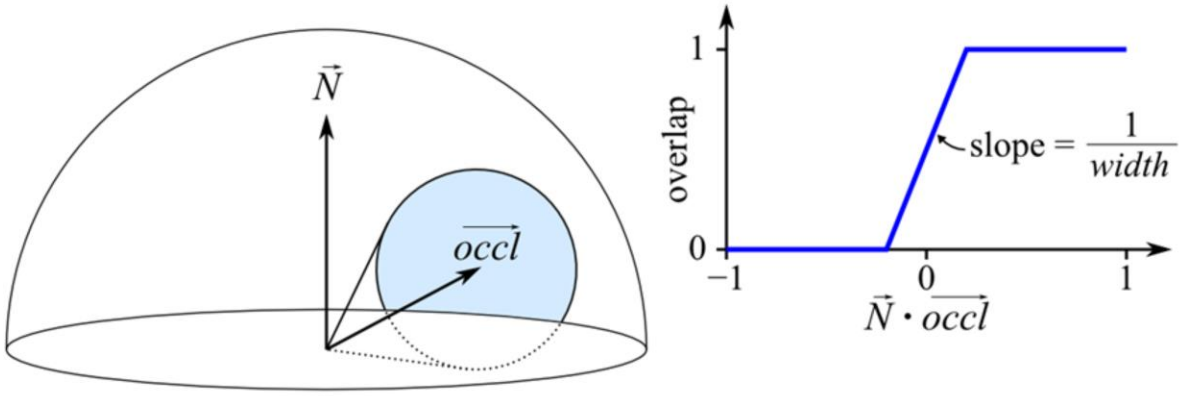
$$AO = 1 - strength \times width \times \text{saturate}\left(\frac{\vec{N} \cdot \overrightarrow{occl}}{2 \times width} + 0.5\right)$$

- Strength is an artist-settable parameter per object; controls how dark the AO gets

Finally, we estimate the AO for the pixel according to this equation, which uses the normal of the target surface and the occlusion vector and width retrieved from the texture. Strength here is an artist parameter that controls how dark the AO gets. It can also be used to fade out the AO fields as they get far away, for LOD.

The saturate factor on the end of this equation deserves a little explanation.

AO Fields: Applying



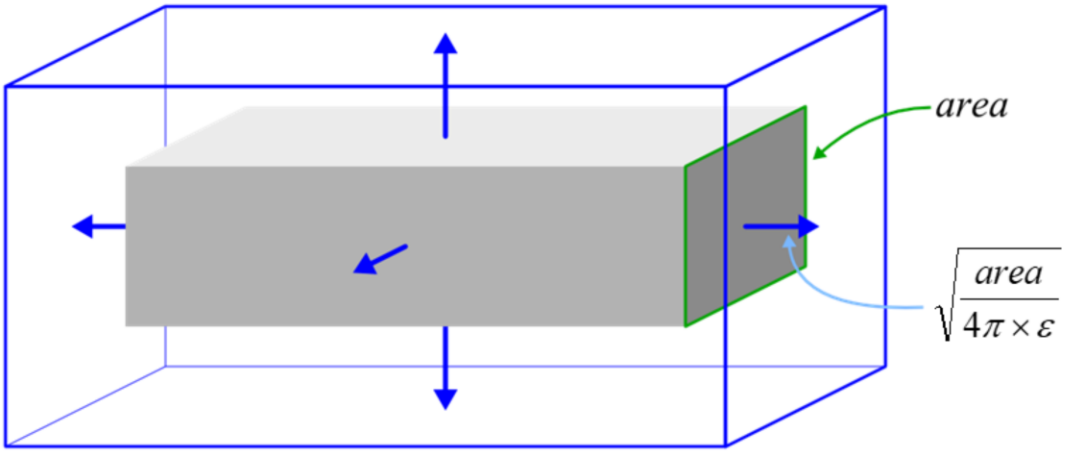
Here's a diagram of what that term does. It's approximating the fraction of overlap between the occlusion cone and the normal hemisphere. The cone might not entirely be within the hemisphere, in which case we shouldn't apply the entire occlusion value. Previous approaches used a more complicated function or a lookup table here, but we just approximate it by this clamped linear ramp based on the dot product of the normal and occlusion vector, with slope based on the cone width. It's a very coarse approximation, but in my experience it works well.

AO Fields: Applying

- Blend result into G-buffer's AO channel using multiplicative blending
  - No special treatment for double-blending – in our use cases, not really an issue

Once we have the AO value, we just multiplicatively blend it into the G-buffer's AO channel. We don't do anything special to work around double-blending issues – in our use cases, we don't typically have AO fields overlapping so much that this would be an issue.

AO Fields: Bounding Box Size



Now for some of the bothersome technical details. The first issue is how large should we make the bounding box? We used a procedure suggested by one of the references, the Malmer paper. Here, the gray box is our car, or whatever source object, and the blue box is the AO field. To get the AO field size, we start with the source object's bounding box and expand it by pushing each face out a distance based on that face's area. The epsilon is a desired error – that is, the error due to cutting the AO field off at a finite distance (since it would ideally go on forever).

AO Fields: Bounding Box Size

- From Malmer paper:

$$extend = \sqrt{\frac{area}{4\pi \times \varepsilon}}$$

- Epsilon is desired error. We used 0.25.

We used an epsilon of 0.25, which is fairly generous but keeps the boxes from getting too large and costly to draw.

AO Fields: Texture Details

- Texture size: chosen by artist, typically 8–16 voxels along each axis
  - Car: 32×16×8 (= 16 KB)
  - Park bench: 16×8×8 (= 4 KB)
  - Trash can: 8×8×8 (= 2 KB)
- Format: 8-bit RGBA, no DXT
  - Density so low, DXT artifacts look really bad
  - No mipmaps necessary

The texture size is chosen by the artist for each object. The car was the largest one in our game, at 32x16x8. Most other objects were only 8-16 voxels along each axis.

We stored the textures in standard 8-bit RGBA format, with no DXT compression, and no mipmaps. The trouble with compression is that because the voxels are pretty large, any DXT artifacts are just enormous and look terrible. At the end of the day. the total texture size is about 2-16K per unique object.

Unfortunately, there are a few artifacts that show up with all this, and I'm going to talk about how we solved them. The first you'll notice with AO fields is that since the field cuts off at a finite distance, the occlusion doesn't go all the way to zero at its edge, so you can see this very obvious box-shaped shadow around the car.

AO Fields: Visible Boundary

- Remap alpha (width) values at build time
  - Find max alpha among all edge voxels
  - Scale-bias all voxels to make that value zero:

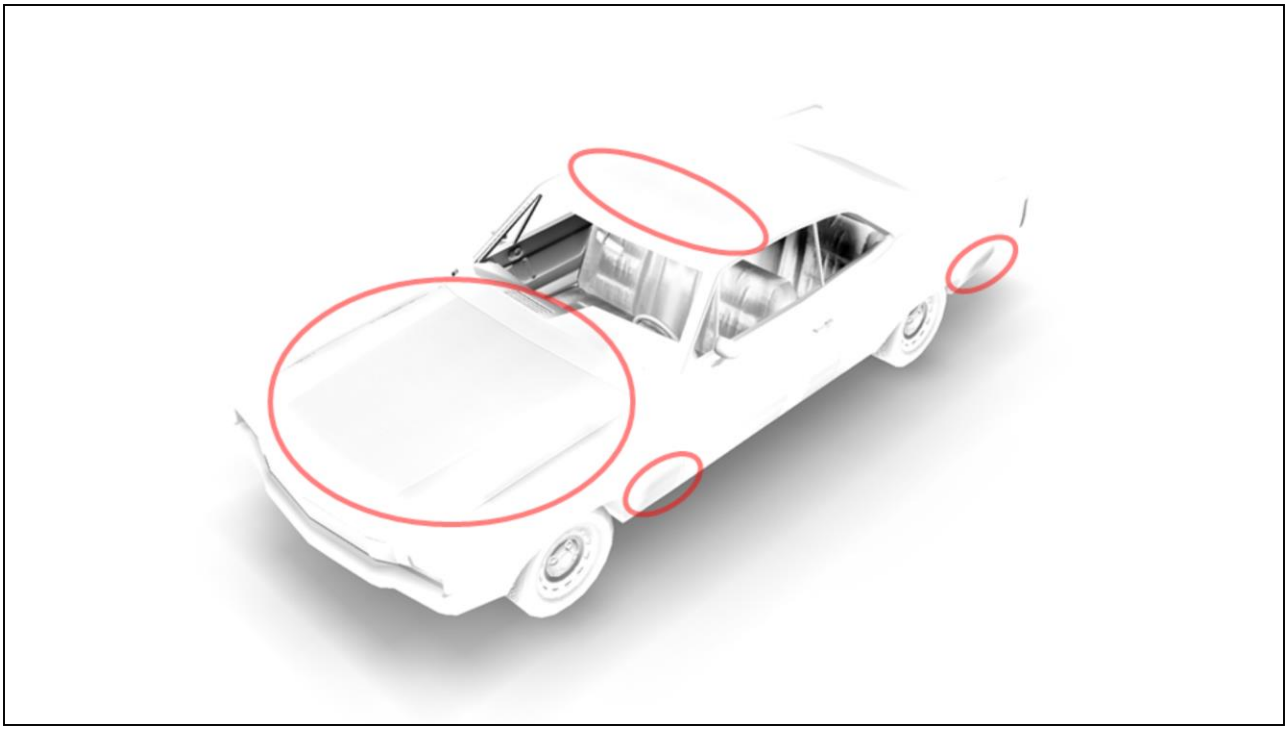
$$alpha := \text{saturate}\left(\frac{alpha - alphaMaxEdge}{1 - alphaMaxEdge}\right)$$

We solve this in the simplest way possible, by just forcing all the alpha values (which are the occlusion cone widths) to be zero at the boundary. We iterate over the edge voxels and find the maximum alpha, then linearly remap all the alphas to send that maximum to zero. Here's the equation to do that.

So here's before that fix…

After

…and here's after. No more box.



Another artifact we saw was getting occasional splotches of incorrect self-occlusion on the surface of the object. The root cause of this is that the occlusion changes rapidly when you're close to a surface, and the low voxel density doesn't capture this well.

Here's the AO on the car. Each of the circled areas contains a dark blotch of incorrect self-occlusion.

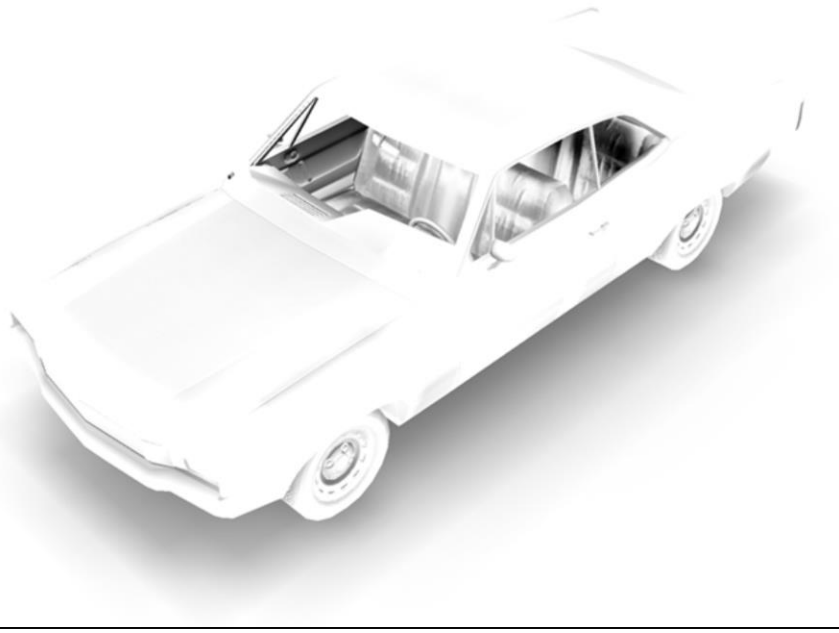
AO Fields: Incorrect Self-Occlusion

- Ideally: detect interior voxels and fix up
  - But identifying interior voxels is tricky
- Bias sample point away from target surface
  - In pixel shader, offset sample pos along normal
  - Bias length: half a voxel (along its shortest axis)

Ideally, the way I'd like to fix this is to detect voxels that are inside geometry and do some sort of fixup on them. However, identifying interior voxels isn't a trivial thing to do in 3D. We can't depend on our geometry to be 2-manifold or anything nice like that, so it's not that easy to define what's inside and what's outside.

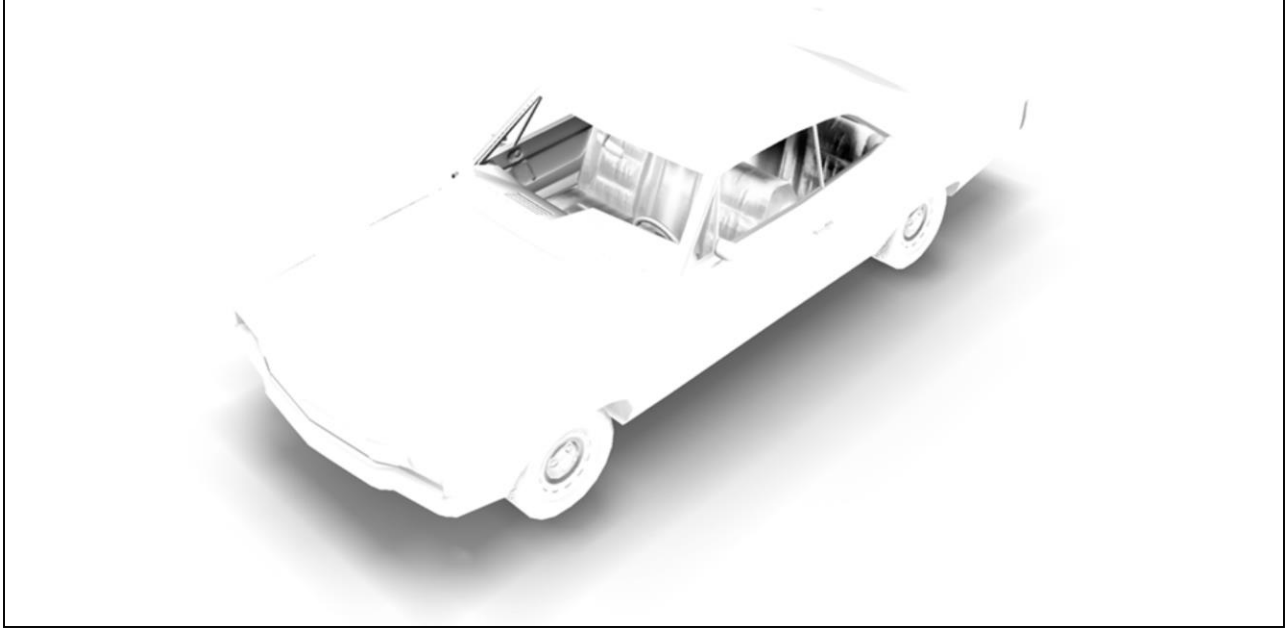
So, I opted for a hack that fixes the worst of the problems in a simple way. We just bias the sample point away from the receiving surface, in real-time. When we sample the volume texture, we push the sample point along the normal of the receiver by a fixed distance of half the size of a voxel. This helps get the sample points away from problem areas.

Before

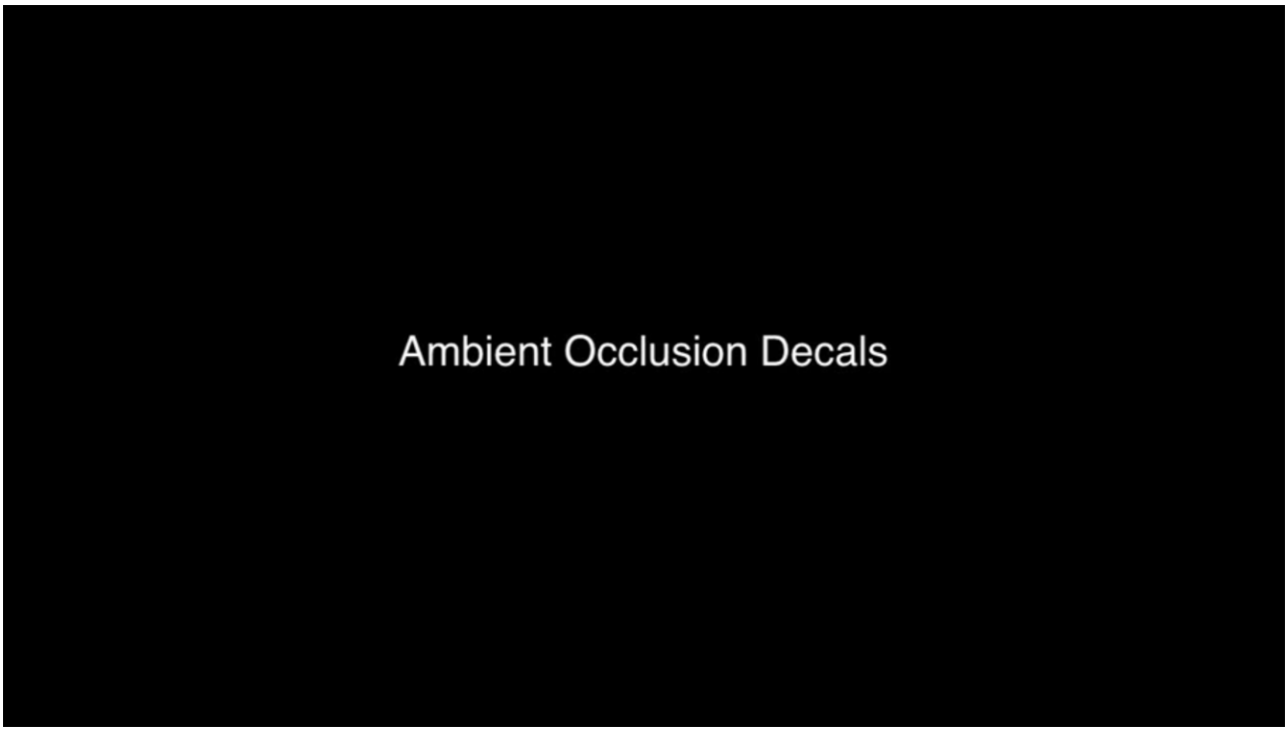


Here's the AO channel before the fix…

After



…and after. You can see that the areas of bad self-occlusion have vanished. The AO under the car also got a little bit more contrasty, because the sample points on the ground are being pushed up toward the car. In general, this normal biasing fixes problems, but it does occasionally create new problems. Still, on balance we think it's a win.



That's it for AO Fields. Now I'm going to switch gears and talk about AO Decals, a variant of this technique. Let's begin with a video of it. (The video is at http://reedbeta.com/gdc)

Again, SSAO is disabled here, and there's an AO decal on each of these windows. I'm switching it on and off there, so you can see they're producing those contact shadows around the window frames. And again, with this electric meter, the AO decal is rendering those shadows around the side of the meter.

AO Decals

- Planar version of AO Field
- Use cases: thin objects embedded in or projecting from a flat surface (wall or floor)
  - Window and door frames, air conditioners, electric meters, chimneys, manhole covers

As you saw, AO Decals are used for thin objects projecting from or embedded in a wall or floor. They're a thin, planar version of an AO Field.

AO Decals: Precomputing

- Store a 2D texture, oriented parallel to the wall/floor
- Four depth slices stored in RGBA channels
  - No directional information stored; just occlusion fraction for hemisphere away from wall

Here, instead of a volume texture, we use a 2D texture. This texture is oriented parallel to the wall or floor, and we store occlusion values for four depth slices in the RGBA channels. Unlike with AO Fields, we're not storing any directional information, just an occlusion fraction for the hemisphere away from the wall.

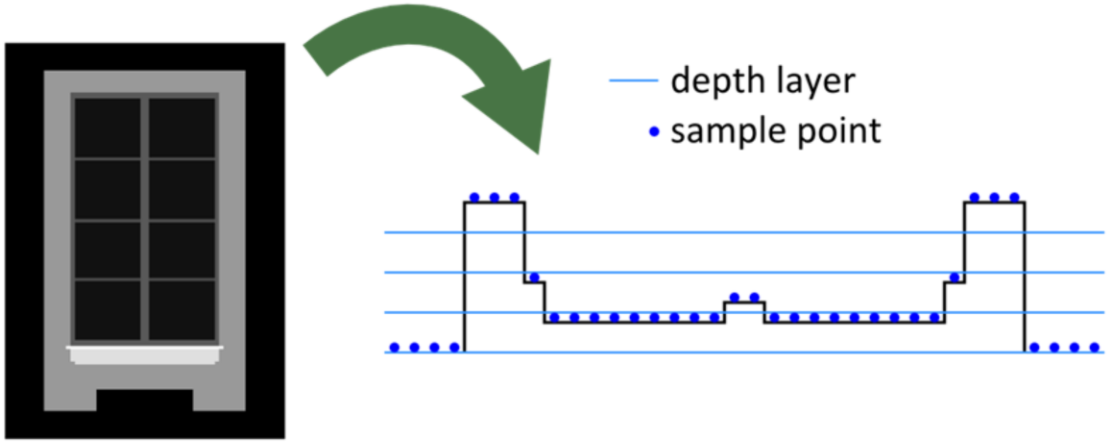
Here's a diagram of the depth slices. Here's a window, and I've added planes representing the four depth slices: red, green, blue, and alpha.

AO Decals: Precomputing

- Render heightmap of source geometry
  - Parallel projection looking at wall/floor from front
  - Draw geom in grayscale, black at back of depth range to white at front
- Iterate over texels, take an AO sample just above heightmap at each texel
  - Trying to make sure we capture AO at the surface well, since that's where it will be evaluated

Eventually we're going to end up with an occlusion sample at each texel on each of those four planes. But first, I'm going to begin by rendering a heightmap of the source geometry, at the same resolution as the decal texture will eventually be. This is just a parallel projection looking at the object from the front; we draw it in grayscale, with black at the back of the depth range and white at the front. Then we use the heightmap to place our initial set of samples. We take an AO sample at each texel just above the height rendered in the heightmap. We're doing this to try to make sure that we capture the AO at the surface well, since we know we're going to evaluate it there.

AO Decals: Precomputing



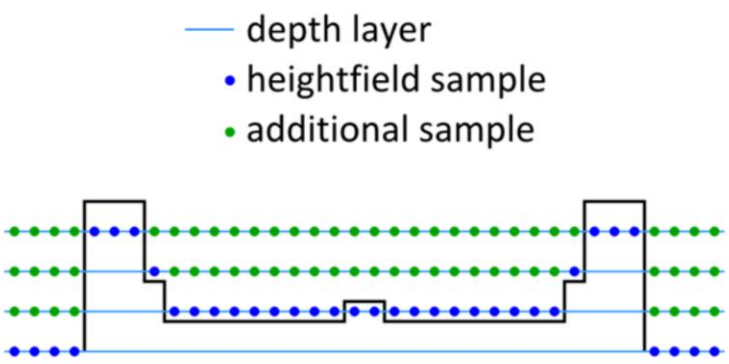
Here's an example of the heightmap for that window, and here's a diagram – this is a top view now – showing the sample points generated from the height field. As you can see, they're not all lined up with one of the depth slices.

AO Decals: Precomputing

- Assign sample to nearest depth slice
  - Depth slice positions are depthRange * i / 4.0 (i = 0, 1, 2, 3)
  - Front of depth range (i = 4) always 0 occlusion
- Take additional samples above heightmap, to top of depth range

So the next step is to assign each of these samples to the nearest depth slice. Note that there isn't a depth slice at the very front of the depth range, since occlusion will always be zero there, so the four depth slices are positioned toward the back of the depth range. Anyway, after we assign the heightmap samples to the nearest depth slice, we take additional occlusion samples up to the top of the depth range.

AO Decals: Precomputing



Here's a diagram showing the heightmap samples in blue, now snapped to the nearest depth layer, and the samples in green are the additional samples we take above each heightmap sample up to the top of the depth range. We're also going fill in the samples below the heightmap, but I'll talk about that in a few more slides.

AO Decals: Applying

- Same as for AO Fields, adjusted to work on depth slices in 2D texture
- No direction, so equation is just:

$$AO = 1 - strength \times occlusion$$

Now when we apply the AO Decals in real-time, it's almost the same as for AO Fields; again, we draw a box and run a pixel shader to evaluate the occlusion at each shaded point. However, we now have no directional information, so the AO value becomes just 1 – strength * occlusion.

AO Decals: Applying

- Trick for linearly filtering samples packed into RGBA channels:

```
half4 deltas = half4(rgba.yzw, 0) - rgba;
half4 weights = saturate(depth*4 - half4(0,1,2,3));
half occlusion = rgba.x + dot(deltas, weights);
```

  - `rgba` is sample from decal texture
  - `depth` goes from 0 at back to 1 at front of depth range

Since we're packing depth slices into RGBA, we no longer get hardware filtering along the depth axis, as we did with the volume textures. So here's a snippet of shader code for doing that filtering. The idea is to calculate deltas from one depth slice to the next, then use the dot product to sum up the right amount of each delta.

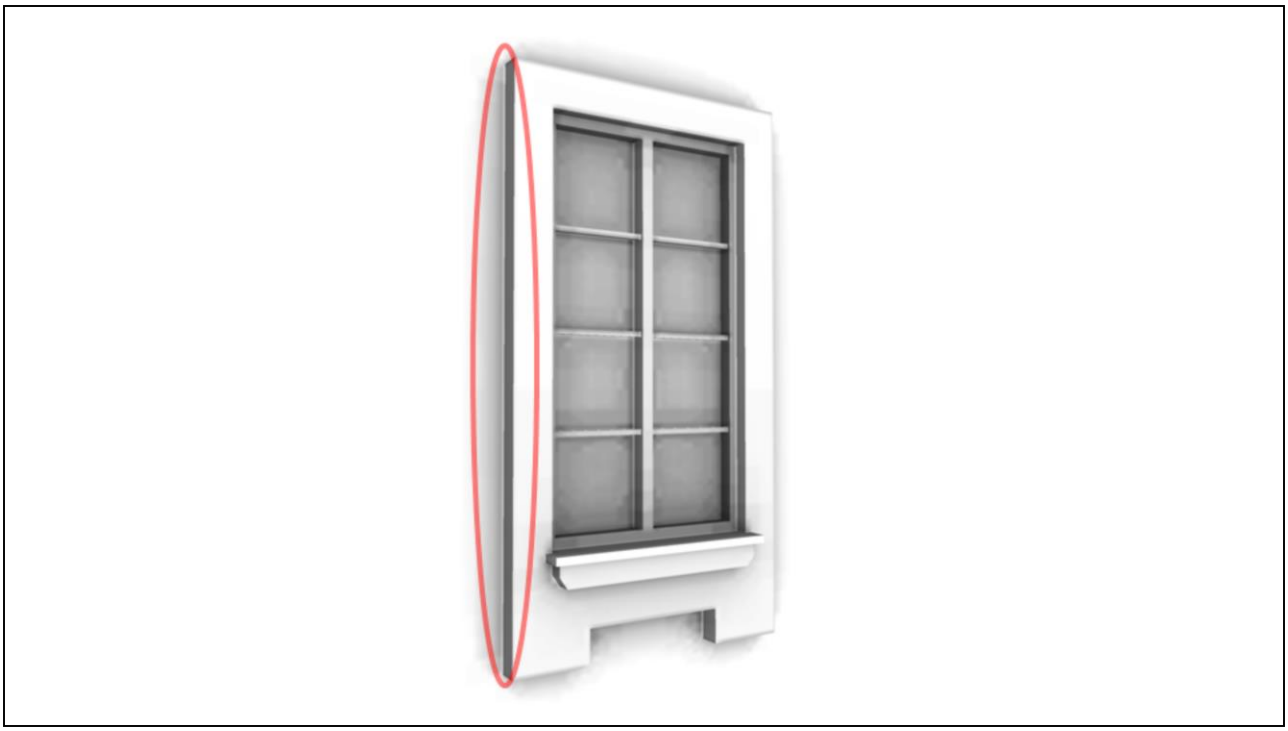
AO Decals: Details

- Bounding box size: same formula as for AO Fields
  - Used 0.7 epsilon instead of 0.25 (smaller boxes)
- Texture size: 64–128 texels on each axis
- Format: DXT5
  - Introduces noise, but in practice not noticable when combined with color/normal maps etc.
  - 4–16 KB per texture

The bounding boxes for AO Decals are sized the same way as for AO Fields, but in this case we used an even higher epsilon, of 0.7. It was particularly important for us to keep these boxes small because we often have an AO Decal on every window of a large building, and it's important to keep them small to keep performance under control.

The texture size for these is usually 64-128, and we use DXT5 compression. That introduces some noise, but it's at a small scale so it's much less objectionable than it would be for AO Fields, and is often hidden by noise in the color textures, normal maps, etc. At the end of the day, the textures are usually 4-16 K, so about the same size as for AO Fields.

An artifact that shows up with AO decals is that we get halos around height changes in the source geometry. Here's a screenshot of one such artifact; you can see a white line where the window frame meets the wall, and various similar white lines elsewhere in the image. This is very similar to the incorrect self-occlusion problem we had with AO fields. It's caused by bilinear filtering, since a height change can fall between two texels, and we end up blending occlusion values above the heightmap with those under the heightmap.



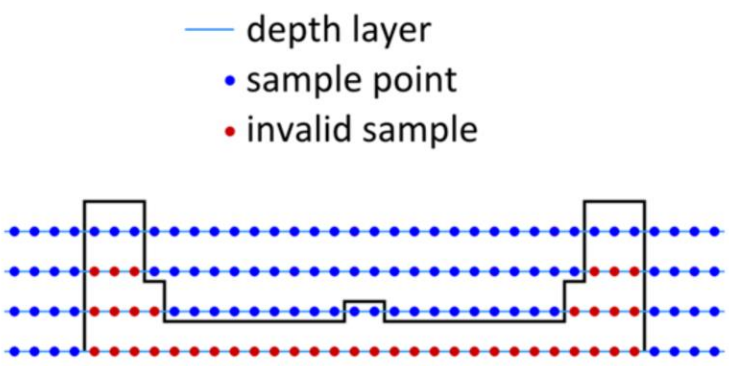
Here’s that artifact again, showing just the AO channel.

AO Decals: Halos Around Height Changes

- Solution:
  - During precompute, mark samples underneath the heightmap as invalid
  - Run a “dilation” step to propagate valid samples into adjacent invalid ones

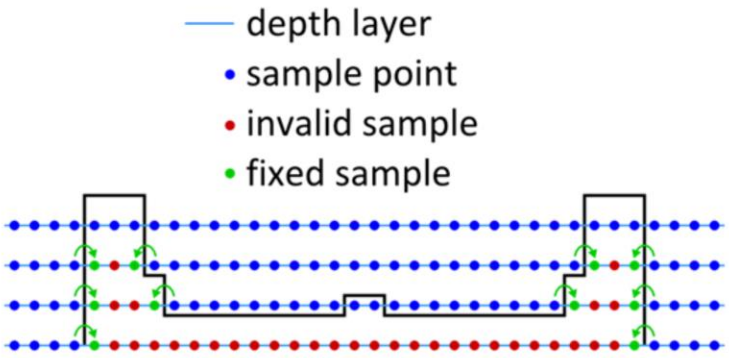
I mentioned earlier that we have special handling for sample points under the heightmap. During the precompute step, we mark all those points as invalid; then, once we’ve gotten all the other samples, we run a dilation step to propagate valid samples onto adjacent invalid ones.

AO Decals: Artifacts



Here's the heightmap again, with the valid samples in blue and the invalid samples (those under the heightmap) in red.

AO Decals: Artifacts



And here we have run one dilation step to copy valid samples onto adjacent invalid ones. We run a couple of iterations of this.

Before

Here's the AO channel before this fixup.

After

And here it is after the fix. It's a subtle issue, but you can see the halos around the window frame have disappeared.

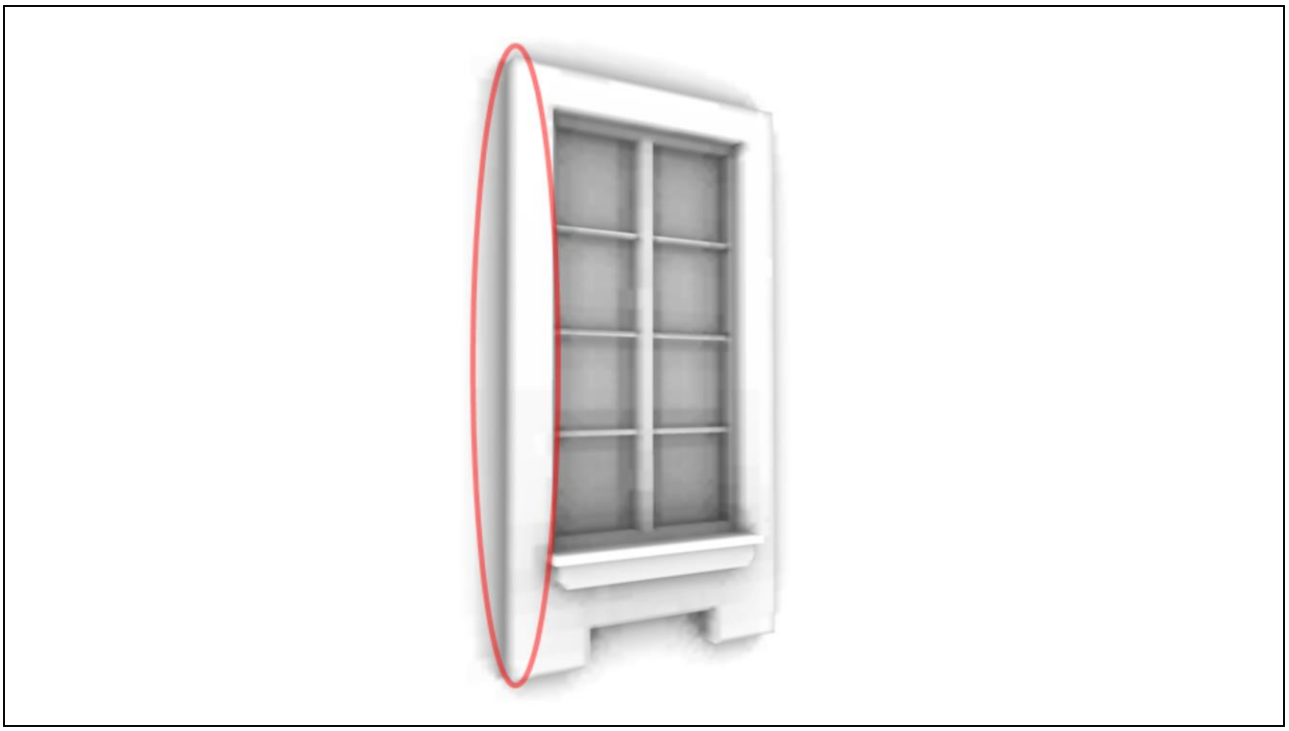
Before

Here's the full render before…

After

…and after.

The other artifact that we run into with this is that we get no occlusion from the wall onto the source geometry – only the other way around. This results in edges looking too soft and rounded, as seen here. The side of the window frame is at a 90-degree angle to the wall, but the edge is hardly visible because the AO is just blurry along that corner.



Here’s what this artifact looks like in the AO channel.

AO Decals: Edges Too Soft

- Solution: bake wall-occlusion term onto vertices

$$AO_{Wall} = 1 - strength \times \left(\vec{N} \cdot (-\vec{D}) \times 0.5 + 0.5\right)$$

  - Unit vector *D* is the direction the decal faces
  - Multiply this into any other per-vertex AO on the source geometry

Our solution to this actually doesn't modify the decal texture at all, but rather includes a wall-occlusion term in the baked per-vertex AO of the window. We calculate this as a dot product with the vertex normal and the direction the decal faces. This is based on the assumption that the wall is an infinite plane behind the object, so we don't need to know its exact placement. It's based on the same hemisphere-overlap calculation I showed for AO Fields earlier. This gets multiplied into any other vertex AO on the window.

Before

Here's the window before this term is applied…

After

…and after. The edges are much crisper and the shape of the whole window is better defined.

Before

Here's the full render before…

After

…and after.

Infamous 2 – Fields/Decals Memory Use

- 116 assets with AO fields or decals applied
  - Heavy reuse: 9604 instances of those assets throughout the game world
- 569 KB total texture data
  - Average 4.9 KB per asset
  - Not all loaded at once (streaming open-world game)

Now for some data. In Infamous 2 we had about 116 unique assets with AO fields or decals applied. That doesn't sound like many, but we reuse assets quite heavily, so there were actually *OVER NINE THOUSAND* instances of those assets throughout the game world. The texture data for these was only a little over half a megabyte in total.

Infamous 2 – Fields/Decals Performance

- Pixel-bound
- Typical frame draws 20–100 fields & decals
- Takes 0.3–1.0 ms on PS3
- Up to 2.3 ms in bad cases
  - Lots of fields in view, field covers the whole screen, etc.

As for performance, the shaders are usually pixel-bound, and in a typical frame of Infamous 2 there are anywhere from 20 to 100 fields and decals being drawn. These typically take 0.3 to 1 ms on PS3, although in occasional bad cases they can get up to 2.3 ms. Still, they're a fairly small part of the frame.

Future Enhancements

- Faster offline renderer – precompute is slow
  - AO Field: 512–4096 samples each
  - AO Decals: 16K–64K samples each
- Handle undersampling better for AO Fields
  - Current solution can introduce additional artifacts
- Try it on characters
  - A field on each major bone

There are some enhancements we'd like to make. The biggest complaint I get from artists and other programmers is that building AO fields and decals takes too long. Computing all the occlusion samples can take several minutes per asset, especially for decals, which tend to have many more samples than fields. Fortunately, there's some low-hanging fruit there, since our offline renderer is very simple and not particularly smart.

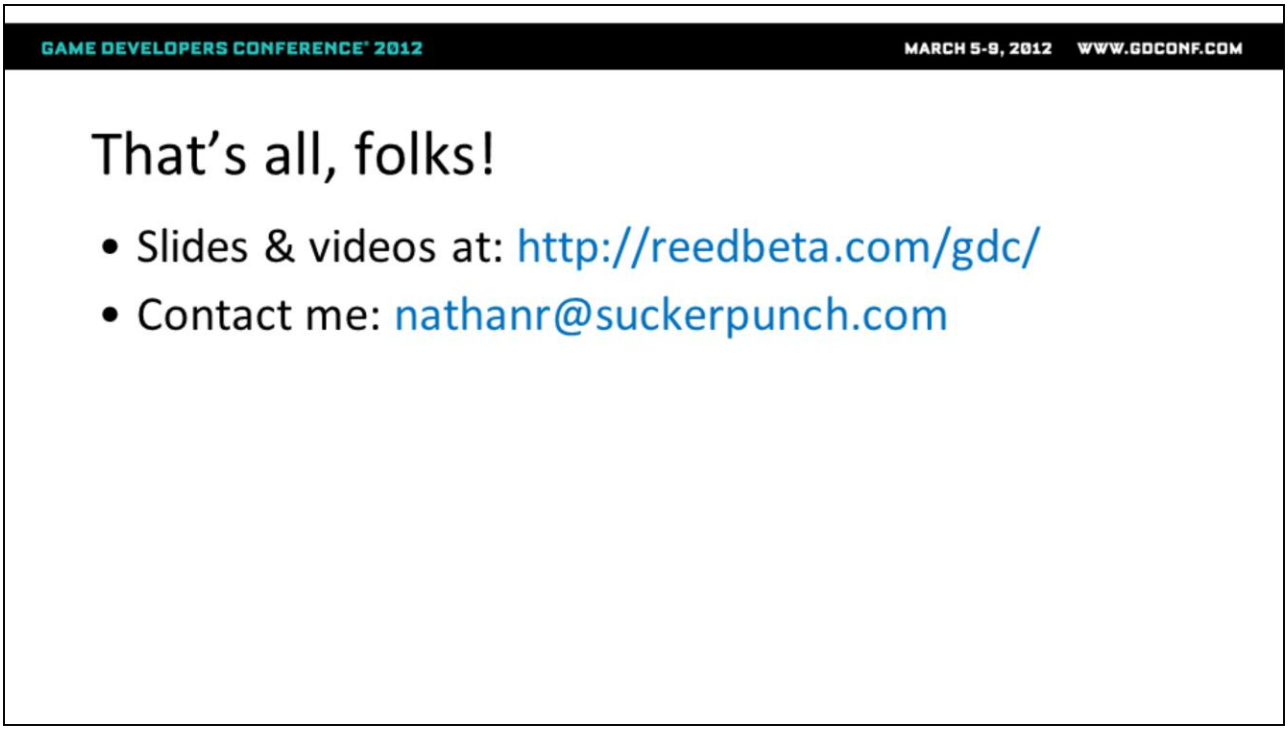
I'd also like to improve our treatment of undersampling artifacts. Currently, we're just kind of hacking around the undersampling problem by biasing samples along the normal vector, as I mentioned earlier. Supersampling the textures should help somewhat, although that will of course make things take even longer to build. And I'd like to find a good way to detect invalid samples (those enclosed inside geometry) in 3D for AO fields, like we do for AO decals.

Finally, I'd like to try using AO fields on characters. If you put a very low-res AO field on each major bone, you should be able to get dynamic AO between the character's arms and body, between the legs, etc.

Wrap-up

- AO Fields & Decals fill in the gap between baked AO and SSAO
  - Medium-scale occlusion
- More interesting & dynamic ambient lighting

To sum up, AO Fields and AO Decals represent a good way to fill in the gap between baked AO and SSAO and get that medium-scale occlusion that neither baked AO nor SSAO handles very well. They were certainly useful techniques for us to get more interesting and dynamic ambient lighting throughout Infamous 2 without spending too much performance or memory, and I hope I've given you some ideas for how to do the same in your own games as well.



Feel free to email me with any questions or comments.