High Quality Antialiasing

Tristan Lorach
tlorach@nvidia.com

Month 2007

Document Change History

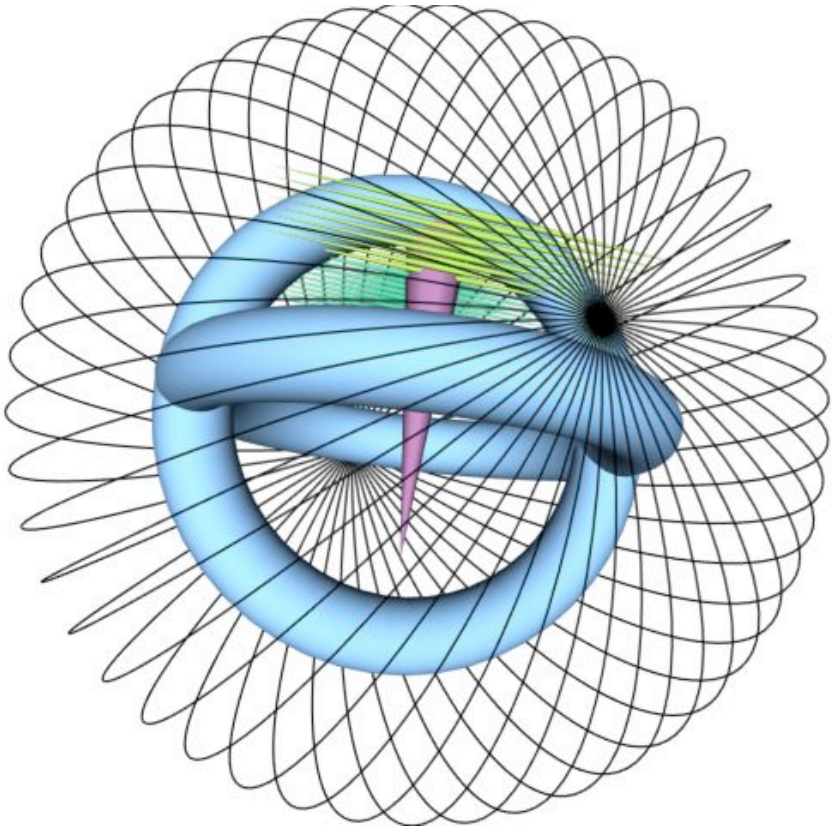
Version	Date	Responsible	Reason for Change
1	01/11/07		Initial release

Abstract

This sample shows different ways of performing anti-aliasing - both by using only the native hardware AA support, and by mixing the hardware modes with additional supersampling. There are various ways in which the supersampled image can be down-sampled. The way we do the downsampling in this example is the same technique that was used in 2 of our latest launch demos – "Froggy" and "Adrianne".

For any details related to hardware CSAA/MSAA mode, please visit our web page at :

http://developer.nvidia.com/object/coverage-sampled-aa.html



NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

Motivation

The NVIDIA *demo team* use many interesting techniques, but people often wonder how these are implemented.

This sample shows how Alexei Sakhartchouk and Eugene D'Eon integrated antialiasing in their respective demos 'Froggy' and 'Adrianne'.

The purpose of this sample is to:

- Show how to activate different hardware MSAA/CSAA modes in OpenGL
- Show how to add on top of this another technique to improve antialiasing : supersampling and filtering

How Does It Work?

For the section related to MSAA/CSAA modes, please refer to the document http://developer.nvidia.com/object/coverage-sampled-aa.html

Next we will discuss how to add supersampling and how we can downsample it.

Simple downsampling

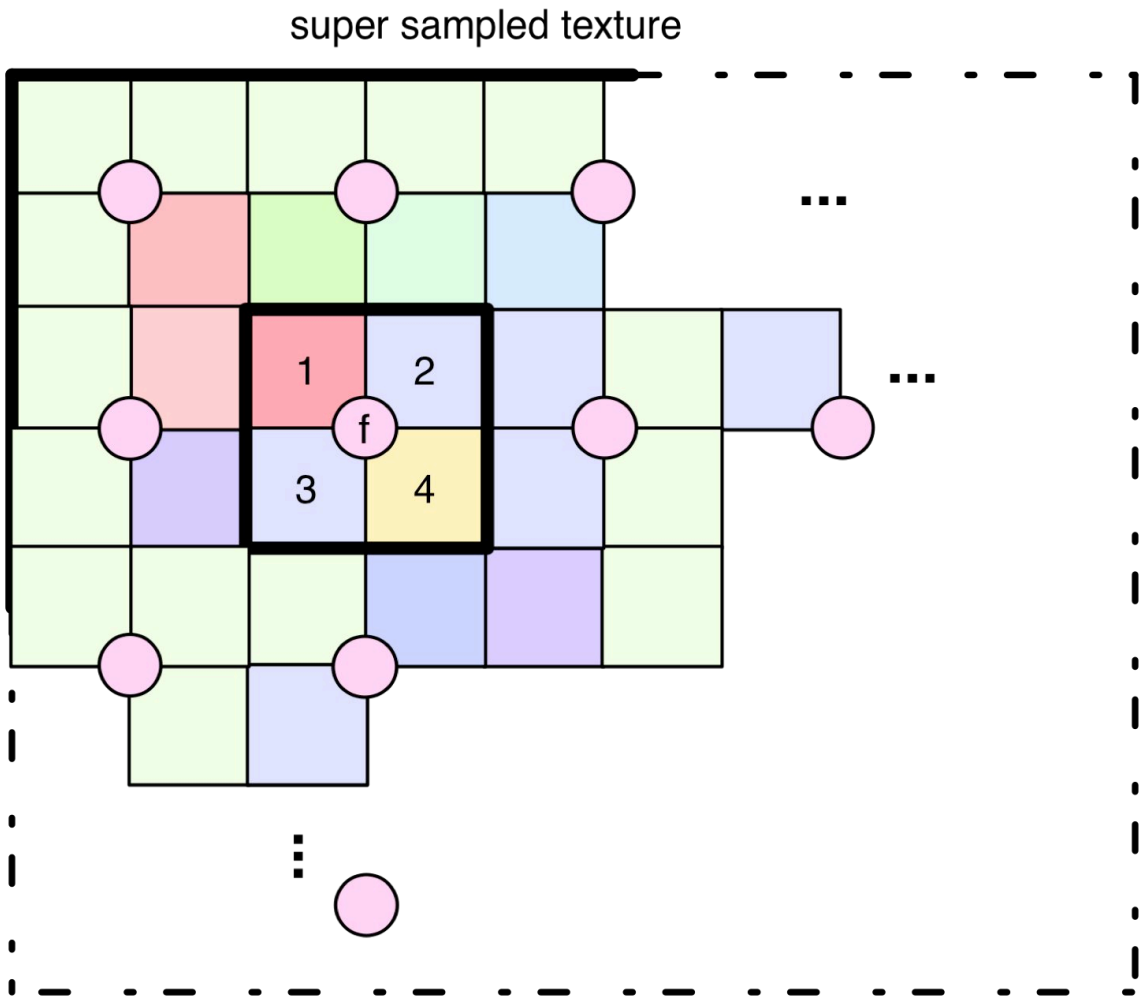


Figure 1

The simple downsampling is just using the hardware bilinear filtering : in this figure, you can see that sample f will be the average of texels 1,2,3 and 4.

Downsampling with Filtering

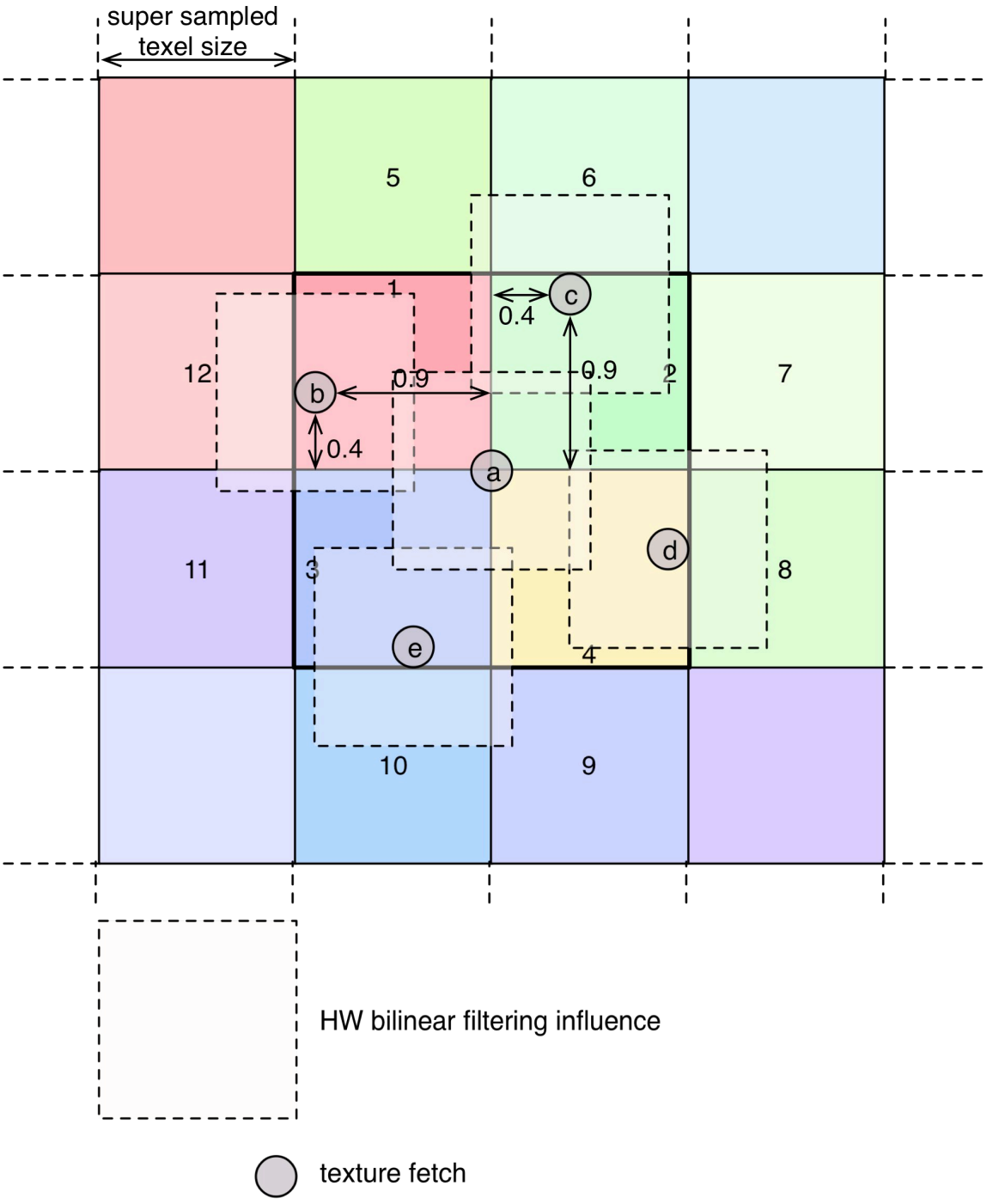


Figure 2 : kernel 1

The second technique is down sampling the texture but also adds a filter by fetching additional samples (b,c,d,e).

The resulting color will be the average normalized color of these 5 samples.

Note that this filter is giving slightly more weight to the center of the kernel : 'a' sample is helping, but also other samples are getting more from texels in the center (1,2,3,4) than from texels around (5,6,7,8,9,10,11 and 12).

Note also that we used a rotated grid to avoid artifacts coming from vertical or horizontal boundaries.

More Filtering

The last technique is mixing two kernels : the previous one with another larger.

Here is how the larger kernel looks :

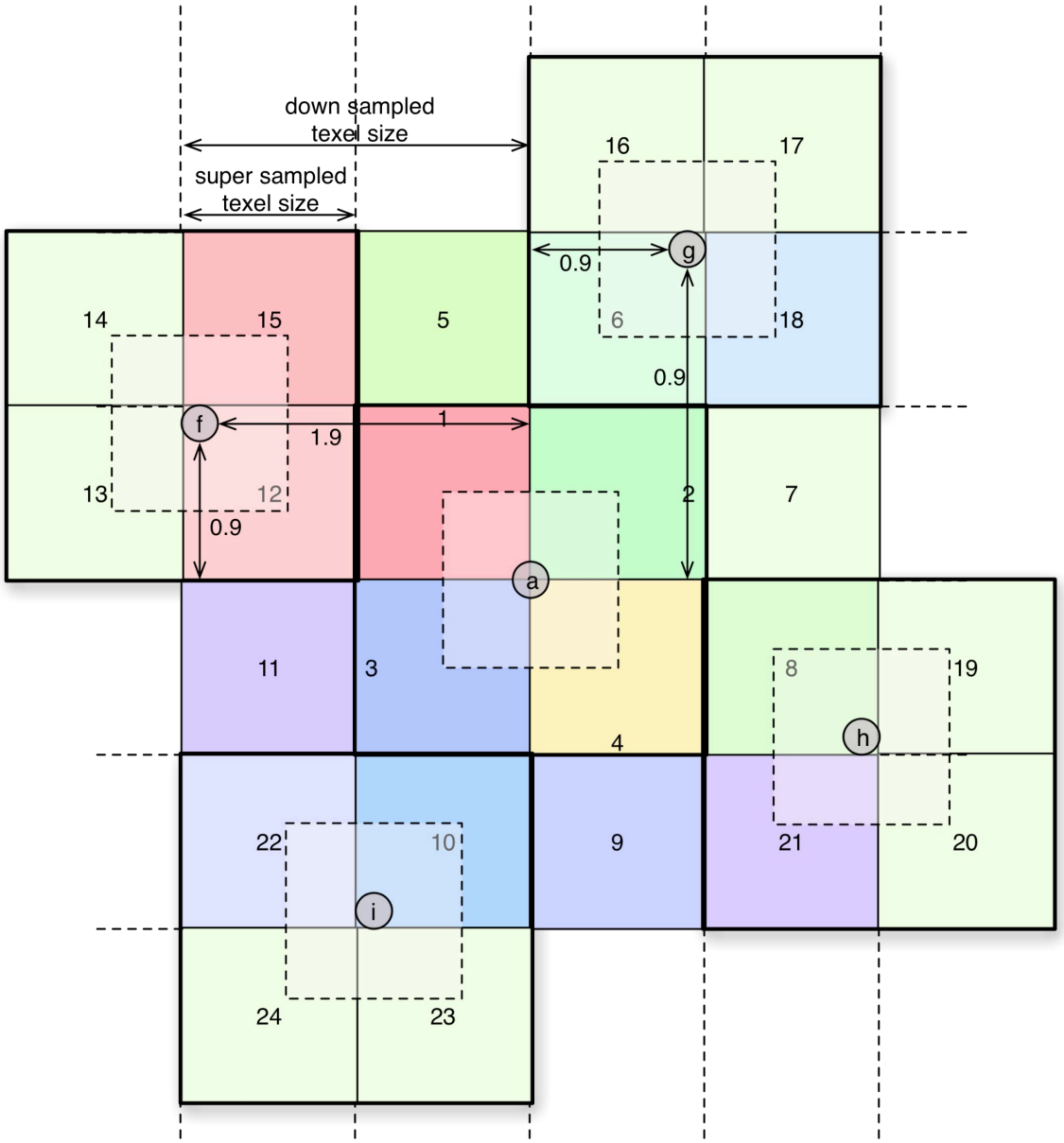


Figure 3 : kernel 2

This larger kernel is using the hardware to filter 2x2 areas that aren't intersecting at all. Then these 5 separate areas are averaged to a final color value.

The combination of these two kernels (fig 2 and fig 3) is made through a *lerp* of the 2 respective filtered results. The lerp will balance between those two filtered values depending on the alpha value we filtered with kernel 1 (fig 2)

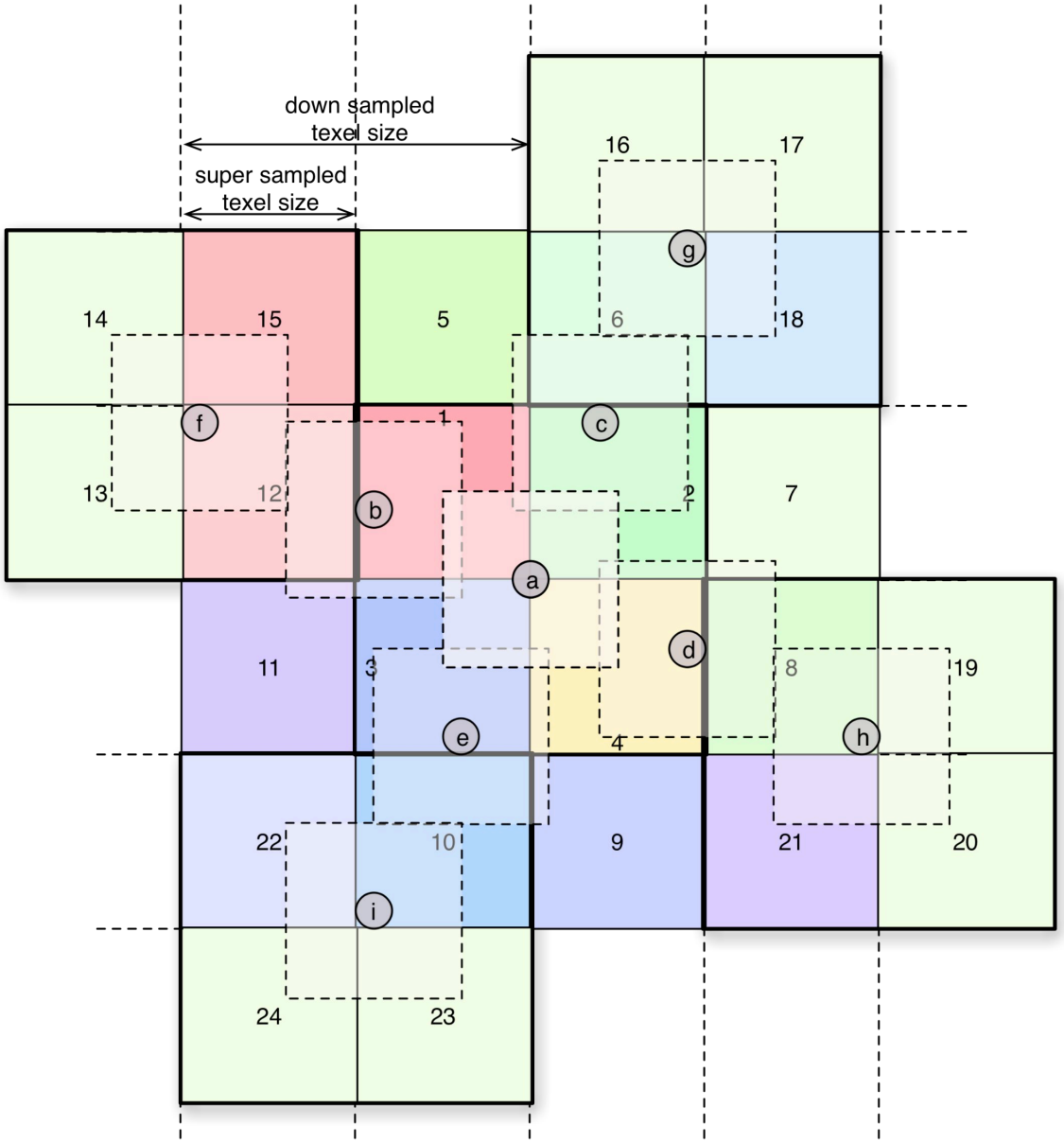


Figure 4 : mix of the 2 kernels

Using a larger kernel can be useful when some sharp objects (i.e high frequencies in color changes) are spread around the model. Sharp objects can easily lead to aliasing artifacts.

A good example is hair and eye lash rendering. Adrianne's hair and lashes are using this mix of 2 kernels by setting the alpha value to 0, while the rest of the body kept alpha=1. As a consequence hair is softened with a larger area, thus adding the feeling that these areas are fluffy.

Issues Related to Using Larger Kernel

However, there is a situation where using a larger kernel depending on alpha does not give the proper result - displaying an abject with alpha=1 on top of objects with alpha=0. For example in Adrianne demo, when we display an earring in front of the hair, the ring is averaged with the hair and will look blurry.

If you consider the blue area (alpha = 0) as the equivalent of Adrianne's hair and the green lines (alpha = 1) as the earring, here is what we get :

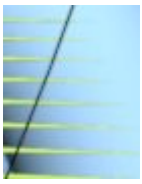


Figure 5

Compared to the version with no kernel depending on alpha :

Figure 6

In figure 5, the green lines and even the black one are somehow blurred by the background in blue (alpha = 0).

Note that the sample is doing the opposite and correct way : the blue object (torus) has alpha = 1 while thin objects have alpha = 0.

Running the Sample

The sample has different menu items and key bindings that allow you to compare different modes :

- '1','2','3' : choose down sampling modes
  - 1 : simple down sampling
  - 2 : down sampling with kernel filtering
  - 3 : down sampling with 2 kernel filtering
- 'a' : toggle between simple OpenGL MSAA (4x - no supersampling or specific MSAA/CSAA) and other more sophisticated AA (CSAA, supersampling and filter pass)
- 'c' : preset to set the application in our 16x CSAA mode.
- 's' : preset to set the application in 16x CSAA mode with supersampling and 2 kernel filtering on top of it

Note that if your driver doesn't have CSAA, the sample will fall back to MSAA 4x instead.

Performance / Results

Our native CSAA is very efficient and provides high quality results. CSAA 16x is slightly more expensive than MSAA 4x. For more details you may want to refer to http://developer.nvidia.com/object/coverage-sampled-aa.html

Drawing the scene to super sampled buffer and processing it through our 3 different down sampling techniques has a cost. You should be since this technique renders 4 times more fragments than the original technique. This means that expensive shaders could be a problem.

However, if you are rendering simple shaded geometries (like in CAD/DCC applications), the fill rate may not be as much of an issue.

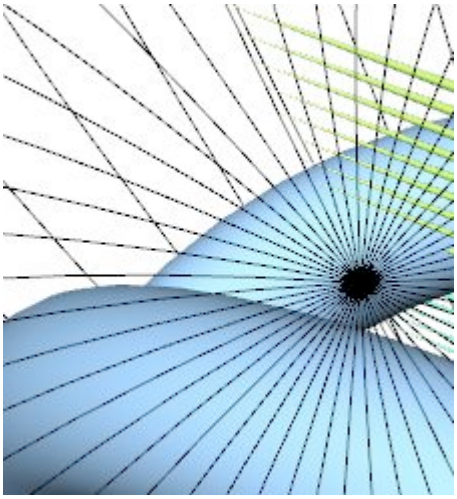
Note that both the "Froggy" and "Adrianne" demos used this AA combination with complex shaders and everything worked well.

Comparisons

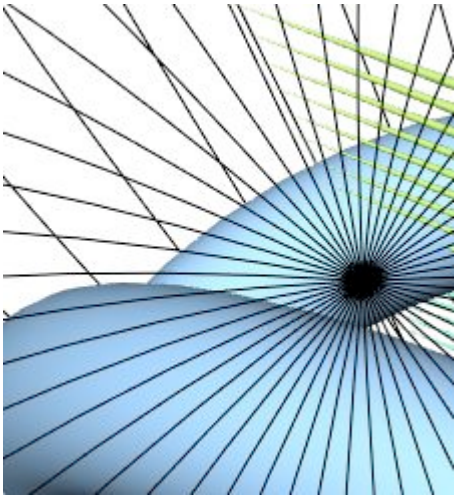
Here are different screen captures to show the differences between MSAA 4x, CSAA 16x and CSAA 16x combined with the use of supersampling and filtering.

The next set of pictures show the differences between the 3 techniques we used to down sample (and filter) the FBO.

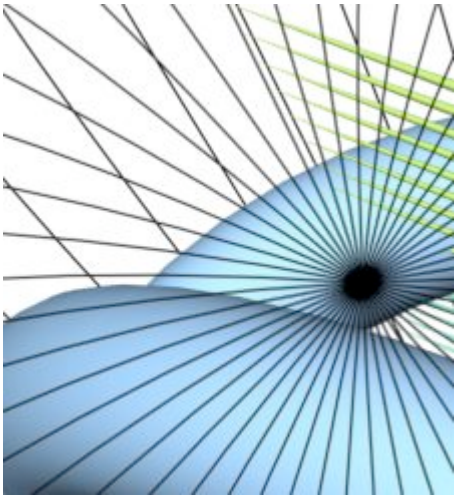
- MSAA 4x. Basic mode from glut initialization :



- CSAA 8x

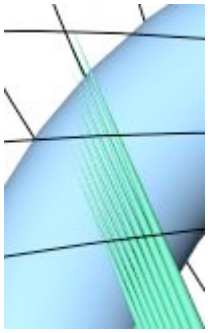
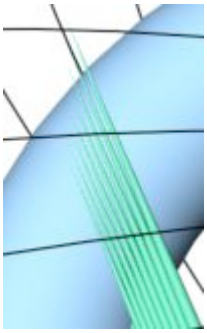
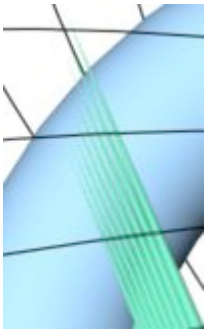


- CSAA 8x + Supersampling AA using 2 kernels for downsampling and filtering :



Downsampling techniques

These 3 samples are usin CSAA 16x

a :  b :  c: 

From left to right :

a) Simple down-sampling using only hardware bilinear filtering
b) Down sampling using 5 tap filtering
c) Down sampling using 2 5 tap filtering. You can see that the thin green cones (having alpha=0) are blurry. This is what the kernel #2 is doing

Conclusion

Choosing the correct technique for antialiasing is really dependent on which kind of application you are running. The additional super-sampling technique described here is not practical in all situations.

Some CAD applications may be particularly interested in the more sophisticated AA techniques described here. The reason is that professional / industrial applications often use lines, wireframe and very thin and high-contrast primitives. Furthermore, clients are very sensitive to image quality and will not tolerate even small artifacts.

Other applications such as games may not get such benefit from complex antialiasing. Developers should make their own decision - this may depend on art assets - the 3D models; the colors and the default resolution. In most of the cases, CSAA would be the best tradeoff for games - for a little additional cost compared to MSAA 4x, the result is very good.

Notice