NEW ANTI-ALIASING AND DEPTH OF FIELD TECHNIQUES FOR GAMES

RICHARD CANT, NATHAN CHIA, DAVID AL-DABASS

Department of Computing and Mathematics
The Nottingham Trent University
Nottingham NG1 4BU.
Email: richard.cant/david.al-dabass@ntu.ac.uk

KEYWORDS
Computer Graphics, Anti-aliasing, super-sampling, depth of field, focus, Open GL.

ABSTRACT

We describe software techniques that will enable Open GL capable graphics cards to implement antialiasing and depth of field effects in software. The methods allow any hardware facilities that are available on the graphics card to be used to improve performance but do not require hardware support in order to work.

INTRODUCTION

Sophisticated graphical and optical effects have in the past been the preserve of pre-rendered animation sequences taking hours or even days to calculate. In other cases these effects were incorporated in real time systems but only in very expensive military simulators, Potmesil and Chakravarty 1981, Cant and Sherlock 1987, Montrym et al, 1997. Since that time high end graphics workstations have also incorporated these techniques, e.g. Silicon Graphics. However recent advances in technology suggest that some of these effects should now be considered for real time implementation even on relatively low cost systems, such as PCs and games consoles. In this paper we will explore the possibilities of implementing some of these features by making use of existing facilities in hardware 3-D accelerators via OpenGL.

ANTI-ALIASING

For the implementation of anti-aliasing, this paper will attempts to replicate nVidia's quincunx anti-aliasing (which is built into the hardware of an expensive GeForce 3 card) Figure 1, by using the existing hardware calls of common 3D accelerators.

nVidia's quincunx does the filtering at the stage where the buffer is rasterized to the screen. The 3D-scene is rendered normally, but the Pixel Shader is storing each pixel twice, Figure-2, in two different locations of the frame buffer. This does not cost more rendering power than the rendering without AA, but requires twice the memory bandwidth of the pixel write operation at the end of the pixel rendering process.

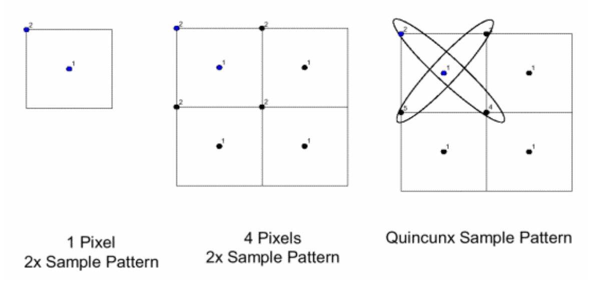


Figure 1: Quincunx Anti-aliasing

By the time the last pixel of the frame has been rendered, the HRAA-engine of GeForce3 virtually shifts the one sample buffer half a pixel in x and y direction (Figure 3).

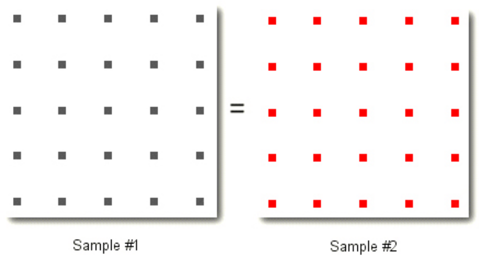


Figure 2: Storing pixels twice

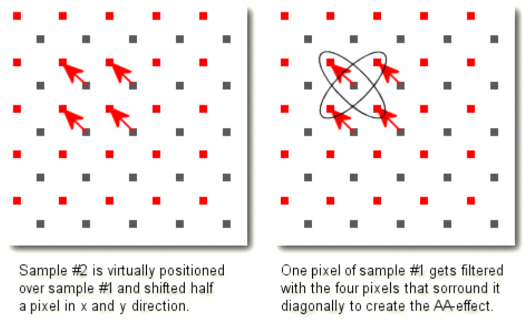

Sample #2 is virtually positioned over sample #1 and shifted half a pixel in x and y direction.
One pixel of sample #1 gets filtered with the four pixels that sorround it diagonally to create the AA-effect.

Figure 3

This has the effect that each pixel of the 'first' sample is surrounded by four pixels of the second sample that are 1/SQR(2) pixels away from it in diagonal direction. The HRAA-engine filters over those five pixels to create the anti-aliased pixel. The weights of the pixels are shown in Figure 4.

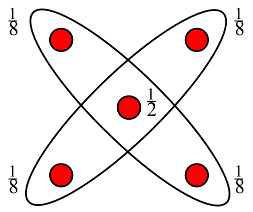


Figure 4: Weights of the quincunx pixels

Figure 5 is a comparison of quality between the anti-aliasing results. These images were captured by www.tomshardware.com for an article on the GeForce 3 video card. The image was taken from a frame in Quake III: Arena. (Nowadays, Quake III: Arena is used more often as a benchmarking tool than a game.)

It is quite clear that the quality of quincunx filtering is quite close to that of the 4x super-sampling anti-aliasing.

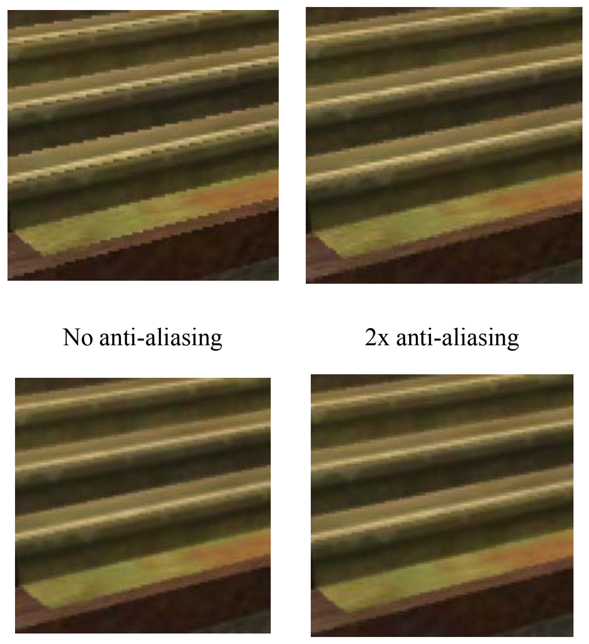


Figure 5: Comparison of anti-aliasing quality

Coming up with a similar technique with the available hardware is quite intuitional. By examining figure 2.3 and 2.4 a little bit closer, one can deduce that the quincunx sample can be reduced to the 2x2 sample in figure 6.

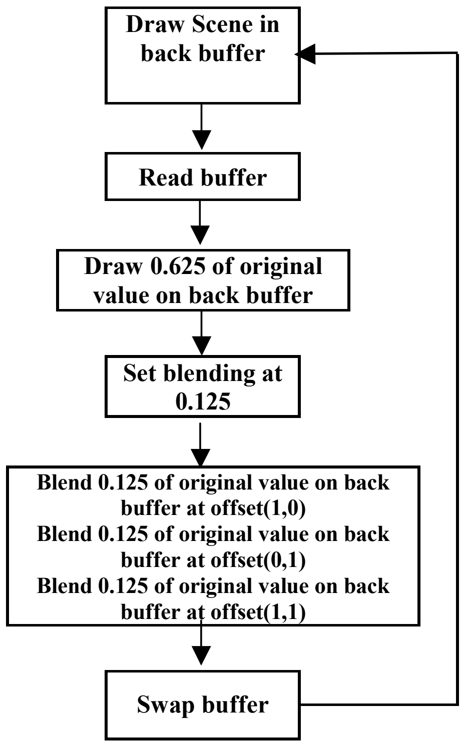
$$\begin{array}{ccc} \frac{1}{8} & & \frac{1}{8} \\ & \frac{1}{2} & \\ \frac{1}{8} & & \frac{1}{8} \end{array} \equiv \begin{array}{cc} \frac{5}{8} & \frac{1}{8} \\ & \\ \frac{1}{8} & \frac{1}{8} \end{array}$$

Figure 6

The top-left pixel and the centre pixel are virtually the same pixel in the case of a quincunx sample. The implemented algorithm simply captures the entire back buffer and draws it back to the same position {*offset(0,0)*} with 0.625 of the original value, blend with alpha value of 0.125 for three other images at *offset(0,1), offset(1,0)* and *offset(1,1)*. This would be equivalent of averaging the 2x2 sample with the new weights in Figure 6:

R= 0.625*offsetR(0,0) + 0.125*[offsetR(1,0) + offsetR(0,1) + offsetR(1,1)]
G= 0.625*offsetG(0,0) + 0.125*[offsetG(1,0) + offsetG(0,1) + offsetG(1,1)]
B= 0.625*offsetB(0,0) + 0.125*[offsetB(1,0) + offsetB(0,1) + offsetB(1,1)]

The algorithm can be summed up in the following diagram:



Draw Scene in back buffer
Read buffer
Draw 0.625 of original value on back buffer
Set blending at 0.125
Blend 0.125 of original value on back buffer at offset(1,0)
Blend 0.125 of original value on back buffer at offset(0,1)
Blend 0.125 of original value on back buffer at offset(1,1)
Swap buffer

MIP MAPPING DEPTH OF FIELD

MIP mapping is a popular technique intended to reduce aliasing. The essence of the technique is to pre-compute the texture at different levels of detail, Figure 7, and to use smaller textures for polygons further away from the viewer. It aims to improve graphics performance by generating and storing multiple versions of the original texture image. The graphics processor chooses a different MIP map based on how large the object is on the screen, so that low-detail textures can be used on objects that contain only a few pixels and high-detail textures can be used on larger objects where the user will actually see the difference. This technique saves memory bandwidth and enhances performance. The acronym MIP stands for Multum In Parvo (Latin for 'much in small')

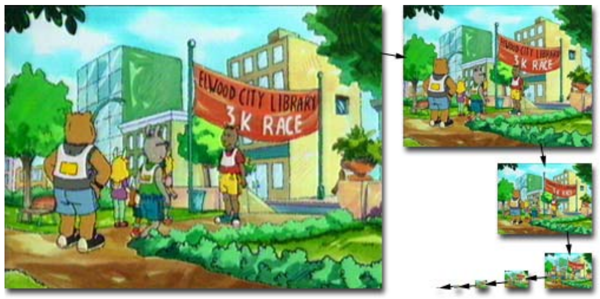


Figure 7: A 320x240 image reduced to 160x120, 80x60, 40x30 and so on…

The 3D hardware bilinear hardware kicks in when the smaller images are enlarged and as it attempts to fill in the missing pixels, an image that is more blurred will be generated (Figure 8). This algorithm will attempt to abuse this aspect of the MIP mapping hardware of a modern 3D accelerator to generate the blurred portions in an image with depth-of-field.

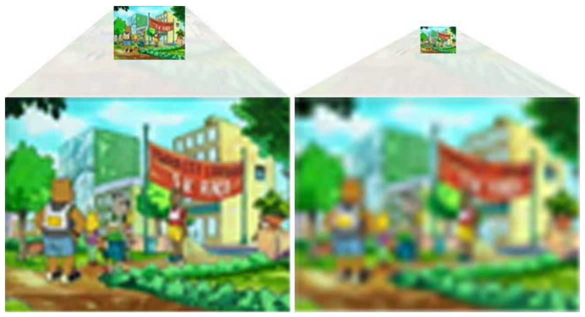


Figure 8: Enlarging a scaled down image

Since MIP mapping is such a standard technique, it would be safe to assume that all hardware accelerators, even the pioneering 3dfx Voodoo card, will be able to do it without much effort at all. The scene can either be rendered on the back buffer or the memory buffer before requesting the hardware to generate the MIP maps.

If the hardware does not natively allow rendering to a buffer other than the back buffer, rendering can be done on the back buffer first and the memory block will have to be copied by hand. And similarly, if MIP-map generation is not supported in hardware, it can also be read by hand and averaged with neighbouring pixels when scaling down.

"MIP-map Pixel Value" = [pixel(1,1) + pixel(1,2) + pixel(2,1) + pixel(1,2)] >>2

(where '>>' refers to shifting the bits to the right)

```
Render Scene onto
back buffer and Z-
buffer.
{
        :
}
```

```
Capture Scene from back buffer
{
    for (y=0;y<height;y++){
        for (x=0;x<width;x++){
            image(x,y)=backbuffer(x,y)
        }
    }
}
```

```
Generate MIPMAPs
{
    int level=0
    Repeat until image is 1x1
    {
        scale image to
        (
            $\frac{WIDTH}{2^{level}}$

            by

            $\frac{HEIGHT}{2^{level}}$
        )
        Use as LOD 'level' image
        Increment 'level' by 1
    }
}
```

```
Draw DOF image back onto back buffer
{
    for (y=0;y<height;y++)
    {
        for (x=0;x<width;x++)
        {
            backbuffer(x,y)=MIPMAP[ Z-value for
                pixel(x,y)](x,y)
            //MIPMAP[float i](int t,int s)
            //      where 'i' is a choice of which
            //      level of mipmap to use
        }
    }
}
```

MIP mapping DOF algorithm

Since the next level of detail is always a quarter of the current level, the averaging isn't as taxing as it sounds. Each destination MIP map pixel is just the average of the four corresponding source pixels, arranged in a 2x2 square.

MIP mapping and bilinear sampling can be merged to form tri-linear sampling where two neighbouring levels of detail from the texture are averaged to generate an in-between image. This will effectively allow the blurring of an image to be varied without the use of a filtering kernel and therefore speeding up the process of generating depth-of-field because multiple different degree of blurring will be needed for every single render.

It may be handy to be able to do a full screen blur with ease but this is an extreme effect and may not be much use when it comes to rendering a 3D scene other than doing transitions. It is, however, more desirable to be able to blur out some objects and leave the others sharp. Depth-of-field effects is can now be possible where only objects in the extreme distance or foreground are blurred, image focussing, in real-time. Depth-of-field would give a very impressive photo-realistic look to a rendered image (as shown with the cartoon image in Figure 9).

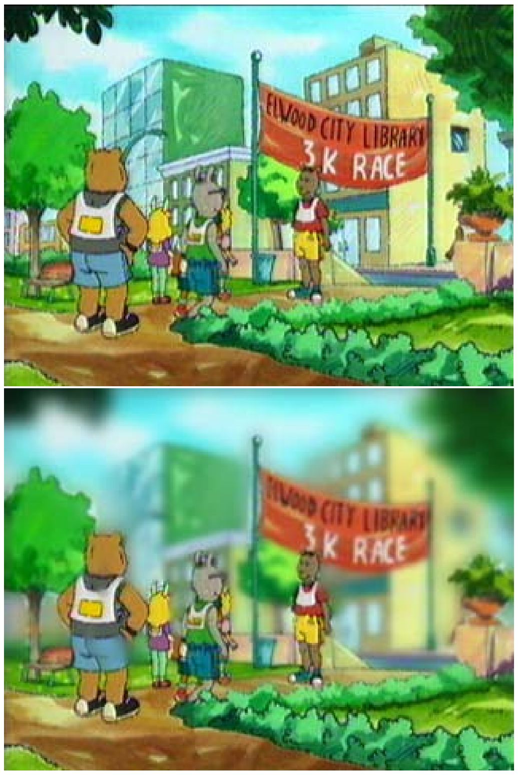


Figure 9: An image after MIP mapping depth of field

The next step in the algorithm is to decide which portion of the image would use what level of blurring. This will have to be done when the image is being rasterized onto the back buffer. The decision for each pixel will be made based on the Z-buffer value at the same position.

By definition depth of field is the total distance, on either side of the point of focus, which , when viewed from an appropriate distance, appears sharp in the final image. The interest of this algorithm is to move real-time graphics away from their usual artificial look which is caused by shaded triangles projected in a 2 dimensional space.

The cartoon image is intentionally chosen as an example because like all or most real-time 3D computer graphics, everything appears to be in the foreground and without the perspective drawn into the cartoon, the confusion may be even greater. Notice how the depth of field increase the realism as well as reducing the confusion between the background and foreground image which plagued the left image of Figure 9. This forces the viewer's eye to concentrate on the more important, non-blurred characters.

Since the backbone of the major part of the algorithm relies on the 3D hardware, the whole process could be automated into the hardware itself. This would provide easy access to the depth-of-field or blurring effect with just an API call.

As with all great things, there is a down side to this algorithm. One of the deficiencies of this technique is the effect which appears like an aura around a focused object. The appearance of this effect is not at all bad as it creates a "Vaseline lens" effect (Figure 10) on an image and brings it closer to photo-realism.

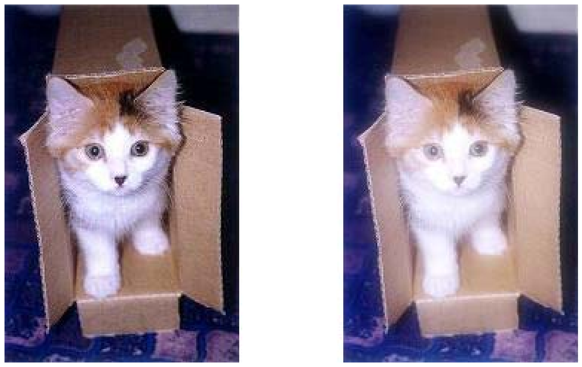


Figure 10: Photo of Xena, the cat, without and with the "Vaseline lens" effect

The other effect that is not pleasant appears when the focused object is in the background. Due to the cut-off of the Z-buffer when the foreground is blurred, instead of fuzzing out the foreground, a distinct line can be seen where the foreground edge meets the background.

So while the effect obtained when focusing the foreground and blurring the background adds realism to an image, the effect when focusing the background and blurring the foreground is less forgiving. This renders this technique only effective when doing depth of field of the former.

A possible solution to stop this effect is to blur out the Z buffer when doing this method of depth-of-field so that no sharp edges would be present, and hence will give a smoother transition from the foreground object to the background object.

Another possible but less appealing solution would be to render the foreground objects (objects before the focal point) onto individual memory buffers. These images will then be used as sprites, drawn on the position where the object will be if drawn in 3D. Do multi-pass blending jitters with

$$Offsets = (focal\_point - object\_distance\_from\_focal\_point) * constant$$

with the sprites to create the fuzzy edged objects that are at the foreground.

For the mentioned MIP mapping DOF implementation to work effectively, one would need to get to the metal of the hardware. For the purpose of this project, a simplified version of the algorithm, which could produce a similar result on existing 3D accelerators, was implemented.

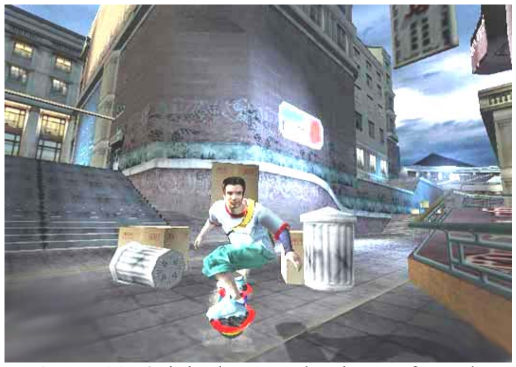


Figure 11: Original screenshot image from the game Airblade.

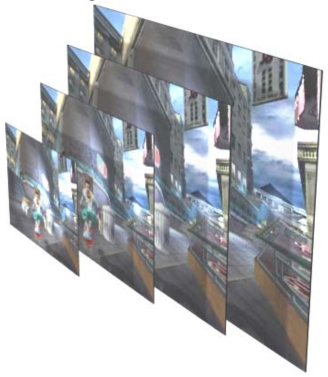


Figure 12: Images with different LOD layered and scaled to appear as the same size and position to the viewer

```
Render Scene onto back buffer and Z-buffer.
{
        .
        .
}
```

↓

```
Capture Scene from back buffer
{
    for (y=0;y<height;y++)
    {
        for (x=0;x<width;x++)
        {
            image(x,y)=backbuffer(x,y)
        }
    }
}
```

↓

```
Generate MIPMAPs
{
   int level=0
   Repeat until image is 1x1
      {
            scale image to
            (
               $\frac{WIDTH}{2^{level}}$
               by
               $\frac{HEIGHT}{2^{level}}$
            )
            Use as LOD ‘level’ image,
            Increment ‘level’ by 1
      }
}
```

↓

```
Draw DOF image back onto back buffer
{
        On Depth Testing but do not write changes to depth buffer
        Let Z be the depth where objects are to be sharpest
        Draw LOD[3] at Z-3*offset
        Draw LOD[2] at Z-2*offset
        Draw LOD[1] at Z-1*offset
        Draw LOD[0] at Z
        Draw LOD[1] at Z+1*offset
        Draw LOD[2] at Z+1*offset
        Draw LOD[3] at Z+1*offset
}
```

MIP mapping DOF algorithm (second implementation)

While the first implementation assumes an infinite amount of LODs in the image, this implementation will reduce that to a finite amount of LODs.

As before, a scene like the one in Figure 11 will be rendered, MIP maps generated. These maps will be layered and scaled in such a way that they will all appear the same size and position to the viewer (Figure 12). These images will be effectively be billboards cutting through the depth of the scene based on the existing Z-buffer values.

By positioning the desired LOD on appropriate position along the Z-axis, the Z-buffer mechanism will automatically cut out the unwanted portions of a layer that might block the layers at the back.

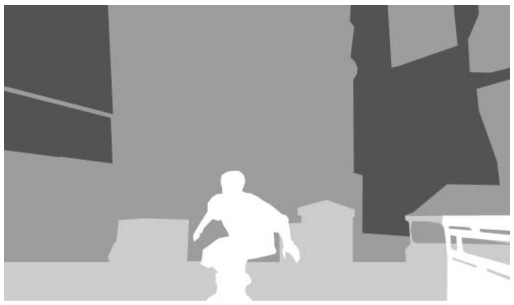


Figure 13: Simplified Z-buffer of the image.

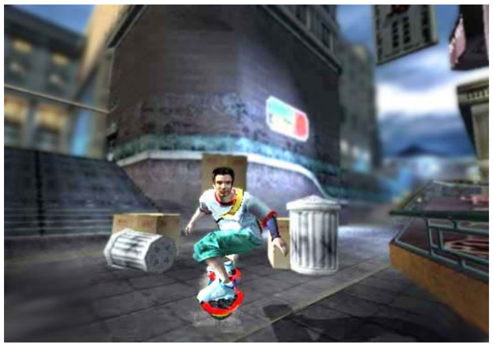


Figure 14: MIP mapping depth-of-field in effect.

In figure 13, the darker portions are objects that are further away and the brighter portions are objects nearer to the front. Figure 14 is drawn by drawing the LOD images in the way illustrated in figure 12 using the Z buffer values in figure 13.

As can be seen from Figure 14, although the levels of LODs have been reduced to a finite amount, the technique can still deliver a convincing effect of depth-of-field.

The effects that are found in the previous implementation will still exist in this implementation, however the 'foreground blurring edge' effect will not be visible in Figure 14 because the focused character is in the foreground.

CONCLUSIONS AND FUTURE WORK

Techniques were proposed and implemented to perform anti-aliasing and depth of field processing using features of OpenGL and current 3-D accelerators. Surprisingly impressive images have been obtained in spite of the fact that these facilities are not supported 'natively' by the equipment in use. Future work will include refinements of these algorithms, exploration of the possibilities for direct implementation in future hardware and investigation of other effects such as motion blur, sun glare and film grain.

REFERENCES

1. M. Potmesil and I Chakravarty, "A Lens and Aperture Camera Model for Synthetic Image Generation", in Proceedings of ACM-SIGGRAPH 81, Dallas, Texas, August 3-7 1981, pp297-305, Vol. 15, No.3.
2. Richard S. Wright, Jr and Micheal Sweet, " OpenGL Super Bible", 2nd Edition, Waite Group Press.
3. Cant, R.J. and P.E. Sherlock. 1987, "CIG System for Periscope Observer Training", in Proceedings of the 9th Inter-Service/Industry Training Systems Conference, 311-314.
4. J. Montrym, D Baum, D Dignam, and C Migdal, "InfinitReality: A Real-Time Graphics System", in Proceedings of ACM SIGGRAPH'97, pp 293-301, August 1997.
5. S Nishimura, and T Kunii, "VC-1: A Scalable Graphics Computer with Virtual Local Frame Buffers", in Proceedings of ACM SIGGRAPH'96, pp 365-372, August 1996.
6. Sergei Savchenko, "3D Graphics Programming", SAMS.
7. Rod Stephens, "Visual Basic Graphics Programming", Wiley.