Real-time antialiasing of Edges and Contours of Point Rendered Implicit Surfaces

Dirk J. Harbinson, Ron J. Balsys,
Faculty of Business and Informatics, Central Queensland University,
Rockhampton M.C., Qld. 4702, Australia.
Kevin G. Suffern
Faculty of Information Technology, University of Technology Sydney,
P.O. Box 123, Broadway NSW 2007, Australia.
d.harbinson@cqu.edu.au, balsys@cqu.edu.au, kevin@it.uts.edu.au

Abstract

We present various algorithms for antialiasing silhouette edges of manifold and non-manifold implicit surfaces. The algorithms are: object space, edge blur, super-sampling, adaptive pixel-tracing, and jitter-based antialiasing. We discuss the strengths and weaknesses of the approaches and compare the results. We also discuss the antialiasing of contours rendered on implicit surfaces.

The algorithms for antialiasing and rendering contours take place in the GPU and so are performed in real time. This allows us, for instance, to animate contours on a surface in real time by changing contour parameters. We give examples showing the results of our antialiasing techniques for various types of contours rendered on surfaces.

Keywords— **antialiasing, points, edges, contours, implicit surfaces, non-manifold surfaces, octree, interval**

1 Introduction and previous work

An implicit surface is a level (iso-valued) surface of a 3D scalar field $f(x, y, z)$, which we take to be the specific surface $f(x, y, z) = 0$. Implicit surfaces can be regular, singular, or non-manifold. Singular surfaces include those that self intersect, and non-manifold surfaces have regions where they are locally non Euclidean, for example, points of infinite curvature.

There are a number of techniques in common use for rendering surfaces, the main ones being polygonisation, scan line, ray tracing, and point based techniques. They all have their advantages, disadvantages, and limitations. Gross[1] recently discussed the growing use of points as a display primitive.

Balsys *et al.*[2] used point based algorithms for rendering the intersections of a surface with a series of level surfaces of other scalar fields $g_i(x, y, z), i = 1..n$. These are the surfaces $g_i(x, y, z) = c$ for different values of c. The intersections are rendered as contour lines on $f(x, y, z) = 0$. Alternatively, the scalar fields may be functions related to the implicit surface, which we wish to visualize on the surface.

The examples we consider here are the Gaussian and mean curvatures of $f(x, y, z) = 0$, which are intrinsic properties of the surface. Spivac[3] gives general formulae and examples for the Gaussian and mean curvature of implicit surfaces; see also Mitchell and Hanrahan[4] for an independent derivation of the formulae. Potzmann and Opitz[5] presented formula for curvature analysis of curvature functions. Guid, Oblosek and Zalig[6] asserted curvature is one of the most important tools for surface analysis. Balsys and Suffern[7] presented general implicit forms for the analysis of curvature.

Balsys *et al.*[8] rendered antialiased contours on ray-traced surfaces. In that work a *slab algorithm* was used to ensure that the contour *slabs* drawn on the surface have an apparent constant thickness. Formulae for *curved* and *non curved* slabs where given. These are used in this work to control the thickness of the contours rendered on the surfaces. In the same work reference is made to antialiasing of the surfaces. In ray tracing extra rays are shot into each pixel to allow for a calculation of the *average* colour of a pixel. The number and distribution of these rays determine the quality and nature of the resulting antialiasing, see for example Foley *et al.*[9] or Whitted[10]. It is not clear how anti-aliasing should proceed on point based surfaces and this is a major contribution of this work.

Stolte *et al.*[11], and Stolte[12] used interval arithmetic for voxelising implicit surfaces and he used interval methods with rectangular coordinates. The voxel space has a resolution of 512^3, and the painter's algorithm is used for rendering the voxels. Antialiasing is not discussed.

Bloomenthal[13] discusses antialiasing of z-buffer pro-

978-0-7695-3359-9/08 $25.00 © 2008 Crown Copyright
DOI 10.1109/CGIV.2008.12

grams. This work developed a simple algorithm that can be used to antialiase the silhouette edges of complex connected regions. This work is an important original contribution to antialiasing what are essentially bitmaps.

Balsys *et al.*[14] developed a sampling technique that uses octree space subdivision with a natural interval exclusion test to minimise the octree subdivision process. The intention was to produce high quality images where there is no possibility that portions of the surface are either rendered incorrectly or are missed. Interval techniques are ideal for this purpose. Criteria for the complete pixel coverage of implicit surfaces was given. This algorithm was a major contribution of the paper. A disadvantage of the work was that for certain surfaces, rendering artifacts occurred due to the use of the interval subdivision method, for example singular lines (rays) where rendered as tubes. Also aliasing was evident on silhouette edges, along surface contours, and along the singular lines.

Harbinson *et al.*[15] reduced the rendering artifacts by using gradient and point sampling information in the plotting node to trim the interval voxels found at the plotting depth. This results in fewer points being rendered around non-manifold features. The new algorithm also resulted in the rendering of singular lines as lines of constant finite thickness. The resultant surfaces are of much improved visual quality and accuracy. However a disadvantage of the work was that the images still suffered aliasing issues along silhouette edges, around surface contours, and along the singular lines.

From previous work it is apparent that it is not clear how antialiasing should proceed on point based surfaces. We report here on a number of approaches to antialiasing along silhouette edges and give recommendations as to the best antialiasing approach. We also consider methods to remove aliasing in contours rendered on the surface and in doing so, show how contours can be rendered in real time. Finally and we show how antialiasing can be done along singular lines. Showing how antialiasing can proceed on point based implicit surfaces is the major contribution of this work.

2 Point based antialiasing methods

We are concerned with the quality of the images produced by our point based method and the speed of the antialiasing technique, as we wish to apply it to rendering manifold and non-manifold implicit surfaces.

Aliasing occurs on the silhouette edges of surfaces, where the surface meets the background or self occludes. Aliasing also occurs along contour edges when contours are rendered on the surfaces. Aliasing along the silhouette edge can be very prominent, as can be seen in Figure 6(a). Aliasing along contour edges can be just as prominent.

Zwicker *et al.*[16],[17] used the elliptical weighted average filter, EWA, of Heckbert[18] to avoid aliasing problems with textures when using splatting to render point clouds. Antialiasing for polygonal surfaces can be performed using the built-in features of modern graphics hardware. Multi-sampling (or super-sampling, see Foley & Van Dam[9]) of scenes composed of polygons is very effective. When antialiasing polygon scenes the geometry's complexity remains constant, regardless of the antialiasing method being applied. The performance is satisfactory as the bottleneck lies in the graphic cards memory bandwidth.

The point based methods used in previous work [8], [2], [14], and [15], where not antialiased, as it was not clear how to proceed with antialiasing point rendered scenes. We can't use n rooks or multi-jittered approaches, Chui *et al.*[19], as these require the re-rendering of the entire scene with slightly different projection matrices. We can't store the sampled points in memory when rendering complex surfaces because point counts are too high. We would need to re-sample points on the surface when re-rendering the scene. As the sampling of the points is the major bottleneck, these methods are not efficient. As we are not using splatting, the antialiasing scheme of Zwicker *et al.*[16] and Heckbert[18] is not applicable and in neither of these is aliasing on the silhouette edge, or contour edges, fixed.

2.1 Object space antialiasing

Aliasing of an implicit function occurs as we are point sampling the function. Take, for example, the curved $2D$ function in Figure 6(a) where each pixel is fully rendered when the surface covers at least 50% of the pixel boundaries and otherwise is not rendered at all. This creates jaggy edges, causing the well known staircase effect which is visually unpleasing. A better approach would be to texture the surface based on the percentage coverage of the pixel by the surface. This is illustrated in Figure 6(b). The normal way of achieving this is to blend the background and surface texture based on the surface coverage. Effectively this can be achieved by blending the surface and background colours of the affected pixel.

In our work we consider the surface colour to have a certain opacity value, ω. The edge pixels' colour is combined with the background pixel colour to determine the boundary pixel colour. For each pixel we need to calculate the opacity level based on surface coverage in the pixel. In our system each pixel is a projection of an octree node onto a regular grid of pixels, so this can be accomplished by analysing each octree node at the final rendering depth. Each octree node has 8 vertices which may or may not be inside the surface. We can divide the number of vertices which are within the surface by 8 to obtain a fraction between 0 and 1 which will be used to represent the opacity

of the surface pixel.

A problem arises in that only octree nodes along the silhouette edge of the surface (or a contour) need to be antialiased and these need to be found. This is done by first creating an image where the value of the pixel is 0 or 1 depending on whether the pixel is along a silhouette or contour edge. In the rendering path, for pixels on the silhouette edge we blend the ω times surface colour with the background colour based on this value. The final result is shown in Figure 1.

Figure 1: Object space anti-aliasing applied to curved surface.

This result was not considered effective, as the resultant anti-aliasing edges where faint and inconsistent.

2.2 Edge blur method

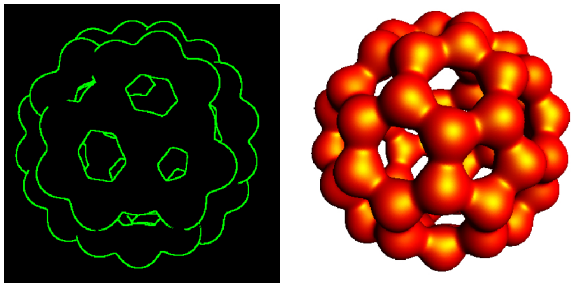
Edge blurring is a post-process technique that reduces aliasing along the edge geometry. The method is applied after the all the points in the complete scene has been found. It is implemented as a filter in image space. This is common in deferred shading applications that, under Direct X 9.0, cannot perform multisampling with multiple render target (MRT) hardware.

Typically a renderer that uses MRT defers the lighting stage to the end of the rendering chain. This significantly reduces lighting computations to $O(1)$ time from $O(n)$. In our work lighting calculations are not the bottleneck, so we do not defer the lighting stage to the end of the rendering pipeline. We do however use a second render target to store the point type (surface or contour) and depth information.

We begin by looking at a figure of the Buckyball surface (C^{60}) which is the iso-potential surface defined by 60 point charges in 3D given by

$$f(x,y,z) = \sum_{j=1}^{60} \frac{q_j}{(x-x_j)^2+(y-y_j)^2+(z-z_j)^2} - c = 0. \quad (1)$$

Here, (x_j, y_j, z_j) is the location of the charge, with charge value q_j, and c is the value of the potential surface. We produce an image showing where the edges are located (see Figure 2(a)). This is done by comparing the depth value between neighbouring pixels in horizontal, vertical and diagonal directions. If the depth offset between these pixels is outside a tolerance value, α, the pixel is an edge pixel (rendered green in Figure 2(a)).



(a) (b)

Figure 2: Edge detection of bucky ball surface (1): (a) silhouette edges in green, (b) aliased surface image.

Pixels that are part of the contour are detected as given in Balsys *et al.*[2]. Pixels that are detected as being part of a contour have a depth offset added to them so that they can be detected in the final edge detection pass and thus be antialiased.

In the next stage we take the image (Figure 2(b)) and overlay it with the edge map (Figure 2(a)). For each pixel not on an edge, the output colour is the same as the source pixel colour. If a pixel lies on an edge then a 3×3 kernel, as shown in Figure 3, is used to average the colour samples.

$\frac{2}{25}$	$\frac{3}{25}$	$\frac{2}{25}$
$\frac{3}{25}$	$\frac{5}{25}$	$\frac{3}{25}$
$\frac{2}{25}$	$\frac{3}{25}$	$\frac{2}{25}$

Figure 3: The 3×3 blur kernal used in weighting pixel colour contributions from adjacent pixels.

The colour contribution of the neighbouring pixels is calculated using the pre-computed weighting values shown in Figure 3. We have found this method to be successful in removing sharp aliasing features in an image. However the antialiasing quality appears to be dependent upon the angle of the edge; it works best on irregular pixel lines. Steep edges produce a very harsh staircase effect, and in our tests the blur method does not handle this well, and only softens / accentuates the problem, as shown in Figure 4.

Figure 4: Edge blur is OK for moderate slopes in either direction but not for steep or shallow edges.

As a result we don't consider the visual quality of this method to be satisfactory compared to other antialiasing methods we explored. The positive aspect of this method is that it is compatible with both standard and deferred rendering systems, and can be applied with moderate efficiency in real-time.

2.3 Super sample antialiasing (SSAA).

The classic brute force method of anti aliasing is known as super sampling and is discussed in Foley *et al.*[9]. Super sampling involves rendering the scene to a much higher resolution than required, and then down sampling this image to a smaller size. Each pixel is a result of multiple samples averaged together from the over-sampled image using some form of weighted average.

Super sampling is very slow, as it significantly increases memory bandwidth. In respect to our point based method, rendering a larger image requires sampling many more points to ensure there are no holes in the final image. To achieve this we need to increase the octree depth, with an exponential increase in running time. For example a surface that takes 5 minutes to render at 512×512 pixels could take over 30 minutes to render at 1024×1024 pixels with 2×2 super-sampling applied. While the result will be of high quality, this is impractical for complex surfaces.

2.4 Adaptive pixel-tracing method

Our next method begins by creating an edge map, similar to that for the edge blurring method with but with an alteration. When a surface self occludes we want to blend the edge of the front surface with the back surface colour. This enables us to fade out the edge of the surface in front, into the colour of the surface in the back. Edge pixels are blended with the background colour.

The edge blur method's limiting factor is that it applies a constant level of colour averaging on detected edges through a 3×3 kernel. In contrast the pixel-trace method is an adaptive technique that tries to locate jagged edges and then soften them, much as in traditional antialiased systems. The pixel-trace method produces sharper and smoother images than the edge blur method. This method is also compatible with both standard and deferred shading systems, although real-time performance has not been a goal, as we focus on image quality.

In this approach a shader algorithm is used to antialias pixels so that they are shaded in relation to surrounding pixels. The shader algorithm determines what surface edge the pixel is located on and how many other pixels share that edge in a particular relationship with the pixel to be antialiased. The shader uses this information to determine the opacity factor applied to the pixel based on its position and style. Position and style is based on the run length of the row of pixels in the same horizontal / vertical line and how the run is terminated.

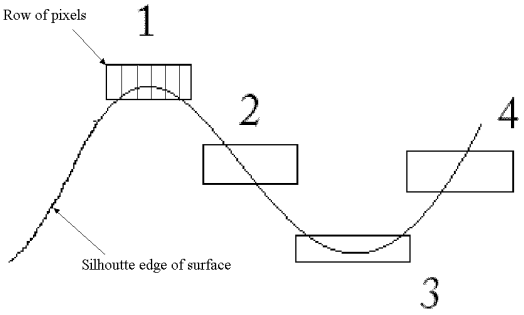


Figure 5: The four position and style types for pixel antialiasing cases in the adaptive pixel-tracing method, labeled as Case 1, 2, 3 and 4.

The different styles of edges that are handled by the shader are listed in Figure 5. The technique begins by detecting monotonically increasing or decreasing edges (Case 2 & 4) as illustrated in Figure 6(a).

Figure 6: (a) Common Aliased staircase effect and (b) the result of our pixel-traced antialiasing pass.

For each pixel on a front surface edge (sampled from the previously described edge map texture), we trace a horizontal row of pixels, left and right, to find the start and end of the horizontal row of pixels along the monotonically increasing or decreasing surface edge. If the width of the row is 1 then we alternately check if the pixel is the start of a vertical column of pixels instead.

If we have detected a horizontal row it needs to be smoothed out at its ends. The start of the edge will have the lowest opacity level and the end of the edge will have the highest opacity level (the $opacity$ is $1 - alpha\ value$

of the pixel).

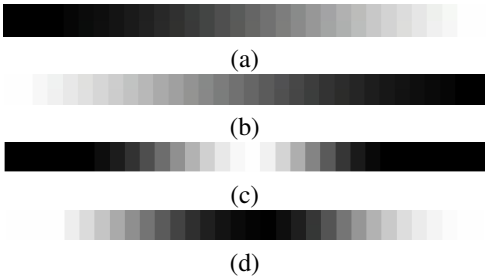


Figure 7: Edge smoothing groups (a) opacity highest at left (bottom) of row (column), (b) opacity highest at right (bottom) of row (column), (c) opacity highest at ends, and (d) opacity highest at centre.

There are four possible positions of horizontal / vertical runs of pixels, see Figure 7. How we calculate the *opacity* depends on the position, which we label left (case 2), right (case 4), top (case 1) and bottom (case 3) given in Figure 5. Analogous cases are used for column runs of pixels.

For a pixel in the column of Figure 7(a) each pixels opacity value, ω, is found by dividing the number, x, of the run of pixels up to the pixel whose opacity is being calculated by the total number of pixels, L, in the row of pixels (see Figure 8).

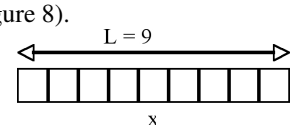


Figure 8: Row of pixels of length L.

$$\omega = x/L \tag{2}$$

Once the opacity value, ω, has been found we use it to calculate the final pixel colour using;

$$\begin{aligned} Color.rgb &= (sourceColor.rgb * \omega) \\ &+ (edgeColor.rgb * (1 - \omega) \end{aligned} \tag{3}$$

If the opacity change is reversed, as in Figure 7(b), we simply invert the opacity value, ω, and use it in Equation 3.

We now consider cases (c) and (d) when an edge is either on the outside of a surface or "inside the surface. These cases must be specially coded as the aliasing needs to fade in and then out again, or vice versa.

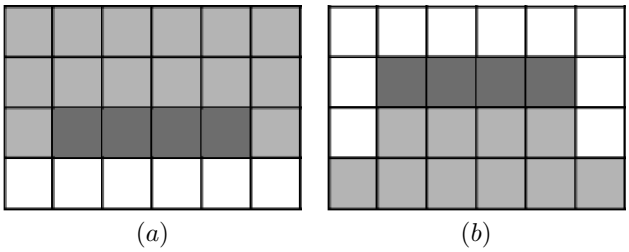


Figure 9: (a) Inside edge shown in dark grey. (b) Outside edge shown in dark grey.

Figure 9(a) illustrates an inside edge occurrence (case 3 in Figure 5) in dark grey, the light grey areas show the surface pixels. In this case the edge does not simply fade out into nothing. It must fade / blend between both sides of the surface to ensure a smooth appearance, as shown in Figure 7(c). Figure 9 (b) illustrates an outside edge occurrence (case 1 in Figure 5). The outside edge in dark grey needs to be faded as shown in Figure 7(d) for a resulting smooth appearance.

For case 3 we modify the existing opacity value, ω, found in Equation (1) and recalculate ω's value as

$$\omega = abs((\omega - 0.5) * 2) \tag{4}$$

For case 1 we invert the opacity value calculated in Equation (3) and use it in equation (4).

If a pixel has been found to lie on a column edge rather than a row edge as described previously, we perform the same procedure but scan up and down the column rather than left and right as in the row. The process is then exactly the same as just described for the rows.

In antialiasing care must be taken when a surface self occludes to avoid problems. Our edge map only includes pixels along the surface edge and has no contribution from background pixels. This solves potential problems and ensures we only smooth the edge.

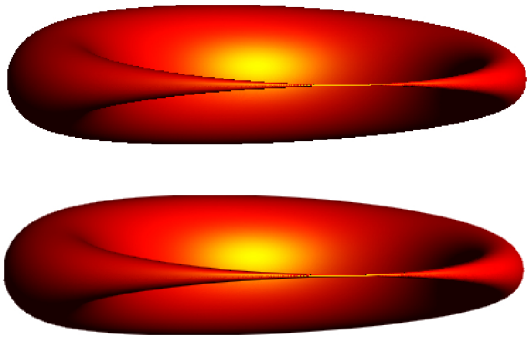


Figure 10: Demonstrating antialiasing on the inside and outside surface edges. (Top) aliased image and (Bottom)

antialiased image of the Cyclide (2) surface.

We conclude by showing an aliased (Figure 10 (Top)) and antialiased version of the Cyclide surface (Figure 10 (Bottom)) given by

$$\begin{aligned} f(x,y,z) &= (x^2+y^2+z^2)^2 \\ &- 2(x^2+r^2)(f^2+a^2) - 2(y^2-z^2)(a^2-f^2) \\ &+ 8afrx + (a^2-f^2)^2 \end{aligned} \quad (5)$$

with a = 10, f = 2 and r = 2.

2.5 Jitter based antialiasing

In OpenGL the frame buffer object allows you to attach a separate z-buffer to each vertex buffer object (VBO). We use this to our advantage, as follows. Typically we create sixteen frame buffer objects each with its own z-buffer and colour buffer. Each frame buffer object has its camera origin jittered with respect to the pixel grid. The jittering values should make an irregular pattern to make them "noisy" to avoid artifacts. The process, and OpenGL code for jittering using an accumulation buffer is given at the OpenGL website[21], as is a table for sample jittering values. The 16 jitter values we used come from this table.

We cannot use an accumulation buffer as discussed on this website as the accumulation buffer requires the z-buffer be cleared before each pass. The use of a z-buffer attached to each vertex buffer object (VBO) allows us to get around this problem.

The image is rendered into each of the 16 z-buffer and colour buffers. The colour values in the separate VBO's are averaged to give the final colour to be assigned to the pixel. This results in antialiasing of the pixels colour. It works equally well for antialiasing silhouette edges and contour edges.

Essentially this is a super-sampling approach, the viewpoint is rendered multiple times, each time the viewpoint is offset by amounts smaller than a pixel, and the colour values are then averaged to give the pixel colour. The method requires the entire scene geometry to be determined beforehand, so each render pass has exactly the same geometry. This can be performed while the samples are being generated, thereby making the sampling process interactive and able to be run in real time.

3 Contour antialiasing

Rendering contours onto surfaces can present problematic aliasing artefacts. Originally we tried to offset the depth of a pixel on a contour away from the object's surface. The contour edge could then be identified in the edge antialiasing pass. However, the erratic nature of runs of pixels sampled along contour edges resulted in severe pixellation artifacts. To solve this problem we moved the contour generation phase from object space into image space.

Previously we tested if a sampled point lay on a contour strip, and changed that pixel colour to the contour colour. This has the limitation of having to test every point in the scene, most of which are not directly visible from the viewpoint. By moving this test into image space, we only test each visible pixel on the screen, eliminating the bottleneck. A simple surface such as a sphere takes the same amount of processing time to contour as a complex surface such as the bucky-ball surface. This opens up the possibility of rendering contours in real-time on a scene, no matter how complex the scene.

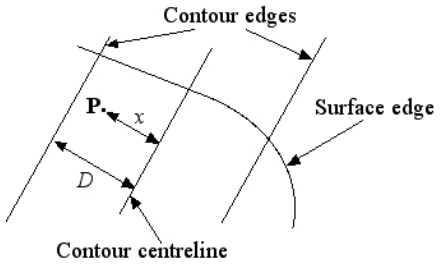


Figure 11: Illustration of the distance, x, of the surface point P from the centre of the contour slab.

To smooth (antialias) the contours appearance we interpolate the distance, x, between each point in the plotting node and the center origin of the contour slab, with D the distance between the centre of the slab and the outside of the slab. This is illustrated in Fugure 11. The pixel colour is the average of $\frac{x}{x+D}$ times the background colour plus $\frac{x}{x+D}$ times the surface colour. The distance value x determines the pixel colour based on a linear average of the surface and contour colours.

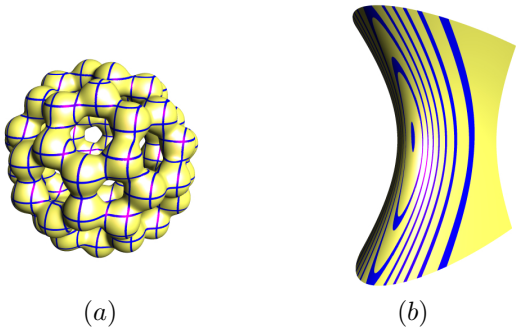
This means that:

- as explained above we dont test every sampled point in the octree, only the points visible from the camera. This results in large speedups in contour rendering times.
- contour calculation is just as fast on a simple surface as it is on a very complex surface.
- deferred shading lets us smooth out the contour values based on all the visible points, and so the smoothing can vary depending on the plot depth in the octree.
- using a deferred renderer means we can render a complex surface from a specific viewpoint and then use the points seen from this viewpoint to perform

real-time visualisation of contours. We can also rotate and modify lights in real-time. In particular we can animate moving contours across a surface at real-time speeds (regardless of how long the surface originally took to sample).

3.1 Examples from real-time visualisation of contours

First, in Figure 12(a) we show anti-aliasing of planar contours such as planes orthogonal to the y and z axes, and in Figure 12(b), we show lines of Gaussian curvature rendered on an ellipse. To smooth (antialias) the contours appearance, we test the distance between each pixel and the center of the contour slab. This distance value determines the strength of the contour colour opacity, ω, overlaying the surface. The width of the contour slab is set using the formulae for curved (Gaussian curvature) and non-curved (planar contours) slabs given in Balsys and Suffern[8].



(a) (b)

Figure 12: (a) Planes parallel to the y and z coordinate axes rendered on a buckyball surface (1) and (b) lines of constant Gaussian curvature rendered on an ellipse.

As the renderer defers the lighting calculations to the contouring pass, surface lighting and changing contours can be rendered in real time using this approach.

4 Filling missing pixels

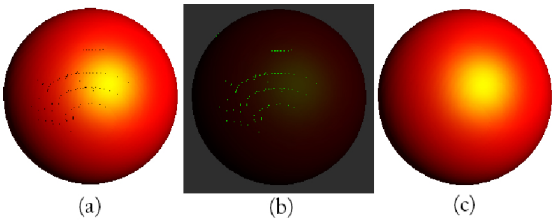


Figure 13: Demonstrating missing pixel filling: (a) rendered image with missing pixels, (b) missing pixels detected in bright green, (c) final surface with missing pixels blended out.

Our renderer represents each surface sample in the plotting node by a single pixel on the image plane. It is well known that point rendering can result in missing pixels on the surface image, indicating we needed to render the surface at a higher plot depth (see figure 13(a)). We can fill in missing pixels without significantly increasing our rendering times in the antialiasing pass as follows.

After the lighting stage in the viewing pipeline we loop through each pixel and compared neighboring pixels for depth changes. If we find a pixel is surrounded by 5 or more points closer to the camera (within a tolerance α) than itself the pixel is considered a missing pixel (hole) in the surface. The colour of the pixel is calculated as a weighted blend of the neighboring pixels colours. Figure 13(b) shows missing pixels in green for a sample sphere. Figure 13(c) shows the final sphere rendering with the missing pixels colour blended out as specified.

5 Summary and Discussion

In this work we have explored a number of methods for antialiasing edges and contours of implicitly defined surfaces. We did not find that an object space method based on counting the number of nodes that project onto a single pixels produced satisfactory results. We then turned our attention to image space methods.

Originally we developed an edge blur method that antialiased pixels around silhouette edges based on a 3×3 weighting kernel. Whilst this produced better results than our object based approach is still suffered when the slope of the silhouette edge was either steep of shallow. Next we looked at super-sample antialiasing, which does provide good results but suffers in that it takes up to four times longer to produce the final result.

We then considered a more in depth analysis of pixel runs in the image in an attempt to improve antialiasing of the point based images. The antialiasing resulting from the adaptive pixel-tracing approach was superior to the edge blur method, in that it did not suffer from systematic problems related to the slope of the surface edge.

Finally we developed an approach based on the functionality of current GPU's. Our jitter based method's antialiasing performance was as good as the previous pixel-tracing method but was simpler to implement and faster, as it runs completely on the GPU.

We then discussed contour antialiasing. We compared two methods for contour antialiasing. The first method used a weighted average of the distance from the point, P, over the slab half-width to weight pixels colour. This worked well for the non-curved contours we used. However adapting this method for use with curved contours, such as occur when rendering lines of constant Gaussian curvature, was non-trivial and was not implemented. For these surfaces we used the jitter based antialiasing meth-

ods described previously for silhouette edges, to antialiase the surface, as this method is simple and provides good results.

Finally, in Figure 14, we present a number of antialiased images on non-manifold implicit surfaces to demonstrate the improvements we have achieved in rendering these surfaces. The interested reader should compare this figure to the results shown in Figure 12 of the work by Singh and Narayanan[20] on real-time ray-tracing of Implicit surfaces on the GPU. We have significantly improved the rendering of many of these images compared to this work. Our point based method produces images comparable in quality to ray-traced algorithms.

In future we wish to improve on the accuracy with which we render implicit surfaces, particularly non-manifold surfaces. In current work their are still a number of issues in rendering surfaces such as Steiner's Roman surface.

6 Acknowledgment

The authors wish to acknowledge the support given for this work by their respective universities.

References

[1] Gross, M., Getting to the Point...?, *IEEE Computer Graphics and Applications*, **26**:5, pp. 96–99. 2006.

[2] Balsys, R.J., and Suffern, K.G. Point Based Rendering of Non-Manifold Surfaces With Contours. *ACM/GRAPHITE2004*, 15-19 June 2004, Singapore. 7-14, 2004. pp. 1–8.

[3] M. Spivac. *A Comprehensive Introduction to Differential Geometry*. Publish or Perish Inc., Berkeley. Volume III, Second Edition, Chapter 3, 1979.

[4] D. Mitchell and P. Hanrahan. Illumination from Curved Reflectors, *Computer Graphics*, **26**(4): 283–291, 1992.

[5] H. Pottmann and K. Opitz. Curvature Analysis and visualisation for functions defined on Euclidean spaces or surfaces. *Comp. Aided Geom. Design*, (11): 655–674, 1994.

[6] N. Guid, C. Oblonsek, B. Zalik. Surface Interrogation Methods. *Comput. & Graphics*, **19**(4):557–574, 1995.

[7] Balsys, R.J., and Suffern, K.G. Visualisation of implicit surfaces, *Computers and Graphics*, **25**:89–107. 2001.

[8] Balsys, R.J., Suffern, K.G. Ray Tracing Surfaces With Contours, *Computer Graphics Forum*, **24**:4, pp. 1–10. 2003.

[9] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics Principles and Practice, Second Edition*, Addison-Wesley, New York, pp. 714–715, 1990.

[10] T., Whitted. An Improved Illumination Model for Shaded Display, *Computer Graphics (SIGGRAPH 83 Conference Proceedings)*, pp. 151–156, 1983.

[11] Stolte, N., Kaufman, A. Parallel Spatial Enumeration of Implicit Surfaces using Interval Arithmetic for Octree Generation and its direct Visualization, *In Implicit Surfaces'98*, 81-87, Seattle, 1998.

[12] Stolte, N. Graphics using Implicit Surfaces with Interval Arithmetic based Recursive Voxelization, *in Computer Graphics and Imaging CGIM 2003*, 398-024, Honolulu, 2003.

[13] Bloomenthal, J. Edge Interference with Applications in Antialiasing, *Computer Graphics*, **17**:3, 157-162, 1983.

[14] Balsys, R.J., Suffern, K.G., Jones, H. Point Based Rendering of Non-Manifold Surfaces, *Computer Graphics Forum*, **27**:1, pp. 63-72. 2008.

[15] Harbinson, D., Balsys, R., Suffern, K.G Rendering Surface Features Using Point Based Methods, *ACM/GRAHITE2007*, 1-4th Dec 2007, Perth, Western Australia. pp. 47–53. 2007.

[16] Zwicker, M., Pfister, H., Van Baar, J., Gross, M. Surface splatting, *In SIGGRAPH 2001*, 371–378. 2001.

[17] Zwicker, M., Rasanen, J., Botsch, M., Dachsbacher, C., Pauly, M. Perspective Accurate Splatting, *Eurographics Symposium on Point-Based Graphics*, 247–254. 2004.

[18] Heckbert, P. Fundamentals of texture mapping and image warping. *Masters thesis, University of California at Berkeley, Dept. of Elec. Eng. and Computer Science*, 1989.

[19] Chiu K., Wang C., and Shirley, P. Multi-Jittered Sampling, *In Heckbert P. S. Ed., Graphics Gems IV*, AP Professional, Boston, MA, 370-374. 1994.

[20] Singh, J.M., and Narayanan, P.J. Real-Time Ray-Tracing of Implicit Surfaces on the GPU, *Report no: IIIT/TR/2007/72*, Report Date: 31-07-2007.Available at http://www.iiit.net/techreports/reports.html

[21] Available at opengl.org: *http://www.opengl.org/resources/code/samples/advanced/advanced97/notes/node63.html*, visited March 2008.

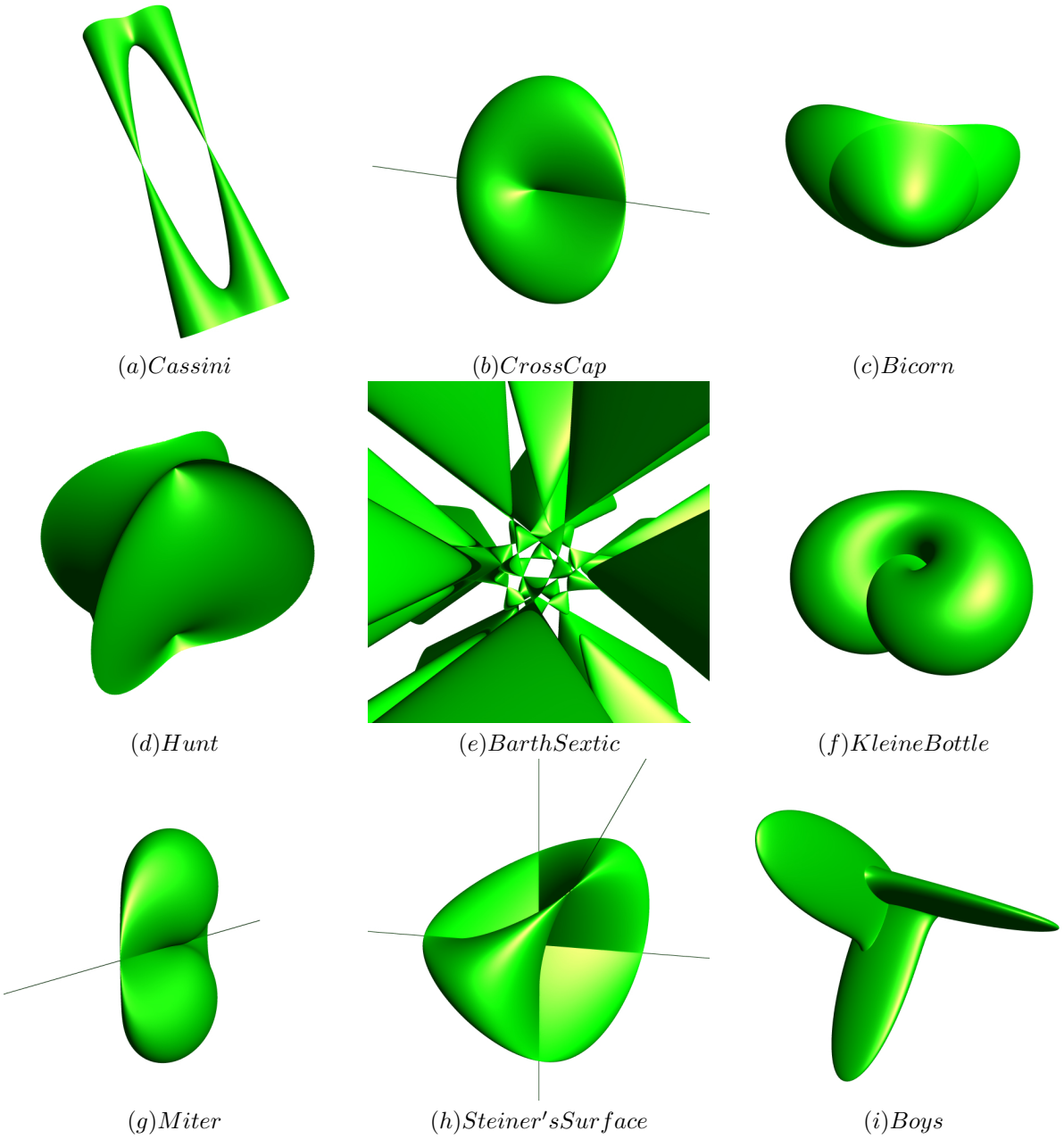


Figure 14: 600x600 images of non-manifold surfaces: (a) Cassini, $((x+a)^2+y^2)((x-a)^2+y^2) = z^2$, (b) Cross cap, $4x^2(x^2+y^2+z^2+z)+y^2(y^2+z^2-1) = 0$, (c) Bicorn, $y^2(a^2-(x^2+z^2))-(x^2+z^2+2ay-a^2)^2 = 0$, (d) Hunt, $4(x^2+y^2+z^2-13)^3+27(3x^2+y^2-4z^2-12)^2 = 0$, (e) Barth Sextic, $4(\phi^2x^2-y^2)(\phi^2y^2-z^2)(\phi^2z^2-x^2)-(1+2\phi)(x^2+y^2+z^2-1)^2 = 0$ where $\phi = (1+\sqrt{5})/2$, (f) Kleine bottle $(x^2+y^2+z^2+2y-1)((x^2+y^2+z^2-2y-1)^2-8z^2)+16xz((x^2+y^2+z^2-2y-1) = 0$, (g) Miter, $4x^2(x^2+y^2+z^2)-y^2(1-y^2-z^2) = 0$, (h) Steiners surface, $x^2y^2+x^2z^2+y^2z^2-2xyz = 0$ and (i) Boys surface, $64(1-z)^3z^3-48(1-z)^2z^2(3x^2+3y^2+2z^2)+12(1-z)z(27(x^2+y^2)^2-24z^2(x^2+y^2)+36\sqrt{2}yz(y^2-3x^2)+4z^4)+(9x^2+9y^2-2z^2(-81(x^2+y^2)^2-72z^2(x^2+y^2)+108\sqrt{2}xz(x^2-3y^2)+4z^4) = 0$.