Polygonization of Implicit Models

by

Mauricio Andres Rovira Galvez

B.Sc., University of Victoria, 2015

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

University of Victoria

Polygonization of Implicit Models

by

Mauricio Andres Rovira Galvez

B.Sc., University of Victoria, 2015

Supervisory Committee

Dr. Brian Wyvill, Supervisor
(Department of Computer Science)

Dr. Kwang Moo Yi, Academic Unit Member
(Department of Computer Science)

Supervisory Committee

Dr. Brian Wyvill, Supervisor
(Department of Computer Science)

Dr. Kwang Moo Yi, Academic Unit Member
(Department of Computer Science)

ABSTRACT

In computer graphics, implicit modelling is a way of representing models by using combinations of implicit functions. These are then polygonized to produce a mesh typically formed from either quads or triangles. We present a comparison of two fundamental algorithms in polygonization in order to produce an algorithm that is better than the current industry standard: Marching Cubes. By using planar cross-sections we are able to bypass several problems from Marching Cubes, as well as produce a mesh of higher resolution and quality. The algorithm is limited to surfaces of genus 0, but it still outperforms Marching Cubes both in runtime as well as overall mesh quality.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank:

Brian Wyvill, for offering me this great opportunity, as well as mentoring, support, encouragement, and patience.

my mother, for all her support, encouragement, and help throughout this journey.

my father for the continuous support he has provided.

I'm not telling you it's going to be easy - I'm telling you it's going to be worth it.
Art Williams

Dedication

To my mother. Another step in this incredible journey has been taken. Always shooting for the stars.

Chapter 1

Introduction

1.1 Implicit Modelling

Implicit modelling (or implicit surfaces) is a subset of computer graphics which focuses on several ways of representing models by way of implicit functions. A function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be *implicit* if it is of the form $f(\mathbf{x}) = c$ for some constant $c \in \mathbb{R}$. The constant c is known as the iso-value (or iso-level). In computer graphics, this value is usually set to either 0 or 0.5 depending on the application. An *implicit surface* is then the set of all points $\mathbf{x} \in \mathbb{R}^n$ such that $f(\mathbf{x}) = c$. In other words, $S = \{\mathbf{x} \in \mathbb{R}^n : f(\mathbf{x}) = c\}$. S is also known as the *zero-set* or *kernel* of f. A simple example of an implicit surface is a unit sphere centred at the origin: $f(x, y, z) = x^2 + y^2 + z^2 = 0$ [Wyvill, 2009].

An implicit function can be re-written into the form $s(\mathbf{x}) = f(\mathbf{x}) - c = 0$, in which case it becomes a *signed-distance field function* (*SDF*). For a given point $\mathbf{x} \in \mathbb{R}^n$, $s(\mathbf{x})$ will return a signed value that corresponds to the position of $\mathbf{x}$ relative to the surface. In other words, if $\mathbf{x}$ is inside, on, or outside the surface, the sign of $s(\mathbf{x})$ will be $-$, 0, or $+$, respectively [Bloomenthal and Wyvill, 1997].

By fixing the value of c, we can combine several *SDF* by applying a variety of

functions. These functions are called *operators*, some examples include:

- The summation blend: $sum(\mathbf{x}) = \sum_{i=0}^{n} s_i(\mathbf{x})$,
- union: $union(\mathbf{x}) = \max_{0 \leq i \leq n}(s_i(\mathbf{x}))$,
- intersection: $instersection(\mathbf{x}) = \min_{0 \leq i \leq n}(s_i(\mathbf{x}))$,
- amongst others [Gourmel et al., 2013].

These operators can then be combined into a hierarchical structure known as the *Blobtree* to create complex models [Wyvill et al., 1998]. An example is shown in Figure 1.1.

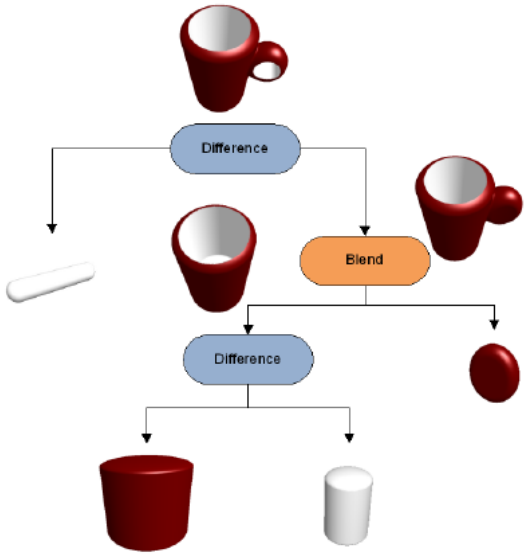


Figure 1.1: An example of a Blobtree.

The last point of interest refers to the "range" of an implicit surface. By our definition, the range of $s(\mathbf{x})$ would be $(-\infty, \infty)$. Consider the case where we have two spheres and we wish to apply a summation blend as previously defined. As we move the spheres further apart, there comes a point at which we would like them not to be blended together. Unfortunately, due to the definition of $s(\mathbf{x})$, they will *always*

blend, regardless of the distance between them. This is known as *global* support. In order to limit the region of influence of a skeletal primitive, we can use a function to clamp it's influence at a certain distance from the surface called a filter-fall-off function. An example is

$$g(d) = \left(1 - \frac{d^2}{r^2}\right)^3$$

Where d is the distance given by $s(\mathbf{x})$ and r is a given radius of influence [Wyvill, 2009]. The usage of such function is known as *compact* support.

1.2 Polygonization of Implicit Models

There are two ways of visualizing implicit surfaces: ray tracing [Hart, 1993; Hart, 1996; Wyvill and Jevans, 1988; Singh and Narayanan, 2010] or by conversion into polygons [Lorensen and Cline, 1987; Wyvill et al., 1986; Bloomenthal, 1988]. We will focus on the latter, which is referred to as *polygonization*.

Polygonization can produce one of two outputs. The first is a polygonal mesh, while the second is a polygonal soup. There are two types of polygons that are generally employed: triangles and quads. Most polygonization schemes output triangles, while quads are usually used by most modelling software [Bommes et al., 2012]. Both a polygon mesh and a polygon soup consist of lists of polygons. The difference is in their properties, which we will outline next. A polygon mesh must have the following attributes:

1. The list of vertices that comprise the polygons must be *unique*. In other words, a vertex may not appear more than once in the list.

2. The mesh must be water-tight. That is to say that there should be no cracks or holes in the mesh.

A polygon soup, on the other hand is just a list of polygons, with no real representation of how these are connected. Furthermore, vertices appear as many times as there are polygons that share that particular vertex. In general, most applications in graphics prefer a mesh over a polygon soup as it is easier to process, modify, and render. We will focus on triangles, as they are the most common type of polygonal mesh.

Finally, the quality of a triangle mesh is measured by the types of triangles from which it is made. In general, a mesh that consists of mostly (or entirely of) equilateral triangles is considered a *high* quality mesh, whereas a mesh consisting of scalene or degenerate triangles is considered a *low* quality mesh. Once more, in general graphics applications tend to prefer high quality meshes as they have lower memory requirements and can also be easily manipulated and rendered [de Araújo et al., 2015].

1.3 Problem Description

Currently, the industry standard for polygonization is an algorithm called Marching Cubes. As we will see in chapter 2, this algorithm does not produce a high quality mesh [de Araújo et al., 2015], but is generally considered to be very simple and easy to optimize. To this end, the focus of this work will be the following:

Problem 1. *We wish to create a polygonization algorithm that has the following properties:*

1. *Be* at least *as fast, if not faster than, parallel Marching Cubes as in [Hansen and Hinker, 1992].*
2. *Produces a better quality mesh than Marching Cubes as in [de Araújo et al., 2015].*

3. *Produces a mesh that accurately represents the topology of the underlying surface as in [Chernyaev, 1995].*

We will focus exclusively on polygonizing models that were created by combining implicit functions together. The reasons for this are:

- We are guaranteed that the surface that we are trying to polygonize will maintain certain continuity properties [Hart, 1996],
- Since we know exactly which functions are used to create the model (and how they are combined), we can reduce the number of field evaluations to only those that are strictly necessary [Wyvill et al., 1986].

1.4 Thesis Structure

This work is divided into the following major sections:

1. Related Work: here we will examine the research that has been done in polygonization. We will overview the major techniques, their advantages and disadvantages. Since our problem is focused on implicit models, algorithms that are exclusively used for reconstruction will not be included.
2. Comparison between Marching Cubes and Bsoid: we compare and contrast two fundamental algorithms [de Araújo et al., 2015] in order to select one that will give us the best advantages.
3. Marching Rings: we present our solution in full detail, along with its limitations and a comparison against Marching Cubes.

The contributions of this work are as follows:

- A comparison between two fundamental algorithms in polygonization using modern optimization techniques.
- An modern implementation of Bsoid which provides an easy way to parallelize it.
- The proposal of a new polygonization algorithm that is faster than the current industry standard while producing meshes of significantly higher quality and fidelity.

Chapter 2

Related Work

This chapter is divided into 2 main sections, dealing with a different class of polygonization techniques. The order is as follows:

- Speed-based: these are algorithms that focus on producing a final mesh as quickly as possible, with little concern given to the quality of the final mesh.
- Mesh-based: these are algorithms that focus on producing the best final mesh possible at the cost of performance. Particular attention is given to generating meshes that consist almost entirely of equilateral triangles.

As previously mentioned, we will focus exclusively on algorithms intended for implicit surfaces (or implicit models). This means that algorithms designed to reconstruct surfaces from point clouds, or other variants lie beyond the scope of this work. For a full survey on polygonization techniques, see [de Araújo et al., 2015].

2.1 Speed-based Algorithms

There are 3 fundamental algorithms that form the base for all the research that has been done in this area. Before we begin our discussion, it is important that they are reviewed in detail, in order to fully understand the course that later research would

take. These algorithms are presented in [Wyvill et al., 1986], [Lorensen and Cline, 1987], and [Bloomenthal, 1988].

We begin by examining [Lorensen and Cline, 1987], known as *Marching Cubes* (*MC*). The algorithm first divides space (the area of interest) into a regular cubical grid. The cubes that conform the grid are referred to as *voxels* (since the same approach can be extended to n-dimensional space). Then, for each voxel, the field is sampled at each one of the 8 corners. Depending on the sign at each corner, an index is generated, which is then used to access a look-up table which contains all the possible configurations that a cube might have. While it is true that there are 2^8 possible configurations (2 possible signs per corner), only 15 are actually required, since the rest can be derived by symmetry. The look-up table is shown in Figure 2.1.

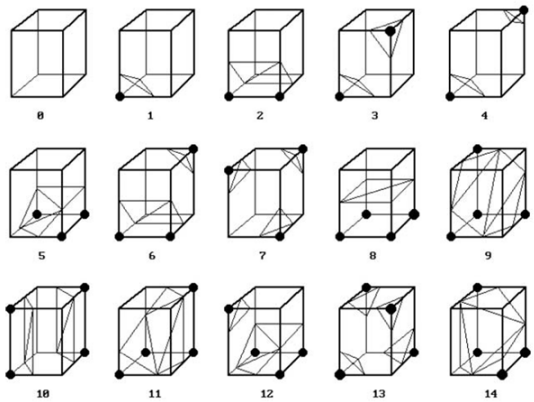


Figure 2.1: The 15 cases in the *MC* look-up table.

Once the voxels are classified, the polygons are created as described in the case table. The vertices are placed along the edges of the cubes using linear interpolation. The end-product of this algorithm is a triangle soup, which can then be further processed into a full mesh.

MC is the current industry standard due to its simplicity, ease of implementation, and trivial parallelization. In fact, it is so well-known that the grand majority of research done since its publication has been based on it [de Araújo et al., 2015].

Next, we look at *Bsoid* [Wyvill et al., 1986]. The algorithm takes as its input a set of skeletal primitives, but can also take a general implicit surface. From there, the region of interest is divided into volumes similar to a bounding volume hierarchy (BVH). These volumes contain a list of all those primitives that influence their respective regions and are referred to as *super-voxels*. In order for these to be effective, all fields must employ compact support, since otherwise all implicits would influence *all* the space. Once the super-voxels have been created, the process continues by dividing the model into the same voxel grid that *MC* uses and selecting the voxels that contain the skeletal points themselves. It is important to note that the voxel grid and the super-voxels are distinct and in no way related. It is preferred that the super-voxels are based on the voxel grid, but it is not required.

With the list of voxels obtained, the voxels are then marched. This process is accomplished by checking each neighbour of each voxel, repeating this process until one intersects the surface, at which point all voxels that do not contain the surface are discarded and we continue to check all neighbouring voxels, keeping track of only those voxels that contain part of the surface. This process is essentially a breadth-first search (*BFS*), where the voxels are the "nodes" of the graph. In order to ensure that voxels are not re-calculated, a hash table is used to maintain uniqueness. At the end, we are left with a list of voxels that contain the surface which are then processed and the triangles for each voxel are computed in a similar way to *MC*. In order to avoid having to compute the same vertex multiple times, the generated vertices are stored in another hash table which is indexed by the edge of the voxels, thereby ensuring the uniqueness of the vertices. This also has the property of generating a water-tight mesh.

It is important to notice two key differences between *MC* and Bsoid. The first is the fact that by virtue of the first hash table and the search method, Bsoid does

not need to store the entire grid, only those voxels which are of interest. *MC* on the other hand requires the storage of the entire grid, which can result in large memory usage for very fine grids.

The second key difference lies in the way Bsoid avoids having to compute things multiple times. By using the first hash table, voxels need only be computed once, as opposed to *MC* where each vertex in a voxel needs to be computed 8 times, one for each neighbouring voxel. The second hash table further reduces computations by allowing generated vertices to be re-used. *MC* on the other hand will re-generate each vertex as many times as needed.

The final algorithm is by [Bloomenthal, 1988]. In this work, the grid that both *MC* and Bsoid use is replaced with an adaptively subdivided octree. This allows the final mesh to have higher-resolution in areas of higher curvature by dividing the leaves of the octree further. It also introduces the idea of dividing a voxel into tetrahedra, which increases the number of triangles produced, but partially avoids ambiguities present in *MC* due to the use of cubical cells. These last two points represent the main advantages of this approach. The disadvantages are: it is slower than *MC* and Bsoid due to the adaptive subdivision of the grid, as well as the division of voxels, and the usage of tetrahedra results in a significantly higher number of triangles being generated.

As discussed earlier, *MC* became the industry standard. It does, however, have a series of problems with it. These are:

- Ambiguities. Due to the way that the field is sampled, it is possible to construct cases where the voxels will not be able to recognize the topology of the surface. A detailed discussion is shown in subsection 2.1.1.

- Topological inaccuracies. While *MC* will capture general topology of the surface (ignoring the ambiguities discussed in subsection 2.1.1), it does miss sharp and

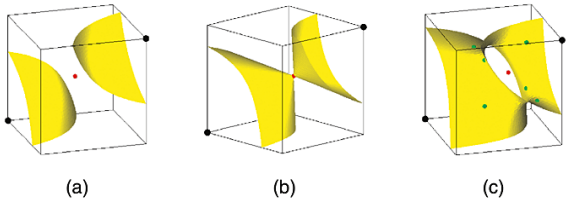
thin features since it is limited by its cube resolution.

- Mesh quality. The final mesh that *MC* produces is not ideal, containing a high number of scalene and even degenerate triangles. This often results in further processing being done on the mesh after it has been created.

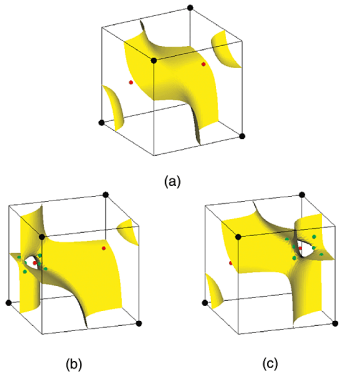
We will examine each problem in the subsequent sections. It is worth noting that the final problem has some overlap with section 2.2, but we will focus our attention on algorithms based on *MC* or its variants. The algorithms discussed in section 2.2 utilize other techniques.

2.1.1 Ambiguities

The ambiguity problem is fully shown in Figure 2.2. Notice that both cases of Figure 2.2a have the same voxel configuration and hence fall under the same case in the *MC* look-up table, yet the underlying surfaces are distinct. The same situation is shown in Figure 2.2b. In both instances, it is impossible to determine what the behaviour of the surface is by sampling alone [Kalra and Barr, 1989], or how to polygonize it. Furthermore, the resulting mesh will have incorrect topological features (such as cracks or holes) or it will miss details of the surface.



(a) Case 1



(b) Case 2

Figure 2.2: Two examples of ambiguities in *MC*. Positive vertices are marked.

One incomplete solution was proposed in [Bloomenthal, 1988] and [Bloomenthal and Ferguson, 1995]. The idea is as follows: since a cube introduces ambiguities, then if a simpler cell was used, the ambiguities would be avoided. The solution consisted of splitting the cube into 5 (or 6) tetrahedra, which are the simplest of all ordinary convex polyhedra and are the only ones that contain less than 5 faces. While this does solve *some* of the ambiguous cases, it does not provide a complete solution.

This approach was expanded in [Hall and Warren, 1990], who replaced the grid of cubical voxels with a "honeycomb" consisting of tetrahedra. As a result, this algorithm is now known as *Marching Tetrahedra*. A complete solution to the ambiguities for tetrahedra was presented in [Hui and Jiang, 1999]. In addition, [Chan and Purisima, 1998] also gave an approach similar to [Bloomenthal, 1988]. The difference is that [Chan and Purisima, 1998] subdivides into 12 tetrahedra instead of 6, thereby increasing the number of samples per cubical cell.

It is worth noting that the name Marching Tetrahedra is also used by algorithms that employ a cubical grid and then subdivide into tetrahedra. Both cases are not commonly used due to the large number of triangles that they produce compared to *MC*.

The approaches described above did not really solve the problem that *MC* had, they simply changed it by using a simpler cell. The ambiguous cases of the *MC* case table were solved by [Chernyaev, 1995]. The approach consisted of expanding the 15-case table from *MC* into a 33-case table that handled all the possible ambiguities. The implementation was refined in [Lewiner et al., 2003] by solving the internal ambiguities (i.e. ambiguities that arise between adjacent voxels), while the table was then modified from 33 to 31 entries in [Lopes and Brodlie, 2003]. The table reduction was accomplished by collapsing 2 cases by symmetry.

The ambiguity problem that *MC* has is closely tied to topological inaccuracies in

the resulting mesh. While solving the ambiguous cases helps this problem, it does not provide a complete solution [Stander and Hart, 2005], as we will see in the following section.

2.1.2 Topological Inaccuracies

While *MC* captures the general topology of the surface, it will not provide all of the details (such as sharp or thin features), which may be critical depending on the use of the mesh. A simple solution to this problem is to increase the grid resolution as seen in Figure 2.3. While this will capture more detail, it has two flaws. First, if the features are too thin, they will not be captured by the grid unless the resolution is increased further. This ties to the second problem, which is that both the time and memory complexity of *MC* grow cubically with the size of the grid, which makes the problem intractable for very fine grids. In order to capture the missing features, other approaches are required.

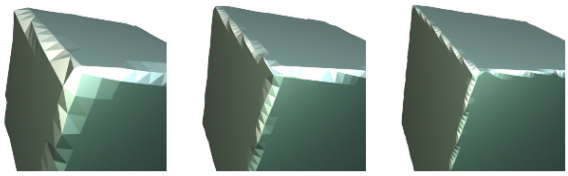


Figure 2.3: Alias errors generated from *MC*. While increasing the mesh resolution reduces the problem (centre and right) it does not fix it altogether.

Various extensions to *MC* have been proposed, where the underlying logic is retained but changes are made to improve topological recognition. First, [Yamazaki et al., 2002] proposed the use of a segmented SDF over a regular SDF which is used by other methods. The segmented SDF allows the algorithm to detect the distance to arbitrary regions, which allows for better topological recognition. Another solution was presented in [Raman and Wenger, 2008]. Here, the corners of the voxels are now allowed to have 3 signs (+, −, and =). This results in a significantly larger

case table (8^3 possible configurations) but allows for a more accurate mesh. Next, [Dietrich et al., 2009] proposed that once the triangles had been computed, they could be transformed along the edges of the voxels. This results in a better mesh and a higher level of accuracy.

As noted, the previous approaches are extensions of MC. We now discuss a branch in polygonization that solved the extraction of sharp and thin features from surfaces.

We begin with [Wyvill and van Overveld, 1996; Kobbelt et al., 2001]. The approach works as follows: inspecting the angles between normals in a surface, it is possible to recognize whether or not a sharp feature exists. If there is one, then the vertex of the corresponding triangle can be placed at the intersection of the two normals. This position can be refined through iterative optimization. This introduces the notion of allowing vertices to be moved beyond the edges and into the space within the voxels. This region is known as the *dual space*. This idea was expanded in [Ju et al., 2002], who allowed any vertex to be moved in the dual space (not just along the intersection of the normals like in [Wyvill and van Overveld, 1996]). This approach is known as *Dual Contours* (DC). It was then expanded in [Varadhan et al., 2003] by combining [Kobbelt et al., 2001] and [Ju et al., 2002]. This allowed the detection of at most one sharp feature per cell and also introduced adaptive subdivision for sharp features.

Adaptive subdivision was exploited in [Schaefer and Warren, 2004] by using an octree. The algorithm also employed two grids: the first one is the same as DC, while the second one is the dual grid. By using an octree on both, it could extract thin features with a lower resolution grid. This was later expanded in [Schaefer et al., 2007] by using vertex-clustering techniques which allowed multiple contour components in one octree cell (while previously only one was allowed). In addition, this was the first mesh that was guaranteed to preserve manifolds.

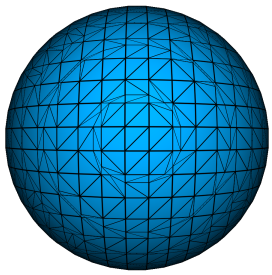


Figure 2.4: A close-up of a sphere polygonized with *MC*.

All of the above approaches solve the topological inaccuracies of *MC*, however they result in meshes that are not ideal. In the next section we discuss the work that has been done to accomplish this.

2.1.3 Mesh Quality

As seen in Figure 2.4, the triangles that *MC* produces are scalene, skinny, or degenerate (either collinear or repeated vertices). An ideal mesh would consist mostly of (or entirely of) equilateral triangles. In addition, there are problems trying to capture the topology of the surface, as seen in the last section. The approaches discussed in this section are mostly focused on either guaranteeing the general topology of the surface (without focusing explicitly on sharp features) or generating a mesh with better triangles. We will first look at Marching Tetrahedra, followed by *MC*, and finally we will conclude with *DC*.

[Treece et al., 1999] extended Marching Tetrahedra by incorporating vertex clustering. By preforming clustering prior to the triangulation step, it preserves the underlying topology of the surface as well as improving the overall quality of the mesh. Another extension to Marching Tetrahedra was presented in [Crespin, 2002] which introduces an iterative subdivision of tetrahedra prior to triangulation. While the resulting mesh is still poor in quality, the iterative subdivision allows for high

resolution meshes to be produced.

The general approach to enhance the mesh produced by *MC* is to post-process the resulting mesh. [Velho et al., 1999] produces a mesh using *MC* with a low resolution grid. The edges of the coarse mesh are then sampled against the surface curvature. If the curvature is not flat, or within some degree of flatness, the edge is then subdivided. This results in a better mesh that captures the smoothness of the surface.

The idea of post-processing the mesh is also applied in [Ohtake and Belyaev, 2002]. In this work, the coarse mesh is refined using the dual mesh (generated from the dual grid like *DC*). This allows the final mesh to have a better triangulation as well as capture sharp features. Finally, both [Bouthors and Nesme, 2007] and [Peir et al., 2007] employ standard mesh processing techniques (such as decimation) to improve the initial mesh from *MC*.

An interesting extension to *MC* to improve the resulting mesh quality is presented in [Galin and Akkouche, 2000]. The approach combines the super-voxels from Bsoid with the adaptive octree from [Bloomenthal, 1988] and *MC*. In addition, it utilizes the Lipschitz constant of the surface to determine when to stop the recursive subdivision of the octree, as opposed to a predefined value as in [Bloomenthal, 1988]. This ensures that the octree does not miss details in the surface. The mesh is then generated with *MC*, which is enhanced with an adjacency graph which is employed to eliminate any cracks that may appear in the mesh.

The post-processing approach was also added to *DC* in [Paiva et al., 2006]. Two important things are presented here: usage of adaptive octrees, and post-processing of the *DC* generated mesh. The octrees allow for better resolution where needed, and the dual space of the octree captures sharp features and details.

Finally, [Varadhan et al., 2006] employs *DC* with a visibility criterion in order to subdivide the grid. The criterion works by inspecting the circumference of a

given point. By examining the vertices that can be "seen", it is possible to determine whether further subdivision is required. This approach also guarantees that the topology of the surface is correctly captured.

2.2 Mesh-based Algorithms

We now turn our attention to those algorithms which focus solely on producing a good final mesh as discussed in section 1.2. The methods presented here are (in general) slower than *MC*. Though some have close to comparable performance, they compare only to *MC* as presented in [Lorensen and Cline, 1987], making those metrics invalid when viewed against more recent approaches.

We begin our discussion with the work presented in [Hilton et al., 1996]. The algorithm works by growing triangles from a starting one (or a mesh) outwards using a hexagonal pattern. The triangles are projected onto the surface to ensure the topology is maintained, while the formation of the triangles themselves is subject to the Delaunay constant. The resulting meshes are very close to ideal. Unfortunately, the algorithm as presented has several flaws:

- It cannot construct closed meshes for closed surfaces. In other words, the triangles don't always align with one another, creating holes and cracks in the mesh.
- It cannot correctly identify sharp features.
- The lengths of the sides of the triangles is fixed.

The second point will become a recurring theme with several of the methods we discuss. The algorithm that [Hilton et al., 1996] presented is called *Marching Triangles* (*MTri* and like *MC*, was expanded in the subsequent years. An example

of a mesh generated by can be seen in Figure 2.5. Notice the quality of the triangles as compared to those seen in Figure 2.4.

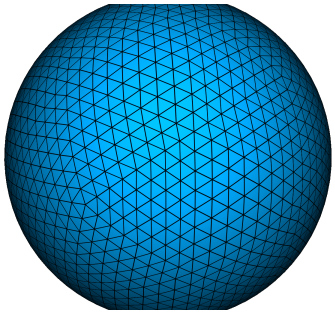


Figure 2.5: A sphere polygonized with *MTri*.

Both [Akkouche and Galin, 2001] and [Karkanis and Stewart, 2001] expanded *MTri*. Instead of growing triangles following a hexagonal pattern it instead grows one edge at a time, essentially "spinning" around the boundary edges of the growing polygon. This allows for cracks in the mesh to be filled easily. The method was expanded in [Cermak and Skala, 2007] by allowing the triangles to be subdivided as needed, thereby improving the resolution of the meshes where it was required. Both approaches share the limitation of being unable to capture sharp or thin features from the surfaces. A comparison between *MTri*, *MC*, and *MTri* with edge spinning can be seen in Figure 2.6.

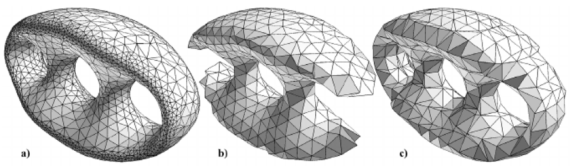


Figure 2.6: A genus 3 surface polygonized with (a) edge spinning, (b) *MTri*, and (c) *MC*, respectively.

The next family of solutions was started in [van Overveld and Wyvill, 2004] and [Stander and Hart, 1997]. Though their approaches use a similar idea, the actual method is reversed between them. [van Overveld and Wyvill, 2004] begins with a

sphere that encompasses the entire surface. The sphere is then iteratively projected onto the surface, subdividing as required. Similar to how [Galin and Akkouche, 2000] used the Lipschitz constant to determine when to stop subdividing the octree, [van Overveld and Wyvill, 2004] uses it to determine when to stop subdividing the mesh. The original algorithm, called *Shrinkwrap*, could only be used for surfaces of genus 0. This was corrected in [Bottino et al., 1996] where meshes of arbitrary topology could be polygonized.

While Shrinkwrap works by iteratively projecting a sphere onto a surface, [Stander and Hart, 1997] works in reverse. By placing spheres inside the surface where critical points exist, they could then be "blown up" to approximate the final surface. This approach did not have the genus 0 limitation of Shrinkwrap, though neither could capture sharp details very well.

The final set of algorithms we will discuss are particle-based methods. The general idea is to lay particles on the surface and allowing them to scatter following a given set of rules [Witkin and Heckbert, 2005]. The end result is that particles will cluster in areas of greater curvature. These can then be used to construct the triangles for the mesh. This exact approach is used in [Liu et al., 2005]. After triangulation, the algorithm can further subdivide the mesh if required.

[Rsch et al., 1997] uses a hybrid approach. The method works by generating an initial mesh with particles similar to [Liu et al., 2005]. This mesh is then fed as an implicit surface to *MC* so it can be polygonized. The resulting mesh is then remeshed to improve triangle quality. Unfortunately, this approach is highly tailored to work on algebraic surfaces, and as such is not easily extended to more general applications.

Chapter 3

Comparing *MC* to Bsoid

As we have discussed in chapter 2, many techniques in polygonization is based on the idea of improving some part of *MC*. With the objectives presented in section 1.3, we examine the original algorithms to see if there is a way of improving them in a different way that has yet to be explored.

As discussed in chapter 2, there are three original algorithms for polygonizing implicit surfaces: [Lorensen and Cline, 1987] (*MC*), [Wyvill et al., 1986] (Bsoid), and Bloomenthal's octree [Bloomenthal, 1988]. Furthermore, there are two real contributions from [Bloomenthal, 1988]: using an adaptive octree and dividing the cubes into tetrahedra. The latter results in a higher number of triangles than *MC* creates, and as a result is not ideal for our proposed solution. The concept of the octrees can be salvaged and adapted to fit in with the super-voxels presented in Bsoid. Hence, we will focus our attention to *MC* and Bsoid.

This chapter is divided into 3 sections: first we give a more in-depth analysis of both *MC* and Bsoid. Then we present a modern, parallel implementation of both, and finally we present the results from both algorithms on a set of surfaces for testing.

3.1 Analysis of *MC* and Bsoid

We begin our analysis with *MC*, since it is both the simplest in terms of overall structure of the algorithm as well as implementation, and the current industry standard. We will then follow with Bsoid, and will conclude with a brief comparison of both.

3.1.1 Marching Cubes (*MC*)

As previously stated in section 2.1, *MC* begins by taking the region of interest and dividing it into a uniform voxel grid, which we can define to be the bounding box of the model we wish to polygonize. The box is then divided into a cubical grid of specified size. While this results in non-cubical voxels, it does not affect the algorithm in any way. For simplicity, assume that the voxel grid is regular. That is to say, that the length, width, and depth of the box are divided into the same number of n steps.

The next step is to sample each vertex in the voxel grid and obtain its field value. There are a few points of consideration here. First, it is fairly evident that the runtime of this step is $O(n^3)$, since *every* vertex of every voxel must be sampled. This leads us to the second point: the field must be fully evaluated for every vertex. This presents us with two fundamental problems: time and memory complexity. We begin by examining time complexity with a simple example.

Suppose we are trying to polygonize a sphere. The equation is as follows:

$$f(x, y, z) = (x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - R^2$$

where (x_0, y_0, z_0) is the centre of the sphere, and R is its radius. In order to compute

the SDF for this sphere, we can reduce it to the following equation:

$$|\mathbf{x} - \mathbf{x_0}| - R$$

where $\mathbf{x}$ is the point we are sampling at, $\mathbf{x_0}$ is the centre of our sphere, and $|\ |$ is the length function. In order to evaluate this for a given point $\mathbf{x}$, we need 4 subtractions, 3 multiplications, and a square root for a total of 8 floating point operations. Suppose we are using a voxel grid with a resolution of 32^3, a modest resolution by today's standards. This means that in field samplings alone, we have to compute a total of $32^3 \cdot 8 = 262{,}144$ operations. This number increases proportionally with the number of fields that need to be evaluated, the number of operations that need to be performed per field, and cubically with the size of the voxel grid.

By design, *MC* requires the entire grid to exist in memory, which means that it also has a complexity of $O(n^3)$. While this is manageable with modern architectures, it can result in performance hits in the form of page faults, though this can be minimized with architecture-specific optimizations. As a final remark, notice that the entire processing of the vertices of the grid is trivially parallelizable, since the results do not interfere with one another. Furthermore, the algorithm necessitates no locking in its operation, since each write occurs to a distinct memory address (assuming that the entire grid is created from the onset) and the implicit model itself is treated as purely read-only memory.

Once the vertices have all been processed, the next step is to iterate over each voxel in the grid and compute its index. This is calculated by taking an 8-bit unsigned integer and flipping each bit depending on whether that particular corner of the voxel is inside the surface or not. Once this is done, the index is used with the look-up table shown in Figure 3.1.

This yields the number of triangles that need to be generated for that voxel.

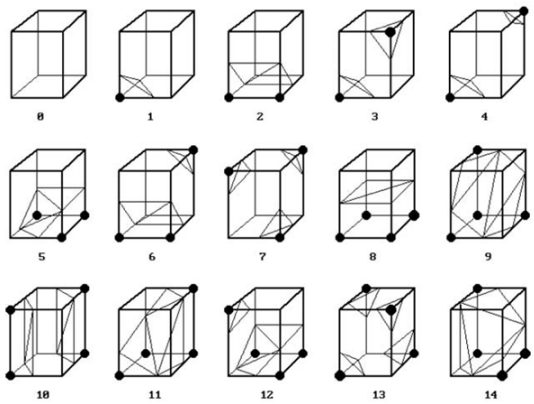


Figure 3.1: The 15 cases in the *MC* look-up table.

The vertices of the triangles are placed along the edge of the voxel, using linear interpolation between the field values at the corners. Once this process is done, the result is a triangle soup which can then be processed into a mesh.

MC on its own does not generate a triangle mesh, though the modification to change this is very simple and can be inserted at the end. Regardless, the second stage requires no more field evaluations, save maybe for the gradient of the computed points if we require accurate normals, and is also trivially parallelizable, since each voxel is purely independent from one another. This stage does necessitate locking in order to write the final triangles, though it is possible to avoid this depending on how the final buffer containing the triangles is set up at the cost of wasted space.

3.1.2 Bsoid

As previously explained in chapter 2, Bsoid has some degree of similarity with *MC*, though in reality they only have two things in common: they both use a voxel grid, and they both generate the triangles based on the values on the corners of the voxels. This is as far as the similarities go, however, as Bsoid uses other techniques to amortize both the runtime and memory usage.

Bsoid begins by dividing the region of interest into super-voxels. In a very broad

sense, these can be viewed as subdivisions of the bounding box of the model into a structure similar to an octree. After the super-voxels are formed, each one of them is processed and assigned a list of fields that influence it by checking whether the area of influence of the each field is contained (partially or otherwise) by the super-voxel. Any super-voxels that do not contain anything are discarded.

There are a few things to discuss about the super-voxels. The first is that while it may be more convenient to align the super-voxels to the voxel grid, it is by no means necessary, though as we will see later on it does simplify things to do so. Second, the time complexity of the super-voxels is $O(m^3)$, where m is the resolution of the super-voxel grid. Admittedly this is a rough estimate as the super-voxels can be made to have dimensions very different from those of the voxel grid itself, but for the sake of simplicity it is often easier to have the super-voxels be multiples of the voxels themselves. The assigning of the fields is linear on the number of fields. The actual checks can be optimized using bounding volumes, but at worst its just a check for distance to their skeletal points. Finally, notice that the super-voxels can be generated in parallel in a way similar to how the voxels in MC are calculated. This stage is trivially parallelizable and requires at most one lock, depending on implementation. Once the super-voxels have been generated, the next step is to find the voxels that contain the skeletal points, which we will call "seed" voxels. In the original design of Bsoid, these are actually given by the skeletal points themselves, though as we will see later this approach is impractical, as it requires the primitives having prior knowledge of the grid. Once the seed voxels are obtained, we must then find a voxel that crosses the surface.

The process for finding the surface starting from the seed voxels, and indeed the step that follows after this, is the same as breadth-first search (BFS). For every neighbour of each seed-voxel, we check the 8 corners to see if any cross the surface. If

the voxel is still inside, all of its neightbours are added to the queue and the process continues until one voxel crosses the surface. Once this occurs, we continue the search from the neightbours of this one voxel, "marching" the voxels across the surface until we have covered it all, which occurs when the *BFS* queue is empty.

We previously discussed the usage of a hash table to ensure that no voxels are re-computed. *MC* computes a field value for every vertex in the grid, which can result in a large performance hit if the field computation is too expensive. The first thing that Bsoid does to amortize this is to use the previously computed super-voxels. Given a voxel, we first check which super-voxel contains it. If we align the super-voxel grid to the voxel grid, then this is trivial. Once we have the super-voxel, we simply sample the fields that influence it, instead of sampling the whole field, which can reduce computations. Ultimately this approach still shares the same worst case, but on average field evaluations do decrease.

Once the voxel is computed, we then store it in the aforementioned hash table. Before we discuss the hash function, we first need to look at the way voxels are represented in Bsoid. Unlike *MC*, where the entire grid is stored, Bsoid simply refers to the voxels by an index corresponding to the coordinate of the lower-left-back corner of the voxel. This allows Bsoid to forego generating the grid, and simply moving the voxels around by increasing or decreasing their indices as needed. Since the voxel index is represented with integers, the hash function is then constructed by packing the x, y, and z coordinates into a single unsigned integer. This function ensures uniqueness of the hash indices up to a grid size of $2^{b/3}$ where b is the number of bits that we are using for the hash. For example, if we use 32-bit integers as our hash indices, then the function guarantees uniqueness for grids up to $2^{10} = 1024$.

By storing the voxels in our hash table, we have a constant look-up time. Furthermore, this allows us to prevent the re-evaluation of a vertex we have already seen.

This all produces an amortized $O(n_f)$ runtime where n_f is the number of voxels that actually contain part of the surface. In the worst-case, $n_f = n^3$ and Bsoid becomes *MC*, but in practice this very rarely happens outside of applications like fluid simulations. Likewise, the memory complexity of Bsoid is $O(n_f + n_s)$, where n_s is the number of voxels searched prior to breaking the surface, though these voxels can be discarded.

Once the list of voxels has been assembled, Bsoid then processes each voxel to determine the triangles that need to be generated. This step is almost identical to *MC* save for one major difference: once a triangle vertex has been computed, it is added to a second hash table. This time, the hash function merely concatenates the two hashes of the voxels whose vertices conform the edge where the new vertex is placed. Provided the hashes for the voxels are unique, this one will be as well. This gives two advantages: first, vertex are not re-computed every time. Second, because we know that the vertex has already been calculated, and we can also know the index of where the vertex is stored in the final list, it enables Bsoid to generate a mesh. This means that, unlike *MC*, no further processing needs to be done on the final output. It is also worth noting that this stage (save for the final writing of the vertices) is also trivially parallelizable.

Now that we have a better understanding of these two algorithms, we proceed to their respective implementation details.

3.2 Implementation for *MC* and Bsoid

This chapter will be divided into two sections describing the implementation details for *MC* and Bsoid, respectively. Before we begin, there are some implementation details common to both that need to be covered. Both algorithms were implemented

using C++ and compiled using the Intel Compiler 18.0. The libraries used were:

- STL: used for general purpose containers, specifically `std::vector`, `std::map`, `std::unordered_map`, amongst others.
- Intel Thread Building Blocks: used for parallelization of the algorithms. Specifically `tbb::parallel_for`.
- Atlas: used for general support for math libraries as well as graphics front-ends. See [Rovira Galvez, 2015] for details.

3.2.1 *MC*: Implementation

We begin our discussion with *MC*. From section 3.1, *MC* consists of two distinct stages: the generation of the voxel grid, and the computation of the triangle soup. We also described how both of these stages are not only trivially parallelizable, but require no locking mechanisms save for the final writing of the triangles. As such, the implementation of *MC* is very straight-forward and has little complexity to it. Due to this, we will focus on how the algorithm is parallelized, as opposed to the actual code itself.

Due to the simplistic nature of *MC*, there are a wide-variety of parallelization schemes, from simple threads [Miguet and Nicod, 1995; Shirazian et al., 2012; Hansen and Hinker, 1992] to GPU parallelization [Johansson and Carr, 2006; Dyken et al., 2008; Goetz et al., 2005]. In this case, we opt for CPU-based parallelization, since Bsoid cannot be easily implemented on the GPU, as we will see in the next section, and this way it remains a fair comparison. With this in mind, we employ a task-based parallelization scheme, as opposed to a thread-based one.

In a thread-based scheme (*TBS*), we would essentially divide sections of the voxel grid amongst the threads, each one executing a either a triple or double-nested for-

loop depending on the choice of division for the grid. There are some downsides to this approach. The first is that the optimal number of threads is limited by the number of available cores. This idea works well in the GPU, as it has the advantage of having a significantly higher number of cores, which increases the optimal number of threads it can run simultaneously. The CPU on the other hand is far more limited, with our architecture running 8 cores. More importantly however, *TBS* still assigns a relatively heavy work load to each thread, which means that each thread spends more time performing computations and makes it more difficult to schedule them and divide the CPU cores effectively. More importantly, since it's possible for some evaluations to take less time, it is plausible that a thread would finish early, meaning that the core running that thread is now idle, which hinders performance.

The task-based scheme completely eliminates all of the aforementioned problems. The idea of threads is replaced with a series of tasks, all of which can be executed in parallel. The scheduler is then in charge of distributing and assigning tasks to the threads (and therefore cores), thereby ensuring that no core is ever idle. With this in mind, we utilize `tbb::parallel_for` and its built-in scheduler to assign all of the grid vertices and ensure they are computed as fast as possible. This leads to better performance and better utilization of CPU resources.

This idea is applied to both stages. For the first stage, the smallest task that can be performed is the evaluation of a single voxel vertex. Using `tbb::parallel_for`, we assign each vertex to a task and let the scheduler handle the rest. For the second part, the task unit is each voxel of the grid. Again, we assign a voxel to a task, and the scheduler decides how to handle the load. There is only a minor point of consideration here, which is that once the triangles have been computed, they must be written to the same buffer. In order to avoid data corruption, we simply use an `std::mutex` to guard the triangle buffer, thereby guaranteeing that the results are

correctly written.

3.2.2 Bsoid: Implementation

Unlike *MC*, Bsoid is an algorithm that is more complex to both implement correctly as well as parallelize effectively, as we will now discuss. For the most part we will focus on both the theoretical and parallelization aspects of the implementation as opposed to the code itself.

The process of generating the super-voxels is fairly straight-forward and in fact can be implemented in a very similar fashion to the grid generation step of *MC*. The sole difference is that we need to use a lock to guard the final list of super-voxels, since we have no way of knowing beforehand how many of them will actually contain parts of the surface. That being said though, there is a point of interest in how we optimize the computation of which skeletal points belong to a given super-voxel. When we create a new implicit surface, we can also define a bounding box for it. The important part here is to make sure that said box is big enough to not just cover the surface itself, but also the full region of influence of the field. Since we are using compact support, it is simply a matter of expanding the box by the radius of influence. Once we have the bounding volumes of each skeletal point, we combine them together to form a BVH, which in turn optimizes the generation of super-voxels, as intersecting with a bounding box is significantly faster than doing distance computations.

The next step is to find the seed voxels. In the original implementation of Bsoid, the skeletal points themselves provide the voxels. This assumes that the skeletal points (i.e. the fields themselves) have prior knowledge of not just the grid, but also its dimensions in order to provide a seed voxel. This is impractical from an implementation standpoint, as usually the model is created before the dimensions of the grid are assigned. Furthermore, it breaks the levels of encapsulation that would

otherwise be in place, as the fields themselves do not need to know of anything beyond how to be evaluated at specific points. Instead, we propose to have the skeletal points simply return a point that sits on their surface. The fact that this point may not necessarily be on the *final* surface after all transformations and operators have been taken into account is not relevant since the points only provide a starting point to begin the search. The seed points are then transformed into seed voxels by simply computing which cell they fit in.

Once the seed voxels are calculated, we must then guarantee that they are on the surface. This is required since operations like difference or intersection can "remove" parts of the original surfaces on which these seed voxels may have been. In the original Bsoid, this was accomplished by BFS on all the neighbours. This leads to a high number of computations depending on the number of voxels that are searched, which depends heavily on how far (in or out) of the surface the seed voxel is. We propose a different method which employs the gradient ∇ of the surface, which points in the direction of greatest change. Furthermore, since we are using compact support, the gradient will *always* point towards the surface even if the point we are sampling at is outside the surface [Ju et al., 2002; Wyvill and van Overveld, 1996; Akkouche and Galin, 2001]. With this in mind, we can simply sample the gradient at the centre of the voxel, and the next voxel we check is determined by selecting the one to which the gradient points. This is accomplished by taking the sign of the components of the gradient and adding them as an offset to the index of the current voxel. The process is repeated until the voxel contains part of the surface. This enables us to take the most direct route to the surface and avoid having to compute more voxels than we absolutely have to. The algorithm is shown in Algorithm 1.

This last step can be done in parallel, along with the computation of the seed voxels. Fortunately, since the number of seed points, and therefore seed voxels, is

Algorithm 1 The process for marching a seed voxel to the surface.

procedure FINDSURFACE(v_s)
  $found \leftarrow 0$
  $v_c \leftarrow v_s$
  while $found \neq 1$ **do**
    $\mathbf{p}_c \leftarrow 2 \cdot v_c + (1, 1, 1)$ ▷ Find the centre of the current voxel.
    $\mathbf{o} \leftarrow convertToCell(p_c, 2n_v)$
    $o_f \leftarrow S(\mathbf{o})$
    $\mathbf{n} \leftarrow \nabla S(\mathbf{o})/|n|$
    if $o_f > 0$ **then**
      $\mathbf{n} \leftarrow -\mathbf{n}$ ▷ Flip normal if outside the surface.
    end if
    $n_s \leftarrow sign(\mathbf{n})$
    $v_c \leftarrow v_c + \mathbf{n}_s$ ▷ Increment to next voxel.
    if $validVoxel(v_c) = 0$ **then** ▷ Check if voxel is inside the grid.
      break
    end if
    if $containsSurface(v_c) = 1$ **then** ▷ Check if voxel does contain the surface.
      break
    end if
  end while
  return v_c
end procedure

known beforehand, there is no need for locking. The only part where locking does become a necessity is when we are computing the voxels as we find the surface. This is tied to the hash table that will be used in the following step, since this is shared memory. Each voxel we compute is added to the hash table, thereby requiring a lock to prevent data races. Arguably, these voxels are (for the most part) irrelevant as they can be either too far out from the surface or too far in. While this may be true in certain cases, it is certainly not true for all, and it is worth storing these extra voxels if some of them end up being needed when we start marching on the surface and can hence prevent repeated computations.

This leads us to the marching of the voxel on the surface to generate the list of voxels that contain it. Since this part is essentially BFS, there is very little that can be parallelized here. In fact, the only part that can be parallelized is the checking of the neighbours of a given voxel, since the main body of the algorithm is a while-loop that cannot be easily (or efficiently) converted into parallel code. That said, there are some considerations at this stage that do come into play. The first is how the hash table is handled. In the original implementation, the hash table was essentially a large array, with sufficient space to handle the maximum hash index that could be generated. This resulted in large parts of the array being empty, thereby wasting memory. Instead, we propose the use of an `std::unordered_map` to act as our hash table, since this will only allocate as much space as is needed and nothing more.

The last point leads to the second thing to consider: the size of the grid itself. The hash function works by packing the x, y, and z coordinates into a single unsigned integer, which meant that the maximum grid size was $2^{b/3}$, where b was the number of bits the hash was to have. Originally, b was set to 32, and that yields a maximum grid size of 1024. This, unfortunately, is not sufficient, as modern computers (and therefore *MC*) can allocate significantly larger blocks of memory. To remedy this, we

propose a 64-bit hash, which has a maximum size of $2^{21} = 2097152$. This is more than sufficient to handle even the finest of grids, which very rarely exceed dimensions of 2048. This does introduce a minor problem in the following step.

The final stage of Bsoid takes the generated list of voxels and computes the triangles for each. This stage is almost identical to that of *MC* and can be implemented the same way. The difference is in the existence of the second hash table that holds the computed vertices for the triangles, which does require an additional lock. The minor issue mentioned before is a consequence of the choice of size for the hash keys. Since the voxel hashes are 64-bits wide, then by necessity, the edge hashes which are used to store the computed vertices, must be 128-bits wide. This is due to how the hashes are computed, by simply concatenating the voxel hashes of the two end-points. This is shown in Figure 3.2. Now it is true that the Intel Compiler offers a 128-bit intrinsic this could be used to store the edge hash. This leads to several complications due to the functions that are offered to manipulate the intrinsic. Instead, we prefer to simply create a new structure that holds both hashes, adding the necessary operations to it so it can be used in our hash table. While this may result in a minor performance hit, the trade-off is acceptable, since the alternative would require more development time than is really necessary.

With these implementation details in mind, we now discuss the results of the comparison between both algorithms.

3.3 Tests and Results

We now compare the two algorithms with the aforementioned implementations. The code was run on an Intel Core i7-4900MQ running at 3.8GHz with 16GB of RAM and Windows 10 as its operating system. The results that we will compare are the

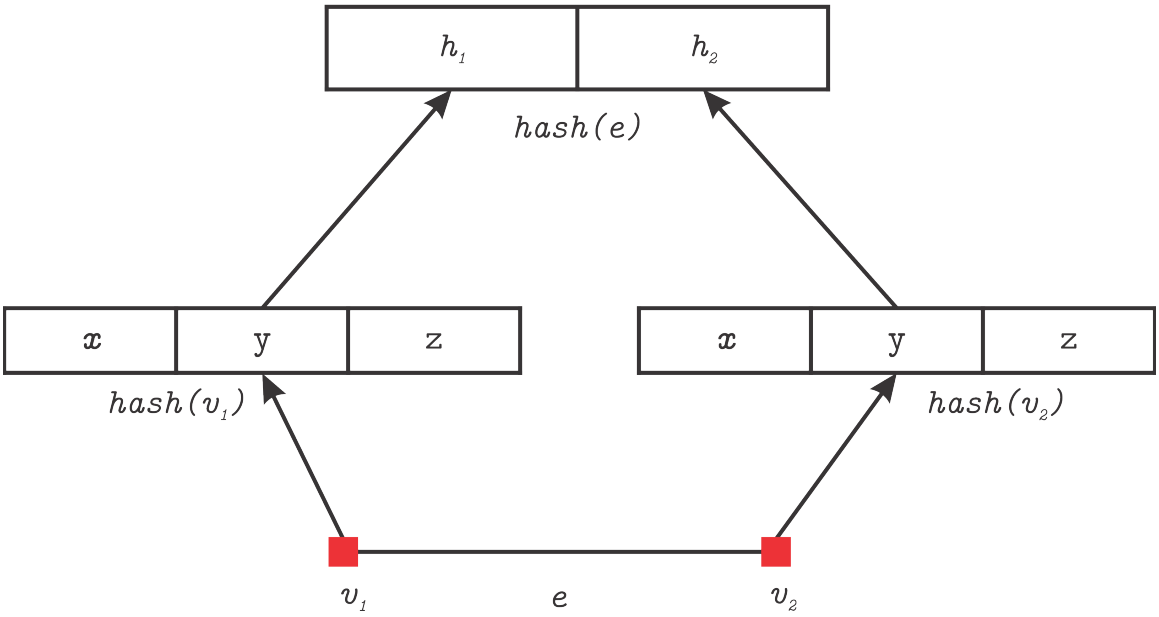


Figure 3.2: The structure of the hash used for edges. Notice how an edge hash is double the size of a single vertex hash.

following:

- Time: the amount of time it takes for the algorithm to produce its *final* output. In the case of *MC*, we stop the clock when we obtain the triangle soup, as the conversion into a triangle mesh is not part of the original algorithm.

- Memory: the total amount of memory that the algorithm requires. This does not include mutexes, auxiliary variables, or extra parameters. In the case of *MC* we focus mostly on the size of the grid and the final mesh, while in Bsoid we focus on the number of voxels, the hash tables, as well as the lists of triangles.

- Field evaluations: this will encompass the total number of evaluations per field, as well as the final count from all fields used in the model. It is worth noting that the fields referred to here are the skeletal points, not the operator fields.

The tests were conducted on a set of models that include the following:

- A unit sphere centred at the origin,

- a torus with inner radius of 3 and tube radius of 1,
- two blended spheres (peanut),
- a butterfly composed of spheres and tori, and
- 100 randomly placed particles represented as spheres.

The models were repeatedly polygonized with different grid resolutions, starting at 8^3 and ending in 512^3. Since we force the super-voxels to be aligned to the voxel grid, sizes must increase in even steps, so the we add 2 at each iteration. The final grid size for step i is then computed as $(8 + 2i)^3$. For the sake of simplicity, we will refer to the resolution of each step by it's index. So grid resolution 0 is then a size of 8^3, while grid resolution 252 is a size of 512^3. The super-voxel grid is computed by dividing the current grid size by 4. Admittedly this does not produce the most optimal super-voxel to voxel ratio, but that must be determined on a case by case basis, and this formula provides a reasonable balance on performance.

Since all the results are very similar, we will focus on the results for the particles model, as it is the most complex shape that we tested and provides the best illustration of the differences between algorithms. Additionally, we will present the results for each model on the highest resolution that we tested for comparison. We now examine each of the aforementioned categories, after which we will discuss the conclusions from these results.

3.3.1 Execution Time

We can see in Figure 3.3 the comparison between *MC* and Bsoid in terms of execution time. In Table 3.1 we can see the comparison between all 5 models. The performance gain shown in this table is computed as T_{MC}/T_{Bsoid}.

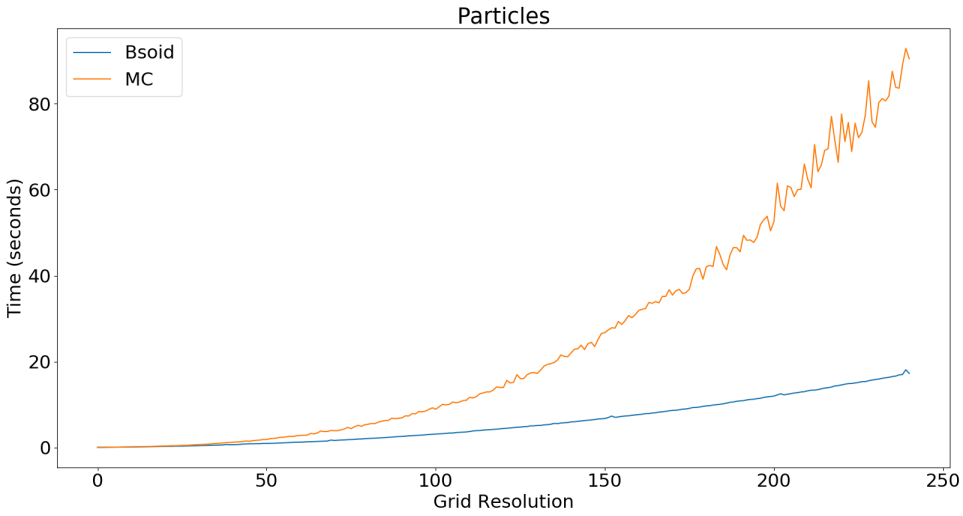


Figure 3.3: Comparison of execution times between *MC* and Bsoid.

Model	**Bsoid**	***MC***	**Gain**
Sphere	9.17442	17.0924	1.8630
Torus	7.21247	14.0235	1.9443
Blended spheres	9.17442	17.0924	1.8630
Butterfly	12.9546	29.7835	2.2990
Random spheres	17.2677	90.468	5.2391

Table 3.1: Comparison of execution times between Bsoid and *MC*. Times in seconds.

For grid resolutions under 10^3, *MC* is faster than Bsoid, though the difference decreases as we approach this number. As soon as the grid resolution exceeds this boundary, Bsoid is faster than *MC*, and in fact grows at a significantly lower rate than *MC* does.

The reason for why *MC* is faster than Bsoid on smaller grids depends on two important factors. The first is that Bsoid has to do more steps in order to reach the voxel processing stage than *MC*, as well as having to maintain more structures when generating the triangles. Provided the grid is "small" enough, *MC* is able to finish both steps before Bsoid has had a time to finish. If we were to look at the breakdown of time distributions, we would realize that Bsoid's bottleneck occurs after the super-

voxels are being created. This is due to the nature of the BFS algorithm that is used to find the voxels that contain the surface.

The second reason has to do with the rate at which *MC*'s memory usage grows. As we will see in subsection 3.3.2, the memory that *MC* requires grows very quickly, which reduces the percentage of the voxel grid that can be held in memory at any one time. This, coupled with the way in which the schedule could assign tasks to each of the threads, could lead to page faults in each thread, which results in time being spent in memory accesses instead of computation, thereby slowing down the process. In addition, while both Bsoid and *MC* suffer from a large number of voxels that need to be searched, Bsoid always has a lower number, as it only needs to search through specific areas, as opposed to *MC* that has to search through all of the voxels in the grid.

3.3.2 Memory Usage

We see in Figure 3.4 the comparison between the memory usage of *MC* and Bsoid. In Table 3.2 we see the comparison across all models.

Model	**Bsoid**	***MC***
Sphere	228.37	2199.18
Torus	449.10	2269.83
Blended spheres	293.71	2220.01
Butterfly	371.61	2246.47
Random spheres	508.62	2288.91

Table 3.2: Comparison of memory use between Bsoid and *MC*. Sizes in MB.

Once again, we can see that *MC* uses less memory than Bsoid for smaller grids, up to 30^3. The smaller number makes sense since *MC*'s memory usage grows cubically on n for the size of the grid. While it is true that initially Bsoid requires more space due to the hash tables and the lists of super-voxels, these are dwarfed very quickly by

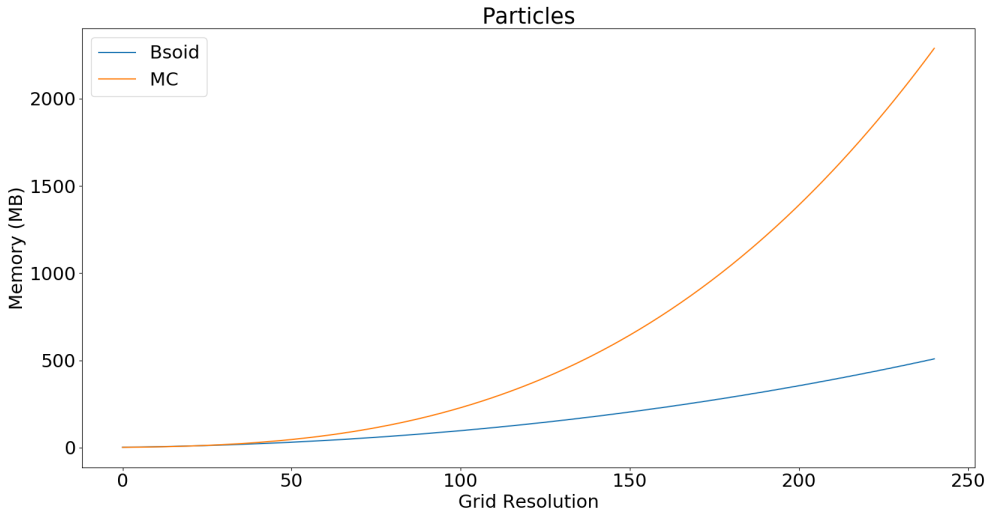


Figure 3.4: Comparison of memory usage between *MC* and Bsoid.

the memory needed to maintain the entire grid in memory as *MC* necessitates. As mentioned previously, this high memory use results in poor cache coherency depending on how the scheduler assigns the task to the threads. Even if the threads were all guaranteed to work in continuous blocks of memory, there isn't enough space to hold the entire grid, especially towards the higher end of the grid resolutions where the size easily exceeds 2GB. This means that after each block is done, it must be offloaded and the new block loaded into the thread's memory space. Ultimately this results in a lot of time spent in memory accesses instead of computing.

3.3.3 Field Evaluations

We see in Figure 3.5 the comparison between the number of field evaluations of *MC* and Bsoid. In Table 3.3 we see the results from all models.

As depicted in Figure 3.5 and Table 3.3, the usage of super-voxels greatly reduces the number of total field evaluations by only evaluating the fields that need to be evaluated depending on the region. Furthermore, Bsoid's usage of BFS to only

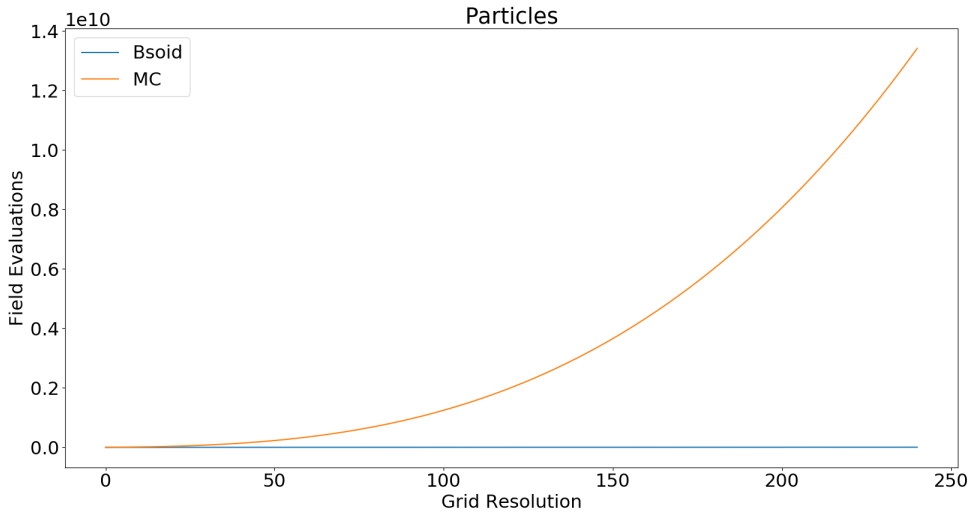


Figure 3.5: Comparison of total field evaluations between *MC* and Bsoid.

Model	**Bsoid**	***MC***
Sphere	9.79×10^5	1.34×10^8
Torus	2.33×10^6	1.34×10^8
Blended spheres	2.03×10^6	2.68×10^8
Butterfly	7.18×10^6	9.40×10^8
Random spheres	4.37×10^6	1.34×10^{10}

Table 3.3: Comparison of field evaluations between Bsoid and *MC*.

compute the voxels that contain the surface as opposed to all of them also helps to greatly reduce the number of total evaluations. From the table we can see that Bsoid evaluates at least 2 orders of magnitude less than MC.

A point of interest arises from this comparison, which could be considered another measure in performance. We define the *field evaluations per vertex* (FPV) as the number of field evaluations required to generate a single vertex of the final mesh as per the following equation:

$$FPV = \frac{V_{tot}}{F_{tot}}$$

where V_{tot} is the total number of vertices generated and F_{tot} is the total number of skeletal field evaluations. In an ideal situation, we would want $FPV = 1$, but we will

declare an algorithm "better" if it's FPV is closer to 1. With this in mind, we see a comparison of the FPV's of MC and Bsoid in Figure 3.6. In Table 3.4 we see the comparison across all models.

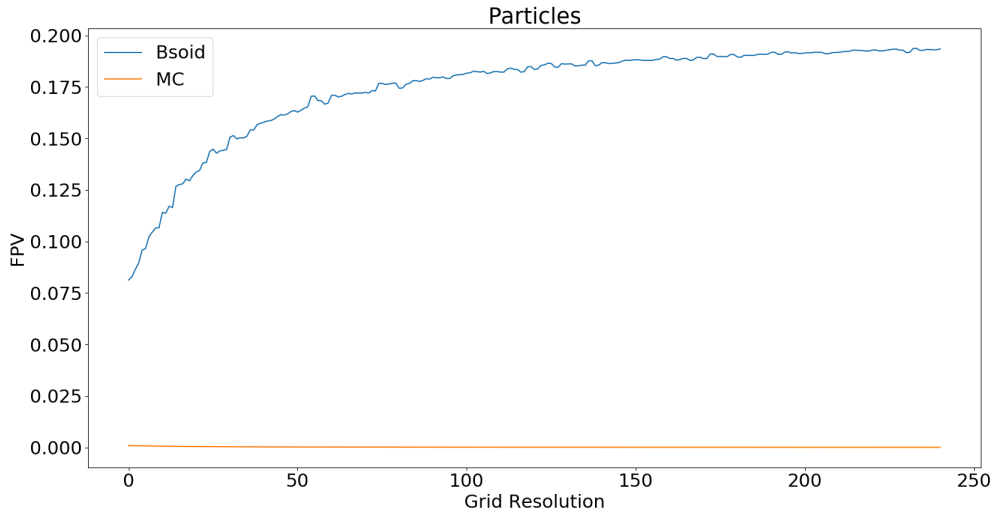


Figure 3.6: Comparison of FPV between MC and Bsoid.

Model	**Bsoid**	***MC***
Sphere	0.315 15	0.002 29
Torus	0.313 19	0.005 44
Blended spheres	0.213 65	0.001 61
Butterfly	0.081 85	0.000 63
Random spheres	0.193	6.30×10^{-5}

Table 3.4: Comparison of FPV between Bsoid and MC.

As can be expected from Figure 3.5, Bsoid has a much higher FPV than MC for the same reasons. While it is true that Bsoid is still very far from 1, it's performance is still significantly higher than MC, whose FPV tends to be near 0 regardless of the grid resolution.

3.3.4 Conclusion

Based on the results seen on the previous sections, we conclude that Bsoid is a good starting point for our new algorithm. The reasons are as follows:

- While it is true that Bsoid is more difficult to parallelize than *MC*, we have seen that it outperforms *MC* for finer grids. This allows for much accurate sampling of the surface while still retaining speed. It also allows for more computation time spent on improving the quality of the final mesh. With careful implementation and optimizations we can create an algorithm that is slower than Bsoid but still faster than *MC*.

- The reduction in field evaluations is closely related to the previous point, as they are the major performance point of both algorithms. By utilizing Bsoid's super-voxels and BFS structure we can spend more time on the generated vertices and refining the mesh.

- The lower memory usage of Bsoid allows the processing of much finer meshes than *MC* would permit while still maintaining performance.

Chapter 4

Marching Rings

As stated in section 1.3, we would like our new polygonizer to:

1. Be as fast or faster than *MC*,
2. Accurately represent surface topology, and
3. Produce a mesh of better quality than *MC*.

As we concluded in chapter 3, by starting with Bsoid we already satisfy point 1 of the above list. In addition, we also ensure that the result is a mesh, as opposed to a triangle soup. Indeed, thanks to the performance improvement that Bsoid has over *MC*, it is possible to spend more time processing the final mesh and ensuring it is of higher quality as opposed to having that be required after the polygonization process is complete. With this in mind, we now present our new algorithm, called *Marching Rings* (*MR*). We first discuss the theoretical background of *MR* and then move on to the implementation details. We will conclude with results, further work, and closing remarks.

4.1 The Theory of Marching Rings

Let S be the surface that we wish to polygonize and B be it's bounding box. Now both *MC* and Bsoid (indirectly) divide this box into a grid of voxels, which are then used to generate the triangles. As we have seen in chapter 2, this does have some problems in terms of ambiguities and topological inaccuracies. Suppose that instead of dividing B into a voxel grid, we first use π_n planes and intersect them with S. This would result in each plane π_i that does intersect the surface having at least one contour tracing the intersection between S and the plane. Following this idea, we are left with a stack of planar cross-sections that approximate the surface. To connect them, all that we would need would be for all the contours to be subdivided into the same number of vertices. This guarantees a 1-1 mapping between the vertices of each contour, which results in a mesh formed by regular quads. This is the general idea behind *MR*. The full algorithm is presented in Algorithm 2.

Algorithm 2 Marching Rings Polygonizer

```
1:  procedure MarchingRings(S, n_π)
2:      B ← S                                   ▷ Bounding box of the surface.
3:      Π ← makePlanes(B, n_π)                  ▷ Construct planes for slicing.
4:      m_v ← −inf
5:      for π ∈ Π do
6:          constructCrossSections(π)
7:          m_v ← max(m_v, π)      ▷ Keep track of the largest number of vertices in a
    contour.
8:      end for
9:      for π ∈ Π do
10:         for C ∈ π do                        ▷ For each contour in π.
11:             resize(C, m_v)                  ▷ Resize all contours
12:         end for
13:     end for
14:     m ← connectContours(Π)
15:     return m
16: end procedure
```

We can summarize the algorithm of *MR* by splitting it into the following steps:

1. Create planes,
2. compute contours,
3. resize contours,
4. connect contours.

We will now examine each stage in more detail.

4.1.1 Creating the Planes

A plane π can be defined by a normal $\mathbf{n}$ and a point $\mathbf{p}$. In order to construct our planes, we restrict $\mathbf{n}$ to be one of the canonical axes. Notice that this is not much of a restriction, since we can easily transform an arbitrary axis to align with a canonical axis. The next step is to determine the distance between planes. This parameter can be set with n_π or by manually specifying the δ between planes. Either way, after this is done we are left with a stack of planes.

Next, we must be able to compute the contours that will result of the intersection between S and each plane. This is accomplished by performing Bsoid in 2D. For this, we execute per plane the computation of super-voxels. These are once again, aligned to the 2D voxel grid that will subdivide each plane. The computation of the super-voxels concludes this stage, which leads us to the generation of the contours.

4.1.2 Computation of Contours

The process here is identical to that of Bsoid, the only difference being that we are using 2-dimensional voxels instead of the classical 3-dimensions. This enables us to remove the natural ambiguities that arise in traditional 3D polygonization, while still

retaining the speed and efficiency of Bsoid. The only major change comes from the way that we obtain the seed voxels.

In order to make the conversion from the algorithm shown in Algorithm 1 which is in 3D to 2D, we must first project each skeletal point to the current plane that we are in, and then convert it into its corresponding voxel. The rest of the algorithm proceeds as described in chapter 3. Once the seed voxels are computed, we use BFS in the exact same way as Bsoid does, using the hash table to avoid having to re-compute seen voxels.

The collapse to 2D brings with it an advantage over Bsoid in terms of the grid sizes that the hash tables can handle. Bsoid's voxel hash is computed by packing each one of x, y, and z coordinates of a voxel into a single unsigned integer, dividing its bits into 3 groups. This limited the maximum grid size to $2^{b/3}$. It also has the side effect of wasting bits at the end. In 2D however, this hash is computed by simply packing x and y into the unsigned integer, effectively dividing the bits in half. This means that the maximum grid size that we can use this hash for is $2^{b/2}$. The implication is that we can represent larger grids with smaller hashes. So a 32-bit hash yields a maximum grid size of 2^{16} as opposed to Bsoid's 2^{10}. It also means that the edge hash that is used later on does not require any special data types, as we can simply use a 64-bit integer instead of Bsoid's 128-bit integer.

Once the list of voxels containing the contours has been obtained, the next step is to compute the line segments that constitute the contours themselves. Since we are in 2D, we use a simplified case table adapted from Marching Squares (the 2D counterpart of *MC*). After all voxels have been processed, we are left with a list of line segments. The calculation of the vertices of the line segments uses the same hash table that Bsoid had, the only difference being that we can use a smaller hash as a result of the lower dimensions.

The list of line segments must then be merged into their corresponding contours. This is accomplished by simply following their end-points. Care must be taken to ensure that distinct contours are not merged together by accident, but this is easily accomplished using an `std::map` and `std::unordered_set`. Once we are done, we have a list of contours, and we can easily recover the maximal number of vertices per contour, which leads to the next stage: subdivision.

4.1.3 Subdivision of Contours

In order to obtain our regular quad-mesh, we must first guarantee that all of the contours have the same number of vertices. This problem is solved by simply making all of the contours have the same number of vertices as the largest contour across all planes. This choice yields two significant advantages:

- By dividing the contours with lower vertex counts, we end up with a mesh that is significantly finer than any that Bsoid or *MC* could have produced at the same grid resolution.

- Due to the higher resolution in the contours, and by virtue of the subdivision scheme, the contours are able to more accurately represent surface topology.

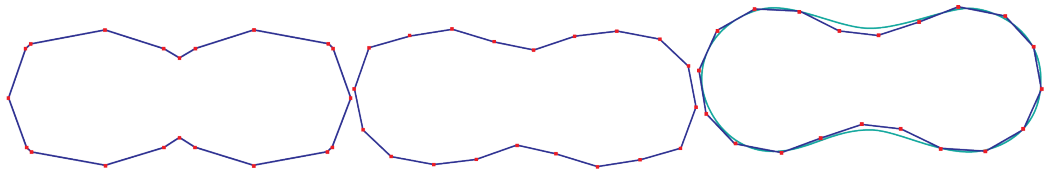
The subdivision is performed based on the approximated arc length of the contour. This is obtained by summing all of the lengths of the line segments that conform the contour, and then dividing by the number of vertices we obtained earlier. Once the contour has been divided, we then "push" each vertex so it sits on the surface. This allows us to ensure that the contour better represents the topology of S, taking advantage of the subdivided contour. The algorithm is presented in Algorithm 3.

The general idea behind this algorithm is the following: if the vertex $\mathbf{p}$ is not on S, then translate $\mathbf{p}$ along its gradient until it is outside the surface. The new vertex

Algorithm 3 Procedure for pushing vertices to the surface.

procedure PUSHTOSURFACE($\mathbf{p}, c_i, \delta$)
  if $\mathbf{p} \in S$ **then**
    return $\mathbf{p}$
  end if
  $in \leftarrow \mathbf{p}$
  $in_f \leftarrow S(in)$
  $\mathbf{n} \leftarrow \nabla S(in)$
  $\mathbf{n} \leftarrow \mathbf{n} - proj_{\mathbf{n}_\pi}\mathbf{n}$
  if $in_f > 0$ **then**
    $\mathbf{n} \leftarrow -\mathbf{n}$
  end if
  $\mathbf{n} \leftarrow \mathbf{n}/|\mathbf{n}|$
  $d \leftarrow \delta$
  $out \leftarrow in + (\delta * \mathbf{n})$
  $out_f \leftarrow S(out)$
  while $sign(out_f) = sign(in_f)$ **do**
    $d \leftarrow d + \delta$
    $out \leftarrow in + (d * \mathbf{n})$
    $out_f \leftarrow S(out)$
  end while
  $\mathbf{p}_n \leftarrow linear(in, out)$ ▷ Linear interpolation.
  return $\mathbf{p}_n$
end procedure

is then computed as the linear interpolation of both end-points. This is based on the same principle as binary search. In Figure 4.1 we can see the progression from the raw contour to the final subdivided contour.



(a) Initial contour. (b) Subdivided contour. (c) Contour vertices pushed to surface (in light blue).

Figure 4.1: A cross-section of two blended spheres. Notice how Figure 4.1c better captures the curves of the blend.

4.1.4 Connecting the Contours

The final stage is to connect each adjacent pair of cross-sections using quad-strips. Let π_i, π_{i+1} be two adjacent cross-sections and let C_i, C_{i+1} be the corresponding sets of discretized contours. By construction, we know that $\forall c \in C_i$, $|c| = v_{max}$ where v_{max} is the maximal number of vertices we computed earlier. The correspondence problem is then defined as finding a mapping $f : C_i \to C_{i+1}$ such that f accurately captures the underlying topology of S. We can see that there are 3 cases:

1. $C_i = \emptyset$ and $C_{i+1} \neq \emptyset$,
2. $|C_i| = |C_{i+1}|$, and
3. $|C_i| \neq |C_{i+1}|$.

We will refer to case 1 as the *cap* problem and case 3 as the *branching* problem. We will focus exclusively on case 2, while the rest will be discussed in section 4.2. We propose the following solution for case 2:

Solution 1. *Let v_{i_k} be the k-th contour of the i-th cross-section. Assume S is a C_1 continuous surface. Then,*

$$f(c_{i_k}) = c_{i+1_k}$$

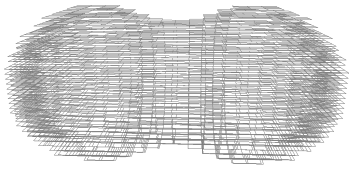
In other words, provided that S is *at least* C_1 continuous, then we can simply map the first contours of C_i with the first contour of C_{i+1}. The reason for restricting S to be C_1 continuous is to prevent cases where the "arms" of the surface could abruptly cross. Since we do not posses much information about what occurs in between cross-sections, we must therefore make stronger assumptions about the topology of the surface.

There are two minor considerations here. The first relates to the ordering of the vertices, which is based on the seed voxel that created the contour. While it is true that the seed point that created the voxel will be the same across all layers (if we were to fully draw out their grids), it is possible that depending on the distance of the voxel to the surface, we may end up starting in adjacent voxels across cross-sections. In other words, it is possible for the seed voxel v_s in π_i to be different from the corresponding v_s in π_{i+1}. What this will result is in the first vertices of corresponding contours being offset from each other, which will cause the final quad strip to become twisted. The solution to this is to simply grab the first vertex of the contour in πi and then look for the vertex with the shortest distance in the corresponding contour of π_{i+1}. This ensures that the quads are properly aligned and resolved the twisting problem.

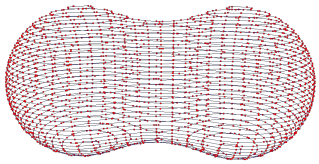
The second is the ordering of the contours themselves, which is somewhat guaranteed due to order in which the voxels (and therefore the line segments) will be generated across cross-sections, since the seed points that spawn the seed voxels are just projections on the corresponding plane and are therefore relatively close together. An alternative would be to use a similar criterion to the one employed to select the initial

vertices.

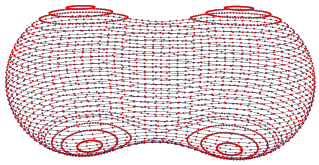
Once the appropriate contours have been matched, they are joined using a quad-strip. The process is repeated for every pair of cross-sections until the final mesh is formed. The full process on two blended spheres is illustrated in Figure 4.2.



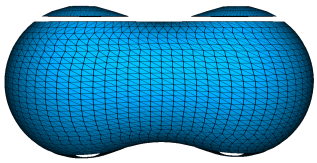
(a) The voxel lattice for each cross-section.



(b) The initial contours for each cross-section.



(c) The subdivided contours for each cross-section.



(d) The final quad-mesh.

Figure 4.2: The full polygonization process of *MR*.

4.2 Limitations of *MR*

As mentioned in the previous section, there are two cases of the correspondence problem that we did not cover, and we will explain why these constitute limitations to *MR*. These two cases are:

1. $C_i = \emptyset$ and $C_{i+1} \neq \emptyset$ (cap problem), and

2. $|C_i| \neq |C_{i+1}|$ (branching problem).

4.2.1 The Cap Problem

The cap problem arises when either one of the following conditions hold:

1. The first region of contact with the surface along the chosen axis lies between two cross-sections.

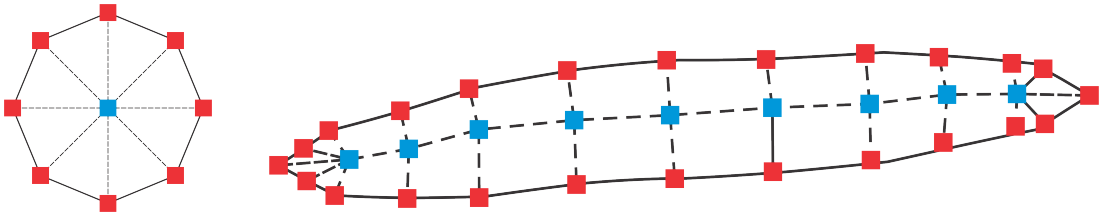
2. The region of contact is too small to be captured by the current resolution. This will always happen if the area of contact is a single point.

Due to the choice of using cross-sections, we lose information about the behaviour of S is in between the slices. In the first case, we simply do not posses enough information to predict where the point(s) should be placed. That being said, assuming that we had a way of knowing where these contours would be placed, the final strip of mesh would consist of either a triangle fan or a quad mesh joined by triangles at each end. This depends on the shape of the final contour. If it is a single vertex, then the only way to connect all other vertices together is by using a triangle fan. While this violates the requirement of having a quad mesh, and may introduce potentially skinny triangles, we believe that this trade-off is acceptable in favour of having a mostly quad-mesh. Additionally, it is possible to convert the triangle fan to quads if we allow for n-gons to be used instead.

The second case arises when the final contour is a curve. In this case, quad strips would be used to connect the vertices on either side of the curve, with triangles conforming the transition between both sides. These solutions are shown in Figure 4.3.

4.2.2 The Branching Problem

The branching problem arises when we have two cross-sections with unequal number of contours. The problem faced here can be summed up in the following question:



(a) Single-vertex cap.

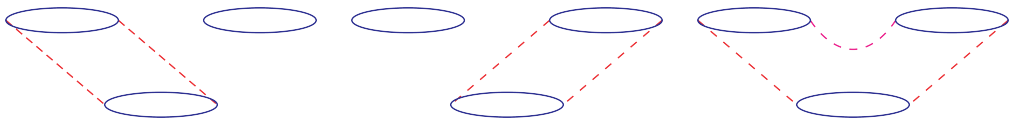
(b) Contour cap. Notice the triangles at either end of the contour.

Figure 4.3: The two possibilities for cap contours. Blue vertices represent the caps themselves, and dotted lines are the new polygons.

how do we know which, if any, contours need to be connected? Formally, the problem can be stated as follows:

Problem 2. *Given two adjacent cross-sections π_i, π_{i+1} with discretized contour sets C_i, C_{i+1} such that $|C_i| \neq |C_{i+1}|$, and neither of them are trivial, find a mapping $f : C_i \to C_{i+1}$.*

The problem is illustrated in Figure 4.4. This leads to the following:



(a) Case 1. (b) Case 2. (c) Case 3.

Figure 4.4: 3 possible configurations of branching. Notice all 3 yield the same cross-sections.

Claim 1. *As stated, the problem is under-constrained and therefore has infinitely many solutions.*

Proof. First, suppose that there are no solutions. Clearly, the trivial mapping where no contours are connected is an acceptable map. Now suppose that there is exactly one solution. We can classify the possible mappings f into 3 groups:

1. Each contour c_{i_k} is mapped to the corresponding contour c_{i+1_k}: construct a mapping f' such that any two contours in C_{i+1} converge in a single contour in

C_i. This mapping also corresponds to the same cross-sections and is therefore valid.

2. At least one n-group of contours in C_{i+1} converge in a contour of C_i: construct f' such that any of the contours in the n-group is mapped to ti's corresponding contour in C_i. This mapping also corresponds to the same cross-sections and is therefore valid.

3. No contours are matched: construct f' such that at least one contour is connected.

□

As a result, *MR* can only polygonize surfaces of genus 0 and along axes that do not incur in any branching. In Figure 4.5 we can see the result of attempting to polygonize a torus by two different axes. In both, the cap problem is present, however only one exhibits branching.

(a) Slicing along y-axis: cap and branching. (b) Slicing along z-axis: cap only.

Figure 4.5: Examples of slicing a torus by 2 different axes.

The final limitation comes by the construction of the contours and is an extension of the branching problem. Suppose that we have a way of producing a mapping that satisfies the above requirements. Furthermore, let's suppose that we have 2 contours in C_i and 1 in C_{i+1} and that both contours are joined with C_{i+1}. By construction,

both contours have n_{max} vertices, and so does the one in C_{i+1}. The problem then is this: construct a tiling that maximizes the number of quads and minimizes the number of triangles and connects $2n_{max}$ vertices with n_{max} vertices. In general, this tiling problem must connect kn_{max} vertices with ln_{max} vertices where $k \neq l$. This can be solved by the introduction of new segments, as we will discuss in the following section.

4.3 Future Work and Conclusions

In this section we will outline some avenues that may yield solutions to both the cap and branching problems. We will also discuss some performance and mesh quality results between *MC* and *MR*.

4.3.1 Solving Caps and Branching: The Shadow Field

We can also view the cap problem as follows: find the local maxima that is closest along the chosen axis for slicing. This formulation presents a new avenue for research: employing the gradient. By definition, ∇S points to the area of greatest change, which means that it will point towards the maxima that occurs along the axis of choice. Unfortunately, because S is in $\mathbb{R}^3$, we have no way of predicting which direction we have to walk in along the gradient to find said local maxima. Moreover, if we attempt to follow the gradients we would leave the known area that belongs to that particular cross-section, effectively walking on the "empty" space that exists between each cross-section. What we can do instead is to look at the *length* of the projection of the gradient onto our plane.

The idea is the following: we know that attempting to leave the cross-section would involve entering a space in which we have no information nor a direct way to

discretize without resulting in MC. So what we need to do is delay this until we have sufficient information about where to go. We also know that the gradient will point in the direction of greatest change. More importantly, the gradient vanishes at the maxima, which is what we are looking for. So we do the following: for each grid point in our cross-section, take the length of the projected gradient. This defines a new implicit function, which we call the *shadow* function since it yields the shadows of where the maxima and minima occur. An example is shown in Figure 4.6.

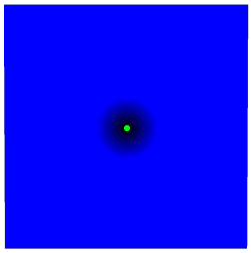


Figure 4.6: The shadow field for a sphere. The area of interest is the lighter circle in the centre.

Once we have the grid populated with this new implicit field, we can then find a specific contour to "polygonize". In this instance, we are interested in the 0 crossing, but we cannot directly evaluate at this iso-level. The reason is two-fold: first the contour line may very well be a single point, in which case no method will detect it. Alternatively, it could be a curve, in which case, again no method will detect it as there will be no change in sign from $+$ to $-$ which is what all polygonization schemes ultimately rely on. Instead, we evaluate at an iso-level that is very close to, but is not 0. This increases the area of interest and allows us the opportunity to detect it. Once the region has been polygonized, there are two cases that need to be recognized and handled separately:

1. The area that was detected corresponds to a single point. The vertices of the generated contour can then be collapsed into a single vertex, possibly by using

the centre of gravity. Once the new vertex is computed, it can be projected onto the adjacent cross-section. The surface vertex can then be computed by linear interpolation and the triangle fan can therefore be constructed.

2. The area that was detected corresponds to a curve. The vertices of the generated contour would then somehow be collapsed so what is left is the projection of the curve. Each vertex can then be projected onto the adjacent cross-section and the final surface vertices computed as before.

This same approach could be used, in theory, to solve the branching problem in the case where one or more contours converge in a single contour. Using the shadow field would provide a way of approximating where the point of inflexion could be located, and then a "bridge" would need to be constructed so that the contours could be connected. An illustration of this is shown in Figure 4.7. This also has the benefit of helping to solve the problem of how to connect the excess vertices that were explained in the previous section.

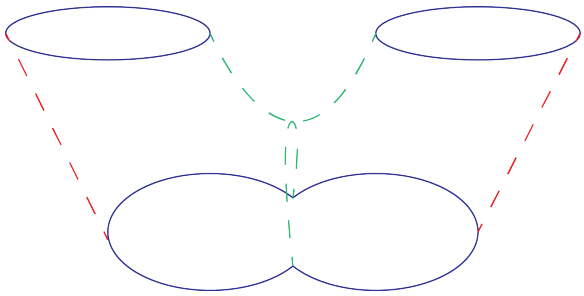


Figure 4.7: A sketch of how to build the connecting bridge marked in green.

The downside of using the shadow field are the following:

1. It provides no way of distinguishing between an area of contact that consists of a single point or a curve.

2. It provides no information as to whether contours that branch need to be connected, only that there may be an inflexion point.

3. It gives false-positives that are neither inflexion points nor areas of contact.

The last point yields another interesting avenue of research: if we were to connect the areas that we know are not points of inflexion or areas of interest, what appears to be formed is the medial axis of the surface. While more research needs to be done in this area, it is possible that using the shadow field can, in fact, yield the medial axis itself. If this is the case, it would give the following two benefits:

1. It would provide a faster way of computing the medial axis for implicit surfaces as in [Bloomenthal and Lim, 1999].

2. By being able to construct the medial axis, the branching problem would be solved, as the axis not only provides a path that could very well avoid branching, but also give information of which contours need to be connected and where the inflexion point could sit. This, coupled with the ideas provided earlier, would solve the problem in its entirety.

4.3.2 Solving Caps and Branching: Edge Spinning

An alternative to using the shadow field is to employ a technique similar to [Akkouche and Galin, 2001; Cermak and Skala, 2004]. Since we are able to continue sampling the field outside of the planar cross-sections, we could apply the concept of edge spinning to create a patch that closes the mesh. This would have the advantage of being able to construct the caps as well as the bridges required to close the branches. There are some problems that need to be resolved in order to employ this approach:

1. The super-voxels need to be extended to cover the area between cross-sections. This would allow us to continue optimizing the number of field evaluations as well as the number of fields that need to be evaluated. This is fairly straight-

forward, as the super-voxels remain roughly the same across slices provided the underlying surface does not change too quickly or the slices are too far apart.

2. Edge spinning was originally implemented to operate on triangles. While it is true that quads are ultimately represented as triangles for rendering, we would still need to adapt the algorithm to grow quads instead of triangles. The matching of the edges would then need to take this into account as well.

   In addition to this, there will be cases when cracks cannot be filled with quads. This will happen as a consequence of what was discussed at the end of section 4.2. As a result, these final cracks would have to be filled with a combination of triangles or n-gons.

4.3.3 Results and Comparison

Before we present the results comparing *MR* and *MC* there are some clarifications that need to be made. The first is that due to the nature of the cap and branching problems, the final mesh that *MR* produces will be incomplete in the following ways:

1. Slices with caps will be closed off at the first slice that has at least one contour in it. The approach used to close the mesh on this contour is not general, as it requires prior knowledge of the exact structure of the model, the slices, and the distribution of vertices.

2. Slices with branching will be ignored.

The second is that for all models chosen, care was taken in choosing the axis through which the surface is sliced to eliminate branching. The third is that, again due to the nature of the cap and branching problems, the sample of models is somewhat limited. That said, the results are promising and they showcase the speed of *MR*.

The tests were conducted on the same architecture used for the comparison between *MC* and Bsoid.

The test was performed on four models: a sphere, two blended spheres, a torus, a chain of 20 blended spheres, and a chain of 200 blended spheres. The grid resolution was set at 64^3 in the case of *MC*, and 64^2 in the case of *MR* with a super-voxel grid of 32^2 and 8 slices along the chosen axis. The reason for the lower number of slices is due to the subdivision of the contours. Since the contours are better fitted, we can use a lower number of slices and get a better quality mesh than *MC*. The results are presented in Table 4.1 and Table 4.2. The performance gains presented here were computed with the following formula:

$$G = \frac{T_{MC}}{T_{MR}}$$

Where G is the performance gain, T_{MC} is the execution time of *MC* and T_{MR} is the execution time of *MR*, both in seconds.

Model	***MC***	***MR***	**G**
Sphere	0.01591	0.00787	2.021
Blended spheres	0.01651	0.00876	1.88
Torus	0.01685	0.01237	1.36
Chain (20)	0.02865	0.01095	2.61
Chain (200)	0.13421	0.01880	7.13

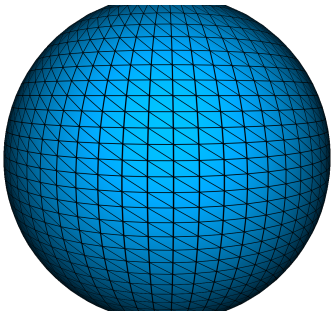
Table 4.1: Performance comparison between *MR* and *MC*. Times in seconds.

Model	***MC***	***MR***
Sphere	1992	1922
Blended spheres	2771	2522
Torus	4687	5888
Chain	5387	3602

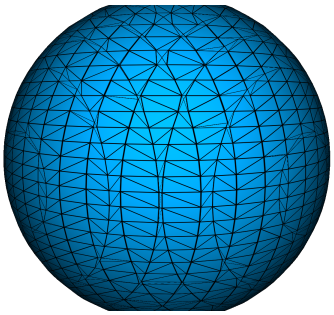
Table 4.2: Comparison of vertices produces by *MR* and *MC*.

As we can see, *MR* is on average 2 times faster than *MC*. This proves that, while

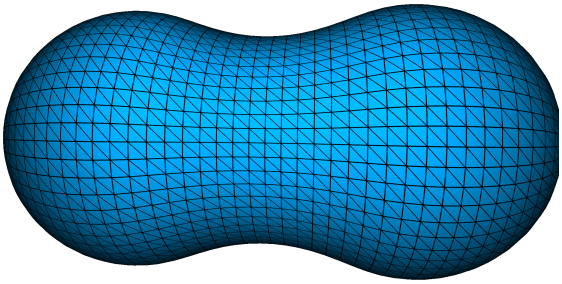
suffering from limitations, *MR* is a viable way of polygonizing implicit surfaces while providing equal or faster performance than *MC* and producing meshes of significantly higher quality. In Figure 4.8 we provide a side-by-side comparison of the meshes produced by *MC* and *MR*.



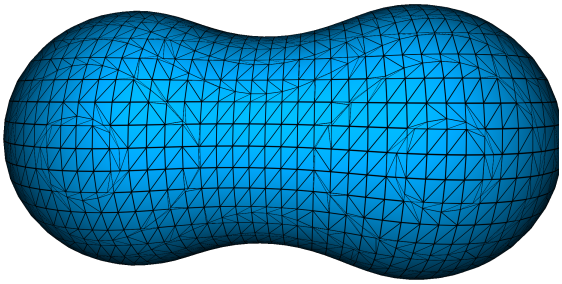
(a) Sphere polygonized by *MR*



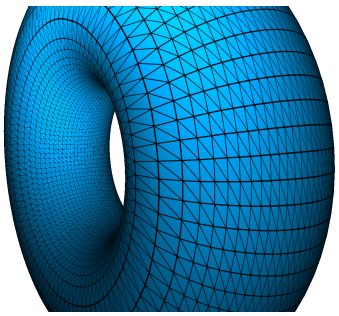
(b) Sphere polygonized by *MC*



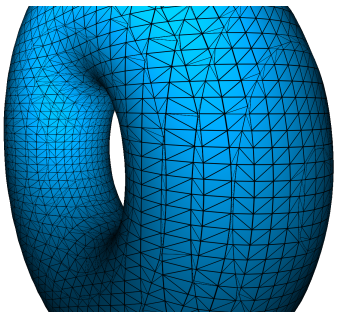
(c) Two blended spheres polygonized by *MR*



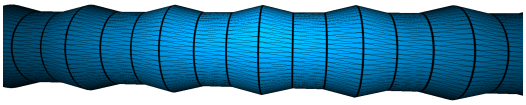
(d) Two blended spheres polygonized by *MC*



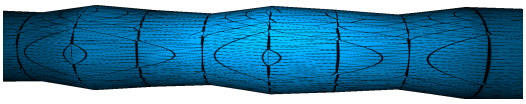
(e) Torus polygonized by *MR*



(f) Torus polygonized by *MC*



(g) Chain of 20 spheres polygonized by *MR*



(h) Chain of 20 spheres polygonized by *MC*

Figure 4.8: Comparison of the meshes produced by *MR* and *MC*.

Chapter 5

Conclusions

As stated in section 1.3, we want to create a polygonizer that has the following properties:

1. Be at least as fast, if not faster than, parallel Marching Cubes.
2. Produces a better quality mesh than Marching Cubes.
3. Produces a mesh that accurately represents the topology of the underlying surface.

Based on the evaluation of related work performed in chapter 2, we chose to compare two of the three fundamental algorithms: *MC* and Bsoid. The goal was to find which one of the two had the best advantages that we could exploit and modify to produce a new algorithm for polygonization. In chapter 3 we compared and contrasted *MC* to Bsoid. We conclude that while it is true that *MC* is easier to parallelize than Bsoid due to the independent nature of all its operations, it is significantly less efficient than Bsoid on all the performed tests. This is due to two main reason. The super-voxels reduce the number of field evaluations by at least two orders of magnitude while the use of hash tables to avoid re-computation of voxels and vertices reduces memory usage by at least one order of magnitude. Finally, execution

time is on average less than half that of MC.

With these results in mind, we select Bsoid as a base to create our new algorithm. Additionally we also introduced the concept of Field evaluations per vertex (FPV) as another measure for comparing Bsoid and MC. While neither algorithm was close to the ideal value of 1, Bsoid was at least two orders of magnitude closer than MC.

Using the idea of planar cross-sections to slice the surface we wish to polygonize, we made a new algorithm called Marching Rings (MR). As discussed in chapter 4, this algorithm has several advantages. First, the use of planar cross-sections allows us to bypass many of the ambiguities inherent to MC by reducing the problem from 3D to 2D. By employing the hash tables and super-voxels from Bsoid we are able to increase performance by reducing the number of field evaluations required to compute the initial contours. Next, the subdivision of the generated contours allows us to obtain a higher resolution than the grid size employed to construct them initially. Finally, by guaranteeing that all contours across all cross-sections contain the same number of vertices, we are able to generate a regular quad-mesh.

In section 4.2 we discussed the two main limitations of MR. These are:

1. The cap problem: this is produced when either the first point of contact along the chosen axis lies between two cross-sections or the region of contact is too small to be captured by the current grid resolution.

2. The branching problem: this occurs when there are different numbers of contours between two adjacent cross-sections.

We also showed that as currently formulated, the branching problem is underconstrained and as such has infinitely many solutions. In section 4.3 we discussed two potential solutions for both the cap and branching problems. The first involves the use of the shadow field, which is obtained through the projection of the gradient

onto the current plane. This has the side-effect of also showing a region that represents the medial axis, thereby being a new potential algorithm for finding the medial axis for implicit surfaces. The second potential solution is to employ edge spinning based on [Cermak and Skala, 2004]. While it does thave the advantage of generating regular quads that would stitch the pieces of the mesh together, care needs to be taken to make sure there are no cracks.

In spite of these two limitations, *MR* outperforms *MC* on genus 0 surfaces by a factor of 2 while producing a better quality mesh and a more accurate representation of the underlying topology of the surface.

Bibliography

[Akkouche and Galin, 2001] Akkouche, S. and Galin, E. (2001). Adaptive implicit surface polygonization using marching triangles. *Computer Graphics Forum*, 20(2):67–80.

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

[Bloomenthal and Ferguson, 1995] Bloomenthal, J. and Ferguson, K. (1995). Polygonization of non-manifold implicit surfaces. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 309–316, New York, NY, USA. ACM.

[Bloomenthal and Lim, 1999] Bloomenthal, J. and Lim, C. (1999). Skeletal methods of shape manipulation. In *Shape Modeling and Applications, 1999. Proceedings. Shape Modeling International'99. International Conference on*, pages 44–47. IEEE.

[Bloomenthal and Wyvill, 1997] Bloomenthal, J. and Wyvill, B., editors (1997). *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Bommes et al., 2012] Bommes, D., Lévy, B., Pietroni, N., Silva, C., Tarini, M., and Zorin, D. (2012). State of the art in quad meshing.

[Bottino et al., 1996] Bottino, A., Nuij, W., and Overveld, K. V. (1996). How to shrinkwrap through a critical point: an algorithm for the adaptive triangulation of iso-surfaces with arbitrary topology. In *In Proc. Implicit Surfaces '96*, pages 53–72.

[Bouthors and Nesme, 2007] Bouthors, A. and Nesme, M. (2007). Twinned meshes for dynamic triangulation of implicit surfaces. In *Proceedings of Graphics Interface 2007*, GI '07, pages 3–9, New York, NY, USA. ACM.

[Cermak and Skala, 2004] Cermak, M. and Skala, V. (2004). Adaptive edge spinning algorithm for polygonization of implicit surfaces. In *Proceedings Computer Graphics International, 2004.*, pages 36–43.

[Cermak and Skala, 2007] Cermak, M. and Skala, V. (2007). Polygonisation of disjoint implicit surfaces by the adaptive edge spinning algorithm of implicit objects. *International Journal of Computational Science and Engineering*, 3(1):45–52.

[Chan and Purisima, 1998] Chan, S. and Purisima, E. (1998). A new tetrahedral tesselation scheme for isosurface generation. *Computers & Graphics*, 22(1):83 – 90.

[Chernyaev, 1995] Chernyaev, E. V. (1995). Marching cubes 33: Construction of topologically correct isosurfaces.

[Crespin, 2002] Crespin, B. (2002). Dynamic triangulation of variational implicit surfaces using incremental delaunay tetrahedralization. In *Symposium on Volume Visualization and Graphics, 2002. Proceedings. IEEE / ACM SIGGRAPH*, pages 73–80.

[de Araújo et al., 2015] de Araújo, B. R., Lopes, D. S., Jepp, P., Jorge, J. A., and Wyvill, B. (2015). A survey on implicit surface polygonization. *ACM Comput. Surv.*, 47(4):60:1–60:39.

[Dietrich et al., 2009] Dietrich, C. A., Scheidegger, C. E., Schreiner, J., Comba, J. L. D., Nedel, L. P., and Silva, C. T. (2009). Edge transformations for improving mesh quality of marching cubes. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):150–159.

[Dyken et al., 2008] Dyken, C., Ziegler, G., Theobalt, C., and Seidel, H.-P. (2008). High-speed marching cubes using histopyramids. In *Computer Graphics Forum*, volume 27, pages 2028–2039. Wiley Online Library.

[Galin and Akkouche, 2000] Galin, E. and Akkouche, S. (2000). Incremental polygonization of implicit surfaces. *Graphical Models*, 62(1):19 – 39.

[Goetz et al., 2005] Goetz, F., Junklewitz, T., and Domik, G. (2005). Real-time marching cubes on the vertex shader. In *Eurographics (Short Presentations)*, pages 5–8.

[Gourmel et al., 2013] Gourmel, O., Barthe, L., Cani, M.-P., Wyvill, B., Bernhardt, A., Paulin, M., and Grasberger, H. (2013). A gradient-based implicit blend. *ACM Trans. Graph.*, 32(2):12:1–12:12.

[Hall and Warren, 1990] Hall, M. and Warren, J. (1990). Adaptive polygonalization of implicitly defined surfaces. *IEEE Computer Graphics and Applications*, 10(6):33–42.

[Hansen and Hinker, 1992] Hansen, C. D. and Hinker, P. (1992). Massively parallel isosurface extraction. In *Proceedings of the 3rd conference on Visualization'92*, pages 77–83. IEEE Computer Society Press.

[Hart, 1993] Hart, J. C. (1993). Ray tracing implicit surfaces. *Siggraph 93 Course Notes: Design, Visualization and Animation of Implicit Surfaces*, pages 1–16.

[Hart, 1996] Hart, J. C. (1996). Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545.

[Hilton et al., 1996] Hilton, A., Stoddart, A. J., Illingworth, J., and Windeatt, T. (1996). Marching triangles: range image fusion for complex object modelling. In *Proceedings of 3rd IEEE International Conference on Image Processing*, volume 1, pages 381–384 vol.2.

[Hui and Jiang, 1999] Hui, K. C. and Jiang, Z. H. (1999). Tetrahedra based adaptive polygonization of implicit surface patches. *Computer Graphics Forum*, 18(1):57–68.

[Johansson and Carr, 2006] Johansson, G. and Carr, H. (2006). Accelerating marching cubes with graphics hardware. In *CASCON*, volume 6, page 39. Citeseer.

[Ju et al., 2002] Ju, T., Losasso, F., Schaefer, S., and Warren, J. (2002). Dual contouring of hermite data. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 339–346, New York, NY, USA. ACM.

[Kalra and Barr, 1989] Kalra, D. and Barr, A. H. (1989). Guaranteed ray intersections with implicit surfaces. In *ACM SIGGRAPH Computer Graphics*, volume 23, pages 297–306. ACM.

[Karkanis and Stewart, 2001] Karkanis, T. and Stewart, A. J. (2001). Curvature-dependent triangulation of implicit surfaces. *IEEE Computer Graphics and Applications*, 21(2):60–69.

[Kobbelt et al., 2001] Kobbelt, L. P., Botsch, M., Schwanecke, U., and Seidel, H.-P. (2001). Feature sensitive surface extraction from volume data. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 57–66, New York, NY, USA. ACM.

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

[Liu et al., 2005] Liu, S., Yin, X., Jin, X., and Feng, J. (2005). High quality triangulation of implicit surfaces. In *Ninth International Conference on Computer Aided Design and Computer Graphics (CAD-CG'05)*, pages 6 pp.–.

[Lopes and Brodlie, 2003] Lopes, A. and Brodlie, K. (2003). Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):16–29.

[Lorensen and Cline, 1987] Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA. ACM.

[Miguet and Nicod, 1995] Miguet, S. and Nicod, J.-M. (1995). A load-balanced parallel implementation of the marching-cubes algorithm. In *Proceedings of High performance computing Symp*, volume 95, pages 229–239.

[Ohtake and Belyaev, 2002] Ohtake, Y. and Belyaev, A. G. (2002). Dual/primal mesh optimization for polygonized implicit surfaces. In *Proceedings of the Seventh ACM Symposium on Solid Modeling and Applications*, SMA '02, pages 171–178, New York, NY, USA. ACM.

[Paiva et al., 2006] Paiva, A., Lopes, H., Lewiner, T., and Figueiredo, L. H. D. (2006). Robust adaptive meshes for implicit surfaces. In *2006 19th Brazilian Symposium on Computer Graphics and Image Processing*, pages 205–212.

[Peir et al., 2007] Peir, J., Formaggia, L., Gazzola, M., Radaelli, A., and Rigamonti, V. (2007). Shape reconstruction from medical images and quality mesh generation via implicit surfaces. *International Journal for Numerical Methods in Fluids*, 53(8):1339–1360.

[Raman and Wenger, 2008] Raman, S. and Wenger, R. (2008). Quality isosurface mesh generation using an extended marching cubes lookup table. *Computer Graphics Forum*, 27(3):791–798.

[Rovira Galvez, 2015] Rovira Galvez, M. A. (2015). Atlas framework. http://marovira.github.io/atlas/.

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

[Schaefer et al., 2007] Schaefer, S., Ju, T., and Warren, J. (2007). Manifold dual contouring. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):610–619.

[Schaefer and Warren, 2004] Schaefer, S. and Warren, J. (2004). Dual marching cubes: primal contouring of dual grids. In *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.*, pages 70–76.

[Shirazian et al., 2012] Shirazian, P., Wyvill, B., and Duprat, J.-L. (2012). Polygonization of implicit surfaces on multi-core architectures with simd instructions. In *EGPGV*, pages 89–98.

[Singh and Narayanan, 2010] Singh, J. M. and Narayanan, P. (2010). Real-time ray tracing of implicit surfaces on the gpu. *IEEE transactions on visualization and computer graphics*, 16(2):261–272.

[Stander and Hart, 1997] Stander, B. T. and Hart, J. C. (1997). Guaranteeing the topology of an implicit surface polygonization for interactive modeling. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 279–286, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

[Stander and Hart, 2005] Stander, B. T. and Hart, J. C. (2005). Guaranteeing the topology of an implicit surface polygonization for interactive modeling. In *ACM SIGGRAPH 2005 Courses*, page 44. ACM.

[Treece et al., 1999] Treece, G., Prager, R., and Gee, A. (1999). Regularised marching tetrahedra: improved iso-surface extraction. *Computers & Graphics*, 23(4):583 – 598.

[van Overveld and Wyvill, 2004] van Overveld, K. and Wyvill, B. (2004). Shrinkwrap: An efficient adaptive algorithm for triangulating an iso-surface. *The Visual Computer*, 20(6):362–379.

[Varadhan et al., 2003] Varadhan, G., Krishnan, S., Kim, Y. J., and Manocha, D. (2003). Feature-sensitive subdivision and isosurface reconstruction. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 14–, Washington, DC, USA. IEEE Computer Society.

[Varadhan et al., 2006] Varadhan, G., Krishnan, S., Zhang, L., and Manocha, D. (2006). Reliable implicit surface polygonization using visibility mapping. In *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, SGP '06, pages 211–221, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Velho et al., 1999] Velho, L., de Figueiredo, L. H., and Gomes, J. (1999). A unified approach for hierarchical adaptive tesselation of surfaces. *ACM Trans. Graph.*, 18(4):329–360.

[Witkin and Heckbert, 2005] Witkin, A. P. and Heckbert, P. S. (2005). Using particles to sample and control implicit surfaces. In *ACM SIGGRAPH 2005 Courses*, page 260. ACM.

[Wyvill, 2009] Wyvill, B. (2009). Implicit modelling. In *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., 3rd edition.

[Wyvill et al., 1998] Wyvill, B., Guy, A., and Galin, E. (1998). The blob tree. *Journal of Implicit Surfaces*, 3.

[Wyvill and Jevans, 1988] Wyvill, B. and Jevans, D. (1988). Ray tracing implicit surfaces.

[Wyvill and van Overveld, 1996] Wyvill, B. and van Overveld, K. (1996). Polygonization of implicit surfaces with constructive solid geometry. *International Journal of Shape Modeling*, 2(04):257–274.

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

[Yamazaki et al., 2002] Yamazaki, S., Kase, K., and Ikeuchi, K. (2002). Non-manifold implicit surfaces based on discontinuous implicitization and polygonization. In *Geometric Modeling and Processing. Theory and Applications. GMP 2002. Proceedings*, pages 138–146.