# Antialias Sample

Collapse All

Multisampling attempts to reduce aliasing by mimicking a higher resolution display using multiple sample points to determine each pixel's color. This sample shows how the various multisampling techniques supported by your video card affect the scene's rendering. Although multisampling effectively combats aliasing, under particular situations it can introduce visual artifacts of its own. As illustrated by the sample, centroid sampling seeks to eliminate one common type of multisampling artifact. Support for centroid sampling is supported by the pixel shader 2.0 model and later.

**Note**

To better see the subtle pixel-level details explored by this sample, it might be helpful to zoom in on the screen with the magnifier tool. To launch the tool, type magnify in the Run dialog.

## Path

| | |
|---|---|
| **Source** | *SDK root*\Samples\C++\Direct3D\Antialias |
| **Executable** | *SDK root*\Samples\C++\Direct3D\Bin*x86 or x64*\Antialias.exe |

## Aliasing and Antialiasing

Raster displays use a finite number of pixels to display the scene; the greater the display resolution, the more accurately the scene can be represented. Visible artifacts exist in scenes where pixels can not adequately represent high-frequency data. This is most apparent at mesh edges, where straight-line data is approximated by a fixed number of pixels, creating a stairstep pattern; this form of aliasing is often called "jaggies," and becomes especially apparent when the polygon is in motion along the screen. Figure 1 shows aliasing in the sample's Triangles scene when multisampling is disabled.

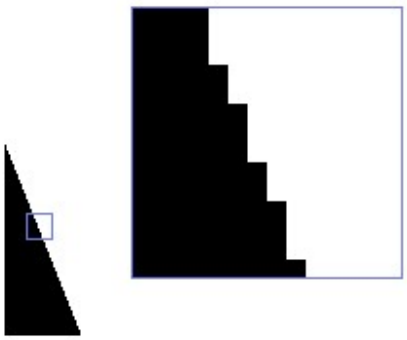

Figure 1: Stairstep pattern on polygon edges is a common type of aliasing artifact.

Aliasing is a natural consequence of rasterization. For a review of raster displays and polygon rasterization, see **Directly Mapping Texels to Pixels**. As long as individual pixels are large enough to be distinguished from neighboring pixels, then there is no perfect way to eliminate aliasing. Ultimately, it is best to make pixels so small and so close together that aliasing can't be noticed. Be aware that inkjet printers also produce raster output, but it's nearly impossible to find any aliasing on a printout. This is because the common resolution for a square inch of paper is 300x300 dots per inch (dpi). The common resolution for a square inch on a computer display is 72x72 pixels. Comparatively, there are more than 17 times as many dots on a printed page than on a monitor. Of course, the other way to make the pixels appear smaller is to increase your viewing distance from the monitor. This appears to make the "jaggies" disappear once you back up far enough. At a certain point, you would be unable to distinguish between a cluster of 4 pixels and a pixel that was 4 times larger but had the same average color.

Following this logic, one way to combat aliasing is with oversampling. Oversampling is a technique that renders the scene at a higher resolution than the display resolution. For example, if your display resolution is

800x600 pixels and you render the scene at 1600x1200, each pixel is responsible for displaying a 2x2 region of the scene texture. The best way to approximate a 2x2 region is the mathematical average of all 4 colors. It has the advantage of noticeably reducing aliasing artifacts in your scene; however, it requires a larger back buffer and, therefore, a higher fill cost. Oversampling is not implemented by Direct3D (nor is it recommended due to the disadvantages just mentioned).

Another technique for combating aliasing is multisampling. Multisampling reduces the overhead of oversampling, by emulating the sub-pixel averaging behavior without actually computing each sub-pixel's color.

Rasterization compares the location of the edges of a polygon against the location of the center of a pixel (the sampling point). If the polygon covers the center of the pixel, then the polygon determines the pixel's color. Multisampling extends rasterization by using a pattern of sampling points instead of the pixel's center (see Figure 2).

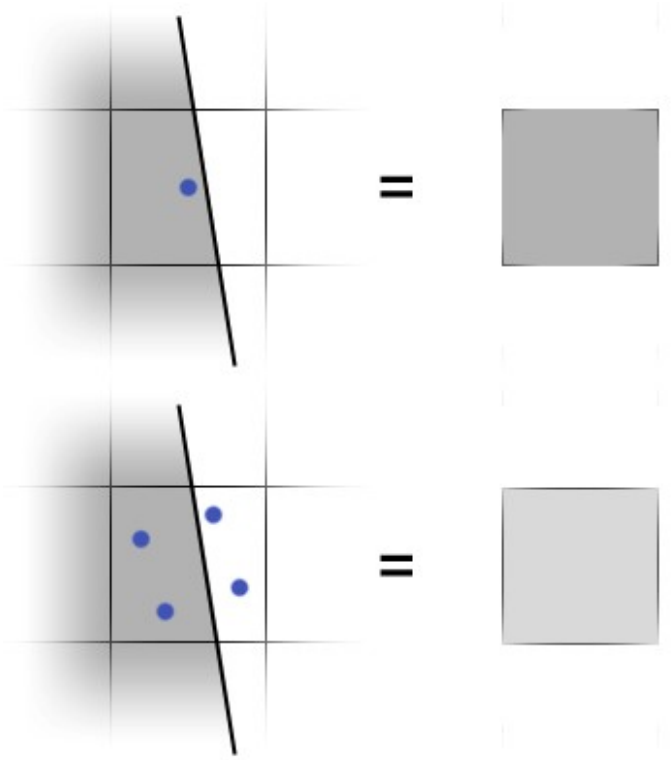

Figure 2: Single point-sample pixel color vs. four-point multisampled pixel color.

During multisampling, the final pixel color is a combination of the sampled pixel color and the number of samples that are covered by the polygon. In the top half of Figure 2, the polygon covers the single sampling point so the pixel color is 100 percent of the polygon's color. In the bottom half of Figure 2, only two of the four sampling points are covered by the polygon, so the polygon color contributes 50 percent of the pixel color.

During multisampling, the rasterization process tests multiple sample points inside each pixel so it might seem that multisampling is just as expensive as oversampling. However, multisampling doesn't run the pixel pipeline at every sampling point. Instead, a single pixel sample determines the color at the pixel center, and the percentage of sampling points covered by the polygon is multiplied by the pixel color. The result is that multisampling increases the accuracy of the pixel color without increasing the amount of sampling that is necessary by oversampling. Figure 3 shows how multisampling greatly reduces the "jaggies" seen in the same scene from Figure 1:

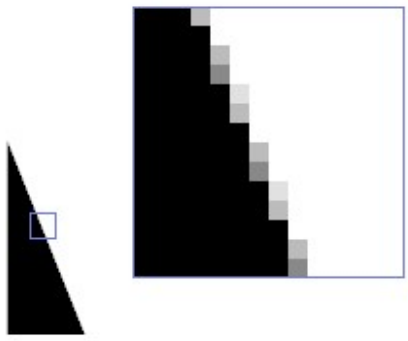

Figure 3: Multisampling reduces the noticeability of aliasing.

One important point to note is that Direct3D does not specify the sampling point patterns, only the number of sampling points (as shown in Figure 4). That means that the exact same scene under the exact same multisampling settings could appear slightly different depending on the graphics card used.

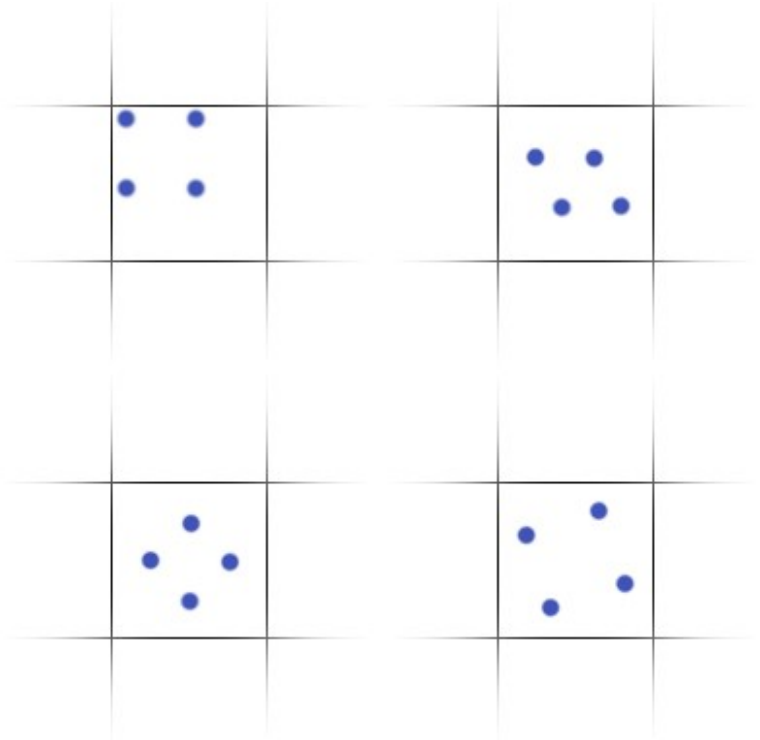

Figure 4: Multisampling patterns from various vendors.

Direct3D does not define the layout of sample points, only the number of sample points; hardware vendors decide patterns for supported multisampling modes. A selection of valid 4-sample multisampling patterns is shown above.

## Centroid Sampling

As mentioned earlier, a key point to the efficiency of multisampling is that the pixel pipeline is only run once per pixel per polygon. The screen location used to determine the polygon's color is typically the center of the pixel, but this can lead to a problem: if a polygon covers one or more sample points but does not cover the pixel's center, the polygon's color will be determined at a point which does not actually lie on the polygon (see Figure 5):

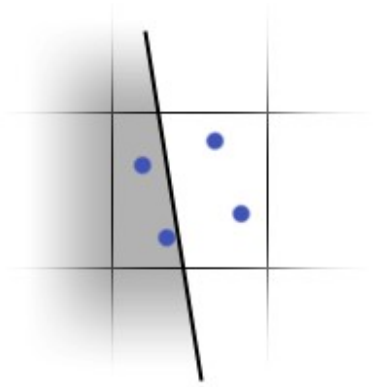

Figure 5: The polygon covers two of the four sampling points and will therefore contribute 50 percent of the pixel's color.

Normally, this isn't a problem; however, if the polygon is textured you should be aware that the sampled texels could lie outside of the polygon's UV boundaries. Developers often pack several small textures onto a large texture, sometimes called an "atlas" texture, in order to minimize texture changes (for example, light maps are often stored this way). If the atlas texture features high-contrast differences, sampling at the pixel's center can introduce artifacts at the polygon edges. This is most visible when the polygon is rotated to be parallel with the view vector. Stated another way, if the polygon is at a high glancing angle, then the iterated texture coordinate at the pixel's center could be well beyond the polygon's UV boundaries.

Try running the sample's Triangles scene with the multisampling type set to **D3DMULTISAMPLE_NONE** and texture filtering set to **D3DTEXF_POINT**. You should not be able to see any non-white pixels under the Texturing Artifacts label. Now enable multisampling and notice that the outline of a triangle appears, as shown in Figure 6:

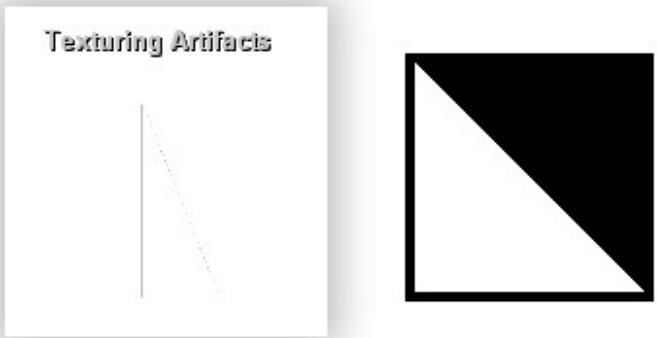


Figure 6: Example of multisampling texturing artifacts (left) when rendering a triangle with this texture (right).

The left side of Figure 6 shows the texturing artifacts that can happen when texture coordinates get interpolated beyond the polygon's UV boundary. In this case, the polygon covers two of the four sampling points but does not cover the pixel's center. The right side of Figure 6 shows the applied to the triangle that highlights the texturing artifacts. Black texels are outside the triangle's UV coordinate boundaries. The triangle is textured such that texels that are contained or crossed by the triangle's UV boundaries are white, and all other texels are black; therefore, the triangle will be invisible against the white background except where texels beyond the UV boundary are being sampled. Note that it's normal for linear and anisotropic filtering to sample from texels outside the UV boundary; this follows directly from how bilinear filtering works.

The texturing artifacts are visible when using point filtering with multisampling because multisampling extends the rasterized area of the triangle to include all pixels in which sampling points are covered, even when the pixel center is not. The solution to this problem is centroid sampling, which adjusts the position used for determining polygon color to be the center of all the sampling points covered by the polygon. The solution to this problem is centroid sampling, which is shown in Figure 7:

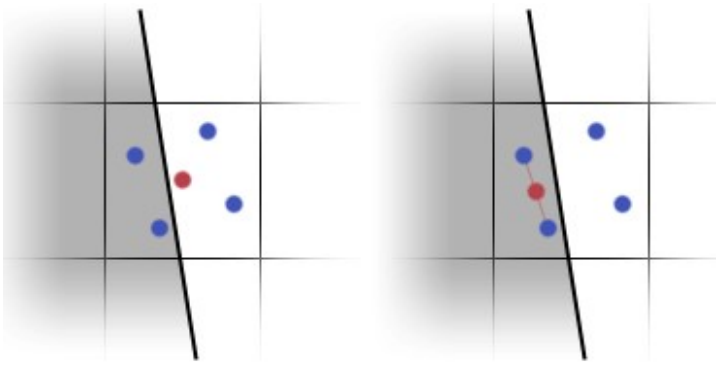

Figure 7: Multisampling versus centroid multisampling.

Figure 7 contrasts a four-point multisampling pattern (on the left) with a four-point centroid multisampling pattern (on the right). Polygon coverage is indicated by the gray shaded region. The four-point sampling pattern is shown with blue dots and the texture sampling location is the red dots.

When centroid sampling is enabled, the sampled pixel position used for determining polygon color is adjusted to be the center of the sampling points covered by the polygon. This guarantees that a centroid-sampled location will always lie within the polygon being rendered. Place a check in the Centroid sampling checkbox and notice that the Texturing Artifacts label is again blank, indicating that centroid sampling is constraining the iterated texture coordinates to the triangle's UV boundary. Centroid sampling is automatically enabled for each pixel shader input that has a color semantic (see **Semantics (DirectX HLSL)**). Alternately, you can enable centroid sampling on any HLSL pixel shader input by appending the centroid semantic as shown here:

```
//--------------------------------------------------------------------------------------
// Texture - Point sampled (centroid)
//--------------------------------------------------------------------------------------
float4 TexturePointCentroidPS( float4 TexCoord : TEXCOORD0_centroid ) : COLOR0
{
    return tex2D( PointSampler, TexCoord );
}
```

Similarly, to enable centroid sampling within an assembly shader, append the centroid semantic to the declaration:

```
dcl_texcoord0_centroid v0
```

Be aware that centroid sampling does not influence pixel rate-of-change calculations. This means that mipmap levels are still chosen from rate-of-change calculations at pixel centers. Also note that centroid sampling should primarily be used with atlas textures as described above; under the typical texture usage where the entire texture maps directly to the mesh, centroid sampling will actually introduce slight texturing errors.

The sample's Spheres scene and the sample's Quads scene contain low-tessellated and high-tessellated meshes textured with a checker pattern. Try playing with various multisampling, filtering, and centroid sampling settings to see how the adjustments influence the scene.

## References


- Mitchell, Jason. DirectX 9 High Level Shading Language. Siggraph 2004 presentation.
- Burrows, Mike et al. Advanced Visual Effects with Direct3D. Game Developers Conference 2004 presentation.