Visualisation of Non-Manifold Implicit Surfaces

by

Dirk Jacob Harbinson

A thesis submitted for the degree of Doctor of Philosophy

CQUniversity

School of Computing Sciences

Faculty of Arts, Business, Informatics and Education

7th March, 2011

Abstract

This thesis addresses several visualisation problems for non-manifold and singular implicit algebraic surfaces and proposes algorithms for fast and correct rendering of many such surfaces. It is well known that non-manifold surfaces are particularly difficult to render and hence these algorithms provide interesting perspectives in rendering these surfaces.

A main contribution is a GPU algorithm for point-based rendering of implicit surfaces. This algorithm is based on a hierarchical decomposition of the bounding volume using an octree spatial data structure and testing the occupied cells of the octree through interval arithmetic. Previous work had difficulty in rendering the non-manifold and singular features. The work also presents point-based anti-aliasing of silhouette and contour edges. The second contribution is a GPU hybrid polygon and point-based rendering algorithm. Previous work using polygons alone could not represent the non-manifold and singular features. The third contribution is an algorithm for visualisation of curvature surfaces of implicit surfaces. A robust visualisation of curvature surfaces is presented here for the first time. A DVD is enclosed showing animations created using the discussed methods.

Even though Computer Aided geometric Design (CAD) uses predominately parametric surfaces, implicit surfaces are also used in this area as blending and offset surfaces in CAD are often implicit surfaces. Curvature analysis is also used extensively in the aeronautical and automotive manufacturing industries.

List of Publications

Publication 1 D.J. Harbinson, R.J. Balsys, and K.G. Suffern. 2007. Rendering Surface Features Using Point Based Methods, *ACM/GRAHITE 2007*, 1-4th Dec 2007, Perth, Western Australia. pp. 47-53.

This work [1] presents a point-based technique that improves the rendering of non-manifold implicit surfaces compared to previous point-based methods, using point and gradient information, to prune plotting nodes arising in octree spatial division, with a subdivision criteria based on the natural interval extension of the surface's function. We successfully rendered non-manifold features of surfaces such as rays and thin sections.

Publication 2 D.J. Harbinson, R.J. Balsys, and K.G. Suffern. 2008. Real-time Antialasing of Edges and Contours of Point Rendered Implicit Surfaces, *International Conference on Computer Graphics, Imaging and Vision cgiv08*, 26-28th August, 2008, USM, Penang, Malaysia. pp. 38-46.

This work [2] deals with solving aliasing issues when rendering non-manifold surfaces with points.

Publication 3 D.J. Harbinson, R.J. Balsys, and K.G. Suffern. 2010. Polygonisation of Non-Manifold Implicit Surfaces Using a Dual Grid and Points. *International Conference on Computer Graphics, Imaging and Vision 2010*, 8-10th August, 2010, Sydney, Australia.

This work shows how polygons and points can be used to render implicit surfaces.

Papers accepted

Publication 4 R.J. Balsys, D.J. Harbinson, and K.G. Suffern. Determining the Point Geometry of Non-Manifold and Singular Implicit Surfaces. *IEEE Transactions on Computer Graphics.*

Papers under development

Publication 5 D.J. Harbinson, R.J. Balsys, and K.G. Suffern. Polygonisation and point hybrid rendering for implicit surfaces. *IEEE Transactions on Computer Graphics.*

Visualisation of Non-Manifold Implicit Surfaces

List of Figures

List of Tables

List of Symbols

$\in$	belongs to or is a member of
$\Re$	the set of real numbers
$f(x,y,z)-c=0$	implicit equation form
$\mid ... \mid$	absolute value of ...
$\overrightarrow{N}$	vector
$\bullet$	vector dot product
$\leq$	less than or equal to
$\geq$	greater than or equal to
$[a,b]$	interval containing set of real numbers defined by $[a,b]=\{x \mid a \leq x \leq b\}$
ε	a very small epsilon value
∞	infinity
Σ	summation symbol
∇	gradient of an implicit function
ω	opacity of a pixel
α	a tolerance value

Table 1: Definitions of implicit surfaces.

#	Surface Name	Formulae
$f[1]$	Boy's surface	$64(1-z)^3z^3-48(1-z^2)z^2(3x^2+3y^2+2z^2)+12(1-z)z$ $\Big(27(x^2+y^2)^2-24z^2(x^2+y^2)+36\sqrt{2}yz(y^2-3x^2)+4z^2\Big)$ $+(9x^2+9y^2-2z^2)\Big(-81(x^2+y^2)^2-72z^2(x^2+y^2)$ $+108\sqrt{2}xz(x^2-3y^2)+4z^2\Big)=0$
$f[2]$	Steiner's Roman	$x^2y^2+y^2z^2+x^2z^2-r^2xyz=0$
$f[3]$	Klein bottle	$(x^2+y^2+z^2+2y-1)\Big((x^2+y^2+z^2-2y-1)^2-8z^2\Big)$ $+16xz(x^2+y^2+z^2-2y-1)=0$
$f[4]$	Bretzel	$\Big(x^2(1-x^2)-y^2\Big)^2+z^2-0.01=0$
$f[5]$	Mitre	$4x^2(x^2+y^2+z^2)-y^2(1-y^2-z^2)=0$
$f[6]$	Barth Sextic	$4(t^2x^2-y^2)(t^2y^2-z^2)(t^2z^2-x^2)-(1+2t)(x^2+y^2+z^2-1)^2=0$ where $t=0.5(1+sqrt(5))$
$f[7]$	Cyclide	$(x^2+y^2+z^2)^2-2(x^2+r^2)(f^2+a^2)-2(y^2-z^2)(a^2-f^2)$ $+8afrx+(a^2-f^2)^2=0$
$f[8]$	Spiky	$x^8+y^8+z^8-x^4y^4-x^4z^4-y^4z^4-1=0$
$f[9]$	$8th$ order Chmutov	$128x^8-256x^6+160x^4-32x^2+128y^8-256y^6+160y^4-32y^2$ $+128z^8-256z^6+160z^4-32z^2+3=0$
$f[10]$	$14th$ order Chmutov	$8192x^{14}-28672x^{12}+39424x^{10}-26880x^8+9408x^6-1568x^4+98x^2+$ $8192y^{14}-28672y^{12}+39424y^{10}-26880y^8+9408y^6-1568y^4+98y^2+$ $8192z^{14}-28672z^{12}+39424z^{10}-26880z^8+9408z^6-1568z^4+98z^2$ $-3=0$

Table 2: Definitions of implicit surfaces.

#	Surface Name	Formulae
$f[11]$	Bohemian Dome	$-y^4-x^4-z^4+2x^2z^2+4y^2-2x^2y^2-2y^2z^2=0$
$f[12]$	Supertoroid	$\left(\left((\frac{x}{a})^{\frac{2}{\varepsilon_2}}+(\frac{y}{b})^{\frac{2}{\varepsilon_2}}\right)^{\frac{\varepsilon_2}{2}}-\frac{d}{\sqrt{a^2+b^2}}\right)^{\frac{2}{\varepsilon_1}}+(\frac{z}{c})^{\frac{2}{\varepsilon_1}}=0$
$f[13]$	Bohemian Star	$y^4x^4+2y^6x^2+2y^4x^2z^2+y^8+2y^6z^2+y^4z^4-4y^4x^2-4y^4z^2$ $-16x^2z^2y^2+16x^2z^2=0$
$f[14]$	Charges Surface	$\sum_{j=1}^{60}\frac{q_j}{(x-x_j)^2+(y-y_j)^2+(z-z_j)^2}-c=0.$
$f[15]$	Kusner-Schmitt	$(x^2+3)(y^2+3)(z^2+3)-32(xyz+1)=0$
$f[16]$	Steiner's Relative	$x^2y^2+y^2z^2-+x^2z^2-xyz=0$
$f[17]$	Cassini	$((x+a)^2+y^2)((x-a)^2+y^2)-z^2=0$
$f[18]$	CrossCap	$4x^2(x^2+y^2+z^3)+y^2(y^2+z^2-1)=0$
$f[19]$	Dodecube	$-\frac{1}{40}\left((-0.64+x^2+y^2)^2+(-1+z^2)^2\right)\left((-1+y^2)^2+(-0.64+x^2+z^2)^2\right)$ $\left((-1+x^2)^2+(-0.64+y^2+z^2)^2\right)=0$
$f[20]$	Tetrahedral	$x^4+2x^2y^2+2x^2z^2+y^4+2y^2z^2+z^4+8xyz-10x^2-10y^2-10z^2+25=0$
$f[21]$	Barth Decic	$8(x^2-t^4y^2)(y^2-t^4z^2)(z^2-t^4x^2)(x^4+y^4+z^4-2x^2y^2-2x^2z^2-2y^2z^2)+$ $+(3+5t)(x^2+y^2+z^2-w^2)^2(x^2+y^2+z^2-(2-t)w^2)^2w^2=0$
$f[22]$	Super Ellipsoid	$\left((\frac{x}{a})^{\frac{2}{\varepsilon_1}}+(\frac{y}{b})^{\frac{2}{\varepsilon_1}}\right)^{\frac{\varepsilon_2}{\varepsilon_1}}+(\frac{z}{c})^{\frac{2}{\varepsilon_1}}=0$
$f[23]$	Sphere	$x^2+y^2+z^2-r^2=0$
$f[24]$	Tangle	$x^4-5x^2+y^4-5y^2+z^4-5z^2+11.8=0$
$f[25]$	Ellipsoid	$\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}-1=0$
$f[26]$	Mitchell	$17x^2(y^2+z^2)-20(x^2+y^2+z^2)+4(x^4+(y^2+z^2)^2)+17=0$

Table 3: Definitions of implicit surfaces.

#	Surface Name	Formulae
f[27]	Chair	$(x^2+y^2+z^2-ac^2)^2-b\Big((z-c)^2-2x^2)\Big)\Big((z+c)^2-2y^2)\Big)=0$
f[28]	Hunt	$4(x^2+y^2+z^2-13)(x^2+y^2+z^2-13)(x^2+y^2+z^2-13)$
		$+27(3x^2+y^2-4z^2-12)(3x^2+y^2-4z^2-12)=0$
f[29]	Bicorn	$y^2\Big(a^2-(x^2+z^2)\Big)-(x^2+z^2+2ay-a^2)^2=0$
f[30]	Star	$-3.849606112048\Big((x-z)(0.309016994219x-0.951056516346y-z)$
		$(-0.809016994568x-0.587785252027y-z)$
		$(-0.809016994086x+0.587785252691y-z)$
		$(0.309016994999x+0.951056516092y-z)\Big)+(1.-0.831253875555z)$
		$(x^2+y^2-1.0+1.927050983125z^2)^2=0$
f[31]	Cayley	$-5(x^2y+x^2z+y^2x+y^2z+z^2y+z^2x)+2(xy+xz+yz)=0$
f[32]	Umbrella	$x^2-yz^2=0$
f[33]	Mobius	$(t^2+u^2)^3-4bt^2u^2=0$
		where $t=cos(\frac{-atan2(y,x)}{2})\Big(xcos(atan2(y,x))+ysin(atan2(y,x))-a\Big)$
		$+zsin(\frac{-atan2(y,x)}{2})$
		and $u=-sin(\frac{-atan2(y,x)}{2})\Big(xcos(atan2(y,x))+ysin(atan2(y,x))-a\Big)$
		$+zcos(\frac{-atan2(y,x)}{2})$

Acknowledgments

I would like to give acknowledgment to all those who assisted me during my time of study at Central Queensland University. Much praise and thank you to my supervisor Ron Balsys, Central Queensland University, who first introduced me to implicit surface research in my honours year. He has provided excellent guidance throughout this thesis, and I have gained many skills as a result of his expertise.

Also thanks to Kevin Suffern, University of Technology Sydney, who reviewed and provided feedback throughout the course of this work. I look forward to further collaborations with Ron and Kevin.

Declaration

This thesis contains no material that has been accepted for the award of another degree. Furthermore, to the best of my knowledge, this thesis does not contain any material previously published or written by another person, except where due reference is made in the text.

.....

Dirk Harbinson

Chapter 1

Introduction to implicit surfaces

This thesis contributes robust methods to visualise manifold and non-manifold implicit surfaces, particularly on issues with respect to rendering non-manifold surfaces. The methods have been designed to be parallel in execution, where possible, for execution on both multi-core CPU's and GPU's. The GPU algorithms were written in *NVIDIA CUDA*. The techniques allow for real-time interaction and stereoscopic viewing. The methods also enable the exploration of families of curvature surfaces, that haven't previously been practical to render.

While the surface shape of many implicit surfaces are well known, the visualisation of the shapes of more complex surfaces, particularly those with non-manifold or sharp features, has been less well studied. Our objective for this research was twofold. First we wish to be able to explore the shape of complex implicit surfaces in as close to real-time as possible. In this context, by explore the shape, we mean to be able to arbitrarily rotate the shape around a vector in 3D space. We also wish to be able to create stereo pairs of the surfaces for viewing in virtual reality systems.

Our second objective was to be able to explore parameterised surfaces, such as the cyclide and super-quadratic surfaces, and vary their parameters so that the effect of the parameter change on the surface shape is immediately obvious.

Examination of the forms of surfaces such as Boy's surface, Steiner's Roman surface and the Klein bottle surface convinced us that for us to be able to produce meaningful results we would have to be able to render features such as rays, cusps, singular points and thin sections to the pixel resolution of our display, for us to faithfully capture the nature of curvature surfaces of these surfaces.

The implicit surfaces being investigated are complex, e.g. include singularities or sharp features. Proper understanding of the surfaces through visualisation is dramatically enhanced by several techniques, such as contouring, curvature maps and intersections.

We propose a visualisation model that combines point primitives and triangles to represent surfaces. Points are used to render non-manifold regions while polygons are used elsewhere. These primitives are all rendered to the same colour buffers for seamless visualisation between manifold and singular sections. The enclosed DVD demonstrates animations using the presented algorithms.

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 $f(x,y,z) - c = 0$, where $c \in \Re$. Implicit surfaces are specified by functions of the form;

$$f(x,y,z) - c = 0 \tag{1.1}$$

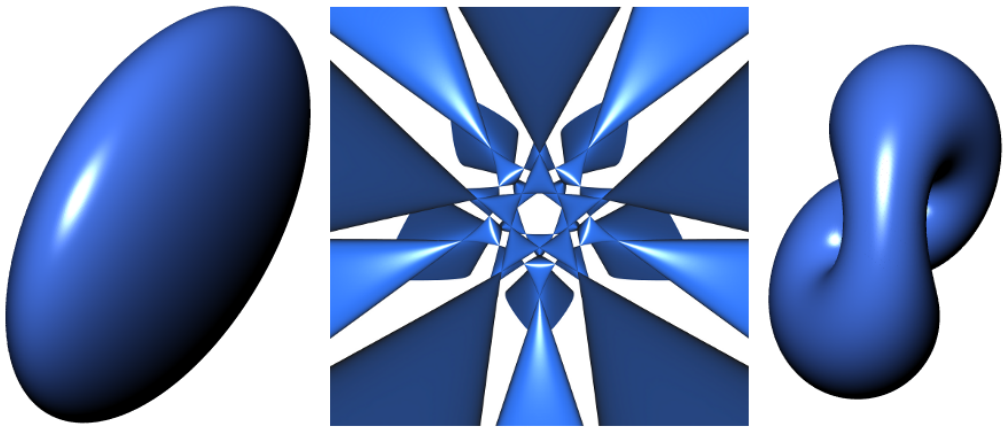


Figure 1.1: (a) Ellipsoid f[25] . (b) Barth Sextic f[6] . (c) Klein bottle f[3] .

such as the function defining the implicit surface, Steiner's surface;

$$f(x,y,z) = x^2y^2 + y^2z^2 + x^2z^2 - r^2xyz \tag{1.2}$$

A fundamental property of implicit functions is their ability to partition space. An implicit function partitions space into three categories, inside, outside and on the surface. By convention, space inside the surface is given by $f(x,y,z) < 0$, space outside the surface is given by $f(x,y,z) > 0$. The visualised points on the surface represent where $f(x,y,z) = 0$.

Implicit surfaces can be manifold or non-manifold. Manifold surfaces are those like a sphere or an ellipsoid (Figure 1.1 (a)). Non-manifold surfaces have regions such as singular points (Figure 1.1 (b)) or points of infinite curvature (Figure 1.1 (c)). A number of problems therefore occur in rendering implicit functions, particularly on, and around, these non-manifold features.

A fundamental property of all continuous surfaces is the curvature of the surface. Curvature describes how curved a surface is at any point on the surface. The Gaussian curvature $K(x,y,z)$ of an implicit surface $f(x,y,z) = 0$ is a 3D scalar field. A specific value of this scalar field $K(x,y,z) = k$, with $k \in \Re$ is another implicit surface. We call this the curvature surface [5] of $f(x,y,z) = 0$. If $k \in [kmin, kmax]$ for $f(x,y,z) = c$, these two surfaces will intersect.

Generating curvature surfaces is an open problem in rendering, especially for non-manifold surfaces. Curvature surfaces are commonly used in CAD / CAM applications where curvature is an important intrinsic quantity of the surface.

Surfaces can be visualised using a number of techniques, such as polygonisation, ray-tracing, or using points. Polygonisation is the generation of a triangle mesh, requiring a closed mesh for hole-free rendering. This mesh can be precomputed and later viewed at faster speeds than non-primitive based methods like ray-tracing. The production of a mesh also permits engineering applications, such as Finite Element Analysis (FEA).

Ray-tracing typically re-samples the surface for each rendered image, limiting applications for interactive visualisation. The modern GPGPU (General-purpose computing on graphics processing units) era has sparked new interest into real-time ray-tracing work, including implicit surface visualisation.

Point-based rendering has a different philosophy to polygonisation and ray-tracing. Point work discards connectivity information and opts for visualising scenes through dis-

crete point samples. When each triangle in a polygonal mesh projects to a single pixel on the image plane, we can get the same results from specifying a single point versus three points for a triangle [6].

All methods employ common lighting models, such as Phong shading [7], to illuminate the surface and provide visual cues. The normal $\overrightarrow{N}$ required in the Phong equation can be computed from the implicit surface in multiple ways. If the function is differentiable, the surface should have a defined normal everywhere; if the polynomial is not differentiable, then the surface contains a singularity [8]. An approximated normal can be generated from sampling the direction of the scalar field. The latter is popular for methods that synthesise implicit functions at run-time and compile into GPU programs, enabling the user to type in implicit surfaces without the functions being hard coded into the application.

As an aid to visualisation, rendering the intersections of surfaces helps illustrate the nature of the surface. An example is to render the intersection of a number of planes, orthogonal to the three coordinate axes, with the surface being visualised. These coordinate planes help the viewer obtain more information than reflection models, such as Phong shading, can supply.

Texturing, shadowing and transparency are other visual cues that improve the interpretation and visualisation of implicit surfaces, such as Figure 1.2 (a).

In chapter two we investigate anti-aliasing of point-based surface renderings. This

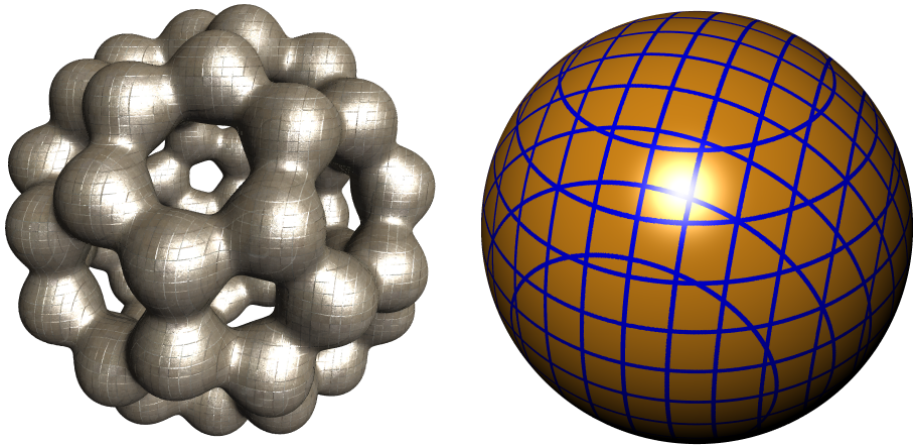


Figure 1.2: (a) Visualisation of a 3D charges surface with texturing f[14] . (b) Sphere rendered with planar contour lines f[23] .

significantly increases the quality of the final images. Chapter three looks at the subdivision and rendering process, and how well it can be controlled using select criteria such as intervals.

In chapter four we address the limitations in the methods from the previous chapter. We present techniques to reduce false-positives. Here we are interested in obtaining a set of points on the surface and their associated normals. In chapter five we propose using a combination of points and polygons to render a surface, reducing geometry needed to render.

Lastly, chapter six presents conclusions on the methods presented in the thesis and chapter seven discusses future work.

1.1 Significance of Research

Implicit surfaces have many uses in mathematics, science and engineering as they are the iso-valued surfaces of scalar fields. Hence visualisations of implicit surfaces are useful whenever we are rendering scalar fields. Also, even though Computer Aided geometric Design (CAD) uses predominately parametric surfaces, implicit surfaces are also used in this area. Blending and offset surfaces in CAD are often implicit surfaces. As implicit surfaces are equations of the general form $f(x,y,z)-c=0$ the proximity of a given point to the surface can be tested by calculating the sign and magnitude of the implicit function at the point. Parametric surfaces cannot be used for this task.

Implicit surfaces also find uses in real world problem solving in science and engineering. The use of octrees as a spatial data structure greatly simplifies operations such as finding surface intersections, locating a point or block in space, dealing with hidden surface removal, connected neighbour finding, anti-aliasing, and polygonisation of polygons, and in solving other topological problems. Thus octrees are useful in the problem domains of solid modelling, computer graphics, computer aided design, computer vision, image processing and robotics [9].

In the manufacturing domain, Beck *et al.* [10] discuss surface cutting by numerically controlled (N.C.) milling machines. They state that algorithms previously used failed to adequately cut the surface due to localised anomalies in the surface that were not detected using simple rendering techniques. They assert that the anomalies are bypassed by per-

forming detailed curvature analysis of the surface to be machined. Such analysis reveals the variations in the magnitude and direction of the principal curvatures across the surface. The extreme values for the curvature constrain the tool size for gouge-free milling. They also assert that curvature maps, that show the range of curvature across the surface are valuable. Curvature analysis is also used extensively in the aeronautical and automotive manufacturing industries.

This thesis makes an original contribution to knowledge in visualisation of non-manifold surfaces, and in visualisation of important properties of implicit surfaces such as the mean and Gaussian curvature. For each implicit surface we can define an infinite series of curvature surfaces whose complexity may increase exponentially along the sequence. To our knowledge, the only published image of families of curvature surfaces is in Suffern *et al.* [11]. They only rendered the first curvature surface of an ellipsoid, which is another ellipsoid.

This area is therefore relatively un-explored territory, which makes it a worthwhile project in mathematics. No-one knows what the curvature surfaces of many interesting surfaces (such as Boy's surface) look like, or the potential applications they may have. It's also worthwhile because of the computational and rendering challenges that are involved. Depending on the original implicit surfaces, we may only be able to render the first few curvature surfaces in the sequences. The reason is the increasing complexity of the equations and the attendant increase in geometric complexity of the surfaces. It is interesting to see how far our current computational and rendering techniques can be

pushed, or adapted, to render these surfaces, or if new techniques will be required.

Any necessary advances in our techniques will benefit the rendering of implicit surfaces in other areas. Improving the rendering and visualisation methods of implicit surfaces ultimately has applications in games, the entertainment industries, and manufacturing, science and engineering. Implicit surfaces have also been used for modelling organic forms, such as trees, leaves and arms as discussed in Bloomenthal [12]. Bloomenthal also reviews other visualisation applications, such as blobby/molecular modelling, of use in the materials science, chemical and bio-medical areas.

1.2 Literature Review

Computer based visualisation is the rendering of multi-dimensional datasets, typically onto a flat two-dimensional display device. The data must be transformed between spaces e.g. three-dimensional data projected onto a two-dimensional plane, as described in Foley [13].

To render implicit surfaces we must first define a viewpoint in the coordinate system, with the same dimensionality as the function, e.g. 2D/3D/4D. The viewpoint can be thought of as a virtual camera directed at the geometry; our display device provides a visualisation of the cameras view. The resulting rendered image is a projection of the function onto a two-dimensional plane. There are a number of techniques in common use for rendering surfaces, the main ones being polygonisation, ray-tracing, and point-based

techniques. They all have their advantages, disadvantages, and limitations, as discussed later. Figure 1.3 illustrates the stages of each render method beginning with the surface equation to the final render.

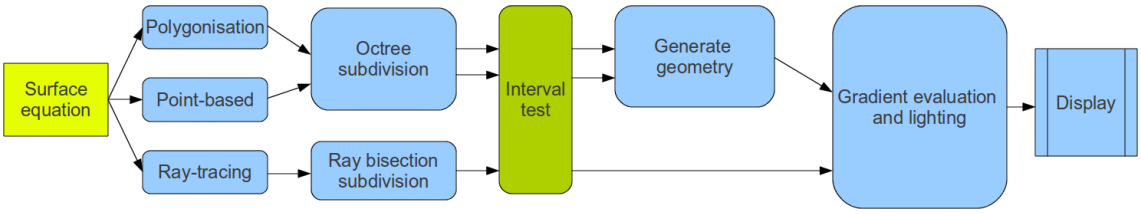


Figure 1.3: The steps taken to produce a rendering using polygonisation, point-based and ray-tracing techniques.

In this thesis we first present techniques for rendering static images of implicit surfaces. We then present techniques to produce animations of surfaces whose 3D shape is hard to conceptualise from static images. We also do animations of parameterised implicit surfaces where we change the parameter as a function of time. To produce these interactive animations, we need to accelerate the rendering process with techniques like space partitioning [13].

1.2.1 Space Partitioning

Rendering systems require the traversal of spatial datasets, in order to produce an image. The speed at which spatial data can be processed is crucial to a rendering algorithms efficiency. Many common data structures are employed in graphics rendering, such as BSP [14], Quadtree [15], Octrees [15] and k-D trees [16]. Their purpose is to provide an

efficient mechanism for querying spatial data. This process is known as a spatial partitioning scheme.

In respect to implicit visualisation, an implicit function may be evaluated millions of times, over a vast area of space. A brute force search over this space becomes costly, and infeasible for producing images of higher quality.

The spatial search needs to be optimised by partitioning the function space into sections. A popular partitioning scheme used during surface visualisation is octree based subdivision. The interested reader is referred to books on this subject, such as those by Samet [15]. Octree methods are most useful as they integrate well with interval arithmetic. Octree methods begin with a root node (a bounds defined in the same dimension as the function), then is split in the middle of each axis to create child nodes. This recursive process is repeated until a specified level of depth is reached.

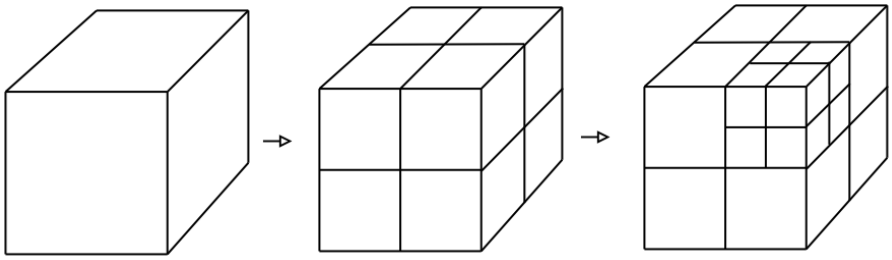


Figure 1.4: Octree showing subdivision of root node into *eight* child nodes.

Octree subdivision aids implicit visualisation in a couple of ways. Firstly, it allows subdivision to occur in an adaptive manner. Nodes that do not contain any part of the surface can be skipped, while remaining nodes can be subdivided to a required detail

level. Secondly, octree information can be retained for generation of polygonal meshes, or further processing, for example of non-manifold sections.

Subdivision can be controlled by a combination of criteria. Point sampling can be used to detect surface intersections against the octree node. Gradient information can be used to detect surfaces inside a node. Curvature information can be used to drive minimum and maximum subdivision levels. Interval analysis can be used to detect if a surface is inside a node, preserving sharp features.

1.2.2 Interval Methods

An interval $I = [a,b]$, $a \leq b$ is a set of real numbers defined by $[a,b] = \{x \mid a \leq x \leq b\}$. Interval Arithmetic defines operations for adding, subtracting, multiplying and dividing intervals, as shown in the following equations.

$$[a,b] + [c,d] = [a+c, b+d] \tag{1.3}$$

$$[a,b] - [c,d] = [ad, bc] \tag{1.4}$$

$$[a,b] * [c,d] = [min(ac,ad,bc,bd), max(ac,ad,bc,bd)] \tag{1.5}$$

$$[a,b]/[c,d] = [min(a/c,a/d,b/c,b/d), max(a/c,a/d,b/c,b/d)] \tag{1.6}$$

See Moore [17], Suffern and Fackerell [18] and Snyder [19] for a full discussion on the use of intervals in computer graphics.

The *natural interval extension* of a rational function $f(x_1, \ldots x_n) : \Re^m \rightarrow \Re$ denoted by $F(X_1, \ldots X_n)$, is obtained by replacing the x_i's by the intervals X_i and evaluating the resultant interval expressions according to the rules of interval arithmetic.

The main reason for using intervals in this context comes from the property that the natural interval extension of a function provides bounds on the variation of the function. The bounds may not be very tight, but the values of the function f are guaranteed not to be outside of this interval. Stolte [20] and Suffern and Balsys [11], show that octree based volume decomposition with a subdivision criteria based on an interval exclusion test, converges to nodes clustered about the implicit surfaces. Using this approach they rendered many different types of surfaces.

The work in this thesis implements interval arithmetic as a *C++* class and details of the algorithms can be found in Balsys and Suffern [21]. An interval package is also implemented in *NVIDIA CUDA* and *NVIDIA* CG for interval processing on graphics hardware, in both a GPGPU and shader based context, respectively.

As the complexity of a function increases, interval arithmetic suffers from overestimating the bounds of the function. Comba and Stolfi [22] developed Affine arithmetic (AA) to produce tighter bounds than possible using traditional interval arithmetic. AA achieves tighter bounds by storing additional information during calculation, keeping track of correlations between AA operations. Consequently, AA is more demanding on computer resources than IA (in both memory and computation) which can offset the gains

of tighter bounds on the end results. The choice to use either IA or AA can ultimately only be determined by executing the rendering algorithm, as early rejection (at an added computation cost) may not be ideal for best performance. Generally AA is used when a tighter surface fit is required at a given plot depth and time is not an issue.

1.2.3 Polygonisation

Early renderings of implicit surfaces were done by constructing a triangle mesh from the implicit surface. This is classified as forward mapping, where geometry is projected forward onto the image plane, then rasterised to pixels. Rasterisation is the projection of a primitive onto a discrete pixel grid, thus the primitive plays an important role in the efficiency and appearance of the image. In the past, renderings were done with coarse triangle meshes, which projected to large numbers of pixels, filling the image in the minimum amount of time.

Early work on polygonising implicit surfaces is by Lorenson and Cline [23] and Bloomenthal [24]. Lorenson and Cline formulated the novel marching cubes algorithm, which extracts a polygonal mesh from a three-dimensional scalar field. The field is divided into voxel cubes where each cube is polygonised separately, the final mesh is comprised of all the polygons from each cube. Bloomenthal devised a method of polygonising implicit surfaces by performing bi-sections on cube edges to have more accurate triangle meshes.

The advantage of assembling a surface into a polygonal mesh is the ability to render

images from any viewpoint, without recalculation of the implicit surface. Polygonisation schemes revolve around either a marching cubes style algorithm [24] or stitching triangles from intersections inside an octree node [25]. Recent work by Balsys and Suffern [25] continued work on polygonisation, aiming to improve the correctness of the surface mesh solution, and reduce problems such as holes in the polgyonisation process, to produce higher quality images. They also use adaptive subdivision driven by curvature, creating a mesh with varying level-of-detail.

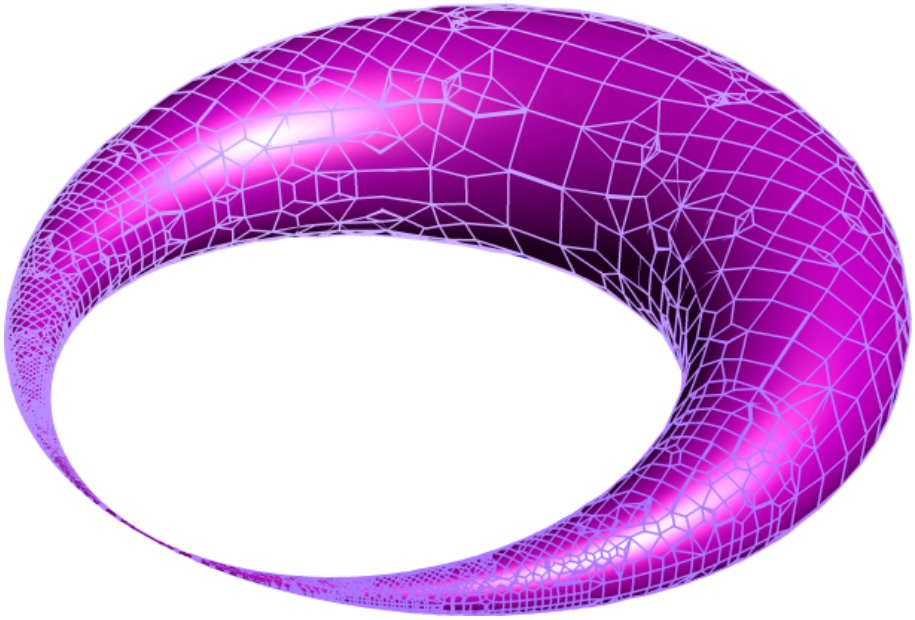


Figure 1.5: Cyclide surface with adaptive tessellation.

Regular implicit surfaces can be polygonised for rendering, but the process is not as straightforward as it is for parametric surfaces. A large number of papers have been published on the polygonisation of implicit surfaces. Relevant examples include: Wyvill *et al.* [26], Lorenson and Cline [23], Bloomenthal [24], Schmidt [27] and Balsys and

Suffern [4]. Thin sections, even if they are regular parts of surfaces, can also be difficult to polygonise, as discussed in Balsys and Suffern [4].

Ohtake and Belyaev [28] used QEM (Quadratic Error Metrics) to measure the quadratic energy inside the octree node. Vertex positions were moved to where there is minimal quadratic energy. This created a better quality mesh that preserved sharp features. Lewiner *et al.* [29] aimed to solve topology problems with traditional marching cubes output. They produce a crack-free manifold mesh where previous marching cube implementations had failed. Ohtake *et al.* [30] furthered work on sharp feature preservation. Instead of using QEM they use a mesh evolution formula which applies an attracting force to a vertex, repositioning it to closely fit the surface.

Dual marching cubes by Schaefer and Warren [31] was a new application of marching cubes to the dual-grid structure. They produce a grid which is topologically dual to the octree structure, and then create non axis-aligned boxes using the dual-grid contour lines. Each vertex is positioned on sharp features by solving a QEF (quadratic error function). Marching cubes is then applied to these generated boxes. Ju [32] presented work that can extract an intersection-free mesh from volumetric data. Paiva *et al.* [33] enhanced the dual-grid marching cube approach by sliding vertices into triangles that had a better aspect ratio.

Polygonisation can break down for complex surfaces that are non-manifold, indicating a more robust operation is needed. In chapter five we show how to combine polygons and

points to create seamless implicit surface renderings.

1.2.4 Points

Triangle based methods have received attention as the most efficient and practical 3D rendering solution. Each new generation of graphics hardware offers a magnitude of increased triangle processing capability over the last. These technological advancements mean higher quality models are used in graphics rendering.

As triangle counts increase in a model, their individual contribution to the final image is reduced. This can reach a point where each triangle projects to a single pixel on the image plane; thereby the triangle representation and rasterisation process is no longer optimal. At this stage, a reduction in processing can be achieved from simply rendering models explicitly as points [6]. We can get the same results from specifying a single point verses three points for a triangle. Chapter four is concerned with rendering implicit surfaces using point-based methods.

Levoy and Whitted [6] first introduced the use of points as a rendering primitive in 1985. A sparse collection of papers have cropped up from then until the late 1990s. Since the year 2000 a great deal of interest has arisen in using models comprised of points, as laser scanning devices became more commonly used. Laser scanners produce their data as scattered point samples (collectively known as a point cloud), such as Stanford's scanning repository [34]. These scanned points would normally be used in a conversion

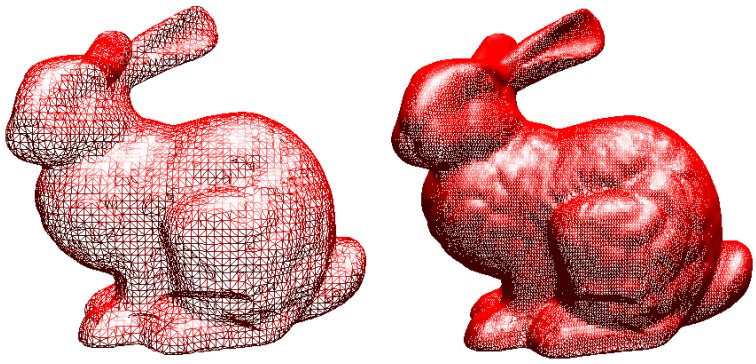


Figure 1.6: (a) Moderate triangle count. (b) High triangle counts.

algorithm to construct a triangle based mesh, which could then be rendered using standard techniques. When dealing with objects that feature millions of point samples, constructing a triangle mesh becomes difficult. Point-based methods enable this data to be visualised in its native form, which is highly considering resources. As sampling an implicit function results in a point that lies relative to the implicit surface, visualising these points directly is a robust solution as no further processing of the geometry is required. Gross [35] recently discussed the growing use of points as a display primitive.

Point-based rendering techniques use points, or objects such as discs, as rendering primitives. Rockwood [36] is an early paper on rendering parametric surfaces using points. Witkin and Heckbert [37] used repulsive particles called floaters that can slide over implicit surfaces for their modelling and rendering. The repulsion creates uniform distributions of particles on the surface which are rendered as discs. They further refined the results in [38] to handle more complex models.

De Figueiredo and Gomes [39] used physically based particles that obey Newtonian equations of motion to render differentiable implicit surfaces. They used points for rendering, but did not produce shaded images. Rosch *et al.* [40] used large discs to interactively render self intersecting, singular, algebraic implicit surfaces with non-manifold points (cusps).

Work by Tanaka *et al.* [41], [42] used particles that obey stochastic differential equations to render twice differentiable implicit surfaces. The points are evenly distributed over the surfaces, and surface intersections can also be rendered. These are nice features of their work, but the method is complex, and would be a lot of work to implement from scratch. Shaded images of single surfaces are produced by splatting with discs.

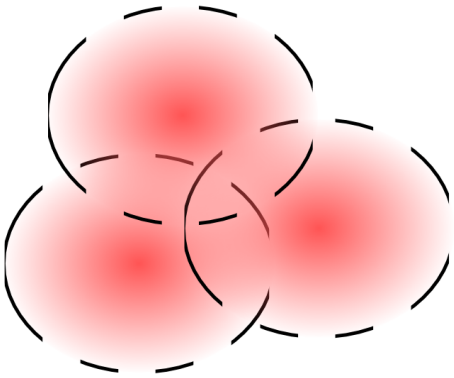


Figure 1.7: Illustration of a surfel disc, used to describe a point with dimensions in 3D space.

Pfister *et al.* [43] introduced the concept of the surface element or surfel. A surfel is an extension of the point primitive, with the following characteristics. It faces a direction using a normal, and it has dimensions. Like polygons it can also store other attributes

like colour, diffuse, specular components etc. Pfister *et al.* developed algorithms to render geometries based on these elements, and used surface splatting for visibility testing.

Similarly, Rusinkiewicz and Levoy [44] developed an algorithm that speeds up rendering of point-based objects based on splatting. Zwicker *et al.* [45] used the elliptical weighted average filter, EWA, of Heckbert [46], to avoid aliasing artifacts with textures when using splatting to render point clouds. In chapter five we compare splatting points sampled on the implicit surface with polygon/points and ray-tracing.

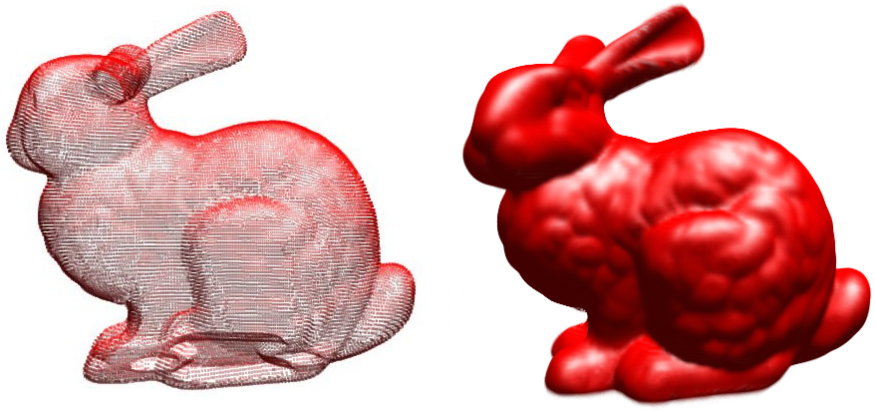


Figure 1.8: (a) Sparsely sampled point model. (b) The same model rendered using EWA splatting for a solid rendering.

More recently, Botsch *et al.* [47] based the lighting of a splat on a linearly varying normal field and showed the resulting Phong splats are superior to that of previous work. They give a simple and effective method of constructing the Phong splats for given input data sets. Zwicker *et al.* [48] use homogeneous coordinate systems to produce perspective accurate splats which provides high image quality with anisotropic texture filtering. They

also present an extension of their rendering primitive to deal with sharp edges and corners, such as those that occur in non-manifold surfaces. We also use Phong shading on the splats in chapter five to improve the quality.

Co *et al.* [49] developed an algorithm for rendering iso-surfaces using a point-based method. As their first step they generate points in the domain of the sample space. In the second step these points are projected onto the iso-surface of interest in the domain. The resulting point set is then rendered using a point splatting approach.

In Balsys and Suffern [50] is a sampling technique that uses octree space subdivision with a natural interval exclusion test to minimise the octree subdivision process. They used a similar technique for polygonising implicit surfaces in Balsys and Suffern [4]. With this technique, they can render implicit surfaces with times comparable to parametric surfaces. Their intention is to produce high quality images where there is no possibility that portions of the surface are either rendered incorrectly or are missed, and the use of interval techniques is ideal for this purpose [18].

Balsys *et al.* [51] also presented criteria for the complete pixel coverage of implicit surfaces. This algorithm was a major contribution of the paper. Other than stopping the rendering, this technique ran automatically with no user intervention. It also has the following advantages over previous point-based rendering algorithms for implicit surfaces; it's simple, in the case of implicit surfaces much faster, it will not miss small surface sections in the plotting volume, and if rendered to the required plot depth will have no

rendering artifacts (holes).

A disadvantage of the work was that for certain surfaces rendering artifacts occur due to the use of the interval subdivision method, and rays were rendered as tubes. In chapter four we reduce these artifacts by pruning away excess nodes around these non-manifold regions.

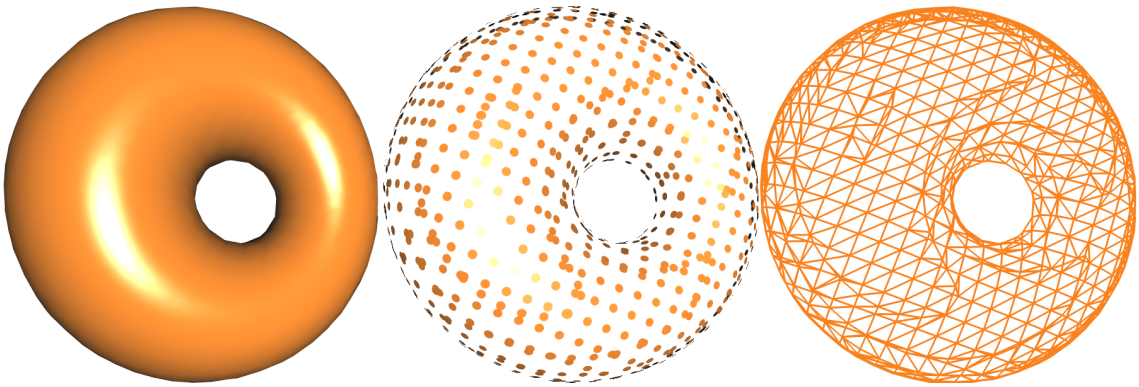


Figure 1.9: (a) Rendered model. (b) Point representation. (c) Triangle representation.

1.2.5 Ray-tracing and GPGPU

In chapter four and five we compare our point and polygon methods to recent work in ray-tracing implicit surfaces [52]. Unlike our forward rendering solution, ray-tracing (backward mapping) traces rays from the viewpoint through a pixel on the image plane. A vector can be defined from the eye position and view direction that can be used to find where the ray first intersects surfaces in the view volume. This process is usually referred to as ray-casting, if the ray is traced to the first surface hit in the scene, or ray-tracing

when the colour contribution to a pixel is determined from multiple reflections.

Rays can be bounced around geometry, making possible complex lighting solutions for photo-realistic rendering. This type of system is termed an off-line renderer, aiming for very high quality images over speed, and it can take a matter of hours to render a single frame of complex scenes at high resolution. This is commonly done in the animation and entertainment industries. Its slow speed is attributed to the fact that, in the past, it was a pure software based solution (CPU). The advancement of both CPU and GPU technology is proving scalable to produce practical ray-tracing solutions.

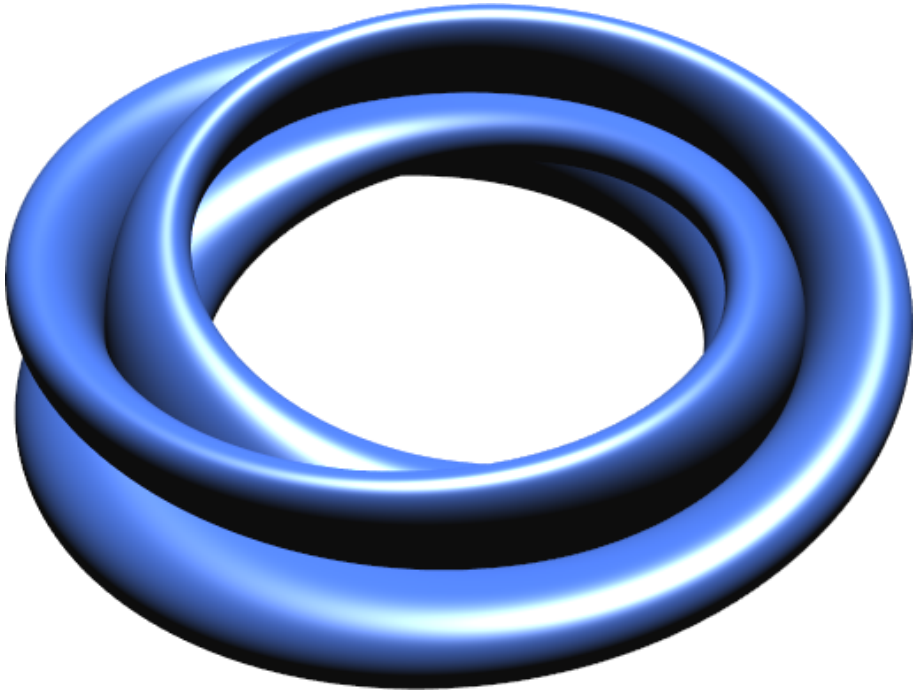


Figure 1.10: Mobius surface f[33] ray-casted on GPU.

Implicit surfaces can be ray-traced in a number of ways. A generalised ray-intersection test can be constructed for a surface, then integrated into a ray-tracing system. Solving

the intersection test becomes immensely difficult as the surface equation increases in complexity, the intersection test must solve all possible roots of the function.

This approach is therefore only suitable for simple surface formulae without many root solutions. Another limitation is the accuracy of non-manifold features such as rays. If the projected ray does not align with a non-manifold feature, it will not show up in the rendered image. Performing a ray-intersection with an implicit surface can miss these important visual features.

Other works [53], [52] have shown that it is not necessary to ray-trace surfaces by explicitly solving the intersection test. Their approach is similar to the interval and sign testing used in implicit polygonisation. The projected rays are partitioned into sections, which can be tested against the function. They show interactive speeds can be achieved through processing the rays in parallel. CPUs that feature multiple cores increase computational power exponentially, through their parallel architecture. Knoll *et al.* [53] uses multi-core CPUs along with SIMD functionality to render implicit surfaces at a few frames per second.

The GPU is even more parallel in design compared with the CPU. As of 2007, GPUs have overtaken CPUs in raw processing power, with high-end graphics cards featuring up to 128 parallel execution units, whereas CPUs currently only have a handful of cores. Unlike the CPU, the GPU is limited to SIMD style processing. *NVIDIA* refers to their design as Single Instruction Multiple Threads (SIMT) processing, in order to maintain high

performance. Conditional logic branching causing thread divergence is not optimal when performed on the GPU, and considerably lowers its maximum theoretical performance. Diverging threads must be serialised and executed in order, until the threads converge and parallel execution can continue. Singh and Narayanan [52] compared logic driven implicit rendering and brute force implicit rendering. As expected using conditional branching, they report logic based methods (such as interval calculations) are inefficient to implement on the GPU. On the other hand, brute force SIMD algorithms were able to get promising speeds in the hundreds of frames per second for 512x512 pixel ray-cast images of implicit surfaces.

However, the accuracy and correctness of the renderings breaks down for certain non-manifold surfaces, such as for Steiner's surface where converging tubes appear in the image instead of as singular lines. This is discussed in the ray-tracing comparisons in chapter four.

1.2.6 Anti-aliasing

A computer rendered image is the result of discretely sampling a continuous dataset. These discrete samples form a raster image, which is displayed on a computer monitor. Discrete sampling a continuous dataset can cause aliasing problems, resultant images will be sharp and jaggy looking such as stair-casing. Anti-aliasing is a technique to produce smooth and production quality images, that are free from such artifacts. In surface visualisation, aliasing is most noticeable on the silhouette edges of surfaces, where the surface

(a) (b)

Figure 1.11: (a) Common aliased staircase effect, and (b) The result of an anti-aliasing method.

occludes 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 well as along contour edges. Non-manifold surfaces suffer the worst, with thin sections appearing jaggy and inconsistent. Anti-aliasing techniques can address the entire image (known as super-sampling) or selectively work on troublesome regions, like polygon edges (multi-sampling). Methods like multi-sampling are an optimisation of super-sampling. Multi-sampling evaluates the pixel colour once, while the depth is sampled multiple times. In chapter two we present methods for anti-aliasing point-based renderings and compare results with existing methods.

Zwicker *et al.* [45][48] used the elliptical weighted average filter, EWA, of Heckbert [46] to avoid aliasing problems with textures when using splatting to render point clouds. Anti-aliasing for polygonal surfaces can be performed using the built-in features of modern graphics hardware. Multi-sampling (or super-sampling, see Foley & Van Dam [13])

of scenes composed of polygons is very effective. When anti-aliasing polygon scenes the geometry's complexity remains constant, regardless of the anti-aliasing 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 [5], [54], [51], and [1], were not anti-aliased, as it was not clear how to proceed with anti-aliasing point rendered scenes. In a native point approach, the anti-aliasing scheme of Zwicker *et al.* [45] and Heckbert [46] is not applicable and in neither of these is aliasing on the silhouette edge, or contour edges, fixed.

Balsys *et al.* [5] rendered anti-aliased 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 were 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 anti-aliasing 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 anti-aliasing, see for example Foley *et al.* [13] or Whitted [55].

Stolte *et al.* [20], and Stolte [56] used interval arithmetic for voxelising implicit surfaces and 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. Anti-aliasing

is not discussed.

Bloomenthal [57] discusses anti-aliasing of Z-buffer programs. This work developed a simple algorithm that can be used to anti-alias the silhouette edges of complex connected regions. This work is an important original contribution to anti-aliasing what are essentially bitmaps.

As is the case with this dissertation, Balsys *et al.* [51] also rendered implicit surfaces using points. For certain surfaces, rendering artifacts occurred due to the use of the interval subdivision method, for example singular lines (rays) were rendered as tubes. Also aliasing was evident on silhouette edges, along surface contours, and along the singular lines.

1.2.7 Contouring and Curvature

A better understanding can be achieved by rendering visual cues onto the surface, such as intersection/contour lines or curvature maps. In chapter four we produce renderings of surfaces with contouring, curvature maps and intersections with other surfaces. Balsys *et al.* [54] used point-based algorithms for rendering the intersections of a surface with a series of level surfaces of other scalar fields $g_i(x,y,z), i = 1..n$. One example is the intersection of an ellipsoid with the curvature of the curvature surface of an ellipsoid. These are the surfaces $g_i(x,y,z) - c = 0$ for different values of c. The intersections are rendered as contour lines on $f(x,y,z) = 0$. They visualise the intersections of various

planes [12] with implicit surfaces. Alternatively, the scalar fields may be functions related to the implicit surface such as its curvature function, which wish to be visualised on the surface.

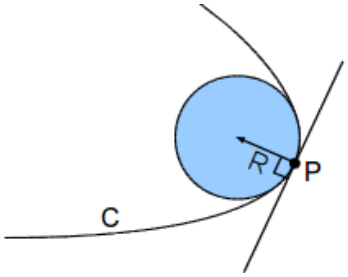


Figure 1.12: The osculating circle tangent to the plane curve.

In chapter four we also visualise curvature surfaces. Curvature is measured as the radius of the osculating circle on the plane curve. In two-dimensions, this is illustrated in Figure 1.12 by positioning a circle at point P tangent to the plane curve C. The radius R of the osculating circle represents the level of curvature. In three-dimensions the curvature at point P will change depending on the orientation of the normal plane. This results in a minimum and maximum curvature value at any point on the curve. The maximum curvature value is known as the maximum principle curvature κ_1, and the minimum curvature value is known as the minimum principle curvature κ_2.

Gaussian curvature, introduced by Carl Gauss, is the product of the two principal curvatures $K = \kappa_1\kappa_2$. Gaussian curvature classifies points on a surface as elliptic, hyperbolic and parabolic. A point is elliptic if $K(P) > 0$. A hyperbolic point is when $K(P) < 0$ and a parabolic point is when $K(P) = 0$.

A curvature map is the visualisation of curvature values across the surface, such as Gaussian curvature. A scalar field they consider in [54] is the Gaussian curvature of $f(x,y,z) = c$, which is an intrinsic property of the surface.

Spivak [8] gives general formulae and examples for the Gaussian curvature of implicit surfaces; see also Mitchell and Hanrahan [58] for an independent derivation of the formulae. Potzmann and Opitz [59] presented formulae for curvature analysis of curvature functions. Guid, Oblosek and Zalig [60] asserted curvature is one of the most important tools for surface analysis. Balsys and Suffern [4] presented general implicit equations to produce a Gaussian curvature function from an implicit function.

According to Spivak the principle curvatures κ_1 and κ_2 are given by

$$\kappa_i = -\frac{1}{\sqrt{(f_x^2 + f_y^2 + f_z^2)}}\lambda_i, \tag{1.7}$$

where $f_x = \frac{\partial f}{\partial x}$, $f_y = \frac{\partial f}{\partial y}$, $f_z = \frac{\partial f}{\partial z}$, and λ_i are the roots of the quadratic equation

$$det \begin{pmatrix} f_{xx} - \lambda & f_{xy} & f_{xz} & f_x \\ f_{yx} & f_{yy} - \lambda & f_{yz} & f_y \\ f_{zx} & f_{zy} & f_{zz} - \lambda & f_z \\ f_x & f_y & f_z & 0 \end{pmatrix} = 0, \tag{1.8}$$

with $f_{xx} = \frac{\partial^2 f}{\partial x^2}$, $f_{xy} = \frac{\partial^2 f}{\partial x \partial y}$, etc. From this expression Balsys and Suffern [4] give the expressions for the Gaussian curvature, K, and the mean curvature [8], H as:

$$K = -\frac{A}{(f_x^2 + f_y^2 + f_z^2)^2} \tag{1.9}$$

where

$$\begin{aligned} A &= 2\Big[f_x f_y (f_{xy} f_{zz} - f_{xz} f_{yz}) + f_x f_z (f_{xz} f_{yy} - f_{xy} f_{yz}) \\ &+ f_y f_z (f_{yz} f_{xx} - f_{xy} f_{xz})\Big] - f_x^2 (f_{yy} f_{zz} - f_{yz}^2) \\ &- f_y^2 (f_{xx} f_{zz} - f_{xz}^2) - f_z^2 (f_{xx} f_{yy} - f_{xy}^2), \end{aligned} \tag{1.10}$$

and

$$H = \frac{B}{2(f_x^2 + f_y^2 + f_z^2)^{\frac{3}{2}}}, \tag{1.11}$$

where

$$\begin{aligned} B &= -2(f_x f_y f_{xy} + f_x f_z f_{xz} + f_y f_z f_{yz}) \\ &+ f_x^2 (f_{yy} + f_{zz}) + f_y^2 (f_{xx} + f_{zz}) + f_z^2 (f_{xx} + f_{yy}). \end{aligned} \tag{1.12}$$

1.2.8 Summary of implicit surface rendering techniques

Upon reviewing available literature sources there are four key issues identified in the rendering of implicit surfaces, common to either a forward or backward mapped system. Each method can be judged and compared based on these criteria;

Image Quality The visual accuracy of the implicit surface image. Anti-aliasing capabilities are required for smooth images, and are dependent upon the rendering algorithm. Rasterised systems can be anti-aliased without extra sampling of the implicit surface, while ray-casted systems need to be sampled multiple times per ray.

Efficiency The practicality of a rendering algorithm is limited upon its efficiency. This is measured from time to produce a rendered frame. A technique that is too slow is infeasible to apply to complex problems;

Scalability A technique has limited usefulness if it is only applicable to a specific task. It needs to be able to scale well to other problems. For instance, brute force ray-casting on the GPU is fast, yet yields inaccurate results when compared to more logic driven methods such as intervals;

Resources Resources, while a lower priority concern, are important when dealing with large datasets, as storage requirements can be a considerable problem;

Implicit surfaces are prone to error when visualised, from complex geometry present on non-manifold and self-intersecting surfaces. Robust polygonisation methods are difficult, requiring handling of special cases to minimise problems in the output mesh, e.g. fixing cracks from adaptive polygonisation, dealing with octree nodes with self intersections, rays etc.

Ray-tracing efforts have previously shown limited success, often missing such features. Recent work in GPU ray-tracing offers much more reliable results at reasonable speeds, as intervals can now be used on a GPU through programmable shaders.

Splatting is ideal for visualisation of a surface with minimal sample points, as each sample can be rendered as an oriented splat disc, but again non-manifold features are problematic, such as rendering singular lines. A pure point-based approach can techni-

cally represent infinitely thin features such as rays, as points by definition have no dimensions, they are infinitely small themselves.

An under-examined aspect of implicit surface research is the scalability of an algorithm when extended to rendering of anti-aliasing, contouring, curvature, shadows and transparency. Some of these will be fast with some methods, while slow with others. For instance, anti-aliasing geometry primitives (polygons or points) can be done efficiently through graphics hardware, while ray-tracing methods need to bounce multiple rays per pixel into the surface, cutting the frame rates down. This further complicates judging which are currently the best options for implicit surface visualisation.

1.3 Contributions

This thesis makes the following contributions to implicit surface rendering.

We present algorithms for anti-aliasing silhouette edges of manifold and non-manifold implicit surfaces. The first algorithm identifies edge regions and applies anti-aliasing. The second uses the GPU to render multiple buffers which are combined into an anti-aliased image. We also present new algorithms for rendering contours on surfaces rendered using points. Contours we examine are planar contours and contours of constant Gaussian curvature. We also discuss the anti-aliasing of contours rendered in real-time on implicit surfaces. The GPU anti-aliasing method also allows us, for instance, to animate contours on a surface by changing contour parameters. We give examples showing the results of our anti-aliasing techniques for various types of contours rendered on surfaces as an aid to their visualisation.

We present a point-based technique that improves the rendering of non-manifold implicit surfaces by using point and gradient information to prune plotting nodes resulting from using octree spatial subdivision based on the natural interval extension of the surface's function. The use of intervals guarantees that no parts of the surfaces are missed in the view volume and the combination of point and gradient sampling preserves this feature while greatly enhancing the quality of point-based rendering of implicit surfaces. We also successfully render non-manifold features of implicit surfaces such as rays and thin sections. We illustrated the technique with a number of example surfaces.

We also present a point-based technique that does not require gradient information. This algorithm uses point sampling and the natural interval extension as a subdivision criteria and for non-manifold sections uses a pruning algorithm based on a signed distance function to prune plotting nodes around the non-manifold features to more accurately render these features.

Using this approach we can produce high quality rendered thin sections of surfaces, cusps, ridges and singular lines of surfaces. These features can be rendered whether they are aligned with the octree node edges or not. So, for example, we can render thin rays at arbitrary angles with this algorithm. This is not possible using the algorithm given in our point and gradient approach.

We also discuss the advantages and disadvantages of the algorithm. Our results compare quite favourably with those obtained by GPU based ray-casting the surfaces and we give timings comparing the approaches. A typical surface will take from seconds to hours to produce, depending on the resolution, and the number of points used representing the surface. The algorithm was implemented on the GPU and we discuss our implementation of this.

We use this algorithm to explore the curvature of implicit surfaces. In particular we examine the curvature surfaces of non-manifold implicit surfaces such as Boy's surface, the cyclide surface and the Klein bottle surface.

Finally, we present algorithms for rendering high quality animations of manifold and

non-manifold implicit surfaces. We show how to produce animations of surfaces whose 3D shape is hard to conceptualise from static images. We also do animations of parameterised implicit surfaces where we change the parameter as a function of time.

The first algorithm uses a mixture of polygons and point data to do shaded rendering of the surface. The second uses point splatting to render the surface. We compare the relative advantages and disadvantages of these algorithms to interval ray-casting.

We discuss the implementation of both algorithms on the GPU using *CUDA*. We compare the GPU ray-casting approach to our GPU polygon/points and point splatting algorithms in terms of; robustness, speed, and quality.

Chapter 2

Point-based Anti-aliasing methods

Rendering implicit surfaces without aliasing problems is a reoccurring problem, as solutions are generally specific to the rendering method in use. 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 2.8 (a). Aliasing along contour edges can be equally prominent. We are concerned with the quality of the images produced by our point-based method and the speed of the anti-aliasing technique, as we wish to apply it to rendering manifold and non-manifold implicit surfaces.

From previous work [61] it is not clear how anti-aliasing should proceed on point-based surfaces. We report here on a number of approaches to anti-aliasing along silhouette edges and give recommendations as to the best anti-aliasing 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, we show how anti-aliasing can be

done along singular lines.

2.1 Approaches

Various anti-aliasing techniques were investigated for use in a point-based rendering system. Each method is described in the following subsections.

2.1.1 Object space anti-aliasing

Aliasing of an implicit function occurs as we are discretely point sampling a continuous function. Take, for example, the curved $2D$ function in Figure 2.8 (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 2.8 (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 [62]. The edge pixel's 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 of 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 eight to obtain a fraction between zero and one which will be used to represent the opacity of the surface pixel.

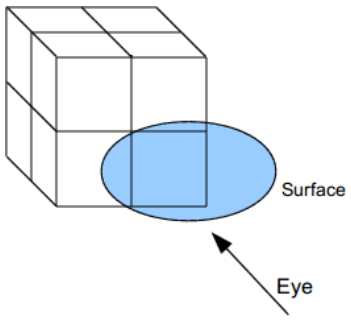


Figure 2.1: Illustration of surface coverage from querying octree nodes from the eye's perspective.

A problem arises in that only octree nodes along the silhouette edge of the surface (or a contour) need to be anti-aliased and these need to be found as shown in Figure 2.1. Nodes should only be blended using surface coverage if the surface normal is perpendicular with the camera. This is done by first creating an image where the value of the pixel is zero or one 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 2.2.

Figure 2.2: Object space anti-aliasing applied to curved surface.

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

2.1.2 Edge blur method

Edge blurring is a post-process technique [63] that reduces aliasing along the edge geometry. The method is applied after all the points in the complete scene have been found. It is implemented as a filter in image space. This is common in deferred shading applications that, under some platforms, cannot perform multi-sampling with floating-point multiple render-target (MRT) hardware buffers.

Typically a renderer [63] 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, illustrated in Figure 2.3.

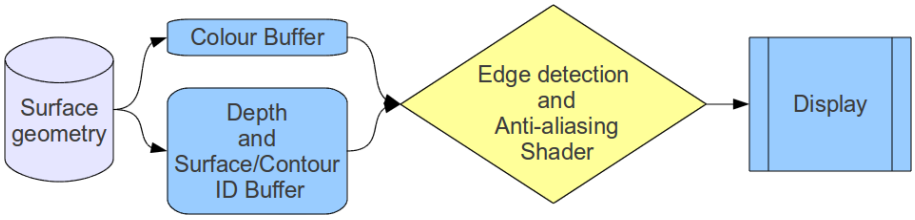


Figure 2.3: Flowchart illustrating steps from surface geometry to an anti-aliased render.

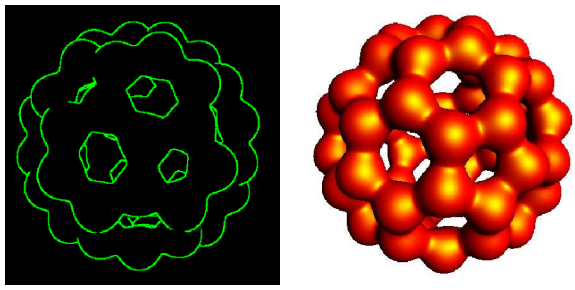
We begin by looking at a figure of the Buckyball surface (C^{60}) which is the iso-potential surface defined by 60 point charges in 3D given by

$$f(x,y,z) = \sum_{j=1}^{60} \frac{q_j}{(x-x_j)^2+(y-y_j)^2+(z-z_j)^2} - c = 0. \tag{2.1}$$

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 2.4 (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 flagged as an edge pixel (rendered green in Figure 2.4 (a)).

Pixels that are part of the contour are detected in a fragment shader using equations as given in Balsys *et al.* [54]. 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 anti-aliased.

In the next stage we take the image (Figure 2.4 (b)) and overlay it with the edge map



(a) (b)

Figure 2.4: Edge detection of bucky ball surface (2.1). (a) Silhouette edges in green. (b) Aliased surface image.

(Figure 2.4 (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 the coverage is examined inside a 3×3 kernel, as shown in Figure 2.5, and 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 2.5: The 3×3 blur kernel 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 2.5. We have found this method to be successful in removing sharp aliasing features in an image. However the anti-aliasing 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 2.6. More expensive methods of estimating edge coverage were not pursued, as a motivation of the research was real-time animations.

Figure 2.6: Edge blur is satisfactory 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 anti-aliasing 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.

2.1.3 Super-sample anti-aliasing (SSAA)

The classic brute force method of anti-aliasing is known as super-sampling and is discussed in Foley *et al.* [13]. 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 use. 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 and limits real-time applications.

2.1.4 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, for a

result that looks closer to super-sampling. 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 anti-alias 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 anti-aliased. 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.

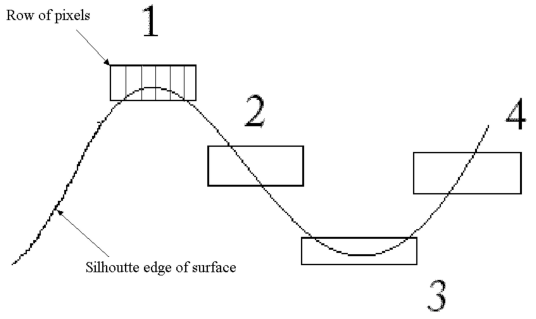


Figure 2.7: The four position and style types for pixel anti-aliasing cases in the adaptive pixel-tracing method, labelled as Case 1, 2, 3 and 4.

The different styles of edges that are handled by the shader are shown in Figure 2.7. The technique begins by detecting monotonically increasing or decreasing edges (Case 2 and 4) as illustrated in Figure 2.8 (a).

(a) (b)

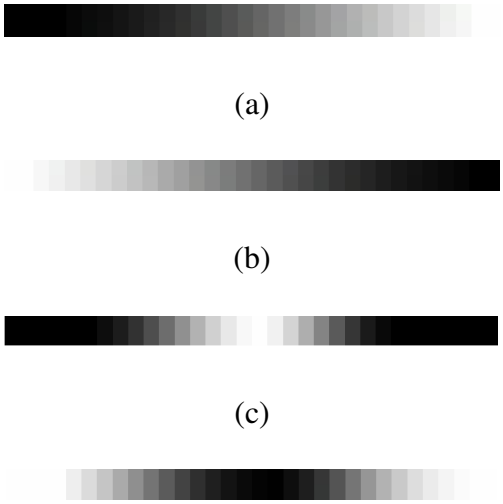
Figure 2.8: (a) Common Aliased staircase effect, and (b) The result of our pixel-traced anti-aliasing pass.

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 one 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).

There are four possible positions of horizontal / vertical runs of pixels, see Figure 2.9. 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 2.7. Analogous cases are used for column runs of pixels.

For a pixel in the row of Figure 2.9 (a) each pixel's opacity value, ω, is found by first finding the distance x from the start of the pixel line. In Figure 2.10 $x = 5$ if the edge



(a)

(b)

(c)

(d)

Figure 2.9: 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.

starts on the left side. The number of pixels in the line L is 9. Dividing the distance x by the number of pixels in the row L gives us the opacity, $\omega = \frac{5}{9} = 55\%$.

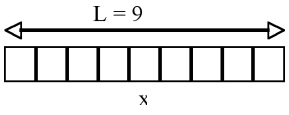


Figure 2.10: Row of pixels of length L.

$$\omega = \frac{x}{L} \tag{2.2}$$

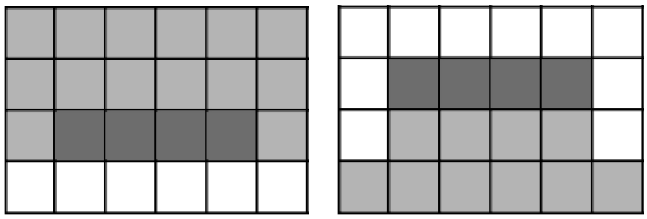
Once the opacity value, ω, has been found we use it to calculate the final pixel colour

using;

$$\begin{aligned} Colour.rgb &= (sourceColour.rgb * \omega) \\ &+ (edgeColour.rgb * (1-\omega) \end{aligned} \tag{2.3}$$

If the opacity change is reversed, as in Figure 2.9 (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.



(a) (b)

Figure 2.11: (a) Inside edge shown in dark grey. (b) Outside edge shown in dark grey.

Figure 2.11 (a) illustrates an inside edge occurrence (case 3 in Figure 2.7) 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 2.9 (c). Figure 2.11 (b) illustrates an outside edge occurrence (case 1 in Figure 2.7). The outside edge in dark grey needs to be faded as

shown in Figure 2.9 (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

$$\omega = abs((\omega - 0.5) * 2) \tag{2.4}$$

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 anti-aliasing 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.

We found this method works well on geometry that features clear silhouette lines on the edge map. When the edge map is too noisy from sharp features, the algorithm has trouble detecting the previous described edge cases, so we invested further solutions.

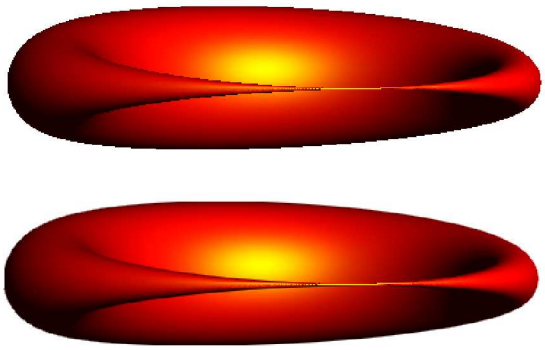


Figure 2.12: Demonstrating anti-aliasing on the inside and outside surface edges. (Top) aliased image and (Bottom) anti-aliased image of the Cyclide (2.5) surface.

The Cyclide surface in Figure 2.12 is given by f[7]

$$\begin{aligned} f(x,y,z) &= (x^2+y^2+z^2)^2 \\ &- 2(x^2+r^2)(f^2+a^2)-2(y^2-z^2)(a^2-f^2) \\ &+ 8afrx+(a^2-f^2)^2 \end{aligned} \tag{2.5}$$

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

2.1.5 Jitter-based anti-aliasing

Anti-aliasing by rendering the scene multiple times to the accumulation buffer is described in [64]. Each pass increments the sample level of every pixel, e.g four passes produces $4x$ anti-aliasing. Before each pass the camera origin is jittered with respect to the pixel grid. As discussed in Cook [65] uniform spaced sampling will cause aliasing artifacts, the jittering values should make an irregular pattern to make them "noisy". Ir-

regular jitter values for different sampling levels are provided in [64].

We cannot use an accumulation buffer as discussed in [64] as the accumulation buffer requires the z-buffer be cleared before each pass. As we render while we are subdividing the surface in real-time, we would clear the depth associated with these rendered points. To get around this problem, we use multiple render buffers with their own attached z-buffer.

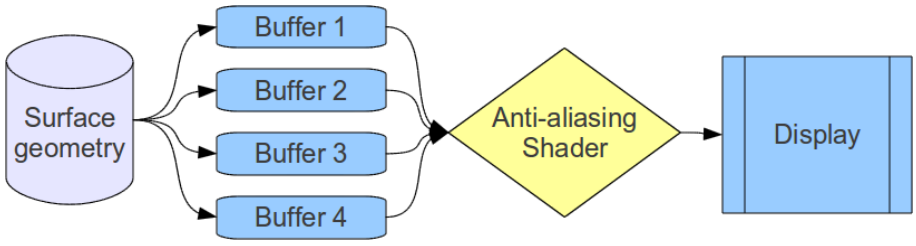


Figure 2.13: Flowchart illustrating steps from surface geometry to a $4x$ anti-aliased render.

For $16x$ anti-aliased images we create sixteen render buffers each with its own z-buffer and colour buffer. The image is rendered into each of the sixteen z-buffer and colour buffers. The colour values in the separate buffers are averaged to give the final colour to be assigned to the pixel. This results in anti-aliasing of the pixels colour. It works equally well for anti-aliasing 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 geom-

etry. This can be performed while the samples are being generated, thereby making the sampling process interactive and able to be run in real time.

2.2 Contour anti-aliasing

Rendering contours onto surfaces can present aliasing artifacts. 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 anti-aliasing 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.

We first 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. In our method we move this test into image space, only testing 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.

To smooth (anti-alias) 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 center of the slab and the outside of the slab. This is illustrated in Figure 2.14. The pixel colour is the average of $1 - \frac{x}{x+D}$ times the contour colour plus

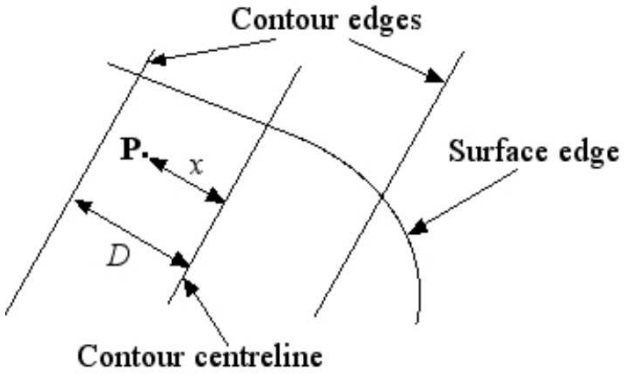


Figure 2.14: Illustration of the distance, x, of the surface point P from the centre of the contour slab.

$\frac{x}{x+D}$ 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 don't test every sampled point in the octree, only the points visible from the camera. This results in large speed-ups in contour rendering [5] 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).

2.2.1 Examples from real-time visualisation of contours

First, in Figure 2.15 (a) we show anti-aliasing of planar contours such as planes orthogonal to the y and z axes, and in Figure 2.15 (b), we show lines of Gaussian curvature rendered on an ellipse. To smooth (anti-alias) 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, $\omega = 1 - \frac{x}{x+D}$, 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 [5].

For a planar slab with width w, the slab thickness d is;

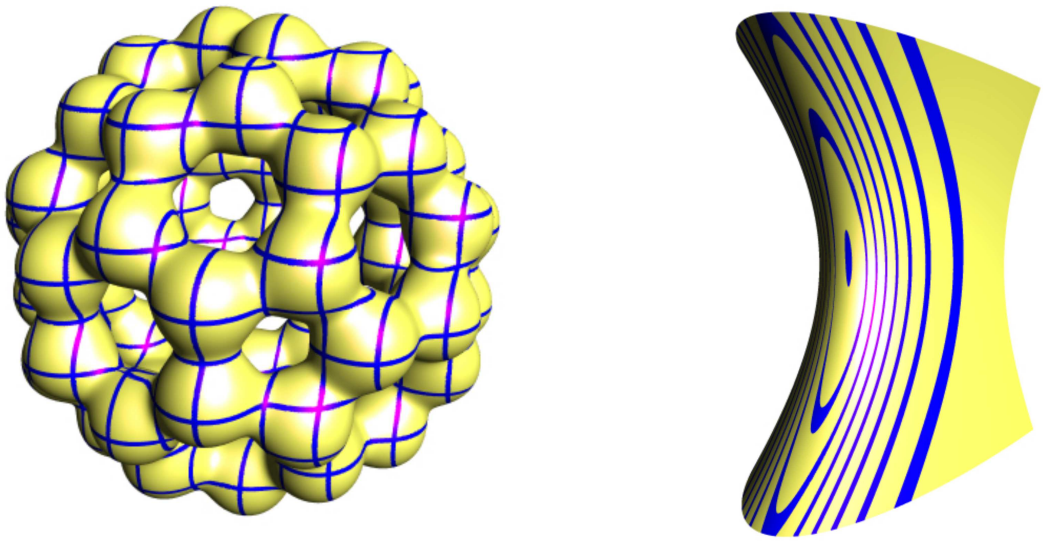
$$d = w\sqrt{1-(n_f \cdot n_g)^2} \tag{2.6}$$

where n_f is the unit normal to the surface, and $n_g = \frac{\nabla g}{|\nabla g|}$ is the unit normal to the slab.

For a curved slab the distance s to the center of the contour is approximately;

$$s = \frac{|\, g(p) - c \,|}{|\, \nabla g(p) sin\theta \,|} \tag{2.7}$$

As the renderer defers the lighting calculations to the contouring pass, surface lighting



(a) (b)

Figure 2.15: (a) Planes parallel to the y and z coordinate axes rendered on a bucky-ball surface f[14] , and (b) Lines of constant Gaussian curvature rendered on an ellipse.

and changing contours can be rendered in real time using this approach.

2.3 Filling missing pixels

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 [5], indicating we needed to render the surface and subdivide to a higher plot depth

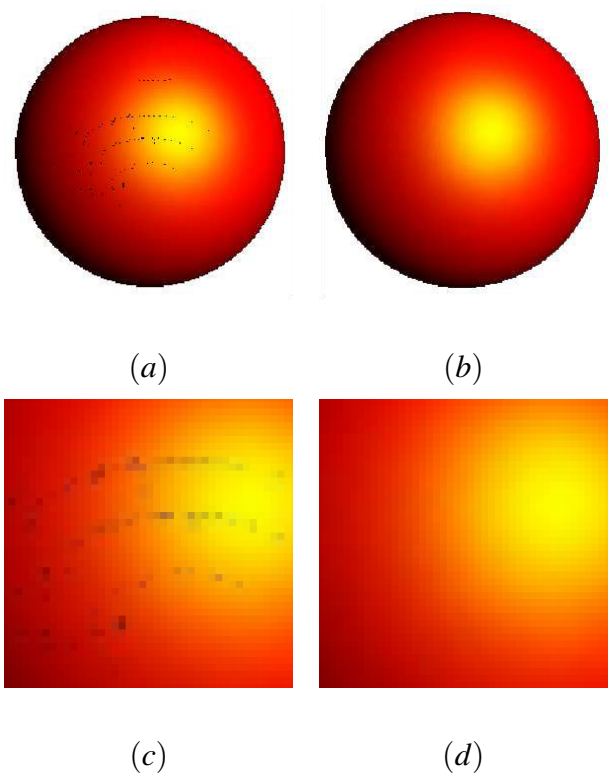


Figure 2.16: Demonstrating missing pixel filling. (a) Rendered image with missing pixels. (b) Final surface with missing pixels blended. (c) and (d) are close-ups of (a) and (b) respectively.

(see Figure 2.16 (a)). We can fill in missing pixels without significantly increasing our rendering times in the anti-aliasing pass as follows.

After the lighting stage in the viewing pipeline we loop through each pixel and compared neighbouring pixels for depth changes, as illustrated in Figure 2.17. 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

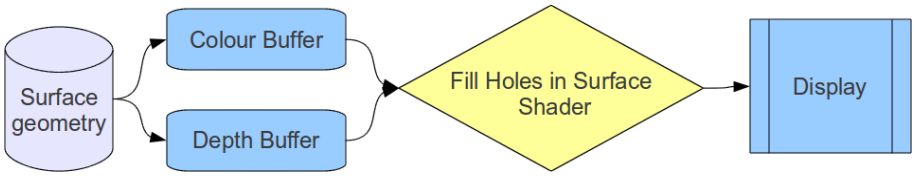


Figure 2.17: Flowchart illustrating steps from surface geometry to filling holes in the final image.

is calculated as a weighted sum of the neighbouring pixels' colours. Figure 2.16 (b) shows the final sphere rendering with the missing pixel's colour blended as specified.

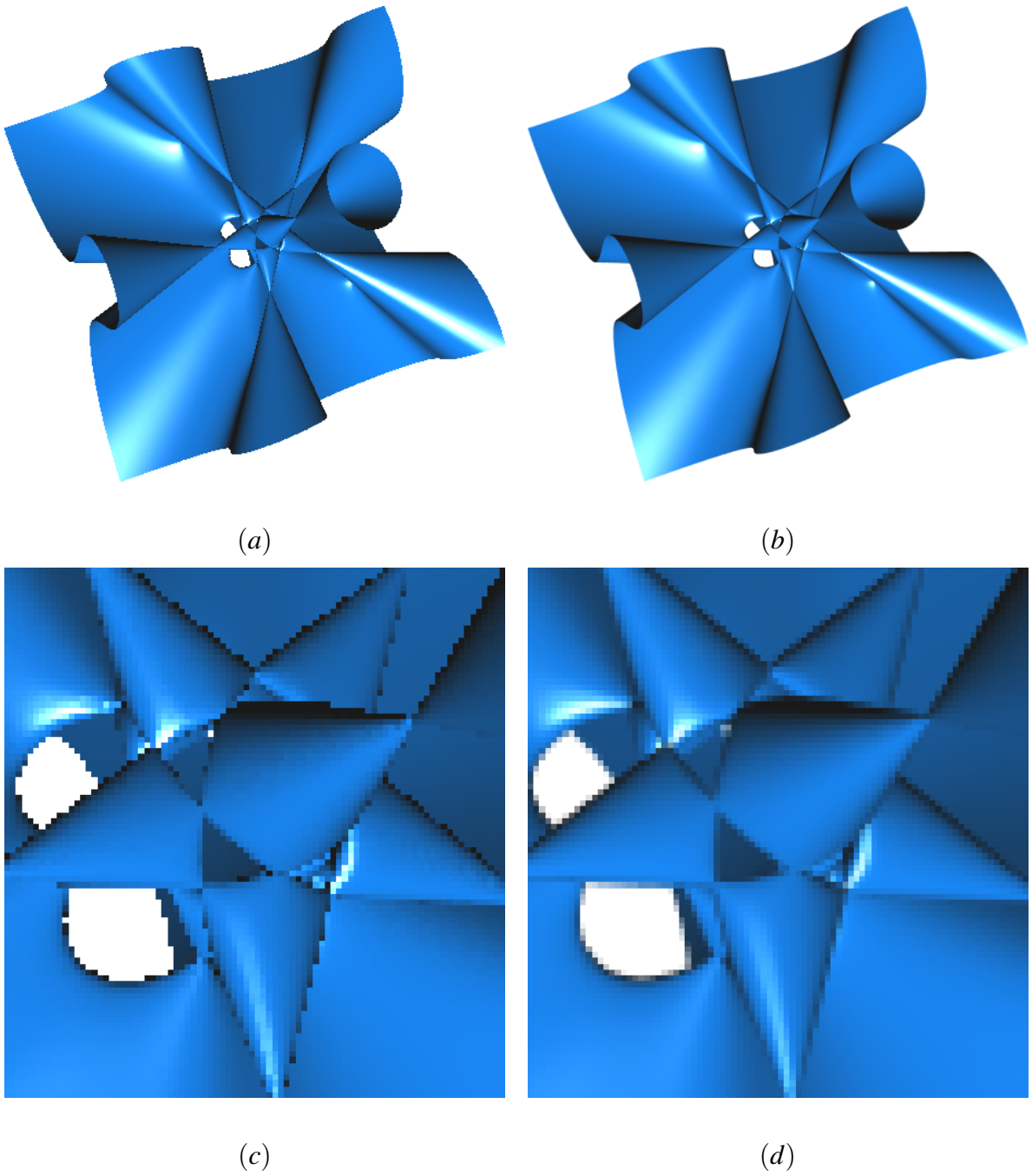
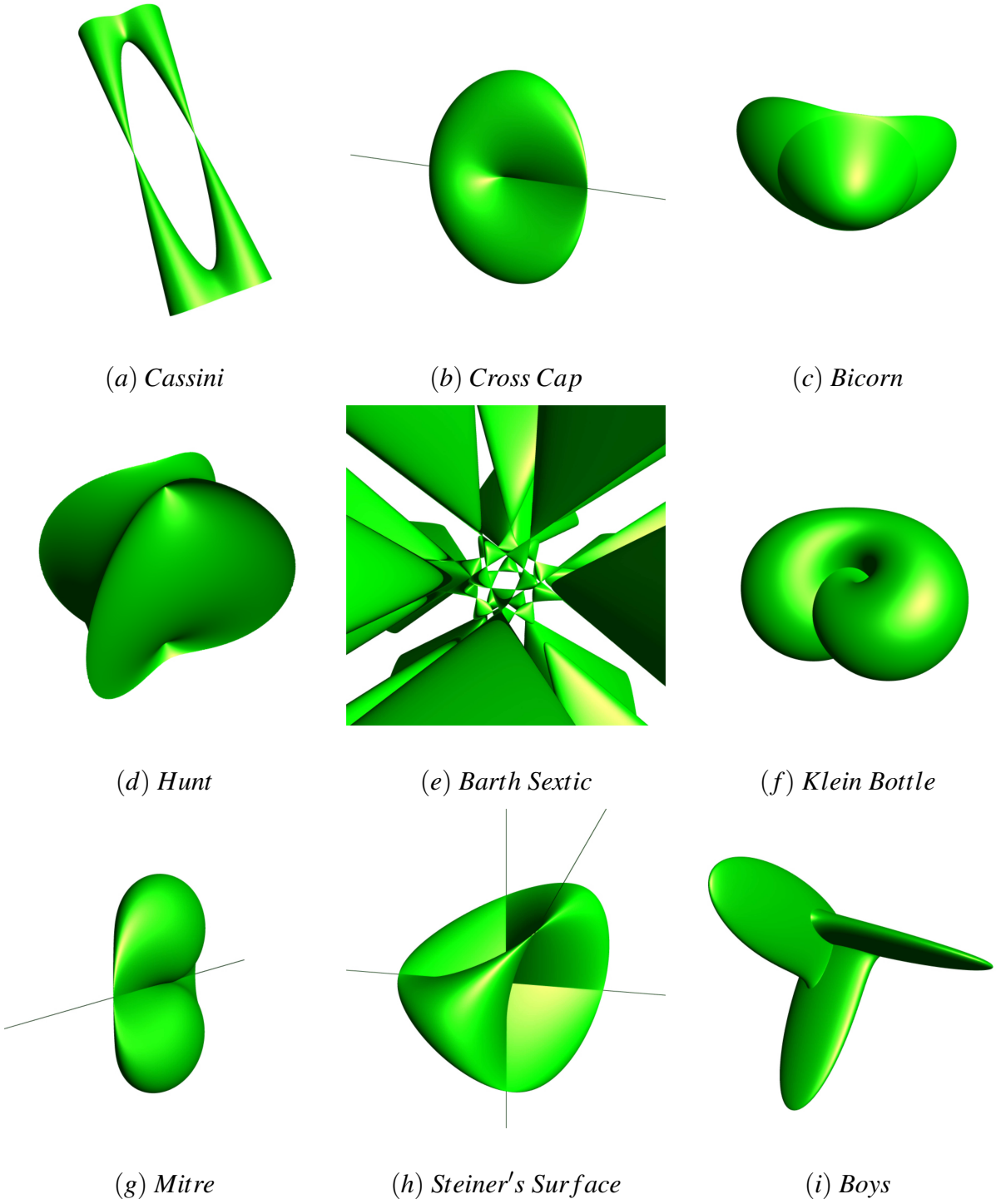


Figure 2.18: Jitter-based anti-aliasing comparison on the star surface f[30] . (a) No anti-aliasing. (b) Anti-aliasing. (c) and (d) are close-ups of (a) and (b) respectively.



(a) *Cassini* (b) *Cross Cap* (c) *Bicorn*

(d) *Hunt* (e) *Barth Sextic* (f) *Klein Bottle*

(g) *Mitre* (h) *Steiner's Surface* (i) *Boys*

Figure 2.19: Anti-aliased non-manifold surfaces. (a) Cassini, (b) Cross cap, (c) Bicorn, (d) Hunt, (e) Barth Sextic, (f) Klein bottle, (g) Mitre, (h) Steiner's surface, and (i) Boy's surface.

Chapter 3

Point Rendering of Implicit Surfaces Using an Octree and Interval Arithmetic

In [1] we presented a point-based technique that improves the rendering of non-manifold implicit surfaces, using point and gradient information, to prune plotting nodes arising in octree spatial division, with a subdivision criteria based on the natural interval extension of the surface's function. We successfully rendered non-manifold features of surfaces such as rays and thin sections. We were not concerned with the time it takes to generate the surface but the correctness (accuracy) and quality of the resultant images.

From previous work on the polygonal rendering of implicit surfaces, Balsys and Suffern [4] and Suffern and Balsys [11], we know that octree based volume decomposition with a subdivision criteria based on an interval exclusion test, converges to nodes clustered about the implicit surfaces. We use this approach by rendering a point within the plotting nodes of an octree recursively subdivided using an interval exclusion test as discussed in [21].

The main reason for using intervals in this context comes from the property that the natural interval extension of a function provides bounds on the variation of the function. The bounds may not be very tight, but the values of the function f are guaranteed not to be outside of this interval. A consequence of the bounds not being very tight is that pixels may be rendered in nodes that do not contain surface elements. This is an undesirable effect as it leads to display anomalies as shown in Figure 3.8 and Figure 3.9. Further testing needs to be performed in each node to filter out those which do not contain the surface.

Our interval arithmetic package includes infinite intervals, to extend the range of functions that we can render. Infinite intervals result from the division of intervals where $0 \in [c,d]$ for the interval $[a,b]/[c,d]$. Hansen [66] gives formulae for infinite intervals and the computer implementation of infinite intervals is given in Ratschek and Rokne [67].

In our subdivision process the view volume (defined as an axis aligned cube of 3D space) is subdivided into 8 sub-nodes, and interval analysis is made at each node to see if it may contain part of the surface. If this is true, the node is recursively subdivided. The recursion proceeds to a fixed maximum depth in the octree, which we call the $plotDepth$. At the plot depth a point is generated within the node and we call such nodes *plotting nodes*. The coordinate of this point is used in the shading algorithm to render the surface. Although the interval test does not guarantee that the surface exists in any plotting node, it does guarantee that any node of the octree that is discarded *does not* contain the surface. We investigated methods to discard nodes at the plot depth that do not contain surface

elements even though the interval test indicates they may contain surface elements.

A single parameter controls the time it takes to generate the surface: the plot depth. The plot depth is set by the user. In [21] they estimated the plot depth required for complete coverage of the surface and gave a formula to calculate the required plot depth so that no surface "holes" appear in the final image.

Balsys and Suffern [21] reported on a number of surfaces which have problems when rendered when using the methods so far described. For example, implicit surfaces with very thin sections or singularities can create problems for point-based methods. Consider the spiky surface f[8]

$$f(x,y,z) = x^{2n} + y^{2n} + z^{2n} - x^n y^n - x^n z^n - y^n z^n, \tag{3.1}$$

as shown in Figure 3.8 for $n = 4$. The thickness of the spikes approaches zero as their distance from the center becomes infinite. As we stop subdividing our octree at a given plot depth and we use interval subdivision an error in rendering the surface occurs. The interval method guarantees that we will not reject a plotting node that may contain a surface section, but we may have a number of plotting nodes that do not actually contain a surface section in the plotting node. This results in pixels being erroneously rendered with the surface colour even though the surface does not lie in the plotting node represented by that pixel. As a result thin sections of diameter less than the plot depth will be rendered as thin tubes. The situation is worse for Steiner's roman surface f[2]

$$f(x,y,z) = x^2y^2 + y^2z^2 + x^2z^2 + xyz = 0, \tag{3.2}$$

which has three (infinitely thin) double lines along the coordinate axes, see Figure 3.9 (a). Our technique renders these with a minimum radius, the fact that they are singularities has confused the technique so that it has rendered them with an increasing radius as they approach the surface. This gives the misleading impression that the Steiner's roman surface exists outside a sphere of radius one centered on the origin, which it does not.

We can improve the appearance by increasing the plot depth but this does not solve the problem. Sederberg and Zundal [68] successfully rendered the singular lines with a scan-line algorithm but it's much more complicated than our point-based algorithm. Balsys and Suffern [25] successfully rendered the singular lines with degenerate polygons, but again, their polygonisation algorithm is much more complicated, than the point-based solution described in the next section.

3.1 Plotting Node Trimming and Feature Extraction

As previously stated interval bounds may not be very tight, f is guaranteed to be zero outside the interval. An undesirable consequence of this is that pixels may be rendered in nodes that do not contain surface elements. We investigated methods to discard nodes at the plot depth that do not contain surface elements even though the interval test indicates they may contain surface elements. This process is illustrated in Figure 3.1

We first evaluate the function value at each vertex of the plotting node. We then use a sign test [24] to determine whether an intersection occurs on an edge of a plotting node.

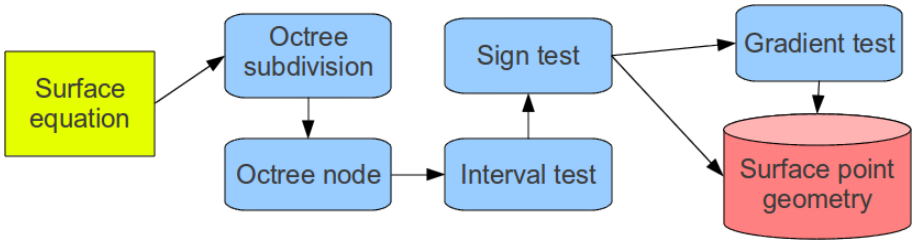


Figure 3.1: The steps to create point geometry from a surface equation. If the sign test passes the point is stored. If the sign test fails the node is passed to the gradient test. If the gradient test passes the point is stored, otherwise it is discarded.

This occurs when there is a sign difference between the vertices of the edge of a node. If there is an intersection on the edge then we have confirmed the presence of the function in the node and a pixel with coordinates equal to the centre of the node is drawn with the Phong reflection model.

However the sign test will fail to detect a surface in the node in a number of situations, see for example Balsys and Suffern [25]. For example the sign test will fail if a surface element is contained within the plotting node, or if the surface function intersect the plotting node edge more than once. In previous work [25] to polygonise an implicit surface with non-manifold features they introduced a gradient test to guard against these possibilities.

The sign and gradient test we use are

$$\begin{aligned} \text{Sign Test:} \quad & f_a * f_b < 0 \\ \text{Gradient Test:} \quad & \nabla g_a \cdot \nabla g_b \leq 0.1 \end{aligned} \tag{3.3}$$

where f_a and f_b are the values of f at the endpoint a and b of an edge and ∇g_a and ∇g_b are the gradients of the function f at the endpoint a and b of an edge. The value of 0.1 in the gradient test corresponds to an angle of divergence of about 84°. This minimum amount of divergence between the gradients has been shown to give good results for the surfaces we have tested.

If the sign test indicates there is no root of f on the edge, but the gradient test indicates there may be, the possibility exists that more than one root of f exists on that edge. As we are already at maximum depth, we choose to render that node as if it does contain a surface element (gradient test) even though the sign test indicates it does not contain a surface element. Figure 3.8 (b) shows the result for the Spiky surface when the sign and gradient tests are applied.

3.1.1 Sign Test Results

The sign test produces highly accurate renderings of the surface. Figure 3.9 (a) shows Steiner's roman surface rendered explicitly with the interval test, and Figure 3.9 (b) shows the same surface rendered including the sign test in the plotting node. The interval test suffers from overestimating the possibility of a surface being inside a node, creating tubes curving into where the rays intersect the surface. The sign test can correctly visualise the rays without such anomalies.

In some cases the sign test will fail, either when a surface intersects an edge more

than once, or if a surface does not intersect an edge but is still contained within the node. The gradient test can successfully detect surfaces that are smaller than the final plotting depth node. The sign and gradient test when applied to Steiner's roman surface produce the results given in Figure 3.9 (b). By itself the sign test will improve the rendering of some surfaces but will cause problems with others.

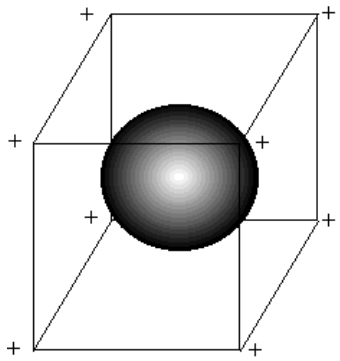


Figure 3.2: A small sphere located inside a plotting node showing values of sign at node vertices. This demonstrates when a surface feature is too small to intersect the node, which the sign test will not detect.

Figure 3.2 demonstrates how a surface function inside a node is not detected by the sign test method as no intersections of the edges take place. Figure 3.3 shows how the gradient, ∇f, differs at each corner for the same surface. The angle between the gradient vectors can be used to indicate the surface is present.

Combining the sign and gradient test provides a significant advantage over the interval

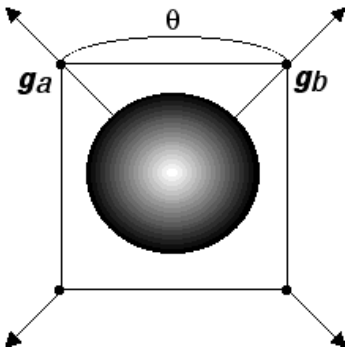


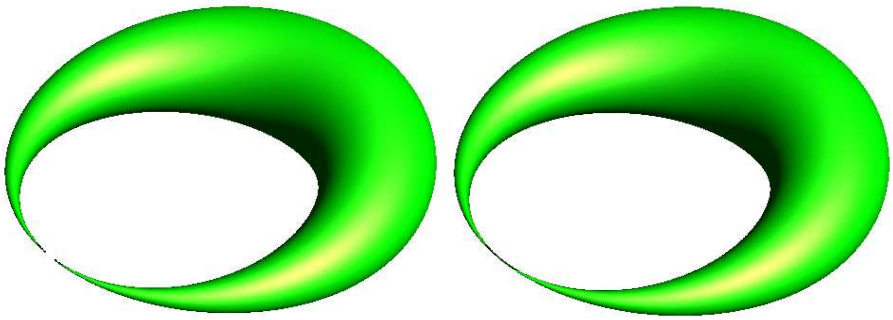
Figure 3.3: A small sphere located inside a plotting node showing values of gradient at node vertices. The gradient evaluation of the function has revealed a surface may be present by calculating a dot product on the node vertices.

method alone. Consider the cyclide surface given by f[7]

$$\begin{aligned} f(x,y,z) &= (x^2+y^2+z^2)^2 \\ &- 2(x^2+r^2)(f^2+a^2)-2(y^2-z^2)(a^2-f^2) \\ &+ 8afrx+(a^2-f^2)^2 \end{aligned} \tag{3.4}$$

with a = 10, f = 2 and r = 2, as shown in Figure 3.4. When interval subdivision and the sign test is used the thin section joining two lobes of this surface is missed, as shown in Figure 3.4 (a). When the gradient test is used with interval subdivision and the sign and gradient test the thin section joining the two lobes is rendered, as shown in Figure 3.4 (b).

Another example that illustrates the necessity of the gradient test is given in Figure 3.5 where many spheres are rendered. Some of the spheres are smaller than a node and would not be detected using the sign test alone. When the gradient test is also used plotting nodes containing spheres are rendered as a single pixel.



(a) (b)

Figure 3.4: The cyclide surface with plot depth = 9. (a) Interval subdivision and the sign test, and (b) Interval subdivision and sign and gradient tests.

3.1.2 Rendering nodes containing rays

The sign and gradient test does not detect rays in some cases. By examining the plotting nodes that surround the singular lines of Steiner's roman surface in Figure 3.9, we found that the lines lie along node edges. We used the sign test for this, and the assumption that if both ends of an edge evaluate to zero, the edge is part of the surface.

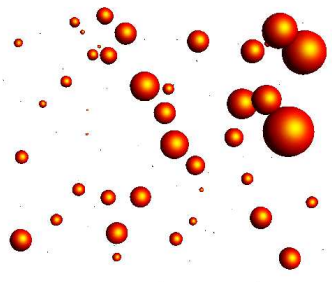


Figure 3.5: 100 randomly sized spheres. Some of the spheres are smaller than the node width at the maximum plot depth.

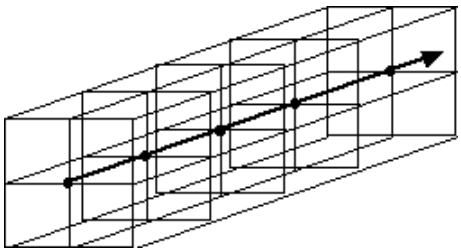


Figure 3.6: Shows the intersection edges detected for the infinite rays present in Steiner's roman surface.

Figure 3.6 shows the vertices that evaluate to zero for a sequence of plotting nodes. In all these plotting nodes we can determine that a single edge of the plotting node is part of the surface and we render these edges by shading a pixel on the mid point of the plotting node edge. This is done for Steiner's roman surface in Figure 3.9 (a) and for Steiner's relative surface f[16] , given as

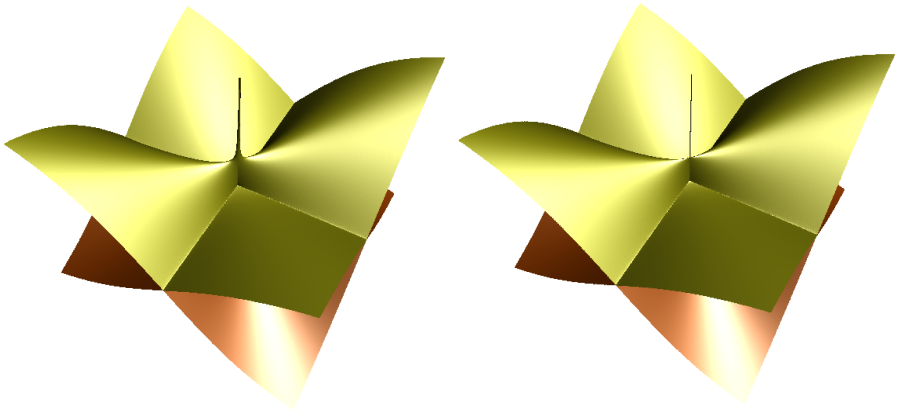
$$x^2y^2 + y^2z^2 - x^2z^2 - xyz = 0, \tag{3.5}$$

and shown in Figure 3.7. In Figure 3.7 we render one side of the surface in orange and the other side in Blue to aid the visualisation of the shape of the surface.

3.1.3 Multi-threading

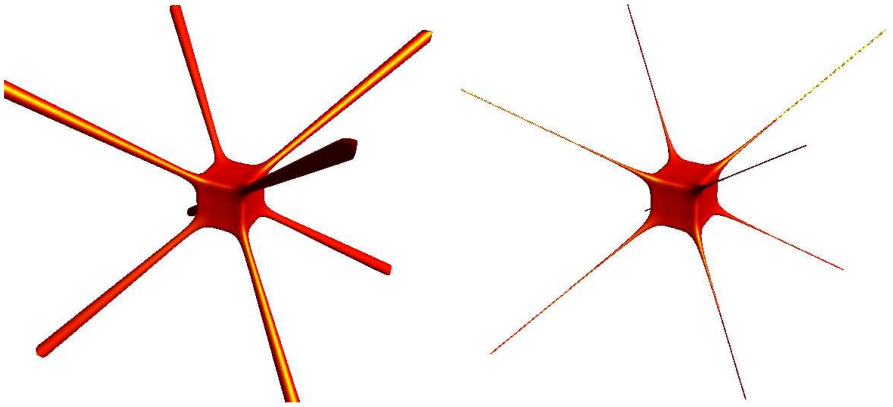
We use octree subdivision and thus our algorithm readily lends itself to acceleration by multi-threading the octree code. This results in improved performance on modern multi-core processors. The root node is split into 8 child nodes, each of which initiate their own thread and subdivide concurrently. The *OpenMP* standard was chosen to implement this

functionality due to its cross-platform ability. Rendering Steiner's roman surface on an Intel Core 2 Duo processor without threading and with threading and taking the ratio of the time required to render the surface in both cases, indicated a speed increase due to threading of 1.9 for a two processor system. This indicates that the octree based subdivision process is an algorithmic method well suited for use on the multi-core processors being actively developed by the chip manufacturers and finding their way into current PC's.



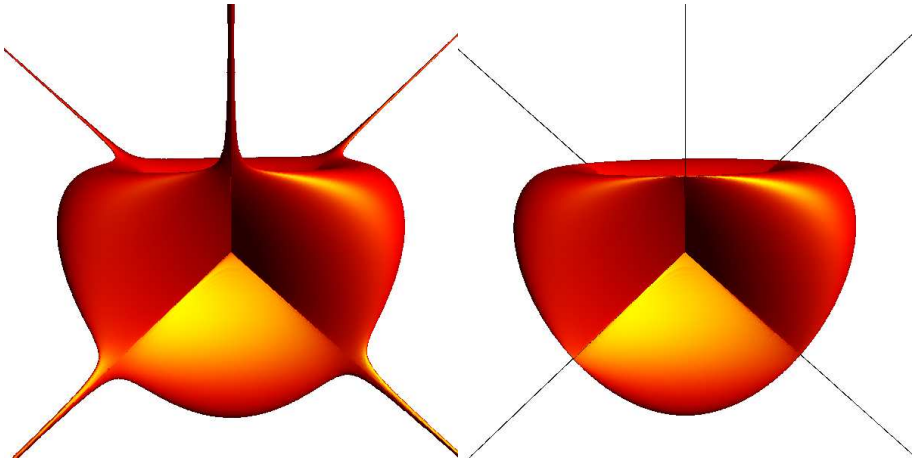
(a) (b)

Figure 3.7: Steiner's relative (3.5) with plot depth = 9. (a) Interval subdivision, and (b) Interval subdivision and sign and gradient tests. Shows a thin ray emanating along the y axis.



(a) (b)

Figure 3.8: Spiky surface with plot depth = 9. (a) Original surface showing spikes as tubes, and (b) New method showing how the tubes converge to lines.



(a) (b)

Figure 3.9: Steiner's roman surface with plot depth = 9. (a) Interval test only, and (b) Interval and sign test.

Chapter 4

Visualisation of the Curvature of Non-Manifold Implicit Surfaces

Curvature is an intrinsic property of surfaces and analysis of curvature has many applications in science and engineering. Two terms are fundamental to the understanding of curvature; these are the minimum and maximum curvatures of the surface, denoted as κ_1 and κ_2. From these the mean curvature, $H = (\frac{\kappa_1+\kappa_2}{2})$ and the Gaussian curvature, $K = \kappa_1\kappa_2$ are derived. The Gaussian and mean curvatures are intrinsic properties of continuous implicit surfaces [8]. For an implicit surface the magnitude of both H and K can also be expressed as an implicit surface.

Our motivation for the work presented in this chapter was to visualise the curvature surfaces [5] of some well known implicit surfaces such as Boy's surface f[1] , Steiner's Roman surface f[2] and the Klein bottle f[3] . Although curvature surfaces of simple implicit surfaces have been visualised in the past, those of more complex implicit surfaces, non-manifold surfaces, and singular surfaces, have not been rendered. Figure 4.1

illustrates the curvature surfaces of surfaces rendered using the algorithms given in this work.

Another issue we would have to tackle was to be able to capture these images and manipulate them in as near to real-time as possible. For instance to visualise the 3D shape of Boy's surface from a single static image is exceedingly difficult. It is not until one is able to rotate this surface in real-time that one is able to appreciate of the 3D nature of this surface. Even better would be the ability to view stereo pairs of the surface in real-time in a VR system so that the depth information is apparent. The mathematical complexity of the implicit function for this surface, however, means that doing this has not been practical until recently.

We were encouraged by recent work in ray tracing surfaces using the GPU, see for example Knoll *et al.* [53] and Singh *et al.* [52], to believe that our goal was feasible. However we did not believe that ray tracing would be an appropriate vehicle for our work as we needed to capture surface geometry so that we could then render our surface from any point of view. Ray tracing is not useful for defining surface geometry, since as the camera rays all diverge, surface points found using the ray-surface intersection calculations result in non-uniform sampling of the surface.

Finally as our surfaces are arbitrarily complex and are generally higher order the ray-surface intersection calculation will be either a bottleneck, or infeasible to calculate. Using interval ray calculations may help in this regard, but achieving the required accuracy

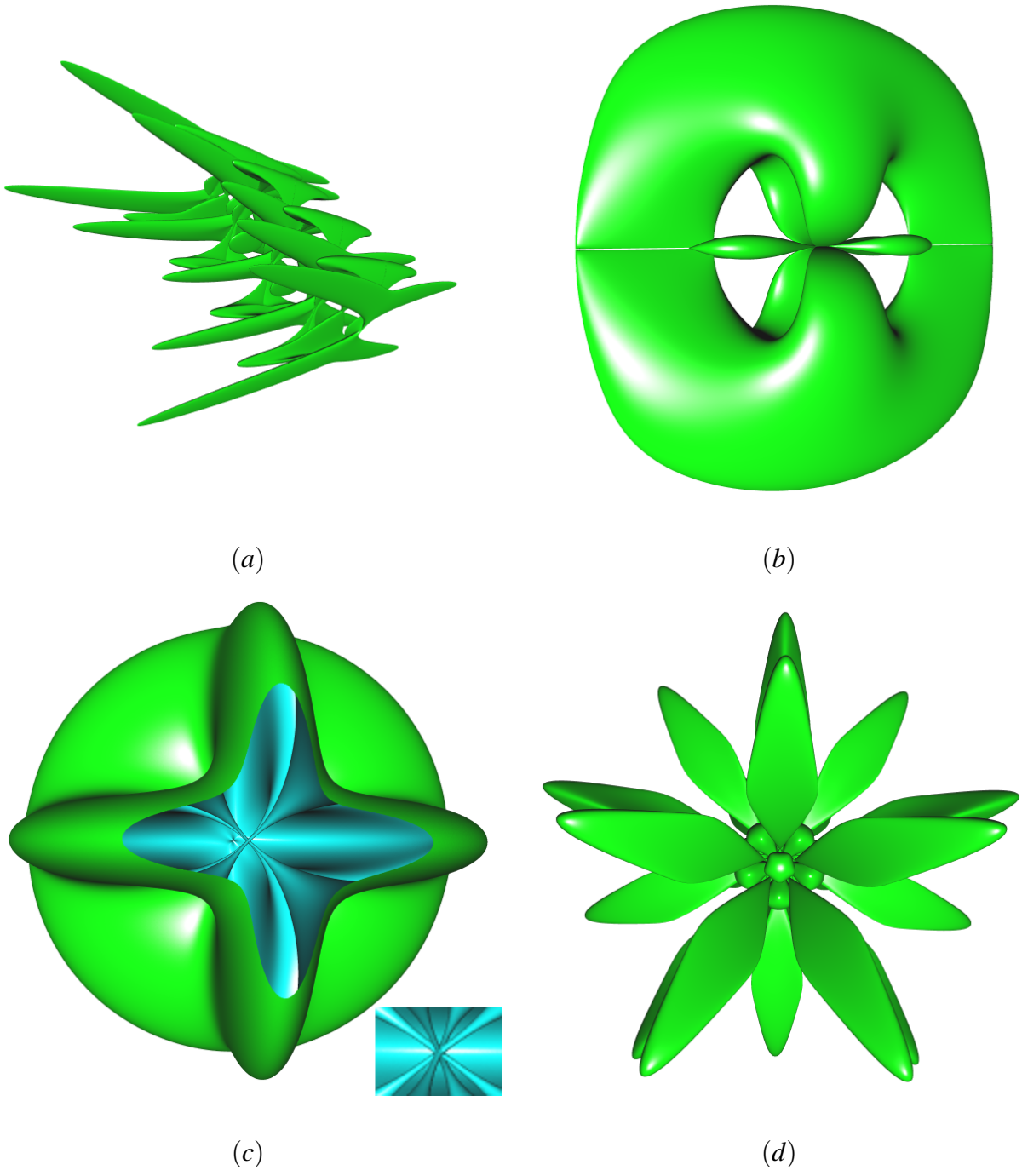


Figure 4.1: (a) The $K = -10$ Bretzel $f[4]$ curvature surface. (b) The $K = 3$ Mitre $f[5]$ curvature surface. (c) The $K = -0.001$ Steiner's $f[2]$ curvature surface cut by the plotting volume to show inner structure with inset showing inner region rotated to reveal 8 tubes meeting at origin. (d) The $K = 0.2$ Barth sextic $f[6]$ curvature surface.

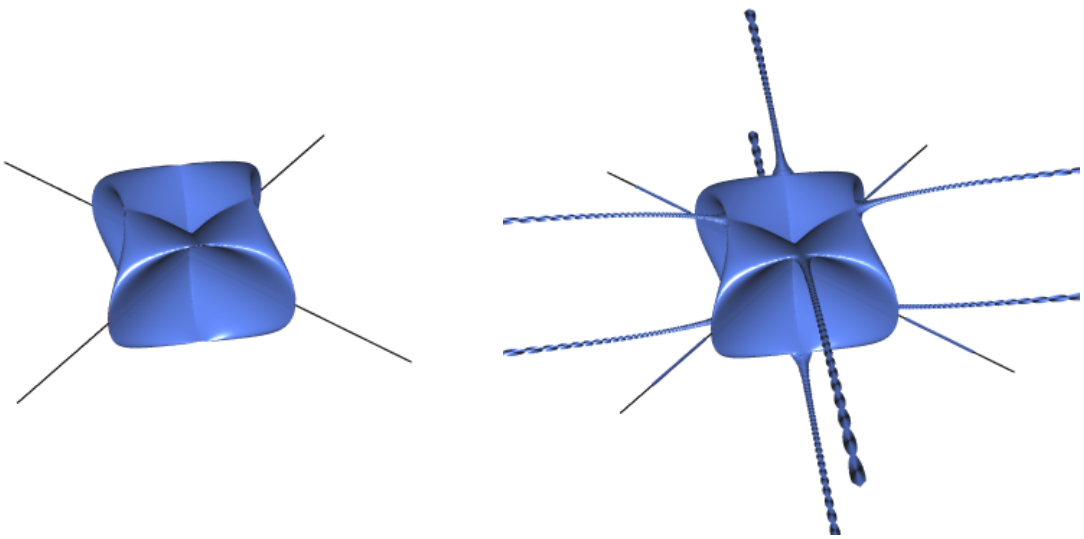
will still be a limiting factor. Another issue with ray tracing is that no information about surface topology is gained and thus numerous model operations cannot be performed.

4.1 Node subdivision algorithm

Bloomenthal [24][69] and Schmidt[27] used octrees with curvature driven subdivision to efficiently polygonise surfaces. Suffern *et al.* [18], Stolte *et al.* [20], Stolte [56], Balsys *et al.* [4] and Harbinson *et al.* [1] all used the natural interval extension to drive the node subdivision.

Here, we present a new algorithm that combines the relative speed of the point sampling approaches with the accuracy provided by the use of the properties of the natural interval extension of the function. This algorithm uses point sampling and the natural interval extension as a subdivision criteria and for non-manifold sections uses a pruning algorithm based on a signed distance function to prune plotting nodes around the non-manifold features to more accurately render these features.

We present this algorithm to address the limitations in the sign and gradient methods from chapter three. As discussed the ray detection using the sign test is limited to those which align with the node edges. Figure 4.2 (a) shows the bohemian star with rays not detected by the sign method. Any parts of the surface which lie inside a node and does not intersect the node edges will be missed. Incorporating the gradient test will detect these rays, however it will also overestimate and detect too many false positives around



(a) (b)

Figure 4.2: Bohemian star rendered with (a) Sign test, and (b) Gradient test.

non-manifold sections as in Figure 4.2 (b). To solve this problem, our new algorithm aims to prune away the excess nodes, without missing parts of the surface like the sign test.

The algorithm begins by defining a viewing cube with center point $C(x,y,z)$, and width w. This defines the parent octree node. Eight child nodes are created with center points, $C_{i=1,8}(x\pm\frac{w}{4},y\pm\frac{w}{4},z\pm\frac{w}{4})$, and width $w/2$. These nodes converge to the surface based on the interval exclusion test. The use of rectangular boxes results in a tiling of space that is guaranteed to bound the surface. We can therefore use each of these nodes as though it were a root node.

If required the subdivision process will then recursively descend to a maximum plot depth d, but a child node will be further subdivided only if the natural interval extension

$F(X,Y,Z)$ of the surface function $f(x,y,z)$ has 0.0 in the interval, see Stolte [56] and Balsys *et al.* [4]. This indicates that the surface *may* be present in the interval defined by the vertices of the octree node. Only if we are at the final plot depth, and if the surface may be present, do we render the surface in the plotting node.

If the surface is not present in the node we do not subdivide further and this is the step that provides the algorithm its efficiency as it eliminates sections of space from further consideration. If the node may contain a surface element then we either further subdivide, or if we are at the maximum plot depth, define a point that will be used to represent the surface in the plotting node. We can do this as we can use plot depths that result in nodes that subtend an area of less that one pixel on the image plane, see Balsys *et al.* [51].

In Algorithm 4.3 the function $find\_point\_in\_node()$ contains the steps we take to decide whether or not a point will be rendered in the node, depending on the established criteria. Here, we use point sampling at the vertices of the plotting node to determine the signs of the function $f(x,y,z)$ at the vertices. If any of these evaluates to 0.0 we push the vertex's coordinates into the point list as the vertex is *on* the surface. Although it's not common, a vertex can lie exactly on a surface, and ignoring this can result in rendering artifacts. This is particularly important for rendering rays aligned along an octree axis.

We need an approximation for the point $f(x,y,z) = 0$ and the surface gradient $g(x,y,z)$ in the plotting node as these are both needed to render the point with shading. As we are working with implicitly defined points the gradient function $g(x,y,z)$ is computed by

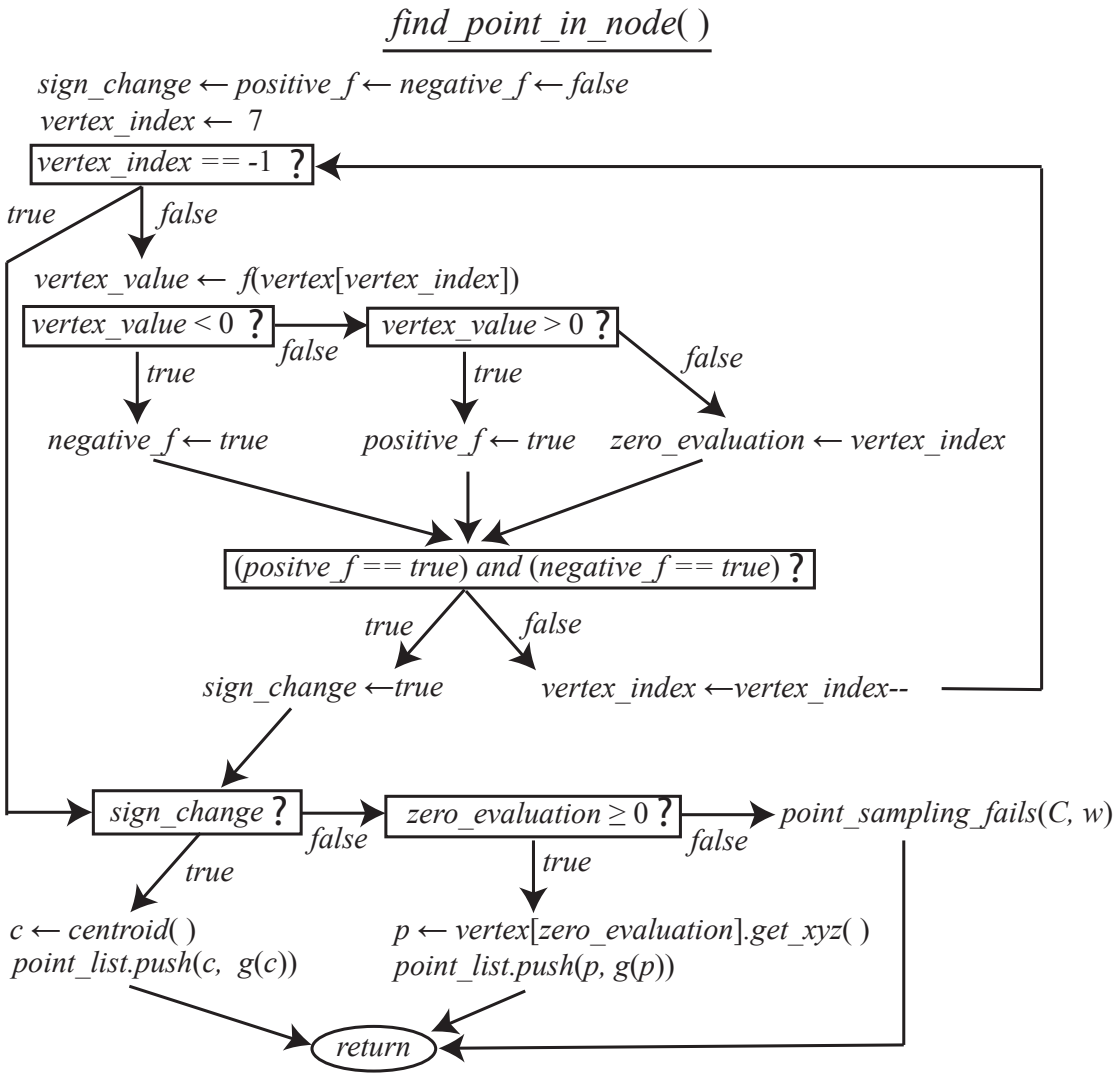


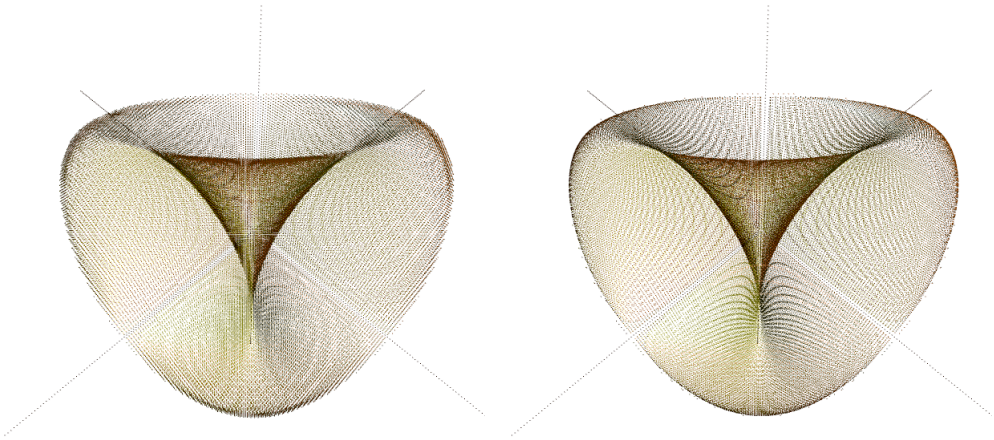
Figure 4.3: **Algorithm 1**. *find_point_in_node*(). Algorithm for putting point data into a vector for display. Here, $C(x,y,z)$ is the center coordinate of the plot node, c the centroid of the polygon contained in the node, p the coordinates of the vertex with a zero evaluation, and w the node width. The *vertex* array holds the 8 computed corner point function values, f, being rendered; $g(x,y,z)$ is the gradient function, ∇f.

numerically evaluating ∇f using forward or backward differencing.

If we detect a sign change between any two vertices of the node, part of the implicit surface must cross an edge of the plotting node, and so the surface is unambiguously present. We explored two options for determining which point to use in the plotting node to represent the surface, with the results shown in Figure 4.4 for Steiner's surface [70], see equation $f[2]$. In Figure 4.4 (a) we use the center of the plotting node. In Figure 4.4 (b) we use the centroid of the polygon. This results in points that are closer to the surface in the node and results in a smoother surface appearance with less shading artifacts. For illustration purposes we used a low subdivision depth so the points do not completely cover the surface. This makes it appear semi-transparent. We timed these approaches, for a number of surfaces with the results given in Table 4.1.

Table 4.1: Surface timing (sec) for rendering using node center (NC) and polygon centroid (PC) as the render point with plot depth of 9 using CPU algorithm.

Surface	NC	PC	Ratio
Mitre	1.72	2.53	0.68
Cyclide	24.50	25.86	0.95
Steiner's Roman	5.83	6.43	0.91
Chmutov 8^{th}	272	278	0.98
Bohemian star	127	132	0.96
Boy's	28.89	29.84	0.97



(a) (b)

Figure 4.4: Steiner's Roman surface f[2] , partially rendered with points, using points that are, (a) Always at the center of the plotting node, and (b) At the centroid of the polygon found by polygonizing the node.

These timings show that there is about a 5% penalty in finding the polygon centroid rather than using the node center. As using the polygon center gives better visual results, we use this approach. Since the surface is present in the node we can find the centroid of the polygon by simply averaging the surface intersections along the node edges. We investigated the use of a gradient descent method to move in the direction of the zero of the $\nabla g(x,y,z)$ in the plot node, but this appreciably slowed the algorithm with little improvement in image quality.

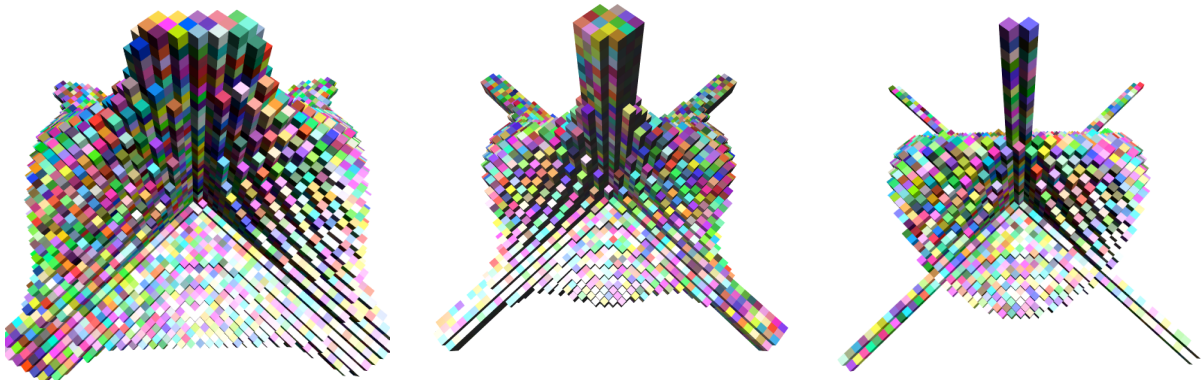
If no intersection occurs along a node edge it is still possible that at least one of the node vertices is exactly on the surface as previously discussed. In this case we use the

coordinates of the zero vertices as the point in the node. This is particularly important for rendering rays aligned along an octree axis.

If neither of the above situations has occurred we have a quandary. Point sampling fails in the plotting node, but the interval test indicates the surface may be present. As we render to high plot depths where the projection of the plotting node is less than the size of the pixel we could, naively, just push a point at the center of all such nodes and thus render all the extra points.

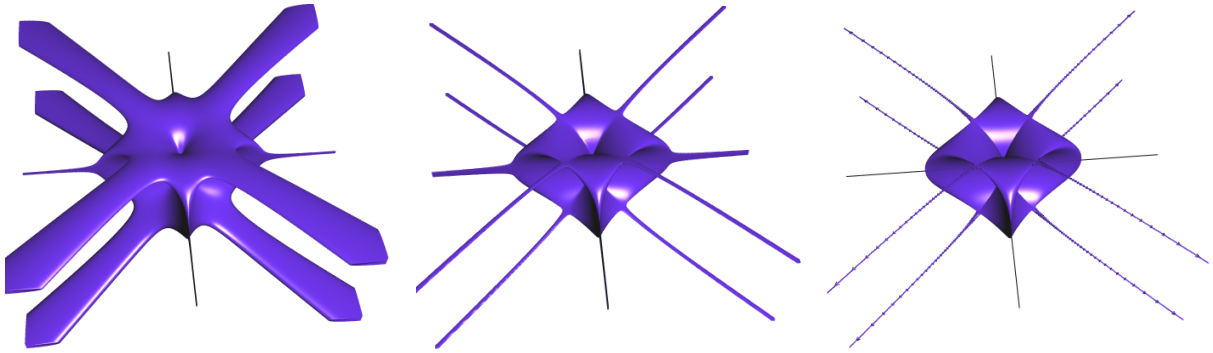
However, it is well known that intervals overestimate the possibility that the surface exists in an interval. The test guarantees not to miss regions that may contain the surface, but it is not an inclusion test. So pushing points in the nodes that fail the sign test will result in too many points being generated, particularly around thin sections, cusps, ridges, tube and ray-like sections. This is undesirable as it results in rendering artifacts around such features. A possible solution is to use affine arithmetic [39], [71], in place of the interval test as affine arithmetic intervals are tighter than those from intervals.

We examined this problem in some detail. Figure 4.5 (a) and (b) shows the nodes of Steiner's Roman surface produced using interval subdivision and affine arithmetic. These illustrate how the number of nodes produced increases around the rays and cusps. Although affine arithmetic produces less nodes than intervals, neither technique produces accurate images of the non-manifold features. Moreover, the time penalty for using affine arithmetic can be high. The images in Figure 4.5 (a) and (b) were rendered on the CPU



(a) (b) (c)

Figure 4.5: Steiner's Roman surface $f[2]$ rendered at a plot depth of 5 showing nodes that are found. (a) Using intervals, (b) Using affine arithmetic, and (c) Using the node pruning algorithm.



(a) (b) (c)

Figure 4.6: The bohemian star surface $f[13]$ rendered using (a) Interval, (b) Affine arithmetic, and (c) Interval pruning exclusion test at a plot depth of 9.

to a plot depth of 5. The resulting timings are given in Table 4.2. For this example affine arithmetic is five times slower than the equivalent interval approach.

Table 4.2: Surface timing (sec), and number of nodes (millions) found using interval, affine, and interval pruning exclusion tests at a plot depth of 9.

	Intervals		Affine		Pruning	
Surface	time	nodes	time	nodes	time	nodes
Mitre	1.35	1.75	41.2	1.42	1.72	1.40
Cyclide	6.98	9.09	87.0	1.83	24.5	1.82
Steiner's Roman	2.04	2.20	20.4	0.61	5.83	0.55
Chmutov 8^{th}	34.7	45.9	251	1.79	272	1.44
Bohemian star	16.8	18.5	142	1.18	127	0.84
Boy's	6.91	5.09	131	0.62	28.9	0.56

As a result we looked for a method to prune the "false positives" from the interval test. Because we subdivide the space using octrees, we can generate the values of points in the neighbourhood of $f(x,y,z)$. In particular, we can represent these neighbour points as $f(x_i,y_i,z_i)$ where $i \in \{-1,0,1\}$. Using these points we test the relation between the center of the plot node at $f(x_0,y_0,z_0)$ and the neighbour centers at $f(x_i,y_i,z_i)$. We use the $|f(x_i,y_y,z_i)|$ to determine their distance to the surface at $f(x,y,z) = 0$. If three neighbour points on a side of the node are all closer to the surface, then the plot node $f(x_0,y_0,z_0)$ is discarded by the algorithm, as there is a planar patch in the neighbourhood of the point that

is closer to the surface. Using this approach we get, for example, a small set of plotting nodes that bound the line for a ray passing through the octree. Even if we zoom in we will get a similar small set of points for the ray. This is evident even at a low plot depth = 5, as in Figure 4.5. In Table 4.2 we give the number of plot nodes produced by the three approaches. From this Table we see that the interval pruning approach produced 29% of the nodes using intervals alone, and 45% of the nodes produced using affine arithmetic for this example. This is typical of the results we get for the wide variety of functions we have tested.

Our solution is encapsulated in the function $point\_sampling\_fails()$ in Algorithm 2. The reduction of nodes produced for Steiner's Roman surface is shown in Figure 4.5 (c). The timing using this approach is also given in Table 4.2. The interval test is fastest, with our algorithm being about half as fast. The affine arithmetic test is about 5 times slower than the interval test and about twice as slow as our approach. From Figure 4.5 it is obvious that more pruning has occurred using our algorithm than occurs with using affine arithmetic, a far slower approach. We conclude that the algorithm is fast and efficient at node pruning, and perceptually produces the highest quality images.

Each node in the $3 \times 3 \times 3$ $voxel\_grid$ is defined with center coordinate points p, of nodes of width w, centered on the current plot node with center $C(x,y,z)$. The 27 points are then the centers of the 26 face, edge and corner neighbours of the plot node, and the plot node itself. Each node is tested if it may contain the surface, and if so, the node is flagged and the function value is stored.

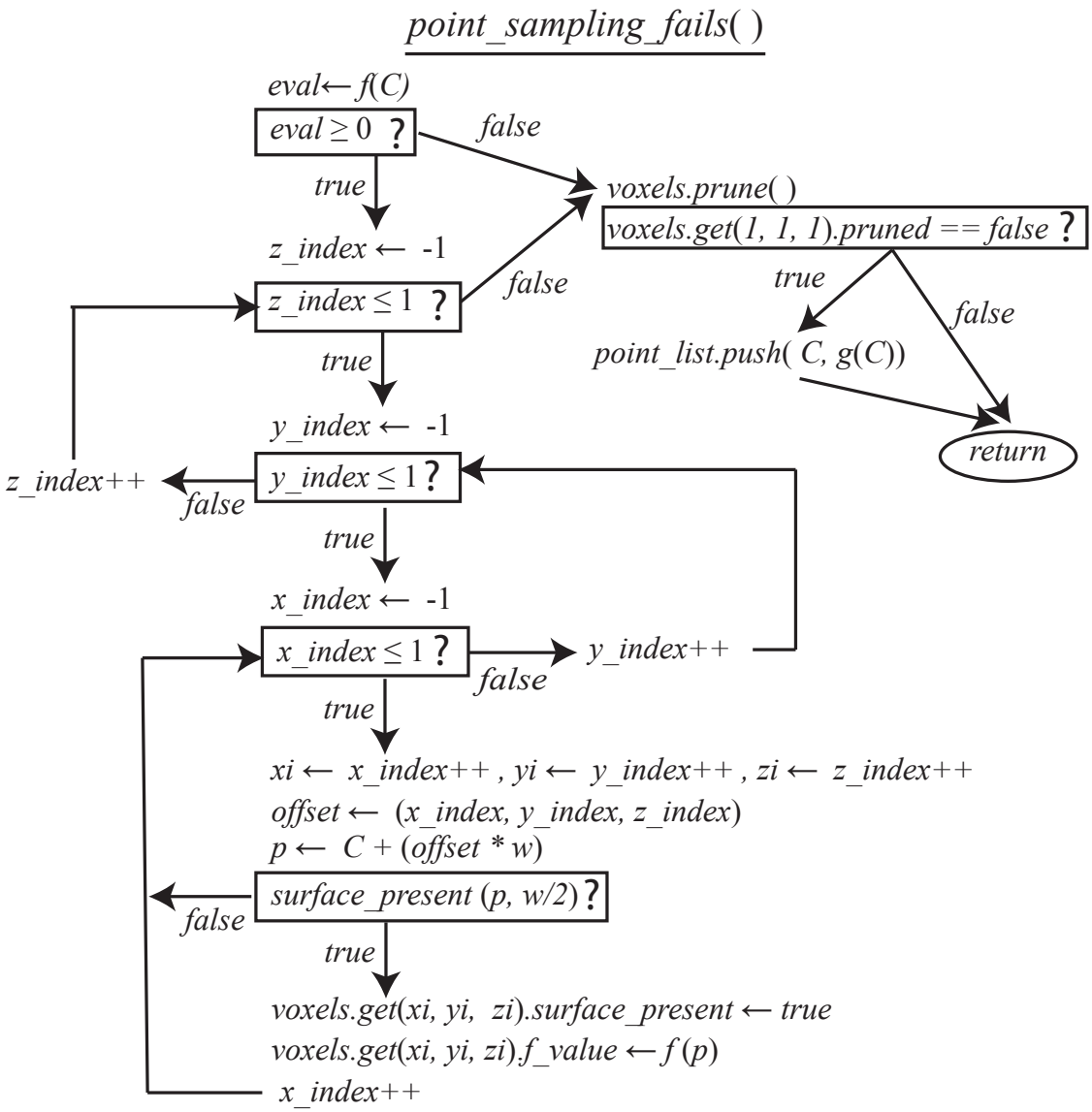


Figure 4.7: **Algorithm 2**. *point_sampling_fails*(). Here f is the function being evaluated. C is the center coordinate of the plot node, and w is the width of the plot node. A *voxel_node* is a *struct* that stores the point coordinate, normal, function value, and flags for whether the surface is present in the node, and whether the node is to be pruned or not. A *voxel_grid* is an array of $3 \times 3 \times 3$ *voxel_nodes*'s centered on the current plot node. Then a *voxels* is of type *voxel_grid*.

Next, the method $voxels.prune()$ is called. This considers all 26 neighbouring nodes to the central plot node, to determine whether the plot node should be added to the vector of points on the surface. We do this by first removing neighbours that do not contain surface elements. Neighbour nodes that may contain surface elements then have the function $f(x,y,z)$ evaluated at their centers. If the value of $|f(x,y,z)|$ of the is less than the value for the plot node, the neighbour is closer to the surface than the point, and we increment a count variable by one. When the count of closer neighbours on a side reaches three we mark the plot node to be pruned, since the three points define a planar patch that is closer to the surface than the plot node.

Only if the plot node is not pruned do we push the plot node centre, $C(x,y,z)$, and gradient function, $g(x,y,z)$, into the output vector of points to be plotted. In this manner we fill the output vector with a large number of points that are close to the surface, and thus used to render the surface. The resultant normal and point values are used with Phong shading and a z-buffer to render the surface. We use the jitter based anti-aliasing approach from chapter two to anti-alias the resulting images.

4.2 GPU implementation

If we directly implemented our CPU algorithm on the GPU, the resultant algorithm would be inefficient. This is because, as we proceed down a subdivision pathway, we get to leaf nodes quickly down some pathways and not others. Ideally we need an algorithm

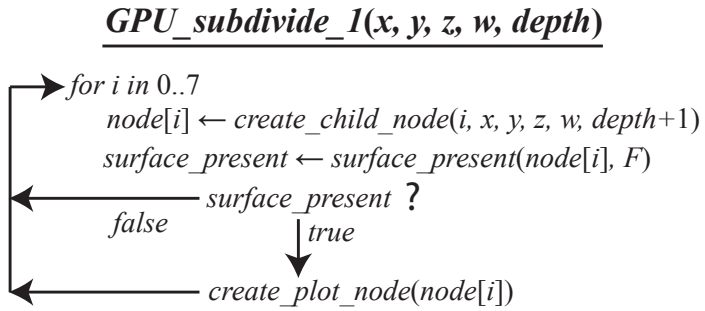


Figure 4.8: **Algorithm 3**. *GPU_subdivide_1*(). Eight child nodes (indexed 0-7) of input node created. If the surface may be present in the node the node is added to the vector of nodes to be subdivided. The nodes found in the first iteration of *GPU_subdivide_1*() each recall *GPU_subdivide_1*() and create new child nodes that repeat the process until the node vector reaches a set limit.

that will terminate all threads at approximately the same time. The worst case scenario is when the entire subdivision takes place by the subdivision of a single node as this would use a single thread.

Typical GPU's are organised as a set of multi-processors each of which has execution units. For instance, the *NVIDIA* 275 has 30 multi-processors with 8 execution units. Each execution unit can run *one* thread at a time, so this card can run 240 simultaneous threads. To obtain good performance from the GPU we need to try and keep all execution units running. This implies that we balance the load across the execution units.

We used *CUDA* to implement a GPU version of our algorithm. *CUDA* threads must begin execution in the same function. The GPU treats the display as blocks (the block

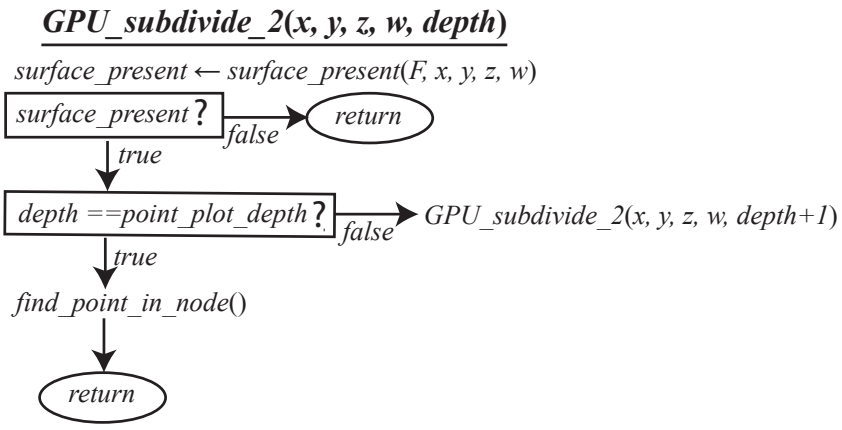


Figure 4.9: **Algorithm 4**. *GPU_subdivide_2*(). The nodes found in *GPU_subdivide_1*() each call *GPU_subdivide_2*(). If the surface may be present the node recursively calls *GPU_subdivide_2*() until a *point_plot_depth* is reached when the $find\_point\_in\_node()$ algorithm is called. Recursion is implemented using a stack on the GPU.

is identified by the *blockId*) of threads, the block contains an array of threads whose array size is defined by the *blockDim* value. An individual thread is then identified by specifying the threads *blockId*, the blocks *blockDim*, and the *threadId* within the block. Before launching a *CUDA* kernel we need to specify values that define the array size of the block and the number of blocks in the grid. The programmer optimises these for their application.

The main problem we solved is how to proceed with the node subdivision process so that each thread processes the same code, and when a thread finishes, it is allocated a new node subdivision task. We also need to be able to collect the points that are found in the subdivision process into a single array.

We arrived at a two step solution shown in Figure 4.8 as Algorithm 3 and in Figure 4.9 as Algorithm 4. In Algorithm 1 we subdivide the root node to a plot depth d. This can generate a theoretical maximum of d^8 nodes to be processed. The actual number generated will be less than this due to node pruning. As the GPU has limited memory we stop generating nodes before we run out of memory. On our GPU card with 980 MB of RAM this occurs when around one million nodes are generated.

The first iteration of Algorithm 3 has a single input, the root node. This will produce up to eight new nodes, the actual number will be returned as the next value of N. Each thread then uses an interval test on each child node to determine if it is to be processed. If it is, an *atomic* index variable is incremented. This index is used to index the position in a node array which stores the child node's center coordinate, width, and depth for later processing. When the function returns, the index is the count of the number of nodes N, to be processed on the next iteration.

The output nodes found in Algorithm 3 becomes the input to the next stage, Algorithm 4. Each starting node is recursively subdivided until its children are discarded or a child reaches the maximum plot depth. Recursion is not supported by the GPU, so we use an array to implement a stack of node elements. While the stack is not empty, we check if the current node's depth is less than the point subdivision depth. If so, we create child data structures for each child node that may have the surface present in it. If the maximum plot depth has been reached we process the node as described in the $find\_point\_in\_node()$ function, but on the GPU. As this is straightforward we don't elaborate it here.

Most of the figures in this paper were rendered at a plot depth between 9 and 11. We used a recursion depth of 5 to 7 during the first subdivision stage (Algorithm 3), and a depth of 3 to 4 during the second stage (Algorithm 4). The depth used in the first stage is limited for two reasons. First, limited depth ensures that the time for each thread to execute is similar, and thus minimises stalls between the parallel tasks. Second, the computationally expensive process of finding a point in a plot node is performed in the second stage.

When we find a point on the surface and compute the associated normal we also use an atomic variable to assign an unique index to the array used to store the point and normal data. In this way each point and normal is added to a shared data structure, such that no data is lost through shared access. By storing the points in this data structure, we can quickly render new views of the surface without further evaluation of the implicit function. Table 4.3 compares the time taken to render a number of surfaces using the CPU and GPU algorithms. This shows that the performance increases on the GPU from a factor of 5 to 84 times, depending on the surface.

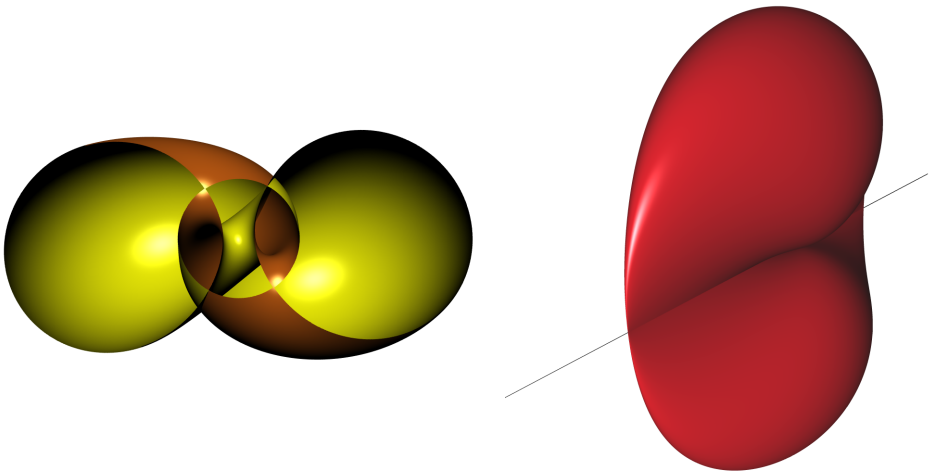
4.3 Examples

Surfaces that self-intersect, such as the Klein bottle surface [72] $f[3]$ in Figure 4.10 (a), and the Mitre surface [73] $f[5]$ in Figure 4.10 (b), are rendered without artifacts.

We can also render rays, as in Steiner's Roman surface $f[2]$, (Figure 4.4, and the

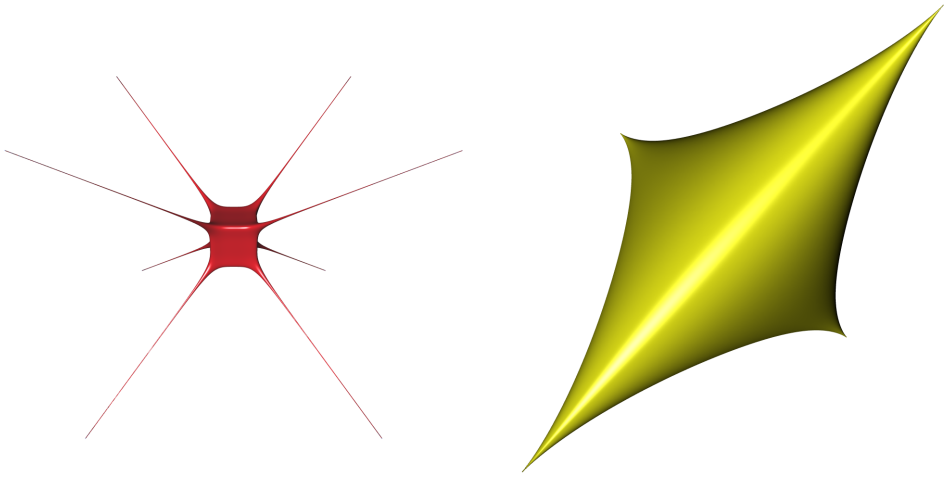
Table 4.3: CPU and the GPU algorithm timing for rendering surfaces at a plot depth of 9.

Surface	CPU (sec)	CUDA (sec)	Ratio
Mitre	2.53	0.44	5.7
Steiner's Roman	6.43	0.82	14.0
Cyclide	25.9	1.84	14.0
Chmutov 8^{th}	278	5.56	50
Bohemian star	132	2.94	44.8
Boy's	29.8	1.52	19.6
Chmutov 14^{th}	2233	26.41	84.5



(a) (b)

Figure 4.10: (a) The Klein bottle surface $f[3]$ cut by the root node to show internal structure. (b) The Mitre surface $f[5]$.

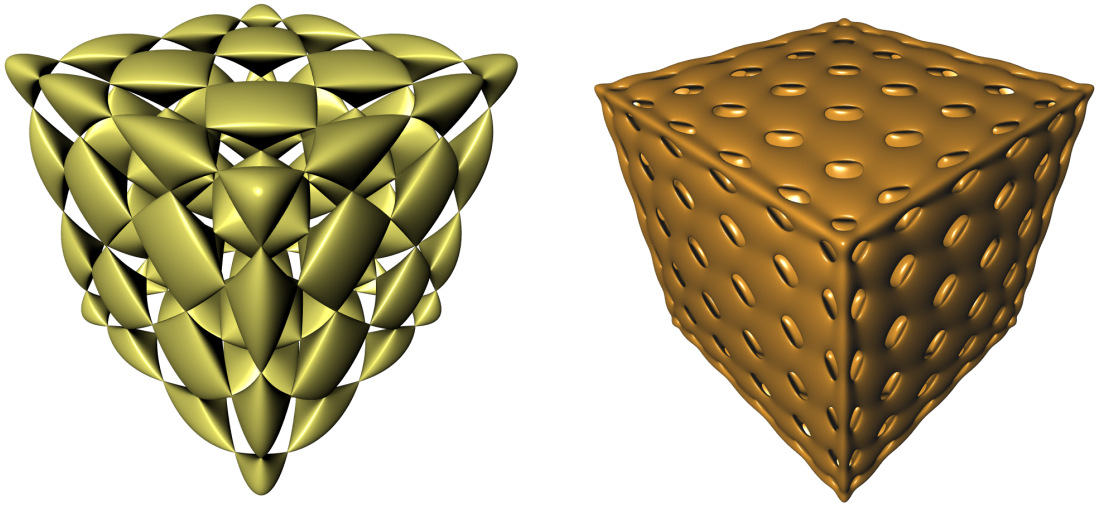


(a) (b)

Figure 4.11: (a) The Spiky surface $f[8]$. (b) The Kusner-Schmidtt surface $f[15]$.

Mitre surface (Figure 4.10 (b)). Furthermore, we can render ray-like tubes as in the Spiky surface [4] $f[8]$ in Figure 4.11 (a), which have elongated spikes asymptotic to the four lines $x = \pm y = \pm z$. These become rays in the limit as $(x,y,z) \to \infty$. Other non-manifold features we can render are cusps / ridges, as on the Kusner-Schmitt surface [74] $f[15]$, in Figure 4.11 (b), and in Figure 4.24 (b) the super-toroid [75] $f[12]$ surface. Finally, we can render thin and flat sections as shown in 4.10 (b) and 4.24 (b), thin sections in 4.11 (b), and multiple double points as shown in Figures 4.12 and 4.21.

Our algorithm can render high order algebraic surfaces. In Figures 4.12 (a) and 4.12 (b) we render the 8^{th} and 14^{th} order Chmutov surfaces [76] given by $T_n(x) + T_n(y) + T_n(z) = 0$, where $T_n(x)$ is an n^{th} order Chebyshev polynomial of the first kind. For $n = 8$ this is equation $f[9]$. In Figure 4.12 (b) the 14^{th} order Chmutov surface $f[10]$ is shown.



(a) (b)

Figure 4.12: (a) The T_8 Chmutov surface f[9] , and (b) The T_{14} Chmutov surface f[10] .

Plot depths of 9 were required to render these surfaces without holes. The complex internal structure of the T_{14} surface is shown in Figure 4.13 by clipping it with the plane $f(x,y,z) = x+y+z = 0$.

We can easily render the intersections of a plane (or planes) with implicit surfaces as illustrated in Figure 4.22 (a), and the intersection of arbitrary implicit surfaces can be done in the same manner. For the intersection of a set of planes with the Klein bottle surface [72] f[3] in Figure 4.22 (a) we test for the point in the plot node being on the plane and on the Klein bottle surface. The test for the intersection of two general surfaces involves testing whether the point $P(x,y,z)$ in the plot node is on both surfaces.

It is also simple to render textures with our point-based algorithms. We illustrate this

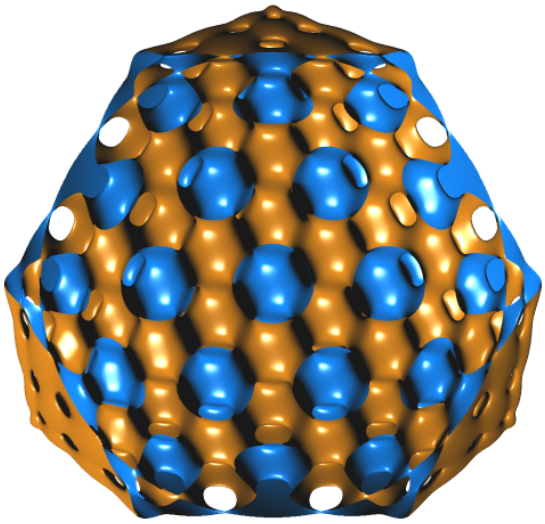


Figure 4.13: The T_{14} Chmutov surface cut by the plane $x+y+z=0$, and colour coded yellow for outside the surface and blue for inside the surface.

with projected textures on the Barth sextic surface [77] f[6] in Figure 4.21 (b). As the implicit surface has no texture coordinates, the textures are projected across each axis onto the surface.

4.4 Comparison with ray-casting

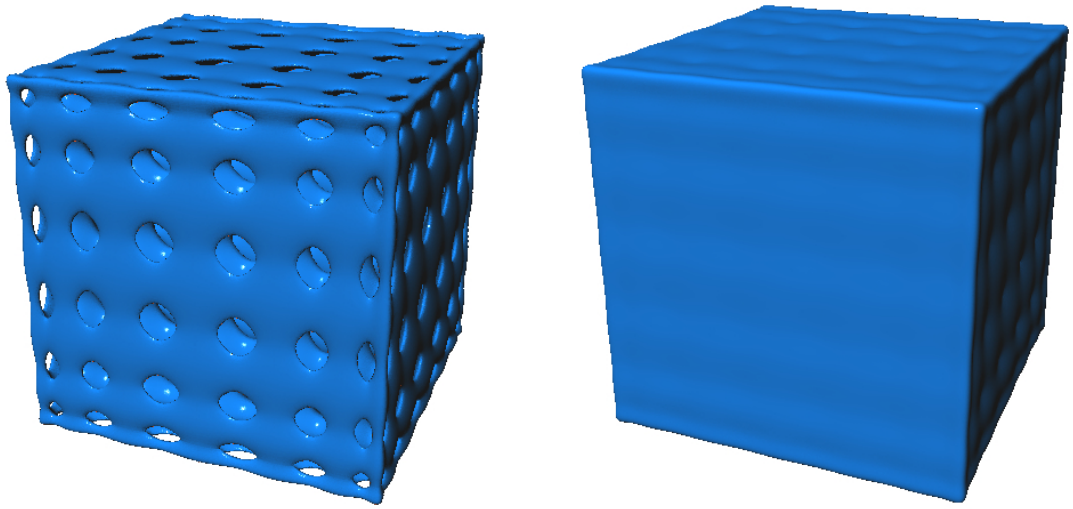
We implemented and compared the GPU interval ray-casting method in Singh and Narayanam [52] with our GPU approach in terms of image quality and speed using an Intel Core2 Quad PC and an *NVIDIA* 275 graphics card. We set the plot depth for the interval pruning method and then adjusted the interval precision term in the ray-caster to produce visually similar results with the same viewing geometry and image resolution. As

Table 4.4 shows, our GPU based algorithm is much slower than ray-casting for the simpler surfaces, but as the complexity of the surface expression increases, the difference in timing decreases. For the 14^{th} order Chmutov surfaces our method is faster than ray-casting. The timings for anti-aliasing with the method discussed in chapter two is very low, but we only used a single ray per pixel for the ray-casting. Ray-traced images are commonly anti-aliased with nine or sixteen rays per pixel, with the rendering times proportional to the number of rays used.

Ray-casting does not converge as fast when rendering high order algebraic surfaces and produces worse results, compare Figure 4.14 (a) with Figure 4.14 (b). Ray-casting does not render non-manifold features as sharply as our approach. Compare Figure 4.15 (a) and Figure 4.15 (b).

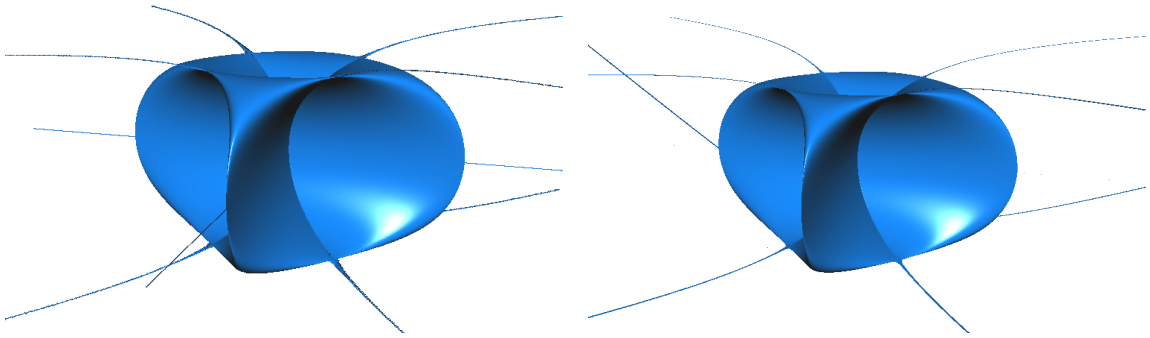
4.5 Rendering shadows on surfaces

As we store the geometry dataset from the subdivided implicit surface, we can render the geometry into shadow buffers and use GPU hardware shadow mapping, as shown in the following examples.



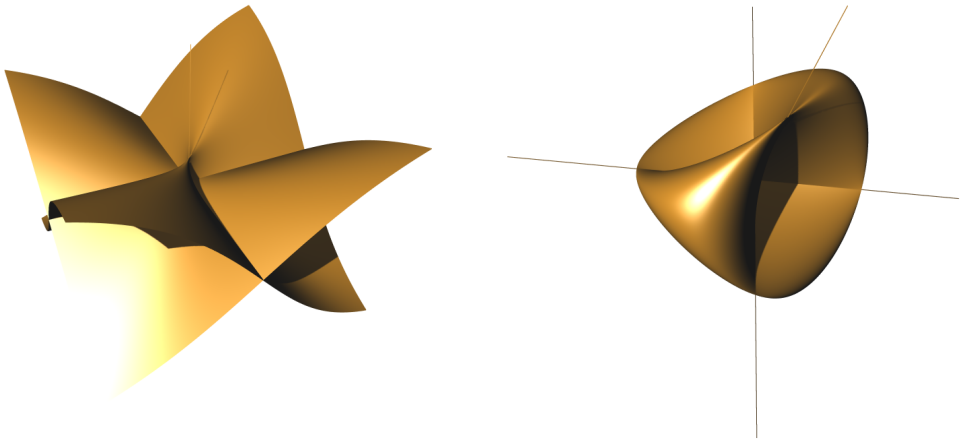
(a) (b)

Figure 4.14: The T_{14} Chmutov surface $f[10]$. (a) Point subdivided to depth 9. (b) Interval ray-cast with e^{-14}.



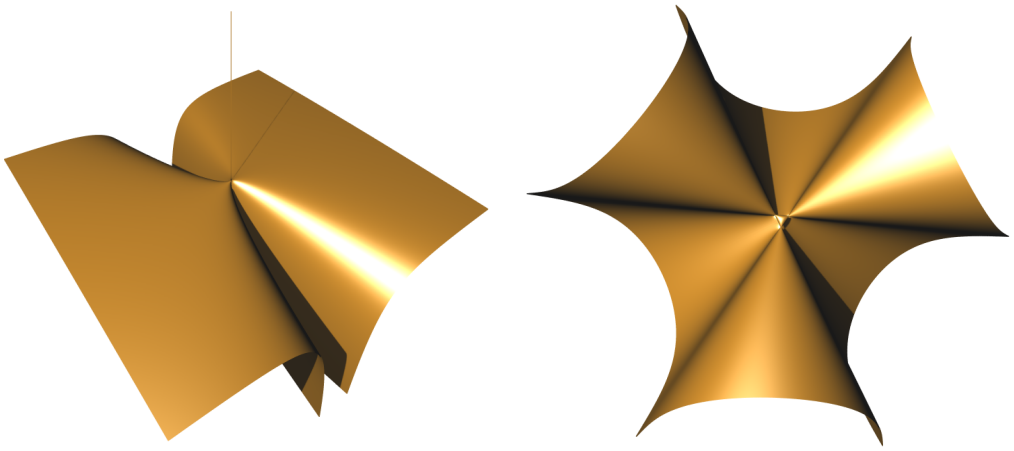
(a) (b)

Figure 4.15: The bohemian star surface $f[13]$. (a) Point subdivided to depth 9. (b) Interval ray-cast with e^{-14}.



(a) (b)

Figure 4.16: (a) Steiner's relative surface. (b) Steiner's roman surface.



(a) (b)

Figure 4.17: (a) Umbrella surface. (b) Cayley surface.

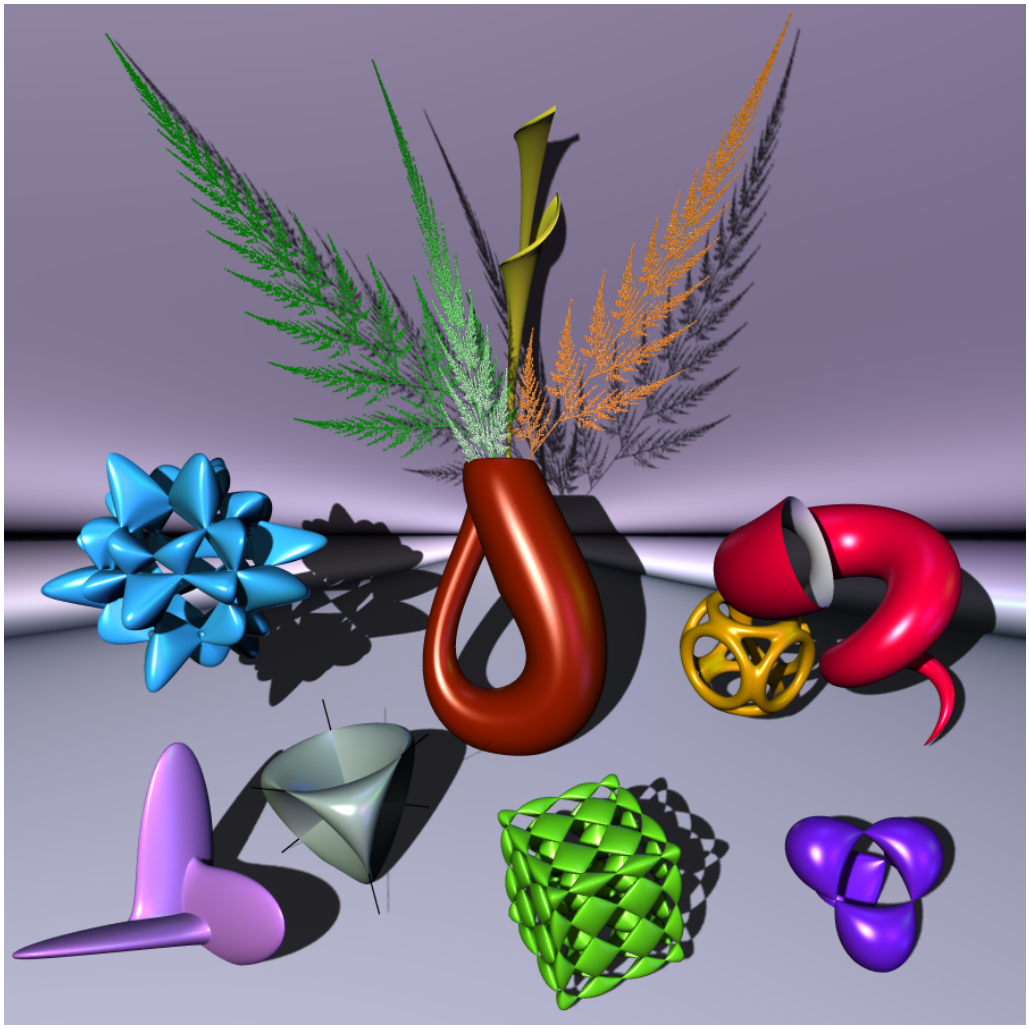


Figure 4.18: Scene composed of multiple objects.

Table 4.4: GPU timing (seconds) for various surfaces using interval pruning vs. ray-casting.

Surface	Interval Pruning	Depth	Ray-casting	eps
Mitre	0.64	10	0.19	e^{-9}
Cyclide	1.20	9	0.76	e^{-9}
Super toroid	1.57	10	0.6	e^{-10}
Barth sextic	3.91	9	1.78	e^{-12}
Boy's	1.87	10	4.60	e^{-16}
Bohemian star	101.	10	74.	e^{-16}
8^{th} Order Chmutov	76.	9	28.49	e^{-14}
14^{th} Order Chmutov	104.	9	473.	e^{-16}

4.6 Rendering Contours on Surfaces

In Balsys and Suffern [5] contours of user-defined thickness (slabs) were drawn on surfaces to aid their visualisation. While the surfaces in [5] were ray-traced, the algorithm can also be used with point rendering. The surface is coloured with the slab colour if the point is inside the slab, otherwise the surface is coloured using Phong shading. In [5] it is assumed the slab is approximately planar in the neighbourhood of the point. In regions of very high or very low curvature this assumption can break down. For slabs parallel to the coordinate planes it's simple to determine if a point is in one of the slabs since the slabs are not curved. We only need to check if the x, y, or z coordinate of the point is within $\pm d/2$ of one of the specified x, y, or z centers. This will work assuming the slab thickness is greater than $d/2$.

When the slabs lie on low or high curvature regions of a surface it's more complex to determine if a point is in one of the slabs. The algorithm discussed in Balsys and Suffern [5] produces contours of approximately constant width provided the slab width w is small compared to the curvature at that point on the surface. Sometimes, this approximation breaks down as the radius of curvature can be arbitrarily small or large. This occurred when rendering lines of constant Gaussian curvature $K(x,y,z) = c$ of the super-toroid surface. See Figure 4.24 (b), particularly around the edges of the cusps. The problem was even more pronounced for Boy's surface when rendering constant curvature slabs across the flattest parts of the blades where the algorithm suffered an almost complete

breakdown. We have therefore developed a number of alternate approaches to rendering curved contours on surfaces. Although we discuss these in terms of Gaussian curvature contours, they are applicable to any curved contours.

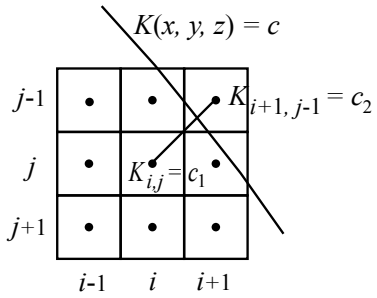


Figure 4.19: Determining whether a pixel has Gaussian curvature contour, $K(x,y,z) = c$, passing through it in image space.

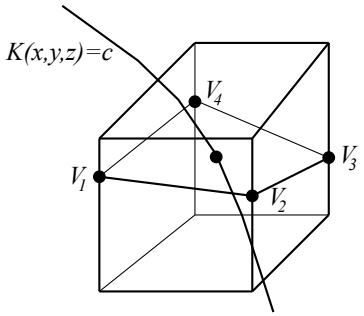


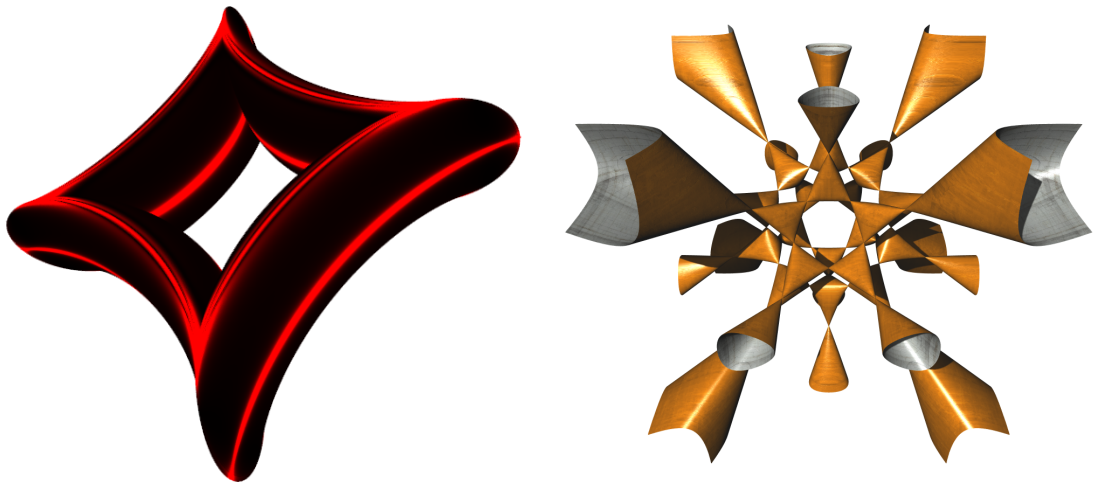
Figure 4.20: Determining whether a pixel has Gaussian curvature contour, $K(x,y,z) = c$, passing through it in object space.

The first of these algorithms works in image space. Along with the OpenGL surface *colour buffer* we use an additional buffer to store the (x,y,z) values of the depth buffer pixel on the GPU as floating point numbers. This allows us to look up the (x,y,z) coor-

dinate for depth buffer pixel (i, j) in the additional buffer. For visible surface pixels we compute the contour value, $K(x,y,z) = c_{i,j}$, of the $(i, j)^{th}$ pixel and all its neighbours (see Figure 4.19) using the positions in the GPU buffer. We compute the Gaussian curvature at the center of every pixel on the visible part of the surface to obtain a value $c_{i,j}$ for each pixel. The Gaussian curvature contour value $K(x,y,z) = c$ to be rendered is passed as an input parameter to our pixel shader. For the center pixel in Figure 4.19 we then make a series of comparisons to determine whether the pixel should be shaded with the contour colour. For example, we compute the Gaussian curvature at pixel (i, j) and $(i+1, j-1)$ as $K_{i,j} = c_1$ and $K_{i+1,j-1} = c_2$. We then evaluate if $c_1 \leq c \leq c_2$ and, if this equality is true, the contour passes near the pixel and we render the $(i, j)^{th}$ pixel with the contour colour rather than the surface colour. The $(i, j)^{th}$ pixel value is set to the contour colour if this condition occurs for any of the immediate neighbour pixels.

Our second approach works in object space. The plotting node is polygonised to find the roots $f(x,y,z) = 0$ on the plotting node edges. The polygon will have from three to six sides (Balsys *et al.* [4]). In Figure 4.20 we show the case where four roots have been found (labelled V_1, V_2, V_3 and V_4). The Gaussian curvatures, $K(x,y,z)$, at these four vertices are evaluated and their minimum (*min*) and maximum (*max*) values are found. We test if $min \leq K(x,y,z) = c \leq max$. If this is satisfied the contour passes through the polygon in the node and the pixel is rendered with the contour colour.

The advantage of the image space algorithm is that it is implemented on the GPU, is faster as it only renders contours for pixels that pass the depth test, and can be used

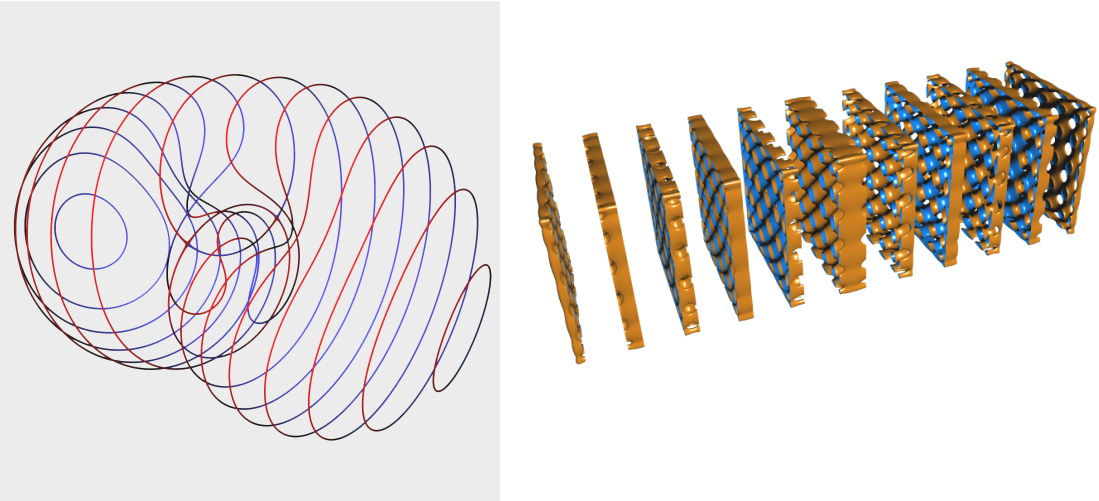


(a) (b)

Figure 4.21: (a) Super Toroid surface coloured with Gaussian gradient magnitude. (b) Barth sextic surface $f[6]$ with $t = (1+\sqrt{5})/2$ and $w = 1$.

for rendering contours in real-time. The disadvantage is that only the parts of the surface visible from the viewing vector $\vec{v}$, can be rendered. Therefore, the algorithm would only render the parts of the curves of self-intersection and zero Gaussian curvature in Figure 4.26 that pass the depth test. For these images we used the object space contouring method just described. As this is essentially a point sampling approach this can result in aliasing problems. The method discussed in chapter two is then used to produce anti-aliased contours.

Figure 4.24 (a) shows an implicit form for the bohemian dome surface $f[11]$ rendered with contours parallel to the (x,y,z) axes using a GPU algorithm. Contours can be rendered on any surface visible from the view point. Figure 4.24 (b) shows contours of

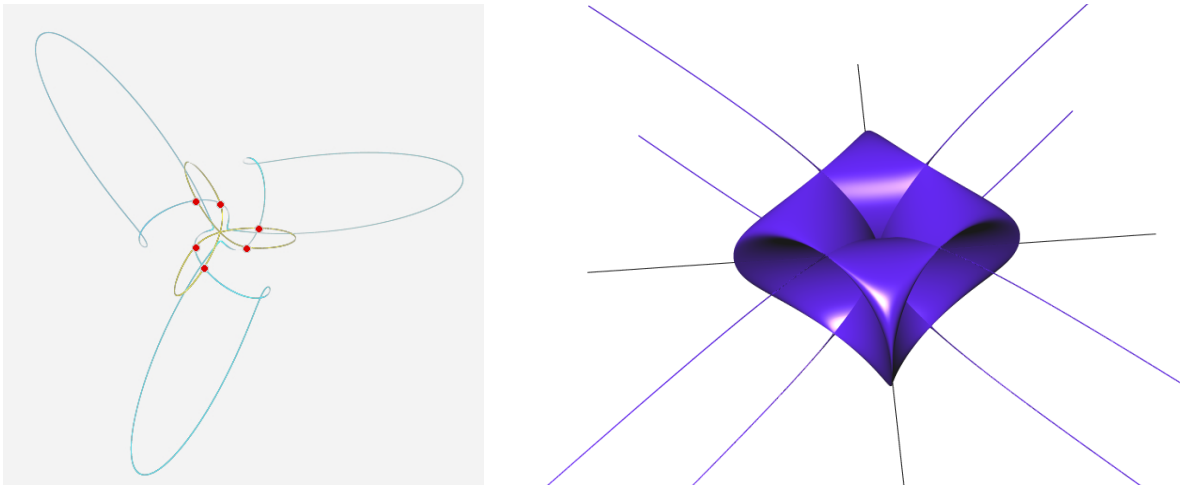


(a) (b)

Figure 4.22: (a) The intersection of the planes $z_i = -3.5, -3, -2.5, \cdots, 3, 3.5$, with the Klein bottle surface $f[3]$. (b) 14^{th} Order Chmutov split apart by planes across the z axis.

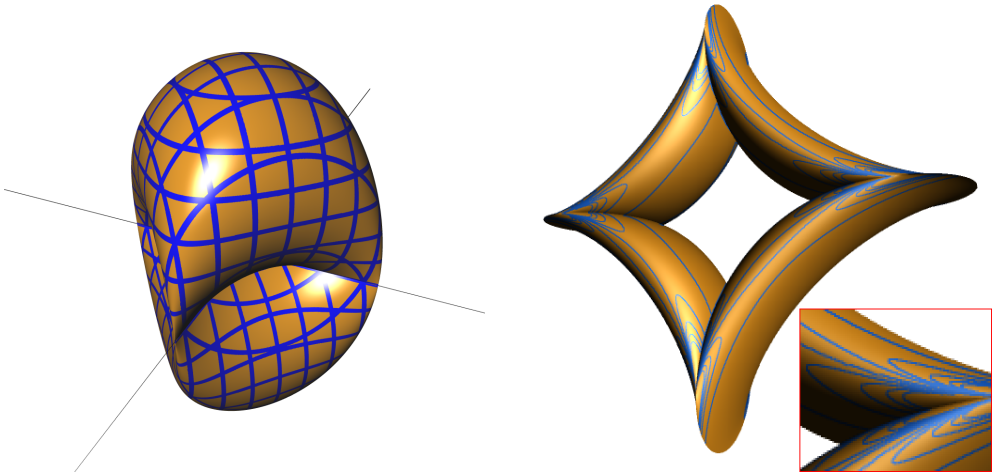
constant Gaussian curvature on the super-toroid surface [75] $f[12]$, rendered with the GPU algorithm.

To plot values of Gaussian curvature we first render the surface at the plot depth to find a range for the plotted points. We can also use a lower plot depth sample for finding the range. From the range we can step from a minimum to a maximum value using a computed step value to obtain various contours of Gaussian curvature on the surface.



(a) (b)

Figure 4.23: (a) The curve of zero Gaussian curvature and the curve of self intersection of Boy's surface $f[1]$ showing six points of intersection of the two curves (red dots). (b) The bohemian star surface $f[13]$.



(a) (b)

Figure 4.24: (a) The bohemian dome surface $f[11]$ with contour slabs parallel to the x, y, and z, principal axes. (b) The super-toroid surface $f[12]$ with $a = b = c = 4$, $\varepsilon_1 = 3$, and $\varepsilon_2 = 3$, and contours of constant Gaussian curvature rendered on the surface using the image space algorithm.

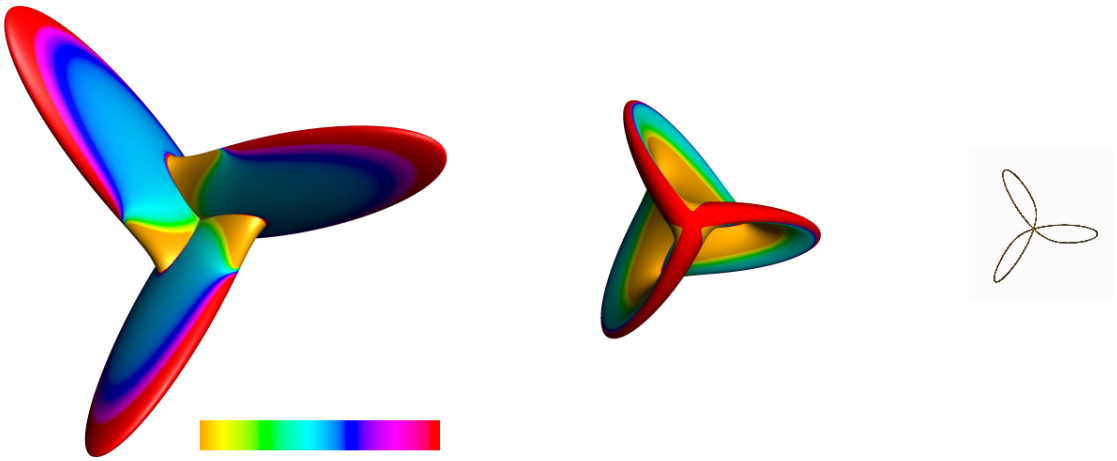


Figure 4.25: Curvature map for Gaussian curvature of Boy's surface $f[1]$, left image viewed from "front" camera position, middle image viewed rotated by 180° around the surface center. The right-most image is the curve of self-intersection of Boy's surface viewed from "front" position.

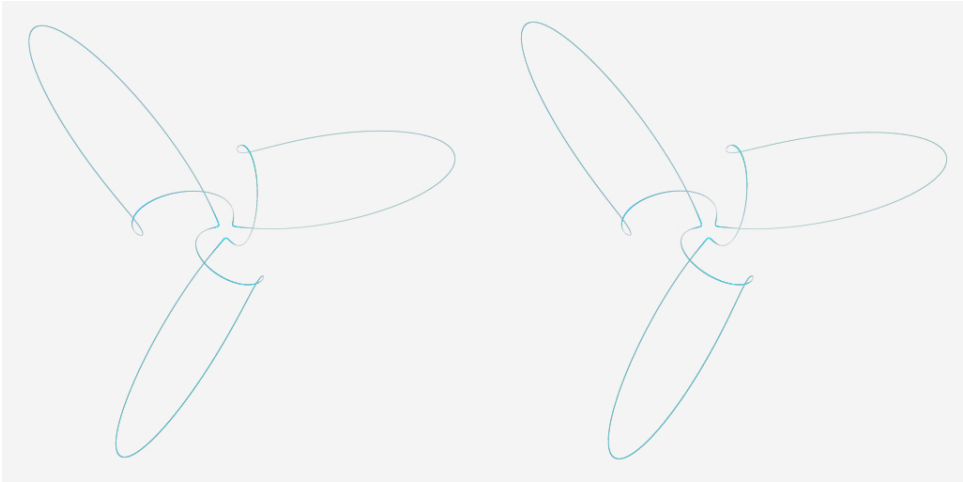


Figure 4.26: Right - left stereo pair for transverse viewing of the curve of zero Gaussian curvature on Boy's surface $f[1]$.

4.7 Curvature Maps and Curves of Self Intersection

Figure 4.25 show Boy's surface [78] $f[1]$ rendered with a Gaussian curvature colour map. The colour gradient represents the range of Gaussian curvature. Figure 4.25, left and middle, show the front and rear views of the surface. Figure 4.25 right shows the curve of self intersection.

Curves of self intersection are easily rendered by Algorithm 1 as the surface is singular along the curve (the gradient is not uniquely defined). We therefore need only plot points that are found in the $PointSamplingFails()$ section of the algorithm.

The formula for the Gaussian curvature of Boy's surface was too complex to calculate by hand. We used *Mathematica* to find all the partial derivatives of the surface and the implicit formulae. We then rendered only that part of Boy's surface whose Gaussian curvature is zero. The resulting curve is shown in Figure 4.26. A composite image, see Figure 4.23, shows that the zero Gaussian curvature line and the curve of self-intersection intersect in six points.

4.8 Curvature Surfaces

We conclude this section by rendering the curvature surfaces of a number of implicit surfaces that are the curvature surfaces of other non-manifold implicit surfaces. We also render the curvature surface intersected against its original surface. This helps visualise

the curvature of the original surface [54]. The Gaussian curvature can be used to classify points on a surface as follows. A point is elliptic if $K > 0$, hyperbolic if $K < 0$, and planar if $K = 0$. The curvatures of an implicit surface are defined from the 3D scalar field $f(x,y,z)$, and are themselves 3D scalar fields. Their values on the surface $f(x,y,z) = 0$ are the curvatures of the surface, while their values at points in space that are not on the surface are the curvatures of other surfaces $f(x,y,z) = c$, with $c \in \Re$.

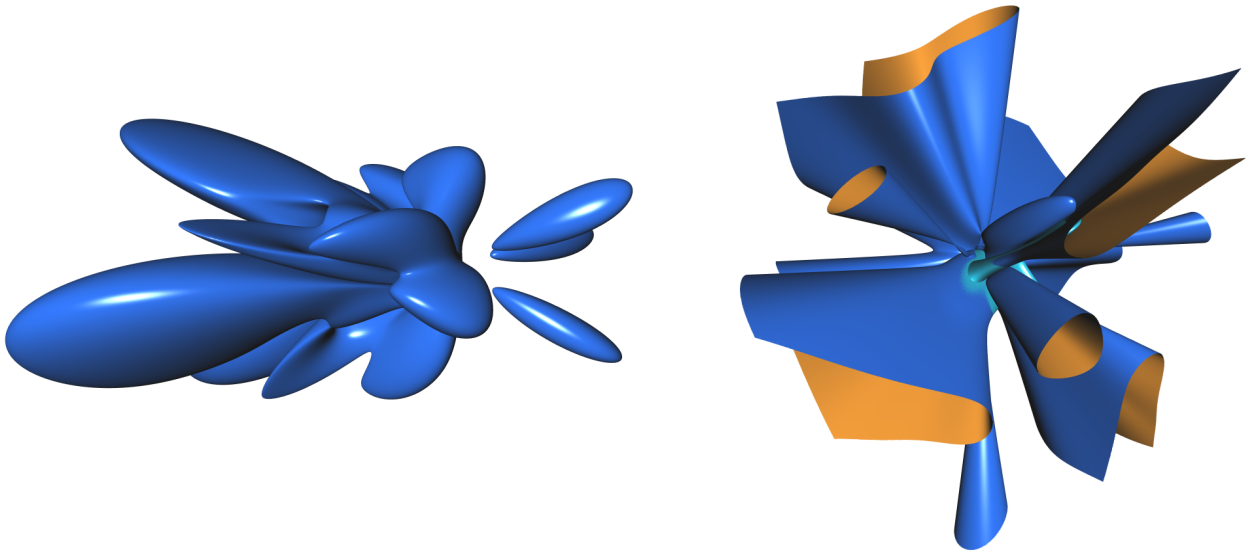
Below we give some examples of the Gaussian curvature surfaces. We do not give the expressions for these surfaces as each of them is many pages long. To shade the surface we also needed to find the first derivatives of the curvature surface, the method of approximating the normal from the scalar field's direction failed to produce acceptable results and so the calculation of the exact form for ∇f was a necessity. For a parent function like Boy's surface these steps were too tedious and error prone to do by hand and so we used Mathematica to implement these steps. As we were interested in curvature surfaces we needed to choose a value for c for the curvature, K, we are interested in rendering.

In Figure 4.27 we chose values of K= -1, 0, and 1 to render, as these represent the three classifications (elliptic, planar, hyperbolic) of curvature that occur on Boy's surface. The cyclide surface f[7] has three forms depending on the value of the parameter r relative to the parameters a and f. In the *ring* form, r is between the values of parameters a and f, in the *spindle* form r is less than a and f and in the *horn* form r is greater than a or f. Figures 4.28 (a)-(c) shows these three forms of the cyclide surface and their corresponding

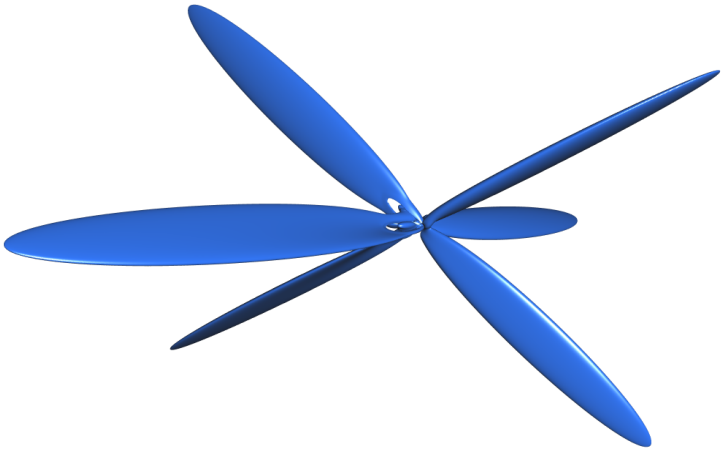
$K = 0$ Gaussian curvature surfaces in one figure. The figures are cut by the plane $y = 0$ and translated apart to reveal internal details.

The final example, Figure 4.29 (a), shows the Klein bottle surface and its $K = 0$ curvature surface again clipped against the plane $y = 0$, to show the internal structure. Figure 4.29 (b) shows the $K = 1$ curvature surface of the Klein bottle surface.

We believe this is the first time that curvature surfaces of these non-manifold surfaces have been rendered.



(a) (b)



(c)

Figure 4.27: The curvature surfaces for Boy's surface $f[1]$ showing, (a) $K_{boys} = -1$, (b) $K_{boys} = 0$, and (c) $K_{boys} = 1$.

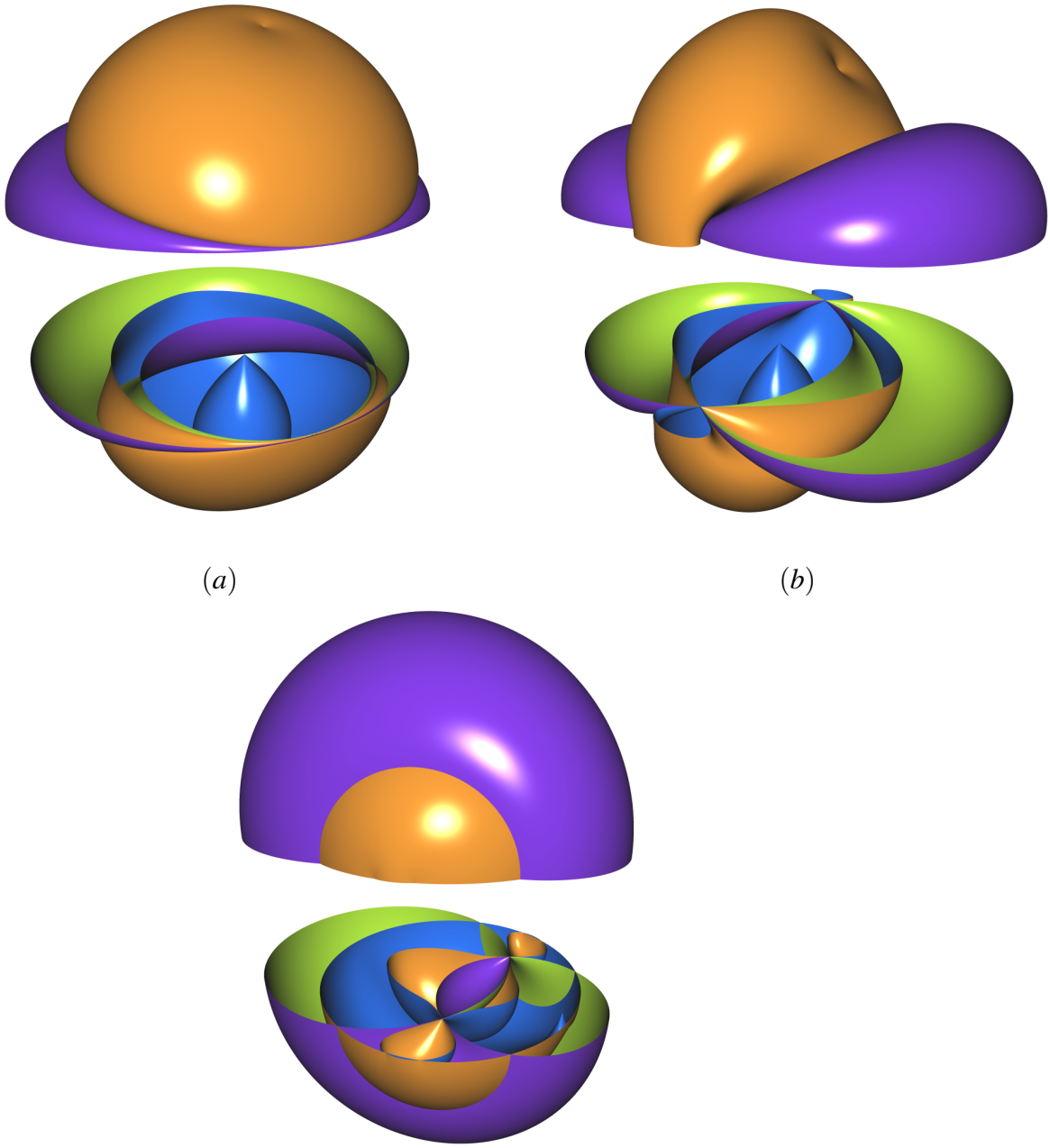
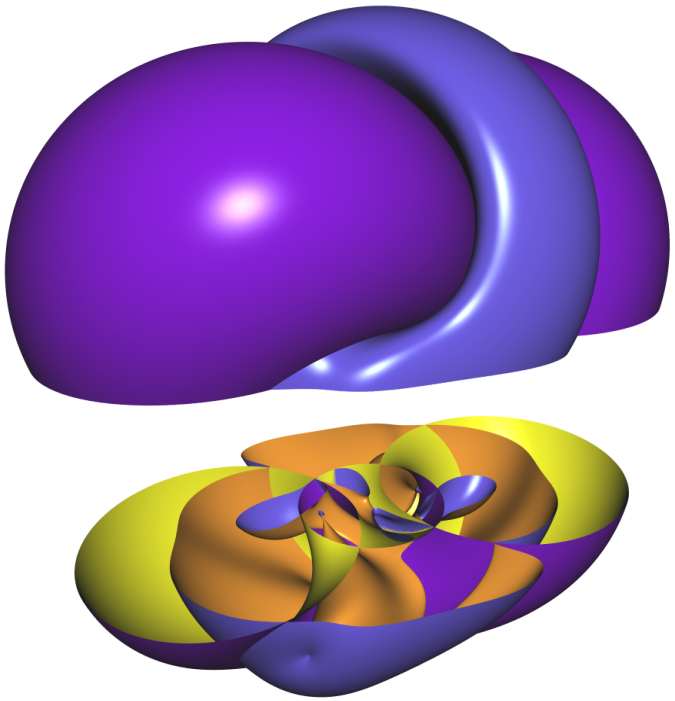
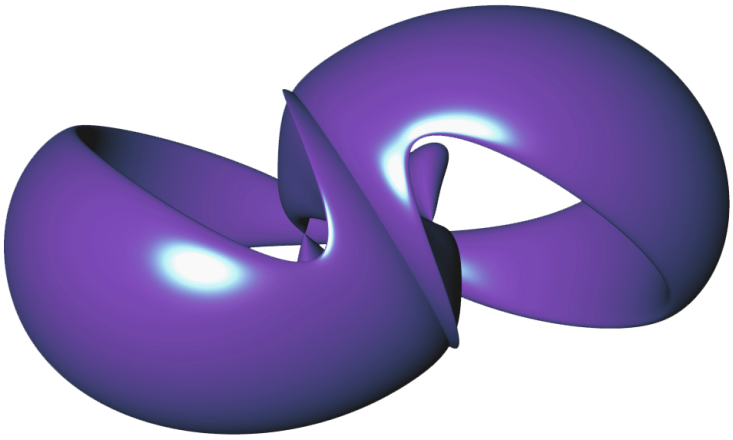


Figure 4.28: The three forms for the cyclide surface [3] surface $f[7]$ and their $K = 0$ curvature surfaces. Top row, (a) $a = 2.95, r = 3$ and $f = 10$, (b) $a = 5, r = 0.25$ and $f = 10$ and (c) $a = 7.5, r = 10$ and $f = 2.5$. Cyclide surface shown in Pink and Lime, Curvature surface in Blue and Gold. Surfaces cut and translated vertically to show internal structure.

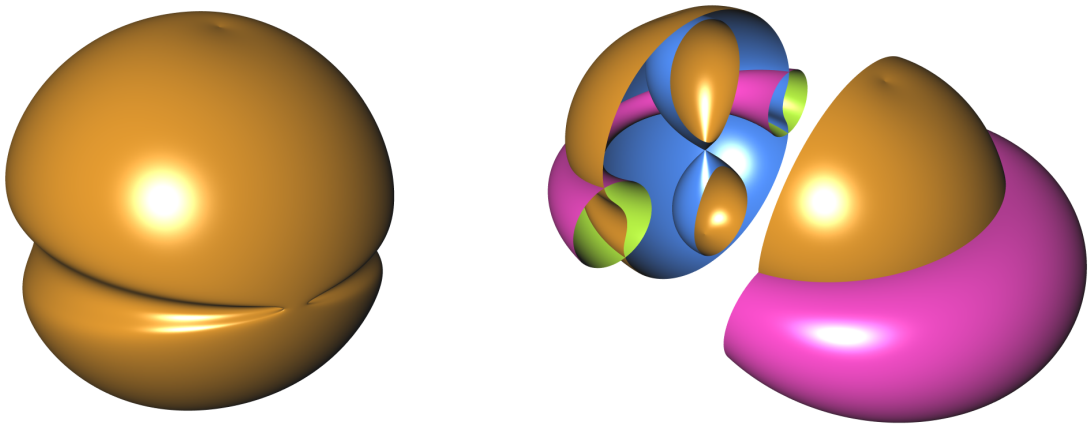


(a)

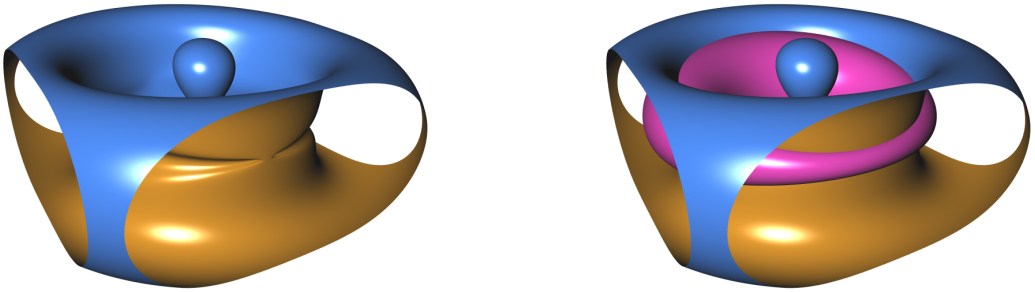


(b)

Figure 4.29: (a) The Klein bottle surface $f[3]$ and its $K = 0$ curvature surfaces shown cut against the plane $y = 0$ to reveal internal structure, and (b) the $K = 1$ curvature surface of the Klein bottle surface.

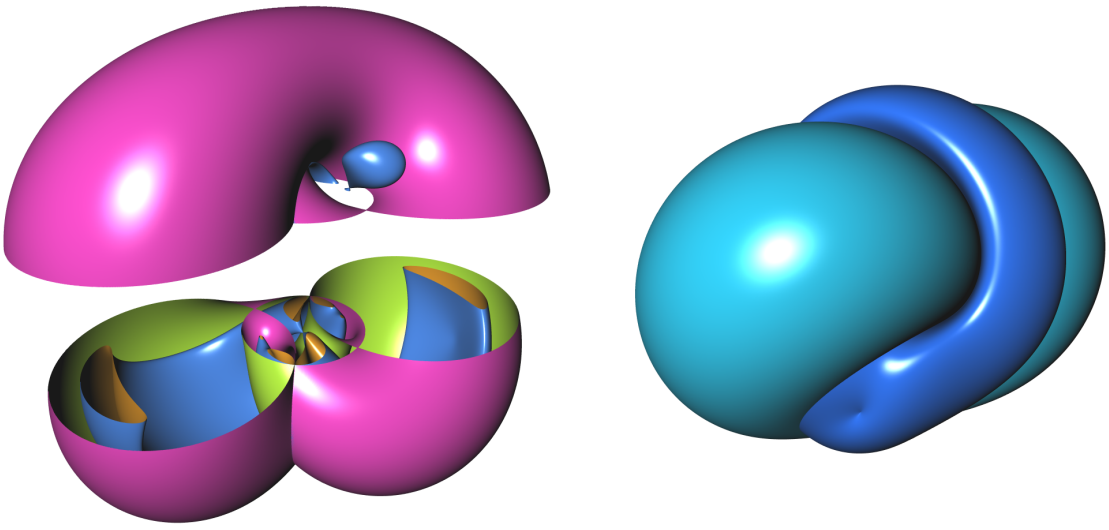


(a) (b)



(c) (d)

Figure 4.30: (a) Cyclide curvature surface for k=0, (b) Combo split of k=0 and cyclide, (c) Clipped cyclide curvature surface for k=0.05, and (d) Clipped combo of k=0.05 and cyclide.



(a) (b)

Figure 4.31: (a) Klein bottle combo for k=1. (b) Klein bottle combo.

Chapter 5

Visualisation using Polygons

We are interested in producing high quality animations of manifold and non-manifold implicit surfaces. We are particularly interested in animations of surfaces whose 3D shape is hard to conceptualise from static images. We are also interested in animations of parameterised implicit surfaces as we change the parameter. We compare three approaches to these visualisations and compare their relative advantages and disadvantages. The approaches we compare are the use of mixed dual-grid polygons / points, point splatting and ray-casting.

Our polygons/points method is octree based, uses interval and point sampling of the surface curvature to drive the subdivision, and uses a pruning algorithm to generate points that are rendered where the surface curvature is very high, or around non-manifold features. This allows us to visualise both manifold and non-manifold implicit surfaces of high complexity. The polygons are generated through the use of a dual-grid [31]. We discuss necessary modifications of the dual-grid method to polygonise the surfaces. The main advantage of the dual-grid is that cracks do not form between shared plotting node faces that

are at different depths. Polygons are formed as a mixture of triangles and quadrilaterals and the rendering can be hardware accelerated. We also discuss improvements to the grid so that slither triangle do not form.

We also discuss the implementation on the GPU. In the GPU approach rather than adaptively subdividing using interval and curvature information, only intervals are used to subdivide the surface to a fixed polygon plotting depth. This reduces divergence in the code path executing on the GPU for improved performance. As all leaf nodes are divided to the same depth, no cracks can form, and so neighbour information is not required. At the plot depth we then use the approach given in Bloomenthal [24] to try to polygonise the node. If the node fails to polygonise we then use a point-based approach to render the surface in the node.

Our point splatting approach uses the vertices and points obtained from the polygon/points algorithm and uses a Gaussian elliptical weighted average filter to render shaded discs (surface splatting) at these points. We also implement this algorithm on the GPU. Finally we compare GPU ray-casting with these approaches.

Our objective is to be able to render implicit surfaces at high frame rates so that exploration of families of surfaces is possible in real-time. We achieve high frame rates for low order implicit surfaces with the polygons/points algorithm and with point splatting. Even for high order surfaces we achieve high frame rates, in many cases approaching that of ray-casting for single images. We compare timing for rendering a set of surfaces using

our mixed polygon/point methods, using surfel splatting, and for ray-casting.

When animating images our method has advantages over ray-casting. We store the triangle and point geometry on the GPU so we can rotate surfaces in real-time. Our approach also allows us to modify the parameters of various interesting families of functions in real-time, and see the effects on the shape of the surface. We illustrate the techniques with a number of example surfaces.

The major contribution of this chapter is a new hybrid algorithm that improves the rendering of non-manifold implicit surfaces by using mixed polygon/point sampling. The algorithm uses the dual-grid method to avoid the "crack" problem that occurs when using polygons derived from leaf node that share faces at different plot depths, see Bloomenthal [24] and Balsys and Suffern [4] for a discussion of this problem. The algorithm is fast enough that high quality images of surfaces can be obtained at interactive frame rates on PCs. We demonstrate the use of our algorithm by modifying the parameters of functions defining implicit surfaces and rendering the effect on the surface in real-time.

In this chapter we show how polygons and points can be combined in a single rendering of complex visualisation problems, including visualising discontinuous or singular regions such as ridges, cusps, self-intersections, singular-lines and thin-sections of surfaces. We also compare our hybrid algorithm with the use of point splatting and ray-casting approaches.

We propose a visualisation model that uses a combination of triangles, quadrilaterals

and point primitives to represent surfaces. These primitives are all rendered to the same colour buffers for seamless visualisation between manifold and non-manifold sections. While quadrilaterals can be converted to triangles, we choose to preserve the quadrilateral sections that are inherit in the dual-grid meshing. Mesh output is comprised mostly of quadrilaterals, and keeping the data as quadrilaterals reduces the memory footprint of the model. Memory use is a concern for more complex surfaces where triangles/quadrilaterals must be smaller to accurately bound the surface.

5.1 Dual-grid meshing

We initially investigated an algorithm that could generate crack-free meshes, rather than algorithms that generate meshes with cracks, and later attempt to fill the cracks. At the heart of our algorithm lies the dual-grid octree structure from [31]. Rather than use marching cubes to generate polygons we generate polygons from the dual-grid lines. The vertices of the dual-grid are positioned at the centroid of the surface intersections in the node. This method forms the crack-free mesh of an implicitly defined surface. The mesh is supplemented by the point-based rendering process used in the previous chapters. Points are used to visualise nodes after the octree depth is above a user defined limit. The method proceeds in a number of stages.

(i) We first subdivide using octree spatial subdivision and, if the planarity and divergence criteria used in Bloomenthal [24] are passed, the subdivision process terminates;

(ii) If we reach the maximum plot depth and the surface is still not flat enough we proceed using the point-based method described in chapter four;

(iii) The mesh is then created from the dual-grid nodes that are flat enough to polygonise;

(iv) The resulting quadrilaterals, triangles and points are loaded onto the graphics card into geometry buffers for efficient rendering;

(v) Points are found in dual-grid nodes that are not flat enough to polygonise.

We now discuss how the polygon mesh is created. In Bloomenthal [24] and Balsys *et al.* [79] they stitch together polygons using the intersection points, however this results in cracks that form between node that share faces at different polygonisation depths, for example see Figure 5.1. The cracks must then be identified with additional triangles added to create a closed mesh. Our solution is to avoid this problem altogether, by using a dual octree approach as in Schaefer and Warren [31]. The use of an octree in our algorithm allows us to find nodes that share a face. As a result we avoid the problem of sliver triangles as discussed in Ju [32], and Schaefer and Warren [31].

Instead of applying marching cubes to the dual octree as in Ohtake *et al.* [30], and Paiva *et al.* [33], we stitch our mesh together based upon shared face intersections between octree nodes. In $findNodeIntersects()$ the roots of the surface function are found along the node edges using the method of *regula falsi* and these define a polygon whose centroid is calculated by averaging the node intersection points (we do not construct the polygon to do this). We use the center of the polygon rather than the center of the plot node as this

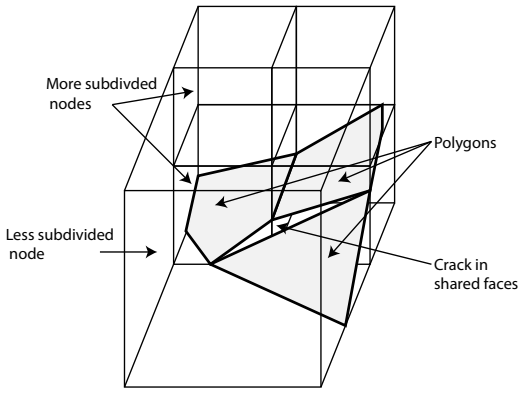


Figure 5.1: Polygonising plot nodes using polygon outlines in adaptively subdivided octree results in cracks occurring that are aligned with the shared faces between nodes subdivided to different plot depths (from Balsys and Suffern [4]).

has the benefit of producing a more optimised surface mesh, particularly when only small polygons are contained in the plotting node. These centroids along with other node data are pushed into a list of plot nodes for later use in the rendering pass.

In the rendering algorithm the centroid is used to define the dual-grid vertices. This makes our dual-grid vertices lie on, or very close to the surface, unlike the work in Ohtake [30]. Their dual-grid vertices begin at arbitrary locations, and are iteratively repositioned through an equation. This equation attempts to find prominent surface features. Alternatively Schaefer *et al.* [31] solves a quadratic error function (QEF) to find the most prominent feature in the node, by minimising quadratic energy.

In Schaefer and Warren [31] the dual-grid lines define the marching cube edges. This can be visualised in Figure 5.2 by tracing cubes from the red outlines. The dual-grid is not

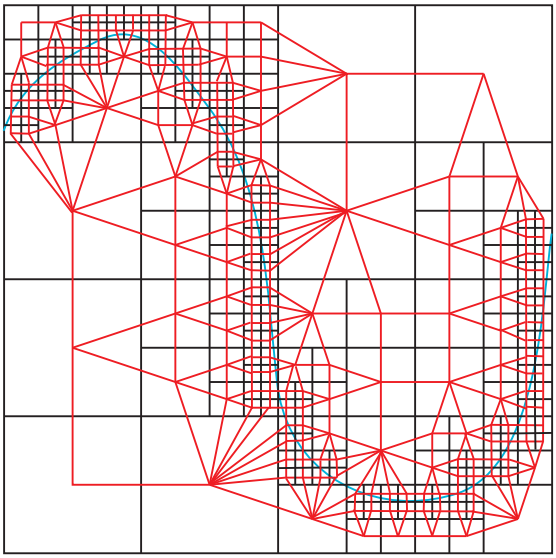


Figure 5.2: Quadtree adaptive subdivision of a 2D curve. The dual-grid is the lines connecting adjacent nodes in the octree. Octree outlines in black, dual-grid outlines in red, and surface in blue.

rectangular and axis-aligned but can be treated as such for the purposes of the "marching cubes" algorithm of Lorensen and Cline [23]. They use a marching cube algorithm in their work for various purposes. We do not use a marching cube algorithm to render surfaces. We proceed to stitch our polygons together using the centroids of polygons in the polygon plotting nodes to form a dual-grid.

According to Ohtake and Schaefer [28] the edges in the dual-grid are guaranteed to form a mesh composed of a mixture of quadrilaterals and triangles (see Figure 5.2 where this is demonstrated for a quadtree). To create the dual-grid mesh of our surface we proceed as follows. We start by looping through all the plotting nodes found in the

CreatePlotNode() method in our algorithm. We store a pointer to the current node and assign the plot node centroid as the dual-grid vertex for the current node. Each time a plot node is found in *CreatePlotNode*() we add to the node data, a list of pointers to faces that also contain plot nodes, and we then update the adjacent node lists with the new plot node pointer. We use these lists to form the dual-grid.

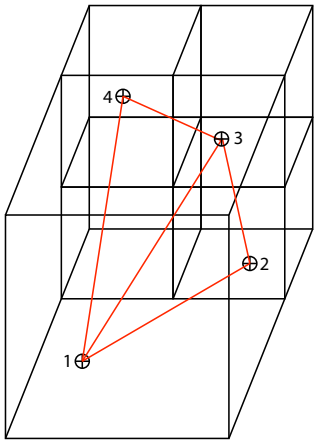


Figure 5.3: Forming triangles using a dual-grid. Plotting node centroids labelled *one*, *two*, *three*, and *four*. Dual mesh edges in red.

Either quadrilaterals or triangles will form using the dual-grid. We start by considering our plot node polygon centroid (centroid *one*) and a neighbour with another vertex in it on a shared face (centroid *two*). We then compare face neighbours for centroid *one* and centroid *two* to see if they share a face neighbour (centroid *three* in Figure 5.3). If there is a shared neighbour then a triangle is formed as in Figure 5.3 between centroid *one*, *two* and *three*. An edge can have no more than two connected primitives. Three or more connected primitives is the result of a non-manifold section (these are the cause of sliver

triangles), which will never occur in our dual-grid as we reject these nodes from being polygonised. Rejected nodes are subdivided using the point method from chapter four, with the resultant points stored into a list. The point list is rendered in conjunction with the triangle mesh.

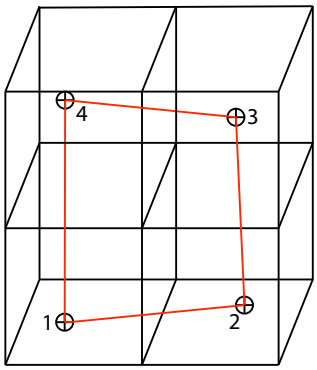


Figure 5.4: Forming quadrilaterals using a dual-grid. Plotting node centroids labelled *one*, *two*, *three*, and *four*. Dual mesh edges in red.

Forming quadrilaterals is illustrated in Figure 5.4. Consider the case where we are starting with centroid *one* and have found centroid *two* in its face neighbour list. We find that centroid *three* is a face neighbour of centroid *two* but does not share a face with centroid *one*. However centroid *three* has a face neighbour (centroid *four*) which is also a face neighbour of centroid *one*. This means that a quadrilateral should be used to connect the four centroids in the dual-grid.

However finding triangles and quadrilaterals is problematic as loops form when cycling through the neighbour nodes neighbour lists. To avoid cycling through the same edges we use a hashing scheme. Each unique key is a structure formed from the three or

four node indices sorted in ascending order. From this structure we generate a hash value to represent the edges of the triangles and quadrilaterals, and no edges are inserted in the dual-grid when a duplicate hash is found. Thus we guarantee not to produce duplicate primitives in our mesh.

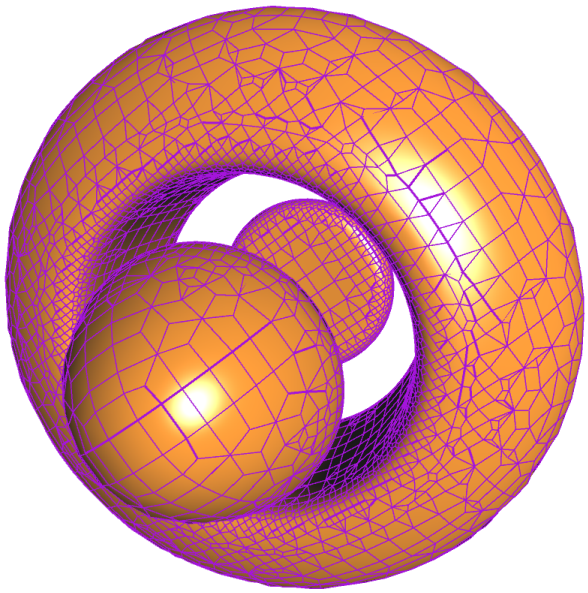


Figure 5.5: Surface rendered using the described dual-grid based meshing.

The mesh algorithm was unsuitable for implementing on the GPU so we devised algorithms for rendering polygon/point and point splatting on the GPU.

5.2 GPU subdivision algorithm

In this work we show how polygons and points can be combined in a single rendering of complex visualisation problems on the GPU, including visualising discontinuous or singular regions such as ridges, cusps, self-intersections, singular-lines and thin-sections

of surfaces. At the heart of our algorithm lies the octree node subdivision process as in [25]. The resultant mesh is supplemented by the point-based rendering process used in the previous chapters.

Initially we tried to modify the adaptive subdivision approach of Bloomenthal [24], Schmidt [27] and Balsys and Suffern [4] for use on the GPU. These approaches have in common the idea of using curvature to drive the subdivision process, stopping subdivision in regions of relatively low curvature. This results in cracks in the surface mesh that then have to be fixed, as shown in Figure 5.1. When implemented on the GPU this results in thread stalls as threads wait for neighbour nodes to be polygonised so that the cracks can be fixed. The timings from initial trials were not promising, and this approach was dropped in favour of those now described.

In our approach rather than adaptively subdividing using interval and curvature information, only intervals are used to subdivide the surface to a fixed polygon plotting depth. As all leaf nodes are divided to the same depth, no cracks can form, and so neighbour information is not required. At the plot depth we then use the approach given in Bloomenthal [24] to try to polygonise the node. If the node fails to polygonise we then use a point-based approach to render the surface in the node.

The algorithm proceeds in three distinct phases. The first phase is shown in Figure 5.6. The subdivision process begins with the view volume (defined as an axis aligned cube of 3D space). This root node is pushed into a vector of nodes to be subdivided. The

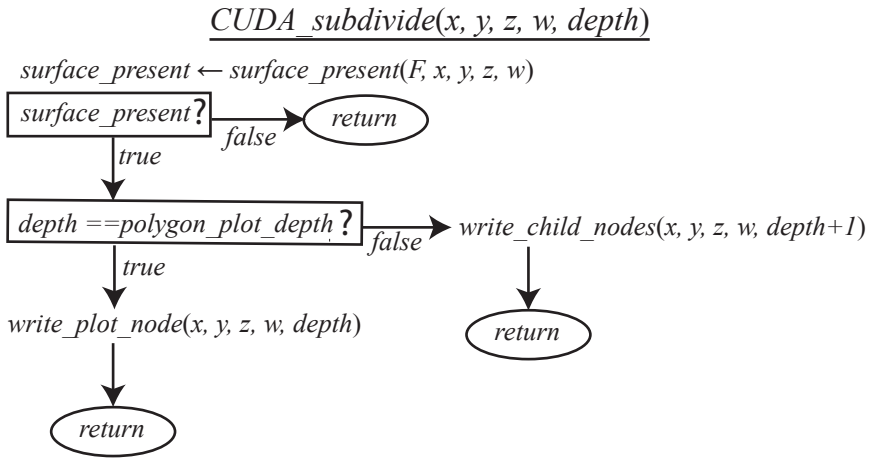


Figure 5.6: Phase 1: *CUDA_subdivide*().

centre coordinate (x,y,z) and width, w are defined by the user for the view volume. The initial $depth$ is one. The first NVIDIA *CUDA* thread takes these values and tests if the natural interval extension of the surface, $F(x,y,z)$, in the node contains 0.0. To do this the intervals $X = (x-\frac{w}{2}, x+\frac{w}{2}), Y = (y-\frac{w}{2}, y+\frac{w}{2}), Z = (z-\frac{w}{2}, z+\frac{w}{2})$ are calculated to evaluate $F(X,Y,Z)$ in $surface\_present()$.

$surface\_present()$ does not guarantee that the surface exists in any plotting node, it does, however, guarantee that any node of the octree that is discarded does not contain the surface. If the surface is present the node is subdivided into 8 child nodes using the $write\_child\_nodes()$ function. This calculates the new centres and width of the child nodes and atomically gets an index used to write to a global node vector. Incrementing the index is an atomic operation to ensure no race condition occurs between the hundreds of threads that may be running.

CUDA_subdivide() is run in parallel threads that atomically get indexes of nodes from

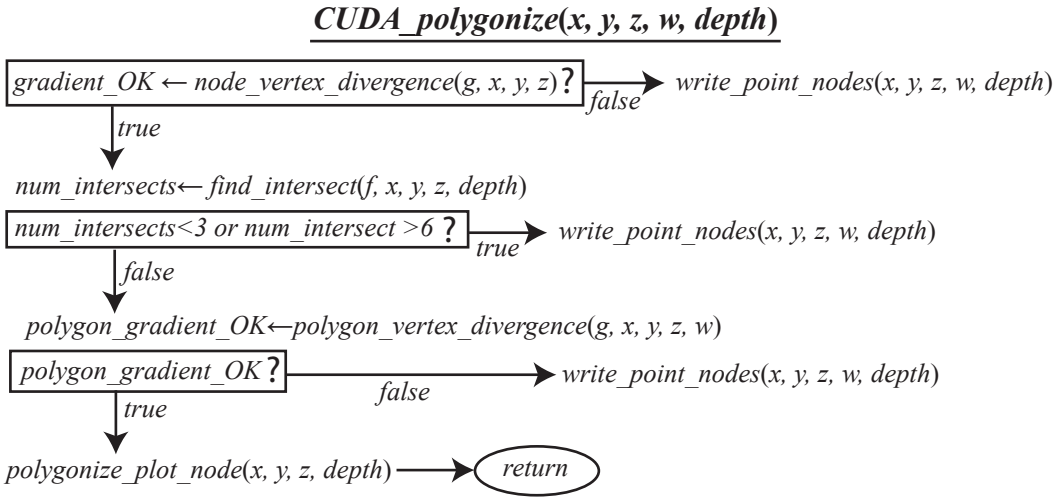


Figure 5.7: Phase 2: *CUDA_polygonise*().

a global node vector, and uses the node data to evaluate *CUDA_subdivide*(). When a node is reached that is at the polygonisation depth, *polygon_plot_depth*, the thread terminates by atomically getting an index to a global polygon plot node vector and uses this to write the polygon plot node data used in phase 2 of the algorithm. Phase 1 produces a list of polygon plot nodes at the polygon plot depth.

The nodes in the polygon node vector are popped by parallel threads running the *CUDA_polygonize*() algorithm in Figure 5.7. We estimate the curvature of the function in the node by evaluating the function divergence at node vertices. In the past there has been some exploration of using estimates of curvature to drive the subdivision process. Bloomenthal [24] used planarity, divergence and chord-distance estimates for the curvature of the surface. In Balsys and Suffern [4] it was found that divergence by itself can be used as a curvature estimate, and so we use this approach in this work. Estimating

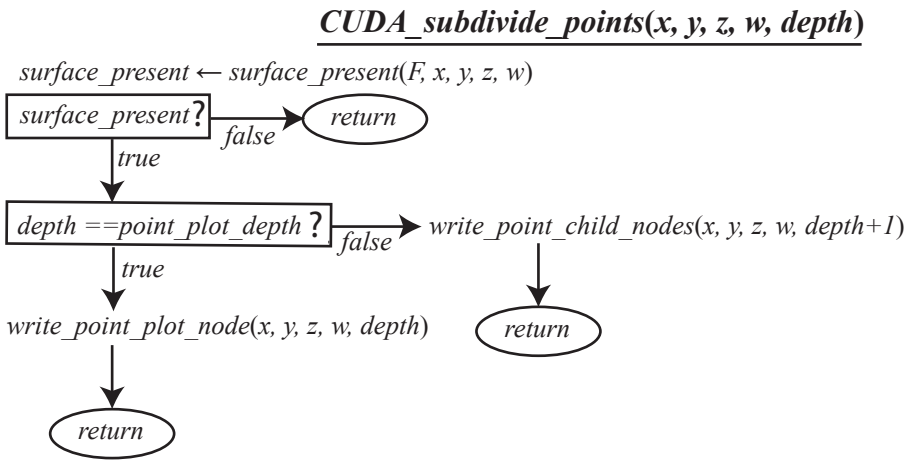


Figure 5.8: Phase 3: *CUDA_subdivide_points*().

curvature at the node edges means we do not have to find the surface intersections with the node edges, computationally expensive, unless the function passes the divergence test. If the function fails the node divergence test the node information is atomically added to a vector of nodes to be used in the point subdivision phase.

In phase 2 the subdivision proceeds using an interval test until the fixed polygon plot depth is reached. We then evaluate the dot product between all surface normals calculated at node vertices. If these dot products indicates that the maximum deviation between any pair of node vertices is greater than the user-settable divergence criteria (30°) then the surface is not flat enough to polygonise. The depth, width of the node, and the centre of the node, are written to a global vector variable using an atomically accessed index. This information is used in the point subdivision phase to uniquely define the node to be subdivided using points.

The polygonisation algorithm considers all 12 plot node edges in turn, and uses a

function sign test to detect an edge that contains surface intersections. To locate the root along the edge in $find\_intersects()$, we use the method of *regula falsi*. If three to six intersects are found along plot node edges we do a curvature test using the divergence between the surface normals at the roots on the edges (divergence $< 30^{\circ}$). If this passes then we have found a (relatively flat) polygon in the node. We then atomically get the value of an index to use to write the required data into a global vector for use in rendering the polygon. Only if the plot node has three to six intersects, and meets or exceeds the divergence criteria, do we generate a polygon in the node. Otherwise we write the node data into a vector to use in the point subdivision stage.

The $CUDA\_subdivide\_points()$ algorithm is very similar in action to the $CUDA\_subdivide()$ algorithm. Multiple *CUDA* threads are launched running this code. The data comes from the point node vector created in phase 2. The initial data in this vector are all the nodes at the polygon plotting depth that failed to polygonise. Each of these will add global point nodes, if the surface is present, until the point plotting depth is reached. At this depth the thread will get an atomic index to add the node data into a global point plotting vector for use in the pruning stage.

Details of pruning algorithm we use are given in chapter four as $point\_sampling\_fails()$. This algorithm is applied to each of the nodes found at the point plot depth of $CUDA\_subdivide\_points()$ to prune the excess points found using interval subdivision as described in chapter four. At the final stage we have a set of nodes to be rendered using polygons and a set of nodes to be rendered using points.

The polygons / points algorithm renders the polygons/points using OpenGL with a depth buffer and Phong [7] shading. The normal for Phong shading is calculated using the surface's gradient function.

5.3 Rendering Non-Manifold Surface Sections

The following figures illustrate the results obtained when this approach is used to render a non-manifold surface. Figure 5.9 shows the case for a surface with singular points, such as the Cassini-Singh surface f[17] ;

$$f(x,y,z) = ((x+a)^2+y^2)((x-a)^2+y^2)-z^2 \tag{5.1}$$

with $a = 1$. The surface gradient function ∇f is calculated for each vertex or point on the surface. By interpolating the per-vertex normals over our triangle using Phong shading, we produce per-pixel surface lighting with quality that approaches point-based and ray-casted methods. We then make use of the Phong lighting model to render all sampled points. This allows us to produce images whose quality is similar to ray-cast images.

In Figure 5.10 we show how rays are rendered by the method described in this work. Figure 5.10 (a) shows the cross cap surface f[18] ;

$$f(x,y,z) = 4x^2(x^2+y^2+z^3)+y^2(y^2+z^2-1) \tag{5.2}$$

rendered using polygons only. Polygonal methods are not well suited to rendering non-

manifold surfaces such as the cross cap surface, particularly where the distance between the front and back of the surface approaches zero. In Figure 5.10 (b) we show the result when polygons and points are used to render the surface. The image is of much improved quality, producing results similar to that produced when the image is ray-cast.

5.4 Comparison of Animations using; Points/Polygons, Surface Splatting, and Ray-casting

In recent years there has been much development of rendering based on a technique called surface splatting. Generally this uses a Gaussian elliptical weighted average (EWA) rendering technique for rendering point sampled geometry, see for example [44], [45, 48], [47], [80]. The splat is an extension of the point primitive which faces a direction defined by a normal vector and can be represented by various shapes. The texture applied to the shape is generally a Gaussian EWA function that blends multiple splats more effectively than a simple linear blend.

The output from algorithm 1 is either triangles and quadrilaterals, with associated normals, or points with their associated surface normal. As a result we can use each of the vertices of the polygons as the origin of a splat. The radius of a splat can be controlled so that all the splats overlap to create a surface without holes. This approach has the advantage that we do not have to stitch the polygons together, we only need to find the polygon vertexes, and this saves time. Also the use of the EWA filter helps to preserve the

surface shading. Due to the blending, silhouette edges can appear softer than if rendered with points or polygons.

We used splat sizes based on the area occupied by the triangle or quadrilateral in the leaf node. This results in an adaptive splatting scheme that can handle non-manifold sections, the splat can be positioned on the original intersection polygon, or on the dual octree nodes. We find the dual octree provides better splatting due to the crack free mesh topology.

An alternate to our use of polygons/points or the use of point splatting is to use interval ray-casting, as first described in Mitchell [58], and developed on the GPU by Loop and Blinn [81], Knoll *et al.* [53], and Singh and Narayanan [52]. The advantages of the ray-cast GPU algorithms are that they can render many surfaces in real-time.

Given the diversity of approaches we decided to do a comparison of the methods so that we could determine the relative advantages/disadvantages of the methods. We implemented our polygons/points algorithm on the GPU. The scheme to balance the use of threads in implementing the polygons/points algorithm was similar to that used in chapter four.

Our ultimate goal is to be able to interact with the surfaces in real-time. For this reason the following points are important to the comparisons;

- the images should be as accurate as possible a representation of the surface shape,
- the frames should be produced at interactive frame rates,

- there should be little or no rendering artifacts between successive frames of the images.

The first two of these points have been discussed in the previous section. To test frame rates of the approaches we rendered a number of implicit test surfaces of varying complexity and type, and measured the time to render a high quality image of a single frame. By a high quality image we mean that the surface boundaries and features are well defined, and the Phong shading is smooth across the surface.

To compare the timings we had to adjust the parameters for the various methods so that images of comparable quality were produced by each method. For the polygon/point method and the splatting approach the only control is the plot depth. For interval ray-casting the final width of the intervals are specified as an ε value, as defined in Singh *et al.* [52]. The resulting timing comparisons are given in Table 5.1.

From these timings we see that the polygon/points algorithm can be slower (15%) or faster (250%) than the timing for ray-casting with intervals. Point splatting is faster for the Mitre surface but otherwise is slower that using mixed polygons / points. As discussed before, the resultant geometry from the point/polygon approach is preferred. Hence these timings are very encouraging, indicating we are achieving frames rates at comparable speed to ray-casting for rendering single high quality images.

Figure 5.12 shows the Mitre surface f[5]

$$f(x,y,z) = 4x^2(x^2+y^2+z^3)+y^2(y^2+z^2-1) \tag{5.3}$$

Table 5.1: Timings (sec) for rendering high resolution images of implicit surfaces using ray-casting, points/polygons and point splatting. Image size 1000x1000 pixels.

Surface	Ray-casting	Polys./Points.	Splatting
Boys	1.697	1.302 (76%)	1.198 (71%)
Cyclide	0.198	0.299 (151%)	0.949 (479%)
Steiners surface	0.179	0.245 (73%)	0.193 (108%)
Mitre	0.112	0.280 (250%)	0.133 (119%)
Bohemian Star	8.837	1.339 (15%)	4.948 (56%)
Barth Sextic	0.851	2.164 (250%)	2.163 (254%)
Chumtov 8	10.913	5.569 (51%)	5.635 (52%)
Chumtov 14	17.5	7.212 (41%)	7.031 (40%)

The polygon/points method fills these gaps with points and will render singular lines as lines, and is anti-aliased around silhouette edges. We anti-alias the edges in the polygons/points method using the process described in chapter two. The point splatting approach renders the singular lines as thin tubes. Also there are aliasing problems around silhouette edges. The ray-casting methods still has aliasing problems along the singular line and suffers aliasing along silhouette edges. Edge anti-aliasing of the ray-cast surface is not discussed in the work of Loop and Blinn [81], Singh and Narayanan [52] and Knoll *et al.* [53]. Anti-aliasing complicates the GPU ray-casting algorithm as many more rays are fired at silhouette pixels so a colour average for the silhouette pixel can be found. This further slows the ray-cast algorithm.

Boy's surface f[1] ,

$$\begin{aligned} f(x,y,z) = 64(1-z)^3z^3 - 48(1-z^2)z^2(3x^2+3y^2+2z^2) \\ +12(1-z)z(27(x^2+y^2)^2 - 24z^2(x^2+y^2) \\ +36\sqrt{2}yz(y^2-3x^2)+4z^2) \\ +(9x^2+9y^2-2z^2)(-81(x^2+y^2)^2 \\ -72z^2(x^2+y^2)+108\sqrt{2}xz(x^2-3y^2)+4z^2), \end{aligned} \tag{5.4}$$

is a hard surface to visualise from static images. With our algorithm we can rotate the surface in real-time so the nature of the surface becomes more apparent. In Figure 5.13 we show individual frames from an animation that flies the camera around Boy's surface looking at the origin of Boy's surface. Figure 5.13 (a) shows the image obtained from the

points / polygon approach, Figure 5.13 (b) using point splatting using the vertex, point and normal information from the points/polygons method and Figure 5.13 (c) shows Boy's surface rendered using the GPU approach of Knoll *et al.* [53]. Boy's surface consists of three blade like structure that intersect one another. The point polygon algorithm does a good job of rendering the non-manifold portions of this image. Animations of Boy's surface using these three methods can be viewed on DVD Chapter 4.

It can be hard for people to see 3D shapes from 2D images. For this reason stereoscopic viewing systems are used when appreciation of visual depth is important to the problem. Stereoscopic viewing systems use two camera positions, usually separated by 2°-4° at the point of interest, to produce left and right eye views of the scene. As one of our interests is in visualisation of 3D shape we also produced stereo pairs using the three approaches and compared the timings. Our points polygons approach is well suited to producing such stereo pairs.

Figure 5.14 uses points/polygons to produce stereo pairs of Steiner's Roman surface f[2]

$$f(x,y,z) = x^2y^2 + y^2z^2 + x^2z^2 - xyz = 0 \tag{5.5}$$

Table 5.2 gives timings for creating a one second animation of flying a camera in a circle around Steiner's Roman surface and producing sequences of stereo pairs using the three approaches. Example animations can be viewed at DVD Chapter 1. The polygon/points model is faster, as ray-casting has to recalculate the geometry for each frame. We have

also showed the effect on timing of anti-aliasing using the polygons/points method. With up to sixteen extra samples per pixel rendering time is still less than no anti-aliasing using interval ray-casting.

Table 5.2: Timing for rendering stereo pairs for 25 frames of Steiner's Roman surface $f[2]$ using points/polygons and ray-casting. Image size 1200x600 pixels.

Method	Time (sec)	Depth/ε
Points/Polygons (no AA)	1.4	11
Points/Polygons (4xAA)	2.6	11
Points/Polygons (16xAA)	5.3	11
Interval ray-casting	7.8	10

Finally we show frames from an animation produced by flying a camera along the z axis of the Klein bottle surface $f[3]$

$$\begin{aligned} f(x,y,z) &= (x^2+y^2+z^2+2y-1)[(x^2+y^2+z^2-2y-1)^2-8z^2] \\ &+ 16xz(x^2+y^2+z^2-2y-1)=0, \end{aligned} \tag{5.6}$$

in Figure 5.15. The full animation can be viewed at DVD Chapter 2. Such animations help in visualising the internal structure of self intersecting surfaces, such as the Klein bottle surface.

5.5 Visualisation of families of surfaces

The previous visualisations are of single implicit surfaces. Another kind of animation involves holding the camera in a fixed position, but varying the surface geometry of parameterised surfaces. The first example is of the cyclide surface

$$f(x,y,z) = (x^2+y^2+z^2)^2 - 2(x^2+r^2)(f^2+a^2) \\ -2(y^2-z^2)(a^2-f^2) + 8afrx + (a^2-f^2)^2 \tag{5.7}$$

where a, r and f are the parameters. Three forms of the cyclide surface occur depending on the scale of the parameters. When $r <= a < f$ the cyclide has two horns as shown in Figure 5.16 (a), (b) and (c). When $a < r < f$ the cyclide has one horn that meets at the centre, as in Figure 5.16 (d), and when $a < f <= r$ the cyclide has an interior surface as in Figure 5.16 (e) and (f). The Figures are cut by the root octree to show the interior sections. We have produced an animation of this, at DVD Chapter 3, where the value of the parameter r is initially less than a and f and then is incremented so $a < r < f$, and eventually r is greater than a and f.

Barr [75] first illustrated the super-quadratic surfaces. The families of the super-quadratics include the super-ellipsoids, super-toroids, and the super-hyperboloids. For each family two parameters, ε_1 and ε_2 control the squareness and roundness of the resulting surfaces. The super-ellipsoid surfaces f[22] are given by

$$f(x,y,z) = \left((\frac{x}{a})^{\frac{2}{\varepsilon_1}} + (\frac{y}{b})^{\frac{2}{\varepsilon_1}} \right)^{\frac{\varepsilon_2}{\varepsilon_1}} + (\frac{z}{c})^{\frac{2}{\varepsilon_1}} \tag{5.8}$$

Figure 5.17 illustrates the surfaces resulting for values of the parameters ε_1 fixed at 1.0 and the value of ε_2 varies from 0.04 to 3.6 in steps of 0.4. The parameters a, b and c control the height, width and length of the surface. These are fixed in the figure. Animations showing the result of varying ε_1 and ε_2 for the super-quadratic surfaces may be viewed at DVD Chapter 5.

We conclude this section by illustrating an interesting family of surfaces that are difficult to render, especially in real-time. The Gaussian and mean curvatures are intrinsic properties of implicit surfaces [8]. The Gaussian curvature $K(x,y,z)$ of an implicit surface $f(x,y,z) = 0$ is a 3D scalar field. A specific value of this scalar field $K(x,y,z) = k$, with $k \in \Re$ is another implicit surface. We call this the curvature surface of $f(x,y,z) = 0$. If $k \in [kmin, kmax]$ for $f(x,y,z) = c$, these two surfaces will intersect.

To illustrate this we show a sequence of Gaussian curvature surfaces of a surface. The surface we choose is Steiner's surface. We first illustrate this in Figure 5.18 which shows three Gaussian curvature surfaces superimposed on the original surface. To aid the visualisation we colour Steiner's surface differently from its curvature surfaces.

While we cannot render an animation of families of curvature surfaces, such as this, in real-time, our polygon/point method is fast enough that the frames can be created off-line and assembled into an animation in a matter of hours. The result for the curvature surfaces of Steiner's surface can be found at DVD Chapter 6.

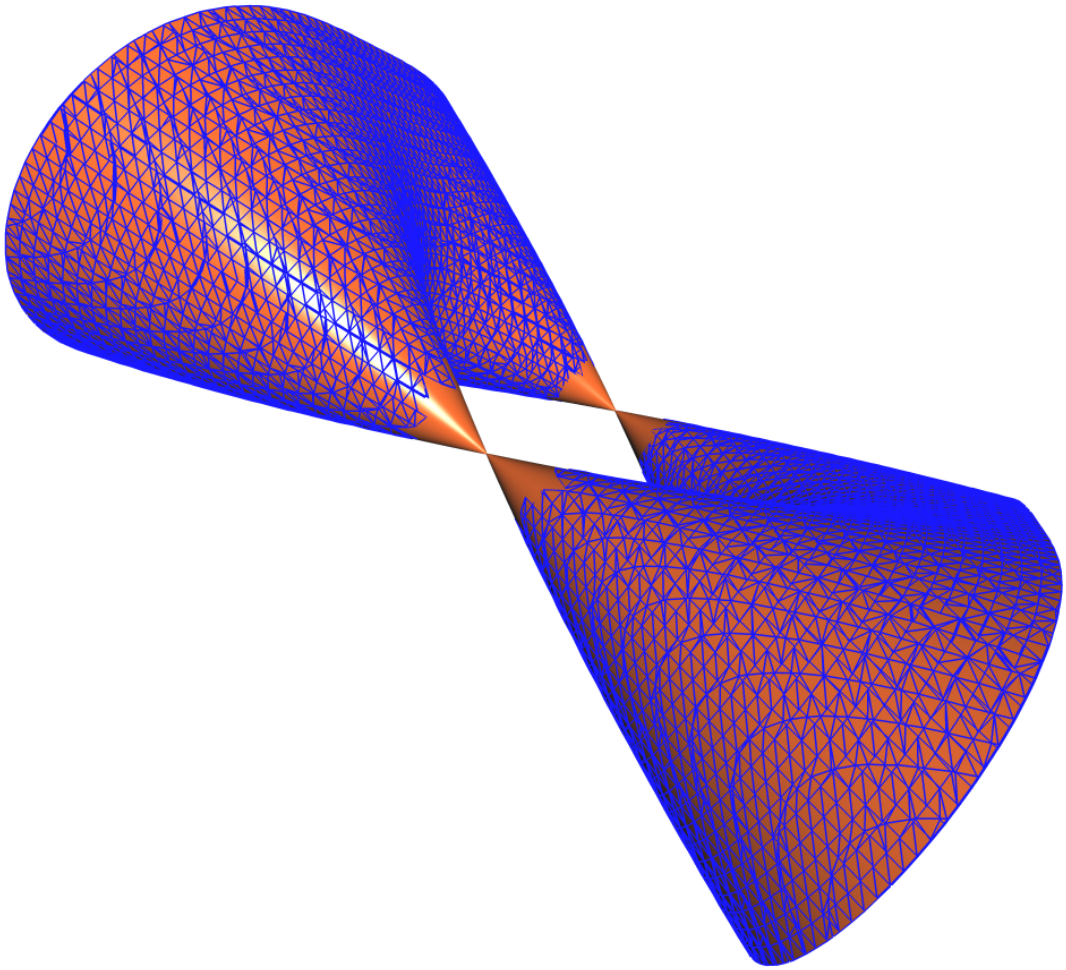
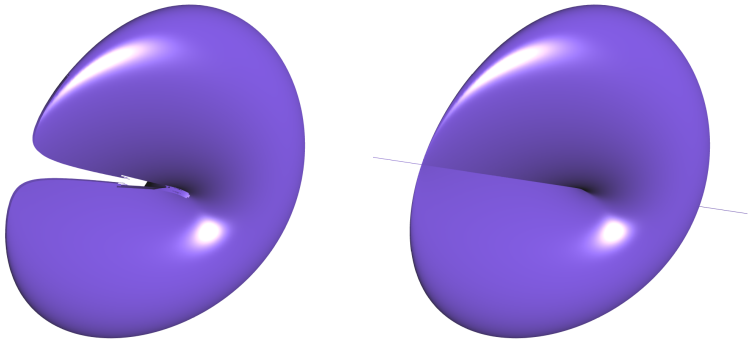
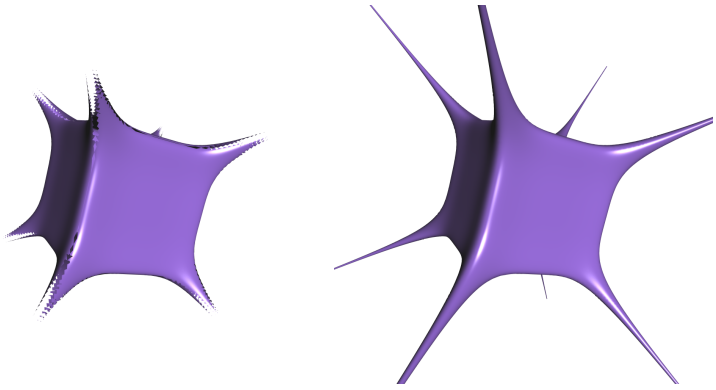


Figure 5.9: The Cassini-Singh surface $f[17]$ showing GPU rendering. Polygon sections are outlined in blue while the middle section is filled with points.



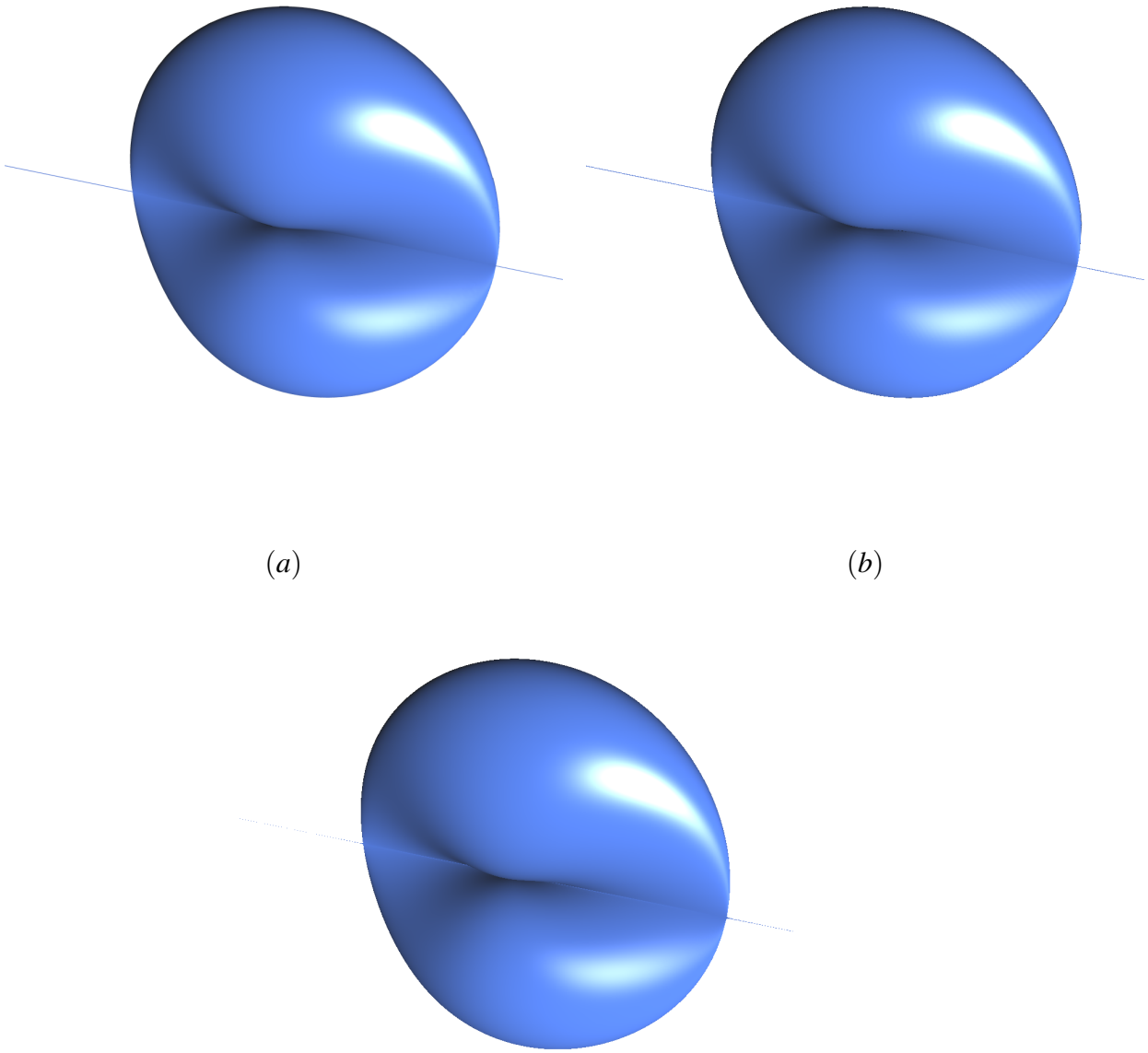
(a) (b)

Figure 5.10: Crosscap surface $f[18]$ rendered with GPU algorithm. (a) Polygons only, and (b) Polygons and points.



(a) (b)

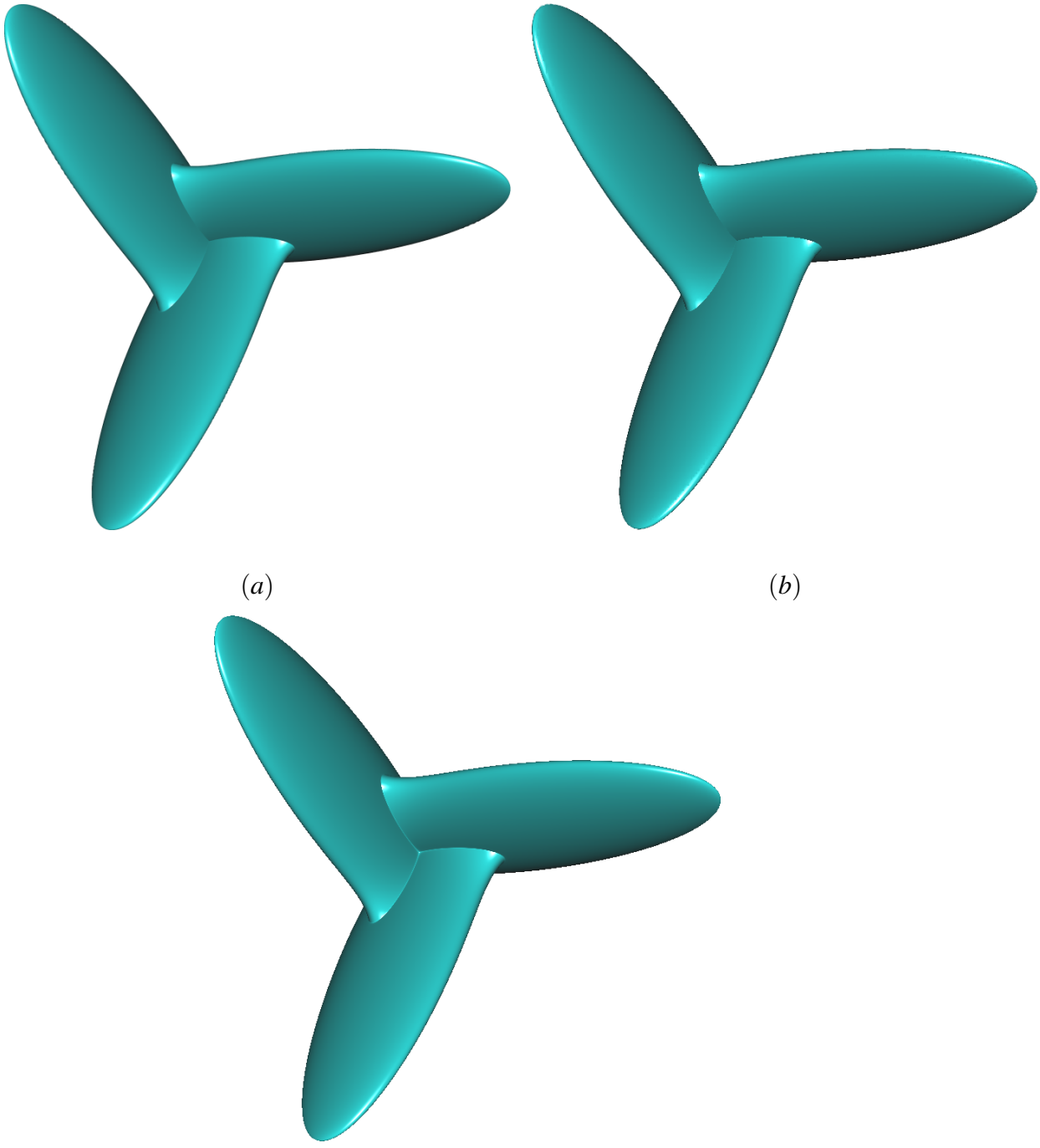
Figure 5.11: Spiky surface $f[8]$ rendered with GPU algorithm. (a) Polygons only, and (b) Polygons and points.



(a) (b)

(c)

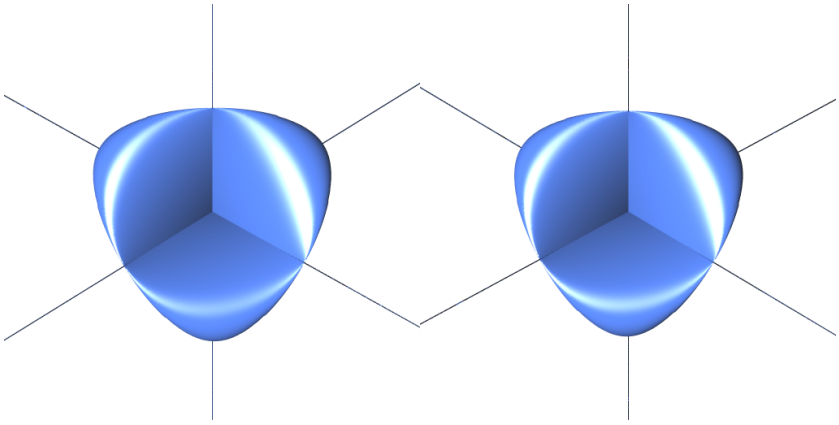
Figure 5.12: The Mitre surface $f[5]$ rendered with, (a) Polygons / points with point depth of 12, (b) Point splatting with a maximum point depth of 12, and (c) Using interval ray-casting with an ε value of 2^{-11}.



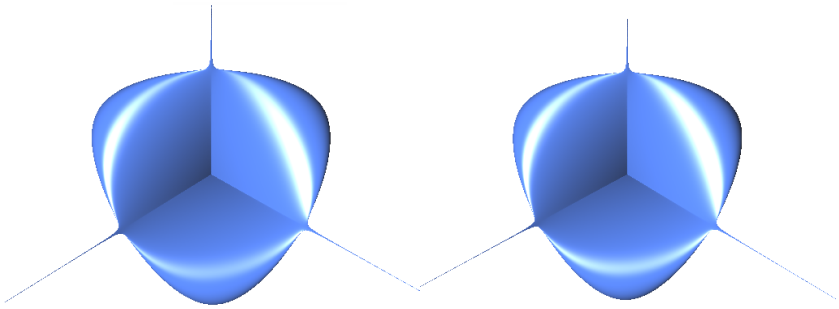
(a) (b)

(c)

Figure 5.13: Boy's surface $f[1]$ rendered with GPU algorithm. (a) Polygons / points with point depth of 12, (b) Point splatting with a maximum point depth of 12, and (c) Using interval ray-casting with an ε value of 2^{-16}.



(*a*)



(*b*)

Figure 5.14: Steiner's Roman surface $f[2]$ rendered using, (a) Polygons / points with point depth of 12, and (b) Interval ray-casting with an ε value of 2^{-12}.

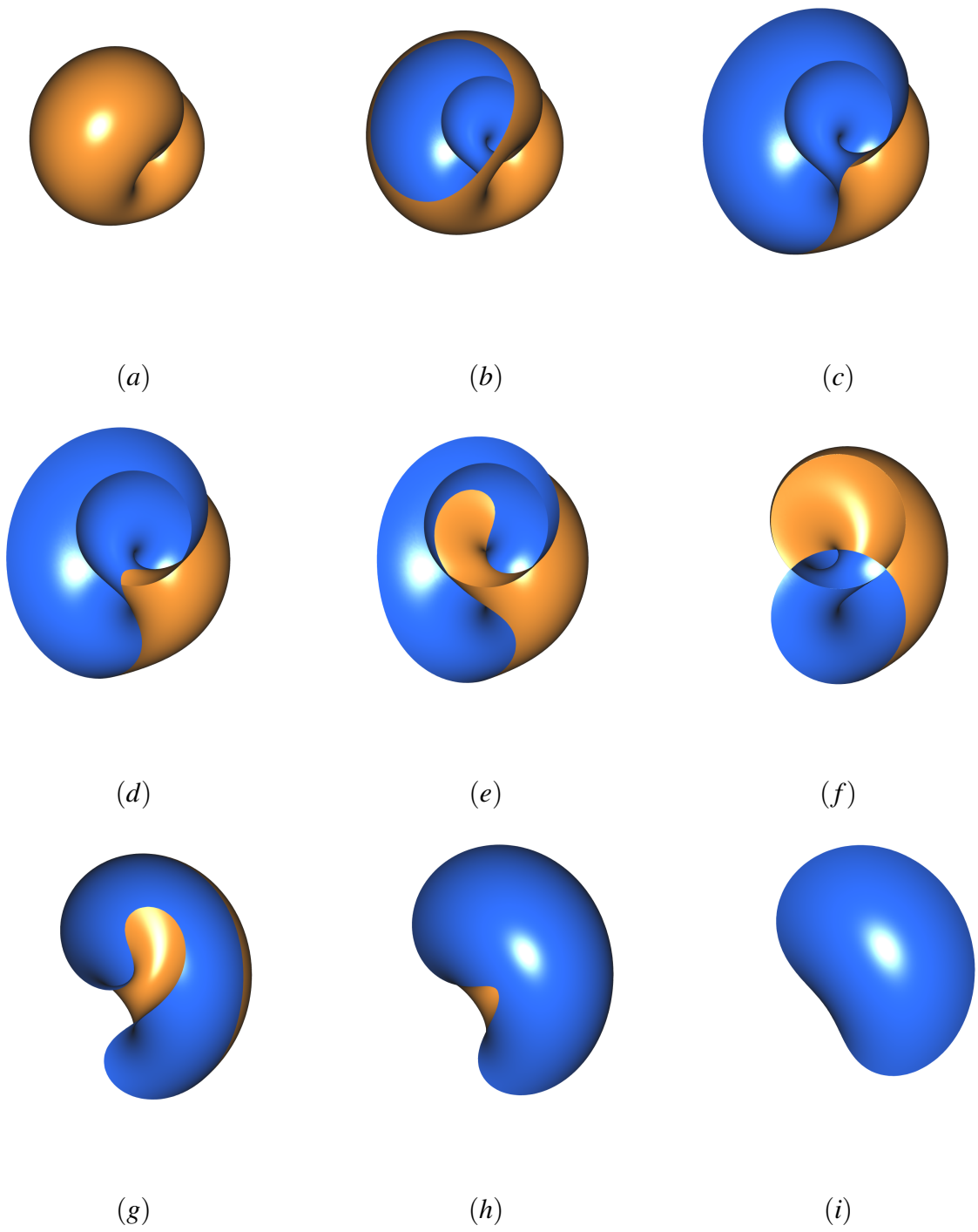
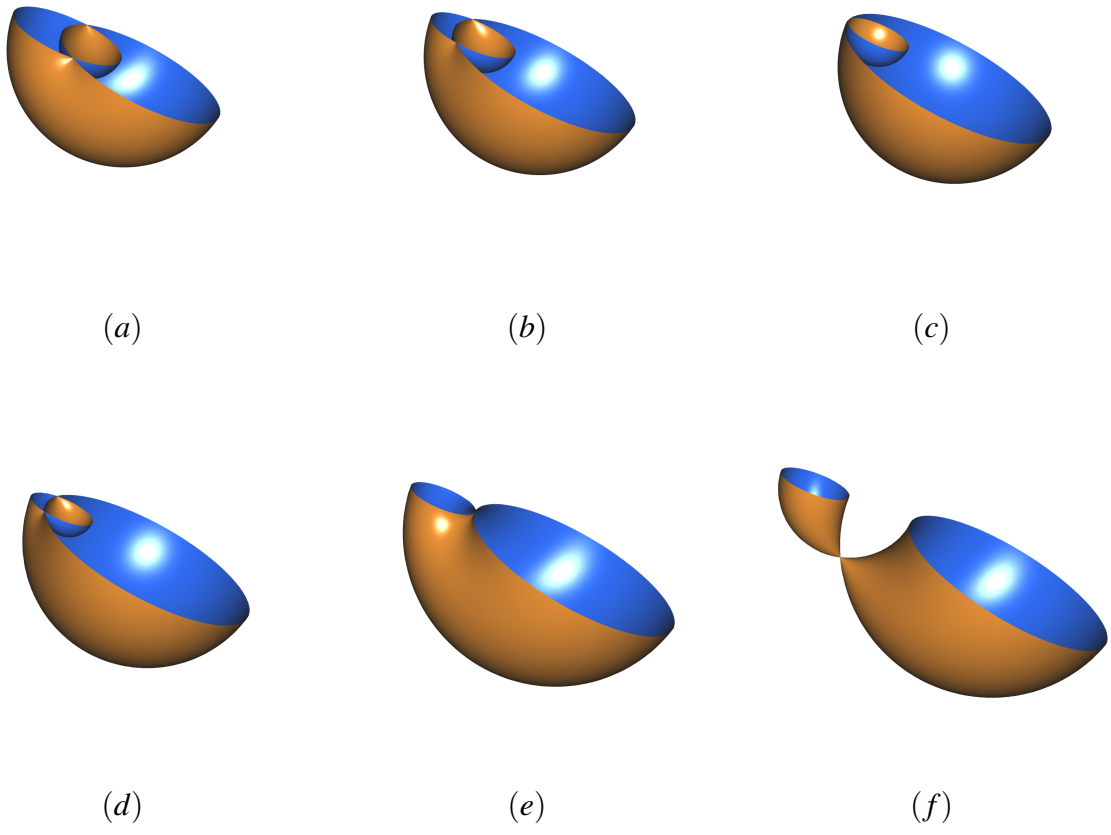


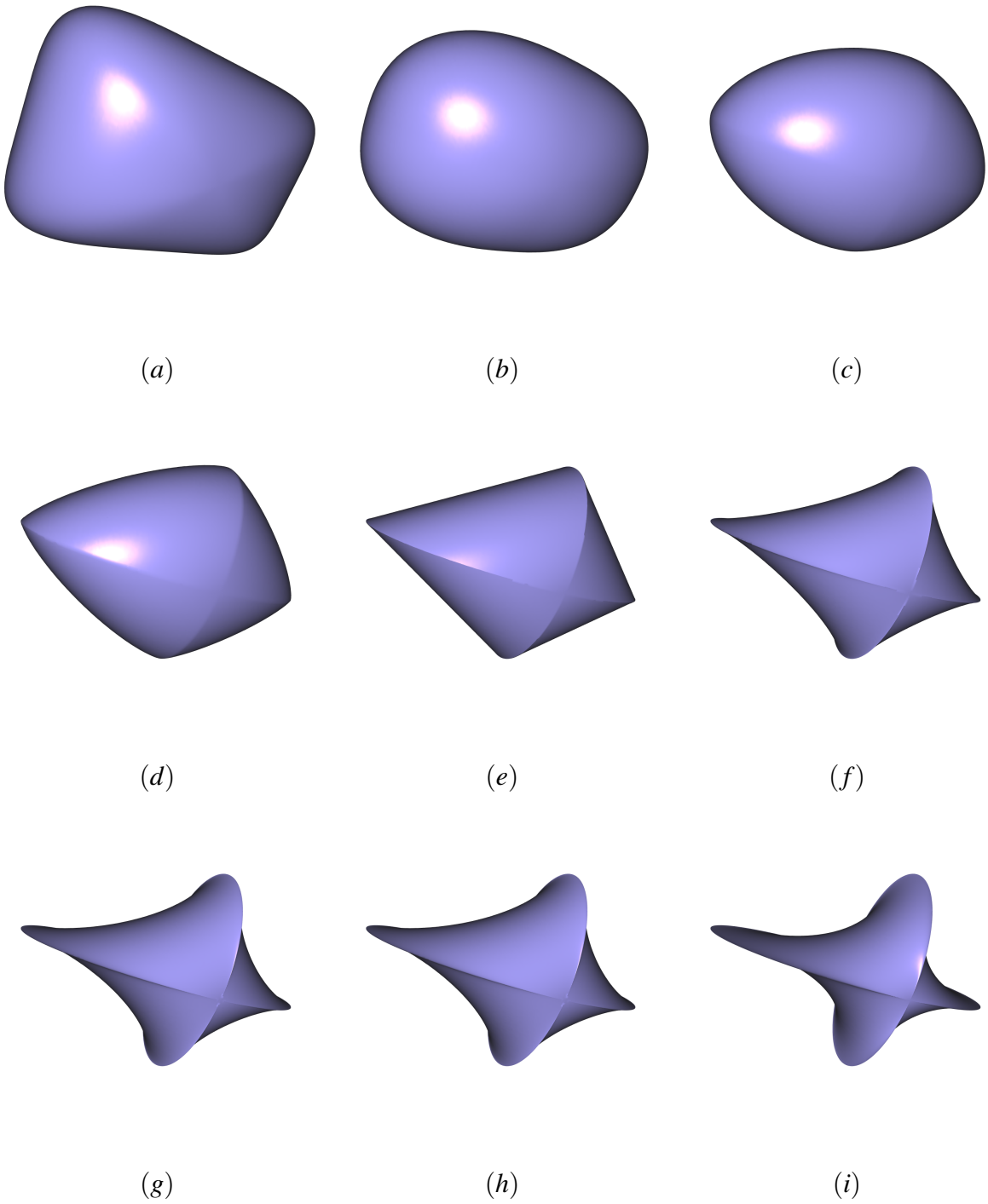
Figure 5.15: Frames from an animation moving down the z axis, where $z_a = 10$ and $z_i = 4$ with steps in z of -$\frac{2}{3}$ between each of the frames of the Klein bottle surface f[3] .



(a) (b) (c)

(d) (e) (f)

Figure 5.16: Forms of the cyclide surface $f[7]$. (a) $r << a < f$, (b) $r < a < f$, (c) $r = a < f$, (d) $a < r < f$, (e) $a < r = f$, and (f) $a < f < r$.

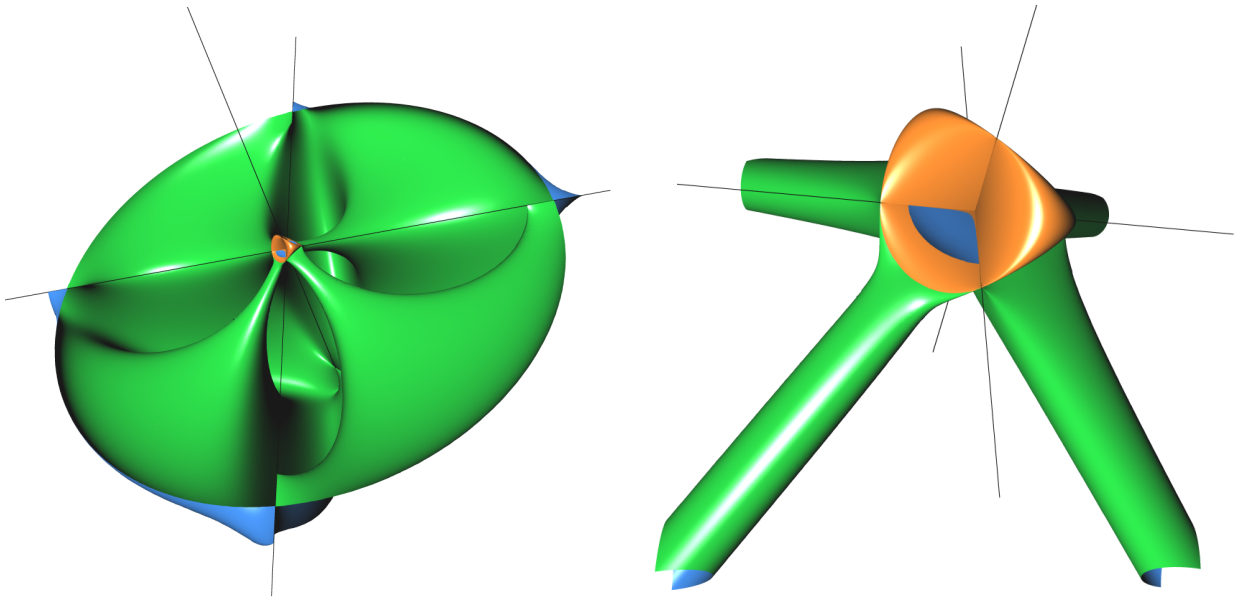


(a) (b) (c)

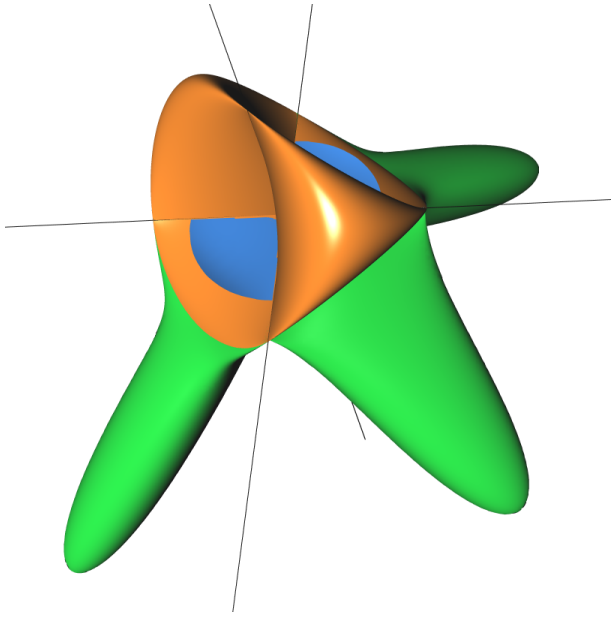
(d) (e) (f)

(g) (h) (i)

Figure 5.17: Various forms of the super-ellipsoid surface f[22] with $a = 12$, $b = 8$ and $c = 5$. $\varepsilon_1 = 1.0$, and ε_2 varies from 0.4 to 3.6 in steps of 0.4 from left to right, top to bottom of the figure.



(a) (b)



(c)

Figure 5.18: Steiner's surface in gold and the (a) $K = -0.03$, (b) $K = 0$, and (c) $K = +0.03$ curvature surfaces in green/blue, superimposed in a single figure with the curvature surfaces clipped to the xz plane.

Chapter 6

Conclusion

In our initial work on rendering non-manifold surfaces we presented a point-based technique that improves the rendering of non-manifold implicit surfaces by using point and gradient information to prune plotting nodes. The use of interval techniques also guarantees that no parts of the surfaces are missed in the view volume and the combination of point and gradient sampling preserves this feature greatly enhancing the quality of point-based rendering of implicit surfaces. This algorithm can successfully render many non-manifold surfaces, and is much simpler than scan-line (i.e. polygon-based) techniques. It can also render surfaces that can't be rendered with traditional ray-casting techniques due to difficulties in solving the ray-surface intersection equation.

We have explored a number of methods for anti-aliasing edges and contours of implicitly defined surfaces. We give two new algorithms for anti-aliasing silhouette edges and contour edges, one using the GPU to analyse edge regions, and the other using the GPU to render multiple buffers which are combined into an anti-aliased image. We did not find that an object space method based on counting the number of nodes projecting

onto a single pixel produced satisfactory results. We then turned our attention to image space methods.

Originally we experimented with an edge blur method that anti-aliased pixels around silhouette edges based on edge coverage in a 3×3 kernel. Whilst this produced better results than the object based approach it still suffered when the slope of the silhouette edge was either steep or shallow. Next we looked at super-sample anti-aliasing, 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 anti-aliasing of the point-based images. The anti-aliasing 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 anti-aliasing 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 anti-aliasing. We compared two methods for contour anti-aliasing. The first method used a weighted average of the distance from the point, $\overrightarrow{P}$, over the slab half-width to weight pixel's 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 anti-aliasing methods described previously for silhouette edges, to anti-alias the surface, as this method is simple and provides good results.

Finally, (see Figure 2.19), we presented a number of anti-aliased images of non-manifold implicit surfaces to demonstrate the improvements we 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 [52] 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.

We then developed a point-based algorithm that reduces point overestimation when using interval methods to drive octree spatial subdivision for implicit surface visualisation. We have also developed two new algorithms for rendering pixel contours on surfaces. One works in image space, the other in object space. Our previous algorithm for anti-aliasing silhouette edges and contours in point-based surface images works well with the new algorithms including with textures. We have illustrated the robustness of these techniques by efficiently rendering a number of implicit surfaces with non-manifold features such as rays, cusps, thin sections, ridges, points, and arbitrarily thin tubes. These non-manifold features can be rendered whether they are aligned with the octree node edges or not, see Figure 6.1 where two planes intersect along an arbitrary placed line in 3D. The methods are able to reduce the overestimation with any interval based spatial

subdivision algorithm without affecting its robustness. This is a major contribution of the work.

Figure 6.1: Visualisation of two intersecting planes.

We have also rendered the intersection curves of an implicit surface with a series of planes, and in principle this can be extended to rendering the intersection curves of arbitrary implicit surfaces. We are able to render the curve of self intersection of a surface by exploiting the fact that surfaces are singular where they self intersect. As an application of the object space contouring algorithm, we rendered the curve of zero Gaussian curvature on Boy's surface. We also illustrated procedural texturing on point-based surfaces, and rendered Boy's surface with a Gaussian curvature map. Finally we use our algorithm to render the curvature surfaces of a number of non-manifold implicit surfaces.

A major advantage of our algorithms is that they are much simpler than scan-line

techniques and can successfully render many non-manifold surfaces. Their simplicity lies in the fact that they only require surface interval function evaluations to generate the points, and gradient expressions to shade them. Other advantages include their efficiency in reducing overestimation, their ability to produce accurate high quality images of non-manifold and singular surfaces, and the ease of performing texturing, including contouring.

One problem with our algorithm still remains. The bohemian star surface in Figure 4.23 (b) shows some aliasing along the eight rays that emanate from the two surface parts that intersect along the fold. The aliasing is more visible in low resolution images. It is encouraging that these rays emerge and bend as clearly the algorithm can track 3D space curves.

Advantages of our approach over ray-casting include that the surface geometry is uniformly sampled in function space, as opposed to screen-space. This allows us to capture the surface once and then use the surface points (stored in a geometry buffer on the GPU) to view the surface from any camera position without recalculating the surface points. Our algorithm will be faster than ray-casting once the geometry has been calculated as the pre-calculated points are simply rendered using a depth buffer approach. However, if we zoom in on the surface, then the spacing between points will increase, and holes will form on the image. This requires the surface to be re-sampled at a higher plot depth for the holes to be filled.

Another advantage is that the hybrid points and polygon approach is not difficult to implement. We avoid complexity by not attempting to polygonise non-manifold regions, where many rare cases must be handled for successful polygonisation. Regions which cannot be polygonsied are subdivided using points. By using points, non-manifold features such as singular lines, thin tubes, cusps, thin sections, and lines of self intersection are accurately rendered. This allows us to produce high quality images while reducing the total geometry.

A limitation of using polygons is the reliability of the criteria for deciding if a node should be polygonised. When "flat" cusps occur, such as in the super-ellipsoid surface, our algorithm fails to correctly detect the non-manifold sections due to a failure of the gradient test. As a result some polygons form across the cusps that are rendered with incorrect shading. This causes the anomalous shading in Figure 5.17 (e) to (h) along the cusp lines toward the centre where the cusps join.

Another issue is when there are too many non-manifold regions on a surface. Time will be wasted subdividing space only to determine that it cannot be polygonised. This detracts from the performance gains of rendering using polygons, a situation where using the point-based algorithm on all nodes would result in a faster render.

In Table 6.1 we summarise the findings in this work. The robustness indicates how well surface features are rendered for the range of implicit surfaces examined for the three methods. Next we compare the time it takes to determine the surface geometry. Quality is

a measure of the combination of accuracy of the rendering, aliasing, and shading issues. Finally we estimate the respective animation time and quality of the competing methods.

Table 6.1: Comparison of speed, robustness, quality and animation speed using ray-casting, points/polygons and point splatting.

	Robustness	Geometry time	Quality	Animation Speed
Ray-casting	high	100%	high	100%
Pts/Poly	very high	15-250%	very high	20%
Splatting	medium	10-300%	medium	15%

The robustness of ray-tracing depends upon how well the interval function bounds the surface. We found that even with a low epsilon ray-tracing had trouble producing sharp intersection lines, such as on Boy's surface in Figure 5.13. Our points/polygon method handled this well, producing the sharp intersections. In this case splatting also performed well. The main limitation to splatting is the size of the splats needed to give a good quality render. If the splats are too small, we can end up needing just as many sampled points as if we rendered using polygons/points.

While ray-casting may be faster for single static images it compares poorly when multiple images are produced. Even for single images the performance comparison drops when anti-aliasing is required. Multiple rays need to be fired to do this in ray-tracing which is linear in time. For the points/polygons method, new views only need to be

rasterised on the GPU as part of the rendering pipeline. This makes anti-aliasing much faster in our method. The quality of splatting is not as good, and results in a softer looking image when compared with ray-casting and points/polygons.

Producing animations of surfaces can be done much faster with our method as we can reuse the geometry. By using points and polygons we minimised the amount of geometry needed to produce a frame and also kept the quality in the non-manifold regions as shown in DVD Chapter 4. We found splatting problematic when the camera moved around and showed parts of the surface at high inclines. We find the point/polygon method to be an overall good solution to rendering a wide variety of manifold and non-manifold implicit surfaces.

Chapter 7

Future Work

To improve the visualisation of the surfaces a number of useful techniques can be incorporated into the current techniques. For example, the modulation of surface colour by functions has a number of applications. This can be used to visualise intrinsic properties of surfaces such as curvature as contour lines or gradient maps. In addition, the contour lines that result from intersecting a surface with the level surfaces of a scalar field can be used to help visualise the surface. See Balsys and Suffern [54].

We intend to further study the curvature surfaces of Boy's surface and the Klein bottle because these are two families of interesting and complex surfaces. Some questions we would like to answer are as follows. How do the different forms of the curvature surfaces in Figure 4.27 metamorphose into each other as K varies? What are the internal structures of the closed surfaces? The surfaces only intersect Boy's surface when $k \in [kmin, kmax]$. What do the surfaces look like when k is outside this interval? We can ask similar questions about the curvature surfaces of the Klein bottle, and also, are these surfaces single sided?

For each implicit surface $f(x,y,z) = 0$ we can define a double infinite series of curvature surfaces with the following recurrence relation:

$$K^{(n+1)}(x,y,z) = K[K^{(n)}(x,y,z)] = k,$$

where $n \in [0,\infty), k \in \Re$, and $K^{(0)}(x,y,z) = K(x,y,z)$. Similar series can be defined using the mean curvature. We intend to explore these sequences for Boy's surface, the Klein bottle, and other implicit surfaces. Because the geometric and function complexity increase exponentially with n, we may not be able to go very far. These surfaces will, however, provide useful test surfaces for rendering algorithms by us and other people on current and future hardware.

In the future devising an algorithm to allow for surface reflection and refraction is a goal. We will use a standard ray-tracing approach for this with the points / polygons algorithm being used to find the ray-surface intersection. Using this approach we should be able, for instance, to generate partially transparent surfaces.

Bibliography

[1] D.J. Harbinson, R.J. Balsys, and K.G. Suffern. Point rendering of non-manifold surfaces with features. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 47–53. ACM New York, NY, USA, 2007.

[2] D.J. Harbinson, R.J. Balsys, and K.G. Suffern. Real-Time Antialiasing of Edges and Contours of Point Rendered Implicit Surfaces. In *Computer Graphics, Imaging and Visualisation, 2008. CGIV'08. Fifth International Conference on*, pages 38–46, 2008.

[3] V. Chandru, D. Dutta, and C.M. Hoffmann. On the geometry of dupin cyclides. *The Visual Computer*, 5:277–290, 1989.

[4] R.J. Balsys and K.G. Suffern. Visualisation of implicit surfaces. *Computers & Graphics*, 25(1):89–107, 2001.

[5] R.J. Balsys and K.G. Suffern. Ray tracing surfaces with contours. *Computer Graphics Forum*, 22(4):743–752, 2003.

[6] M. Levoy and T. Whitted. *The Use of Points as a Display Primitive.* University of North Carolina at Chapel Hill, 1985.

[7] B.T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[8] M. Spivak. A comprehensive introduction to differential geometry. *Publish or Perish*, 3:204, 1979.

[9] H.H. Chen and T.S. Huang. A survey of construction and manipulation of octrees. *Computer Vision, Graphics, and Image Processing*, 43(3):409–431, September 1988.

[10] J. Beck, R. Farouki, and J. Hinds. Surface analysis methods. *IEEE Comput. Graph. Appl.*, 6(12):18–36, 1986.

[11] K.G. Suffern and R.J. Balsys. Rendering the intersections of implicit surfaces. *IEEE Computer Graphics And Applications*, 23(5):70–77, 2003.

[12] J. Bloomenthal. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers, CA, 1997.

[13] J.D. Foley, A. Van Dam, S.K. Feiner, and J.F. Hughes. *Computer graphics: principles and practice*. Addison-Wesley, Reading, MA, (2 ed.) edition, 1990.

[14] H. Fuchs, Z.M. Kedem, and B.F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, New York, NY, USA, 1980. ACM.

[15] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[16] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[17] R. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs.

[18] K.G. Suffern and E.D. Fackerell. Interval methods in computer graphics. *Computers & graphics*, 15(3):331–340, 1991.

[19] J.M. Snyder. *Generative modeling for computer graphics and CAD: symbolic shape design using interval analysis*. Academic Press Professional, Inc. San Diego, CA, USA, 1992.

[20] N. Stolte and Z. Arie Kaufman. Parallel spatial enumeration of implicit surfaces using interval arithmetic for octree generation and its direct visualization. In *Implicit Surfaces 98 Seattle*, ACM Workshop, pages 81–87, 1998.

[21] R.J. Balsys, K.G. Suffern, and H. Jones. Point based rendering of non-manifold surfaces. *Computer Graphics Forum*, 12(2):0–0, 2007.

[22] J. Luiz and J. Stolfi. Affine arithmetic and its applications to computer graphics. *Relation*, 10(1.56):9–18, 1993.

[23] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[24] J. Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5(4):341–355, 1988.

[25] R.J. Balsys and K.G. Suffern. Adaptive polygonisation of non-manifold implicit surfaces. In *CGIV '05: Proceedings of the International Conference on Computer Graphics, Imaging and Visualization*, pages 257–263, Washington, DC, USA, 2005. IEEE Computer Society.

[26] G. Wyvill, C. McPheeters, and B. Wyvill. Data Structure for Soft Objects. *The Visual Computer*, 2(4):227–234, February 1986.

[27] M.F.W. Schmidt. Cutting cubes—visualizing implicit surfaces by adaptive polygonization. *The Visual Computer*, 10(2):101–115, 1993.

[28] Y. Ohtake and A. Belyaev. Dual-primal mesh optimization for polygonized implicit surfaces with sharp features. *J. Comput. Inf. Sci. Eng.*, 2(4):277–284, 2002.

[29] T. Lewiner, H. Lopes, A.W. Vieira, and G. Tavares. Efficient implementation of marching cubes' cases with topological guarantees. *Journal of graphics tools*, 8(2):1–15, 2003.

[30] Y. Ohtake, A. Belyaev, and A. Pasko. A.: Dynamic mesh optimization for polygonized implicit surfaces with sharp features, the visual computer. *The Visual Computer*, 19:115–126, 2002.

[31] S. Schaefer and J. Warren. Dual marching cubes: Primal contouring of dual grids. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, pages 70–76, Washington, DC, USA, 2004. IEEE Computer Society.

[32] T. Ju. Intersection-free contouring on an octree grid. In *Proc. 14th Pacific Conf. Computer Graphics and Applications*. unknown, 2008.

[33] A. Paiva, H. Lopes, T. Lewiner, and L.H. de Figueiredo. Robust adaptive meshes for implicit surfaces. *Computer Graphics and Image Processing*, 0:205–212, 2006.

[34] The stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep/.

[35] M. Gross. Getting to the point...? *IEEE Computer Graphics and Applications*, 26(5):96–99, 2006.

[36] A. Rockwood. A generalised scanning technique for display of parametrically defined surfaces. *IEEE Computer Graphics and Applications*, 7(8):15–26, 1987.

[37] A.P. Witkin and P.S. Heckbert. Using particles to sample and control implicit surfaces. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 269–277, New York, NY, USA, 1994. ACM.

[38] J.C. Hart, E. Bachta, W. Jarosz, and T. Fleury. Using particles to sample and control more complex implicit surfaces. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 269, New York, NY, USA, 2005. ACM.

[39] L.H. de Figueiredo and J Gomes. Sampling implicit objects with physically-based particle systems. *Computers & Graphics*, 20(3):365–375, 1996.

[40] A. Rosch, M. Ruhl, and D. Saupe. Interactive visualization of implicit surfaces with singularities. *Computer Graphics Forum*, 16(5):295–306, 1997.

[41] S. Tanaka, A. Morisaki, S. Nakata, Y. Fukuda, and H. Yamamoto. Sampling implicit surfaces based on stochastic differential equations with converging constraint. *Computers And Graphics*, 24(3):419–431, 2000.

[42] S. Tanaka, A. Shibata, H. Yamamoto, and H. Kotsuru. Generalized stochastic sampling method for visualization and investigating of implicit surfaces. *Computer Graphics Forum*, 20(3):359–367, 2001.

[43] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[44] S. Rusinkiewicz and M. Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[45] M. Zwicker, J. van Baar, and M. Gross. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378. ACM New York, NY, USA, 2001.

[46] P.S. Heckbert. Fundamentals of texture mapping and image warping. Technical report, Berkeley, UC, USA, 1989.

[47] M. Botsch, M. Spernat, and L. Kobbelt. Phong splatting. In *Eurographics Symposium on Point- Based Graphics*, pages 25–32, 2004.

[48] M. Zwicker, J. Räsänen, M. Botsch, C. Dachsbacher, and M. Pauly. Perspective accurate splatting. In *GI '04: Proceedings of Graphics Interface 2004*, pages 247–254, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[49] C.S. Co, B. Hamann, and K.I. Joy. Iso-splatting: A point-based alternative to isosurface visualization. In *In Proceedings of the Eleventh Pacific Conference on Computer Graphics and Applications - Pacific Graphics 2003*, pages 325–334, 2003.

[50] R.J. Balsys and K.G. Suffern. Point based rendering of implicit 4-dimensional surfaces. In *CGIV '07: Proceedings of the Computer Graphics, Imaging and Visualisation*, pages 31–40, Washington, DC, USA, 2007. IEEE Computer Society.

[51] R.J. Balsys, K.G. Suffern, and H. Jones. Point-based rendering of non-manifold surfaces. *Computer Graphics Forum*, 27:63–72(10), 2008.

[52] J.M. Singh and P.J. Narayanan. Real-time ray-tracing of implicit surfaces on the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 99(IIIT/TR/2007/72):261–272, 2009.

[53] A. Knoll, Y. Hijazi, C. Hansen, I. Wald, and H. Hagen. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. *Computer Graphics Forum*, 28:26–40, 2008.

[54] R.J. Balsys and K.G. Suffern. Point based rendering of non-manifold surfaces with contours. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 7–14, New York, NY, USA, 2004. ACM.

[55] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

[56] N. Stolte. Graphics using implicit surfaces with interval arithmetic based recursive voxelization. In *Proceedings Computer Graphics and Imaging*, pages 200–205. Honolulu, USA, 2003.

[57] J. Bloomenthal. Edge inference with applications to antialiasing. *SIGGRAPH Comput. Graph.*, 17(3):157–162, 1983.

[58] D. Mitchell and P. Hanrahan. Illumination from curved reflectors. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 283–291, New York, NY, USA, 1992. ACM.

[59] H. Pottmann and K. Opitz. Curvature analysis and visualization for functions defined on euclidean spaces or surfaces. *Comput. Aided Geom. Des.*, 11(6):655–674, 1994.

[60] N. Guid, C. Oblonsek, and B. Zalik. Surface interrogation methods. *Computers And Graphics*, 19(4):557–574, 1995.

[61] H. Jones, R.J. Balsys, and K.G. Suffern. Point based rendering of surfaces with singularities. pages 267–268. ACM/GRAPHITE2003, 2003.

[62] Alvy Ray Smith. Digital paint systems: An anecdotal and historical overview. *IEEE Ann. Hist. Comput.*, 23:4–30, April 2001.

[63] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[64] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[65] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5:51–72, January 1986.

[66] E.R. Hansen. Global optimisation using interval analysis: the multidimensional case. *Numerische Mathematik*, 34, 1980.

[67] H. Ratschek and J. Rokne. *New computer methods for global optimization*. Halsted Press, New York, NY, USA, 1988.

[68] T.W. Sederberg and A.K. Zundel. Scan line display of algebraic surfaces. *SIGGRAPH Comput. Graph.*, 23(3):147–156, 1989.

[69] J. Bloomenthal and K. Ferguson. Polygonization of non-manifold implicit surfaces. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 309–316, New York, NY, USA, 1995. ACM.

[70] J. Steiner. *Gesammelte Werke*. Prussian Academy of Sciences, 1971.

[71] L.H. de Figueiredo and J. Stolfi. Adaptive enumeration of implicit surfaces with affine arithmetic. *Computer Graphics Forum*, 15:287–296, 1996.

[72] F. Klein. *Uber Riemann's Theorie der algebraischen Functionen und ihrer Integrale*. Teubner Leipzig, 1882.

[73] C. Jessop. *Quartic surfaces with singular points*. Cambridge University Press, jes16.

[74] R. Kusner and N. Schmitt. The spinor representation of minimal surfaces. GANG, 1995.

[75] A.H. Barr. Superquadratics and angle-preserving transformations. *IEEE Computer Graphics and Applications*, 1:11–23, 1981.

[76] S.V. Chmutov. Examples of projective surfaces with many singularities. *Journal of Algebraic Geometry*, 1:191–196, 1992.

[77] W. Barth. Two projective surfaces with many nodes admitting the symmetries of the icosahedron. *Journal of Algebraic Geometry*, 5:173–186, 1996.

[78] F. Apéry. The Boy's surface. *Adv. Math.*, 61:185–266, 1986.

[79] R.J. Balsys and K.G. Suffern. Adaptive polygonisation of non-manifold implicit surfaces. volume 24, pages 215–233, Washington, DC, USA, 2005.

[80] W. Chen, L. Ren, M. Zwicker, and H. Pfister. Hardware-accelerated adaptive ewa volume splatting. In *Proceedings of the conference on Visualization '04*, pages 67–74. IEEE Computer Society, 2004.

[81] C. Loop and J. Blinn. Real-time GPU rendering of piecewise algebraic surfaces. *ACM Trans. Graph.*, 25(3):664–670, 2006.