SIMD and GPU-Accelerated Rendering of Implicit Models

by

Pourya Shirazian
Iran University of Science and Technology, 2008
Azad University of Tehran, 2005

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

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

SIMD and GPU-Accelerated Rendering of Implicit Models

by

Pourya Shirazian
Iran University of Science and Technology, 2008
Azad University of Tehran, 2005

Supervisory Committee

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

Dr. Roy Eagleson, Outside Member
(Department of Electrical and Computer Engineering, Western University)

Dr. Paul Lalonde, Departmental Member
(Department of Computer Science, University of Victoria)

Supervisory Committee

ABSTRACT

Implicit models inherently support automatic blending and trivial collision detection which makes them an effective tool for designing complex organic shapes with many applications in various areas of research including surgical simulation systems. However, slow rendering speeds can adversely affect the performance of simulation and modelling systems. In addition, when the models are incorporated in a surgical simulation system, interactive and smooth cutting becomes a required feature for many procedures.

In this research, we propose a comprehensive framework for high-performance rendering and physically-based animation of tissues modelled using implicit surfaces. Our goal is to address performance and scalability issues that arise in rendering complex implicit models as well as in dynamic interactions between surgical tool and models.

Complex models can be created with implicit primitives, blending operators, affine transformations, deformations and constructive solid geometry in a design environment that organizes all these in a scene graph data structure called the BlobTree. We show that the BlobTree modelling approach provides a very compact data structure which supports the requirements above, as well as incremental changes and trivial collision detection. A GPU-assisted surface extraction algorithm is proposed to support interactive modelling of complex BlobTree models.

Using a finite element approach we discretize those models for accurate physically-based animation. Our system provides an interactive cutting ability using smooth intersection surfaces. We show an application of our system in a human skull craniotomy simulation.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

My parents and my sister, For supporting me at all stages of my education and their unconditional love.

Brian Wyvill, For his mentoring and support, encouragement and patience.

Jean-Luc Duprat, For reviewing my code and excellent hardware architecture guidance in my project.

Zahed, Hamed, Nina, Kazem, Herbert, For their friendship and exciting conversations, proof-reading the papers and technical help.

Roy and Sandrine, For seeing me in Los Angeles, the awesome dinner at Hard Rock Cafe and guiding my research on the way.

GRAND, Mitacs, For funding and supporting my research. Reviews for posters and positive feedbacks.

Intel Corp. For providing the latest hardware and servers for my research.

Arash Mohammadi, For his great friendship and tennis practices in Victoria, all the good memories.

The cause of everything that happens to you is in you; you should therefore look within yourself to find the cause.
Ostad Elahi

To *Dr. Bahram Elahi* for his valuable lessons and scientific approach to life.

Chapter 1

Introduction

1.1 General aim of this Research

Rendering complex, dynamic implicit models in real-time and with some level of user interaction is a challenging problem with many applications in different areas of research including virtual surgery [25]. A complete model should be quite realistic, interactive and should enable the user to modify the topology of the objects. Rendering such models on commodity hardware and supporting high fidelity which in general implies high accuracy and interactive frame rates are two contrasting constrains. This is due to the lack of algorithms and techniques that can efficiently map the modelling problems in this domain to the parallel architecture of the graphics processor to support high quality generation and rendering of those objects.

A number of proposals have been recently presented to fulfill these two objectives. But even if the efficiency of the simulation models have been largely improved in the last few years, object modelling and deformation remains a rather complex task and can be solved in interactive time only on models composed by a few hundreds of cells. In addition, integrating effects such as cut and lacerations, makes the simulation model more complex. In modern interactive simulation environments the ability to cut 3-dimensional geometry in real-time is of fundamental importance. This creates the need for efficient cutting algorithms that process the underlying representation. In surgery simulation, interactive cutting algorithms enable the dynamic simulation of scalpel intersections that open immediately behind the scalpel [54]. Cutting a volumetric mesh under deformation is a non-trivial problem, due to several conflicting requirements. First, the cutting process should not create badly shaped elements, which could cause numerical instabilities during deformation calculation. Second, cut trajectory should be closely approximated for realistic appearance and third, performing volume and surface mesh connectivity changes in real-time requires performance tuned data-structures and algorithms which are not trivial for implementation. So far, most methods have concentrated only on one of these issues.

Therefore the following major problems are identified in the surgical simulation domain:

- Modelling complex tissues that are readily available for simulation [52, 50, 29].
- Real-time visualization of those tissues [13, 11].
- Performing interactive mesh connectivity modifications on complex models while under deformation [35, 81, 19, 34].

We present a comprehensive solution to these problems as following. First, our proposed modelling solution captures the key advantages found in volumetric modelling approaches using implicit surfaces [11, 87, 84, 85, 86, 63, 6]. Automatic blending and compact representation are the major benefits of using implicit surfaces for modelling. In addition, the ability to perform inside-outside tests easily is an inherent advantage in implicit models when implementing physically based simulations requiring collision tests. The *BlobTree* [84] combined blending, affine transformations and constructive solid geometry (CSG) operators in a comprehensive and compact scene graph data-structure. *BlobTree* provides the ability to create complex models incrementally [63]. Our contribution is the novel technique that bridges the gap between modelling and physically-based simulation of implicit objects, i.e. the created models are immediately available for interaction with surgical tools and with other tissues in the environment.

Secondly we propose a solution for high performance and scalable visualization of complex models created by *BlobTree* method. Volumetric models in general are often several orders of magnitude slower during visualization [10, 11]. We propose a data-driven algorithm for rendering complex implicit models in real-time on multi-core processors [66], later, we fine tune that algorithm for running on many core architectures such as the ones in high-end graphical processing units (GPUs).

Third, we take a novel approach in development of a stable and realistic cutting system. Our GPU-assisted interactive cutting algorithm allows arbitrary cuts in the model and can enable many scenarios for tissue manipulation while under deformations.

In what follows, the implicit modelling approach to model design will be studied. To achieve the initial goal of this research, a computational framework for designing, rendering and animating tissues has been developed and the details of the process is documented in the following chapters.

1.2 Motivation

Laparoscopic surgery brought new technologies into the operating room and created a distance between the surgeon and the patient. More recently, other minimally invasive techniques have been proposed, such as natural orifice transluminal endoscopic surgery, which can be considered as an evolution of laparoscopic surgery. The new surgical techniques developed in Laparoscopy required the surgoens in the field to go under training to adapt themselves to the new environment. With the loss in depth perception and decreased levels of tactile sensation, the new technique was certainly different from conventional open surgery.

Without open organ surgery, modern surgeons do not get training in the motor skills of the previous generation. Thus there is a motivation to develop realistic surgical simulations using real-time deformable models, and haptic rendering for further realism [45]. The following benefits have been reported from using surgical simulation systems [46, 4]:

- Systematic training and objective assessment of technical competence;
- Skills learned thanks to the simulator are transferable to the operating room;
- In case of rare pathological cases or when the best surgical strategy could not be found, The simulation can aid in developing patient-specific surgical techniques;
- The ability to use augmented reality for image-guided surgery (i.e. to improve the accuracy and limit the adverse effects of surgery).

In order to achieve these benefits, accurate, real-time bio-mechanical models are needed together with interactions with medical devices. Such interactions involve tissue manipulation and tissue dissection.

In this context, modelling and high-performance rendering are the core requirements for a simulation scenario. The development of fast algorithms for rendering, contact response, cutting and haptic feedback of soft tissues can enable a number of the aforementioned applications.

1.3 Limitation of Current Models

The requirements listed below are considered in our design of a robust modelling system for simulation:

- Complex models should be created incrementally from simpler, primary components that promote reusability.
- Visualization of complex models should be interactive and the result of modifications to the model should be seen in real-time.
- Interactions with the surrounding anatomy and with medical devices need to involve advanced contact models that can be computed in real-time.
- Guided cutting with scalpel and other variants of dissection operations e.g. drilling, should be supported.
- Modelling and simulation should be tightly coupled to enable the design of realistic environments for surgical training.

Popular existing techniques are based on mass-spring networks, methods based on linear elasticity, and explicit finite element models for non-linear materials [29, 50].

Mass-spring networks are quite simple to implement and very fast to compute, but they fail to properly characterize tissue deformation as they introduce artificial anisotropy through the choice of the mesh, and make it difficult to relate spring stiffness to material properties such as Young's modulus [19].

Most methods assume Linear elasticity based on small displacements and pre-computed response values to accelerate the computations. The small strain assumption is very restrictive. In addition, during any topological modifications e.g. in cutting, the pre-computed values have to be recalculated which masks their effectiveness in the overall performance of the system.

Cutting deformable tissues is one of the most sought after features in a surgical simulator. In an interactive system with high expectations of realism and performance, the implementation of topological modifications can become very complex. The proposed solutions suffer from smoothness of the cutting plane or slower solve time due to lots of extra nodes added to the system. In chapter 6 we review all the related work in this topic and present our high performance cutting algorithm which is built into our physically-based simulation system.

1.4 Contributions

In this research we take into account the requirements of modern surgical procedures and introduce a new modelling framework with the ability to perform real-time cutting in a complex model (i.e. complex in terms of the number of implicit primitives and operators used to construct the model and also the number of finite element cells involved in the physical simulation), providing a high performance system for modelling and visualization with applications in surgical simulation.

Contributions described in this thesis fall into four broad categories: a modelling system to create complex tissues under the heading of the *BlobTree* ; a high-performance subsystem for rendering; a high-performance mesh connectivity modification algorithm to support cutting and a real-time Craniotomy simulation for neurosurgery simulation. The full contributions list is as following:

- A comprehensive modelling framework supporting a broad set of skeletal implicit primitives, sketched primitive objects, warping, blending, affine transformations and constructive solid geometry operators in the compact *BlobTree* structure. Our framework also provides a software architecture for physically-based animation of rigid and deformable models.

- An algorithm for interactive polygonization of implicit surfaces on multi-core architectures with SIMD instructions (Peer reviewed contribution [66]).

- An optimized GPU-assisted algorithm for high-performance polygonization of implicit surfaces on many-core architectures. As opposed to related work in this domain which is based on static implicit functions or constant range data, our rendering method applies to dynamic, data-driven *BlobTree* scene graph for complex implicit models.

- A high-performance algorithm for cutting rigid and deformable tissues interactively.
- Smooth cutting of complex volumetric meshes to avoid producing the jagged lines without the need for a post-processing step.
- A novel mesh data-structure suitable for storing dynamic meshes on the GPU to support real-time modifications during cutting
- Real-time Craniotomy simulation for neurosurgery and biopsy simulations.

1.5 Overview

In the next chapter we start by providing background material on implicit modelling technique, the *BlobTree* scene-graph and the concept of sketch-based, incremental modelling. We continue by reviewing the related work in surface extraction algorithms from volumetric models and mesh connectivity modifications.

Chapter 3 presents our rendering framework to visualize complex *BlobTree* models using multi-core architectures. Building on the outcomes of chapter 3, the improved results are reviewed in chapter 4.

Chapter 5, presents our system to support deformations and elastic behavior of tissues. The overall software pipeline to support the entire simulation system is presented in this chapter as well.

In Chapter 6 we present one of the main contributions of this thesis which is the high performance tissue cutting. After a brief overview of the related work we present our novel technique in cutting complex soft tissues interactively. Chapter 7 showcases a skull craniotomy simulation scenario and provides comments on the operation itself and the achieved results.

In Chapter 8 we provide a summary of the results in the previous chapters and review the limitations of the current system and some discussions on the future work in this research topic.

Chapter 2

Background Material

This chapter provides a short summary of the material which is relevant to the subject of this research. We describe the concepts underlying implicit modelling, CPU and GPU architectural differences and scalability issues. The chapter concludes with a review of the related work in this domain.

2.1 Implicit Modelling

Parametric surfaces such as Bezier patches has a generative form that enables two dimensional iteration over the surface directly. However, in the case of implicit models, the surface is surrounded by the object's volume and an extraction process is required to access the *iso-surface*. The formal definition of the surface and volume in this case is as following:

$$S = \left\{ M = (x, y, z) \in \mathbb{R}^3 | F(x, y, x) = c \right\} \tag{2.1}$$

$$V = \left\{ M = (x, y, z) \in \mathbb{R}^3 | F(x, y, x) \geq c \right\} \tag{2.2}$$

Function F computes the field value at a certain position M. c is a constant and is called the *iso-value* which is set to 0.5 in our system. For each point in space if the field is greater than c the point is considered inside the model otherwise outside.

Most of the primitives used in the *BlobTree* are built from geometric skeletons, which are incorporated in other implicit modelling software packages such as BlobTree.net [20] or ShapeShop [63]. They are ideally suited to prototype shapes of arbitrary topology [31, 11].

The modeller has access to various skeletal primitives which are the basic building blocks of the system for constructing complex shapes. Each primitive is a distance field dS which encompasses a volume of scalar values. In its raw form dS is unbounded, by modifying dS with a field function g, the influence of the field is bounded to a finite range.

Usually the function maps the distances to the range $[0, 1]$, where the field has values of 1 at the skeletons and 0 after a certain distance to the skeleton (usually at distance 1). A discussion of field function is provided in [67]. Skeletal implicit primitives are combined using binary operators, which are applied pair-wise to field-values f, and represented by a node in the *BlobTree* , whose children are either primitives or operators themselves.

Field values are computed for the child-nodes and combined to yield a new value according to the operator type. This makes it possible to go beyond the classical Boolean operators, and define general blend operators that e.g. create smooth transitions between shapes. The most common operator that creates a smooth transition between several values is called the summation blend [11]:

$$F_A(x, y, z) = \sum_{i=1}^{i=N_A} F_i(x, y, z) \tag{2.3}$$

Where an implicit model A is generated by summing the influences of N_A skeletal elements: The field value due to an skeletal element at a point in 3D-space is computed as filtered distance to its skeleton where the filter function (i.e. falloff function) is defined as follows [84]:

$$g_{\text{wyvill}}(x) = \begin{cases} 1 & \text{if } x \leq 0 \\ (1 - x^2)^3 & \text{if } 0 < x < 1 \\ 0 & \text{if } x \geq 1 \end{cases} \tag{2.4}$$

In equation 2.4, x is clamped to the range $[0, 1]$. This polynomial smoothly decreases from 1 to 0 over the valid range, with zero tangents at each end. An important property of this skeletal primitive definition is that the scalar field is *bounded*, meaning that $f = 0$ outside some sphere with finite radius. Bounded fields guarantee local influence, preventing changes made to a small part of a complex model from affecting distant portions of the surface. Local influence preserves a “principle of least surprise”that is critical for interactive modelling.

Normals can be derived from gradients which are computed by evaluating 4 field values and performing a numerical approximation:

$$\nabla F(x,y,z) = \begin{cases} F(x+\delta,y,z) - f \\ F(x,y+\delta,z) - f \\ F(x,y,z+\delta) - f \end{cases} \tag{2.5}$$

Where $f = F(x,y,z)$ is the field at point (x,y,z).

Each skeletal primitive has a bounded region of influence in space. For each node in the tree an axis-aligned bounding box is computed which is used to trivially reject those field queries that are outside the box. The bounding box of the entire model is computed as the union of all primitive nodes bounding boxes.

For evaluating the field at a point P in a *BlobTree* model such as the one shown in figure (2.1), the tree structure should be traversed from root to leaves recursively. Each operator combines the values of its children according to its type. For example, for a simple blend the values are summed. A leaf node represents a primitive, and returns the value by applying equation 2.4 to the distance of P from the primitive.

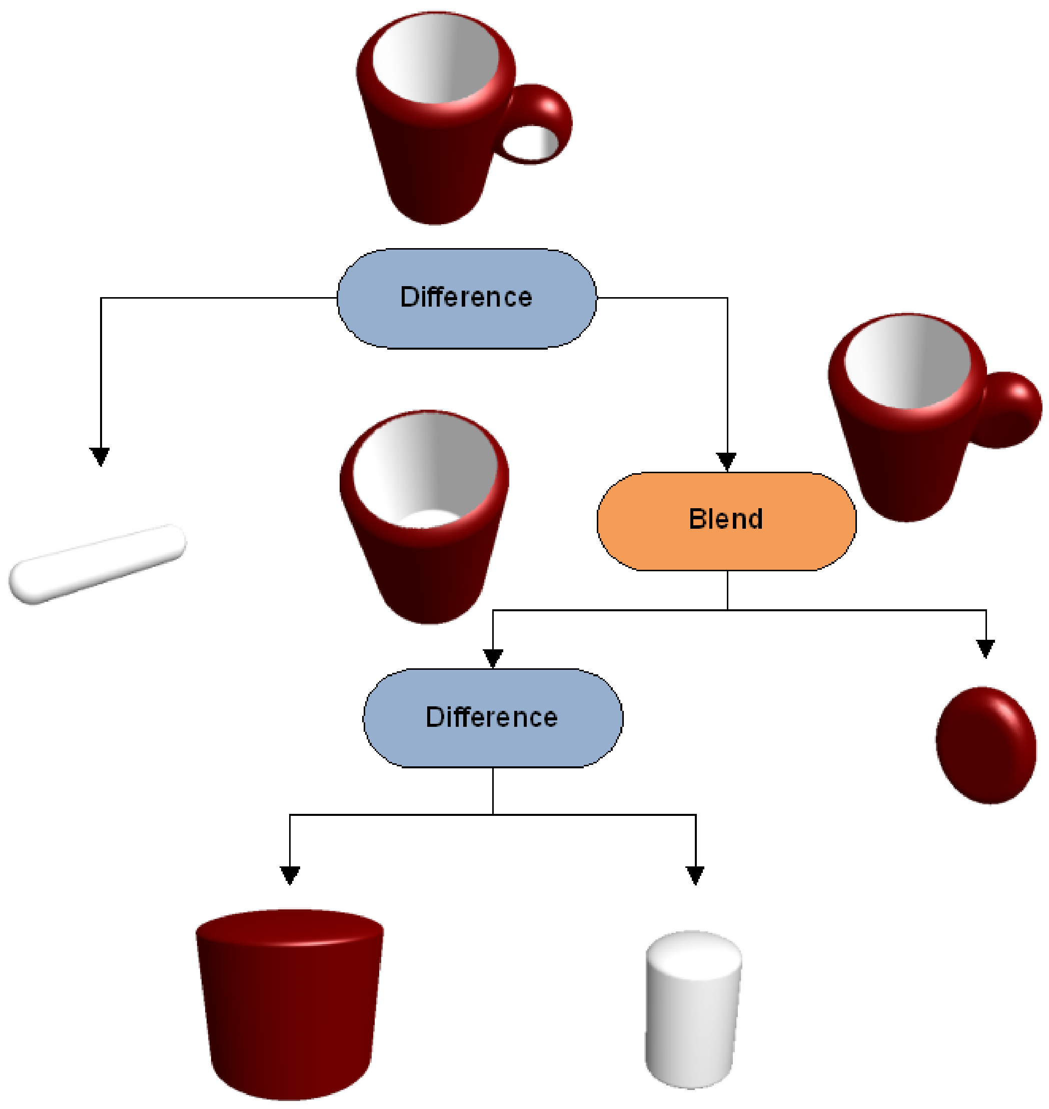


Figure 2.1: *BlobTree* structure of a coffee mug created with CSG and skeletal implicit primitives.

For visualization purposes the *BlobTree* is queried numerous times to evaluate the field. As suggested in [62] accelerating field computation will have a large impact on the overall surface extraction process.

2.2 Sweep Surfaces and Sketching

Implicit primitives in our system are created from skeletons which are simple geometrical shapes such as points, line segments or polygons from which volumetric distance fields are created. In order to support more complex geometries *Schmidt* et al. proposed the implicit sweep objects technique where the 2D shape sketched by the user is sampled and an implicit approximation is created from the sample points [61]. This is done by fitting a thin-plate spline as a base shape to the sampled points using variational interpolation [78]. One advantage of creating the base shape using variational interpolation is that the resulting implicit field is C^2 continuous, a property needed when the shape is involved in several blending operations [3].

A continuous 2D scalar field is created from several field value samples $(\mathbf{m}_i, v_i)$, where $\mathbf{m}_i$ describes the position of the sample and v_i is the desired field. The thin-plate spline used to create the variational implicit field $f_c(\mathbf{u})$ is defined in terms of these points weighted by corresponding coefficients w_i combined with a polynomial $P(u) = c_1 u_x + c_2 u_y + c_3$.

$$f_c(\mathbf{u}) = \sum_{i \in N} w_i(\|\mathbf{u} - \mathbf{m}_i\|)^2 ln(\|\mathbf{u} - \mathbf{m}_i\|) + P(\mathbf{u}) \tag{2.6}$$

The weights w_i and coefficients c_1, c_2, and c_3 are found by solving a linear system defined by evaluating equation 2.6 at each known solution $f_c(\mathbf{m}_i) = v_i$. The resulting thin plate spline can then be used as the basis of several different primitives:

- Inflated Objects
- Swept object along a trajectory
- Revolving object around axis

These sketched objects can then be used in the same way as the standard skeletal implicit primitives to create unique 3D shapes. Such unique shapes were not possible to create in previous collaborative environments, especially given the small memory footprint needed to transfer the information when using this technique.

2.3 Related Work

In the following we review the related work associated with the major contributions of our research. The surface extraction methods that attempted to enhance the performance of the sampling process on multi-core CPU processors are reviewed first. Some others attempted to exploit the processing power on the GPU and mapped the same problem to many-core processors. Later the previously proposed algorithms for cutting volumetric meshes and the application of this framework in a surgical simulation (Craniotomy) are studied.

2.3.1 Surface extraction methods

Several methods for polygonization of implicit surfaces have been proposed which can be classified based on speed, accuracy of the output mesh or quality. Comparing these methods in terms of performance reveals that space partitioning methods are the fastest and the most popular. The paper [87] was the first to introduce a method for finding iso-surfaces using uniform space subdivision into cubic cells. A seed cell on the surface was found by starting at a vertex close to each primitive and evaluating the field at cell vertices along each of the three axes to find a surface crossing. Vertices inside the volume were classified as 'hot' and 'cold' outside. A hash table was used to keep track of processed cells to avoid redundant field evaluations and to avoid storing any cells that did not contain part of the surface. Only adjacent cells that share an intersecting edge with their parent were processed, and a second cubic subdivision served to reduce the number of primitives considered in each field evaluation. Ambiguous cases were ameliorated by taking another sample from the center of the face. A similar method was later introduced as Marching Cubes in [47]. The main difference between the two algorithms was that Lorenson et al. applied their method to discrete volume data instead of sampling a continuous function and in Lorenson's method the space was completely partitioned into cubic voxels and all cubes were visited.

Bloomenthal showed that the ambiguous cases can be dealt with by subdividing cells into tetrahedra [9], and also that a six tetrahedron subdivision was superior to subdividing into five [32]. The fact that tetrahedral simplices have 4 vertices reduces the total number of configurations to 16 (or 3 by symmetry), however, the number of redundantly generated triangles as a result of this decomposition increases significantly. We will refer to marching cubes and tetrahedra, with MC and MT respectively throughout this chapter.

There have been many enhancements proposed for both MC and MT. Some gain advantage by classifying cubes according to different criteria and surface edge intersection calculation and number of field function evaluations. For example, Dietrich et al. [22], did a statistical analysis of cube configurations in MC that are responsible for most of the degenerate triangles in the output mesh. Their algorithm avoids those cube configurations by inserting an extra vertex into the cell when generating triangles as was done in [87] where an extra sample was taken. This reduces the statistical occurrence of the problem.

Triquet et al. [77] enhanced MT by applying time-stamps on calculated values and using hash tables for retrieving them. They also pre-computed surface vertices along crossing edges which are shared with adjacent voxels and referenced previously calculated values to avoid re-evaluating them. This latter enhancement was also done in Bloomenthal's polygonizer [9] and was a fairly common feature of implicit surface polygonizer's of the 1990s.

Beside enhancing serial algorithms some attempts were made to increase the performance of MC by dividing the workload between multiple CPUs or on a network grid of computers. Mackerras proposed an MIMD implementation of MC algorithm [48]. The bounding volume is divided into uniform blocks and each processor runs a serial implementation of MC on one or more blocks. They reported that because of efficient usage of cache their method showed a speed-up greater than the total number of physical processors involved. Hansen and Hinker presented a parallel implementation of MC [33]. They labeled each cube with a virtual processor identifier to avoid complexities in communicating between processors, then each cube is processed independently. They reported linear speed-up by increasing the number of physical processors. Their method spends constant time on each processor regardless of the number of polygons in a cubic cell.

The advent of shader programs and GPGPU computing interested some to port serially computationally intensive programs to the GPU. Space partitioning methods like MC and MT are good candidates for these devices since each cell (either tetrahedra or cube) can represent an independent volume to be processed on a separate SIMD core.

Kipfer and Westermann proposed a GPU-accelerated Marching Tetrahedra algorithm that stores the topology of the surface on the GPU [39]. They used a span-space interval tree to cull tetrahedral elements that don't intersect with the surface. Caching edge-surface intersections helped them to avoid redundant calculations. For computing edge-surface intersections they used linear interpolation for finding roots along each edge which is less accurate and degrades the quality of the output mesh.

Johansson et al. accelerated iso-surface extraction using graphics hardware [36]. They stored MC cases on the GPU and used a vertex program to compute surface intersection points. They used a span-space data-structure similar to [17] to enhance the cell classification phase in MC. Their method shows a speedup order of 13 over the naive algorithm.

Tatarchuk et al. presented an iso-surface extraction algorithm implemented using DirectX [76]. They used graphics hardware to visualize medical data. They maximized utilization of SIMD units on the GPU by separating their polygonization (which is a hybrid of marching cubes and marching tetrahedra) into two phases: Fast cube tetrahedralization and a marching tetrahedra pass. Each input voxel position is dynamically computed in the vertex shader, then they used the geometry shader and stream-out features of DirectX 10 to tessellate voxels into six tetrahedra spanning the voxel cube. However their method is limited to medical volume datasets.

The polygonization method proposed by Yang et al. [88], is the enhanced version of the Araujo's method [58]. The main idea is to subdivide the bounding box of the entire model into eight parts and then process them in parallel.

For each part, a seed point is found on the surface and is increasingly expanded to form a local mesh for that part by using the surface tracking approach. Using local curvature of the implicit surface, triangles of varying sizes are produced. However, their method is not scalable since it can not guarantee finding a seed point per each sub box in case the number of sub boxes increases. They reported very slow rendering times even for the simple models that they have tested their system with.

In a similar work Knoll et al. [41] proposed interactive raytracing of arbitrary implicits with SIMD interval arithmetic. They used SSE instructions for fast computation of interval arithmetic and ray traversal algorithm. However, their method is restricted to static implicit functions and algebraic surfaces.

None of the proposed methods above used a modelling framework to define their input data in a hierarchical structure similar to the *BlobTree*. The proposed methods are limited to constant volume data or static algebraic implicit functions to represent the underlying volume.

In a closely related work, Schmidt et al. [62] used a field caching mechanism inside the *BlobTree* to perform fast potential field reconstruction without traversing the entire tree. They used a trilinear reconstruction filter for field value and a triquaratic filter for gradient interpolation. They evaluated cache efficiency by polygonizing a *BlobTree* model once using cache nodes and the other time without. They reported up to 16 times speedup for polygonizing a model with different resolutions. However, their method is not scalable since the cache nodes cannot be updated from different processing threads without using locking mechanisms or a data race condition can occur.

2.3.2 GPU-Accelerated rendering of implicits

GPU accelerated rendering techniques has been the topic of interest for the graphics community in the past two decades. Several GPU-accelerated algorithms have been proposed for fast triangulation and rendering of iso-surfaces that are defined by volume data-sets, algebraic surfaces and radial-basis functions. In this section we will review the most related works.

Chochlík et al. proposed a GPU accelerated polygonization algorithm for dynamically changing implicit surfaces [16]. Their method is based on the marching tetrahedra (MT) algorithm. The model is partitioned into cubic cells first and then each cell is subdivided into 6 tetrahedra to be further processed using the GPU geometry shading stage. The vertices are marked inside if their associated field is above zero and outside otherwise. A configuration index is computed per each tetrahedra based on the inside/outside vertices. The triangle mesh is produced which is shaded using the fragment shader stage. No further analysis has been made on the performance of their algorithm and the input models are limited to time varying simple algebraic surfaces. The used a linear interpolation root finding method which produces low quality output.

Buatois et al. proposed a GPU accelerated iso-surface extraction method based on MT, similar to Chochlík et al. [14]. The texture memory to transfer the position and field values of the grid vertices. They presented an analysis of the performance of their algorithm using a fluid simulation volume data-set. They reported that excessive texture fetches can be a bottleneck in the performance of their method.

Tatarchuk et al. proposed a GPU iso-surface extraction algorithm which is a hybrid of marching cubes and marching tetrahedra [75]. They start by voxelizing an implicit grid into discrete cube cells and then convert that to a tetrahedral representation. They implemented an MT algorithm on geometry shader stage of DirectX10 API. For root finding method they fitted a parabola along the intersecting edges and evaluated a quadratic equation which produces a better approximation. They tested their system using the visible human volume data-set. Their method is limited to static volume datasets and is not usable in a data-driven setting where the topology of the underlying model changes.

With modern hardware and fast GPUs, ray tracing of implicit surfaces is the subject of much research. Knoll et al. [40] presented CPU and GPU algorithms that can achieve interactive visualization for common algebraic surfaces. The surfaces used in by Knoll et al. are not arbitrary implicit models but surfaces generated using traditional kernels or functions. Similar surfaces are presented by Singh and Narayanan for real-time ray tracing of implicit surfaces using the GPU [70].

Kipfer and Westermann [39] accelerated rendering of implicit surfaces by avoiding redundant computation of edge surface intersections. Our method also employs this feature to reduce the overhead. They also use features of the GPU to reformulate the iso-surface identification and reduces numerical computations and memory access operations. They used a span-space data-structure to avoid processing non surface intersecting elements.

Kanai et al. [38] proposed a rendering method for sparse low-degree implicit surfaces (SLIM) using ray casting approach. The ray and IS intersection test has been carried out on the fragment processing stage. They employed level of detail rendering and view frustum culling to speedup the rendering. The coefficients for the IS are passed in using textures. They reported high quality and interactive rates for several models. The large number of processed fragments is the bottleneck in this process and models with lower number of nodes could be rendered slower than more complex models that cover less fragments. Although Kanai et al. 's work is data-driven but the increasing cost of fragment processing is the main bottleneck in their system. Also since they are not producing any mesh the computations will be lost after rendering.

2.3.3 Volume mesh cutting

A number of approaches has been proposed by the computer graphics community to enable cutting of deformable and rigid models. Except for a few methods most of them use tetrahedral meshes for the volumetric mesh representation. Bielser et al. performed an adaptive refinement of the tetrahedral elements cut by a virtual scalpel [8]. In another work Bielser et al. presented a progressive approach to cutting [7], where the decomposition of a tetrahedron is changed depending on the movement of the cutting tool inside an element. However, the approach is highly non-trivial to implement and also poses some stability problems due to badly-shaped elements.

Mor et al. tried to reduce the number of sub-elements created while cutting tetrahedral meshes [51]. One of the major issues in cutting is the creation of ill-shaped elements i.e. skinny elements, which can adversely affect the performance and stability of the system solver. Some work attempted to avoid such elements via mesh alignment techniques [53, 74]. Other methods tried to solve the issue by removing them completely which resulted in volume loss and jagged lines along the cut surface.

Wu et al. [81] proposed an algorithm for 3D mesh cutting using a combination of the adaptive octree refinement with an iterative composite element hierarchy to enable simulating high-resolution cuts with a small number of degrees of freedom (DOFs). They used the dual contouring method [37] to keep the sharp creases along the cut. Due to the high computational cost and naive implementation their method is not scalable and has yet to become an interactive cutting approach.

In a closely related work Courtecuisse et al. presented a soft-tissue cutting system with haptic feedback [19]. Their cutting strategy follows Mor et al. [51] work and suffers from jagged lines along the cut surface as shown in their examples of a laparoscopic hepatecotomy. They also produce too many new nodes when subdividing cut elements.

Jerabkova et al. proposed a solution to the ill-shaped elements problem by using hexahedral elements instead of tetrahedra [34]. Their approach relies on fine-level voxels to model object surface and simulate cutting. The volume data requires more memory space than traditional, surface-based models. Cutting is performed by removing voxels. For sufficiently small voxels this typically remains unnoticeable but it may result in significant volume loss in case of a large number of cuts.

Jin et al. proposed a meshless total Lagrangian adaptive dynamic relaxation cutting algorithm to predict the steady-state responses of soft tissue [35]. A cloud of points is used for discretization and approximation of the deformation field within the continuum without generation of finite element meshes. They didn't report any performance measurements and the quality of the cuts could not be verified with the simple truth cube model they reported in their paper.

Sifakis et al. [69] proposed a geometric algorithm for placing cracks and incisions on tetrahedralized deformable objects. Their method is similar to the virtual node algorithm in that they avoid sliver elements and their associated stringent timestep restrictions. Producing ill-conditioned triangles on the material surface can have a negative effect on collision handling specially in case of a self collision. Also in their system a cut that partially intersects a tetrahedron without separating it into disconnected fragments will not allow the material to separate within that embedding tetrahedron.

Steinemann et al. [73] created a hysteroscopy simulator and minimized the number of added elements after a tetrahedral subdivision by cutting along existing edges and faces. The problem with their system is that the result of the cutting is produced only after it has been completed and this leads to a delay in the system. Unfortunately they didn't report any performance statistics of their algorithm.

In the following sections we provide an overview of the system and the data structures involved in the process and our cutting algorithm. The chapter is concluded by the analysis of the simulation results.

Chapter 3

High Performance Rendering on Multi-Core Architectures

One of the main challenges in animating deformable tissues is their rendering which requires to support high frame-rates [66]. In order to leverage the benefits offered by *BlobTree* modelling the rendering issue has to be tackled accordingly. In this chapter we present a parallel method for speeding up the generation of a polygon mesh from an implicit model [66]. Although the method is applicable to many types of implicit surfaces, we focus on surfaces generated from fields surrounding geometric primitives, known as skeletal implicit surfaces, [11] that are discussed in chapter 2. The model data structure is a tree whose leaf nodes are primitives, and internal nodes are operators; the *BlobTree*, [84]. Currently the *BlobTree* supports operations such as; arbitrary blends, boolean operations, warping at a local and global level including contact deformations. Geometric transformation matrices are also stored as nodes in the tree so the data structure is also a scene graph.

A *BlobTree* is typically visualized by polygonization to produce a triangle mesh to be rasterized by the graphics processor. Direct ray tracing [11] can also be used, to produce high quality images. Both methods require computation of the field value which can only be evaluated by traversing the *BlobTree* structure. The field due to each operator depends on its child nodes and the leaves are the primitives which can be any implicitly defined function; e.g. distance field due to geometric skeletal elements.

Implicit modelling using the *BlobTree* has several advantages over other modelling methods. Various different blends are simple to represent, as are free-form volume deformations and constructive solid geometry operations (CSG) [30]. Other operators such as detecting contact, and warping surfaces accordingly (see [15]), can easily be represented as nodes in the *BlobTree*.

An incremental, sketch based *BlobTree* system was built by Schmidt et al. [63], promoting flexibility and modular design for the creation of complex models, and most of the earlier problems with the methodology have been overcome [6]. Although direct manipulation is possible [63], very complex models can only be visualized interactively as coarse meshes. Hence the need for a faster polygonizer. The *BlobTree* facilitates incremental modelling, a strategy that promotes flexibility and modular design for creating complex models.

The main contribution presented here is a high performance polygonization algorithm that scales well with the number of physical cores and SIMD vector width available on modern processors.

As opposed to previous work that attempted to render implicit surfaces defined by static algebraic surfaces or volumetric scanned data, our method is data-driven where the definition of the surface can change over time. This feature is particularly useful in collision detection applications such as surgical simulations where the interaction of the surgical tools and deformable tissues should be visualized in real-time [43].

In addition we have improved the performance of the algorithm that finds the intersection of a cube edge and the surface, by making use of the SIMD architecture, to find the intersection in a single run of a field evaluation kernel.

The chapter is organized as follows; In section 3.3 our algorithm is explained along with the improvements made to the distance-field computation process. Our performance results and future work are presented in sections 3.5 and 3.6, respectively.

3.1 Architecture Constraints

In this section we define some processor architecture constraints, i.e. minimum requirements from the hardware side to implement our algorithm as efficiently as possible. The algorithm scales with the number of physical cores and the SIMD vector width available on the processor. See results section (3.5).

Our current implementation leverages both Intel SSE with 4 float wide and Intel AVX with 8 float wide SIMD instruction sets. Using a cache-aware technique our algorithm is designed to minimize the movement of cache lines in and out of the processor's on-chip memory. To this end the technique requires at least 256 kilobytes of last level cache memory per each processor core. The input data structures take about 192 kilobytes of memory in our implementation.

Although our test environment was Intel based, our algorithm should be implementable on any multicore machine with SIMD instructions and sufficient cache.

3.2 Naming Conventions

The polygonization method used, is a space partitioning algorithm based on [87], which uses a uniform grid of a user defined cell size (*cellsize*). In order to leverage the SIMD parallel computation capabilities of the processor, the bounding box of the model is divided into axis-aligned grids of 8x8x8 vertices where each grid is called model partitioning unit (MPU).

An MPU is $7 * cellsize$ as shown in figure 3.1. Each MPU contains 7*7*7 or 343 cubic cells. In section 3.4, a detailed study is made on various dimension sizes and the impact of this parameter on the overall performance. An MPU is called *empty* if it does not intersect with the iso-surface of the model. The list of all MPUs is called the $MPUSET$ and a half open interval $[a, b)$ over $MPUSET$ is called an $MPURANGE$ which contains consecutive MPUs from a to $b - 1$.

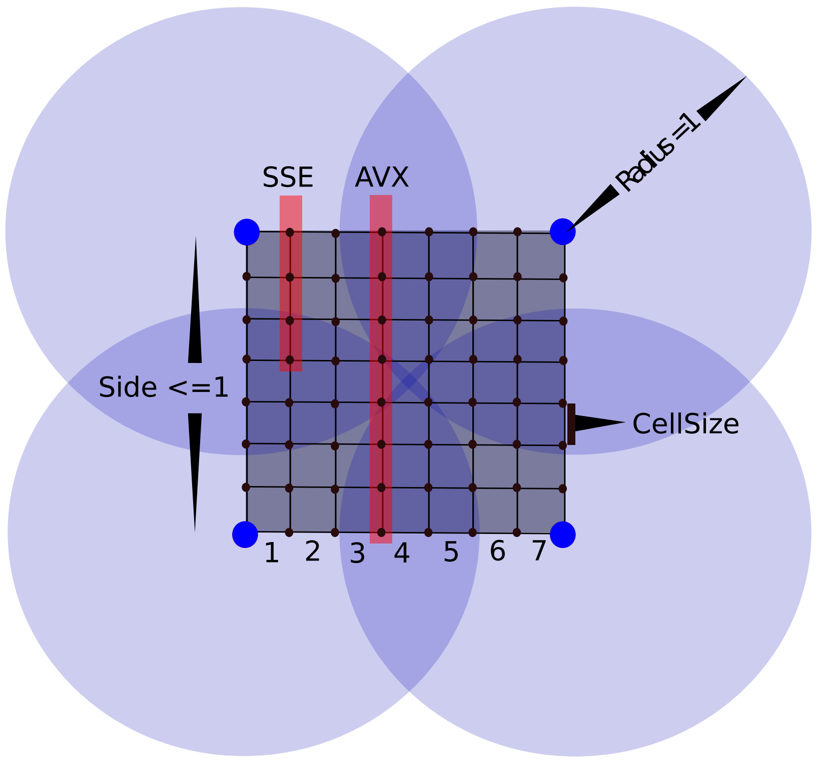


Figure 3.1: The *MPU* is our unit of computation per each core illustrated as a 2D cross section here. Field-values due to every 4 or 8 points are computed in parallel with SSE or AVX instructions, respectively. When the field at a vertex is zero no iso-surface will pass in the neighborhood of a unit circle (sphere in 3D) centered at that vertex.

3.3 Algorithm

The input to our algorithm is a *BlobTree* data structure, representing an implicit model whose iso-surface we wish to find. Output is a triangle mesh. The model bounding box and the cellsize parameter are supplied by the user to control the resolution of the final mesh.

The *BlobTree* structure is first converted into a compact, linear structure required for SIMD optimization techniques, then the model bounding box is divided into the *MPUSET* with respect to the *cellsize* parameter. The *MPUSET* is processed in parallel using multiple cores; with a fast empty *MPU* rejection method and SIMD surface extraction algorithm the mesh contained within intersecting *MPU*s is extracted. The algorithm has no synchronization points except after all *MPU*s are processed and the triangle mesh is sent to the GPU for rasterization. The left column in figure (3.2) displays these preparation steps in order. The following sections describe this whole process in detail.

We start by describing the initialization phase and continue with the surface extraction details in the next section. The algorithm starts by computing the size of an *MPU* side (7 cells) and dividing the bounding box of the model into a 3D grid of *MPU*s, where each *MPU* is assigned a unique global identifier. The main idea of our algorithm is parallel processing of the set of all *MPU*s (*MPUSET*) using multicore and SIMD processing techniques.

Our algorithm recursively splits *MPUSET* into disjoint *MPURANGE*s where each *MPURANGE* is assigned to an idle core on the processor. The granularity of the divisions can be determined by the average amount of machine cycles spent to process an *MPU*, however, in our implementation we resort to the solution provided by Intel Threading Building Blocks (TBB) [57], which provides a non-preemptive task scheduling system to take care of the differences in task loads by monitoring processors and starting new tasks on idle cores automatically (work-stealing) [57].

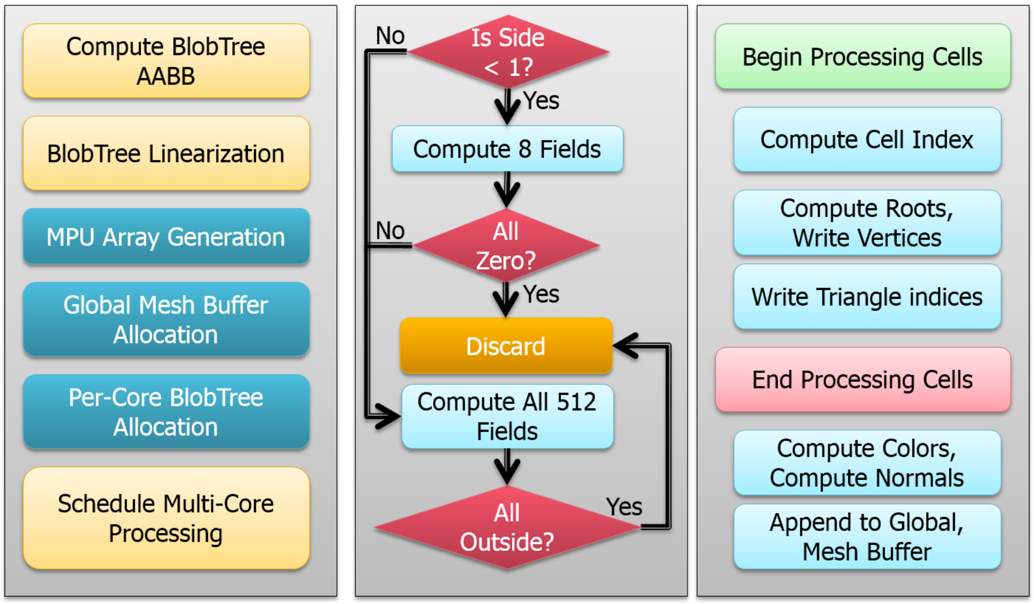
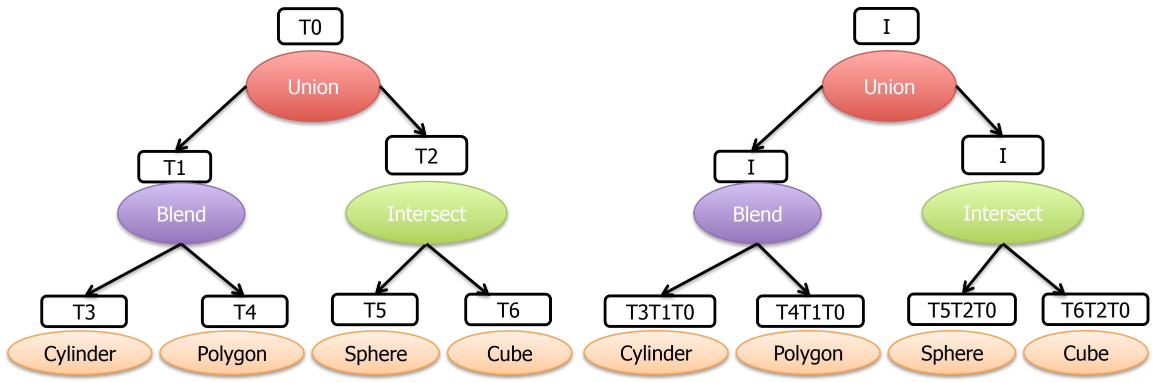


Figure 3.2: Left Column: The one-time preparation steps before scheduling kernel functions for computation. Middle Column: The early discard kernel function. Right Column: The *MPU* processing kernel function.

3.3.1 BlobTree Linearization

The first step in our algorithm is the *BlobTree* reduction and pruning as suggested by Fox et al. [24]. As shown in figure 3.3, after the reduction process, the leaf nodes are associated with the combined transformations of all nodes in that branch of the *BlobTree* . Using this technique the transformation for the internal nodes can be removed since they are simplified to identity matrices.



(a) The original *BlobTree* . (b) The *BlobTree* after the flattening process.

Figure 3.3: The reduction process combines all transformation nodes at primitives. After the operation, all internal operators are associated with identity transformation matrices.

In the second step, using the same linearization algorithm proposed for quadtrees [44]; the *BlobTree* is converted into a pointerless representation to achieve cache-memory efficiency by keeping all input data structures at aligned memory addresses and fitting the entire *BlobTree* model into the last level cache memory of the processor (see figure 3.4).

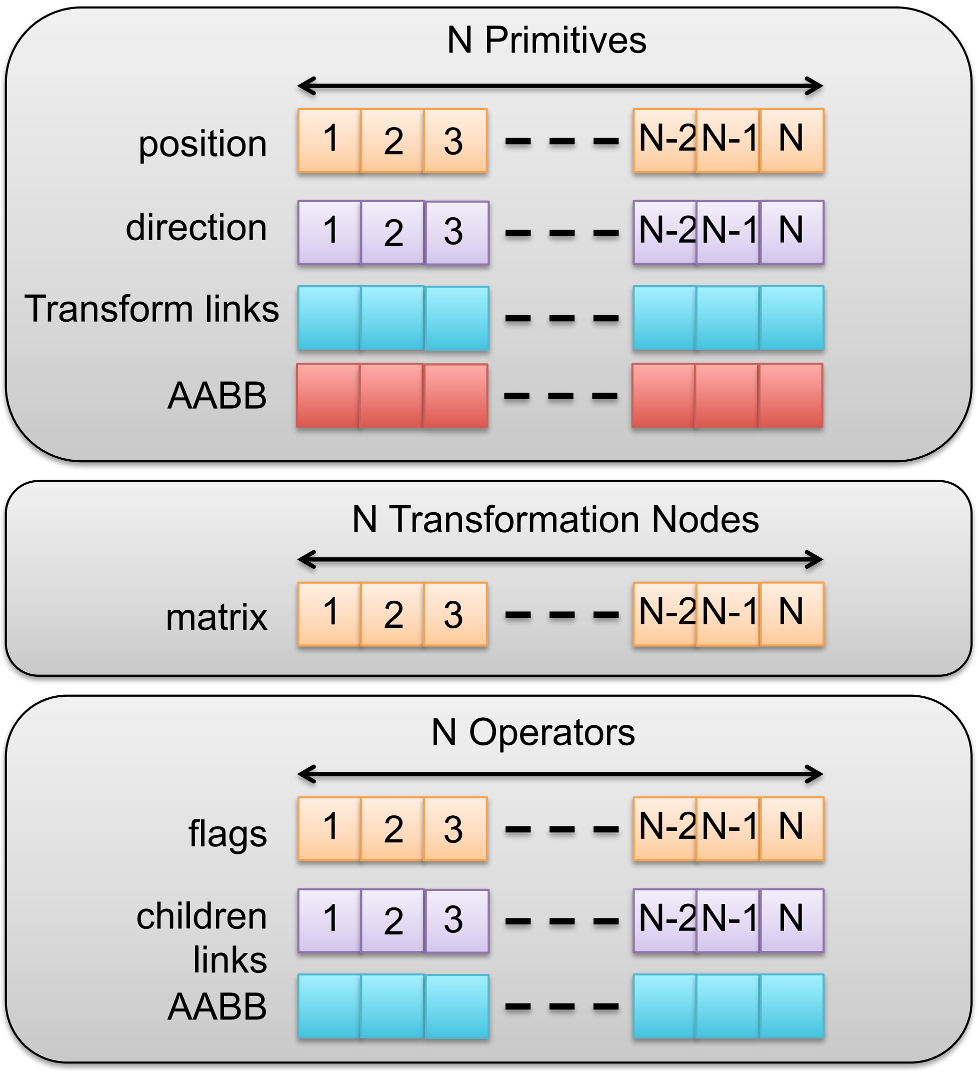


Figure 3.4: The *BlobTree* is flattened to a structure of arrays (SOA), and is stored at aligned memory addresses for performance reasons. Each row represents an array of values of the same type. N is determined in such a way that the entire structure fits in the cache memory of the processor.

The flattening process converts the pointer based *BlobTree* into a structure of arrays. The operators have access to their children using the integer based links for direct access. The operator flags determine the type of children i.e. primitive or operator. Each primitive has an associated link to a transformation node (e.g. link zero is associated with the identity transformation matrix to be reused globally.) N is computed based on the size of memory in the last level cache of the processor. An approximation of N can be computed as following:

$$N = \lfloor \frac{0.6 * LLC}{sizeofOp + sizeofPrim + sizeofTransform} \rfloor \tag{3.1}$$

LLC denotes the size of cache memory available per each core in bytes. Other variables, as their name suggests, are the sizes of one operator, one primitive and one transformation node, respectively. Only 60 percent of the cache storage is assigned for the *BlobTree* structure, since there are certainly other input structures and intermediate variables that need to be stored for the processing on each core.

Using this estimation the computed value for N is 1200 in one of our systems with a total cache memory of 256 KiBytes per core. In our current implementation the memory usage is 96, 56 and 64 bytes per each primitive, operator and transformation nodes, respectively.

The final linearized *BlobTree* is in the format of cache-aligned structure of arrays. With this arrangement several computations can be optimized with SIMD instructions, e.g. applying a transformation matrix on a vector of 4 or 8 vertices as opposed to scalar computation. The output mesh is also in the format of cache-aligned structure of arrays which is the key to compute colors, normals and other attributes in SIMD fashion.

3.3.2 Surface Extraction

In our algorithm we assign field values for every vertex of every MPU that is not trivially rejected with the method explained in the following, and compute the triangular mesh representing the iso-surface. This approach combines elements of several algorithms ([87, 47, 9]).

We extended the method proposed by Zhang et al. [89] to trivially reject all empty MPUs. The observation made is that according to equation 2.4, if the field value at a given vertex is zero then the shortest distance from that vertex to the iso-surface is greater than or equal to one (See figure 3.1). Using this fact empty MPUs can be identified very fast by evaluating the fields at the 8 vertices of each MPU and rejecting it of all 8 fields are zero. However, this test is only applicable when the cellsize parameter is smaller than or equal to 1/7 or 0.1428. For larger *cellsize*s the iso-surface may still intersect with the MPU while the fields at vertices of the MPU are zero. This process is depicted in the middle column of figure (3.2). For a discussion on *cellsize* versus performance see section 3.5.

For larger *cellsize*s we shoot 8 rays from the center of the MPU to its eight vertices, using the technique of Zhang et al. per each step we march $0.866c$ (0.866 is half of the diagonal of an *MPU* with side one and c being the cellsize parameter) along each ray. At each step we compute the fields for the 8 vertices along the rays; if a non-zero field is found then the *MPU* is further processed, otherwise we march along the rays until we reach the vertices of the *MPU*.

If an *MPU* is not rejected then it is further processed for surface extraction. A local copy of the linearized *BlobTree* is provided per each core in order to avoid *false-sharing* among cores [12]. Using SIMD processing techniques field values for all 512 vertices of *MPU* are computed. With SSE or AVX instructions this step requires 128 or 64 field evaluation kernel runs, respectively (figure 3.1).

All the fields are stored in a memory aligned array of 512 floating points. This technique avoids reevaluating field values while processing cells in the next step. Storing field values from a SIMD register into memory aligned address can be accomplished with a SIMD instruction in parallel. After this step all 343 cells of the *MPU* are processed. Per each cell, the 8 vertex field values are gathered in SIMD fashion. Each vertex with a field greater than or equal to *iso-value* is labeled one otherwise zero. The configuration index of the cell is computed using the SIMD method shown in algorithm 1. A configuration index is computed to access the table as in [47]. We used the modified marching cubes table proposed by Dietrich et al. that eliminates many of the degenerate triangles produced in the original MC algorithm [22]. For the ambiguous cases we take another sample from the center of the cell [87, 22].

Algorithm 1 SIMD computation of cell configuration. Pseudo code provided for AVX SIMD computation. Similar code can be written in SSE.

```
1: Gather the 8 vertex field values of the cell
2: simd $index = cmp\_ge8(fields, simd(0.5))$
3: $index = and8(index, simd(1.0))$
4: $index = mul8(index, maskPower)$
5: $index = hadd8(index, index)$
```

In algorithm (1) fields is an array of 8 vertex field values, line 2 performs a parallel comparison between *iso-value* and fields. In line 4 maskpower shifts the field values into the appropriate slot in the SIMD array and finally line 5 performs a horizontal add operation on the values to compute the configuration index.

For each intersecting edge there is one inside and one outside vertex. Using a root finding method the point of intersection of the iso-surface is computed and stored in a hash table to be reused by the neighboring cells that share that vertex.

For the root finding methods that do not require gradient information such as regula falsi or bisection method, the field value should be evaluated multiple times along the edge, which will degrade the performance of the system. Other methods such as Newthon-Raphson require gradient information, and as mentioned in equation (2.5) each gradient computation involves 4 extra field evaluations. We describe a root finding technique based on SIMD instructions that computes the root with only one extra field evaluation in AVX (two with SSE) with adequate precision. By subdividing the intersecting edge into 8 vertices and evaluating the field values, the exact interval containing the final root can be identified. Performing linear interpolation in that interval will produce the final root (figure 3.5), it is trivial to show when the number of intervals increases the interpolation error decreases [49].

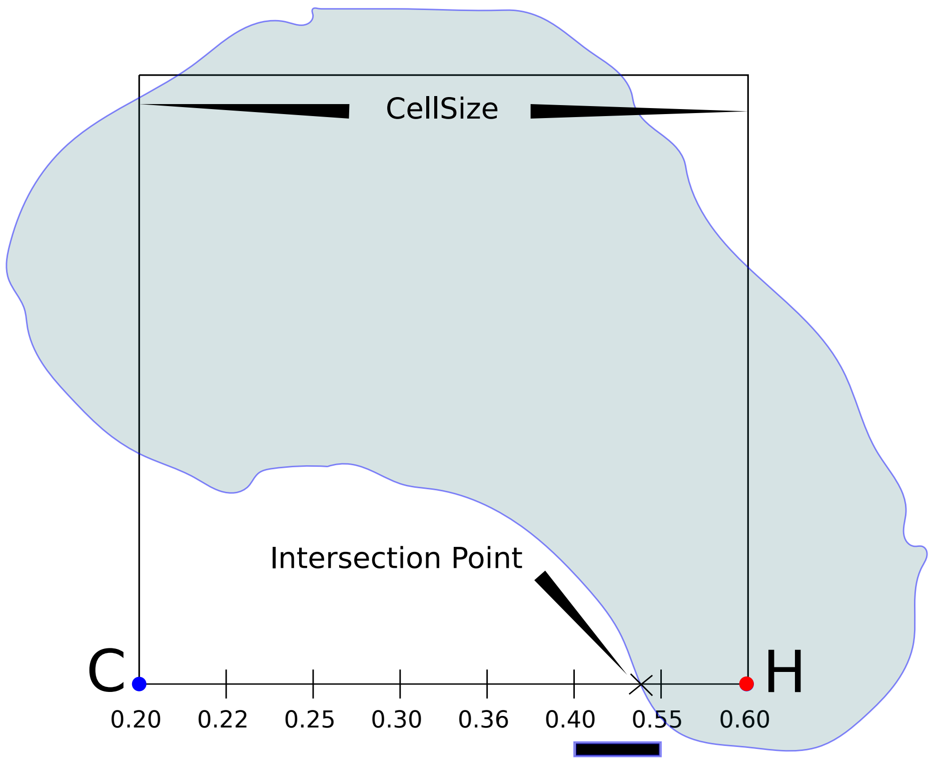


Figure 3.5: Top: A cell edge is intersected with part of the surface shown in blue. By performing one field evaluation using AVX or two with SSE instructions the interval containing the intersection point can be identified. The final root is computed using linear interpolation within the interval marked with bold line segment.

Algorithm (2) summarizes the process of surface extraction which is run per each *MPU*. Lines 1 through 25 are related to the *MPU* discard method explained earlier in this section. Lines 26 through 42 shows the cell processing technique which is optimized using SIMD cell configuration computation and our root finding method. Since color and normal attributes should only be computed for final mesh vertices, this step is performed last to fully leverage SIMD optimizations by performing every 4 or 8 attribute computations in one SIMD call which greatly enhances the throughput of the system and minimizes *BlobTree* traversals.

3.4 *MPU* size Analysis

In a separate experiment we studied the impact of various *MPU* dimension sizes in the overall performance of the algorithm.

As shown in figure 3.1, the default dimension for an *MPU* is 8 along each side. Smaller sizes result in more *MPU* work items and larger *MPU*s consume more processing time per each item. Finding the optimal value for the dimension size is the primary goal of this experiment.

One requirement for the *MPU* size parameter is that it has to be a multiple of SIMD width of the processor that it runs on (4 on SSE and 8 on AVX), otherwise the computations are wasted in the regions outside the boundary of the *MPU* (See figure 3.1).

For this experiment, the model shown in figure 3.15, is polygonized with increasing values of the *MPU* size.

Algorithm 2 Algorithm for surface extraction of an *MPU* using AVX SIMD instructions, Similar code can be written for SSE instruction set. Input is linearized *BlobTree* T, lower vertex of *MPU* and the *cellsize* parameter. Output is the local mesh contained in the *MPU*

```
 1: $side \leftarrow cellsize * 7$
 2: simd $v$ ←Compute $MPU$ vertices
 3: **if** $side \leq 1$ **then**
 4:    simd $f \leftarrow T.compute\_field8(v)$
 5:    **if** $f == 0$ **then**
 6:       return;
 7:    **end if**
 8: **else**
 9:    $flag \leftarrow true$
10:    $incr = 0.866 * cellsize$
11:    $d = incr$
12:    **while** $d \leq side * 0.866$ **do**
13:       Shoot rays from center of $MPU$ to its 8 vertices
14:       simd $v$ ←Travel along the rays for distance $d$
15:       simd $f \leftarrow T.compute\_field8(v)$
16:       **if** $f! = 0$ **then**
17:          $flag \leftarrow false$
18:          $break$;
19:       **end if**
20:       d = d + incr;
21:    **end while**
22:    **if** $flag == true$ **then**
23:       return;
24:    **end if**
25: **end if**
26: float fieldCache[512];
27: **for all** simd $vertex$ in $mpu$ vertices **do**
28:    simd $f \leftarrow T.compute\_field8(vertex)$
29:    Store $f$ in appropriate location in $fieldCache$
30: **end for**
31: **for all** $cell$ in $mpu$ **do**
32:    $f \leftarrow gather8(cell, fieldCache)$
33:    $edges$ ←Compute cell config from $f$ to access table
34:    **for** $i = 1$ →count of edges **do**
35:       **if** root for $i$th edge is not stored in edge table **then**
36:          Compute and store root associated with $i$th edge
37:          Add root to mesh vertices
38:       **end if**
39:    **end for**
40:    Add $cell$ triangles to mesh
41: **end for**
42: compute color and normal for all vertices (every 8 vertices in parallel)
```

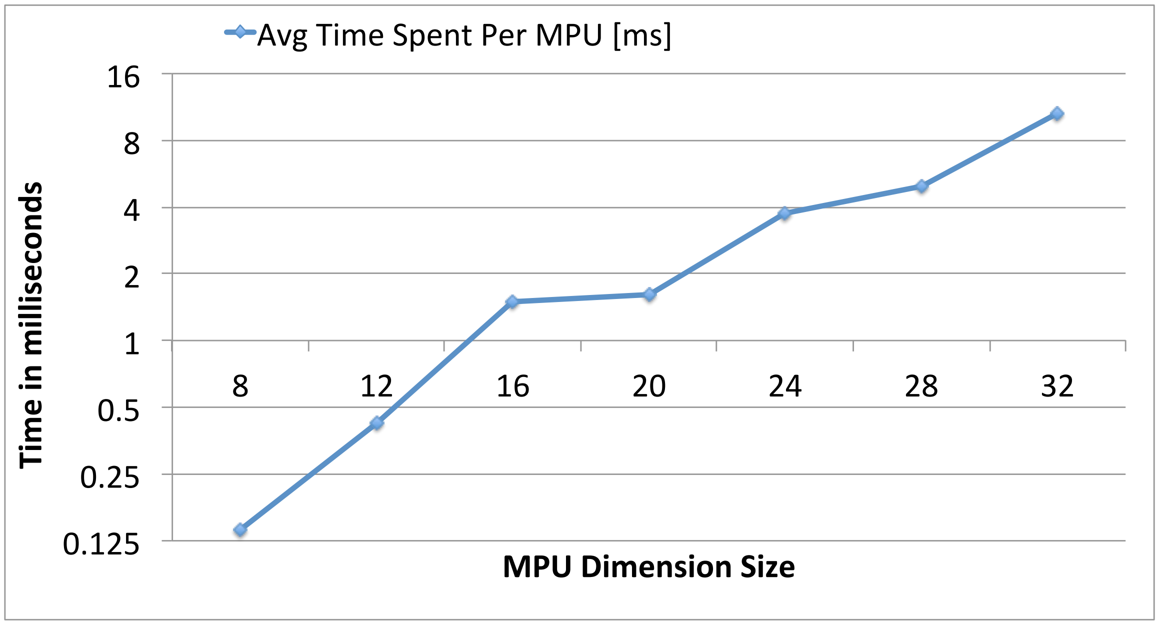


Figure 3.6: Average time spent per each MPU when polygonizing the Towers model.

The MPU dimension is increased from 8 to 32 in the steps of 4 units at a time. As shown in figure 3.6, the amount of processing time spent per each MPU is increased as larger units of work are being processed.

The fastest polygonization time is when the MPU dimension is set to 20 in this experiment. This is depicted in figure 3.7. All the numbers used in these graphs are the average of 10 runs.

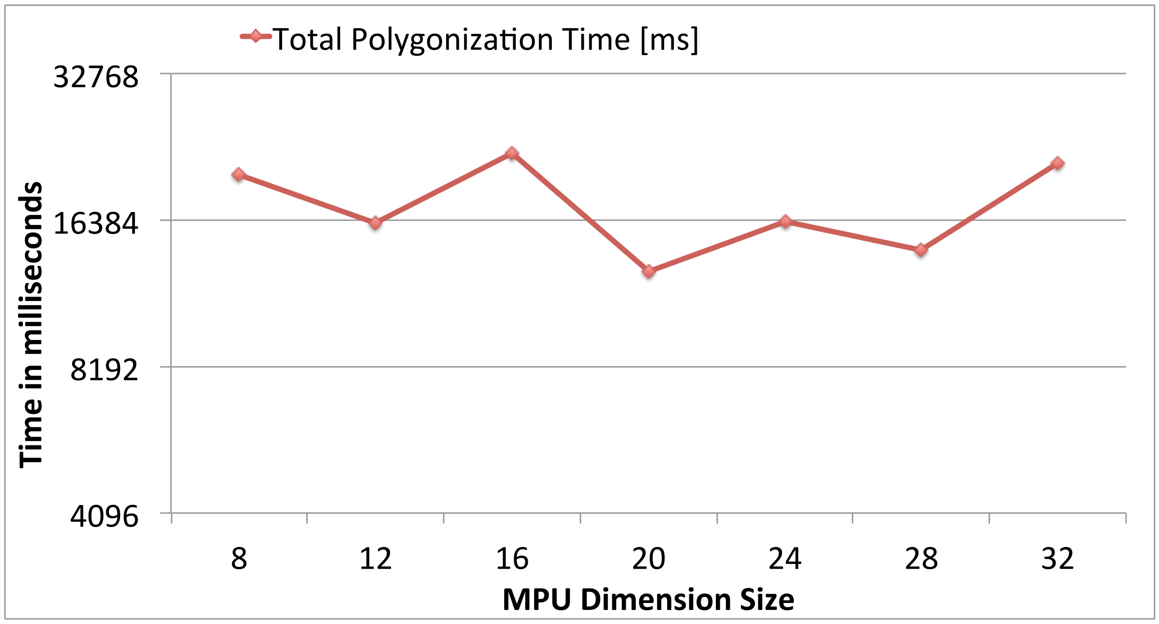


Figure 3.7: Total polygonization time in milliseconds when polygonizing the Towers model.

By keeping the cellsize parameter as constant, the early discard method described in section 3.3.2, is used only in case of a *MPU* with the dimension size of 8. Since the side length is still smaller than one with this dimension.

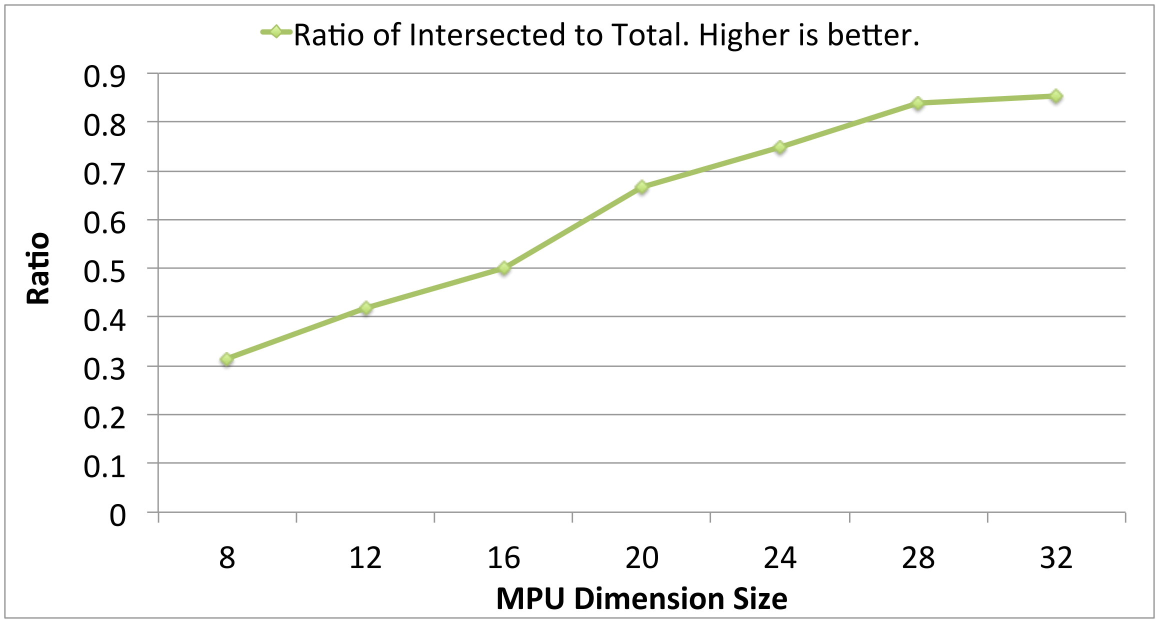


Figure 3.8: Ratio of intersected to total number of *MPU*s.

One of the interesting graphs is figure 3.8, which illustrates the ratio of intersected *MPU*s to total count as the dimension size increases. The highest ratio (more than 85 percent) is achieved when the largest dimension size (32) is used for processing.

However as shown in figure 3.9, increasing the *MPU* dimension will adversely widen the time difference between all threads to finish their processing. The reason for this phenomenon is that when the units of work are large, having even a few more of them in the processing queue of a thread can lead to a major imbalance in the workload distribution among all threads.

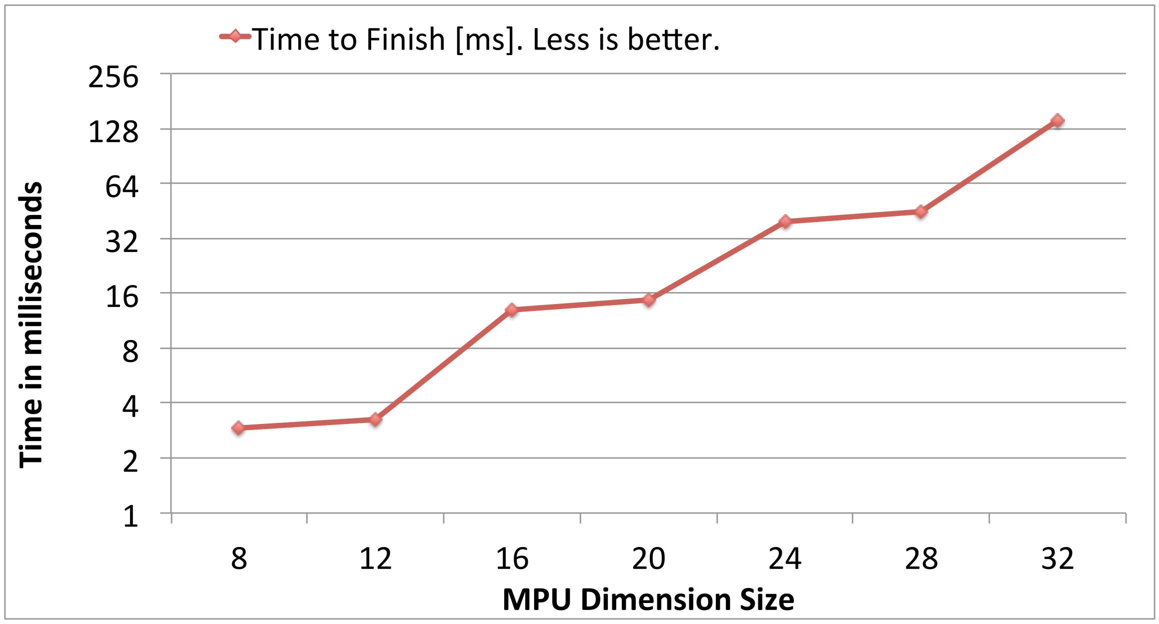


Figure 3.9: Maximum time in milliseconds spent in idle mode while other active threads finish their current work. The less is better, since it indicates a more uniform workload distribution across the running threads.

Figure 3.9 exhibits an expected behavior: With the dimension size of 8 there are as many as 145152 *MPU*s for processing, by refering to graph 3.6 the time spent per each item in this case is a fraction of a millisecond (approximately 140 microseconds).

In case of the dimension 32, a *MPU* is 64 times larger than the one with dimension 8 and the total count suddenly drops to 2023 and with the average processing time of 10 milliseconds per each *MPU* (10 milliseconds is almost 72 times larger than the amount of time reported for a dimension 8 *MPU*).

The last graph in this section is associated with the amount of physical memory required to store the input data-structures for processing a unit of work as efficiently as possible. We made an effort to fit all the required data-structures inside the cache memory available on the processor cores. One of the main reasons that we chose the default dimension size of 8 for the *MPU* is the fact that this memory is smaller on the older generations of processors and increasing the dimension size will increase the required memory as shown in figure 3.10. However, our algorithm is scalable and for modern processors with access to larger cache memory, the higher *MPU* dimensions can be used.

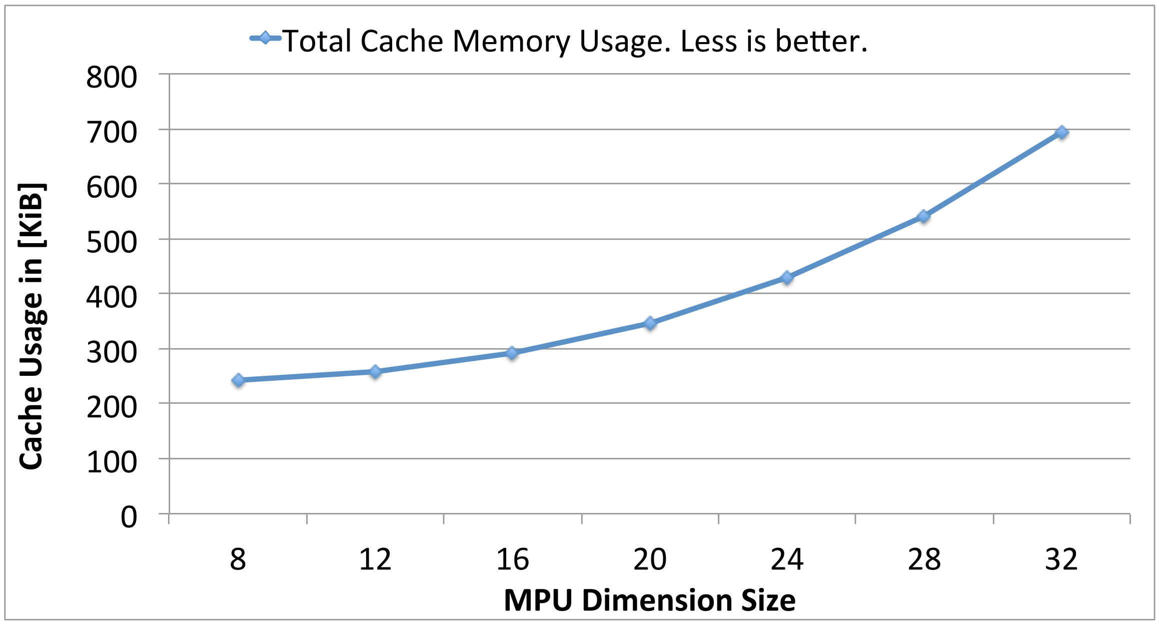


Figure 3.10: Total memory usage per each *MPU* processing, reported here in kilobytes. The limit in this case is the last level cache memory available on the processor to fit the entire data structures required for the processing of a *MPU* efficiently.

3.5 Results

We have implemented our algorithm using Intel threading building blocks in C++ on a Linux platform. We used two systems with different configurations. On the first system which has Intel i7-3960X processor with Sandy Bridge architecture, there are 6 physical cores given that each core runs in hyper-threaded mode; up to 12 threads can run in parallel on this machine. This processor supports both SSE and AVX instructions and there is a last level cache memory of 15 megabytes which is shared between all cores.

The second system is a server with 4 Intel X7560 processor with Nehalem architecture. Each processor has 8 physical cores or 16 in hyper-threaded mode and has

24 megabytes of last level cache memory and it does not support AVX instructions. Together these 4 processors provide us with as many as 32 physical cores (64 when hyper-threaded) on this server. We refer to these two systems with SNB and NHM respectively.

On the first experiment our goal was to prove the scalability of our algorithm. Figures (3.11, 3.12) show the average running time of the algorithm when rendering towers model (figure 3.15) on SNB and NHM systems, respectively. The *BlobTree* of the towers model has 7360 operators and 7296 primitives and a depth of 64 levels. In this test the cellsize parameter kept as a constant value of 0.14 which we found it to be a balance between number of triangles produced and the quality of the output mesh.

In order to show the effect of SIMD optimizations we have tested our algorithm with scalar, 4-wide SSE and 8-wide AVX instructions. SSE being on average 4.58x faster than scalar and AVX being on average 7.35x faster than scalar run. As illustrated in figure 3.11 when the number of threads increases past 6, two threads run on every core; sharing hardware resources on the hyperthreaded cores. The slope is reduced because each thread gets less resources than it would if it ran alone on the core. Past 12 threads, we schedule multiple threads per core, and they start to thrash the cache; making the algorithm memory bound.

Figure 3.12 shows the performance of our algorithm when running on the NHM system. Doubling number of threads, doubled the performance of the algorithm on this machine up to 33rd thread. The same behavior is shown and hyper-threaded cores start to compete for memory access when having more than 32 threads running on this machine.

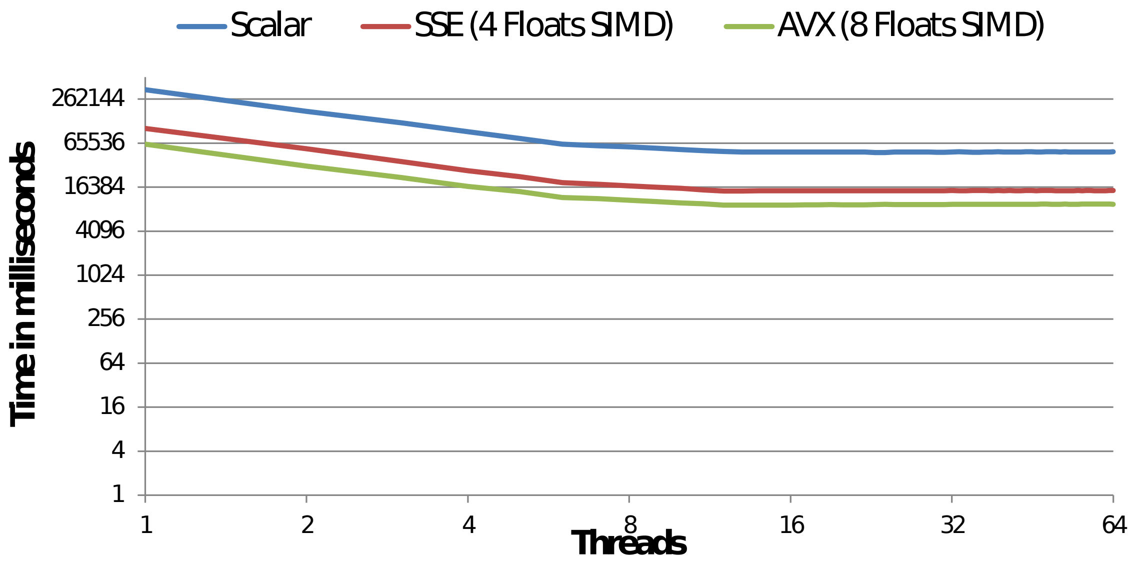


Figure 3.11: Average polygonization time of the towers model when running on SNB processor. Horizontal axis is the number of threads. Vertical axis is time measured in milliseconds.

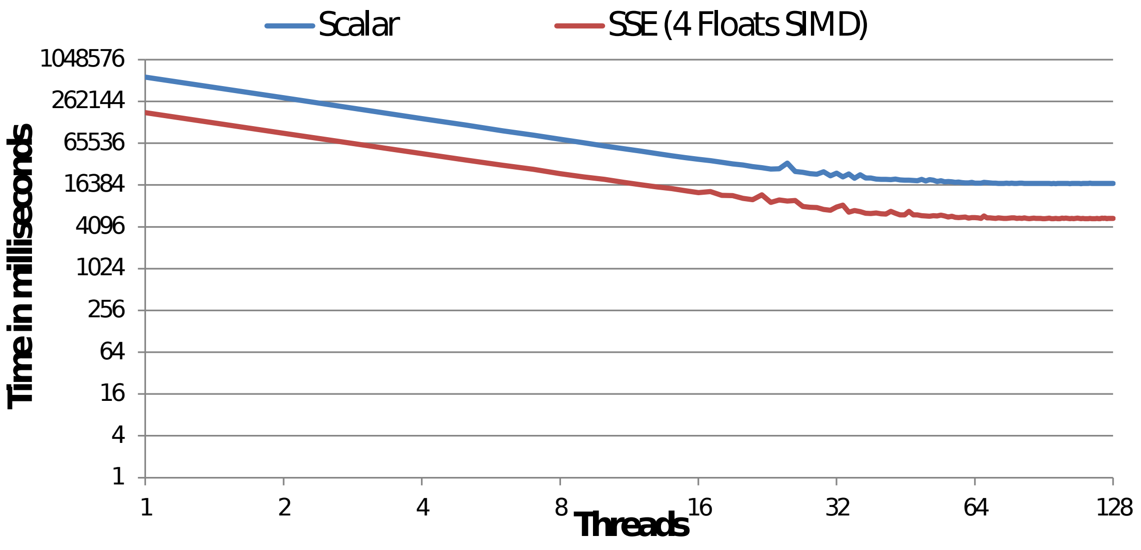


Figure 3.12: Average polygonization time of the towers model when running on NHM processor. Horizontal axis is the number of threads. Vertical axis is time measured in milliseconds.

Table 3.1: Comparison of speedups and field value evaluations per triangle ($FVEPT$) for polygonization of Tower model with different SIMD instruction sets. Note that FVEPT was 17 before adding SIMD optimizations.

Processor	SIMD Method	Speedup	FVEPT
SNB	SSE	4.58x	4
SNB	AVX	7.35x	2
NHM	SSE	4.25x	4

Table 3.1 shows the effect of using SIMD optimizations in our algorithm. With SSE and AVX the theoretical speedups are 4 and 8 times, respectively. Due to memory alignment techniques and proper caching mechanisms the speedup with 4-wide SSE is greater than 4. The AVX speedup can be improved more once scatter/gather instructions are implemented on the SNB processors which will improve the performance of surface extraction algorithm. Number of field evaluations per triangle shows the average amount of times the field evaluation kernel called to compute a single vertex in the output mesh.

In another experiment we studied the effect of our early discard method when the side of each MPU is less than one (figure 3.13). Starting from a large cellsize, we reduced the cellsize in uniform steps and measured the polygonization time. The red curve shows the polygonization time when the discard method described in section 3.3.2 is not being used and the blue curve is the timing when that method is in effect. Note that with the blue curve as soon as the MPU side is less than one; ($cellsize = 0.14$) empty MPUs started to get discarded efficiently thus the constant part of the time value is reduced at that point.

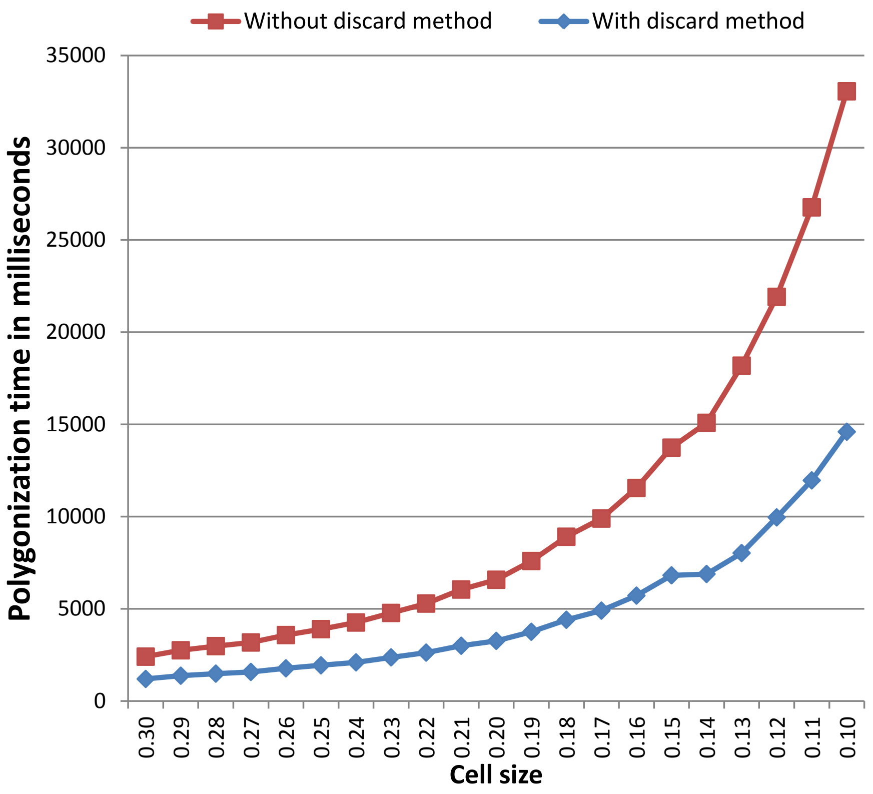


Figure 3.13: Reducing cellsize parameter results in more MPU generation and increase in polygonization time. However, at a certain cellsize our early discard method stops polygonization time increase by rejecting all empty MPUs more efficiently.

Figure 3.14 shows the polygonization time breakdown when rendering the towers model on SNB processor. Horizontal axis is the core number for a total of 12 cores on that system. As can be seen from the top of this chart; the idle time is very short and the cores are active almost all the time. This shows that the work stealing algorithm scales well. 190463 MPUs are processed and 116723 of them are intersected with the iso-surface (40 percent were empty). 40 percent of the $MPUSET$ has been processed in less than 10 percent of the total polygonization time.

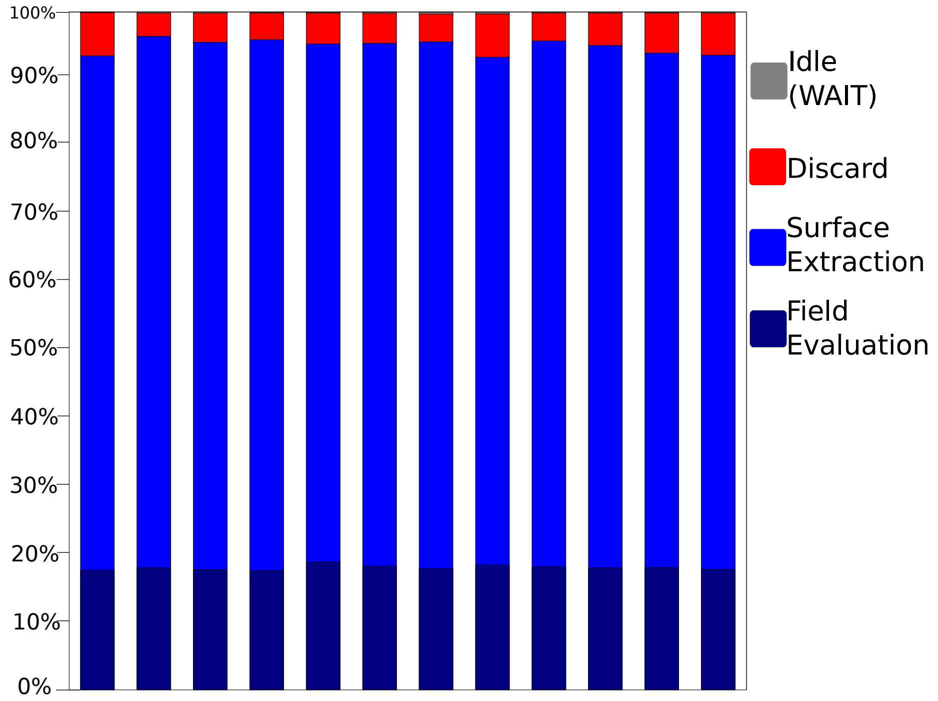


Figure 3.14: Towers model per-core time breakdowns. Each bar represents a logical core on the processor for a total of 12 cores. Vertical axis is the total polygonization time. 190463 MPUs processed with 12 cores in 9283 milliseconds. This chart shows the portion of time spent in each step of the algorithm when rendering the towers model on the SNB processor with 8-wide AVX instructions.

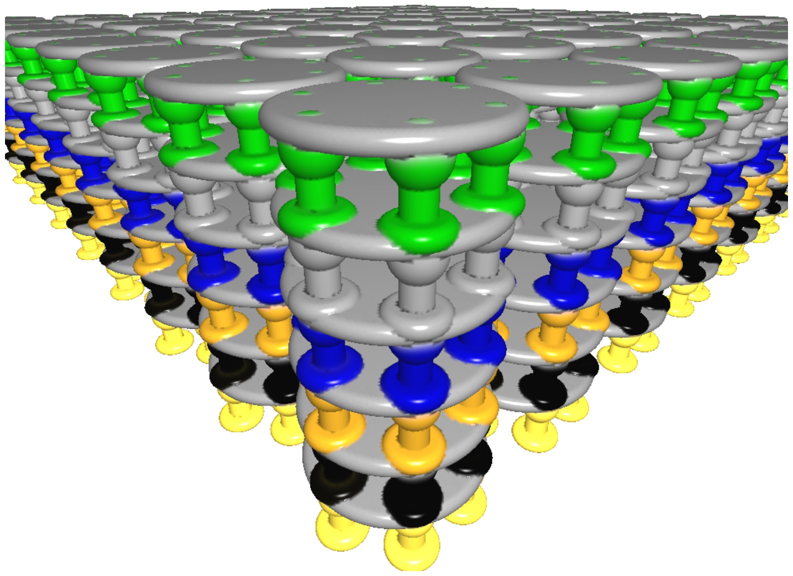


Figure 3.15: Towers model created with skeletal primitives and binary operators in our incremental designing system. The model is a grid of 8 by 8 towers for a total of 7360 operators and 7296 primitives.

These results demonstrate the scalability of our algorithm both in the number of SIMD vector lines and the number of cores available on each processor.

Finally, we compare our method against Schmidt et al. 's [62] using the Medusa model provided by them which has 2920 primitives and 11 operators and the tree structure has a depth of 6 (figure 3.16).

The Medusa model is programmatically reconstructed in our modelling system and the results of its polygonization are already published by Schmidt et al. [62].

We divided polygonization timings reported in [62] by 8 as the best AVX optimized version of Schmidt's method. Then we ran our polygonization algorithm optimized with AVX instructions on a single core for Medusa model (See table 3.2).

Table 3.2: Comparison of our polygonization method against Schmidt et al. 's [62] when rendering Medusa model at 5 different resolutions on one single core with AVX instructions. All timings are in milliseconds.

CellSize	Our method	Schmidt's method	Speedup
0.01	5220	6228	1.19x
0.03	3441	3653	1.06x
0.05	1071	2175	2.03x
0.10	264	1292	4.89x
0.14	108	721	6.67x

The results shows that our algorithm outperforms that of Schmidt et al. by a factor of 6 when running on a single core in lower resolutions.

3.6 Chapter Conclusions

In this chapter we presented a new parallel polygonization algorithm using SIMD processing techniques that takes advantage of a multi-core machine. Our main contribution is a scalable algorithm both in terms of the number of cores available on multicore architectures and the number of SIMD vector width as shown in the results section. We also presented a SIMD technique for finding the intersection of an iso-surface and a cube edge.

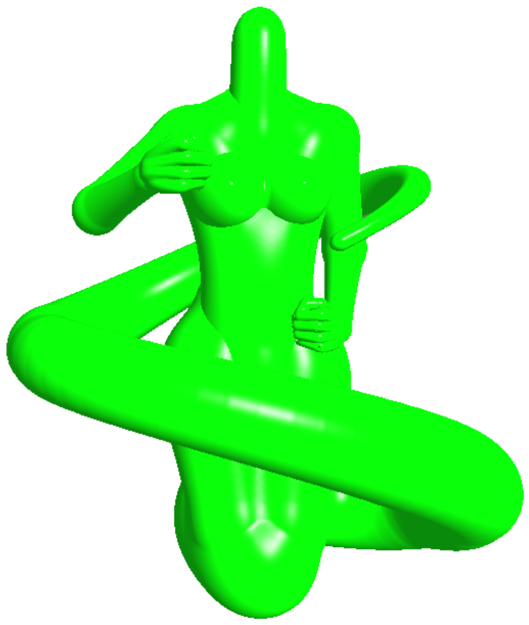


Figure 3.16: Medusa model courtesy of Schmidt et al. [62].

Chapter 4

GPU discretization

In the previous chapter we presented an algorithm for polygonization of the *BlobTree* scene graphs on multi-core architectures. The output of that algorithm is a list of vertices with their associated attributes such as position, color and normal and a list of triangles that defines the connectivity of the surface mesh. While the surface mesh is used for rasterization it can also facilitate collision detection and contact modelling algorithms as we will see in the following chapters. In this chapter we present an improved version of that algorithm that can take advantage of the processing capabilities of the GPU and provides scalable rendering performance on many-core architectures.

Building on top of the results from our SIMD polygonization algorithm, in this chapter we present our work in optimizing that algorithm for many-core architectures such as GPU. The following are some of the reasons behind this effort:

1. The polygonization process extracts a triangle mesh which ultimately requires rasterization for rendering. Using a GPU implementation both steps can be fused together thus avoiding the expensive read-backs.

2. GPUs provide a higher degree of SIMD processing which is also called SIMT for single instruction multiple threads. Using the same strategy, multiple units execute the same fetched and decoded instruction; however, on the GPU side there is a wider access to register sets, multiple flow paths and access to multiple memory addresses that can scale the performance of each running thread significantly.

3. Using flexible and portable OpenCL programming, the polygonization code can be written in simple, scalar arithmetic and the runtime environment will handle

multi-threaded task parallelism and vector width SIMD processing on the target hardware. This is certainly much more readable and extensible code than using non-portable, machine intrinsic functions for SIMD processing.

When compared to the related work in the field, our proposed method has the following benefits:

- It is dynamic, i.e., the input to our algorithm is a dynamic *BlobTree* structure that can change in every frame. This is a key advantage that can enable interactive modelling sessions in an implicit framework, i.e. the result of modifications to the model can be viewed in real-time. After each change the associated meshes are updated dynamically.
- It is high performance. The proposed GPU-based data-structure is compact enough to enable rendering complex *BlobTree* models in the order of 60,000 nodes. (64,000 nodes only requires about 20MiB of video memory in our system).

The next section provides an overview on the design differences between a CPU and a GPU. Other considerations for memory and access patterns are discussed in that section. Next the data-structure and the new algorithm are explained. The chapter concludes with the results and analysis.

4.1 Many-cores architectures

One of the main differences between a multi-core CPU versus a many-core GPU architecture is the fact that the former is optimized for the fast execution of a single task by supporting a broad-set of instructions in hardware, where as the latter can accommodate heavy parallel processing in the format of single instruction multiple data (SIMD).

CPU designers incorporate complex multi-cycle pipeline methodologies to amortize the amount of computation in a single machine cycle, e.g. by overlapping the execution of independent instructions at different cores, all cores of a processor are kept busy working on a different instruction. GPU designs take a different approach to parallelism. They achieve higher SIMD throughput by assigning more of the chip area to arithmetic logic units (ALUs) and less to fancy branch predictions and flow control. GPUs also have access to higher memory bandwidth when compared to their CPU counterparts.

The advent of programmable graphics pipeline opened up new avenues to exploit computational resources of the GPU hardware. Before the introduction of general-purpose GPU (GPGPU) programming languages, such as open compute language (OpenCL), researchers in the field were forced to use computer graphics shading languages (e.g. glsl, cg) to gain access to the many-core architecture of GPUs.

Many of the terms associated with solving rendering issues were borrowed to explain solutions to problems in other domains of science. Terms such as vertex, fragment and texture which define entities in a graphics shading problem were used to refer to the processing threads, work items and the input for a solution to a problem in high performance computing domain. In addition to the lack of a general language for computing on the GPUs, the complexity of mapping the problem domain to the graphics pipeline complicated the implementations and lead to various ad-hoc solutions to the same problems. For instance, Georgii et al. created an interactive cloth simulation by implementing a mass-spring system using fragment shader programming on an ATI X800 graphics card [28]. As another example Sørensen et al. reviewed several shader programming techniques for accelerating linear system solving on GPUs with applications in surgical simulation [71].

At the time of this writing, the following languages are commonly used for GPGPU computing:

- Nvidia CUDA (Compute Unified Device Architecture)
- Microsoft's DirectCompute
- OpenCL which is a standard language maintained by Khronos Group

Among all, OpenCL is the only language that is designed to be platform and host operating system independant, DirectCompute supports all hardware vendors but is limited to Windows and CUDA is only available on Nvidia's hardware.

OpenCL and CUDA share similar concepts in their programming model [27]. One of the major drawbacks of OpenCL when compared to CUDA is the initialization delay associated with the kernel compile process at runtime. The delay is significant for long and complex kernels and at the time of this writing, no offline compiler is offered by Nvidia to alleviate this issue. One way to improve this is to compile and store the binaries on disk upon the first kernel invocation and reuse the precompiled binaries for subsequent executions.

The other technical issue that we faced with OpenCL is that the performance optimizations made on one platform are not transferred to other platforms. This situation has been experienced by other researchers in the field and essentially there is no guaranteed portability for the tuning and optimizations of the kernels. The algorithms that are presented in this chapter are implemented using OpenCL.

OpenCL and CUDA use similar memory model to execute kernels on the GPU. Each thread of execution on the GPU has access to a limited amount of local memory that is private to that thread. The local memory for the thread is used to store the stack frames. It is also used in register spilling, which occurs when the local and private variables of the thread can not fit into the register file of the physical core and they spill to the local memory. Each thread belongs to a thread block. In OpenCL terms a thread is referred to as a processing element and a thread block is called a compute unit.

All processing elements within a compute unit has access to a shared memory block, which is accessible by all processing elements within the same block. Beside the local and shared memory spaces, all processing elements have access to a global memory block which has the slowest access time when compared to the other two spaces.

Physically, the local and shared memory spaces can be seen as level 1 and level 2 cache memory on the CPU and the global memory as the main memory (i.e. video memory on the GPU).

4.2 Data Structures

The *BlobTree* linearization step introduced in (section 3.3.1) is modified to create a compact representation of the input model in our GPU polygonization algorithm.

All the structures are aligned at 16 bytes (four floating points) memory addresses. This is similar to the texture accessing techniques in graphical shader programming languages such as GLSL or Cg, where four floats represent the RGBA values of a texel accordingly. If a primitive node has an associated transformation node with a non-identity transformation matrix, that matrix will be stored in the primitive matrices section of the input structure and an associated identifier (id) will be provided to the primitive. The inverse of the primitive matrix is computed and the first three rows are stored for further field computations. To transform the axis-aligned bounding boxes of the primitives, the full forward transformation matrix is stored in the box matrices section of the structure and it can be accessed using the same id provided for the primitive matrix. Figure (4.1) depicts this pointerless representation in details.

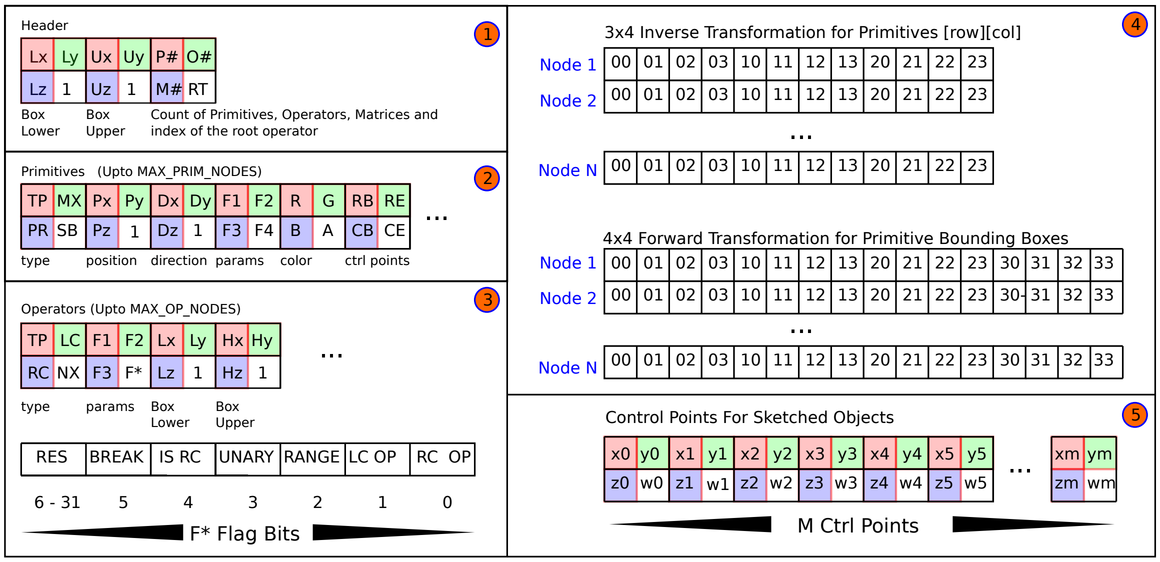

Header
Lx Ly Ux Uy P# O#
Lz 1 Uz 1 M# RT
Box Lower
Box Upper
Count of Primitives, Operators, Matrices and index of the root operator
1
Primitives (Upto MAX_PRIM_NODES)
TP MX Px Py Dx Dy F1 F2 R G RB RE
PR SB Pz 1 Dz 1 F3 F4 B A CB CE
type position direction params color ctrl points
2
Operators (Upto MAX_OP_NODES)
TP LC F1 F2 Lx Ly Hx Hy
RC NX F3 F* Lz 1 Hz 1
type params Box Lower Box Upper
3
RES BREAK IS RC UNARY RANGE LC OP RC OP
6 - 31 5 4 3 2 1 0
F* Flag Bits
3x4 Inverse Transformation for Primitives [row][col]
Node 1 00 01 02 03 10 11 12 13 20 21 22 23
Node 2 00 01 02 03 10 11 12 13 20 21 22 23
Node N 00 01 02 03 10 11 12 13 20 21 22 23
4
4x4 Forward Transformation for Primitive Bounding Boxes
Node 1 00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33
Node 2 00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33
Node N 00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33
Control Points For Sketched Objects
5
x0 y0 x1 y1 x2 y2 x3 y3 x4 y4 x5 y5 xm ym
z0 w0 z1 w1 z2 w2 z3 w3 z4 w4 z5 w5 zm wm
M Ctrl Points

Figure 4.1: The compact *BlobTree* scenegraph representation for GPU polygonization and tetrahedralization algorithms. The structure is aligned at 16 bytes (4 floats). 1- The header. 2- Skeletal implicit primitives. 3- Operators. 4- Affine transformation nodes. 5- Control points for sketched objects. Refer to section 4.2 for details.

Each input data-structure is numbered in figure (4.1) for further reference. We review the details in the following:

1. The header section defines the lower and upper corner of the axis-aligned bounding box enclosing the entire model, i.e. the convex hull of the input *BlobTree* . The header also contains the count of primitives, operators and the transformation nodes. The id associated with the root operator of the tree is also defined in the header.

2. The definition of each primitive is encoded in 6 texels. The TP field in the first texel numerically encodes the type of skeletal primitive. The other three fields in this texel are MX, PR and SB that link this primitive to its inverse transformation matrix, the parent and the sibling elements respectively. The following texels define the position, direction, skeleton-specific parameters and the color of the primitive. For the primitives that are computed from sketched

control points (e.g. the thin-plate spline primitives [78, 31]) the indices to the associated first and last control points are stored in the last texel.

3. A *BlobTree* operator is defined in 4 texels. Similar to the structure shown in figure 3.4, links are provided to the child nodes. Other fields added to support our stackless *BlobTree* traversal algorithm which is explained in the following sections. The NX field defines the next operator node in the *BlobTree* traversal route. The F^* field contains the flag bits that provide more control over the operator and the definition of each can be found at the bottom of figure 4.1. Bits 1 and 0 are set in case the left child or the right child of the current node are operators as well. Bit 2 is set if the *LC* and *RC* indices are actually defining a range of indices for primitive children. If the unary flag is set then the operator has only one child which is stored in the *LC* field. Bit 4 is set when the current operator appears as a right node for its parent. Bit 5 is the break route flag and is discussed further in our stackless *BlobTree* routing algorithm. The rest of the bits are reserved for future use.

4. As mentioned above the inverse of the transformation matrices are computed and the first three rows are stored in our input structure for field computation purposes. The elements are depicted in the format of [row][column]. The forward transformation is stored as a 4x4 matrix in a separate input structure for performance reasons.

5. The control points associated with the sketched primitives are all stored in this section. Each control point is defined with their XYZ coordinate and an associated weight value.

4.3 Memory foot prints

After analyzing the performance of our OpenCL kernels, register spilling often found to be the major cause for many latencies. Resorting to shared memory spaces, reducing number of intermediate variables in field-evaluation kernels and avoiding loop unrolling in some cases helped to optimize the performance. Upon every change in the input *BlobTree* the linearized *BlobTree* is updated and transferred from main memory to the GPU. Currently the maximum number of nodes is set at 64K which covers all of the complex cases we modelled for this thesis. However, for larger *BlobTree* models we can easily increase this amount to support them. The memory footprint of our current *BlobTree* representation is summarized in table 4.1. Each row in the table represents a *BlobTree* with a specific number of nodes. Starting from the simplest *BlobTree* with only one node (e.g. a sphere primitive) to the most complex *BlobTree* with one million nodes. The middle columns in the table are the break down of the required memory size per each component in the compacted *BlobTree* data structure.

Table 4.1: Memory footprint of the input *BlobTree* in our GPU polygonization algorithm in bytes. The entire *BlobTree* for a model with 64K nodes (primitives and operators) takes up about 20 MiB in our current system.

Nodes	Header	Primitives	Operators	Prim Mtx	Box Mtx	Ctrl Points	Total
1	48	96/Prim	64/Op	48/Prim	64/Prim	16/Point	320
1K	48K	96K	64K	48K	64K	16K	320K
64K	3072K	6144K	4096K	3072K	4096K	1024K	20M
1M	48M	96M	64M	48M	64M	16M	320M

4.4 Stackless *BlobTree* traversal

As shown in figure 3.14 the major bottleneck of the algorithm presented in chapter 3 is the surface extraction process. The expensive operations in that stage is the computation of normals and colors which require extra field evaluations and hence performance degradation.

In this section we describe our novel stackless *BlobTree* traversal algorithm. Without a stack to be maintained the *BlobTree* traversal incurs less memory footprint. When dealing with small local memory available per thread on the GPU this reduction in memory usage improves the performance of the running kernels significantly. The idea behind the stackless traversal is to pre-compute a serial route to visit and evaluate all the operator nodes in the tree without using a global temporary storage to keep track of the visited nodes. An index based data structure is designed to provide sibling node access as well as two-way parent child relationships. This type of links help to perform horizontal and vertical moves in the tree structure.

In high performance rendering algorithms the use of hierarchical spatial data structures for acceleration reasons is common. Visiting nodes in such structures requires a stack-based depth-first search (DFS) traversal algorithm. Unfortunately, even the latest GPU architectures are poorly suited for implementing such algorithms.

A complex *BlobTree* scene-graph data structure may contain thousands of primitives and operators which can lead to deep tree structures. Using the original DFS algorithm, the fieldvalue evaluation process has two stages: A "down traversal" followed by an "up traversal".

During the down traversal stage all operators starting from the root node are visited and pushed onto an operators stack until a leaf node (primitive) is reached at which point the field due to the primitive is computed and pushed onto a separate fields stack and the next stage which is the up traversal begins.

During the up traversal stage the operators are popped out of the stack and per each child an associated field is popped from the fields stack for the operator to combine them in its own specific way. The resulting field is again pushed back on the fields stack. This process continues until the operators stack get emptied and the final fieldvalue due to the root node is computed and returned.

Performing many push and pop operations limit the performance of the traversal process. The other issue relates to the inherently dynamic storage requirement for the stack itself. Although, creating a fixed-size stack is possible but since the stack size is a function of the number of nodes in the input, this will ultimately limit the maximum number of nodes in a complex scene. If N is the maximum number of nodes (primitives plus operators) allowed in a *BlobTree* model then the minimum number of elements to be stored in the stack during the traversal process is equivalent to the depth of the tree. Implementing a stack in the global video memory will require very expensive memory transfers and is not an option for a real-time rendering algorithm.

Our algorithm is based on the neighbor cell-links concept in the stackless traversal of spatial subdivision trees which is first introduced by Samet et al. [59, 60]. Using a similar technique Popov et al. presented a stackless KD-Tree traversal for high performance GPU ray tracing algorithm [56].

The novelty of our technique is in the route computation and evaluation process which is completely different than what is proposed in Samet's and Popov's work. In our system, the geometry is not explicitly defined in the input structure at the time of route computation. Both Samet's and Popov's algorithms are based on spatial data-structures that provide explicit access to the geometry and therefore it is possible to recompute the missing sibling and parent-child relationships if they are not pre-computed. The second difference is the fact that not all the child nodes in accelerator structures such as KD-Trees or BSP-trees need to be visited to answer an inclusion query, but in case of a *BlobTree* structure all nodes are visited in order to compute the final field due at a point in space. Once the field is computed then it is trivial to answer the inclusion query. A third difference is in the order of evaluating the operator nodes. Some operations such as difference, are not commutative and need a specific handling when computing the route. Such differences require special handling that clearly show the need for a different tree traversal algorithm.

The algorithm is divided into two stages. A preprocessing stage to compute a traversal route for the entire *BlobTree* and the GPU-based stackless traversal stage to compute the field-value due to a given point in space. The following will describe these two stages in greater details.

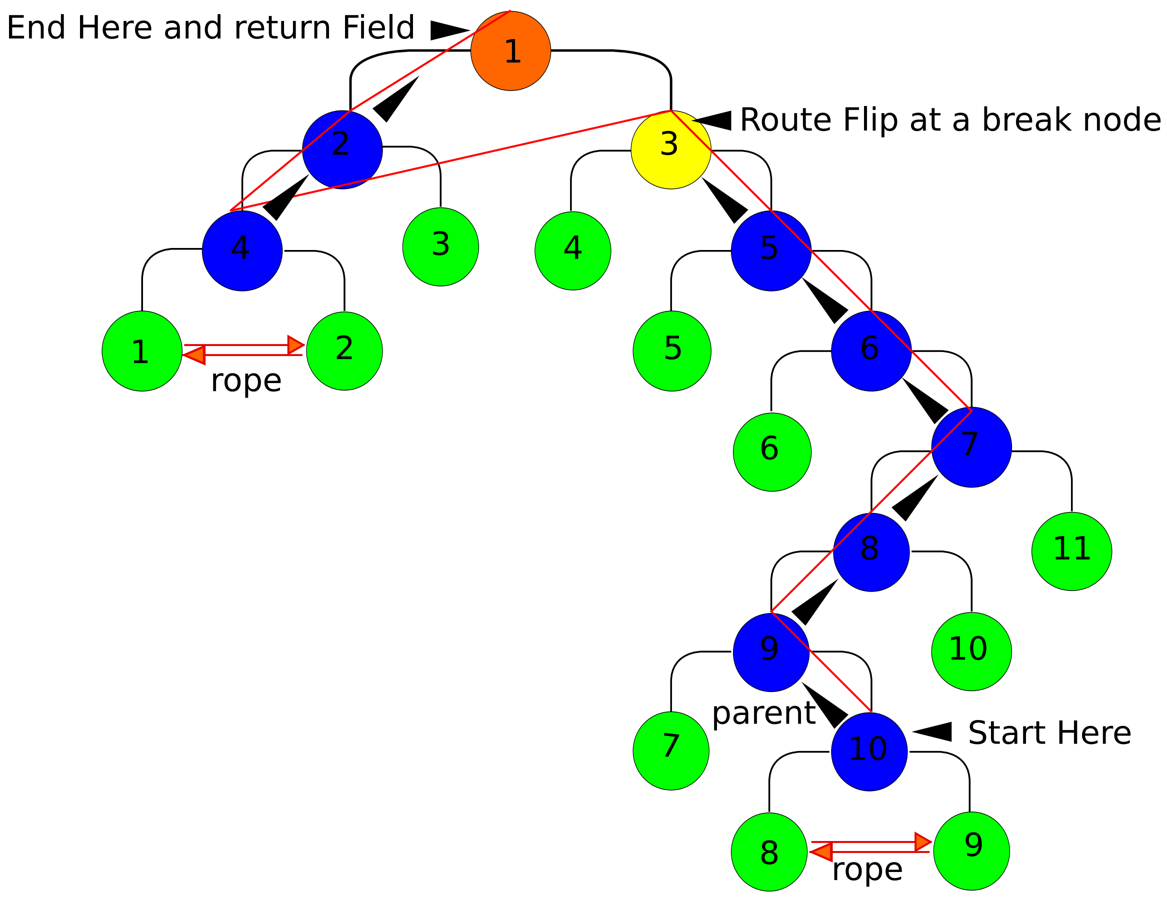


Figure 4.2: Stackless *BlobTree* traversal algorithm performs faster on deep tree traversals. The route is computed once and encoded into the tree upon transferring the input data structures to the GPU.

Encoding traversal route

The first stage in our algorithm is to compute a route to visit all nodes in the *Blob-Tree* and finally encode that route in the GPU input data-structure. This process is performed only once and is not a bottleneck in the system. The one-time fixed cost paid for this stage on the CPU is regained when many fieldvalue evaluations are performed in parallel on the GPU without the extra cost associated with the stack storage.

To compute the route the following steps are performed on the CPU side as a preprocessing stage:

Using the naive *BlobTree* traversal algorithm the nodes are visited from root to leaves. Two stacks are maintained in this process, an operators stack $S1$ and a break nodes stack $S2$. The latter is defined in the following:

1. If one of the children is an operator A and the other is a primitive B. The parent for A is set to the current node and then it is pushed onto $S1$.
2. If both children are operators. The right child is set as a *break* node and is pushed on to $S2$. The route at break nodes is flipped to the left branch of the tree, see figure 4.2.
3. If the two children are both primitives: First the root of the *BlobTree* is set to the current node if it has not been set before. (Refer to the *BlobTree* header format in figure 4.1 for the location of the root field shown as RT.) A rope is created between the two primitives, linking the two nodes and a break node B is popped from $S2$. The next node for B is set to the current node.
4. This process is continued until $S1$ is emptied.

Up-sweep traversal on the GPU

Figure 4.2, shows the final route for a sample *BlobTree* . The benefits of this type of routing encoded into the tree is that there is no need for storing a deep stack for the intermediate operators. Using this new approach the tree is now only evaluated from bottom to top.

The *BlobTree* root node RT marks the starting point of the traversal. This field is stored in the header section of the data structure as illustrated in figure 4.1. Using the new traversal route this field is set to the operator node 10 for the example *BlobTree* shown in figure 4.2.

In case of an operator where all its children are primitives, the field due to each child is computed before the operator evaluation. The computed field is stored in the global field variable $F1$ and using the link to the next node provided in the structure (NX) the evaluation continues to the next operator in the tree at an upper level (up-sweep). When evaluating an operator with one primitive child then the field due to the primitive child is computed before applying the current operator to $F1$ and performing the up-sweep (nodes 5 to 9 in figure 4.2).

When visiting a break node the field is computed as usual but this time it is stored in the special variable $F2$, instead. In addition at a break node, the NX field in the structure points to a non-parent operator (operator 4 for break node 3 in the example).

The process continues until the first node in the tree is evaluated at which the $F1$ and $F2$ values store the children fields for their parent. The NX field associated with the first node points to itself which marks the end of the traversal process.

4.5 GPU Surface Extraction Algorithm

In this section we present our GPU polygonization algorithm which is based on our novel field value evaluation technique described in the previous section. Since the following steps are implemented using OpenCL on the GPU we will use the term *kernel* which is a single thread of execution on the GPU. Please refer to section 4.3 for a description of the memory model for general purpose GPU (*GPGPU*) programming.

We start by computing the axis-aligned bounding box of the model by traversing the tree from root to leaves and applying the transformation matrices at leaf nodes (primitives). Using the *cellsize* parameter supplied by the user the bounding box is subdivided into a grid of voxels. The kernel *ComputeAllFields* then computes a fieldvalue per each vertex of the voxel grid and stores that value in the format of *XYZF* where XYZ denotes the position and F is the so-called field at that point.

After computing all the fields, the grid edges are processed in the following order. Each vertex in the grid is connected to at most 6 other directly adjacent vertices. Starting from the lower corner of the voxel grid the kernel *ProcessEdges* visits the corresponding vertex in the grid and examines only the edges that start from that vertex and extend to the adjacent vertices in the next index step. This way all edges in the voxel grid are checked and redundant traversals can be avoided. Needless to say that at boundary vertices the kernel may process less than 3 edges per each vertex (i.e. the ones that are within the convex-hull of the model). Upon each kernel run at this stage the index address to the corresponding vertex and its 3 other adjacent vertices is computed as shown in the algorithm 3. Per each vertex an inclusion query is performed (i.e. The fieldvalue at that vertex is compared against the iso-value. If it is greater than or equal the isovalue the vertex is considered inside otherwise outside the model). Two values are stored before completion of this kernel call. The first is the count of crossed edges at that vertex and the second one is a 3 bits flag which basically locates the intersected edges along x, y or z axes.

Algorithm 3 *ProcessEdges* kernel counts the number of intersected edges and their corresponding axes. This kernel runs per each vertex in the voxel grid.

```
1: $v = queryVertexInclusion(i, j, k)$
2: $vx = queryVertexInclusion(i + 1, j, k)$
3: $vy = queryVertexInclusion(i, j + 1, k)$
4: $vz = queryVertexInclusion(i, j, k + 1)$
5: $count = (v \oplus vx) + (v \oplus vy) + (v \oplus vz)$
6: $flag = (((v \oplus vx) << 2)or((v \oplus vy) << 1)or(v \oplus vz))$
```

In the above algorithm the *queryVertexInclusion* function checks whether the vertex at a specified voxel grid index is inside the model i.e. the field value at that vertex is greater than or equal to the *isovalue*. Then it performs the same test for the end-points of the edges emanating from the current vertex along the primary axes. If both endpoints of an edges are inside (or outside) the model then there is no intersection between the iso-surface and the voxel grid. Variable *count* stores the number of intersections associated with the current vertex at the grid address (i, j, k). In addition a 3 bits *flag* variable holds the state of the intersections along the primary edges in the format of XYZ.

After processing all edges, the array *EdgeBuffer* contains the count of intersected edges per each vertex in the voxel grid. In order to compute the total number of output vertices in the triangle mesh the prefix-sum [64] of *EdgeBuffer* is computed and stored in a separate gpu memory buffer called *ScannedEdges*. The total number of vertices is the sum of the last elements in the *EdgeBuffer* and ScannedEdges arrays. Before computing the vertex attributes such as the position, color and normal, the associated memory buffers are allocated on the device using the total number of vertices computed in the previous step.

After this stage the vertex attributes of the mesh can be computed by executing a root finding method on the intersected edges and storing the output vertices in their appropriate buffer locations using the offsets in *ScannedEdges*. We use a Newton-Raphson root finding method which converges to the iso-surface using the gradient of the field [49]. At each iteration the root is displaced closer to the surface according to the method given by Overveld et al. ([79]):

$$r = r + \frac{(iso - f(r))}{\nabla V(r).\nabla V(r)} \tag{4.1}$$

Where r is the root, $f(r)$ is the field at r and $\nabla V(r)$ is the gradient of the field at r. A maximum of four iterations was sufficient to provide smooth results in our tests. After computing the root position other attributes such as the color and normal at that vertex are also computed and stored in their designated buffers. The next step is to process the cells in the voxel grid in parallel and compute a configuration index per each cell for extracting the topology of the triangles. There is no *BlobTree* traversal at this stage since the computed fields will be provided to the kernel function. The configuration table is supplied as a texture and can be accessed using sampler unit for fast access. The number of triangles that are output per each cell is stored in a buffer called *FaceBuffer*.

In order to find the total number of triangle elements, a prefix-sum scan is applied to the *FaceBuffer* array in the same way that total number of vertices is computed previously. The buffer *ScannedFaces* will be used to hold the offset values per each cell.

The final stage in our GPU polygonizer is producing the triangles. For this purpose the kernel function *GenerateFaces* is called per each cell in the voxel grid. No *BlobTree* traversal is required for this stage. Only the cells which intersect with the iso-surface are processed. Upon each kernel run the indices for the eight vertices of the current cell are computed.

To process cell configurations we used the improved marching cubes table by Dietrich et al. [22], which avoids most of the small and badly shaped triangles. The table is supplied to the kernel as a texture of size 256 rows by 16 columns. To access the entries in the table the texture sampler on the hardware can be used. Per each cell the configuration index is computed using the previously stored fields. Each entry in the table is the index of an edge in the cell (There are 12 edges per each cell). Algorithm 4 shows how the triangle elements are computed in this stage.

Algorithm 4 *GenerateFaces* kernel function computes the triangle indices per each cell and outputs them directly into an OpenGL index buffer for rasterization. All the buffers can be read back from the GPU and stored.

```
1: $index = globalCellIndex(i, j, k, dim)$
2: $config = cellconfig(i, j, k)$
3: if $config == 0$ or $config == 255$ then
4:     return;
5: end if
6: $cellcorners = cellCornerIndices(i, j, k, dim)$
7: $offset = ScannedFaces[index]$
8: $count = FaceBuffer[index]$
9: for $i = 1 \rightarrow$count do
10:     $edge = sample(\text{configtable}, int2(edge, index))$
11:     $start = EdgeStartIndex[edge]$
12:     $axis = EdgeAxis[edge]$
13:     $elements[offset + i] = ScannedEdges[cellcorners[start]] + axis$
14: end for
```

Upto five triangles can be extracted from each cell. Lines 11 and 12 in the algorithm assign the start index of an edge and its associated axis using two constant buffers supplied to the kernel for this purpose. The element entry is computed as an index to the global vertex buffer. The *ScannedEdges* buffer holds the global offset for all the intersected edges as previously discussed.

4.6 Analysis and Results

In this section we review the effects of the previous optimizations on the overall performance of the system. In our experiments we tested the effect of the stackless *BlobTree* traversal algorithm using a set of models created with our incremental modelling system. To make a fair comparison the same algorithm implemented on the GPU using the OpenCL framework once with the stack and the other time with the stackless method presented in section 4.4. The results are shown in the following table:

Table 4.2: Stackless *BlobTree* traversal improved the performance of our *BlobTree* field evaluation significantly. Here is the comparison between our novel stackless approach versus the stack-based implementation for various models. Timings are the average of 100 runs.

Model Name	Field Queries	Grid	Stack-based (ms)	Stack-less (ms)	Speedup
Tumor	16240	29*20*28	21	0.8	26x
cake	18975	33*23*25	17	1	17x
3slabs	28750	46*25*25	30	2	15x

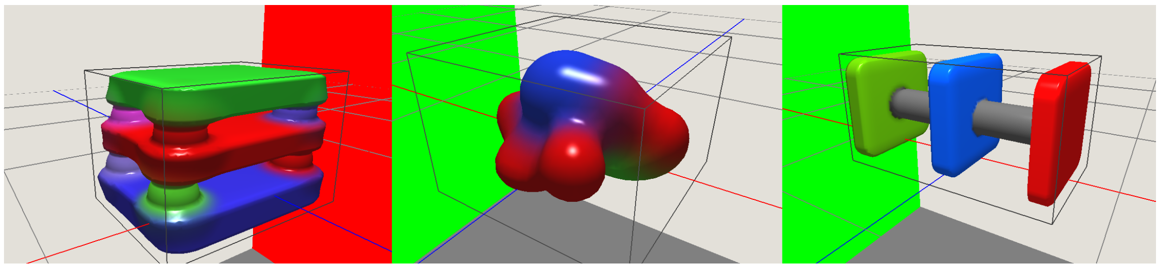


Figure 4.3: Sample models for testing our GPU polygonization method. From left to right: Cake, tumor and 3slabs.

When using stacks, the register spilling phenomenon mentioned previously degrades the performance due to the higher cost of accessing the shared memory. Conditional push and pops in the stack-based method also stalls the performance of the kernels.

Faster field evaluations using the stackless algorithm also improves the performance of our root-finding method and the overall polygonization time. In the following we review per kernel time break-time which provides a close look at hotspots (most time-consuming locations) in our implementation.

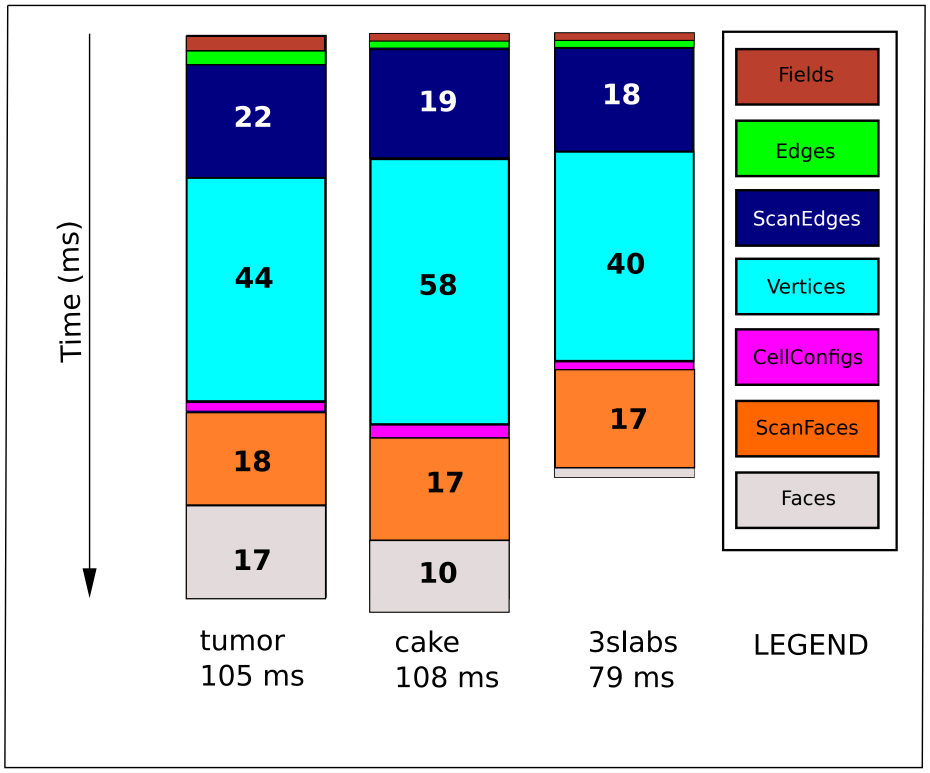


Figure 4.4: Polygonization time breakdown in milliseconds for the three models shown in the previous section. Vertex processing is the most compute-intensive stage due to the Newthon-Raphson root finding method employed and the evaluation of colors and normals which require additional traversals.

As it can be seen computing vertices is still the most time-consuming stage due to the extra tree traversals required for high quality gradiant-based Newthon-Raphson root-finding method. Computing other attributes such as colors and normals would also need 1 and 4 traversals respectively. The prefix-sum scan operator employed here uses multi-pass computing. The extra cost of kernel calls has increased the cost of using these operators. Several optimizations can be performed to benefit the overall performance. By vectorizing all kernels the core SIMD units can be used more efficiently. The prefix-sum scan operator can also be implemented in such a way that the memory-bank conflicts are completely avoided and the data-accesses are performed in parallel.

Chapter 5

Deformable Model

Interactive visualization techniques developed in the previous chapters support a *Blob-Tree* based modelling system for the incremental construction process. In a surgical simulation scenario, tissues undergo various levels of deformations and topological modifications. Researchers in the field studied different approaches to recreate the elastic behavior of tissues. The two main challenges that are reported continuously in this field are interactivity and fidelity of motion [50].

Based on the foundation built so far, we create a deformable model to support the required interactions with surgical tools i.e., elastic deformations by pushing the tissue. The novelty of our method is in the real-time collision and contact response computation using the implicit model, which is discussed in section 5.3.

The next chapter will address the problem associated with cutting, which is a primary operation in many surgical simulation scenarios. The goal is to make a comprehensive system that bridges the gap between modelling and simulation of tissues.

5.1 Overview

A software pipeline is developed to support physically based simulation of deformable tissues using our implicit modelling framework, figure 5.1.

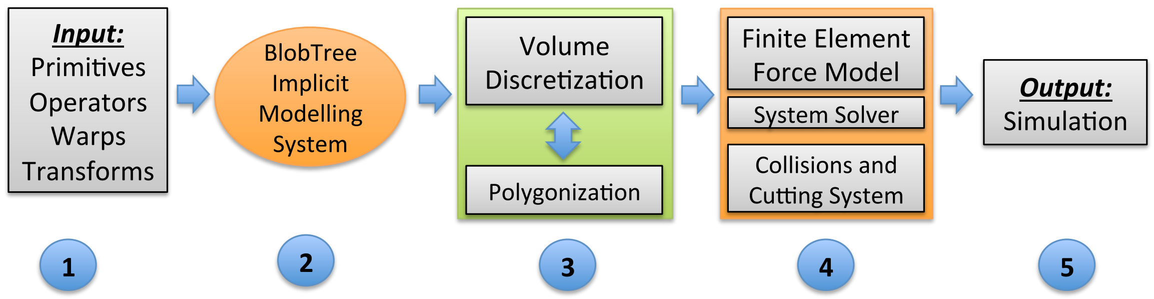


Figure 5.1: High-level system software pipeline to support deformations and cutting.

Stages 1 and 2 in the pipeline are associated with the modelling process which is already discussed in chapter 2. Stage 3 is associated with the polygonization algorithms that are developed in chapters 3 and 4. Also in stage 3 is the volume discretization technique that converts the *BlobTree* model into a tetrahedral mesh for physical simulation, this process is discussed further in section 5.2. The material that is discussed in the current and the next chapters are associated with the stage 4 of the pipeline shown above. The output of the system is the simulation environment that facilitates deformations and cutting which is the final output.

5.2 Physical Model

The deformable model has to be considered as a continuous volume in order to reproduce its elastic, volumetric effects. In our system, the *BlobTree* model is discretized into a tetrahedral mesh using the algorithm proposed by Labelle et al. [42]. This step is performed once the modelling stage is completed and the tissue is ready for the simulation. This is a one time process and is not a bottleneck in our system.

A fine-grained cell-size parameter captures the exterior part of the model while a coarser sampling is used to produce the internal tetrahedral cells. This approach allows us to maintain a balanced number of tetrahedral cells for the physics simulation. The output of this stage is called *volume mesh* in order to differentiate it from the *surface mesh* extracted using the polygonization algorithm in the previous chapters. An edge-based data structure representing the tetrahedral meshes in our system is described in section 6.2.

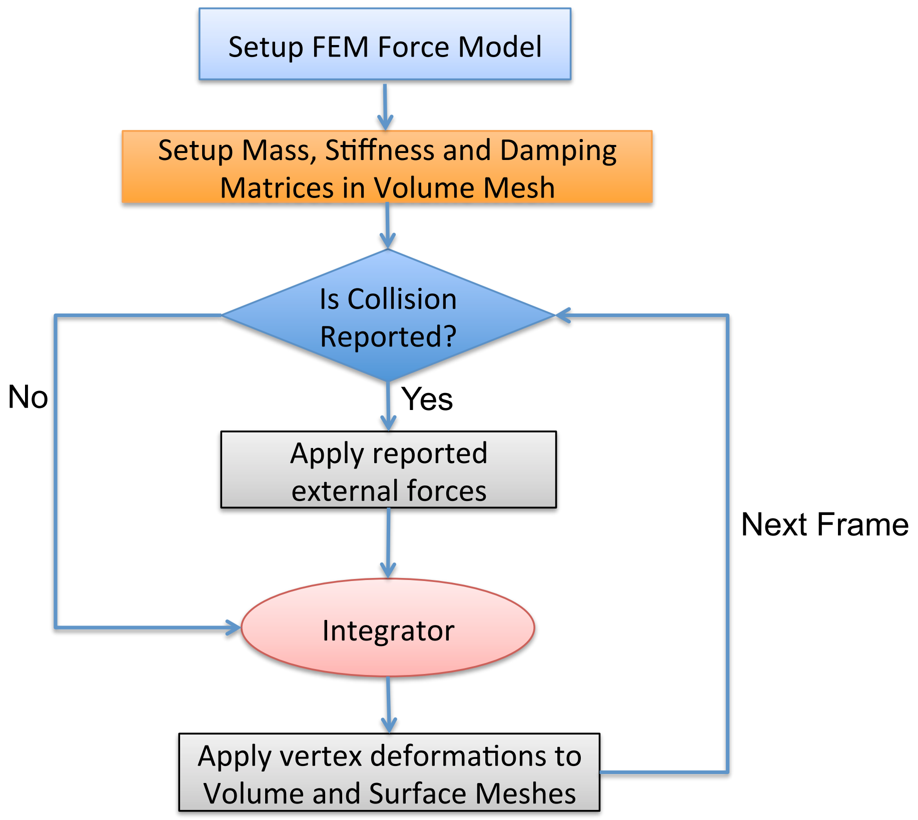

Setup FEM Force Model
Setup Mass, Stiffness and Damping Matrices in Volume Mesh
Is Collision Reported?
No
Yes
Apply reported external forces
Next Frame
Integrator
Apply vertex deformations to Volume and Surface Meshes

Figure 5.2: System solver component for the stage 5 of our pipeline. External forces are computed using our collision detection technique which is discussed in the next section.

As shown in figure 5.2, the volume mesh is augmented with a finite element force model. For a detailed discussion on finite element concepts, refer to [5]. In our system we chose the co-rotational linear elastic force model. One of the drawbacks of pure linear force models is that under large deformations, they can lead to inflated volume artifacts, which is not desirable in a deformable model for surgical simulations. The co-rotational model assumes that deformation at each element is composed of a rotation and a small displacement. This separation is achieved by using polar decomposition of the deformation gradient, and it removes the linearization artifacts mentioned earlier [5]. A validation step such as the one performed in the work of Shahingohar et al. helps to compare the stability and accuracy of the deformable model [65].

The elastic properties of tissues are applied during the force model setup. From figure 5.2, in case of collision with medical devices, the external forces due to the contact response are computed and integrated in our system. This step is further explained in the next section. Finally all computed displacements are applied to the meshes.

5.3 Collision and Contact with tools

The interactions of medical devices and the tissue model has to be accounted for to support surgical training scenarios. The way these contacts are handled impacts the overall behavior of the system with respect to the deformable tissues. Figure 5.3, shows the collision and contact handling module in our system.

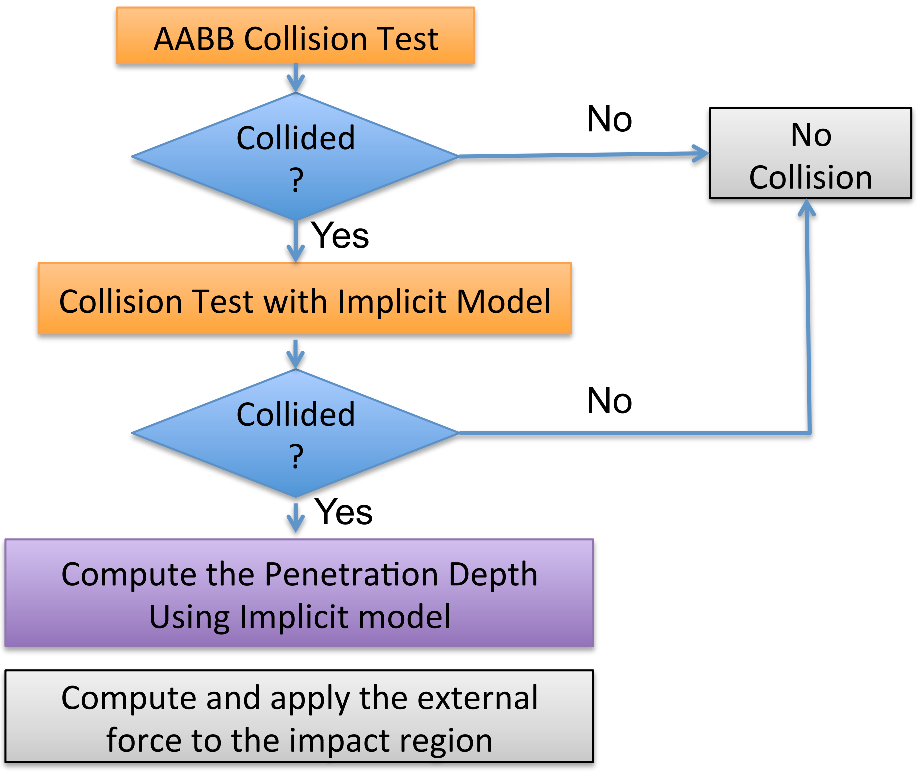

AABB Collision Test
Collided ?
No
No Collision
Yes
Collision Test with Implicit Model
Collided ?
No
Yes
Compute the Penetration Depth Using Implicit model
Compute and apply the external force to the impact region

Figure 5.3: To compute the external forces applied by the medical tools, we exploit the trivial collision detection facilities of the underlying implicit model.

Medical tools are modelled as simple polygonal shapes in our system. The user moves the tool freely and to compute the collision of the tool and the model several steps are performed in order. First, the axis aligned bounding box (AABB) of the tool is tested against the AABB of the model and if an intersection is detected the process continues further. The second collision test is performed by computing the field at vertices of the tool model. In case of our probing tool which is modelled as a cube, the field at vertices of the cube are computed. If a field at a vertex is above the *iso-value*, then that vertex is inside the model and an intersection is detected.

To compute the distance, the inverse of the field function from equation 2.4 is used:

$$dist_{\text{wyvill}}(f) = \begin{cases} 1 & \text{if } f = 0 \\ \sqrt{(1 - f^{\frac{1}{3}})} - k & \text{if } 0 < f \leq 1 \end{cases} \tag{5.1}$$

In equation 5.1, f is the input field and k is called the *iso-distance* value which is the perpendicular distance from the iso-surface to the nearest skeleton of the model. k can be computed easily by using the the $iso-value$ which is set to 0.5 in our system as the output of equation 2.4, solving that equation for x yields to the constant value of 0.4541 for k.

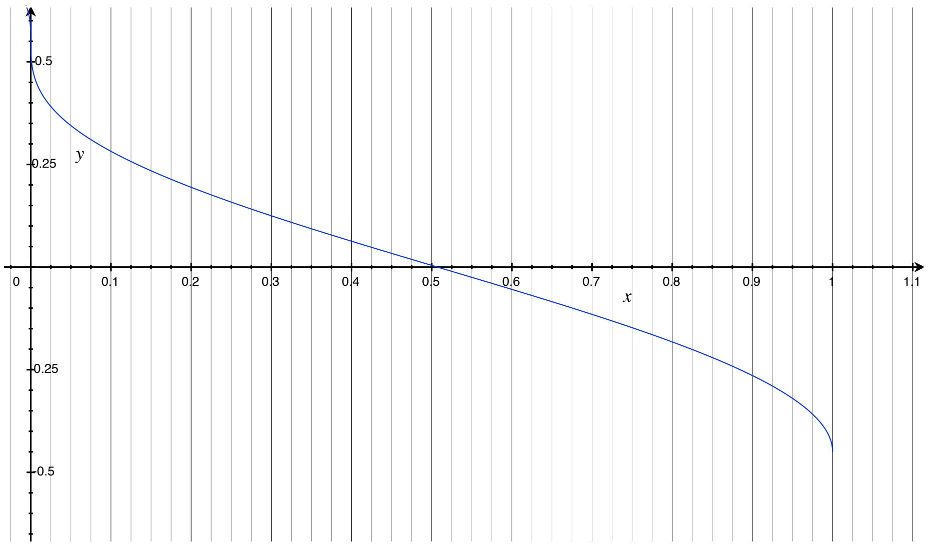


Figure 5.4: Plot of equation 5.1. Horizontal axis is the field value which is changing from 0 to 1 in this graph. The vertical axis is the distance to the *iso-surface*.

Figure 5.4, is the plot of equation 5.1 in the range $[0, 1]$ for the field value. When the field is equal to the *iso-value*, then the associated vertex is on the surface, hence the zero distance. A positive value for distance is due the fact that the collision has not happened yet but the vertex is in close proximity of the surface, whereas negative values indicate a penetration into the model.

In case of a collision, the penetration depths are computed for the all colliding vertices of the tool. The contact force is computed based on the penetration depth, the direction of the tool and the gradient of the field as shown in the following equation:

$$CF(v) = (\frac{\nabla F(v) + T}{\|\nabla F(v) + T\|}) * dist_{\text{wyvill}}(g_{\text{wyvill}}(v)) * s \qquad (5.2)$$

The output of equation 5.2 is a force vector at point v, T is a directional vector that represents the tool trajectory. The direction of the final external force is adjusted using the gradient of the field i.e., instead of using the tool trajectory vector (T), directly; we use the average of T and the field gradient at v. This choice is made to compensate for the tangential tool trajectories that can cause unintended shearing at the boundary edges of the model.

$g_{\text{wyvill}}(v)$ and $\nabla F(v)$ are the field and gradient at vertex v, respectively. s is the impact factor to adjust the external force magnitude.

Using this technique, the contact forces are computed upon collisions of the tool and the model. To apply this external force to the physical model we use a force propagating technique based on the volume mesh data structure described in section 6.2. This step is further explained in the following.

The edge-based data structure for the tetrahedral mesh provides access to all the adjacent vertices to a given vertex. This functionality can provide the access to the K-ring neighborhood of a contact point to perform force propagation. Figure 5.5, shows an example of this function for a contact point p. The external force computed for P is propagated to the first and second rings with decreasing magnitudes at each step. This technique provides a more realistic distribution of the collision force for the simulation of tool-model interactions.

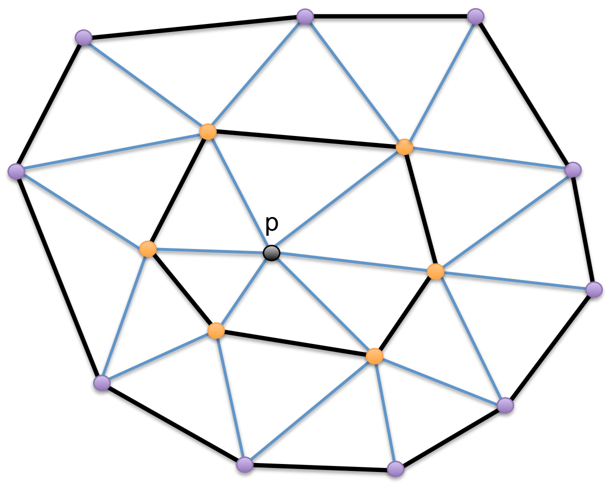


Figure 5.5: The K-ring neighborhood of a vertex p can be accessed using our tetrahedral mesh data-structure. Here, the first and second rings of vertex p are shown in orange and pink, respectively.

5.4 Results

In this section we present our results in simulating the behavior of several deformable models created using our implicit modelling framework. Table 5.1, provides details of the models. The number of tetrahedral cells reported in this table are extracted using a small cellsize of 0.1.

Table 5.1: The following deformable models are used in our experiments.

name	primitives	operators	finite element cells
tumor	10	1	130K
3slabs	4	1	120K
cake	11	1	85K

Figure 5.6 shows the effect of cellsize parameter in the number of finite element cells generated for the physical mesh. As shown in this graph, increasing the step size for the volume discretization process, results in larger finite element cells and decreases the total count of elements.

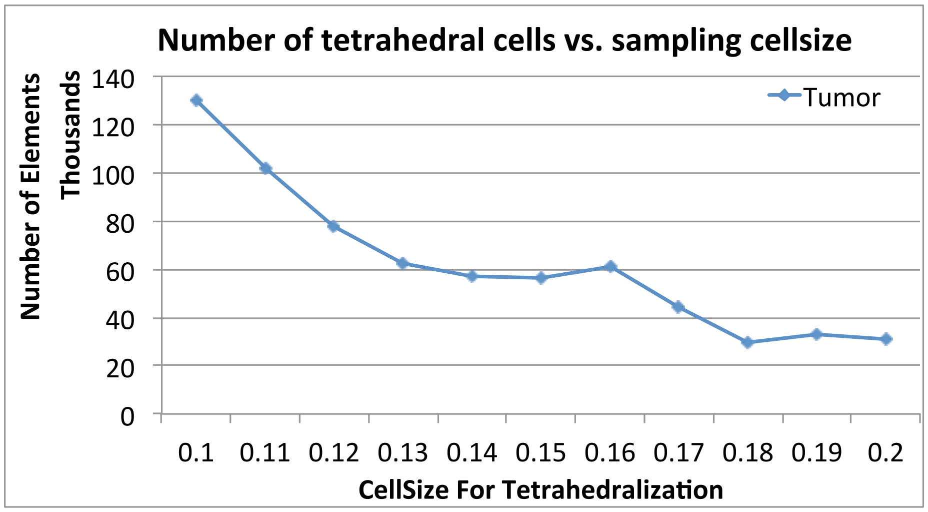


Figure 5.6: The effect of cellsize parameter in the total number of generated tetrahedral cells for the physics mesh. Vertical axis is reported in thousands of elements.

Figures 5.7 and 5.8 show the tumor and the 3slabs models when compressed with the probe tool, respectively. The surface mesh which is extracted with our polygonization method is deformed using the computed displacements for the volume mesh. The 3slabs model in figure 5.8 is fixed to the green wall and the external forces applied by the probe tool barely reach the green slab.

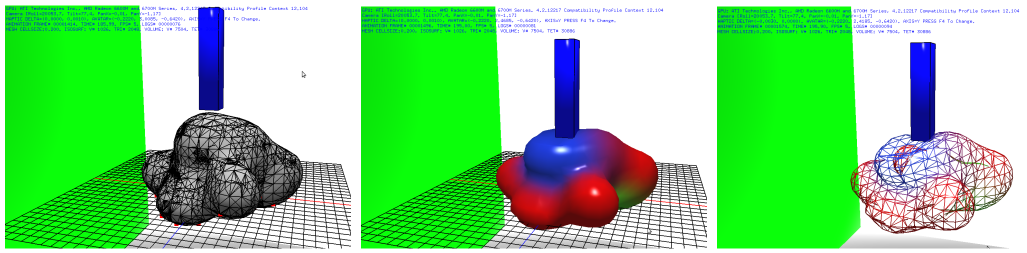


Figure 5.7: Tumor model is pushed from the top using our probe tool. Left: The volume mesh used as physics model shown in gray. Middle: Model pushed down for compression. Right: The surface mesh deformations are in sync with that of the volume mesh.

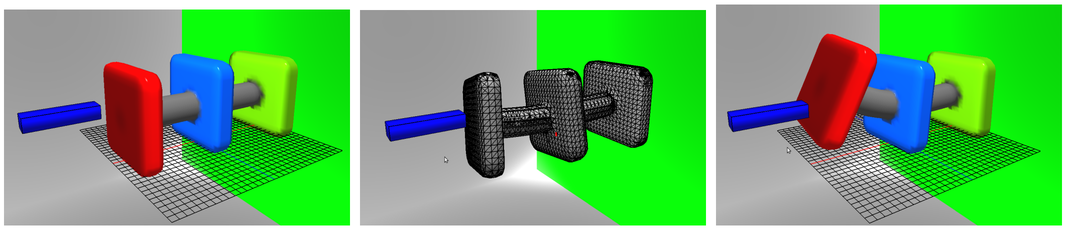
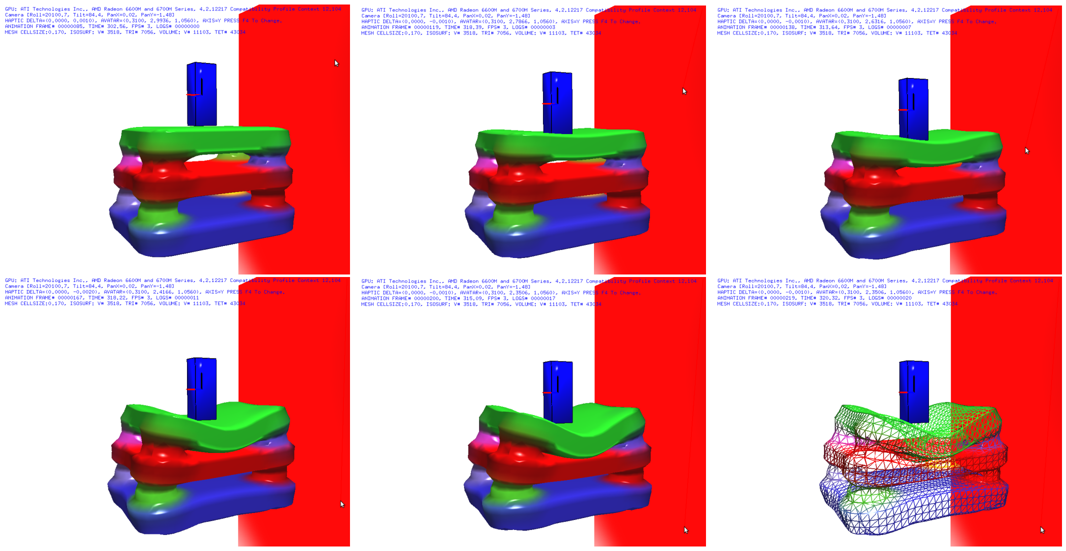


Figure 5.8: The force is applied to the 3slabs model horizontally. The green slab is fixed to the wall in this experiment. Deformations are wider on the red slab. Middle: the volume mesh shown in gray.

In the third experience, the cake model which is placed on the ground is compressed from the top using the probe tool. The 6 images shown in figure 5.9 illustrate the progression of deformations sequentially.



```
GPU: ATI Technologies Inc., AMD Radeon 6600M and 6700M Series, 4.2.12217 Compatibility Profile Context 12.104
Camera [Roll=20100.7, Tilt=84.4, PanX=0.02, PanY=-1.48]
HAPTIC DELTA=(0.0000, 0.0010), AVATAR=(0.3100, 2.9936, 1.0560), AXIS=Y PRESS F4 To Change.
ANIMATION FRAME# 00000085, TIME# 302.56, FPS# 3, LOGS# 00000000
MESH CELLSIZE:0.170, ISOSURF: V# 3518, TRI# 7056, VOLUME: V# 11103, TET# 43034
```

```
GPU: ATI Technologies Inc., AMD Radeon 6600M and 6700M Series, 4.2.12217 Compatibility Profile Context 12.104
Camera [Roll=20100.7, Tilt=84.4, PanX=0.02, PanY=-1.48]
HAPTIC DELTA=(0.0000, -0.0010), AVATAR=(0.3100, 2.7866, 1.0560), AXIS=Y PRESS F4 To Change.
ANIMATION FRAME# 00000119, TIME# 318.39, FPS# 3, LOGS# 00000003
MESH CELLSIZE:0.170, ISOSURF: V# 3518, TRI# 7056, VOLUME: V# 11103, TET# 43034
```

```
GPU: ATI Technologies Inc., AMD Radeon 6600M and 6700M Series, 4.2.12217 Compatibility Profile Context 12.104
Camera [Roll=20100.7, Tilt=84.4, PanX=0.02, PanY=-1.48]
HAPTIC DELTA=(0.0000, -0.0010), AVATAR=(0.3100, 2.6316, 1.0560), AXIS=Y PRESS F4 To Change.
ANIMATION FRAME# 00000138, TIME# 313.64, FPS# 3, LOGS# 00000007
MESH CELLSIZE:0.170, ISOSURF: V# 3518, TRI# 7056, VOLUME: V# 11103, TET# 43034
```

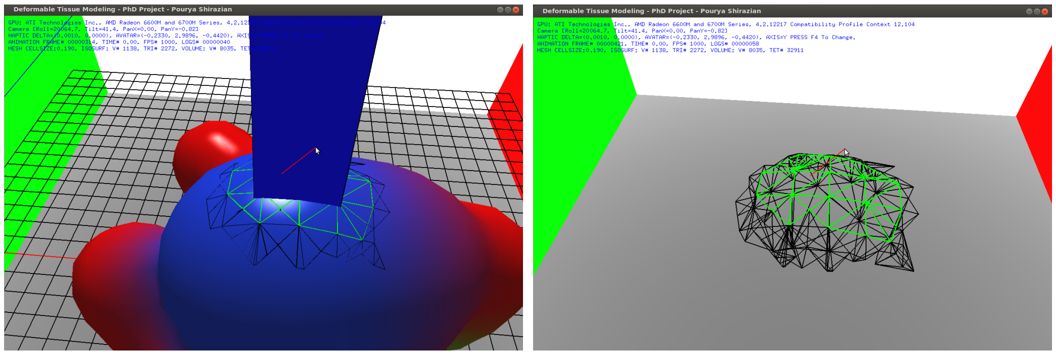
```
GPU: ATI Technologies Inc., AMD Radeon 6600M and 6700M Series, 4.2.12217 Compatibility Profile Context 12.104
Camera [Roll=20100.7, Tilt=84.4, PanX=0.02, PanY=-1.48]
HAPTIC DELTA=(0.0000, -0.0020), AVATAR=(0.3100, 2.4166, 1.0560), AXIS=Y PRESS F4 To Change.
ANIMATION FRAME# 00000167, TIME# 318.22, FPS# 3, LOGS# 00000011
MESH CELLSIZE:0.170, ISOSURF: V# 3518, TRI# 7056, VOLUME: V# 11103, TET# 43034
```

```
GPU: ATI Technologies Inc., AMD Radeon 6600M and 6700M Series, 4.2.12217 Compatibility Profile Context 12.104
Camera [Roll=20100.7, Tilt=84.4, PanX=0.02, PanY=-1.48]
HAPTIC DELTA=(0.0000, -0.0010), AVATAR=(0.3100, 2.3506, 1.0560), AXIS=Y PRESS F4 To Change.
ANIMATION FRAME# 00000200, TIME# 315.09, FPS# 3, LOGS# 00000017
MESH CELLSIZE:0.170, ISOSURF: V# 3518, TRI# 7056, VOLUME: V# 11103, TET# 43034
```

```
GPU: ATI Technologies Inc., AMD Radeon 6600M and 6700M Series, 4.2.12217 Compatibility Profile Context 12.104
Camera [Roll=20100.7, Tilt=84.4, PanX=0.02, PanY=-1.48]
HAPTIC DELTA=(0.0000, -0.0010), AVATAR=(0.3100, 2.3506, 1.0560), AXIS=Y PRESS F4 To Change.
ANIMATION FRAME# 00000219, TIME# 320.32, FPS# 3, LOGS# 00000020
MESH CELLSIZE:0.170, ISOSURF: V# 3518, TRI# 7056, VOLUME: V# 11103, TET# 43034
```

Figure 5.9: The cake model is a 3 level structure which is compressed from above in this experiment. The sequence of images show the increasing stages of deformation from the beginning to the end. Last image on the bottom row is the surface mesh which is always in sync with the physics model.

Figure 5.10 shows the contact surface of the probe tool and the volume mesh. The external forces are propagated to the volume mesh via the green vertices in the figure. All collisions are detected correctly using the technique described in section 5.3.


Deformable Tissue Modeling - PhD Project - Pourya Shirazian
Deformable Tissue Modeling - PhD Project - Pourya Shirazian

```
GPU: ATI Technologies Inc., AMD Radeon 6600M and 6700M Series, 4.2.12217 Compatibility Profile Context 12.104
Camera [Roll=20064.7, Tilt=41.4, PanX=0.00, PanY=-0.82]
HAPTIC DELTA=(0.0010, 0.0000), AVATAR=(-0.2330, 2.9896, -0.4420), AXIS=Y PRESS F4 To Change.
ANIMATION FRAME# 00000314, TIME# 0.00, FPS# 1000, LOGS# 00000040
MESH CELLSIZE:0.190, ISOSURF: V# 1138, TRI# 2272, VOLUME: V# 8035, TET# 32911
```

```
GPU: ATI Technologies Inc., AMD Radeon 6600M and 6700M Series, 4.2.12217 Compatibility Profile Context 12.104
Camera [Roll=20064.7, Tilt=41.4, PanX=0.00, PanY=-0.82]
HAPTIC DELTA=(0.0010, 0.0000), AVATAR=(-0.2330, 2.9896, -0.4420), AXIS=Y PRESS F4 To Change.
ANIMATION FRAME# 00000421, TIME# 0.00, FPS# 1000, LOGS# 00000058
MESH CELLSIZE:0.190, ISOSURF: V# 1138, TRI# 2272, VOLUME: V# 8035, TET# 32911
```

Figure 5.10: The contact surface of our probe tool and the tumor model shown in green triangles. The computed contact force is applied to all vertices in the green area.

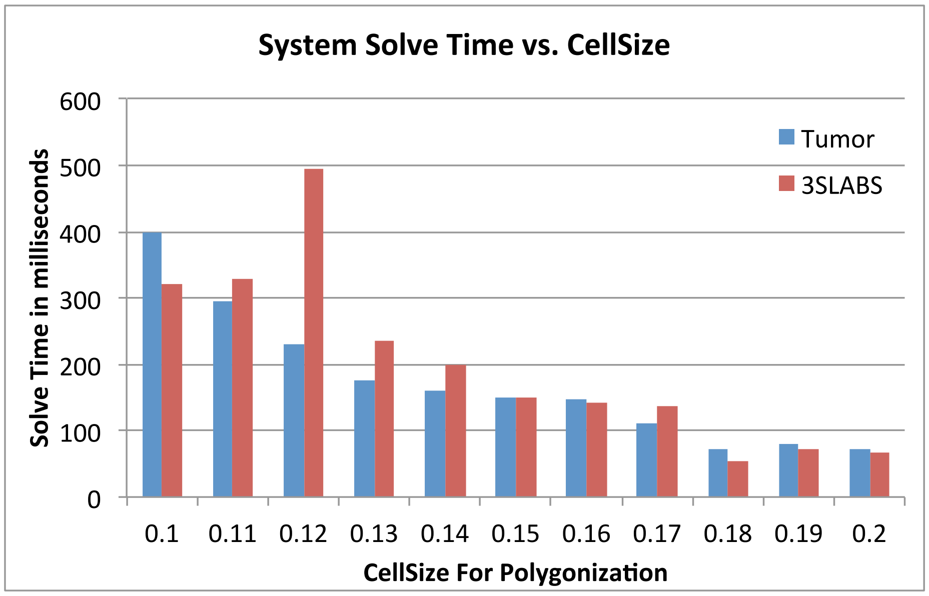


Figure 5.11: System solve time reported in milliseconds vs. the cellsize parameter used for the volume discretization.

Figure 5.11, illustrates the system solve time as the cellsize parameter increases uniformly. Larger cellsize results in smaller amount of finite element cells which takes less time to solve for displacements. One way to keep the number of internal tetrahedral cells low is by using the adaptive volume discretization technique that uses a coarser grid to extract the elements. As overshoot in figure 5.11 at cellsize 0.12 is due to a sudden increase in the number of finite element cells despite using a coarser grid size. At this point the algorithm was unable to use the coarser grid for the internal sampling and a higher percentage of elements are created using the finer grid.

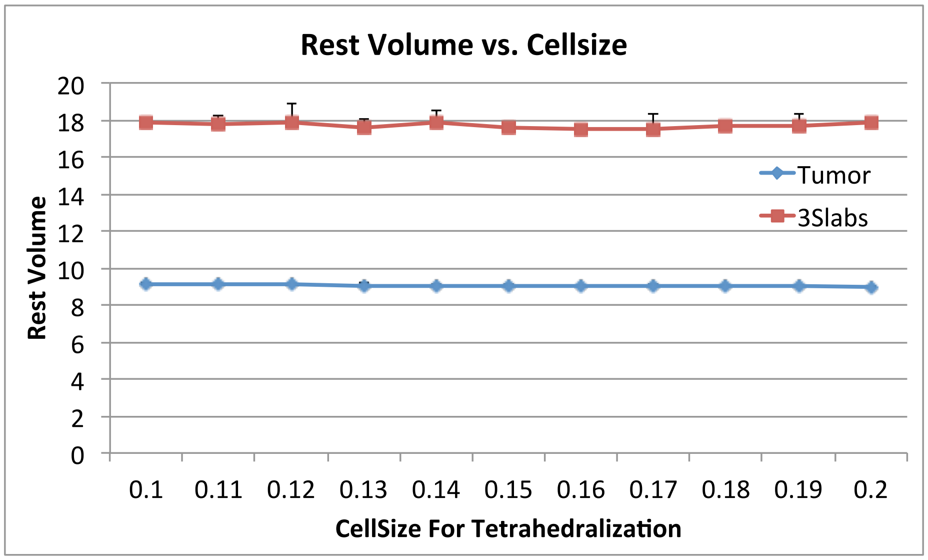


Figure 5.12: Total volume versus the cellsize parameter for discretization. Maximum volume change in all of our experiments was less than 1 percent.

One of the required features of a model for surgical simulation scenarios is volume preservation. The total volume of the model should not change when it goes under deformations.

In an experiment, the volume is computed and logged while the model is deformed using the probe tool (see figure 5.12). Maximum volume change is always less then 1 percent in this test for both models and the same results holds under various cellsizes. Both models used a constant Poisson ratio of 0.45 in all tests.

Chapter 6

Real-time Cutting

One of the main objectives of a virtual reality based surgical simulation system is the removal of pathologic tissue [73, 54]. Cutting imposes many challenges to the development of a robust, interactive surgical simulation, not only because of the nonlinear material behavior exhibited by soft tissue, but also due to the complexity of introducing the cutting-induced discontinuity.

We developed our new volumetric mesh cutting system named "VolCut" in the context of a human skull craniotomy simulation. When an abnormality of the brain is suspected, a brain needle biopsy is performed and guided precisely by a computer system to avoid serious complications. A small hole is drilled into the skull, and a needle is inserted into the brain tissue guided by computer-assisted imaging techniques (CT or MRI scans). The actual biopsy process can not be seen by the surgeon. For this reason, non-progressive cutting, where a tetrahedral element is decomposed after the sweep surface traverses the tetrahedral elements, is a reasonable approximation for that application. Also, in the current stage of our system, we do not model any haptic interaction of the cutting tool with the tissue during a cut.

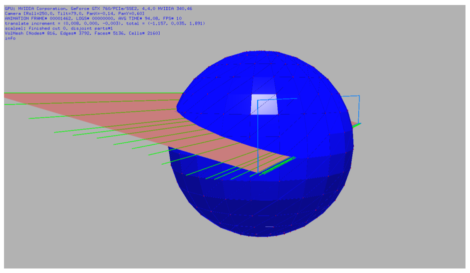
6.1 Overview

The physics simulation in VolCut uses tetrahedral meshes to compute deformations. In this section, we present our GPU-assisted approach to cutting tetrahedral meshes in real-time. The input to VolCut is a cut trajectory and an edge-based data structure representing the tetrahedral mesh.

The tetrahedral mesh itself is extracted from the *BlobTree* model that is created using the incremental modelling system described in section 2.1. The complete process is summarized below:

- The user models the elastic tissue using the *BlobTree* approach (section 2.1)
- The proposed GPU-accelerated polygonization algorithm provides real-time updates for the modifications that are made to the model throughout the design process (section 4.5)
- The final model is converted to a tetrahedral mesh with the algorithm presented by Si et al. [68]. This is a one-time process and is not a bottleneck in the system. The triangle mesh extracted in the previous stage is used as an input to the tetrahedralization algorithm. The output is a volumetric tetrahedral mesh.
- Our physics simulation handles the dynamic behavior of the model and handles the collisions with other objects in the environment including the scalpel.

As shown in figure 6.1, the user moves the cutting tool and the system records the path of the blade endpoints shown in purple. The first intersection between the recorded trajectory and the model marks the beginning of the cutting process. When the tool completely traverses the model the system computes the cutting configurations as described below.


(a) The trajectory of the cutting tool

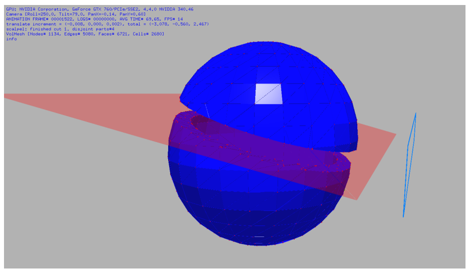

(b) The shell model after cut

Figure 6.1: The cut trajectory in blue and the sweep-surface shown in pink. The scalpel passes through the shell model for cutting.

The following steps are performed to complete the cut induced by the scalpel on the mesh:

1. Using the GPU-accelerated algorithm described in section 6.3.1, the intersection of the sweep-surface and all the edges of the mesh are computed. The output of this stage is a list of cut-edges and their associated intersection points.

2. A GPU kernel function is used to compute the distance of the nodes in the cut-edges and the end points of the cut tool (section 6.3.2). This way the nodes that are too close to the sweep-surface are identified and a different configuration is used to produce subdivided elements in the next stage to avoid ill-shaped elements. The output of this stage is a list that associates edges with their cut-nodes.

3. Using a look-up table all the cut tetrahedra are decomposed into sub-elements (section 6.3.4).

4. The nodes identified to be close enough to the cut trajectory are snapped to the sweep surface.

5. The solver system is synchronized with the latest mesh changes, all the mass, damping and stiffness matrices are updated.

6.2 Tetrahedral Mesh Data structure

Figure 6.2 shows the structure of a tetrahedral element and the order we chose to name the nodes, edges and faces in its canonical orientation. In this figure P_0 to P_3 are the nodes (i.e. degrees of freedom in the context of system deformation computation), e_0 to e_5 the edges and F_0 to F_3 are the faces of the element.

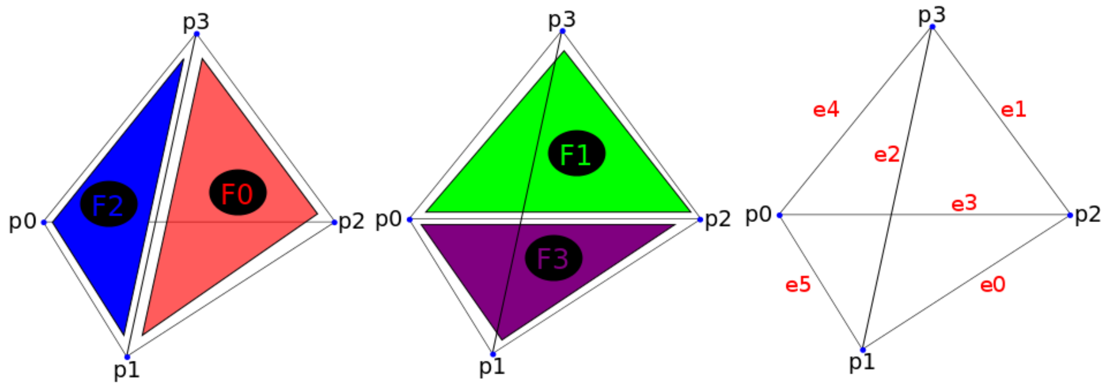


Figure 6.2: A tetrahedral element in its canonical view. Iterating over nodes, edges and faces of each element is one of the primary operations in a geometric algorithm that manipulates such elements. The order we chose here is not the only possible one but it simplifies the cutting algorithm and element subdivision process as we will see later.

In a complex mesh of tetrahedral elements accessing each of these components is a necessary requirement for implementing any topological modifications. Therefore the main module in our cutting algorithm is an edge-based data structure that maps tetrahedral elements to their associated faces and the faces to their associated edges and finally the edges to their nodes. The minimal set of operations frequently used by most algorithms are as following:

- Access to individual vertices, edges, faces and tetrahedral elements. This includes the enumeration of all elements in unspecified order.
- Per each vertex, access to all the directly adjacent nodes to that vertex (i.e. one ring neighborhood see figure 6.3). A typical use-case for this operation is the uniform distribution of the external forces applied to the mesh. Also many mesh simplification algorithms are based on such operators.
- Top-down and bottom-up hierarchical access to the mesh entities (An example is shown in figure 6.3). Top-down access is inherently provided in the structure of the mesh entities e.g. elements are comprised of faces and faces are made up of set of edges etc. In case of the bottom up access some algorithms will benefit to have the incident edges of a certain node, or in case of edges all the incident faces of a given edge and for a given face all the incident elements to a particular face in the mesh. This type of access patterns are particularly useful for topological modification scenarios and the required book-keeping operations.

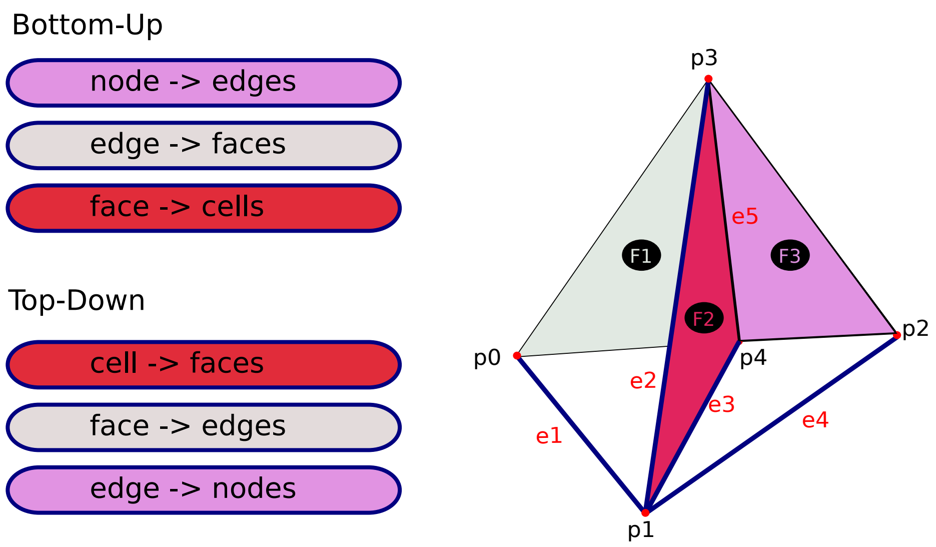


Figure 6.3: Top-down and bottom-up mesh links. The top-down relationships is explicitly defined in the structure of the mesh. An example of the bottom-up links is shown here: Edges $e_1..e_4$ are incident to node p_1 and therefore the set $\{p_0, p_2, p_3, p_4\}$ is the one-ring neighborhood of node p_1. Faces $F_1..F_3$ are incident to edge e_5 and both tetrahedral cells are incident to face F_2.

- Insertion and removal operations for nodes, edges, faces and elements. These operations require extra care in order to keep the top-down and bottom-up links up-to-date. As we will see in our cutting algorithm deferred removal operations can make process much simpler by delaying the actual removal until after all the identified entities are visited and the new entities are inserted to the structure. This is due to the fact that removal operations will update the internal links between mesh entities therefore two consecutive access operations might find the mesh at different states which is not intended in the original proposed algorithm.

- Update operations for nodes, edges, faces and elements. What is important here is to keep the top-down and bottom-up links up-to-date. For-instance in case of an edge update as soon as the two endpoints of the edge is modified all the incident faces (higher level entity) of that edge and also all the incident edges of the end points of the edge (lower level entity) should be updated.

The rest position of each node is stored which can be used to update the physics model and to interpolate the position of newly added nodes in case of cutting. Accessing edges using the bottom-up structure is expensive: First the list of incident edges to one of the endpoints of the edge is accessed and then a serial search on that list is done to find an edge with the matching end points. This has logarithmic complexity with respect to the number of edges in the structure. Instead we chose to use a hash-table to access edges using a key derived from the two end-points of the edge. Below is how this key is computed for a given edge [13, 11]:

$$key64 = (nodes\,[1] << 32) \;\;\&\;\; nodes\,[0] \tag{6.1}$$

This computation is performed when a new edge in inserted into the structure and can produce keys for 2^{64} edges uniquely. The hash-map then associates this key to its corresponding edge index thereby providing constant-time access. Using this technique the faces can also be accessed though their associated node indices which can be convenient for some applications.

Nothing is removed directly from the mesh storage buffers but rather upon cutting, the elements are added to the $freelist$ and later the garbage collection removes all the $freelist$ items from the mesh storage buffers.

When cutting an edge, an edge-update process is performed followed by a new edge insertion. The update process splits the original edge in two. Algorithm 5 describes this process (also see figure 6.4):

Algorithm 5 Splitting an edge in our volumetric mesh data structure. The input to this algorithm is the index of the edge to be splitted and the distance t along the edge where the intersection happens. Figure 6.4 shows this operation in detail.

```
1: $edge \leftarrow fetchEdge(index)$
2: $n0 \leftarrow fetchNode(edge.from)$
3: $n1 \leftarrow fetchNode(edge.to)$
4: $newp0.rest = n0.rest + (n1.rest - n0.rest) * t$
5: $newp0.pos = n0.pos + (n1.pos - n0.pos) * t$
6: $idxNewP0 \leftarrow addNewPoint(newp0)$
7: $newp1 \leftarrow newp0$
8: $idxNewP1 \leftarrow addNewPoint(newp1)$
9: $setEdge(index, edge.from, idxNewP0)$
10: $insertEdge(idxNewP1, edge.to)$
```

Algorithm 5 starts by computing the co-located intersection points $newp0$ and $newp1$ using the provided distance t and the end-points of the original edge. Then both the current and rest positions of the new points are computed. The new points are appended to the appropriate mesh storage lists and the current edge is updated to end at $newp0$. Another edge from $newp1$ to the original end point is added later. Figure 6.4 shows this process.

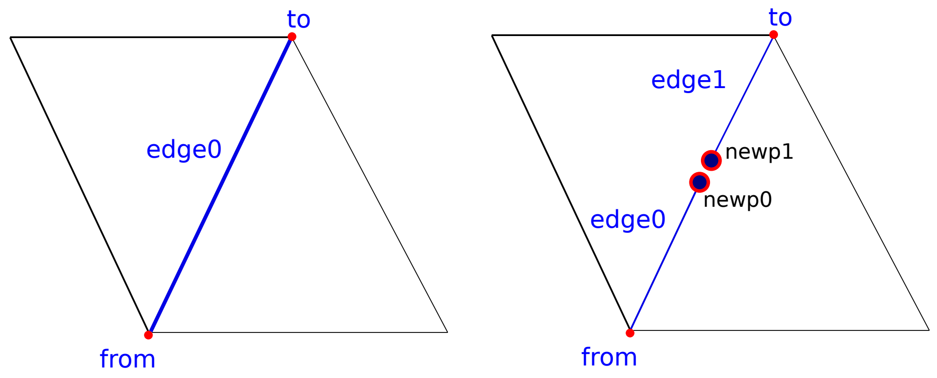


Figure 6.4: Left: The edge to be split. Right: Splitting an edge produces a new edge from the point of intersection to the original endpoint. New points $newp_0$ and $newp_1$ are initially co-located.

Upon topological modifications, events are generated to notify cutting algorithms of the internal changes in the mesh structure. This is also useful for debugging and evaluation purposes and can be logged for accounting the sequence of changes made to the original mesh. The events include update, insertion and removal of nodes, edges, faces and elements.

In the next section we describe the cutting algorithm which is based on the edge-based data structure presented in this section.

6.3 Cutting Algorithm

Our cutting algorithm follows the same strategy presented by Ganovelli et al. [26] which suggests use of lookup tables to handle different configurations. We also applied the optimizations suggested by Steinemann and Mor et al. [73, 51] to have minimal new elements added to the mesh cut. The novelty of our method lies in its generality in the scope of cut tools e.g., scalpel versus tube-shaped, and its performance for updating mesh connectivity in real-time, and finally its ability to handle cuts of various sizes and trajectories e.g., freehand versus axis-locked cuts.

The first stage in our cutting algorithm is detecting the cut sweep-surface. The input to this stage is the cut trajectory which is a list of points that the scalpel passed through in Euclidean space. The cut trajectory is not collected until the axis-aligned bounding box of the cutting tool intersects with that of the tissue. The tissue bounding box is expanded to rule out the boundary cases where the surface of the model contacts with its own bounding box. In such cases the scalpel might miss the surface if the bounding box test fails to detect the initial contact.

6.3.1 Edge Intersections

Using a GPU kernel function the intersection of the sweep-surface and all the edges of the model are computed. The input to this stage is the list of edges of the model and 4 points defining the sweep-surface quadrilateral. Since the intersection of a triangle and a line segment is faster to compute than a quadrilateral, we use two intersection tests per each edge to figure out whether the segment is split or not. The implementation of our edge triangle intersection follows the Ray-Triangle intersection test given in [2].

Similar to our GPU polygonization method in section 4.5, a prefix-sum operator counts the number of intersections and also compacts the resulting array of intersection points. The final output of this stage is a list of intersection points and the associated edge indices.

Algorithm 6 *EdgeIntersections* The kernel function that computes intersections of edges and the sweep-surface. The prototype of the kernel follows the listing above and the algorithm here represents one thread of the execution.

```
1: if $dim.x \geq countEdges$ then
2:     return
3: end if
4: $scanFlags[dim.x] \leftarrow 0$
5: $tri0 \leftarrow triangle(0, 1, 2, sweepSurface)$
6: $tri1 \leftarrow triangle(0, 2, 3, sweepSurface)$
7: $edge0 \leftarrow edgeBuffer[dim.x * 2]$
8: $edge1 \leftarrow edgeBuffer[dim.x * 2 + 1]$
9: $res \leftarrow IntersectSegmentTri(edge0, edge1, tri0, p)$
10: if $res = 0$ then
11:     $res = IntersectSegmentTri(edge0, edge1, tri1, p)$
12: end if
13: if $res \neq 0$ then
14:     $scanIntersections[dim.x] \leftarrow p$
15:     $scanIndices[dim.x] \leftarrow dim.x$
16:     $scanFlags[dim.x] \leftarrow 1$
17: end if
```

Each thread of execution will examine one edge of the mesh. The blade quadrilateral is divided to two triangles named $tri0$ and $tri1$ as shown in algorithm 6. The two endpoints of the current edge are retrieved from the mesh storage buffer called $edgebuffer$. The function named “IntersectSegmentTri” computes the intersection point of a line segment and a triangle. If the first triangle does not intersect with the blade end-points the test is repeated for the second triangle. If there is a valid intersection point, it is stored in the appropriate output buffer “scanIntersections”, the index of the intersected edge is stored at “scanIndices” and a flag that later identifies successful intersection tests is written to “scanFlags”. These buffers are compacted later using sum-scan operators discussed in the previous chapter.

6.3.2 Produce cut-node list

Following the improvement made by Steinmann et al. [73] to minimize the number of subdivided elements; per each intersection point which is “too close” to one of the edge endpoints; the sweep surface is snapped to that endpoint (see figure 6.5). The associated endpoint is also stored in a separate list called “cut-nodes”. In our system if an intersection point lies within the 20 percent of its associated edge length radius then it is considered as a “cut node”. This value avoided the most skinny elements. An analysis on the quality factors of the tetrahedral elements is made later in this chapter. The cut-nodes are later used to produce special subdivision cases which output non-skinny tetrahedral elements.

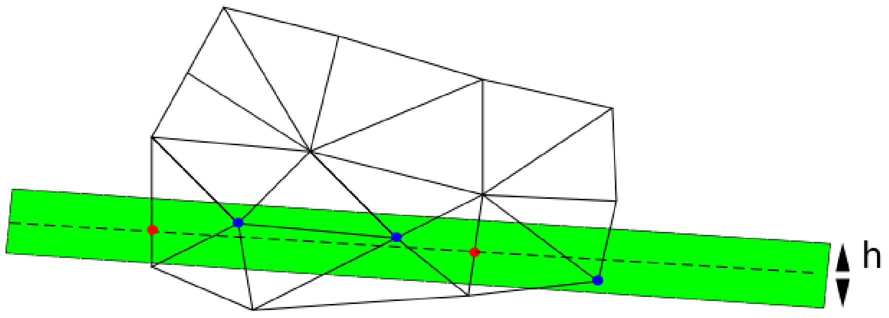


Figure 6.5: Dashed line represents the cut trajectory. For all the edges intersected by the cut sweep surface the end point closest to the sweep surface is selected. If the node lies within a threshold h, it is marked as a cut-node painted in blue and all the incident edges to that node are removed from the cut-edges list. The red dots are the intersection points associated with the remaining cut-edges.

Algorithm 7 *ProduceCutNodeList* The function that builds the cut-nodes list from the intersected edges. If an intersection is within the predefined distance of an edge endpoint it is considered as a “cut-node”.

```
 1: for $i = 0$ to $cutEdges.count - 1$ do
 2:     $ce \leftarrow cutEdges[i]$
 3:     $edge \leftarrow mesh.edge(ce.handle)$
 4:     $normalizedT \leftarrow ce.t/length(edge.from, edge.to)$
 5:     if $normalizedT < h$ then
 6:         $cutNodes.insert(edge.from)$
 7:     else
 8:         if $normalizedT > 1.0 - h$ then
 9:             $cutNodes.insert(edge.to)$
10:         end if
11:     end if
12: end for
```

In algorithm (7) the intersection distance is normalized using the length of the associated edge. If the normalized distance is within the predefined threshold h then the start point of the edge is a cutnode. Otherwise if the condition is true on the other end point then that end point is added as the cut-node.

6.3.3 Filter intersected-edges

All the edges that are incident to a "cut-node" are removed from the list of intersected-edges. Since our mesh data structure already has the bottom-up links, it is trivial to access all the incident edges per each node.

6.3.4 Compute configuration codes for cut elements

After building the list of cut-edges and cut-nodes, all the tetrahedral cells are inspected for intersection. The list of cut-edges and cut-nodes are implemented as hash-tables in our system, therefore, performing a query operation can be done in constant time and this step is not a bottleneck in our system. Per each cell all the 6 edges are checked against the cut-edges hash table in the same order given in figure 6.2. If an edge is included in the cut-edges hash table then a flag bit is set for that particular edge. The same process is performed for the 4 nodes of each cell and a 4 bits cutnode code is computed for that cell. Using the lookup table given in table 6.1 and the number of cut-edges and cut-nodes; per each cell a cut configuration is computed. If the number of cut-edges and cut-nodes are both zero then that cell is left intact otherwise it is added to the list of intersected cells to be processed later.

Table 6.1: The following look-up table is used to differentiate between different cutting configurations based on the number of cut-edges and cut-nodes. All cases subdivide tetrahedral elements into smaller cells except for case Z where the original cell is left intact.

type	state	#cut edges	#cut nodes
A	complete	3	0
B	complete	4	0
C	progressive	1	0
D	progressive	2	0
E	progressive	3	0
X	complete	2	1
Y	complete	1	2
Z	complete	0	3

Based on its cut configuration a tetrahedral cell is subdivided into smaller elements. In our system a separate lookup table is implemented per each of the above cut configurations. The reason is that each configuration leads to a different number of subdivided elements, however; it is also possible to combine all lookup tables into one. Table 6.2 shows the generated cells for the cut type A. In this configuration only 3 out 6 edges of a tetrahedra are cut and that should lead to $C(n, r) = C(6, 3) = 20$ different cut edges (without repetition and order does not matter). However, valid cut edges are the ones that separate one node from the other three that will lead to the compact lookup table in 6.2. The first column is the cut-edge code associated with that configuration row e.g. A56 denotes cut type A with edges 3, 4, 5 being cut where each each represent one bit in the code. The second column is the list of generated tetrahedral cells. Cut A produces 4 cells. Nodes 0-3 are the original nodes creating the original cell. When each edge shown in figure 6.2 is cut in the middle two additional nodes are added to that cell these extra nodes are defined at indices 4-15 as shown in figure 6.6.

The lookup table for type B is shown in table 6.3. Four edges are being cut in configuration type B and per each cell 6 sub-elements are generated. Cutting 4 out of 6 edges should result in $C(n, r) = C(6, 4) = 15$ distinct cut-edges. However, using the same analogy only 3 valid cuts are able to split the cell in two disjoint parts. Those cuts are summarized in table 6.3.

Case Z is trivial since the cell is left intact. For case Y two tetrahedral elements are being generated and one edge is being split only. Case X is very similar to A in the sense that one original node is being separated from the rest of the simplex. In cases X and Y the cut-nodes are duplicated and the new cells are generated based on the split edges and the original nodes and their associated duplicates. Since we haven't implemented progressive cutting yet cases C, D and E are left for future work.

Table 6.2: Lookup table for generating sub-elements for type A configuration

config	cut edges	generated cells
A56	3, 4, 5	{0, 12, 10, 14}, {3, 13, 11, 15}, {3, 1, 15, 11}, {1, 2, 3, 11}
A37	0, 2, 5	{1, 4, 8, 15}, {3, 9, 5, 14}, {0, 3, 14, 5}, {0, 2, 3, 5}
A11	0, 1, 3	{2, 5, 6, 11}, {3, 7, 10, 4}, {3, 1, 4, 10}, {0, 1, 3, 10}
A22	1, 2, 4	{3, 13, 9, 7}, {1, 8, 12, 6}, {1, 2, 12, 6}, {0, 1, 2, 12}

Table 6.3: Lookup table for generating sub-elements for type B configuration. Sub-elements are separated by semi-colon to fit on the line.

config	cut edges	generated cells
B46	1, 2, 3, 5	{2, 6, 11, 1; 8, 11, 15, 1; 6, 11, 8, 1; 3, 7, 9, 10; 3, 10, 0, 9; 0, 14, 10, 9}
B51	0, 1, 4, 5	{3, 7, 13, 15; 15, 1, 4, 7; 15, 1, 3, 7; 0, 12, 14, 6; 2, 5, 6, 14; 0, 14, 6, 2}
B29	0, 2, 3, 4	{4, 10, 12, 0; 0, 8, 12, 4; 0, 8, 1, 4; 3, 9, 13, 5; 2, 5, 11, 3; 3, 13, 5, 11}

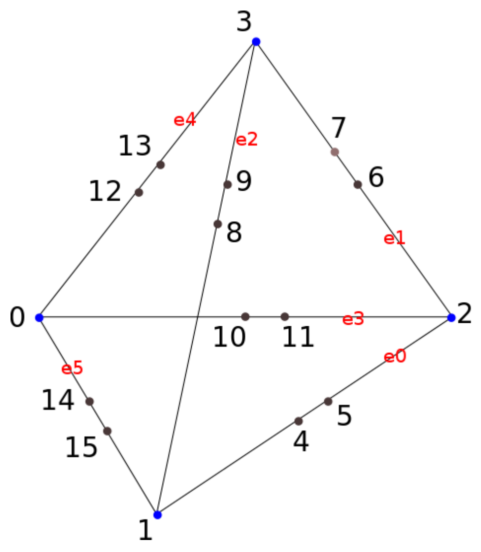


Figure 6.6: Nodes 0-3 are the original cell nodes shown in blue dots. Splitting each of the 6 edges can produce two additional nodes up to 12 more nodes which are placed at indices 4-15 shown in black dots.

6.3.5 Topological Updates

During the cutting process, the intersected cells are being replaced by sub-cells as discussed in the previous section. Cell removal operation involves updating the entire mesh structure which can be expensive if performed during the cut process. In our system all the intersected cells are being placed on a pending list for later deletion which will be visited by a post-processing stage called "garbage collection". This way the indices of the mesh entities are left intact during the cutting process and then once the cut is finalized the mesh can be cleaned and any unused entities such as cells, faces, edges or nodes are being purged to keep the structure as compact as possible for the next cutting operations.

The mentioned "garbage collection" process starts by visiting all the cells in the list of pending items for deletion. First, the list is sorted in increasing order of cell indices, this step is required to correctly update only the position of the items appearing in the subsequent index levels.

Then using the bottom up access links (list of incident cells per each face), the list of unreferenced faces is extracted which contains all faces that are not associated with any cells. The same operation is then performed for edges and nodes. At each step the list of unreferenced entities are sorted and all the items in those lists are removed from the internal container structures. Finally, all the containers are compacted and the indices of remaining entities are updated accordingly.

6.4 Cutting Results

In this section we review the results for the cutting algorithm presented in this chapter. Two models are considered for this analysis and a more complex cutting scenario is presented in chapter 7. The model shown in figure 6.7 is composed of two implicit spheres that are blended and tetrahedralized for physics simulation. The resulting volumetric mesh is cut horizontally, vertically and diagonally using the scalpel tool.

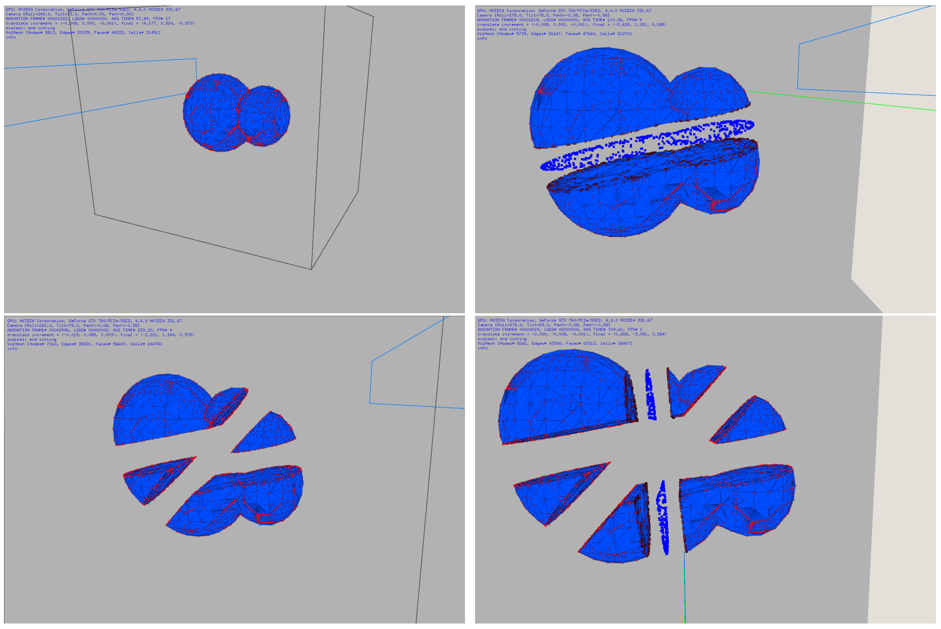


Figure 6.7: Two implicit spheres are blended and tetrahedralized for our physics simulation system. The peanut model is cut 3 times. Top-Left: The original volumetric mesh. Top-Right: Model cut horizontally with the scalpel tool. Bottom-Left: Diagonal cutting, Bottom-Right: Vertical cut. Blue dot represent the intersection points on the original edges.

The second model is the tumor model composed of 10 blended spheres. Using the same process the model is tetrahedralized for physical simulation and cut 3 times. Figure 6.8 shows the result of these topological modifications.

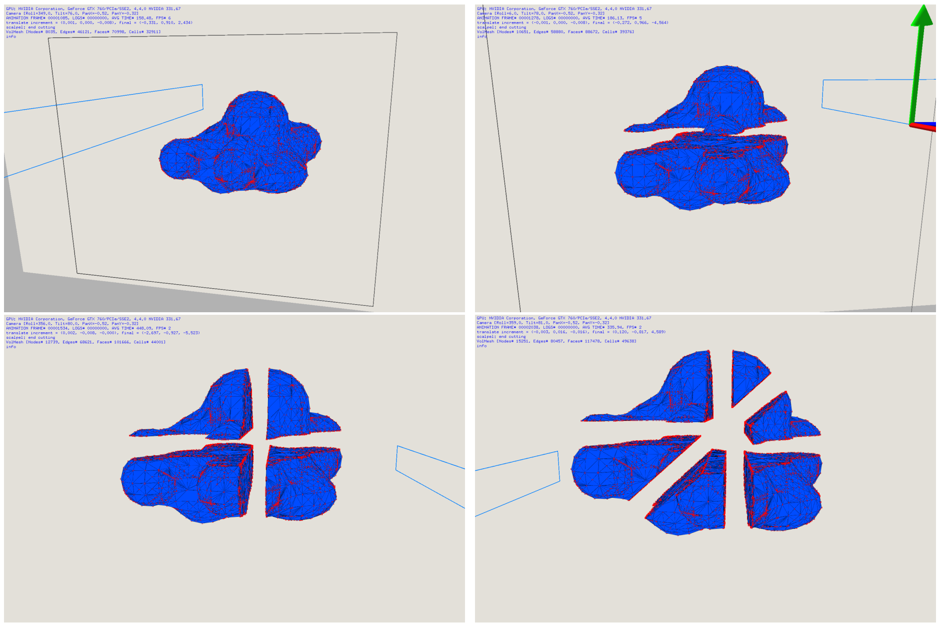


Figure 6.8: The tumor model above is composed of 10 point primitives and a blending operator. Top-Left: the original mesh, Top-Right: The mesh after a horizontal cut, Bottom-Left: The vertical cut, Bottom-Right: mesh after a diagonal cut.

To analyse the quality of the tetrahedral mesh after the cutting process several attributes are being considered. First, the number of elements before and after each cut. More elements will increase the system solve time and degrade the performance of the physical simulation due to the larger matrix dimensions involved in the process. Badly shaped (skinny) tetrahedral elements impose stricter constraints on the simulation time step [73, 26]. In order to get an understanding of the quality of the tetrahedral mesh after the cuts, element counts and several other quality measures are considered [55].

- The ratio between largest and smallest tetrahedron volume
- The ratio between largest and smallest edge length
- The lowest aspect ratio of all tetrahedra

The aspect ratio of a tetrahedra is measured as following:

$$\beta = \frac{CR}{3 * IR} \tag{6.2}$$

Where CR is the circumsphere radius of an element and IR is the inscribed-sphere computed by the following equation [55]:

$$IR = \frac{4V}{\sum_{i=1}^{4} SA_i} \tag{6.3}$$

Where V is the volume of a tetrahedra, SA_i is the surface area of the ith triangle face of an element.

Table 6.4: Mesh quality measurements.

	original peanut	peanut after 3 cuts	original tumor	tumor after 3 cuts
Nodes	4437	8933	8035	15821
Edges	25420	46466	46121	82711
Faces	39108	67388	70998	120212
Cells	18124	28322	32911	50605
V_{max}/V_{min}	84.927	58.253	316.898	287.596
l_{max}/l_{min}	42.367	47.547	40.045	130.556
$AspectRatio_{min}$	3×10^{-7}	3×10^{-7}	10^{-6}	10^{-6}

Graichen et al. presented an excellent study on the tetrahedral mesh quality measures [55]. Thin, wedge-like, flat and sliver elements (where 4 points of the element are co-planar) are the source of poor simulation results. Table 6.4 summarizes the statistical quality factors before and after cutting the two example meshes. The original volumetric mesh of both of these models are extracted from their associated *BlobTree* representation using our GPU tetrahedralization algorithm presented in chapter 4.

As it can be seen from the results the cutting method presented in this chapter does not increase the ratio of maximum to minimum volume after the cuts are made. This means that the element subdivision stage in our system does not introduce ill-shaped elements to the mesh. The ratio of the maximum to minimum edge-length did not change drastically. However, a local re-meshing process after the cut specially in the vicinity of the cut region can improve the quality of the mesh significantly.

The element count has increased steadily after each cut operation. Figure 6.9 shows the trend of element increase after N cuts are made to the mesh.

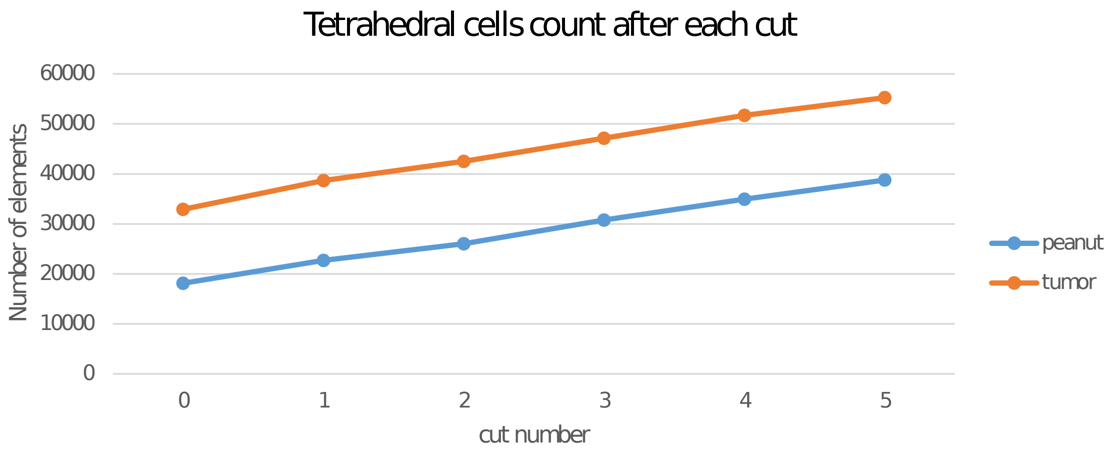
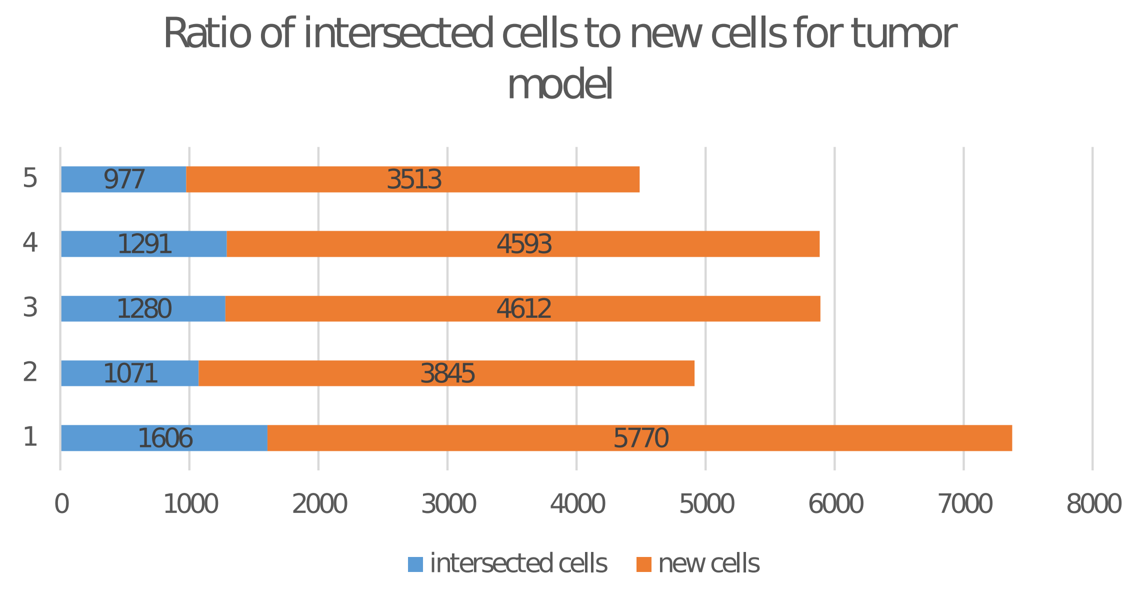


Figure 6.9: Number of tetrahedral cells after each cut operation. The horizontal axis is the cut number starting from cut 0 or the original mesh. The vertical axis is the number of cells.

Figure 6.10 provides a clear understanding of how many new cells are added to the mesh after each cut operation. The blue bar in this figure shows the number of cells that are intersected with the cut surface per each cut. Following the algorithm given in section 6.3.4 the intersected cells are subdivided and replaced by new set of generated cells shown in orange. On average the ratio of newly generated cells to intersected cells is 3.6 from these results. This is due to the fact that most of the cut configurations are of type A which result in a 1 to 4 subdivision.



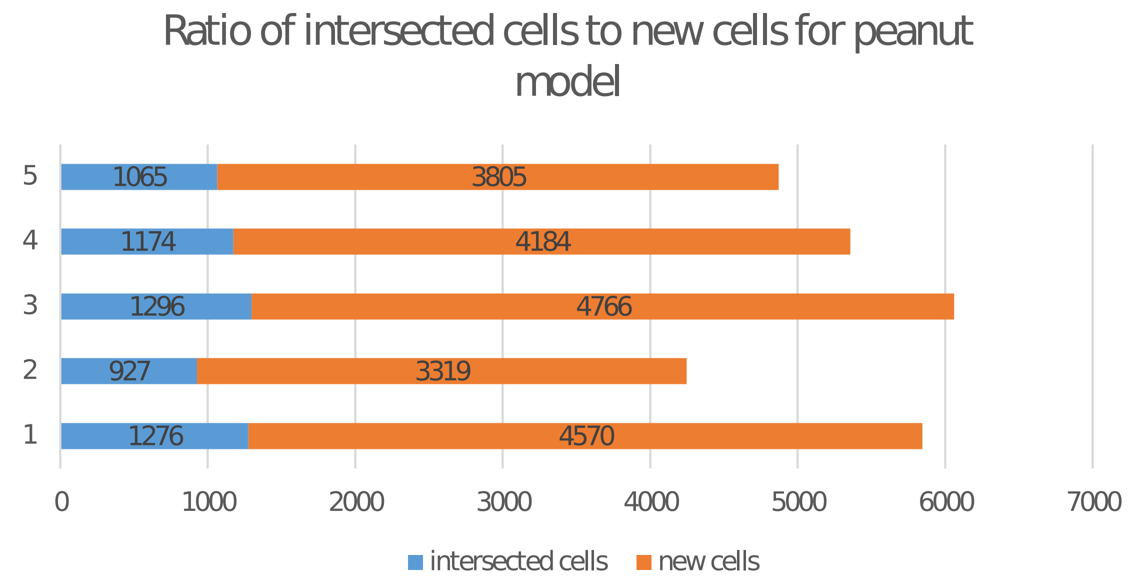


Figure 6.10: The ratio of intersected cells to newly added cells for tumor (top) and peanut model (bottom). Each blue bar represents the count of cells intersected with the scalpel tool while the orange bar next to it, is the number of newly generated cells after subdividing those intersected cells.

Chapter 7

Evaluation, Analysis and Comparisons

During the past decade a new thrust has appeared with the development of tissue modelling techniques which could make it possible to develop patient-specific simulations. Whenever the best surgical strategy is unclear or the patient presents a rare pathology such simulations could be beneficial. Other use-cases for such systems is in surgical skill training which is a long and tedious process of acquiring fine motor skills. In all those scenarios physically-based animation of tissues is challenging and has lots of room for improvement.

Our methods presented in the previous chapters have been integrated into our surgical simulation system. The result is a comprehensive environment for tissue modelling and simulation with support for cutting. We present our results in the context of a skull craniotomy procedure.

Craniotomy is a surgical operation in which a part of skull, called a bone flap is temporarily removed to access the brain. The operation is performed on patients with brain lesions, traumatic brain injury (TBI) or for brain biopsy purposes. Some treatment procedures are done by deep implantation of brain stimulators; Parkinson disease, epilepsy and cerebellar tremor are examples of such cases that require a craniotomy for the implantation process.

Small dime-sized craniotomies are called burr holes or keyhole craniotomies. In order to precisely control surgical and biopsy instruments through these small holes, the surgeons frequently use image-guided computer systems or endoscopes. Burr holes or keyhole craniotomies are used for minimally invasive procedures to:

- insert a shunt into the ventricles to drain cerebrospinal fluid (hydrocephalus)
- insert a deep brain stimulator to treat Parkinson Disease
- insert an intracranial pressure (ICP) monitor
- remove a small sample of abnormal tissue (needle biopsy)
- drain a blood clot (stereotactic hematoma aspiration)
- insert an endoscope to remove small tumors and clip aneurysms

In the following sections we review the previous work in this domain and then describe our experimental setup . The chapter concludes with results and analysis.

7.1 Previous work

Abe et al. fabricated plastic skull models of seven individual patients by stereolithography from three-dimensional data based on computed tomography (CT) bone images [1]. Surgical approaches and areas of craniotomy were evaluated using the fabricated skull models. They reported a better understanding of anatomic relationships, preoperative evaluation of the proposed procedure and improved educational experiences for the residents and medical staff as the benefits of their system. They also reported that the time and cost of making such models are the main disadvantages of using them.

Wong et al. loaded patient specific CT scans of cranial bone and CT angiography of intracranial circulation into the Dextroscope workstation supplied by Volume Interactions Pte. Ltd [80]. They showed various use-cases of the zoom, rotate, move and crop functions of the Dextroscope to visualize various angles of positioning the craniotomy. However, their system does not provide a physically-based simulation of the procedure.

Stadie et al. performed a study on the effectiveness of virtual reality systems for placing the craniotomies in minimally invasive procedures [72]. They used the Dextroscope and RadioDexter workstations supplied by Volume Interactions Pte. Ltd. to visualize and annotate the 3D VR models. Those systems are also used to measure the curvilinear distances of the proposed craniotomy centers on the patients skull model but they can not perform a cutting procedure on the input VR models.

7.2 Architectural constraints

We are able to perform interactive cuts on a model with more than 60,000 cells. The GPU accelerated cutting algorithm presented in chapter 6 requires a modern GPU with OpenCL support. The system we used for this experiment is equipped with an Nvidia Geforce GTX760 with 2GB of video memory and 1152 CUDA cores. The CPU is an Intel Core i7-4770K with 256 KB, 1MB and 8MB of L1 to L3 cache. This processor has 4 cores and up to 8 threads can run in parallel. Our system is also equipped with 16 GB of main memory.

7.3 Experiments

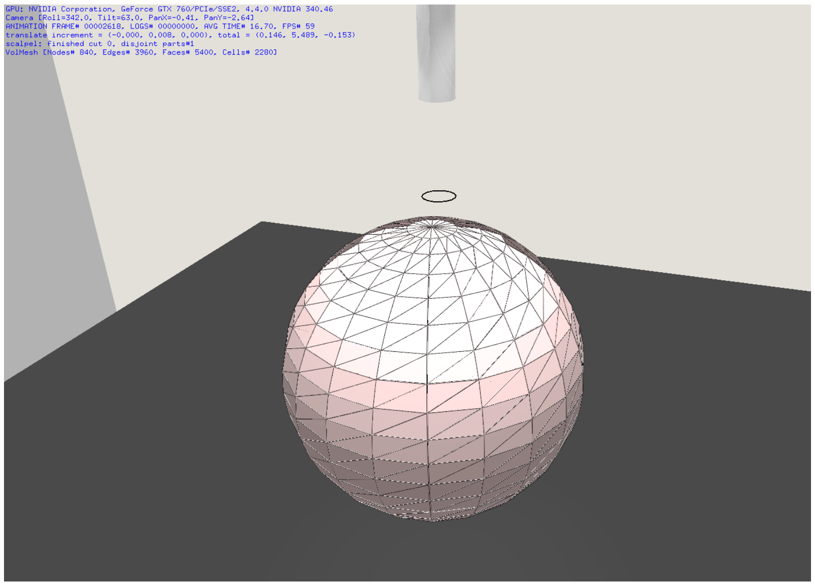
In this section we review the experiments that are performed to evaluate the performance and quality of our cutting algorithm in the context of the Craniotomy procedure. In the first experiment we use an eggshell model that helps us to study the quality of the cut surfaces and the amount of time it takes to apply mesh modifications in the associated data-structures. Fewer finite element cells in the eggshell model helps us to isolate the mesh quality issues in the vicinity of the cut surfaces.

Later we use the human skull model and drill multiple holes at different locations in the bone tissue. The location of the drills are chosen according to the recorded videos of various brain biopsy procedures. Beside the experiments with the drilling tool, the scalpel tool is also used for the visualization of the internal mesh cross-sections, which can be helpful in the study of scanned volume data-sets from arbitrary viewing angles.

Our system also allows controlled cutting along the major axes. The main benefit of this feature is in studying the effects of vibrations and other hand movements when cutting models interactively, i.e. by comparing the axis-locked, controlled cutting to the same scenario with free-hand movements, one can investigate the effect of small vibrations along the cut trajectory and how it can lead to the generation of small and wedge-shaped elements in the volume mesh.

7.3.1 The Eggshell Model

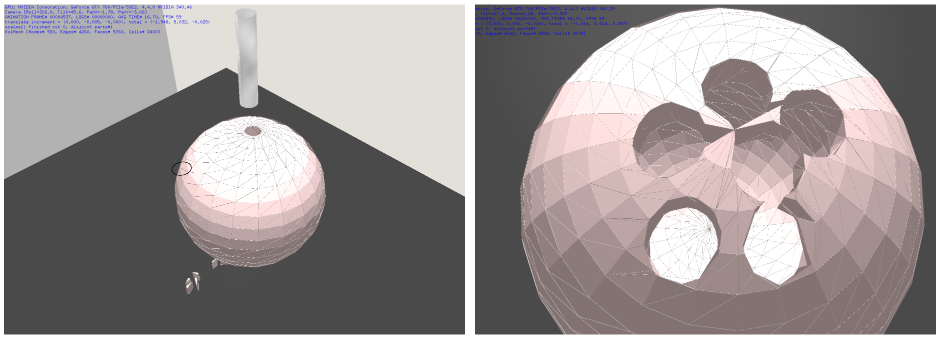
Before drilling an actual skull mesh with many tetrahedral elements we tested our system on a spherial model similar to an eggshell. This model is composed of 840 nodes and 2280 tetrahedral cells. The outer surface is composed of two layers of tetrahedral elements only (see figure 7.1). To create this model implicitly a smaller sphere is subtracted from a larger one.



```
GPU: NVIDIA Corporation, GeForce GTX 760/PCIe/SSE2, 4.4.0 NVIDIA 340.46
Camera [Roll=342.0, Tilt=63.0, PanX=-0.41, PanY=-2.64]
ANIMATION FRAME# 00002618, LOGS# 00000000, AVG TIME# 16.70, FPS# 59
translate increment = (-0.000, 0.008, 0.000), total = (0.146, 5.489, -0.153)
scalpel: finished cut 0, disjoint parts#1
VolMesh [Nodes# 840, Edges# 3960, Faces# 5400, Cells# 2280]
```

Figure 7.1: Eggshell model before being drilled by our cutting tool.

After the drilling operation, only 120 new elements are added to the mesh for a total of 2400 elements. The entire process is completed interactively. The small disjoint parts fall down on the ground due to gravity. Each disjoint part becomes an independent deformable model in our system and subject to forces and deformations. Figure 7.2 left, shows the mesh after being drilled for the bare hole. In order to perform a stress test on the model 6 holes are drilled in various locations of the eggshell model (see figure 7.2, right).



```
GPU: NVIDIA Corporation, GeForce GTX 760/PCIe/SSE2, 4.4.0 NVIDIA 340.46
Camera [Roll=316.0, Tilt=45.6, PanX=-1.78, PanY=-3.06]
ANIMATION FRAME# 00004537, LOGS# 00000000, AVG TIME# 16.70, FPS# 59
translate increment = (0.000, -0.005, -0.000), total = (-1.948, 5.032, -0.025)
scalpel: finished cut 0, disjoint parts#1
VolMesh [Nodes# 920, Edges# 4260, Faces# 5760, Cells# 2400]
```

```
ation, GeForce GTX 760/PCIe/SSE2, 4.4.0 NVIDIA 340.29
, Tilt=17.6, PanX=0.28, PanY=-0.33]
0008490, LOGS# 00000000, AVG TIME# 16.70, FPS# 59
t = (0.001, 0.000, -0.016), total = (-1.414, 2.414, 3.257)
cut 0, disjoint parts#1
23, Edges# 6642, Faces# 8994, Cells# 3678]
```

Figure 7.2: Eggshell model after being drilled by our cutting tool. Left: The first drill, Right: After drilling 6 holes to the model.

After completing this experiment successfully our algorithm showed to be robust enough to handle more complex meshes. In the next stage a real data-set of skull tissue is used for the craniotomy operation.

7.3.2 Craniotomy

Fang et al. published a tetrahedral mesh data-set of the brain tissues [23]. The MRI scanned data is segmented into the following four regions:

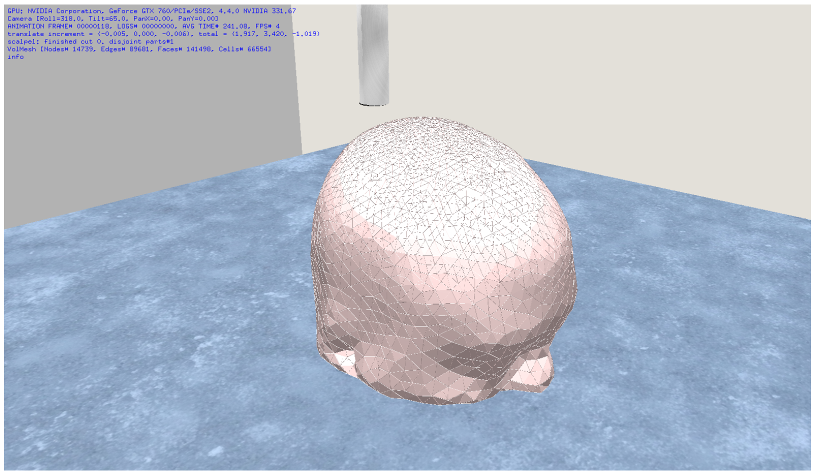
1. Skull and scalp
2. Cerebro-spinal fluid (CSF)
3. Gray-matter
4. White-matter

The high-resolution version of these segments are not published at the time of this writing so we used the lower resolution which has enough details of the organ for our simulation scenarios. The data-set is converted from its original format (MATLAB mat file) to our volumetric mesh format. Table 7.1 shows number of nodes, edges, faces and tetrahedral cells per each segment after the conversion process:

Table 7.1: Segmented brain data-set statistics.

	skull	csf	gray matter	white matter
Nodes	14739	37136	50741	23737
Edges	89681	181593	268300	126441
Faces	141498	251823	384989	184536
Cells	66554	107460	167528	81833

Due to its stiff material properties the skull tissue is modelled as a rigid material in our simulation system. Figure 7.3 shows the skull mesh in its initial position.



```
GPU: NVIDIA Corporation, GeForce GTX 760/PCIe/SSE2, 4.4.0 NVIDIA 331.67
Camera [Roll=318.0, Tilt=65.0, PanX=0.00, PanY=0.00]
ANIMATION FRAME# 00000118, LOGS# 00000000, AVG TIME# 241.08, FPS# 4
translate increment = (-0.005, 0.000, -0.006), total = (1.917, 3.420, -1.019)
scalpel: finished cut 0, disjoint parts#1
VolMesh [Nodes# 14739, Edges# 89681, Faces# 141498, Cells# 66554]
info
```

Figure 7.3: The scene setup for the craniotomy operation.

The cutting tool in this scenario is a tube-shaped device which can drill into the skull tissue and separate the bone matter. In our system, the tool is defined as a curve approximated by N line segments. The tool movement is tracked in the space and the system checks for collisions between the tool and the model continuously. Figure 7.4 shows the polygonal shape of the cutting tool while in contact with the skull tissue.

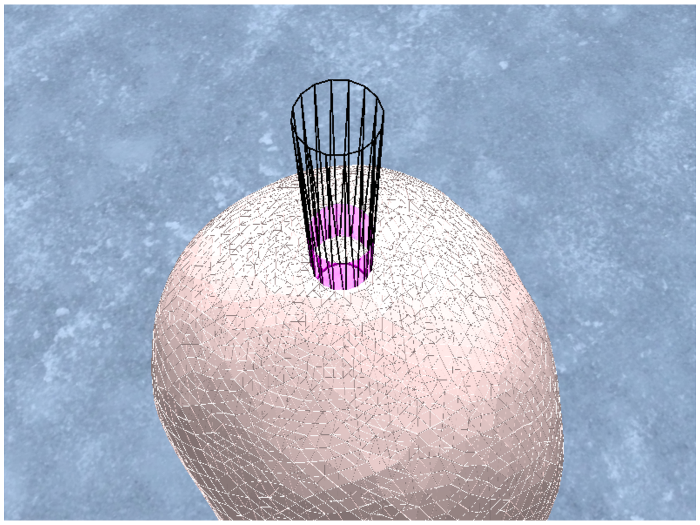


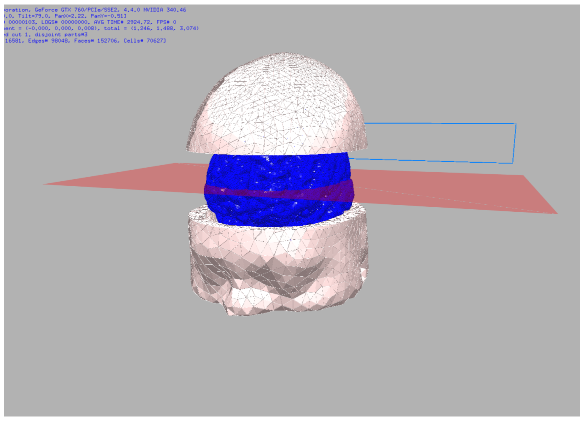
Figure 7.4: Cutting tool is defined as a tube with a base composed of a curve approximated with N line segments. Collisions between the tool and the tissue are monitored constantly.

When in contact with the skull tissue the intersection of the side wall of the tool is computed against the edges of the skull model. With N line segments the cutting tool has N quadrilateral faces on its side wall, the edge intersection test is performed in our system by calling the kernel function given in algorithm 6 once per each quad. After each call the hash-table storing the cut-edges is filled with the new cuts.

The cutting configurations presented in section 6.3.4 are extracted based on one intersection per edge, therefore it's not possible to cut a given edge more than once. In our system we only accept the first cut-edge and this did not produce any visual defects. Perhaps a more robust implementation would be to approximate the cutting tool curve based on the size of the longest edge in the mesh. After the cutting is made the mesh is separated and each of the disconnected parts is converted into a separate node in our scene-graph structure. This operation results in correct detection of the self-collisions in the subsequent frames of the simulation.

The internal gray, white and CSF matters are also included in this simulation. The drilling operation only affects the skull tissue and in fact it does not pass the Dura layer. This condition is a requirement for the successful completion of this procedure.

Figure 7.5 shows the cross section view of the brain. The skull is cut with a scalpel tool to show the internal tissues which are drawn in blue for better visibility.



```
poration, GeForce GTX 760/PCIe/SSE2, 4.4.0 NVIDIA 340.46
).0, Tilt=79.0, PanX=2.22, PanY=-0.51]
# 00000103, LOGS# 00000000, AVG TIME# 2924.72, FPS# 0
nent = (-0.000, 0.000, 0.008), total = (1.246, 1.488, 3.074)
ed cut 1, disjoint parts#3
 16581, Edges# 98048, Faces# 152706, Cells# 70627]
```

Figure 7.5: Cross section view of the brain layers. The skull shown in pink is cut using a scalpel avatar to show all the other layers depicted in blue.

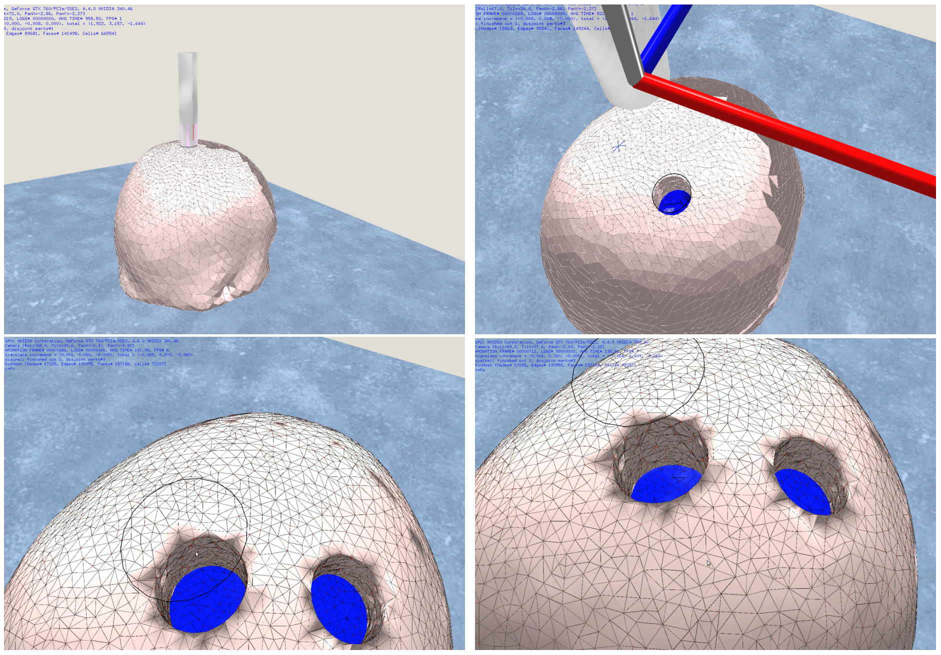


Figure 7.6: Simulation of the craniotomy operation using our surgical simulation framework with support for interactive cutting.

Figure 7.6 shows the operation (before, during and after drilling the first and the second holes). During the drilling process 845 and 940 tetrahedral cells are being cut in the vicinity of the drilling tool for the first and second holes, respectively. The interaction of the drilling tool and the skull was interactive at all times, supporting at least 60 frames per second.

Chapter 8

Conclusions

We presented a system for modelling and interactive rendering of complex implicit objects which bridges the gap between modelling and simulation. This is better than what has been done by Cani et al. [15].

Our SIMD polygonization method is peer reviewed [66]. The proposed algorithm is scalable, dynamic and data-driven as opposed to the related work in this area where the input model can be either simple static functions or constant range data-sets [36, 75, 41, 88]. The *BlobTree* pruning and linearization described in section 3.3.1, is an important step for designing compact and cache-line optimized input data-structure and the presented method is generic enough to be reused in applications involving hierarchical and tree-based structures.

Our proposed SIMD polygonization method is later optimized for using GPU acceleration. The proposed compact data-structure for *BlobTree* in section 4.2 enables the transfer and rendering of large *BlobTree* in the order of 60,000 nodes interactively (as shown in that chapter using our compact data-structure a 64K nodes *BlobTree* only takes about 20 MB in video memory). The result is a high performance polygonization method that enables real-time updates in our incremental modelling system. This result is better than the work of [41, 70] in rendering implicit models and the work of [88, 16] in polygonization techniques.

An intuitive volumetric mesh data-structure is proposed which is suitable for storing dynamic meshes on the GPU to support real-time modifications during cutting. Our cutting results show that the presented data-structure is more performant and can benefit the related work in this domain [82, 83, 19].

Our proposed GPU-based data-structure enables real-time updates of the volumetric meshes upon cutting. Our cutting algorithm as shown in section 6.3 is interactive for approximately 100,000 finite element cells. This is sufficient to cover many surgical scenarios. The resulting finite element cells are of high quality (the tetrahedral cells are not flat or wedge-shaped) as shown in section 6.4. This is better than what has been done in the work of Courtecuisse et al. [19]. The cut edges are smooth, not jagged and a minimal amount of tetrahedral elements are created as the result of elements subdivision and this is better than the results published by Courtecuisse et al. and Steinemann et al. [19, 73].

We presented a Craniotomy simulation based on our real-time cutting algorithm and the segmented brain data-set published by Fang et al. [23]. Given the large number of finite element cells in the brain mesh (around 100,000), our simulation still runs at interactive rates and the cutting output is smooth for a high-quality simulation as shown in section 7.3.2. Colchester et al. used superimposed surface mesh for guiding a Craniotomy simulation [18]. Abe et al. used plastic skull models for training this procedure [1]. To the best of our knowledge our proposed method is the only physically-based simulation for this specific procedure.

The following provides a summary of all the contributions made in this work:

1. A comprehensive modelling framework based on the *BlobTree* scenegraph for designing complex tissues. Our framework also provides a software architecture for physically-based animation of rigid and deformable models.

2. An algorithm for interactive polygonization of implicit surfaces on multi-core architectures with SIMD instructions (peer reviewed).

3. An optimized GPU-accelerated algorithm for high-performance polygonization of implicit surfaces on many-core architectures.

4. A novel mesh data-structure suitable for storing dynamic meshes on the GPU to support real-time modifications during cutting.

5. Smooth, interactive cutting for complex tissues

6. A real-time Craniotomy simulation for neurosurgery and biopsy simulations.

8.1 Future Work

There are many areas in which the proposed system may be improved. The polygonization method introduced in chapter 4 can be further extended to support more complex implicit primitives such as skeletal curve primitives. Such primitives can be helpful in modelling vain and other tube-shaped tissues. Support for implicit decals as suggested in [21] can help in creating realistically textured organs.

Our cutting algorithm can also be extended to support progressive cuts. Progressive cuts can enhance the level of realism perceived in cutting. Also incorporating a fluid simulation will enhance the cutting scenario for simulating blood and CSF fluids in the brain. Procedural models such as L-Systems can be used to simulate the small blood vessels in the brain biopsy simulation.

As computational power continues to increase, and optimization algorithms continue to improve, implicit surfaces are likely to play a much larger role in computer graphics. It is the sincere hope of the author that this research demonstrates what that role might be, and will encourage others to explore this domain.

Bibliography

[1] Masamitsu Abe, Kazuo Tabuchi, Masaaki Goto, and Akira Uchino. Model-based Surgical Planning and Simulation of Cranial Base Surgery. *Neurologia medico-chirurgica*, 38(11):746–751, 1998.

[2] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[3] Loïc Barthe, Brian Wyvill, and Erwin De Groot. Controllable Binary Csg Operators for "Soft Objects". *International Journal of Shape Modeling*, 10(02):135–154, 2004.

[4] Cagatay Basdogan, Suvranu De, Jung Kim, Manivannan Muniyandi, Hyun Kim, and Mandayam a Srinivasan. Haptics in minimally invasive surgical simulation and training. *IEEE computer graphics and applications*, 24(2):56–64, 2004.

[5] K J Bathe. *Finite element procedures*, volume 2. Prentice hall Englewood Cliffs, NJ, 1996.

[6] Adrien Bernhardt, Loic Barthe, Marie-Paule Cani, and Brian Wyvill. Implicit Blending Revisited. *Computer Graphics Forum*, 29(2):367–375, June 2010.

[7] D. Bielser, P. Glardon, M. Teschner, and M. Gross. A state machine for real-time cutting of tetrahedral meshes. *Graphical Models*, pages 377–386, 2004.

[8] Daniel Bielser, VA Maiwald, and MH Gross. Interactive Cuts through 3 Dimensional Soft Tissue. *Computer Graphics Forum*, 1999.

[9] J. Bloomenthal. An implicit surface polygonizer. *Graphics gems IV*, 1:324–349, 1994.

[10] J. Bloomenthal and B. Wyvill. Interactive techniques for implicit modeling. In *Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 109–116. ACM New York, NY, USA, 1990.

[11] Jules Bloomenthal, Bajaj Chandrajit, Jim Blinn, Marie-Paule Cani-Gascuel, Alyn Rockwood, Brian Wyvill, and Geoff Wyvill. Introduction to implicit surfaces. *Morgan Kaufmann*, 1997.

[12] William J Bolosky and Michael L Scott. False sharing and its effect on shared memory performance. *USENIX Systems on USENIX Experiences*, 1801:1–15, 1993.

[13] Mario Botsch, Leif Kobbelt, Pauly Mark, Alliez Pierre, and Levy Bruno. *Polygon Mesh Processing*. AK Peters, Ltd., 2010.

[14] Luc Buatois, Guillaume Caumon, and B. Lévy. GPU accelerated isosurface extraction on tetrahedral grids. *Advances in Visual Computing*, pages 383–392, 2006.

[15] Marie-Paule Cani-Gascuel and Mathieu Desbrun. Animation of Deformable Models Using Implicit Surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 3:39–50, 1997.

[16] Matúš ChochlÍk. GPU-Accelerated Polygonization of Implicit Surfaces. *Journal of Information, Control and Management Systems*, 10(2), 2012.

[17] P Cignoni, P Marino, C Montani, E Puppo, and R Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[18] AC Colchester and J. Zhao. Craniotomy simulation and guidance using a stereo video based tracking system (VISLAN). *Visualization in Biomedical Computing*, 2359:541–551, September 1994.

[19] Hadrien Courtecuisse, Hoeryong Jung, Jérémie Allard, Christian Duriez, Doo Yong Lee, and Stéphane Cotin. GPU-based real-time soft tissue deformation with cutting and haptic feedback. *Progress in biophysics and molecular biology*, 103(2-3):159–68, December 2010.

[20] Erwin De Groot. *Blobtree modelling*. PhD thesis, University of Calgary, 2008.

[21] Erwin de Groot, Brian Wyvill, Loïc Barthe, Ahmad Nasri, and Paul Lalonde. Implicit Decals: Interactive Editing of Repetitive Patterns on Surfaces. *Computer Graphics Forum*, 33(1):141–151, 2014.

[22] Carlos a. Dietrich, Carlos E. Scheidegger, João L.D. Comba, Luciana P. Nedel, and Cláudio T. Silva. Marching Cubes without Skinny Triangles. *Computing in Science & Engineering*, 11(2):82–87, March 2009.

[23] Qianqian Fang. Mesh-based Monte Carlo method using fast ray-tracing in Plücker coordinates. *Biomedical optics express*, 1(1):165–175, 2010.

[24] M. Fox, C. Galbraith, and B. Wyvill. Efficient use of the BlobTree for rendering purposes. In *Shape Modeling and Applications, SMI 2001 International Conference on.*, pages 306–314, May 2001.

[25] Laure France, Alexis Angelidis, Philippe Meseure, Marie-Paule Cani, Julien Lenoir, François Faure, and Christophe Chaillou. Implicit Representations of the Human Intestines for Surgery Simulation. *ESAIM: Proceedings*, 12:42–47, 2002.

[26] F Ganovelli and P Cignoni. A multiresolution model for soft objects supporting interactive cuts and lacerations. *Computer Graphics Forum*, 19(3):271–281, 2000.

[27] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.* Newnes, 2012.

[28] Joachim Georgii and Rüdiger Westermann. Mass-spring systems on the GPU. *Simulation Modelling Practice and Theory*, 13(8):693–702, November 2005.

[29] SFF Gibson and B Mirtich. A survey of deformable modeling in computer graphics. *MERL - A MITSUBISHI ELECTRIC RESEARCH LABORATORY*, 1997.

[30] Abel Gomes, Irina Voiculescu, Joaquim Jorge, Brian Wyvill, and Callum Galbraith. *Implicit Curves and Surfaces: Mathematics, Data Structures, and Algorithms.* Springer Verlag, 2009.

[31] Herbert Grasberger, Pourya Shirazian, Brian Wyvill, and Saul Greenberg. A data-efficient collaborative modelling method using websockets and the BlobTree for over-the air networks. *Proceedings of the 18th International Conference on 3D Web Technology - Web3D '13*, pages 29–37, 2013.

[32] A Guéziec and R Hummel. Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):342, 1995.

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

[34] Lenka JeÅŹábková, Guillaume Bousquet, Sébastien Barbier, François Faure, and Jérémie Allard. Volumetric modeling and interactive cutting of deformable bodies. *Progress in biophysics and molecular biology*, 103(2-3):217–24, December 2010.

[35] Xia Jin, Grand Roman Joldes, Karol Miller, and Adam Wittek. Computational Biomechanics for Medicine. pages 73–80, 2013.

[36] Gunnar Johansson and Hamish Carr. Accelerating marching cubes with graphics hardware. *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research - CASCON '06*, page 39, 2006.

[37] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. *ACM Transactions on Graphics (TOG)*, 21(3):339–346, July 2002.

[38] Takashi Kanai, Y Ohtake, H Kawata, and K Kase. Gpu-based rendering of sparse low-degree implicit surfaces. *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 165–171, 2006.

[39] Peter Kipfer and R. Westermann. GPU construction and transparent rendering of iso-surfaces. In *Proceedings Vision, Modeling and Visualization*, volume 5, 2005.

[40] A. Knoll, Y. Hijazi, A. Kensler, M. Schott, C. Hansen, and H. Hagen. Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic. *Computer Graphics Forum*, 28(1):26–40, March 2009.

[41] Aaron Knoll, Younis Hijazi, Charles Hansen, Ingo Wald, and Hans Hagen. Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic. *2007 IEEE Symposium on Interactive Ray Tracing*, pages 11–18, September 2007.

[42] F Labelle and JR Shewchuk. Isosurface stuffing: fast tetrahedral meshes with good dihedral angles. *ACM Transactions on Graphics (TOG)*, pages 1–10, 2007.

[43] S.D. Laycock and A.M. Day. A Survey of Haptic Rendering Techniques. *Computer Graphics Forum*, 26(1):50–65, March 2007.

[44] Michael Lee and Hanan Samet. Navigating through triangle meshes implemented as linear quadtrees. *ACM Transactions on Graphics (TOG)*, 19(2):79–121, April 2000.

[45] Ming Lin and Kenneth Salisbury. Haptic Rendering - Beyond Visual. *Ieee Computer Graphics And Applications*, (April):22–23, 2004.

[46] A Liu, F Tendick, K Cleary, and C Kaufmann. A survey of surgical simulation: applications, technology, and education. *Presence: Teleoperators and Virtual Environments*, 12(6):599–614, 2003.

[47] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, volume 87, page 169. ACM, 1987.

[48] P Mackerras. A fast parallel marching-cubes implementation on the Fujitsu AP1000. *Computer Science Technical Report TR-CS-92-10, The Australian National University*, 1992.

[49] JH Matthews. Numerical Methods for Computer Science, Engineering and Mathematics. 1987.

[50] U Meier, O López, C Monserrat, M C Juan, and M Alcañiz. Real-time deformable models for surgery simulation: a survey. *Computer methods and programs in biomedicine*, 77(3):183–97, March 2005.

[51] AB Mor and Takeo Kanade. Modifying soft tissue models: Progressive cutting with minimal new element creation. *Medical Image Computing and Computer-Assisted Intervention–MICCAI*, pages 598–607, 2000.

[52] Andrew Nealen and Matthias Müller. Physically based deformable models in computer graphics. *Computer Graphics Forum*, 25:809–836, 2006.

[53] Han-Wen Nienhuys and A Frank van der Stappen. A Surgery Simulation Supporting Cuts and Finite Element Deformation. In *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2001: Lecture Notes in Computer Science*, volume 2208, pages 145–152. 2001.

[54] HW Nienhuys and AF van der Stappen. Supporting cuts and finite element deformation in interactive surgery simulation. *Procs. of the Fourth International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, pages 145–152, 2001.

[55] V N Parthasarathy, C M Graichen, and A F Hathaway. A comparison of tetrahedron quality measures. *Finite Elements in Analysis and Design*, 15(3):255–261, 1994.

[56] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007.

[57] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*, volume 23. O'Reilly Media, Inc., 2007.

[58] B Rodriguesdearaujo and J Armandopiresjorge. Adaptive polygonization of implicit surfaces. *Computers & Graphics*, 29(5):686–696, 2005.

[59] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, (2), 1984.

[60] Hanan Samet. Hierarchical spatial data structures. *Design and Implementation of Large Spatial Databases*, pages 191–212, 1990.

[61] R. Schmidt and B. Wyvill. Implicit sweep surfaces. Technical report, Citeseer, 2005.

[62] R. Schmidt, B. Wyvill, and E. Galin. Interactive implicit modeling with hierarchical spatial caching. *International Conference on Shape Modeling and Applications 2005 (SMI' 05)*, pages 104–113, 2005.

[63] R. Schmidt, B. Wyvill, MC C Sousa, and JA A Jorge. Shapeshop: Sketch-based solid modeling with blobtrees. In *ACM SIGGRAPH 2006 Courses*, page 14. ACM, 2006.

[64] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, (GH '07):97–106, 2007.

[65] Aria Shahingohar and Roy Eagleson. A framework for GPU accelerated deformable object modeling. *International Journal of High Performance Computing Applications*, 26:203–214, November 2012.

[66] Pourya Shirazian, Brian Wyvill, and Jean-Luc Duprat. Polygonization of implicit surfaces on Multi-Core Architectures with SIMD instructions. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 89–98, Cagliari, Italy, 2012.

[67] P Shirley, S Marschner, M Ashikhmin, M Gleicher, N Hoffman, G Johnson, T Munzner, E Reinhard, K Sung, W Thompson, P Willemsen, and B Wyvill. *Fundamentals of computer graphics*. AK Peters, Ltd., 2009.

[68] Hang Si. TetGen - A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator. Technical report, 2006.

[69] Eftychios Sifakis, KG Der, and R Fedkiw. Arbitrary cutting of deformable tetrahedralized objects. *ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 73–80, 2007.

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

[71] Thomas Sangild Sø rensen and Jesper Mosegaard. An Introduction to GPU Accelerated Surgical Simulation. pages 93–104, 2006.

[72] Axel T. Stadie, Ralf A. Kockro, Luis Serra, Gerrit Fischer, Eike Schwandt, Peter Grunert, and Robert Reisch. Neurosurgical craniotomy localization using a virtual reality planning system versus intraoperative image-guided navigation. *International Journal of Computer Assisted Radiology and Surgery*, 6:565–572, 2011.

[73] D. Steinemann, M. Harders, M. Gross, and G. Szekely. Hybrid Cutting of Deformable Solids. *IEEE Virtual Reality Conference (VR 2006)*, pages 35–42, 2006.

[74] Denis Steinemann, MA Otaduy, and Markus Gross. Fast arbitrary splitting of deforming objects. *Eurographics/ ACMSIGGRAPH Symposium on Computer Animation*, pages 63–73, 2006.

[75] N. Tatarchuk, J. Shopf, and C. DeCoro. Real-Time isosurface extraction using the GPU programmable geometry pipeline. In *ACM SIGGRAPH 2007 courses*, page 137. ACM, 2007.

[76] N Tatarchuk, J Shopf, and C Decoro. Advanced interactive medical visualization on the GPU. *Journal of Parallel and Distributed Computing*, 68(10):1319–1328, 2008.

[77] Frederic Triquet, Laurent Grisoni, Philippe Meseure, and Christophe Chaillou. Realtime visualization of implicit objects with contact control. *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Austalasia and South East Asia - GRAPHITE '03*, 1(212):189, 2003.

[78] Greg Turk and JF O'Brien. Shape transformation using variational implicit functions. *ACM SIGGRAPH 2005 Courses*, 2005.

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

[80] George K C Wong, Canon X L Zhu, Anil T. Ahuja, and Wai S. Poon. Craniotomy and clipping of intracranial aneurysm in a stereoscopic virtual reality environment. *Neurosurgery*, 61:564–568, 2007.

[81] Jun Wu, Christian Dick, and Rüdiger Westermann. Interactive High-Resolution Boundary Surfaces for Deformable Bodies with Changing Topology. *Proceedings of 8th Workshop on Virtual Reality Interaction and Physical Simulation (VRI-PHYS)*, pages 29–38, 2011.

[82] Wen Wu and Pheng Ann Heng. A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting. *Computer Animation and Virtual Worlds*, 15(34):219–227, July 2004.

[83] Wen Wu and Pheng Ann Heng. An improved scheme of an interactive finite element model for 3D soft-tissue cutting and deformation. *The Visual Computer*, 21(8-10):707–716, August 2005.

[84] B. Wyvill, A. Guy, and E. Galin. Extending the CSG Tree - Warping, Blending and Boolean Operations in an Implicit Surface Modeling System. In *Computer Graphics Forum*, volume 18, pages 149–158. John Wiley & Sons, 1999.

[85] B. Wyvill and K. van Overveld. Polygonization of implicit surfaces with constructive solid geometry. *International Journal of Shape Modeling*, 2(4):257–274, 1996.

[86] Brian Wyvill and Kees van Overveld. Warping as a modelling tool for csg/implicit models. In *Shape Modelling Conference, University of Aizu, Japan*, volume m, pages 1–20, 1997.

[87] G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *The visual computer*, 2(4):227–234, 1986.

[88] Bin Yang, Gui-Lin Chen, and Ming-Yong Pang. Parallel Polygonization of Implicit Surfaces. *2010 International Symposium on Intelligence Information Processing and Trusted Computing*, pages 220–223, October 2010.

[89] Yi Zhang, Xin Wang, and X.J. Wu. Fast Visualization Algorithm for Implicit Surfaces. In *Artificial Reality and Telexistence–Workshops, 2006. ICAT '06. 16th International Conference on*, pages 339–344. IEEE Computer Society, 2006.