

# Dynamic Coverage Anti-aliasing

Real-time graphics research



While conducting graphics research in early 2006 we tested a new method for anti-aliasing that provided a visual quality improvement comparable to 4x MSAA, but with about half the storage cost. In this presentation we'll walk through the basics of the approach, and discuss our handling of several challenging edge cases.

## The Problem



**Duke Nukem Forever**  
1152x640 (No AA)

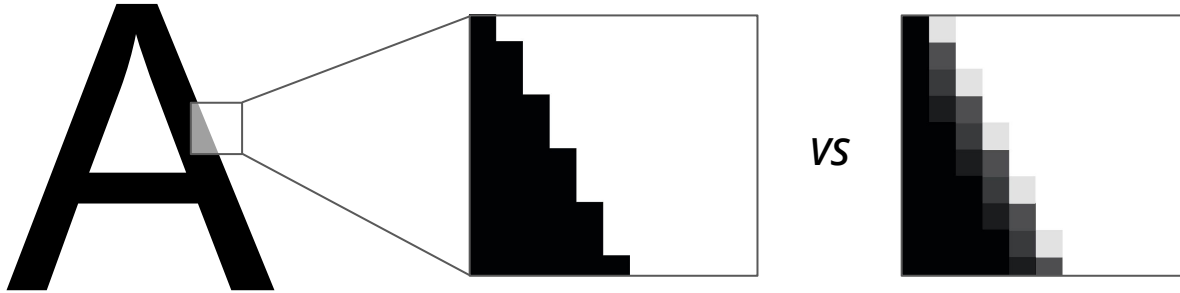
## The Problem



**Duke Nukem Forever**  
1152x640 (No AA)

## The Problem

Jagged lines along object boundaries in raster based graphics systems (*aka spatial discontinuities in high frequency visual data*)



**Lots of ways to solve this.** The simplest approach is simply increasing resolution (*if possible*). More advanced methods seek to smooth jagged edges where needed, while minimizing overall processing and storage costs.

### **Super-sample anti-aliasing:**

Render the scene at a higher resolution and then downsample using an intelligent filter. Significantly improves visual quality but carries a high cost (*typically 4x-16x the video memory, and 4x-16x additional shader processing*)

### **Multi-sample anti-aliasing:**

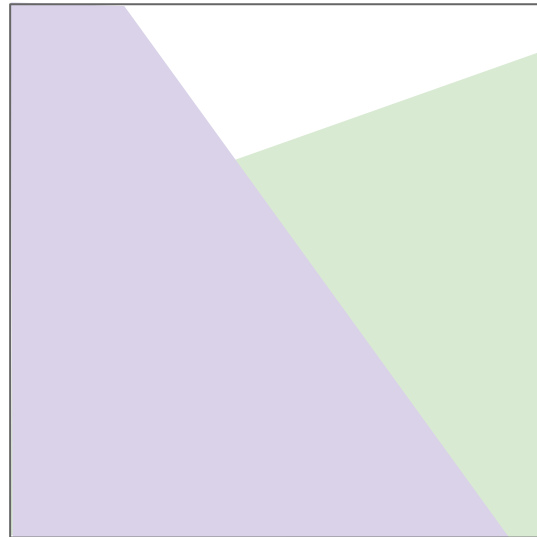
Render the scene at a normal resolution, but maintain multiple sub-pixel samples that can be blended together to produce a smoother final result. Significantly improves visual quality but also carries a high cost (*typically 2x-16x the video memory*)

## Walkthrough

Let's walk through a simple example of anti-aliasing.

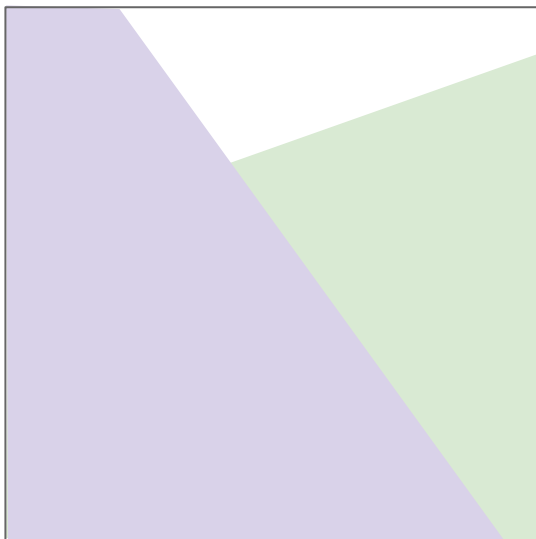
Two overlapping triangles intersecting our pixel.  
Purple triangle is *in front*, green triangle is behind.

**Let's assume that one color fragment is 16 bytes, and one depth fragment is 2 bytes.**



*our example pixel*

## Walkthrough

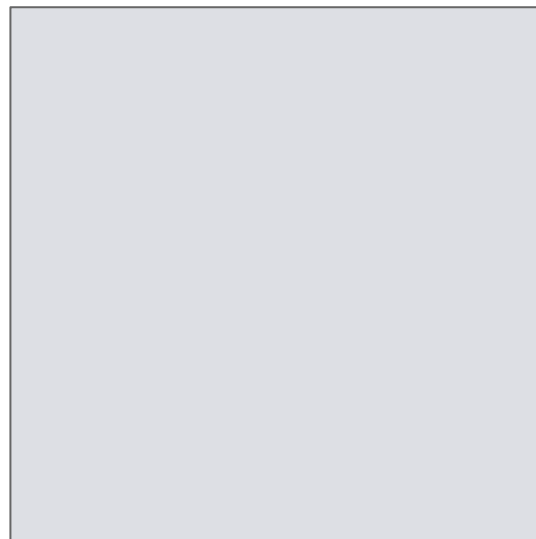


*our example pixel*

**Theoretically correct result**



If we precisely compute  
the contribution of each  
color *by area of coverage*



*rgb(221, 223, 228)*

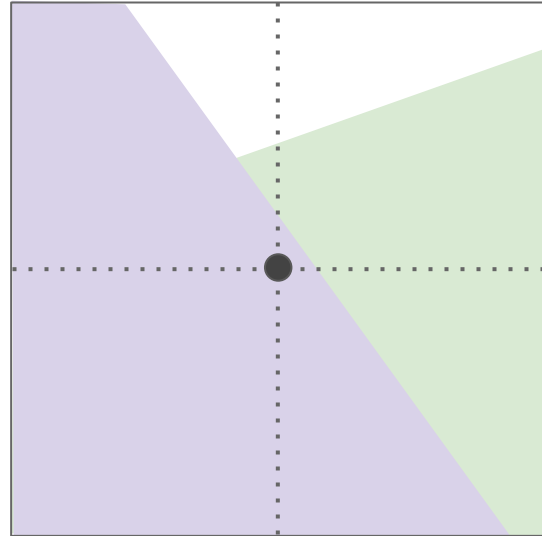


## No Anti-aliasing

Without anti-aliasing, the final pixel value is determined by the nearest object that overlaps the pixel center. **This will result in jaggies at sub-retina DPis**, but the storage and processing costs are minimal.

**Storage cost:** 18 bytes (1 color, 1 depth)

**Shader cost:** 1x pixel shader pass

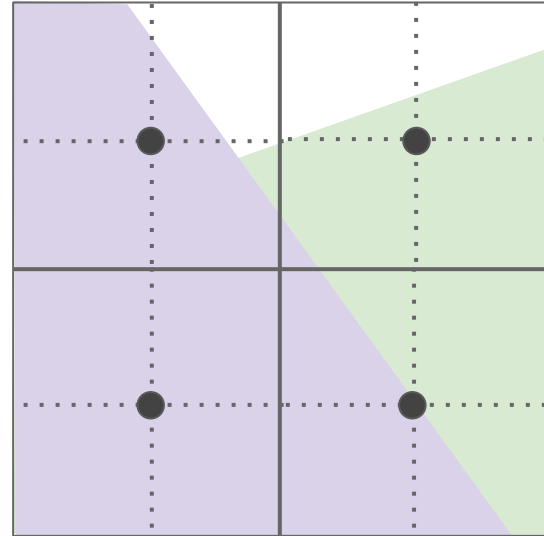


## Supersampling

4x supersampling effectively increases the resolution of the image by 4x. The final pixel value is determined by filtering 4 neighboring pixels down to a single value. The results are significant, but the storage and processing costs are great.

**Storage cost:** 72 bytes (4 color, 4 depth)

**Shader cost:** 4x pixel shader pass



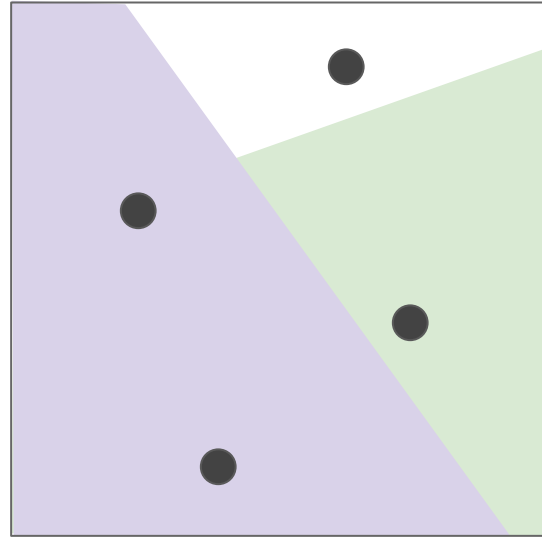
## 4x Multisample Anti-aliasing

With 4x multisample anti-aliasing, we record color and depth values at four *carefully selected* sample locations within the pixel. Depth is required for sample-level depth testing against incoming fragments.

Our shader is run only once, and then samples are updated based on coverage.

**Storage cost:** 72 bytes (4 color, 4 depth)

**Shader cost:** 1x pixel shader pass



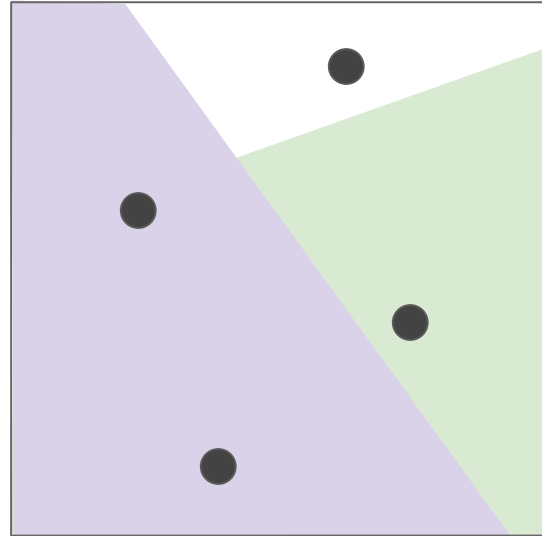
## 4x Multisample Anti-aliasing

### **Update phase:**

```
for each triangle that overlaps our pixel:  
  for each sub-sample within the pixel:  
    if sub-sample is covered by the triangle:  
      if triangle depth is < sub-sample depth:  
        update sub-sample color to triangle color  
        update sub-sample depth to triangle depth
```

### **Resolve phase:**

```
final pixel color = 0  
for each sub-sample within the pixel:  
  final pixel color += (sub-sample color) / sub-sample count
```



## Dynamic Coverage Anti-aliasing

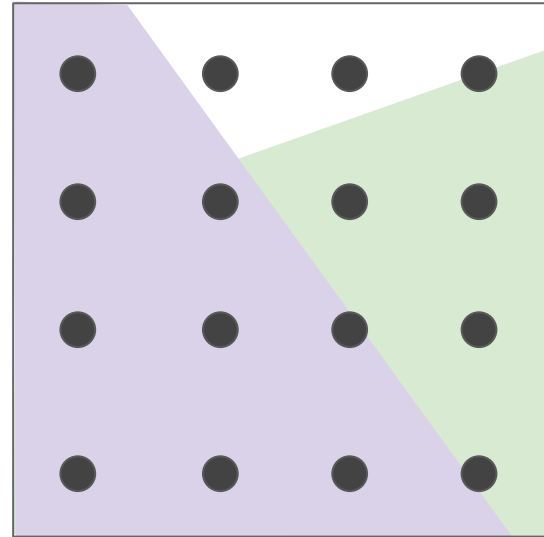
With dynamic coverage anti-aliasing, ***we maintain two color and two depth values that indicate the two fragments that most contribute to the final pixel value.***

We also record two 16 bit coverage masks that indicate the percent of coverage for each sample.

Similar to multisample antialiasing, our shader is run only once, and then samples are updated based on coverage.

**Storage cost:** 40 bytes (2 color, 2 depth, 2 masks)

**Shader cost:** 1x pixel shader pass



Note that in practice sample locations are carefully (not evenly) distributed

# Dynamic Coverage Anti-aliasing

## **Update phase:**

```
for each triangle that overlaps our pixel:
    IM = 0
    for each coverage sample within the pixel:
        if coverage sample is covered by the triangle:
            set IM[coverage sample index] = 1
    if ID is < SD0 or ID is < SD1:
        if SD0 < SD1:
            SC1 = IC, SD1 = ID, SM1 = IM
        else:
            SC0 = IC, SD0 = ID, M0 = IM
```

## **Resolve phase:**

```
if SD0 < SD1:
    output color =
        (SC0 * SMC0 + SC1 * (SM1 & ~SM0)) / TCC
else:
    output color =
        (SC1 * SMC1 + SC0 * (SM0 & ~SM1)) / TCC
```

For each pixel we maintain the following storage. At the start of the frame all masks are initialized to 0xFFFF, colors to clear color, depth to DEPTH\_MAX.

```
SC0 = sample 0 color,  SC1 = sample 1 color
SD0 = sample 0 depth,  SD1 = sample 1 depth
SM0 = sample 0 mask,   SM1 = sample 1 mask
```

When writing shaded fragments to our buffers we also define:

IC = incoming color, ID = incoming depth, IM = incoming mask

At the end of our frame we resolve our samples with:

```
TCC = number of unique bits set across SM0 and SM1
SMC0 = count of bits set in SM0
SMC1 = count of bits set in SM1
```

## Dynamic Coverage Anti-aliasing

### Update phase:

```
for each triangle that overlaps our pixel:
    IM = 0
    for each coverage sample within the pixel:
        if coverage sample is covered by the triangle:
            set IM[coverage sample index] = 1
            if ID is < SD0 or ID is < SD1:
                if SD0 < SD1:
                    SC1 = IC, SD1 = ID, SM1 = IM
                else:
                    SC0 = IC, SD0 = ID, M0 = IM
```

For each pixel we maintain the following storage. At the start of the frame all masks are initialized to 0xFFFF, colors to clear color, depth to DEPTH\_MAX.

```
SC0 = sample 0 color, SC1 = sample 1 color
SD0 = sample 0 depth, SD1 = sample 1 depth
SM0 = sample 0 mask, SM1 = sample 1 mask
```

If our incoming depth is closer than at least one of our cached samples, then we evict the farthest one and replace it with the incoming values (color, depth, mask)

mask

### Resolve phase:

```
if SD0 < SD1:
    output color =
        (SC0 * SMC0 + SC1 * (SM1 & ~SM0)) / TCC
else:
    output color =
        (SC1 * SMC1 + SC0 * (SM0 & ~SM1)) / TCC
```

At the end of our frame we resolve our samples with:

```
TCC = number of unique bits set across SM0 and SM1
SMC0 = count of bits set in SM0
SMC1 = count of bits set in SM1
```

## Dynamic Coverage Anti-aliasing

### **Update phase:**

```
for each triangle that overlaps our pixel:
    IM = 0
    for each coverage sample within the pixel:
        if coverage sample is covered by the triangle:
            set IM[coverage sample index] = 1
    if ID is < SD0 or ID is < SD1:
        if SD0 < SD1:
            SC1 = IC, SD1 = ID, SM1 = IM
        else:
            SC0 = IC, SD0 = ID, M0 = IM
```

For each pixel we maintain the following storage. At the start of the frame all masks are initialized to 0xFFFF, colors to clear color, depth to DEPTH\_MAX.

```
SC0 = sample 0 color,  SC1 = sample 1 color
SD0 = sample 0 depth,  SD1 = sample 1 depth
SM0 = sample 0 mask,   SM1 = sample 1 mask
```

When writing shaded fragments to our buffers we also define:

IC = incoming color, ID = incoming depth, IM = incoming mask

### **Resolve phase:**

```
if SD0 < SD1:
    output color =
        (SC0 * SMC0 + SC1 * (SM1 & ~SM0)) / TCC
else:
    output color =
        (SC1 * SMC1 + SC0 * (SM0 & ~SM1)) / TCC
```

At the end of our frame we resolve our samples with:

Output equals the weighted average of our cached samples, with weights computed by sample coverage of the pixel



A whiteboard with a silver frame and a black eraser at the bottom center. The text is written in a black, hand-drawn, monospace-style font.

WHITEBOARDING TIME

(LET'S EXPLORE EDGE CASES)






## ***What cases did you discuss?***

Partial triangle overlap of a pixel without any coverage?

Thin triangle overlap with no majority coverage?

Many triangle overlaps with equal coverage?

## Summary

Method	Storage costs	Pixel shader costs	Output color
No AA	18 bytes (1x color, 1x depth)	1x pixel shader passes	 <i>rgb(217, 210, 233)</i>
4x Supersample	72 bytes (4x color, 4x depth)	4x pixel shader passes	 <i>rgb(217, 216, 228)</i>
4x Multisample	72 bytes (4x color, 4x depth)	1x pixel shader passes	 <i>rgb(227, 227, 233)</i>
2x Dynamic coverage	40 bytes (2x color, 2x depth, 2x masks)	1x pixel shader passes	 <i>rgb(217, 219, 225)</i>
		<b>Theoretically correct value:</b>	 <i>rgb(221, 223, 228)</i>