Using Mesh Cutting in an Interactive Podiatric Orthopaedic Surgery Simulator

Andrew Boothroyd

A thesis submitted to the University of Huddersfield in partial fulfilment of the requirements for the degree of Master of Science by Research

The University of Huddersfield

January 2011

Copyright Statement

Abstract

Serious games are an established pedagogical tool, with applications in a wide variety of fields. Although this includes the area of medical training, there is currently no podiatric orthopaedic training simulator. We attempt to overcome one of the obstacles to the creation of this kind of simulator, namely how to simulate podiatric bone surgery. In order to simulate this surgery appropriately, it is necessary to be able to cut through a virtual representation of a patient's foot on-screen in real-time. We investigate several methods of cutting through simulated objects in general, and evaluate their usefulness in simulating real-time interactive bone surgery. We determine that none of these conventional methods are fully suitable and instead propose, develop and test a method using planar slicing of polyhedral mesh geometry.

In addition, we describe and test some optimizations to vertex and index buffer representations of polyhedral meshes. These optimizations alleviate a performance bottleneck constraining the speed at which progressive mesh modifications can be made, facilitating the near-real-time modification rate required for simulating surgery. We also investigate the use of haptic feedback to simulate the feel of surgery, and employ two games technologies to increase the realism of the simulation: particle engines and stereoscopic 3D rendering.

Contents

Word count: 23264

List of Figures

List of Tables

Glossary

CSG	Constructive Solid Geometry; A system of representing geometry as a series of Boolean set operations performed on primitives
Index buffer	A contiguous chunk of memory for storing indices which reference entries in a vertex buffer, allowing deduplication of vertices
Mesh	A collection of connected polygons, commonly stored as triangles in index and vertex buffers
Metaballs	A system of representing implicit geometry, often collections of ‘blobby’ spheres
Octree	A spatial partitioning system that allows efficient searching through geometry
Orthopaedic	The branch of medicine specialising in correcting problems relating to the skeletal system
Podiatric	The branch of medicine concerning the feet and lower extremities
Serious game	An activity that uses methods commonly employed for interactive entertainment to instruct or teach
Vertex buffer	A contiguous chunk of memory for storing properties of a mesh’s vertices, each entry usually containing its position, normal, texture coordinates, colour, etc.
Volumetric	A method of representing geometry by storing its volume as a collection of uniform elements in 3D space
Voxel	An element indicating the presence or absence of a unit of volume within a volumetric model, usually a cube

Chapter 1

Introduction

The concept of training simulators has been around for many years (Bell and O'Keefe, 1987). Perhaps the most well-known type of simulator is the flight simulator, used as part of a training regime to train pilots and to provide entertainment. In the field of Serious Games, software applications are developed that repurpose entertainment-based computer game technologies for training, education and other serious purposes (Sørensen, 2005). Several serious games provide medical training, with some of the more advanced examples simulating surgery (e.g. France et al., 2005).

This research investigates methods for simulating orthopaedic foot surgery. This kind of surgery is carried out by both podiatric surgeons and orthopaedic surgeons, and is used to treat a variety of ailments affecting the bones of the lower extremities (O'Kane and Kilmartin, 2007). The goal of this research is to lay the foundation for developing a full-featured serious game simulating podiatric orthopaedic surgery, by overcoming one of the major technical obstacles. Such a simulator would be useful to consultant surgeons for demonstrating surgical procedures to trainee surgeons, and useful to trainees by allowing them to practice such procedures in a safe, controlled environment (Haluck et al., 2001).

In this introduction, we intend to briefly discuss how serious games technology can be used in a podiatric orthopaedic surgery simulator, and to provide an overview of the main problem this research intends to overcome.

1.1 Simulating Surgery

The intention of this research is to facilitate the future development of a training simulator that allows the user to experience performing bone surgery as realistically as possible, in a safe, repeatable, cost-effective manner. The main feature of surgical operations we intend to simulate is cutting through a bone. This is performed in some operations in order to remove an unwanted portion of the bone

— for instance, in a bunionectomy (Mitchell et al., 1958) — and in others to temporarily sever part of a bone in order to reposition it, such as in the Scarf and Akin osteotomy (O'Kane and Kilmartin, 2002). Repositioning of bones can be used to lengthen or shorten a given bone if necessary, by making a diagonal cut all the way through a bone and sliding the two parts along the tangent of the cut.

Ideally, the user of the finished simulator would be able to repeatably experience a variety of podiatric orthopaedic operations, with both visual simulation and realistic tactile feedback provided by the simulator. Since we are primarily concerned with bone surgery, we do not intend to simulate the soft tissue surrounding bones, so the simulator will focus on operating on the skeleton of the simulated patient's foot. Therefore, all the techniques of direct bone surgery used in the operation will be simulated, but supporting techniques such as anaesthetics, applying tourniquets, performing incisions into the skin to reveal the bone, suturing the wound, etc. will not be included and presumed to have been completed successfully.

1.2 Use of Serious Games

Although physical simulators can be used to train surgeons, this research aims to produce software and algorithms to be used in a computer-based training simulation. Since surgical operations take place in 3 dimensions in the real world, and the experience and motor skills we intend to convey cannot reasonably be transferred to a 2-dimensional representation, any surgical simulation software will necessarily entail a 3D rendered view of the operation scene. Such a scene will contain the subject of the operation (in our case, the skeleton of a simulated patient's foot) and all tools relevant to the operation being performed. The simulated tools available in the application at this stage of research will primarily include a surgical bone-saw, but in the finished product may also included a drill, cutting guide and wire insertion tool.

Although some serious games are physical games (board- or card-games), and some are 2D games, there are a large selection of serious games that attempt to fully immerse the player in a realistic 3D-rendered environment. Such games primarily employ technologies originally developed by the (entertainment-focussed) games industry for simulating 3D game worlds, objects, characters, tools, weapons, in-world controls, out-of-world information displays, and many other relevant technologies and methodologies.

Therefore, in order to develop an accurate computer simulation of podiatric orthopaedic surgery, we will employ techniques that are used in 3D serious games — whose roots lie in the entertainment games industry. We intend to present the user with a 3D view of the skeleton of the patient's foot, and provide a necessary control interface for the required simulated surgical tools to carry out an operation. The flow of the application will be more similar to that of a serious game than a conventional window-based application, with the user progressing through a series of screens filling the entire application window. For example, one screen will allow the user to select an operation to attempt, which will lead to a second screen featuring a 3D environment replicating the chosen operation. In a finished serious game derived from our work, the user would also be given a score based on their performance while conducting the operation. The user's speed, accuracy and appropriate use of surgical tools will be assessed to determine their score. This, combined with a system for managing multiple users, would provide metrics allowing them to measure their skill at performing given operations, and allow improvements in their performance to be measured over time. However, implementing specific, realistic scenarios with actual pedagogical benefit is beyond the scope of this work.

1.3 Mesh Cutting

In order to allow the user to perform orthopaedic operations in a simulation, it is first necessary to present virtual representations of the patient's foot and the surgical implements to the user. There are several systems of representing 3D objects in a computer simulation, the most relevant of which are discussed in chapter 2. However, our system uses 3D polygon meshes rendered in a conventional manner, discussed in section 5.2. In order to perform surgical manipulation of the simulated patient's foot when represented this way, it is necessary to be able to cut through polygon meshes in a manner simulating cutting through actual bone.

Typically, in 3D graphics, meshes are either rendered unmodified (for instance, with static scenery) or animated. Meshes can be animated in a variety of ways, which include simple geometric transformations, skeletal animation, and facial animation. The most common manner in which these methods are employed is in replaying pre-recorded animations, although some animation systems generate animations on-the-fly in response to the mesh's environment and stimuli. Notable examples of such animation systems include various common rag-doll physics systems, and the Euphoria dynamic animation engine (McEachern, 2008). However, even with dynamically generated animations and interactions, the meshes in question are generally animated using a pre-determined 'skin' and a collection of jointed 'bones' — sets of vertices that are associated with a simulated skeleton. Such animation methods can be used to provide a full range of motion for the 'bones' attached to the mesh — often these correspond with accurately-simulated limbs and joints — but such methods can only move vertices around and not normally break or cut into the mesh.

In order to simulate a user cutting into a mesh, it is necessary to perform geometric computations to deform the shape of the mesh in real-time, whilst maintaining the illusion of the mesh being a solid object. Any changes made to the mesh must also be reflected in the data processed by the underlying graphics rendering pipeline before they are visible to the user. In order to provide an accurate interactive simulation, changes to the mesh made in response to the user's input must be made at the refresh rate of the application. This presents the challenge of carrying out the relevant geometric changes using only the limited CPU, GPU, main memory, graphics memory, memory bus and disk I/O resources available within each frame. A suitable cutting algorithm must be able to operate using minimal amounts of these resources, as the remainder of the simulation's functionality (such as rendering the 3D environment, drawing the user interface, responding to events, etc.) will also consume a considerable portion of the available resources.

1.4 Conclusion

This thesis explores different methods used to cut through meshes, aiming to find and implement a suitable algorithm that allows realistic cuts to be made into a mesh in real-time. Chapter 2 reviews the literature on the subject of serious games, relevant medical simulations and mesh manipulation techniques currently available. Chapter 3 discusses the research methodology used to obtain and test a suitable cutting algorithm. Chapter 4 evaluates existing mesh manipulation methods, and proposes a method using planar slicing. Chapter 5 discusses how we store mesh geometry, and optimisations that can be made to this. Chapter 6 discusses the details of our particular implementation for testing purposes. Chapter 7 presents the results of testing our optimisations and cutting algorithm. Chapter 8 presents our conclusions and discusses future work.

Chapter 2

Literature Review

This chapter provides an overview of the current literature available on the topics of serious games, surgical cutting simulations and mesh cutting algorithms. The chapter discusses the nature of serious games, and describes some of the design elements, methodologies and technologies that are adopted from entertainment computer games. It also introduces the concept of mesh cutting and provides a brief look at the cutting algorithms currently described in the literature.

2.1 Introduction

The concept of serious games has been around for some time. Serious games aim to utilise technology and methodology normally employed in entertainment-based games, and convert them to serve educational or other more practical purposes (Stapleton, 2004). Some serious games take the form of board and card games, but recently many computer- and video-games have been produced specifically for serious, non-entertainment purposes. For instance, the US government funds the creation, distribution and maintenance of *America's Army* (Li, 2004). This is a game that attempts to accurately simulate actual military service — for recruitment and propaganda purposes — whilst maintaining the entertainment features underpinning games of its genre. Another example is *BioHazard*, a simulator used to train firefighters how to deal with emergency situations resulting from chemical and biological weapon attacks (Squire and Jenkins, 2003). The following section reviews literature available on the current state of serious games, and examines how they can be used for medical simulation.

2.2 Use of Serious Games Technology and Methodology

Stapleton (2004) points out that "Game studios are in the business of creating products where success is dependent on the ability of players to learn and overcome the various challenges and obstacles within the game." Very few, if any, video games can be completed by a player who has no prior knowledge of the game platform, genre and controls. Similarly, how to 'beat' the game must be learnt — the layout of game levels, the strategies required to beat certain enemies and the method to solve puzzles must all be discovered by the player as they play the game. Stapleton contrasts the learning that takes place within a computer game to that which takes place within a classroom. He notes that, in computer games, the player has full control over their learning, including the pace and content of their learning; whereas with conventional classroom-based learning, the lesson is conducted at a pace dictated by the instructor and as such is more focussed on the teacher. Stapleton believes this gives serious games greater teaching and learning potential than traditional methods.

Blackman (2005) explores several advantages that computer-based serious games offer over traditional off-line educational methods. She cites the example of learning to identify components in a piece of machinery, which would traditionally use a diagram of an 'exploded' view of the machinery to indicate the identity of components and how they fit together. This is contrasted with a theoretical interactive view of machinery components that allows the user to select components to view individually, and display pertinent information accordingly. This avoids the overwhelmingly large amount of information that is necessary for the traditional diagram to be used with machinery containing a large number of components. Additionally, Blackman notes that entirely new learning methods are possible through serious games, such as allowing the user to view a 3D model of the working machinery. Of particular relevance to our research, she expands thus:

> *"Educational I3D [Interactive 3D] applications can be as complex and serious as visualizations that teach medical students everything from bone, organ or tissue identification... to their first look at the steps involved in performing a complicated medical procedure. Almost any situation that requires previous knowledge, skill or decision-making could be set in an interactive 3D environment for considerably more effective results."*

Indeed, research into medical simulation has already yielded several such applications (Meier et al., 1985; Edmond et al., 1998; Fosse et al., 2001; Delp et al., 2002; Neubauer et al., 2004; Chen et al., 2005; Vidal et al., 2005; De and Lim, 2006). Although some medical applications simulate general medical practice and planning (for example, Ramis et al. (2001) simulates different methods of managing a surgery centre in order to optimise patient throughput), many other applications provide 3D simulations of surgical procedures and soft tissue manipulation.

2.3 Surgical Simulation

2.3.1 Soft-tissue Simulation

Choi et al. (2002) propose and develop a system of deforming soft tissue in real-time using a mass-spring elasticity model. Although soft-tissue simulation had previously been implemented by others, earlier methods were typically computationally intensive and not capable of running at the refresh rates required for a fully-interactive real-time simulation (Delingette, 1998). In order to solve this problem, the authors use a force-propagation algorithm to vastly reduce the number of calculations requires per simulation step whilst maintaining a good approximation of real-world tissue elasticity. In one example, the system displays a 3D model of the human stomach, to which a force is applied. The system is then used to iteratively calculate the relevant forces applied to or exerted by all the vertices near the force application point, causing the mesh to deform realistically.

2.3.2 Bone Simulation

Other research has been focussed on the simulation of bone surgery. Pflesser et al. (2000) presents a volume rendering system for visualizing bone dissections (such as mastoidectomies), which uses a sub-voxel rendering method to accurately simulate sections of bone being cut with a variety of tools. A further-developed version of this system is also described in the literature (Petersik et al., 2002), which incorporates haptic feedback to make the system more realistic. Similar systems are presented by Morris et al. (2004, 2006), again using volume rendering to provide haptic feedback. However, this system employs a hybrid data structure for the rendering, which produces a triangle mesh from the volumetric model data for conventional rendering. The vertex positions are inferred from the volumetric data, using pre-calculated normal and texture co-ordinate information. Cutting is only performed on the volume data, with the polygon mesh being regenerated from the resulting voxels. Indeed, as the cutting performed with this system simulates that of a burr — a type of surgical drillhead — a volume data structure is well suited to this task, as the volumetric nature of the geometry allows arbitrary-shaped volumes to be removed at will. This kind of volumetric system is also employed by Agus et al. (2002, 2003).

2.3.3 Evaluation

Mass-spring soft-tissue simulations and voxel-based bone surgery simulations both contrast with the kind of surgical cutting performed in podiatric surgery, where an electric bone saw is commonly used. Cutting through bone in this kind of surgery has been described as being analogous to carpentry, with a similar set of tools and skills being required. Cuts performed through bone with an electric saw do not typically result in an elastic deformation of the bone as when cutting through soft tissue — cuts will typically be straight, and remove a narrow channel of bone in front of the saw blade. Therefore, although soft tissue cutting is somewhat relevant to our research, and uses many of the same concepts as we intend to use, the mass-spring elasticity algorithms for deforming soft tissue will not produce a suitable result when simulating bone.

Regarding volumetric approaches, we also do not believe these solutions to be optimal for the kind of cutting we intend to perform. This is partly because of the performance costs and complexity associated with volume rendering, but also because we believe that the cuts we intend to simulate will not be modelled very accurately or efficiently by a volumetric model. There are three reasons for why we believe this to be the case: first, only a very narrow sliver of bone matter will be removed when cutting through a model, so the vast amount of data in a volumetric model would be unused. Second, it will be necessary for part of the volume to be severed, moved and reattached at a different angle. The movement of a severed piece of bone could be simulated within the same volumetric model as the rest of the bone, in which case a large amount of data must be moved around per frame and likely cause a large performance loss. Alternatively, the severed bone could be split off into its own volumetric model, which would be more performant, but would entail running two separate volumetric models in very close proximity, with the possibility of them overlapping. Both of these alternatives lead to additional, undesirable complications. Third, when cutting through a volumetric model at any non-axis-aligned angle, a 'stair-step' of voxels will be left behind by the cut, resulting in a cut whose roughness will depend on the angle of inclination of the cut and the granularity of the voxels. Thus, in order to smoothly cut through a volumetric model, the model must be extremely fine-grained in order to alleviate this issue. Using a volumetric model of uniform density with the required granularity would be inefficient, as cutting operations will only involve a small part of the model, and therefore only relevant areas of the model require this granularity. However, to limit the overhead of increased density in the model, a low-density model could be used initially, and the level of detail in areas affected by cutting be increased as necessary. Voxels near to the local area affected by cutting would be subdivided to increase the density in the relevant area.

2.4 Mesh Cutting

We believe that a solution combining the geometric nature of the soft-tissue simulation with the non-deforming nature of the bone simulation will be the best approach. Our proposed solution would model the patient's foot as a 3D polygon mesh (akin to how the soft-tissue is modelled), and respond to the cutting action of the user by employing geometric methods to remove matter from the mesh under the cutting implement. We have reviewed literature regarding geometric mesh cutting to determine if any commonly-used cutting algorithms are suitable for this task, whilst being mindful that real-time cutting may require a custom solution to be developed.

2.4.1 Mesh Sculpting

One of the simplest methods of altering a 3D polygon mesh is to sculpt its surfaces. Generally speaking, when a mesh is sculpted, some of its constituent vertices are moved based on their proximity to a user-controlled tool in a manner governed by one of several sculpting algorithms. Of particular note, one of the earliest computer sculpting systems allowed the user to control the sculpting operation using a polyhedral tool (Parent, 1977), an idea considered advanced for its time. More impressively, the tool could be modelled using the sculpting software itself to allow the user a finer degree of control. The system we propose will support multiple polygon-mesh-based tools that the user can control, although ours will be predefined meshes based on real-world equivalent surgical tools. (Notably, this system also allowed the user to cut out a section of the sculpted object, using the volume of the tool as a guide. However, this was achieved using constructive solid geometry techniques — which are discussed later — as opposed to sculpting techniques.)

In most early sculpting systems, standard computer controls (such as a keyboard and mouse) were used to control the sculpting tool. However, LeBlanc et al. (1991) proposed a system using the Spaceball 6D human interface device, which allows the user 6 degrees of freedom in their input. This device is used with one hand, and a computer mouse in the other hand, to more accurately perform sculpting. We intend to use a similar, but more modern, device (the Sensable Phantom) to allow the user to control our system.

In terms of the algorithms used to perform sculpting, Parent used 'decay functions' to determine the effect of the user's input on individual vertices, based on proximity to the position of the user's tool. These decay function were refined by later researchers, who also included the use of techniques such as polygon subdivision. This technique was applied to the whole mesh by Allan et al. (1989), and later refined into an adaptive subdivision by Bill (1994), in order to afford the user a finer degree of control over the manipulated surface.

2.4.2 Metaballs

Metaballs are a class of implicit algebraically-described geometries, whose surfaces contain points satisfying one or more functions of the form $F(x, y, z) = 0$. Computer-rendered algebraic geometries of this nature were initially described by Blinn (1982), with what he termed "blobby" modelling. Methods for rendering parameterized surfaces individually were already well-known at this point (for example, see Lane et al., 1980), but Blinn blended multiple surfaces together — in some cases using 4000 separate surfaces for modelling chains of molecules. Lone metaballs will produce a single geometric shape, typically a sphere or ellipsoid, but large groups of metaballs can be used to represent more complex shapes. Visually, when lone metaballs approach each other, the effect on their geometry is similar to that of coalescing water droplets due to the overlapping influence they provide on nearby geometry. Since the surfaces created by metaballs are the result of blended continuous algebraic functions (as opposed to the discrete geometric segments in polygon mesh geometry), they provide a smooth transition between their different functions and as a result produce very smooth, soft-looking structures.

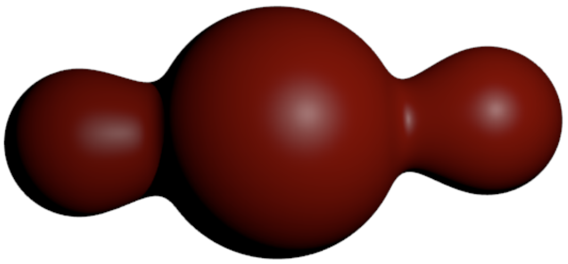

Figure 2.1: Three metaballs

A simple way of using metaballs is to define a set of control points with corresponding radii of influence. The described geometry would then consist of the surface(s) containing all points whose total influence from all control points equals a given threshold value.

For example, with control points $\mathbf{c}_n \in \Re^3$ and control radii r_n, the influence I applied to any point $\mathbf{p} \in \Re^3$ from each metaball could be calculated individually as:

$$I_i = \frac{r_i}{||\mathbf{c}_i - \mathbf{p}||_2}$$

In this case, the surface to render contains all points in Euclidean 3-space where the sum of the influence from all metaballs equals the threshold, i.e.

$$\sum_{i=1}^{n} I_i = T$$

This can be expressed in the form $F(x, y, z) = 0$, where $\begin{bmatrix} x \\ y \\ z \end{bmatrix} \equiv \mathbf{p}$, as follows:

$$\begin{aligned} F(x, y, z) &= -T + \sum_{i=1}^{n} I_i \\ &= -T + \sum_{i=1}^{n} \frac{r_i}{||\mathbf{c}_i - \mathbf{p}||_2} \\ &= -T + \sum_{i=1}^{n} \frac{r_i}{\sqrt{(c_{ix} - x)^2 + (c_{iy} - y)^2 + (c_{iz} - z)^2}} \end{aligned}$$

It should be noted that the influence provided by a given metaball need not be positive. In order to ‘cut through’ a metaball surface, it is possible to define metaballs with negative influence. These negative metaballs create indentations in positive metaball geometry, by reducing the total influence of points within an area below the required threshold. One or more suitably positioned metaballs can be used in this manner to create a soft cut through a metaball surface.

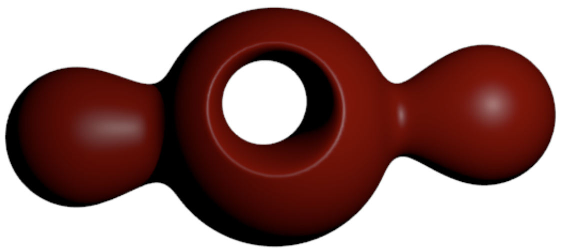

Figure 2.2: Three positive metaballs with a negative metaball

Once a function has been determined that defines the metaballs' implicit surface, there are a number of methods of producing a rendering of the geometry described. The surfaces of the metaballs can be ray-traced by solving their equations algebraically — as in Blinn (1982) and with any of several numerical methods described by Hart (1993) — or a polygon mesh can be generated as an approximation of the metaball's implied surfaces. A polygonal approximation can be generated by progressively subdividing the space enclosed by the implicit surface, for instance by using an octree (Bloomenthal, 1988), and creating polygons based on the implicit surface's intersections with the subdivision structure.

The advantage with having a polygonal approximation of a metaball structure is that modern graphics hardware is specifically designed to quickly and efficiently render polygon meshes, whereas ray-tracing does not currently enjoy such support. However, whilst the output of a metaball system can be a polygon mesh, the input can not. In other words, traditional polyhedral geometries cannot be automatically converted to metaball-based structures. An approximation could be made using numerous tiny metaballs, in a manner analogous to the voxels used in volumetric rendering — however, using this approach would be very inefficient, with an excessively large number of metaballs being required. Using a simple volumetric representation of the polyhedra would likely provide a more efficient solution. As such, in order to use metaballs within an orthopaedic simulation, a model of the patient's foot must first be constructed entirely from metaballs. This would require a modelling expert to be available for the laborious task of remodelling any and all polyhedral meshes desired to be used with the system.

2.4.3 Constructive Solid Geometry

Constructive Solid Geometry (CSG) is a method of defining geometry in terms of two or more other geometries combined using Boolean set operations (Requicha and Tilove, 1978). The three binary set operations used in CSG are union (or addition), difference (or subtraction), and intersection, which are applied to the volume enclosed by the combined geometries. The surface produced as a result of these operations contains the volume previously occupied by either object, only one object, or both objects, respectively. Entire geometries can be represented by a hierarchy of CSG operations performed on implicit primitives and the results of previous operations, forming a binary tree of set operations (Requicha, 1980). The terminal nodes in such a tree represent implicit primitives, and the nonterminal nodes represent a single binary set operation performed on the two child nodes. Geometry represented this way is often used in computer-aided design for manufacturing purposes, as it allows highly accurate smooth surfaces and intersections to be represented, and may also be used with other operations, such as translational and rotational sweeps.

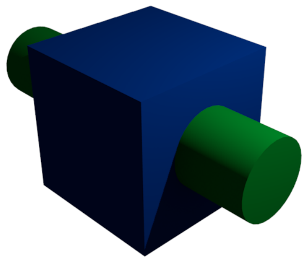
Figure 2.3: Two shapes before a CSG operation

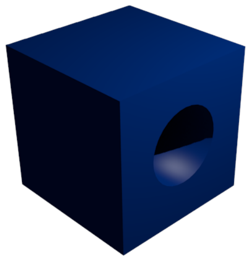
Figure 2.4: The result of a CSG operation

As with other implicit surface representations of geometry, such as the metaball technique discussed earlier, CSG trees may be rendered either by ray-tracing or by polygonization (Jansen, 1987; Wyvill and van Overveld, 1996, 1997). Of these approaches, ray-tracing is slower and must be recomputed for any change in the CSG tree or viewing angle, but produces a smoother and more accurate rendering. Polygonization requires initial overhead to generate the polygon structure (which must be updated with any modifications to the CSG tree), but once generated can be rendered as fast as any other polygon mesh for any viewing angle. A polygonization of the CSG tree will necessarily reduce the level of detail and accuracy in the rendering, but its approximate nature is considered an acceptable tradeoff since it allows the geometry to be rendered at refresh rates suitable for interactive viewing and manipulation of the CSG tree. This tradeoff would be especially important for an application such as surgical simulation, where viewing and modifying the geometry would be required to take place in real-time.

2.4.4 Polyhedral Boolean Set Operations

Another disadvantage that CSG trees share with metaballs is that polygon-mesh geometry cannot be easily converted to this format. If a tetrahedralization of the polygon mesh could be generated, then the resulting list of tetrahedrons could be rendered as a CSG tree. However, doing so would be highly inefficient and would likely be unsuitable for real-time rendering. Therefore, reference data of medically-accurate foot models represented as polygon meshes cannot be used as a CSG tree.

However, operations topologically equivalent to the Boolean set operations used in CSG trees may be applied directly to polygon mesh geometry (Aftosmis et al., 1998). When applied to polyhedra, the boundary representation of each object's surface is segmented based on its intersection with the other object. The segments of each object are then combined into a new polygon mesh or discarded, based on their relationship to the other object. Of particular relevance is the Boolean subtraction (difference) operation, which forms a new object containing the volume enclosed by one object but unenclosed by the other. The object to be subtracted from discards all portions of its surfaces which lie within the subtracting object and retains the remainder; whereas the subtracting object discards all portions of its surfaces that lie outside the subtractee object and retains those which lie inside it. This Boolean subtraction operation is commonly available in 3D-modelling software, and has also been used in industrial simulation software — for example, in simulated log sawing (Occena and Tanchoco, 1988).

Boolean subtraction has also been used to simulate soft-tissue cutting in surgical operations. In a system used to simulate operating on gunshot wounds (Delp et al., 1997), the user can manipulate a virtual scalpel to cut through soft tissue. In this case, a 'cutting object' is described by the movement of the user's scalpel through the model, which is then used as an operand in a Boolean subtraction to remove the affected matter. The polygons added to the mesh that form the boundary of the newly-cut geometry are assessed physiologically by the software, to provide feedback to the user regarding the healthiness of the soft tissue now revealed. Although it appears that, with this system, the cut is only performed once over the final path of the blade, a similar system may be adapted to simulate cutting through bone in our surgery simulation. This will require the Boolean subtraction algorithm to be implemented efficently with complex, high-polygon models.

2.5 Conclusions

Serious games provide a safe, cost-effective way of providing information and training in many disciplines. When designed correctly, they engage their users effectively and provide an incentive for the user to learn. Serious games permit the automation of many tasks, such as evaluating users' performance, and therefore allow more efficient use of instructors' time. Users of serious games can learn at their own pace, with education customized to their level. Serious games complement, but cannot completely replace, real-world training.

Several studies have used serious games technology for medical training, some for basic planning and organisation simulations, but others have created virtual surgery training applications. Some have even simulated bone surgery, using a volume-rendering based approach. However, the type of bone surgery commonly simulated uses a surgical burr to 'drill' away bone as opposed to cutting it with a surgical electric saw. Our research intends to simulate a different kind of cutting, which will largely involve making narrow cuts through (and potentially repositioning) simulated bone. The anatomy being simulated in our application will be the skeleton of a foot. For our purposes, we believe representing the foot using polygon meshes will provide benefits over using volumetric data and as such require a suitable algorithm to allow realistic cutting through polygon meshes.

Four distinct methods widely used to manipulate geometry could potentially be used to simulate cutting through bone (mesh morphing, metaballs, CSG trees and non-CSG Boolean subtraction performed on polygon meshes). Mesh morphing, as commonly used, does not provide the necessary control over the deformed polygon mesh to realistically simulate cutting through bone. An adapted version of this method may be more suitable. However, the required modifications would be tantamount to implementing Boolean set operations, over which the system would provide no major benefit. Metaballs would provide a simple method of representing the path of a cut through bone — however, they are computationally expensive to modify at real-time refresh rates, and cannot be used with polygon mesh representations of medically-accurate foot models. CSG trees present similar benefits and drawbacks to metaballs, but allow more accurate, sharply-defined cuts to be performed. The Boolean difference operation, commonly used in CSG trees, may be applied instead to polyhedral mesh-based geometry, with a modified algorithm that operates on the surfaces of the geometry. This operation may prove viable in performing real-time simulation, assuming it can operate on complex models containing large numbers of polygons. An approximation and refinement of this method is discussed in section 4.5.

Chapter 3

Research Methodology

This chapter discusses how we intend to reach the stated goals of our research, and describes the methodology we intend to follow.

3.1 Introduction

The main focus of this research is to design, develop and test a software simulation of bone surgery. It must be capable of allowing users to cut through a virtual represention of the bones of a simulated patient's foot, in a surgical manner. In the literature reviewed in the previous chapter, four methods of mesh cutting were described. The most suitable method, performing a Boolean subtraction operation on a polyhedral mesh, will be investigated further. It is expected that a conventional implementation of this algorithm will be unsuitable for real-time manipulation of meshes, as high frequency iterations of Boolean subtraction in the same local area of a mesh would recursively create more triangles. Therefore, we will investigate how to achieve a similar end-result to repeated Boolean subtractions along a path, whilst avoiding such high levels of computational complexity and the potential of exponentially increasing the number of triangles in the target mesh.

3.2 Hardware

To perform our research and development work, we will be using desktop PCs equipped with graphics hardware suitable for most contemporary 3D gaming and rendering tasks. We also have the SensAble Phantom Omni 3D input and haptic feedback device available, as well as a Geforce 3D-Vision-enabled monitor and a similarly-equipped projector for large-scale demonstrations. The presence of a 3D input device allows the user to control virtual tools much more accurately than traditional 2D input devices. Additionally, the combination of presenting the output of the system in stereoscopic 3D and providing haptic force feedback enhances the user's perception of the virtual environment in which they operate. These combined technologies allow an application to take real 3D input and produce apparently real 3D output, which can potentially provide a much greater sense of immersion in the virtual world than would be provided by many conventional simulation applications.

3.3 Preparation

As a framework to facilitate the implementation and testing of various 3D algorithms, a simple application will be set up to provide basic user interface and rendering capabilities. This application will allow the user to select one of several scenarios, each of which may have a different 3D model and predefined parameters associated with it. The user will also be able to move, rotate and zoom their view of the virtual world and provide 3D input through the use of the Phantom haptic controller linked to a virtual saw. This application is a supporting part of the research, and not its current focus. However, the application may provide a suitable basis for future work developing a complete virtual surgery simulation.

We are not using a pre-built game or rendering engine, as doing so would add unnecessary extra development complexity in return for a large corpus of unused functionality. We are developing for Microsoft Windows-based systems, and using the DirectX D3D9 rendering API. Since the cutting methods being investigated will primarily be executed on the CPU, and the graphics pipeline will only be used for simple rendering of the results, the specific graphics API used should have little bearing on the project. As the graphics API will merely be used to render a handful of meshes and present a 2D user interface, we chose a platform with which we are familiar enough to develop quickly and efficiently. Algorithm implementations will be done in C++, in an object-oriented manner. The classes required for investigating mesh cutting will be implemented in a static library, which the supporting application will link to. In order to manipulate mesh geometry more easily, an object-oriented representation of the geometry will be stored in addition to the standard hardware-optimised buffers (see section 5.4).

3.4 Development

The previous chapter discusses several mesh manipulation algorithms, the most suitable for our purposes being the Boolean subtraction algorithm for polyhedra. However, subsequent iterations of the algorithm with a randomly-moving subtractor will produce excessive amounts of small, irrelevant triangles, in addition to being highly computationally expensive. The main part of our research will focus on how to achieve a suitable cut though a mesh over multiple frames, with a result similar to that which would be produced by the Boolean subtraction method. We will begin by analysing each of the limiting factors present in the Boolean subtraction algorithm, and comparing them with the desirable properties a suitable cutting method would possess. We will search for methods that will either optimise the Boolean subtraction algorithm, or provide an alternative method of cutting. When a suitable algorithm has been finalised, this will be implemented with the framework application discussed earlier and developed to a usable state. Assuming a viable algorithm can be implemented, it will be assessed for speed, cutting accuracy and visual realism. This will include gathering metrics on the algorithm's performance on multiple test machines with multiple test meshes.

3.5 Testing Vertex and Index Buffer Modifications

One challenge to modifying meshes in real-time is updating the version of the geometry given to the graphics pipeline. Updating the index and vertex buffers containing the hardware-preferred representation of the geometry of the mesh can introduce a non-trivial delay. Performing large numbers of updates to the respective buffers per refresh cycle can create a performance bottleneck severe enough to slow down the application. In order to counteract this bottleneck, we discuss optimizations to the use of the respective buffers in section 5.3.1. To gauge the impact of these optimizations, we will run a series of benchmarks with an automated process set up to repeatably modify a mesh, and compare the performance of the process both with and without these optimizations.

The operations being optimised are appending and removing vertices; and appending and removing indices. Our benchmarks will record the time required to perform each of these operations on large numbers of vertices, both with and without the optimisations discussed in section 5.3.1. We will perform the benchmarks on several meshes, recording the time taken to update the buffers for each operation. For the unoptimised, naïve version of operation, a new buffer will be allocated and data will be copied in from the old buffer. This will be compared and contrasted with the optimised version presented in section 5.3.1.

The optimization for appending vertices or indices to an existing buffer requires buffers to have extra space preallocated, to allow for future expansion. We will test the performance difference obtained when using different allocation schemes of adding between 1% and 20% extra, in increments of 1%. As a control group, we will use the unoptimised method of not preallocating any extra space. We will test each method by repeatedly adding triangles over several rendered frames and recording both the total time taken, and the time taken to add the additional triangles. In order to simulate the adding of extra triangles, we will use two processes — first, we will use a cutting algorithm and simulate cuts through different parts of each mesh. Second, we will simply progressively subdivide the mesh over a long series of frames, by subdividing 0.1% of the mesh's original triangles each frame.

The optimizations for removing indices and vertices do not involve preallocation, but simply a more efficient algorithm than regenerating the buffers. The performance improvement from both of these algorithms will also be tested, by simulating the removal of triangles from a mesh. In order to do so, we will set up a test environment similar to that mentioned above, except that this time a proportion of the mesh's triangles will be removed per frame.

3.6 Testing Cutting Algorithms

The key metric for determining the cutting algorithm's performance is the refresh rate at which it operates. The algorithm's refresh rate will directly affect the overall frame-rate of the application which employs it. In order for an application to be suitable for interactive use, it must maintain a frame-rate averaging above 20 frames per second. Spikes and troughs in the frame-rate of the application are to be expected, as the computational intensity of the algorithm will vary with the precise angle of cut being made, the number of polygons being cut and other factors such as the current viewing angle for the mesh. Since the application is running in a shared environment with other processes on the machine, factors not directly controlled by the algorithm may significantly contribute to the frame-rate.

As the particular hardware capabilities of the machine will have an enormous impact on an application's frame-rate, we will test the final algorithm on two test machines. One machine will be significantly more powerful than the other, and we will compare the application's performance on each. We will run the automated cutting process described above on each machine, and measure the approximate processing time for each frame in order to analyse performance more thoroughly than simply taking the average value. Of particular importance are the minimum and maximum frame times, since these give an indication of the fastest and slowest frame-rates to expect. Additionally, the mean and standard deviation of each run will be calculated, to determine the variance of the frame-rate. It may also prove useful to inspect graphs of the frame times, to detect any patterns that may be present.

Chapter 4

Mesh Cutting Operations

Performing interactive cutting through a mesh requires a cutting algorithm which can operate at real-time refresh rates. It must do so within the constraints of the limited resources available to an application each frame after normal processing and rendering is taken into account. The purpose of this chapter is to briefly evaluate the suitability of several conventional methods of cutting meshes to our system, and to develop an algorithm suitable for real-time cutting.

4.1 Introduction

There are several methods of removing a given volume from geometry representing a solid object. Some common geometric operations that can remove part of the volume of a mesh include: mesh sculpting, use of negative metaballs, constructive solid geometry trees, and polyhedral Boolean subtraction. Each of these methods has a number of advantages and disadvantages, and a varying degree of complexity to implement. Although geometry represented by some of these methods can be ray-traced for rendering, all methods either operate on polygon meshes or can generate approximate meshes. Polygon meshes are the preferred representation of geometry by conventional graphics hardware for real-time rendering.

When meshes are modified or generated, care must be taken to ensure the user does not see inside the mesh, otherwise its hollow nature is revealed and the illusion of solidity is lost. The simplest way to maintain this illusion is to ensure the mesh is a closed surface — this requires any holes cut into a mesh to be sealed with additional polygons. These extra polygons would generally demarcate the boundary of the volume removed, which must not intersect the outer faces of the mesh and thus protrude through the opposite site of the mesh in an unrealistic manner.

One additional point that should be noted is that whenever geometry is constructed automatically care must be taken to avoid generating overlapping coplanar polygons. When coplanar polygons are rendered in real-time such that they intersect, a visual distortion known as z-fighting occurs (Pool et al., 2009). The result of this effect is that the overlapping area inhabited by two or more coplanar (or nearly co-planar) polygons will be rendered with fragments arbitrarily chosen from the competing polygons, resulting in semi-random noise. The z-fighting is due to the overlapping fragments having the same depth values, and since they are rendered in parallel, a race condition occurs between the competing fragments. It can be avoided simply by ensuring coplanar polygons do not overlap.

4.2 Overview of Conventional Methods

4.2.1 Sculpting

Sculpting, the simplest of the above operations, does not primarily add or remove any polygons, but instead moves existing vertices in a manner defined by the given sculpting algorithm (see section 2.4.1). Affected polygons are usually subdivided when being sculpted. This increases the resolution of the sculpting operation by increasing the density of vertices in the affected area, and as a result produces a much smoother deformation of the mesh.

The actual deformation caused by a sculpt operation is determined by the sculpting algorithm used and the shape of the 'tool' used to sculpt with. For instance, a simple algorithm is to repel all vertices within a given radius of the tool's origin, pushing them along the vector between the tool's origin and the vertex position. The result of this example operation is a spherical indentation forming around the tool. To increase the smoothness of the scuplt operation, any polygons that are sufficiently stretched will be subdivided to increase the resolution of the deformation. Stretching of polygons can be determined by comparing their current surface area to their original surface area, and an increase exceeding a given percentage of area will cause the polygon to be subdivided.

Sculpting is very useful when crafting natural-looking shapes, such as animals, plants and human faces, as it allows the user to easily fashion smooth, curved surfaces. However, this is not an advantage for simulating bone surgery, as no such curved surfaces need to be made. Cuts made into bone will generally have a hard, angular edge, and will necessarily protrude all the way through the mesh. Although hard-edged tools can also be simulated using sculpting algorithms, it must be ensured that any cut made with such tools never adds matter to the target mesh. In order to satisfy this requirement, the geometry repelled by the saw blade must be carefully controlled to ensure that it never protrudes through the surfaces of the mesh, but instead progressively forms holes whenever one surface intersects another. To correctly sculpt the mesh geometry in this manner, control algorithms will be required to perform geometric operations substantially more complex than any basic sculpting algorithm. Correctly and progressively forming holes around intersecting surfaces in itself is more similar in nature to constructive solid geometry operations, and a system capable of performing this kind of cut would retain only a passing resemblance to common sculpting software. So although sculpting is a fast operation, and relatively straightforward to implement in its standard forms, in order to be useful the process would have to be substantially modified with complex constructive solid geometry techniques. Therefore, it seems unwise to start with sculpting as the basis of the cutting algorithm.

4.2.2 Metaballs

Metaballs are a system of storing and representing an object's volume data without storing its exact surface geometry (Blinn, 1982). Typically, each metaball has an origin and radius of influence. Based on this, a level of influence is calculated for any given point in 3D space based on the inverse square of its distance from the metaball's origin, and the metaball's radius. Whether a given point is part of the volume represented by the metaballs is determined by whether the sum of all influences from all metaballs in the scene is greater than a given threshold value. When rendered, a single metaball will visualise as a perfect sphere, as will any metaballs of sufficient distance from one-another (relative to their radii). However, when two or more metaballs come into close proximity to each other, the sum of the level of influence for some points between the metaballs that would otherwise not be rendered will become greater than the threshold value, leading to the surface each metaball warping towards the other. Visually, the effect of two metaballs approaching each other resembles water droplets coalescing.

Metaballs can exert negative influence as well as positive influence. The effect of a positive metaball (visualising as a perfect sphere) approaching a negative metaball is that the sphere will become 'indented' in the area affected by the negative metaball. The negative metaball appears to exert a repelling force on the surface of the sphere. A line of small, negative metaballs passing through a large, positive metaball would have the appearance of a hole being drilled through a sphere. More complicated shapes can be cut out of the volume created by positive metaballs by using larger number of small negative metaballs. The shapes created by metaballs always have rounded edges, due to the spherical nature of all the components used to create them. In order to remove a more precise volume from a shape, constructive solid geometry operations can be used — specifically the Boolean subtraction (or difference) operation.

A major caveat of using metaballs, as discussed in 2.4.2, is that the entire structure being manipulated must be created using metaballs. This means that standard anatomical polygon mesh models cannot be used, unless manually converted to a metaball-based approximation by a skilled 3D modeller. Additionally, rendering metaballs directly though ray-tracing is typically more computationally expensive than rendering polygon meshes. Although a polygon mesh can be generated from metaball geometry, the mesh must be regenerated whenever the metaball geometry changes. It is unlikely that this can be achieved at real-time refresh rates.

4.2.3 CSG trees

Another system of storing and displaying geometry through means of implicit surfaces is to use constructive solid geometry (CSG) trees. A CSG tree is constructed from terminal nodes (which represent implicit solids) and non-terminal nodes (which represent operations carried out on the child nodes of the given non-terminal node). The three operations used in CSG trees are set operations — union, difference and intersection. The results of performing these operations can be used as operands in further operations. In this way, it is possible to build up a tree of operations, adding extra primitives where necessary (Requicha, 1980).

CSG trees can be rendered using ray-tracing, where each pixel casts a hypothetical ray into the scene and determines if and at what distance the ray will intersect the geometry resulting from the tree of CSG operations. They may also be polygonized for more efficient rendering by conventional graphics hardware.

The major drawback to using CSG trees to simulate bone cutting is similar to that of using metaballs — the entire geometry of the CSG tree must be represented using CSG operations and implicit primitives. There is no easy way to automatically convert an arbitrary polygon mesh to a CSG tree representation, so this again would require a skilled modeller to laboriously create a CSG approximation of any reference foot meshes. However, each of the Boolean set operations used in CSG modelling can be applied to polyhedral meshes using geometric methods instead of implicit surface methods. We believe these geometric Boolean set operations may be useful for simulating cutting, especially Boolean difference (or subtraction).

4.2.4 Boolean subtraction

Boolean subtraction takes two intersecting polygon meshes and removes the volume enclosed by one from the volume enclosed by the other (Aftosmis et al., 1998). Let A be the mesh to be subtracted from, and B be the mesh to subtract. Both meshes form closed surfaces that each enclose a volume. After the subtraction operation concludes, only the portion of A's surface that lies outside of B remains and, conversely, only the portion of B's surface that lies within A remains. The result is that the remaining parts of each surface are joined together along the curve of intersection to form a new mesh. The closed surface of the new mesh encloses the entire volume of A, minus any volume previously occupied by B.

For both meshes, a polygon's presence or absence in the final mesh depends on the polygon's relation to the other mesh. Polygons are either retained intact, removed, or have a portion of their area retained and a portion removed. For mesh A, any polygons lying entirely outside of B are kept; any polygons lying entirely inside of B are removed; and any polygons lying partially inside B must be split along the curve of intersection between the two meshes. For mesh B the reverse applies — polygons outside of A are removed, polygons inside A are kept, and the rest lie on the curve of intersection so must be split.

Polygons which are to be split are intersected with the opposing mesh, and are tested against all intersecting polygons. For each polygon to split, a list of line segments lying on the surface of the triangle can be generated which denote the curve(s) of intersection between the splitting polygon and the intersecting polygons from the opposing mesh. When intersecting with a closed mesh, all polygons will be adjacent to other polygons on all sides, so the line segments generated will always form a closed path when combined with any segments of the splitting triangle's edges that lie within the other mesh. From these closed paths, a new set of polygons can be generated that cover the area of the splitting triangle either enclosed or excluded by the paths. These polygons are then added to the list of retained polygons for the final output of the operation.

The resultant mesh is created from the polygons retained from A and B and those generated by splitting partially retained polygons.

4.2.5 Conclusion

The mesh sculpting method cannot feasibly be used to simulate bone surgery, due to the lack of realism that would be apparent. Modifications to the sculpting algorithm could be made to compensate for this, but the end result would essentially be Boolean subtraction. Using metaballs would have some of the same drawbacks as sculpting, with both being more suited to defining rounded edges. However, the major caveat of using metaballs is the inability to easily convert a mesh with a large number of polygons to a metaball-based structure. The Boolean subtraction method should provide accurate cutting simulation. However, it is not designed to be employed repeatedly on a mesh at high refresh-rates. The following section describes some of the limitations that cause this.

4.3 Disadvantages of Boolean Subtraction

4.3.1 Geometry-in-polyhedron Testing

In order to compute the result of a Boolean subtraction operation, it is necessary to know which polygons in each lie wholly or partially inside or outside the other mesh. Therefore, it is generally necessary to test all the polygons in each mesh to determine which lie entirely outside of the other mesh, which lie entirely inside of the other mesh, and those that lie partially inside and partially outside the other. To test for which of these conditions applies to each polygon, it is usually only necessary to determine whether each of its vertices lie inside or outside the other mesh. This test can be performed in a number of ways — the easiest being the ray-cast method.

Ray-casting for the purpose of collision detection calculates the intersection of a hypothetical 'ray' with object geometry. In order to determine whether a given point lies within a given triangular mesh or not using ray-casting, rays must be emitted from the point, and the number of triangles the ray intersects counted. A simple test with a closed mesh can be performed using only two rays, both emitted in opposite directions — if both rays pass through an odd number of surfaces, then the point is enclosed on both sides. Where the number of surfaces is even or zero, the point is outside the mesh.

All polygons in each mesh must be tested to determine whether they intersect the other mesh, which entails testing all vertices in each mesh. Assuming a suitable spatial subdivision technique is used to optimise the ray test, each test will be of O(1) complexity in the average case, and O(n) in the worst case (Szirmay-Kalos and Márton, 1998). Since all vertices must be tested per Boolean operation, this results in an overall complexity of O(n) (average case) or O(n^2) (worst case) for the intersection tests.

The remainder of the algorithm will require classifying all polygons based on their intersection or non-intersection of the opposing mesh (O(n)), adding each new vertex or index (reallocating buffers each time would be O(n^2); with optimised buffers this is closer to O(n)) and removing deleted vertices and indices (O(n)).

4.3.2 Segmentation of Triangles

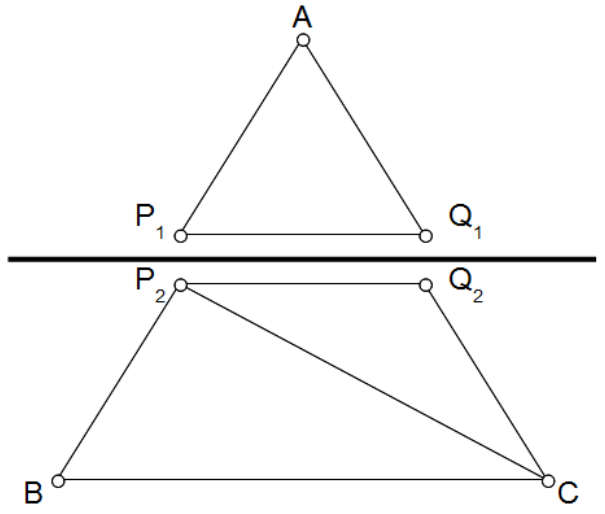

Figure 4.1: Triangle split into three.

When two irregular polyhedral meshes intersect, the intersection will occur along one or more curves of intersection. As the meshes in our system are comprised of triangles, each curve of intersection will consist of a set of contiguous line segments. The total set of line segments is the lines of intersection between triangles in the two intersecting meshes, and the curves of intersection are cycles formed by these line segments. In order to segment the polyhedra for combination in the relevant Boolean operation, any of their constituent triangles containing part of a curve of intersection must be split along it. In the best case of segmenting a triangle like this, 3 triangles must be added for each triangle removed. However, for triangles containing multiple segments of a curve of intersection, a minimum of five additional triangles are added for each triangle removed. Triangles containing two opposing edges of a curve of intersection must be separated in a more complex manner; triangles containing multiple curves even more so.

These operations do not present a significant time penalty when the operation is to be carried out once, even for large meshes. However, for our purposes, the cutting operation must be carried out and visualized once per frame. This requires the Boolean subtraction to take a maximum of 1/25 seconds, or 40ms. Under the circumstances the operation will be used for — few polygons will lie inside the cutting volume, many more will lie on the boundary and require splitting. Unless both polyhedra have a similar density of vertices around the intersecting area, the segmentation will generate a large number of small triangle fragments — there will be a constant net increase in polygons. As more polygons are generated, more polygons will be split per frame, potentially resulting in an exponential growth in the number of polygons in the target mesh. This leads to a constant slowdown of the cutting operation.

The problem of exponentially-increasing polygon count can be mitigated somewhat by mesh simplification. There are several techniques that reduce the number of polygons in a mesh whilst retaining an approximation of its original shape (for instance, see Turk, 1992). However, simplification algorithms typically operate on an entire mesh and this adds to the processing time required (Kalvin and Taylor, 2002). Assuming a suitable algorithm could be implemented that efficiently simplified only the geometry modified by the Boolean operation each frame, it must keep the number of polygons in the mesh largely constant over several iterations of the cutting algorithms.

4.4 Simplification of Serial Boolean Operations

4.4.1 Cutting Cross-sections

Consider the case where one or a series of Boolean subtractions (using a convex polyhedral cutting volume) cut through the complete cross-section of a convex triangle mesh. This will sever one part of the mesh from another, to form two new meshes. Once the cut is complete, all the triangles in the original mesh which are outside of the cutting volume will have been separated into two groups — one group on each side of the cutting volume. Triangles intersecting the surface of the cutting volume will have been split accordingly, and the resulting triangle fragments will have been placed into the appropriate group given their position. The newly-hewn sides of the two new meshes (which previously intersected the cutting volume) are defined by two surfaces. These surfaces are bounded by the curves of intersection formed between the sides of the cutting volume and the mesh.

An equivalent operation to the series of Boolean subtractions would be to obtain these two surfaces, and separate triangles into groups based on which side of the surfaces they lie on. The geometry that lies between the two cutting surfaces would be discarded, and the rest would be added to whichever of the two newly-created meshes is appropriate, based on its position relative to the surfaces. The geometry of each cutting surface would be appended to the new mesh adjacent to it.

4.4.2 Biplanar Subtraction

When cutting through bone (and for that matter, most hard surfaces over short distances), the path of the saw will typically follow a plane with a normal vector matching that of the saw blade. For our purposes in simulating bone surgery we shall assume that, for each cut made, the blade movement becomes limited to the plane defined by its position and normal when it first touches the bone. This implies that the saw blade cannot move along its normal vector due to matter either side of the blade, and the only axis it may rotate around is defined by its normal vector. Within these constraints, the series of Boolean subtraction operations described previously will remove a volume bounded by two surfaces lying on planes parallel to the cutting plane, separated by the width of the simulated blade.

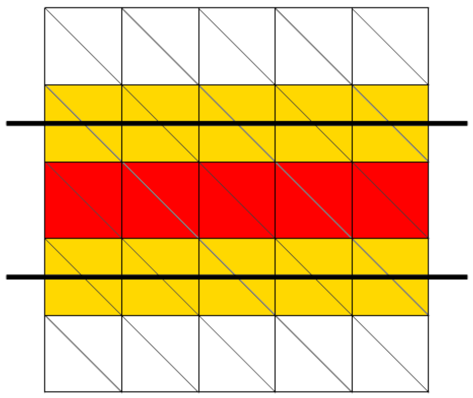
Figure 4.2: Before biplanar subtraction. Red triangles are removed; yellow triangles are sliced.

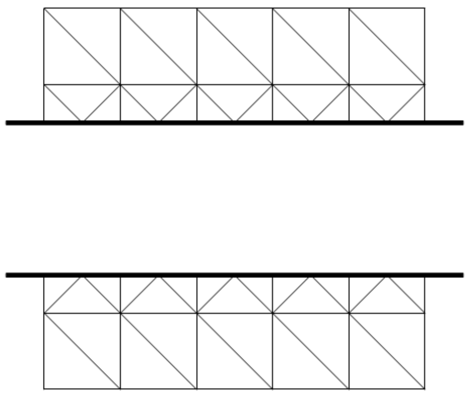
Figure 4.3: After biplanar subtraction

Using this assumption — that the removed volume will be bounded by two planes lying parallel to the cutting plane — the process of separating the convex mesh geometry to produce the end-result of a complete cut can be greatly simplified. The mesh can be divided into two new meshes, with the geometry between the two planes being removed and the rest being divided into one of the new meshes, based on its position relative to the cutting planes.

Slicing a triangle mesh along a plane is much less complex than intersecting it with a polyhedral object. Individual triangles can be determined to lie on one side of a plane or the other simply by comparing the side of the plane on which their vertices lie. Determining this for an individual vertex can be done simply by calculating its signed distance from the plane, relative to the plane's normal vector. If all of a triangle's vertices lie on the same side of a plane, that plane does not intersect with the triangle. However, if one of the triangle's vertices lies on a different side of a plane to the other two, the triangle must be divided along that plane to form three triangles, two of which will lie on one side of the plane, whereas the third triangle will lie on the other side. With the biplanar slicing method described above, this test is repeated for both planes. With large triangles that intersect both planes, there may be up to seven new triangles formed.

In order to categorise triangles using the biplanar slice method, a vector which is normal to both planes is used to arbitrarily define whether vertices lie 'above' or 'below' each plane. As the planes lie a distance apart, one plane is above the other as defined by this normal vector. After triangles have been sliced by both planes, those that lie between the two planes are discarded. Of the remaining triangles, those that lie above the topmost plane are collected into one mesh, whereas those that lie below the bottom plane are collected into the other. Finally, the vertices that lie on each cutting plane are used to create a 'cap' that seals the hole made by slicing the mesh into two. For slices with a convex cross-section, this can simply be done by creating a triangle 'fan' linking all the affected vertices.

4.4.3 Single Plane Slicing

The caveat with using the biplanar slice method is, of course, that as soon as it is applied, it appears that the user has sliced through the entire mesh. In order to overcome this, there are two possible approaches. After the biplanar slice is applied, the new geometry could be extruded to meet in the middle (along the plane defined by the constrained cutting motion of the saw), and as the blade passes near to the vertices involved, they can be returned to their original (separated) locations to create the illusion of slowly separating the mesh. However, for any triangles not exactly perpendicular to the cutting plane, this will cause a visual modification to the mesh at the instant that the biplanar method is applied, which is undesirable.

An alternative method is to perform the slice using only one plane (see figures 4.4, 4.5). All the triangles in the mesh lying on the slice plane will have been split along the plane, and no geometry will have been removed. (Unsliced triangles lying on the cutting plane will be removed, but the triangle fragments formed by slicing each of these along the plane will fill the exact same area that these previously filled, hence no geometry is lost). This planar partitioning process is used in binary space partitioning (BSP trees), where it is commonly used to divide a 3D scene graph into segments (Fuchs et al., 1980). A scene divided into a BSP tree can be used for more efficient rendering and collision detection, and BSP trees can be merged to perform CSG operations on the polyhedra they represent. In BSP trees, segements created by slicing do not have geometry added to create a visible partition - however, this is required for our purposes.

In our method, the vertices lying on the slice plane will be used to form two cross-section caps, one for each mesh, effectively creating two barriers across the partition sliced. At the instant that a uniplanar slice is performed, there will be no visible difference in the mesh, despite it being separated into two discrete entities. However, as the blade of the saw passes near to each pair of the coexisting vertices in the sliced cross-section, they can be moved apart to create a gap in the mesh. This is consistent with the mesh being separated only as the blade of the saw passes through the geometry.

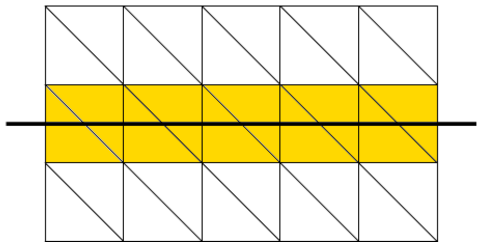
Figure 4.4: Before single plane slice. Yellow triangles are sliced.

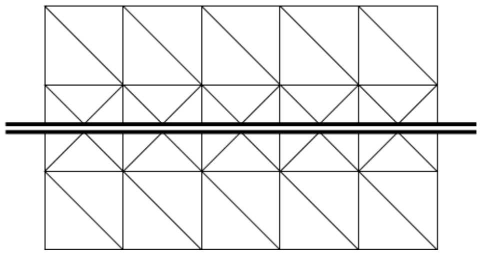
Figure 4.5: After single plane slice.

At any point whilst performing a series of Boolean subtractions on a mesh based on the position of the saw blade, all volume occupied by the blade throughout the previous series of iterations of the Boolean subtraction algorithm will have been removed from the initial mesh. Performing a uniplanar slice and separating the vertices as described above does not guarantee that this holds true, as the exact geometry of the blade is not removed per frame. However, since the geometry of the rendered saw blade obscures the cut whilst it is being made in the mesh, this inadequacy should not be noticable to the user.

4.4.4 Benefits of Single Plane Slicing

In terms of performance, this method requires an initial overhead of processing time (to compute the planar slice), but in subsequent frames the separation uses a minimal amount of processing time. This should allow a smooth frame-rate throughout the user's cutting operation, enabling real-time cutting to be approximated.

Unlike Boolean subtraction, planar slicing does not require the geometry-in-polyhedron test to determine whether the cutting mesh (in this case, the cutting plane) lies within the target mesh. All the polygons in the target mesh must instead be tested to determine which side of the plane their vertices lie on. Although this is an operation of O(n) complexity, it need only be performed once (at the start of a cutting operation) instead of per frame as with Boolean subtraction.

Furthermore, the number of triangles generated by this method cannot rise exponentially with the duration of the cut. The number of extra triangles added is fixed after the initial slice operation, with all subsequent frames in a given cutting operation simply moving vertices. The maximum number of triangles added is limited to $3n$ for the worst-case of all triangles in the mesh lying on the cutting plane. However, in practice, the percentage of polygons in the mesh that intersect a given plane diminishes as the total number of polygons increase. From testing, we see a maximum of around 25% of polygons intersecting the cutting plane for meshes with low numbers of polygons, and around 1% for meshes with higher totals of polygons.

4.4.5 Drawbacks of Single Plane Slicing

With single plane slicing, the accuracy of the cut during intermediate frames is not as accurate as that possible when using Boolean subtraction. When a triangle fan is used to cap the two cross-sections, moving the vertices apart individually has the visual appearance of ‘folding’ the caps apart. However, the inaccuracy is typically only visible between the two cross-sections of the slice, which is difficult to see from a viewpoint outside of the mesh, and the geometry of the saw will tend to obscure these details in any event. Once the cut is complete, these inaccuracies are no longer visible. A more likely cause of visual inaccuracies is if when vertices are moved a given distance from the plane, they share edges with other vertices whose distance to the plane is less than the movement distance. This will cause edges that previously pointed towards the plane to now point away from it (or vice-versa), which in the best case will introduce new concavities into the mesh, and in the worst case will cause the extrusion away from the plane to crease back over other geometry, creating a visible fold around the edges of the slice. One way of mitigating this is to move all vertices within the movement distance from the cutting plane, instead of just the new vertices created along it. This will collapse all nearby vertices and edges onto a plane at the given distance from the cutting plane, eliminating any overlap.

A second drawback is that, although planar slicing works well with meshes consisting of a single convex surface, the method will not always work correctly with meshes which consist of separate parts (such as the toe bones on a foot model) or meshes which have concave areas. This is due to the slicing being performed with an infinite plane, which will cut all the way through any part of the mesh lying on it. For instance, when cutting across the width of a toe bone on a foot mesh, this limitation will cause the slice to cut through all toe bones, not just the one being operated on. Futhermore, cutting across the concavity at the end of a bone will cause both sides of the concavity to be sliced. In order to overcome these limitations, it is necessary to only slice triangles that can be reached from the starting point of the cut by traversing adjacent triangles that lie on the cutting plane. An algorithm for determining these triangles is presented in section 4.5.4.

4.5 Implementation of Single Plane Slicing

4.5.1 Dividing Along A Plane

Although the planar slice operation uses similar triangle intersection and splitting routines, it is somewhat simpler to implement than that of the Boolean subtraction operation. In its simplest form, a mesh is divided along a single infinite plane. The constituent polygons of the mesh are divided into two groups — whether they are 'above' or 'below' the plane relative to its normal. Those lying entirely above or below the plane are simply placed into the relevant group; those lying on the plane are split along it. This usually results in three additional polygons being generated per polygon sliced — two on one side of the plane, and one on the other side.

Whether a triangle intersects the cutting plane can be determined by comparing the signed distances of its vertices from the plane with one-another. If the distances of all three vertices are of the same sign, then the triangle does not intersect the plane and the triangle is placed whole into either group based on the sign. On the other hand, if one of the triangle's vertices' distance has a different sign to the other two, then the triangle does intersect the plane and must be sliced.

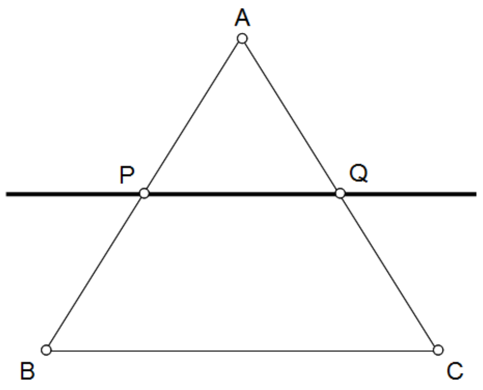

Figure 4.6: Before splitting a triangle along a plane (line of intersection with plane marked as thick line).

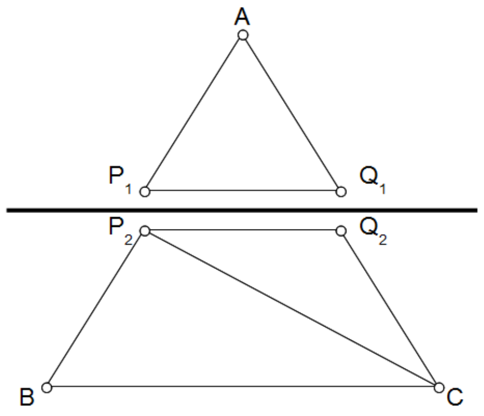

Figure 4.7: After triangle is split.

When slicing a triangle that intersects the cutting plane, the triangle will split into three (see figure 4.7). The solitary vertex on one side of the plane is labelled **A**; the other two **B** and **C**. The points of intersection between **AB** and **AC** are labelled **P** and **Q** respectively. In order to be able to manipulate geometry on both sides of the split independently, the vertices **P** and **Q** must be duplicated to create $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{Q}_1$ and $\mathbf{Q}_2$.

One triangle ($\mathbf{AP}_1\mathbf{Q}_1$) will lie on one side of the plane, whereas the other two will form a convex quadrilateral on the other side ($\mathbf{BCP}_2$ and $\mathbf{CP}_2\mathbf{Q}_2$). Of the original edges, BC is retained to form the base of the quadrilateral. The other two edges are split at their point of intersection with the plane to form four smaller edges — $\mathbf{AP}_1$ and $\mathbf{BP}_2$ derived from $\mathbf{AB}$, and $\mathbf{AQ}_1$ and $\mathbf{CQ}_2$ derived from $\mathbf{AC}$.

4.5.2 Splitting Triangles

Calculating the point of intersection between an edge and a plane gives a distance along the edge, which can be normalized by dividing it by the total length of the edge. For edge $\mathbf{pq}$ where $\mathbf{p}, \mathbf{q} \in \Re^3$, the distance from vertex $\mathbf{p}$ to the point of intersection with a plane in the form

$$Ax + By + Cz + D = 0$$

is calculated with the formula:

$$d = \frac{A(q_x - p_x) + B(q_y - p_y) + C(q_z - p_z) + D}{\sqrt{A^2 + B^2 + C^2}}$$

The normalized distance can then be obtained thus:

$$d_{norm} = \frac{d}{||\mathbf{q} - \mathbf{p}||}$$

This normalized distance can be used to produce a new vertex that lies on the plane. All the properties of the vertices at the opposing ends of the edge are linearly interpolated to produce the new blended vertex. As well as their positions, these properties will include all fields in the vertex format, including their normal vectors, colours, texture coordinates, tangents, binormals, etc.

Since the edge is to be divided into non-adjacent separated polygons, a copy of the interpolated vertex must be created. One of the vertices is associated with the edges and triangles on one side of the plane and the other vertex is associated with the edges and triangles on the opposite side of the plane.

When both intersecting edges have obtained a pair of interpolated vertices, three new triangles can be formed. On the side of the plane containing only one of the triangle's vertices, a single new triangle is formed with the lone vertex and the two new vertices. On the opposite side of the plane, two triangles are necessary to form the base of the severed triangle — one takes the two pre-existing vertices along with a new vertex, whereas the other takes both new vertices and the pre-existing vertex adjacent to the new vertex that was not previously selected.

In order to avoid unnecessary duplication of the vertex pairs, each pair of new vertices can be cached in an associative container that associates the pair with the edge they were formed from. This not only increases data reuse, but also ensures that traversing the mesh's geometry still functions correctly.

4.5.3 Capping

After the mesh has been split along the plane, the new edges created by splitting polygons should describe one or more closed paths on the surface of the plane. The area enclosed by these paths is then efficiently filled with non-overlapping co-planar polygons to form a cap. Such caps are necessary to maintain the illusion of solidity of the object, and are formed in pairs. A minimum number of polygons should be used to create the cap, as a large increase to the polygon count is undesirable.

For cuts with convex cross-sections, a simple method of capping the sides of the cut is to create a triangle fan. A new vertex is added at the midpoint of the cross-section, which allows the original vertices to be sorted according to their bearing from the midpoint vertex. The vertex is created from a blend of the properties of all vertices in the cross-section, as described above. Once the vertices have been sorted, each vertex contributes a triangle to the cap. Each vertex's triangle is comprised of the current vertex, the vertex following it in the list, and the midpoint. In order to ensure the fan seals the cap all the way round, the last vertex wraps around to use the first vertex as its following vertex.

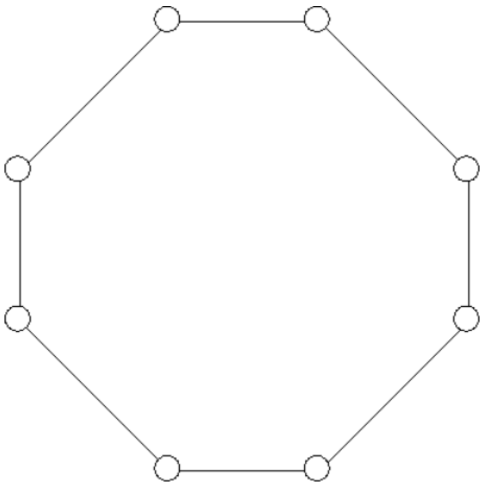
Figure 4.8: Cross-section vertices before capping.

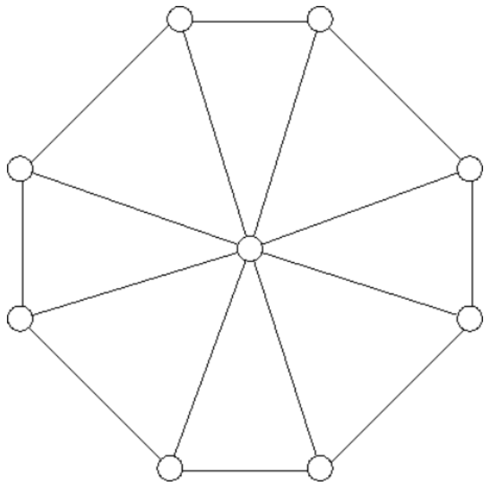
Figure 4.9: After capping.

Although the above method will also correctly triangulate some convex cross-sections, a more suitable triangulation algorithm is required to work in the general case with more complicated convex cross-sections. One such algorithm is the ear-clipping method (Eberly, 1998), which recursively finds 'ears' of the cross-section polygon. An ear in this context is defined as three consecutive connected vertices that form a triangle such that no other vertices lie within the triangle, and the edge joining the first and last vertices forms a diagonal of the cross-section. Each iteration of the algorithm finds an ear, adds it to the final triangulation and removes it from the initial polygon, until there is only one triangle left. The algorithm can also cope with holes in the cross-section, although we do not expect to need this capability.

4.5.4 Localised Slicing

The disadvantage of performing the planar slice operation on all polygons comprising a mesh is that it will make a cut all the way through a mesh, separating any polygons that lie on the cutting plane. With convex meshes this will be desirable, but for our purposes of simulating surgery with concave geometry, it is necessary to cut through only the parts of the mesh which would be affected by the surgical tool. For example, when cutting into the side of a toe bone, the plane on which the cut is performed will almost always intersect the other toes, so an unbounded slice along this plane would cut matter not relevant to the operation. A localised version of the planar slice must therefore be used, by limiting the triangles on which the operation is performed.

Once a physical blade begins a cut on bone in real life, it naturally follows approximately the same plane due to pressure from matter either side of the blade. Therefore, when simulating this cut, all the relevant polygons will intersect with the plane whose normal vector matches that of the blade as it first begins the cutting operation. Additionally, once the blade cuts entirely through a section of matter, the cut is finished and any further cutting will be considered a separate operation. Therefore, the only relevant triangles are those whose lines of intersection with the cutting plane form a contiguous cross-section, starting with the triangle first intersected by the blade as it begins the cut.

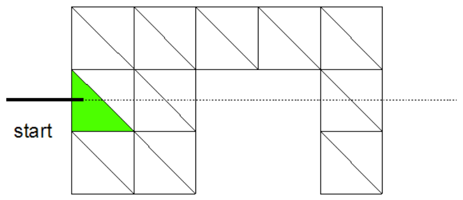

Figure 4.10: Initial traversal. The green triangle is marked as to be sliced.

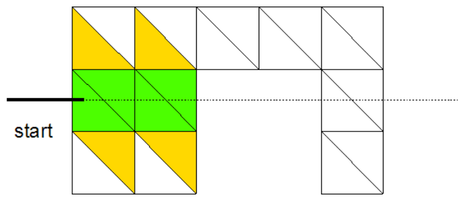

Figure 4.11: Final traversal. The yellow triangles are traversed to, but not sliced.

In order to determine which polygons are relevant to a given operation, it is necessary to traverse the mesh from a given starting point. In our case, this will be the point at which the surgical tool first touches the mesh. The polygons forming a contiguous border around the cross-section containing the starting point can then be determined by traversing adjacent polygons that lie on the cutting plane. A starting polygon is chosen, and marked as visited and added to a stack to test. This pseudocode fragment describes the algorithm:

```
Mark starting triangle as visited
Add starting triangle to the stack
While stack is not empty:
  Pop triangle T off the stack
  If T intersects the cutting plane:
    Add T to the list of relevant triangles
    For each triangle N which is a neighbour of T:
      If N is not marked as visited:
        Mark N as visited
        Add N to the stack
```

Operations to determine if a triangle intersects a given plane, or if a triangle is adjacent to another given triangle can be written as methods within the Triangle class, as described in the previous chapter.

In order to determine whether a triangle intersects a plane, it is determined whether each of its vertices lie above, on, or below the plane. Evidently, a triangle whose vertices do not all lie above or below the plane must intersect the plane in question. To determine the position of each vertex relative to a plane, the signed distance from the plane to the point along the plane's normal is calculated. A positive distance is in the direction of the plane's normal (arbitrarily referred to hereafter as 'above' the plane) and a negative distance is in the opposite direction ('below' the plane). A distance of 0 indicates that the point lies exactly on the plane. The signed distance is calculated for a plane described in the form

$$Ax + By + Cz + D = 0$$

and a vector $\mathbf{v}$ using the following point-plane distance equation:

$$d = \frac{Av_x + Bv_y + Cv_z + D}{\sqrt{A^2 + B^2 + C^2}}$$

where d is the signed distance. Since the magnitude of the signed distance is irrelevant apart from its sign, this can be simplified to:

$$d = \text{Sign}(Av_x + Bv_y + Cv_z + D)$$

where

$$\text{Sign}(x) = \left\{ \begin{array}{ccc} 1 & \text{if} & x > 0 \\ -1 & \text{if} & x < 0 \\ 0 & \text{if} & x = 0 \end{array} \right\}$$

If the signed distances for each of the vertices do not all have the same sign, or any of the signed distances equal 0, then the triangle intersects the cutting plane.

Once a list of polygons has been obtained that form a contiguous boundary around the maximum volume of the mesh to be cut, the planar slice algorithm can be applied to them. The effect is that the infinite plane slice is carried out on only a small part of the mesh.

4.6 Simulating Cutting

The result of a slice being carried out is a boundary being drawn within the mesh with the two caps, invisible to the user from the outside of the mesh (the normal circumstances for viewing a mesh). The process of creating a visible cut in the mesh then simply consists of moving the coplanar caps apart. If all vertices comprising the caps are moved simultaneously, the two sides of the mesh are separated instantly.

However, it is also possible to separate the vertices individually — this can cause a gap to slowly form in the mesh over several rendering frames. This effect is most realistic when the position of the cutting blade (controlled by the user) is used to determine which vertices are separated. As with a real-world cut, only areas that the blade has touched are modified. In order for a user to completely sever part of the mesh, they must cut through its entire cross-section.

After the cut has completed, the mesh will still be represented internally as a single set of polygons, regardless of whether the polygons form a single closed surface or otherwise. If the result of the cut has severed part of the mesh from the rest, the separated part can not be manipulated separately unless it is treated as a separate mesh. For a number of orthopaedic surgical procedures in podiatry, it is necessary to cut off part of a bone, reposition the severed piece, and then reattach it to the main bone through the use of pins and / or wires. Therefore, after a cut has completed, it is necessary to ascertain whether the mesh still forms a single closed surface and, if not, to divide its polygons into separate meshes.

The method of traversing a mesh by visiting adjacent polygons can be used to determine if all the polygons in a mesh still form a single contiguous surface. Any polygon in a contiguous surface can be reached by traversing adjacent polygons from any other starting polygon in the surface. To test whether this is true for a given mesh, the following algorithm can be used:

```
Initialise working_stack as an empty stack
Initialise unvisited_set with all polygons in the mesh

While unvisited_set is non-empty:
  Initialise visited_set as the empty set
  Pick random polygon P in unvisited_set
  Move P from unvisited_set to visited_set
  Push P to working_stack

  While working_stack is non-empty:
    Pop polygon T off working_stack
    For each polygon A adjacent to T:
      If A is in unvisited_set:
        Move A from unvisited_set to visited_set
        Push A to working_stack

  If unvisited_set is non-empty:
    Create a new mesh from visited_set
```

```
Else:
  If this is the first iteration:
   No severed meshes have been created; stop
  Else:
    Create a new mesh from visited_set
```

If the mesh has been separated into two or more disjoint meshes, the preceding algorithm will generate a list of polygons in each of these disjoint sub-meshes. Each list can then be converted into a new mesh, which can be treated as a separate object in the virtual world. The separate meshes can then be manipulated individually — for instance, they can be moved independently of one-another.

Once the mesh has been separated into two or more lists of polygons, all but the original mesh will need to generate vertex and index buffers for their respective polygons. The original mesh's buffers will still contain data for the removed polygons, which in this case may be numerous — potentially consisting of a large proportion of the original mesh. Regenerating the buffers in a single frame is generally undesirable when dealing with large numbers of polygons, as this may result in late-rendered frames and perceived interface lag. However, since the visible structure of the mesh is not being modified (merely its internal representation), the reallocation of buffers can occur over several frames. During this time, the user will be unable to perform another cut, but the simulation will still respond to their other input. The total duration of the regeneration process should be a few tenths of a second and occur immediately after the user has finished a cut, therefore the delay in starting a new cut should not be noticeable.

4.7 Conclusions

Of the conventional geometry cutting methods described (morphing, metaballs, CSG trees and Boolean subtraction), the most suitable method to simulate bone cutting is Boolean subtraction operating on polygonal meshes. However, although Boolean subtraction provides an accurate cut, it is not suitable for use in real-time cutting.

Therefore, using assumptions of how the cutting will be used to perform surgery, we proposed a simplification of the Boolean process that obtains the same result by slicing mesh geometry with two planes. A refinement of this planar slice technique using one plane allows the mesh to be cut without undergoing immediate visual modification. After the initial cut using this technique, the two sides of the mesh can be seperated over the course of many frames simply by moving vertices, which is an operation fast enough to run at real-time interactive rates.

Slicing along an infinite plane works well on convex meshes, but with concave meshes (especially those with multiple parallel protrusions, such as the bones of a foot) slicing this way will cause distant, unrelated parts of a mesh to be sliced if they happen to intersect the cutting plane. To overcome this limitation, we propose traversing the mesh by triangles that intersect the cutting plane to determine the set of relevant triangles. Furthermore, in order to detect if part of the mesh has been severed from the rest, a similar traversal algorithm can be used to find all contiguous groups of triangles.

Chapter 5

Mesh Geometry

Cutting through a mesh necessarily requires efficient manipulation of mesh geometry. The purpose of this chapter is to outline how 3D mesh geometry is typically stored, rendered and manipulated. The chapter then describes several optimisations that can be used to make mesh modification more efficient.

5.1 Introduction

In chapter 4 (especially section 4.4), we highlighted our desire to use polygon meshes to store and render patient geometry for the purposes of simulating surgical cutting. Our reasoning for this is that most modern graphics hardware, including all current consumer-level graphics hardware, is optimised specifically to render geometry represented this way. However, most conventional 3D applications will typically not modify the topological structure of the geometry that they render. For instance, most games using 3D polyhedral models will only manipulate them according to a set of preset animations, often manipulating a ‘skin’ of vertices with a set of internal ‘bones’ in order to simulate the movement of limbs, appendages etc. Our application requires direct manipulation of mesh geometry, according to user input controlled by a simulated bone saw. This will involve adding, removing and moving many vertices and polygons within the mesh. In order to perform these manipulations efficiently, we describe below how meshes are stored and used in the graphics pipeline. Based on this, we propose a set of optimisations for enhancing the efficiency of modifying geometry data in its preferred format for the graphics pipeline, and suggest a supplemental format for storing geometry that aids performing geometric calculations.

5.2 Real-time 3D Rendering

5.2.1 Model-space Geometry

In 3D computer graphics terms, a 'mesh' is a set of polygons that collectively describe the exterior surfaces of an object (Foley, 1995, pp. 473–478). Meshes are used as a hollow approximation of solid, voluminous geometry, and typically consist of adjacent triangles. The illusion of a mesh being solid is maintained by ensuring its constituent polygons (which describe all the visible exterior faces of the object) always form closed surfaces.

To maximise rendering efficiency, the geometry of a mesh is typically stored as a list of vertices and a list of indices (Foley, 1995, pp. 296–299). The vertex list describes each unique vertex in the mesh, containing various data such as its position, colour, normal vector, texture coordinates, etc. The index list defines which vertices belong to each polygon, and maximises the reuse of vertex data by referring to vertices by their numerical index within the vertex list. This allows efficient use of vertex caching on the GPU to reduce the amount of processing required.

5.2.2 Transformation to Screen-space

One of the simplest rendering methods that utilises mesh data in this format is that of rendering out the indices as a list of triangles, where each triplet of index numbers refers to the three vertices comprising a single triangle. The graphics hardware will then transform the three vertices from their local mesh coordinate system into 'screen space' — the coordinate system used by the display screen — and rasterize the triangle that is formed by the three points on the screen.

The transformation from local geometry world-space to screen-space is carried out by the graphics device through the use of matrix multiplication (Eberly, 2007, pp. 89–90). The position vector of each vertex is treated as a 4x1 matrix (4 rows in one column) and multiplied out as such. Conventionally, there are three matrices used in the transformation – the world (or model) matrix, the view matrix and the projection matrix. However, some pipelines (such as that used in OpenGL ES 1.1) combine the view and world matrices into a single matrix.

The world matrix is unique to each model or object rendered in the scene, and represents the mesh's position in world-space. Multiplying each vertex's position by the mesh's world matrix transforms the vertex into its simulated position in the game world. The view matrix represents the camera's viewpoint in the world, and usually remains constant for the duration of each frame rendered. Multiplying by the view matrix transforms each vertex's position in the game world into coordinates relative to the point of view of the camera. Similarly, the projection matrix is used to transform coordinates from the camera's 3D view-space into 2D screen-space. If the view is being rendered in perspective — that is, with objects exhibiting parallax motion based on their distance from the camera — the projection matrix is used to set the field of view of the viewport and transforms vertices accordingly.

For each set of vertices from a mesh, the world, view and projection matrices will remain constant for that batch of vertices. Therefore, the three matrices can be multiplied together to create a single world-view-projection matrix which reduces the number of matrix multiplications required. Furthermore, since the view and projection matrices generally remain constant for the current frame (and between frames where the camera does not move) the number of matrix multiplications can be further reduced by multiplying the view and projection matrices together to create a view-projection matrix.

5.2.3 Rasterization

Rasterization involves filling each polygon formed by its constituent points in screen-space (the set of coordinates resulting from matrix multiplications) with an appropriate colour value (Eberly, 2007, pp. 89–90). The data from each vertex comprising the triangle is interpolated for each of its rasterized pixels, so a weighted average of the data is used to calculate the pixel's final fill colour and depth. The vertex data may include a diffuse colour, one or more pairs of texture coordinates (from which texture lookups can be performed on 2D images), and various parameters for calculating lighting values — such as the normal vector, tangent, binormal, etc. — at the given point.

The colour of each rastered pixel is often determined by using a combination of the pixel's diffuse colour and/or texture lookups as a base colour, on which various lighting calculations may be performed. One common lighting model applied to 3D geometry is the Phong illumination model (Phong, 1975), although several others exist (Gouraud, 1971; Ward, 1992). Furthermore, with the advent of programmable pixel shaders, the application may define an arbitrary method of calculating the final pixel value from the geometric parameters passed in.

5.2.4 Depth-buffering

Almost all 3D-rendered scenes will contain one or more overlapping objects at varying depths. In order to render these objects such that the nearer objects appear in front of the more distant objects, the pixels obtained from rendering the nearer objects must take priority over those from more distant objects. A simple method for achieving this is known as the "painter's algorithm" — all the objects in the scene are rendered in order of their depth (starting with the most distant, progressing to the nearest), so the nearer objects are drawn over the top of the more distant objects. However, this approach does not work when objects intersect each other — one object will always appear entirely in front of the other. A refinement of this approach sorts all individual polygons by their depth from the camera, so intersecting objects' polygons will appear in the correct order, but again this approach will not work with intersecting polygons. Furthermore, ordering all polygons in a scene can be a complex and inefficient operation to perform, especially when the camera and/or objects move around at speed.

In order to provide more accurate depth sorting, a hardware depth buffer (also known as a z-buffer) is often used (Eberly, 2007, p. 419). The depth buffer is equal in width and height to the frame buffer which it is attached to. In its most common configuration, the depth buffer will be cleared before each frame, with all values being set to the maximum depth value. When each pixel is rasterized, its depth value is compared to the value already present at that location in the depth buffer. If the pixel's depth is less than that found in the depth buffer, its depth value becomes the new depth at that location in the buffer, and the pixel's colour is calculated and written to the frame buffer. If the pixel's depth is greater than the depth already found, the pixel is behind some other geometry in the scene and thus ignored.

The depth buffer is biased such that nearer geometry has a higher resolution of depth values than more distant geometry (Blythe et al., 1999). This is due to the limited range of values available in the depth buffer, which will cause z-fighting to occur on non-coplanar polygons if their separation distance is insufficient for any of their overlapping fragments to be assigned different depth-buffer values. Since geometry rendered closer to the camera is typically larger on-screen and requires more detail, any geometric distortions such as z-fighting will be much more pronounced and noticeable on nearby geometry than on geometry further away. The bias in the depth buffer therefore reduces the effect of distortions in the most important geometry, at the expense of possibly allowing distortions in more distant geometry.

5.3 Limitations of Real-time Mesh Manipulation

To simulate the effect of cutting through a mesh, it is necessary not only to visualise the cut being made through the mesh, but also to split the mesh into two separate objects if the cut completely severs one part of the geometry from the rest. Simulating the cut in real-time will require geometric modifications to be made to the mesh in each rendered frame as the cut progresses. The modified geometry may be that of the target mesh, as would commonly be expected, but depending on the method of visualisation, it may also be an intermediate mesh representing only the additional surfaces of the cut. Each time that a mesh is modified, which should occur a maximum of once before each frame is rendered, the update geometry must be presented to the graphics hardware before the change is visible to the user.

5.3.1 Modifying Vertex and Index Buffers

The performance penalty associated with sending the entire vertex and index buffers of a large mesh to the graphics hardware each frame is not insignificant. In the case of very large meshes (consisting of hundreds of thousands or millions of polygons), the delay from passing the mesh's entire buffers to the hardware each frame would reduce an application's frame rate substantially enough to render it unusable. Obviously, it is therefore desirable to update as few vertices and indices per frame as possible.

Updating the data of an existing vertex or index (where the position of the entry in the list is known) is the most straightforward modification to make, as it only requires overwriting the data already present. This is achieved simply by locking the target position in the buffer, writing the new data, and unlocking the buffer. However, other operations — such as adding and removing vertices and indices — require a different approach.

5.3.2 Appending Vertices and Indices

Appending new vertices or indices is relatively straightforward with proper preparation. Since both types of buffers are of fixed size when created, if the buffer size matches the data size at its creation time, there is no room for expansion afterwards. However, creating the buffers with a percentage of extra space alleviates this problem. When the buffer has extra capacity, it is simply necessary to write the new element data at the position following the last element, just like updating an existing buffer element. In the case of appending new elements to the index buffer, the polygon count is incremented accordingly.

If the maximum capacity of an index or vertex buffer is reached, it is necessary to create a new buffer with additional capacity and copy over all data from the previous buffer. This is an expensive operation, but choosing an appropriate amount of additional space for each buffer reduces the number of times that this reallocation occurs.

5.3.3 Removing Vertices

Since both vertex and index data are stored and referenced contiguously, removing entries from within either is undesirable. As vertices are referenced from corresponding indices, unused vertices can simply be ignored as long as all indices referencing them are removed or changed. The performance penalty for having a small number of unused vertices within a vertex buffer is trivial when rendering indexed polygons. By contrast, to remove a vertex from the list entirely would require shifting all the following vertices down and updating all indices referring to any following vertices — potentially a very expensive operation.

5.3.4 Removing Indices

Removing polygons requires a different method, as all indices within a given range of the index buffer will be rendered. With the common case of rendering indices as a list of triangles, two methods are available with similar results. The first method is to overwrite the indices of the target triangle with those of the last triangle in the list, and shrink the list size by one triangle. This allows a triangle to be removed from the middle of the list without requiring all of the following elements to be moved down. The ordering of the list is not preserved however, but unless the ordering is important to the application performing the cutting, there should be no impact apart from a possible minor reduction in cache utilisation.

Secondly, all three indices representing the triangle to remove can be set to the same value. This results in the hardware attempting to render a triangle made of the same vertex three times, which is a degenerate triangle and should be considered a no-op. Depending on the architecture of the graphics hardware, this 'triangle' could potentially be rendered as a single point, but if the duplicate index chosen is in use by other triangles, the resulting pixel would be rendered anyway and thus not noticeable to the user.

5.3.5 Detecting Disjoint Meshes

To detect if a mesh is disjoint (and therefore has been cut into two or more separate parts) a simple test involves traversing the mesh by following triangles that share one or more edges. If all the triangles in the mesh are reachable by this method, then the mesh is still a solid object. However, if some triangles are not reachable, then the mesh is disjoint. Traversing the mesh again from one of the omitted triangles will yield another set of triangles, which form a separate mesh. Repeating the traversal process until there are no more omitted triangles will divide all the triangles into their respective disjoint meshes. When meshes are stored in vertex and index buffers, the index buffer can be inspected to determine which triangles are adjacent. These are any triplets of indices that share two indices with another triplet. A simple iterative inspection for an individual triangle will involve ni checks, where n is the total number of triangles and i is the triangle index. Checking for all triangles will have approximately $O(n!)$ complexity.

Although it is not necessary to perform this check every frame — only when it is likely that a cut has finished — it is still less than optimal to perform the check in this manner. As will be discussed later, cutting the mesh will require several geometric operations to be carried out on the constituent triangles and vertices in the mesh regardless of the precise algorithm used. Although it is possible to operate directly on the vertex and index buffers, in practice this is a rather unwieldy way of handling the data. A more suitable method of storing geometry may provide a more natural way of operating on the data.

5.4 Methods of Storing Geometry

5.4.1 Data Formats

The most common method of storing mesh data is in vertex and index buffers. The vertex data may be stored in any of several different vertex formats. The index data is stored as unsigned integers, either 16-bit (allowing a maximum of 65,536 indices) or 32-bit indices (allowing over 4 billion indices). Depending on whether indices are rendered as a triangle strip or triangle list, either $n-2$ or $\frac{n}{3}$ triangles can be rendered from this many indices. We will be using triangle lists, as they allow a single triangle to be modified without affecting prior or following triangles. We will therefore have a maximum of 21,855 (16-bit) or approximately 1.4 billion (32-bit) triangles per mesh, depending on index format.

5.4.2 Vertex Formats

A vertex format is simply the order in which data values are interleaved within the vertex buffer when sent to the graphics hardware. Vertices are grouped sequentially within the buffer, with the total data size in bytes for a single vertex referred to as its 'stride'. Each vertex is comprised of a number of elements, the type of each and offset within the stride of each being specified by the vertex format. Types of vertex elements include the vertex's position, diffuse colour, texture coordinates, surface normal vector, tangent vector, binormal vector and others. Not all element types are used in all geometry for example, a simple model may only specify its vertices' positions, normals and texture coordinates. However, within the same vertex buffer, the same format should be used for all vertices.

5.4.3 Object-oriented Geometry

Although the final mesh data must be stored in vertex and index buffers in order for the graphics hardware to process it, intermediate calculations have no such requirement. When using an object-oriented programming language, it can be much more useful to represent the mesh data as collections of objects. Different types of data are arranged into different 'classes', such as Vertex, Edge, Triangle and Mesh classes. Each class has member variables ('fields') that contain the data of the object represented by the class, and also member functions ('methods') that allow operations to be performed on or with the object's data. For example, a Triangle class may have a method that calculates its area or perimeter, or whether it intersects a given plane.

5.4.4 Advantages of Object-oriented Geometry

The main advantage of using object-oriented data structures to hold mesh data is that it greatly simplifies the implementation of geometric algorithms. Metadata can be stored along with geometric data to allow properties of the structure represented to be determined more easily. For example, if each Edge object stores a list of the Triangle objects it is part of, then finding the adjacent neighbours of any triangle simply requires iterating through its child Edge objects and querying each for its parent Triangle objects. The data stored in an object-oriented mesh can be derived from existing mesh data stored in the more conventional vertex/index buffer format. Since there are multiple possible vertex formats, we chose to add an additional class specifically dealing with converting mesh data between any vertex format and the object-oriented format. This 'helper-class' allows any vertex buffer to be converted into a series of Vertex objects, and conversely, allows any Vertex to have its internal data written out into a buffer with any given vertex format.

5.4.5 Importing Vertices from Device Format

When importing vertices from a vertex buffer, we have found it necessary to ensure that vertices do not have duplicate positions. Although it is possible for two vertices to lie on the same position but have different vertex data (for instance, as would be necessary for a cube with different coloured faces), this can cause a problem for geometric algorithms operating on the mesh. For example, testing whether two triangles share an edge by quickly comparing their references to Vertex objects will no longer work – the positions of the vertices must be compared instead. Similarly, collecting all vertices that lie within a given area will give more results than expected. Although most, if not all, of the geometric algorithms we need to apply to geometry can be reworked to allow for duplicate vertices, we chose to sidestep this problem by simply not allowing duplicate vertices. This also brings the additional benefits of potentially decreasing memory usage and processing time on any operations that iterate through all vertices, but may result in cosmetic anomalies in some meshes.

5.4.6 Exporting Vertices to Device Format

Exporting object-oriented mesh data back into vertex/index buffer format is a three-step process. First, each vertex must be associated with a unique index number. This can be achieved either by mapping each Vertex object to an index using an associative container, or simply by setting a field in each Vertex to its index value. If a Vertex stores its index number, it must either only be a member of one mesh, or must store the index numbers for each mesh it is part of in an associative container.

Second, the vertex data must be written to the vertex buffer. Each vertex must be written at the position indicated by its associate index, to ensure that when indices are generated they map to the correct vertex data.

Finally, all the triangles in the mesh are iterated through, and the index numbers of their child vertices are written to the index buffer sequentially. To determine the index for a given vertex, its value is retrieved either from the associative container or from the Vertex object itself, depending on how this was previously stored.

It is important that vertices remain associated with their index number throughout their lifetime, to ensure that updates to the vertex buffer are performed on the correct vertex. Likewise, when updating a triangle in the index buffer, it is important that any Triangle object's position within the index buffer can be retrieved to perform later updates. If either of these criteria are not met, the entire buffer needs to be regenerated to ensure consistency, which is a comparatively long operation as previously discussed.

5.4.7 Use of Octrees

When performing various geometric operations on a mesh, it is often desirable to quickly find all geometry within a given area. For example, this is useful for computing collisions between meshes, as it allows precise collision tests to be carried out on only the most relevant polygons. To this end, meshes in our object-oriented geometry system store geometry in two spatial partitioning constructs known as 'octrees'. Like the name suggests, an octree is a tree structure, where each node in the tree has up to eight child nodes. Each node in the tree represents a cuboidal volume, and the volumes represented by its eight possible child nodes are formed by dividing this cuboid exactly in half, along each of the three coordinate axes. In this manner, the initial node (or 'octant') contains the entire mesh, and child node are generated recursively for each octant segment that contains geometry. Our implementation uses an octree to store vertices that lie within each octant, and an octree to store polygons that fully or partially lie within each octant. To reduce overhead, we place a lower bound on the number of entries an octant would contain before it is generated, and we also place an upper bound on the depth of the octree to prevent excessive recursion.

5.5 Conclusions

Mesh geometry is commonly stored in efficient vertex and index buffers, which are the preferred format for the graphics device. Vertices and indices can be added and removed to and from their respective buffers without reallocating the entire buffer, if some simple optimisations are used. Storing a copy of mesh geometry in an object-oriented manner provides benefits over only storing geometry in efficient buffers when manipulating the geometry. When geometry is stored in an object-oriented structure, geometric methods can be associated with each object to make complex geometric operations easier to implement. Additionally, finding adjacent polygons is more efficient this way. Geometry data can be converted between the two formats, which is necessary when rendering object-oriented geometry.

Chapter 6

Implementation

6.1 Introduction

This chapter provides additional details on our implementations of the mesh cutting algorithms and the supporting application. The aspects of the implementation discussed herein are not considered critical to mesh cutting, but are added for completeness' sake and to highlight several measures taken to enhance the realism of the cutting procedure from the user's point of view.

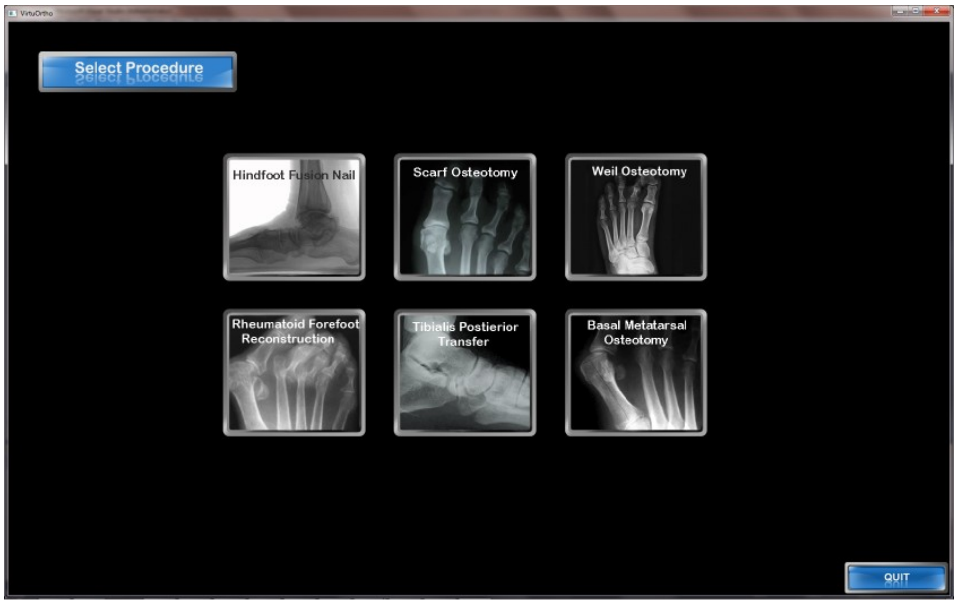
Select Procedure
Hindfoot Fusion Nail
Scarf Osteotomy
Weil Osteotomy
Rheumatoid Forefoot Reconstruction
Tibialis Postierior Transfer
Basal Metatarsal Osteotomy
QUIT

Figure 6.1: Operation selection screen, with first icon highlighted

6.2 Operation Selection Screen

In a full-featured, real-world surgical training tool, it would be desirable for users to practice several different operations. In order to provide the user with a choice of the operation to attempt, the first screen displayed to the user allows them to select an operation. Currently, this is largely for illustrative purposes, as no unique, operation-specific features or data are enabled depending on the user's choice, and we instead use the choice made to select which test model to load. Figure 6.1 shows this screen, with the first operation selected.

6.3 Model Loading

The models used to test the various algorithms are stored as files in Direct X's '.x' model format. The Direct X API contains built-in routines to load in these files and automatically allocate vertex and index buffers to contain the geometry within. In order to use the models in an object-oriented manner, the conversion routines described in section 5.4.5 have been implemented. Since all the models are meant to represent bones or collections of bones, no individual textures are loaded. Instead, a bone texture is applied to all meshes. A programmable shader is used to apply a simple Blinn-Phong per-pixel lighting model to the mesh. Figure 6.2 shows a reference foot model that has been loaded into the application.

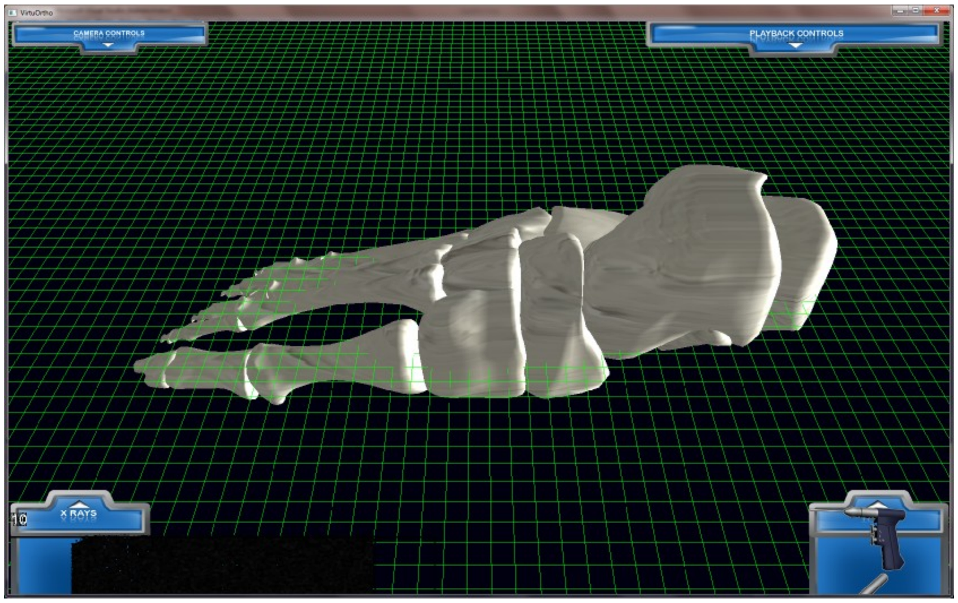

Figure 6.2: Main operation screen

6.4 Input

In order to more realistically simulate the experience of performing orthopaedic surgery, an input device that provides 3D input and haptic feedback is used to control the mesh-cutting algorithm. The obvious benefit to 3D input is that the user can quickly and accurately indicate a location and direction in 3D-space. Traditional 2D input devices – such as a mouse and keyboard – can be used to indicate locations in 3D, but not intuitively.

The x and y axes of 2D input devices can be mapped to two of the dimensions in the 3D world, possibly aligned to screen-space, and a third axis can be used to indicate depth, if a third input axis is present on the input device (for example, a second thumb-stick on a game pad, a throttle control on a joystick or a scroll-wheel on a mouse). If no third input axis is available, a key or button can be held down to remap one of the two available axes to modify depth. By comparison, a 3D input device (such as the Sensable Phantom Omni) can simply be moved in real-world 3D space to move an indicator in the virtual world to the required position. This provides a direct, intuitive mapping from the position of the user's hand to the position indicated in world-space.

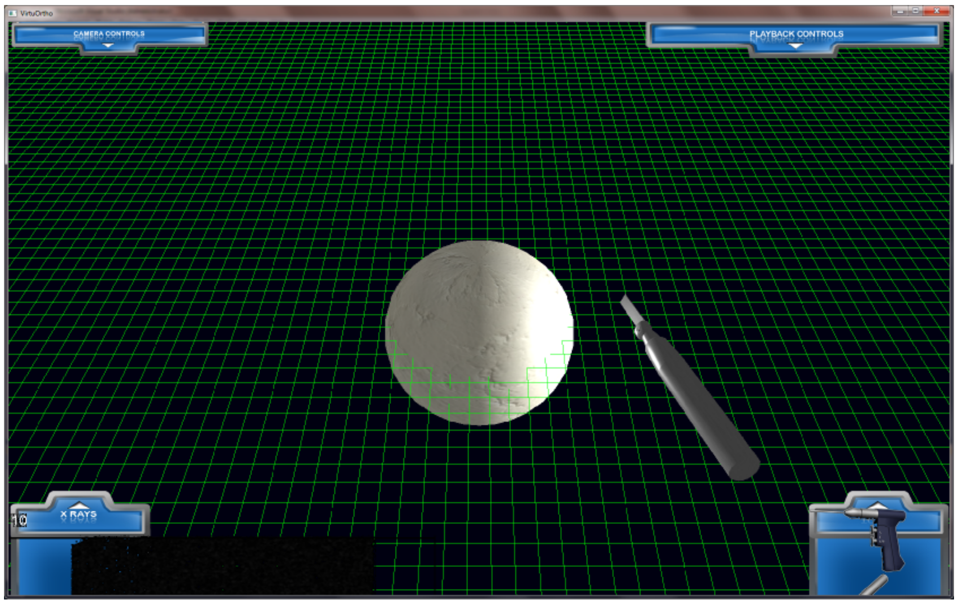

Figure 6.3: Surgical saw with test model

Indicating direction is also possible when using 2D input devices — for instance, first-person shooter games typically map the input device's horizontal axis to turning left and right, and map its vertical axis to looking up and down. This bearing-and-elevation approach gives a simple gimbal-locked indication of direction. From a first-person perspective, it is relatively simple to judge the direction to a given object or position in the scene, as the user 'looks' at the position by centring it in their viewport. However, from a third-person perspective such as we are using, it is harder to judge the direction between two points in 3D without an appropriate visual cue such as a virtual laser beam being emitted in the currently-indicated direction. The 3D input device solves this problem by allowing the user to freely orientate its control handle, whose orientation is replicated in the simulation. This combination of input measurements allows the user to freely move and orientate a device in real-world space, which maps intuitively to the position and orientation of a marker in the simulation.

6.5 Haptic Feedback

With a device that supports haptic feedback, the application can respond to the user's input by providing reactionary forces whenever the user touches an object in the virtual world. A force may be applied to move the position of the haptic device in any direction, limited only by the device's range of motion and maximum motion output. The force applied must take into account the velocity of the device's control handle, the normal of the surface collided with, the force being applied by the user, and any residual force being applied by the device.

The motion of the device is simply calculated as the difference between the device's position in two consecutive frames. At the instant of collision between the simulated tool and a surface, a reactionary force can be calculated based on the collision vector and the surface normal. The reactionary force must be reflected by the surface. Given a surface normal N, an incident vector V, the reflection vector R is calculated as follows:

$$R = V + 2(V \bullet N)$$

Ignoring any other forces acting on the device, the device can be accelerated to match the reflected restitution velocity by applying a suitable force for a single delta interval. The interval between measurements, which should be the application's frame-rate, is used as the acceleration duration for the purposes of calculating the force to apply. The appropriate force is calculated for each axis as follows:

$$f = m \cdot a$$

where f is the force, m is the mass of the device and a is the acceleration rate calculated as follows:

$$v = u + a \cdot t$$
$$\Rightarrow a = \frac{(v-u)}{t}$$

where t is the delta time interval (usually 1/60 seconds), u is the initial velocity and v is the final velocity. For any target velocity given any starting velocity for a known mass of the device and constant time delta, the force can therefore be calculated thus:

$$f = \frac{m \cdot (v - u)}{t}$$

In addition to the force generated by the device, the application must also take into account the force applied by the user. The device does not measure the force applied by the user, but only measures its position and orientation at regular intervals. The force applied by the user must therefore be inferred from the device's motion between frames, and the force previously applied by the device itself. By inferring the acceleration of the device between two frames, the total force applied is calculated for each axis as:

$$f = m \cdot a$$

This provides the total force applied to the device in each axis on the previous frame. To obtain the part of the force applied by the user (or other real-world forces not controlled by the device), the force known to have been applied by the device itself can be subtracted from this. This remaining force needs to be opposed if the virtual tool simulated from the device is in contact with a surface. The direction and magnitude of the opposing force can be calculated by reflecting the user-induced force off the surface collided with. This can then be added to the forces the device applies, to provide adequate resistance to the user pushing against the simulated surface.

6.6 Use of particles

When cutting through bone in real life, a small quantity of dust and swarf flies into the air as a result of the sawing action. In order to increase the realism and accuracy of the simulated cutting process, a particle effect was added to the application whenever the virtual saw blade removes material from the mesh.

Particle systems are used to simulate various effects, such as rain, dust and shrapnel after an explosion, exhaust trails, water splashing, blood splatters, and more (Reeves, 1983). Early particle systems would render dots of varying colours, often used to produce special effects for motion pictures or television shows. Modern particle systems usually take a very simple mesh, typically consisting of a handful of polygons, and render many duplicates of the mesh with slightly different parameters. Many particle systems use "screen-aligned quads", which are simply rectangles that always face the screen (regardless of camera position), onto which a semi-transparent texture is mapped.

Almost all particle systems cause all their particles to move in a similar predefined manner. Constant acceleration formulae may be used to specify simple motion. For instance, a simple particle system may generate particles flying out of an explosion. In this case, the system may choose a random direction, velocity and maximum distance for each particle, with the starting point being the origin of the explosion. Over the next frames of the game, the position of the particles is calculated based on these parameters, and when the particle's maximum distance or lifespan has been exceeded, the particle is removed from the scene.

The number of particles may vary, as may the time at which they are generated or destroyed. For instance, a particle spawner may continuously generate particles at regular intervals. The destruction of a particle may occur after a set number of frames, after it has traversed a set distance, or after another condition is satisfied.

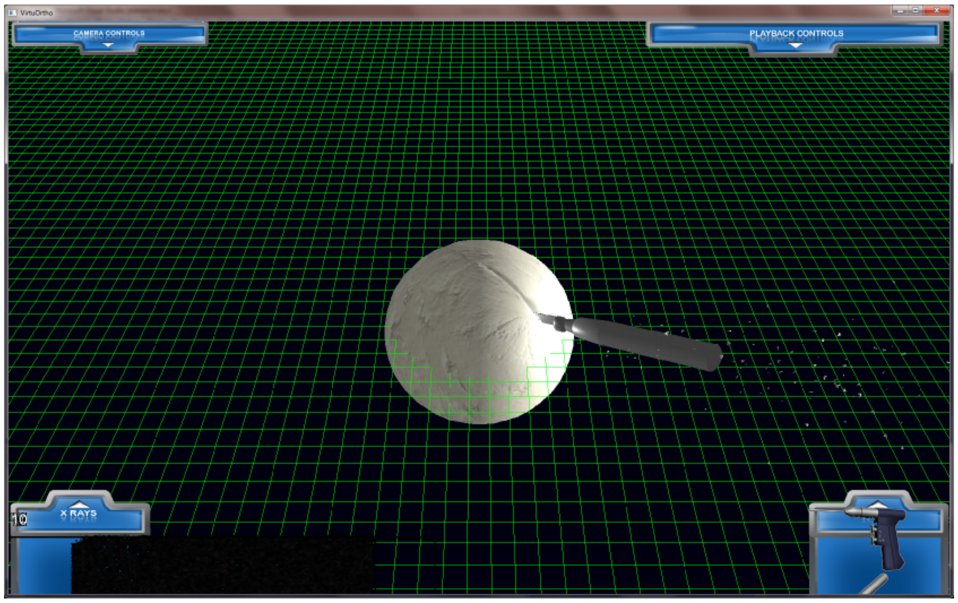

Figure 6.4: Cut in progress, producing particles

The particles used in our application are very small tetrahedrons, emitted in the opposite direction to the saw handle. The orientation, velocity and precise direction of each particle are randomised somewhat to make the particles appear more realistic. After the particles travel a small distance, they fade out over a couple of frames. The high quantity of particles appearing and disappearing whilst flowing away from the blade emphasises that the effect is not focussed on any one particle but on their collective presence indicating that cutting is taking place. In this way, the particles provide a visual indication that the saw is cutting through the bone, which may sometimes be otherwise difficult to discern from the user's viewpoint of the simulation.

The quantity of particles generated is determined by how much of the mesh is being cut. When generating particles for the separation stage of the planar slice method, the number of particles generated is proportional to the number of vertices being separated by the saw. This provides a rough estimate of the volume of the mesh being cut away. However, in models with a high vertex density, there will be a larger number of particles generated when cutting a particular area than would be generated when cutting the same area in a mesh with a lower density of vertices. In future work, this problem could be alleviated by accurately calculating the volume being removed from the mesh, and generating particles whose total volume reflects this.

6.7 Use of Stereoscopic 3D

Stereoscopic images have been around for almost as long as photography, and computer-generated stereoscopic images have been rendered for many years (Okoshi, 1980; Wickens, 1989). In recent years, stereoscopic 3D rendering has also been made available to consumers for gaming and other entertainment purposes. All forms of stereoscopic 3D technology display a slightly different 2D view of a 3D scene to each eye of the viewer, such that the combination of views allows the user to perceive objects in the scene as having depth. The main benefit of using stereoscopic 3D in this work is that the user is more able to perceive the three-dimensional position of the saw blade relative to the patient's foot. This makes performing cuts much more accurate and realistic.

There are several methods of achieving stereoscopy in 3D, including the older anaglyph (blue and red) method, the more modern active shutter and polarized lens methods and newer autostereoscopic methods. In our particular implementation hardware, all 3D stereoscopic calculations are performed in the 3D graphics driver, which allows the 3D stereoscopic effect to be used by all full-screen 3D applications without requiring any specific consideration from the developer. The graphics driver automatically produces the second (shifted) view of a scene and presents this to the output hardware at the next refresh. To view the steroscopic images, we use a consumer-grade monitor specifically designed for displaying 3D images, which refreshes at twice the normal rate (120Hz instead of 60Hz) in order to display the successive interleaved left and right frames. This refresh rate is synchronized with active shutter glasses worn by the viewer(s) in order to alternately present the left and right frames to each eye as they are displayed.

Chapter 7

Testing

7.1 Introduction

This chapter documents the testing of our implementations of the cutting algorithms discussed earlier. First, we test the performance benefit of using three different geometry optimisation; two consisting of different buffer allocation schemes for storing dynamic vertex and index buffers, and the third being a fast method of removing triangles from geometry. Then, we evaluate the performance of the planar slice cutting method by performing a set of cuts on two test meshes. We measure the frame rate at which each cut takes place, and analyse this metric in the context of different cuts and meshes.

7.2 Geometry Optimisations

In section 5.3.1, three optimisations of dynamically adding and removing geometry from meshes are evaluated. We have run tests to compare the performance of modifying geometry when these optimisations are used, and when they are not. The optimisations in question are:

- Pre-allocating additional space in the mesh's vertex buffer
- Pre-allocating additional space in the mesh's index buffer
- Setting indices to 0 instead of removing them from the index buffer

The tests we ran were performed on two meshes - the first, a foot mesh with a high polygon-count ('high-poly'); the second, a sphere mesh with a low polygon-count ('low-poly'). The high-poly foot mesh (model 1) is representative of the complexity of the geometry that would be used in a real-world foot simulation. The low-poly sphere (model 2) provides a baseline to contrast operations on model 1 with, especially operations whose running time scales linearly or polynomially with the number of polygons

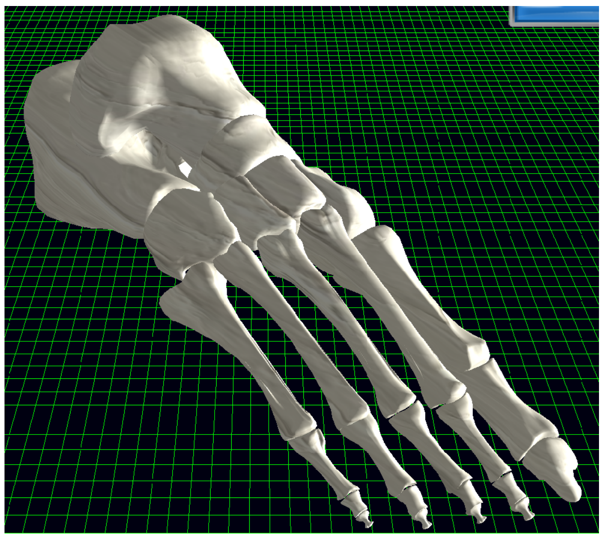

Figure 7.1: Mesh 1 - a foot model with a high polygon-count

in the mesh. Comparing the change of performance for model 2 when parts of the cutting algorithm are optimised and when they are not should make any constant-time overhead more apparent.

In order to simulate adding triangles to a mesh in a repeatable manner, two techniques were used. The first method was to perform a series of 12 cuts into the mesh, using the planar slice algorithm described in section 4.5. Each cut followed a plane aligned to the y and z axes, and was separated in the x axis by 1/13 of the model's width. The saw blade was oriented such that it pointed along the z axis, with the width of the blade aligned to the y axis. The simulation automatically moved the saw quickly up and down and gradually forwards along the cut, following a sine wave to cover the model's entire height and depth. The second method of adding triangles gradually subdivided all of the model's initial triangles over a series of frames, by bisecting each of them along a line from one vertex to the mid-point of the opposite edge.

Simulating the removal of triangles was done progressively over a series of frames, as with the subdivision method of adding triangles — except that in this case, each entire triangle was simply removed. This will cause the mesh to no longer form a closed surface, but this has no consequences for the purpose of testing performance.

When each batch of tests occurred, the entire process was automated and left running overnight, in order to generate a substantial amount of data. To eliminate any bias caused by the effects of running an application too long (such as memory leaks), each test ran once, then the program exited and an automated script relaunched it. A 'test' here refers to carrying out a testing operation (performing the series of cuts, subdividing the mesh, removing triangles, etc.) for one value of the variable being tested. The next instance would then run with the variable incremented — for instance, when testing buffer allocations, the allocation percentage would increase (or, after the final value, wrap around to

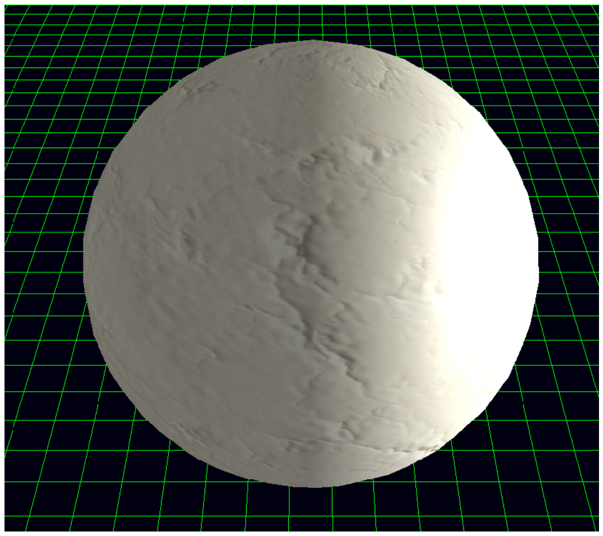

Figure 7.2: Mesh 2 - a sphere with a low polygon-count

its starting value). This process was intended to mitigate the effects of any performance fluctuations that might occur on the host machine, since other running processes will occasionally require system resources in an unpredictable manner.

7.2.1 Appending Vertices

As described in section 5.3.2, the addition of extra triangles to a mesh can be optimised by pre-allocating a given percentage of extra space in its vertex and index buffers. This facilitates the appending of additional data to the buffer without having to allocate and copy data into a new buffer each time. A similar approach has commonly been used in ‘vector’ resizable arrays for some time, but it is not generally used with vertex and index buffers.

In order to determine an appropriate amount of additional space to allocate in each buffer when cutting meshes, we performed a series of tests using different allocation schemes. The experiment’s control sample was the naïve approach of only allocating exactly the required space each time a buffer is created. Using this approach will require the entire vertex and index buffer to be reallocated and have its entire contents copied in again each time a triangle is added to the geometry. Obviously, this is far from optimal. The optimised buffers being tested used allocation schemes that allocated between 101% and 120% of the required space to hold their data each time they were reallocated.

Each buffer (vertex and index) was tested using each of the 21 allocation schemes when adding triangles from two sources — mesh cutting, and subdivision of existing triangles. The number of triangles added by mesh cutting varies greatly with the particular model and location of the slice, whereas progressively subdividing triangles as described above will result in a constant increase in the total number of triangles. A summary of the time spent adding vertices to the mesh's vertex buffer (for both meshes) is shown in figures 7.3 — 7.6.

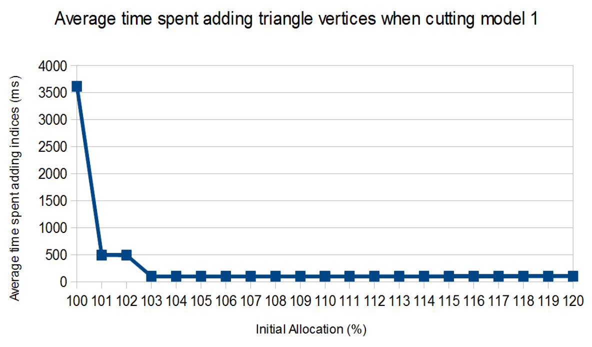

Figure 7.3: Cutting Model 1

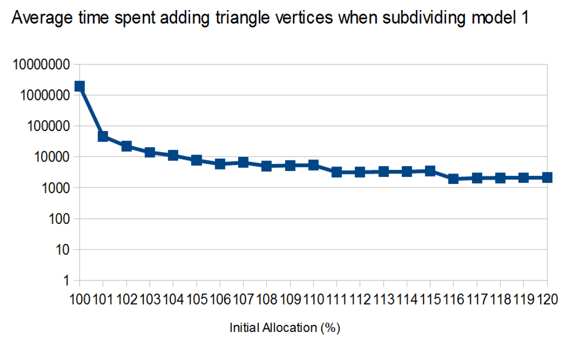

Figure 7.4: Subdividing Model 1

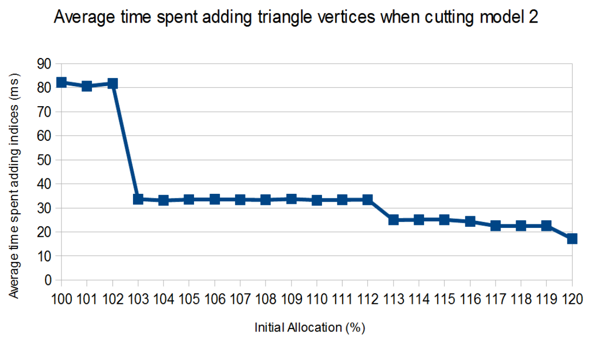

Figure 7.5: Cutting Model 2

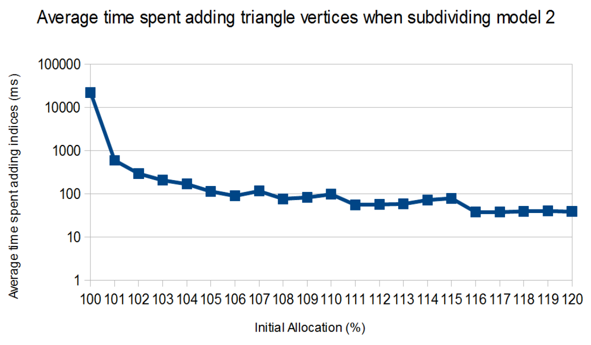

Figure 7.6: Subdividing Model 2

As can be seen from these figures, there is correlation between the time taken to add polygons to a mesh and the allocation scheme used to allocate the vertex buffer. Using the naïve approach of only allocating the minimum required space for each buffer resulted in very slow additions of vertices.

The sharpest increase in performance when cutting either mesh is seen when increasing the buffer allocation from 100% to 103%. Increasing the allocation above 103% causes more gains to be made, but less significantly so. When subdividing the meshes, the addition time decreases more smoothly as the initial allocation is increased. Some interesting, and somewhat unexpected, features of the subdivision results are the spikes that occur around 107%, 110% and 115% for each model. This may be caused by the amount of geometry data at these particular levels not aligning well to the machine's memory page size or CPU cache size, or it may be caused by the individual buffer allocation and copy time increasing at these levels and the number of reallocations not decreasing enough to offset this.

The greatest benefit is seen in cases where the mesh consists of a large number of triangles initially, or has a large quantity of new triangles added. This is expected, as large buffer sizes resulting from large numbers of polygons result in longer copy operations, and adding larger quantities of polygons is more likely to fill up empty space in buffers.

7.2.2 Appending Indices

The same experiment for testing vertex buffer optimisations was also carried out for index buffers. The setup was identical, except that the index buffer's allocation percentage was modified instead of the vertex buffer's. The results are presented in figures 7.7 — 7.10.

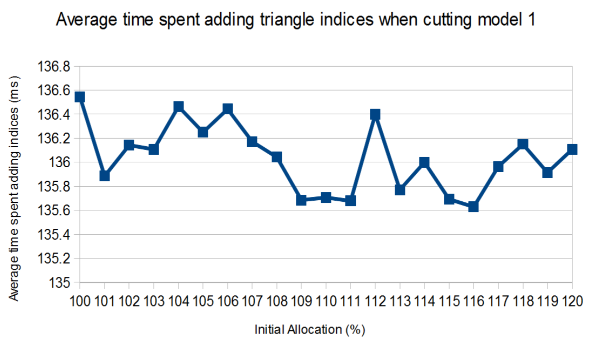

Figure 7.7: Cutting Model 1

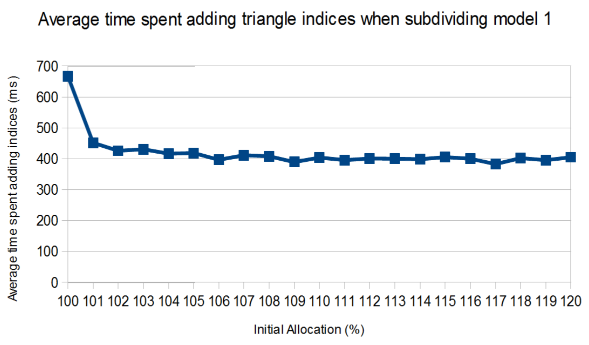

Figure 7.8: Subdividing Model 1

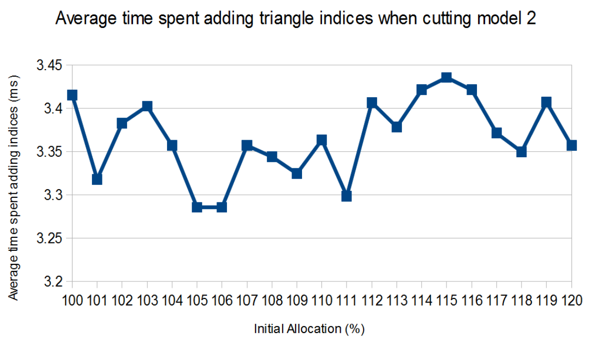

Figure 7.9: Cutting Model 2

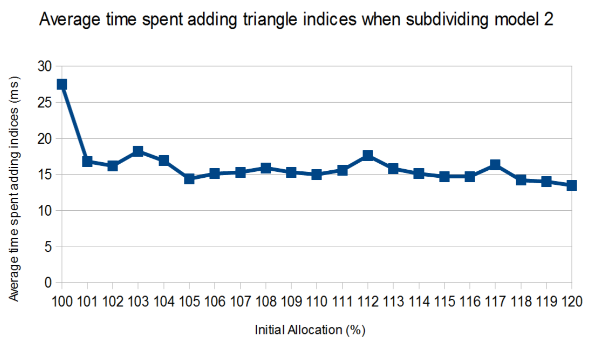

Figure 7.10: Subdividing Model 2

Adding indices was much faster than adding vertices, which is to be expected since index data is typically an order of magnitude smaller than its corresponding vertex data.

In the cutting results, the changes in average time as the allocation percentage increases appear to be much more irregular than those seen in the index buffers. However, all of the measurements in each cutting set are within a single millisecond of each other, which is less than the expected precision of the timing method used (itself being one millisecond). The number of triangles added by cutting is much smaller than the number added by subdividing, and the amount of index data being manipulated is again much smaller than the corresponding vertex data. Therefore, the lack of improvement when appending indices whilst cutting may be due to the smaller amounts of data involved.

In the subdivision results, the data is less noisy and shows an improvement in time taken to append indices as the allocation percentage increases. There a steep drop in time taken between the 100% and 103% allocations, as with all the vertex buffer results, which indicates an improvement over the naïve approach. As with the vertex buffer data, there are also spikes in these data sets, but they do not appear to be as regular as those seen with the vertex buffers. However, the root cause may be the same - a caching, memory alignment or allocation overhead issue.

7.2.3 Removing Triangles

When removing triangles from a model, an unoptimised approach would be to reallocate the mesh's entire index buffer. However, as described in section 5.3.4, there are two methods of optimising removal of indices from an index buffer. The first method provides an intial improvement of removing items from unordered lists, by replacing the item to be removed with that which is last in the list, and shrinking the list by one item. In the case of an index buffer storing indices of triangle vertices, this would require copying the last three indices in the buffer over the indices to be removed and reducing the number of polygons thought to be stored in the buffer by 1.

Although this method would provide a large performance improvement over simply reallocating the buffer, there are several disadvantages from losing the ordering of the list in this way — for instance, this would make it difficult to accurately identify the indices associated with a given triangle for future removals. This limitation can be overcome by noting within a triangle's data structure if its associated indices have been moved from their original position, but to be robust, this approach would require storing a mapping from indices to triangles and from triangles to indices — both of which would consume a large amount of memory and must be kept up-to-date. Therefore, this method was not tested.

The second, preferred method is simply to overwrite the given indices with an arbitrary index number. Since the index buffer is rendered as a triangle list, each triplet of indices is used to select three vertices from the vertex buffer which are rendered as the vertices of a triangle. If all the indices are set to the same arbitrary value, the same vertex will be selected three times, resulting in a degenerate triangle with no surface area. The graphics pipeline should correctly ignore such degenerate triangles, resulting in the triangle not being rendered. This should have the desired effect of removing the original triangle from the visible geometry of the mesh.

To determine the effectiveness of the second (overwrite) method, a series of tests were carried out. These tests systematically removed triangles from two test meshes and noted the time spent removing the triangles. Half of the tests for both meshes used the overwrite optimisation; half did not and merely

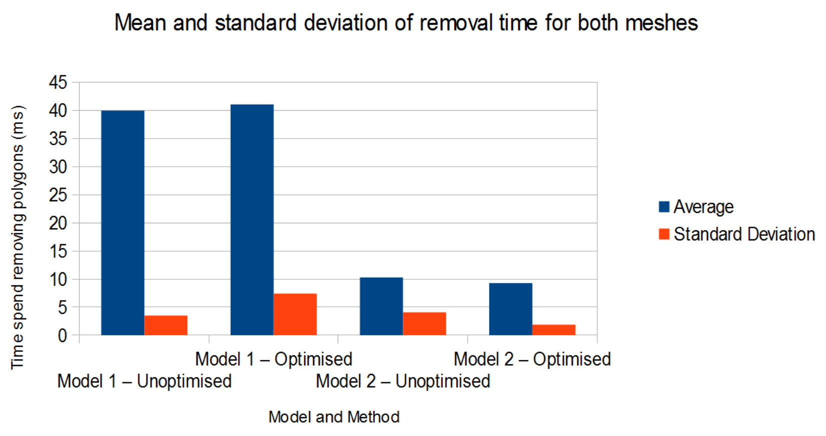

Figure 7.11: Time spent removing triangles with and without the optimisation, for both meshes

reallocated the entire buffer (the unoptimised method). We would expect to see the optimised method presenting a moderate increase in performance over the unoptimised method due to the reduction in data copying required. However, as the results in figure 7.11 show, this is not always the case.

The first model tested had a very high polygon count; the second model had a comparatively low count. With the low-poly model, using the optimised method meant that the mean time spent removing triangles was decreased, as was its standard deviation. However, with the high-poly model, the trend reversed — using the optimised method increased the mean and standard deviation of the time spent cutting. The reason for this may be that the algorithm for selecting the indices to remove does not scale well, which future work could investigate to determine the exact cause.

7.3 Evaluation of Cutting Frame-Rate

In order to be usable as a real-time interactive simulator, the application should maintain an average frame rate of at least 20 frames-per-second. To measure this, we reused the automated cutting routines described earlier to perform a repeatable series of cuts into two test meshes. The time taken to render each frame was recorded, from which the average, minimum and maximum frame rates can be calcuated, along with the standard deviation.

We used two test machines, one of which was significantly more powerful than the other. The more powerful machine was a 64-bit desktop PC, with a quad-core Intel Core i5 750 CPU clocked at 3GHz, 8GB of DDR3 RAM at 666MHz and an nVidia GeForce GTX 285 graphics card with 1GB of VRAM. The lesser of the two machines was a 32-bit Dell XPS laptop, with an Intel T2600 Core Duo CPU clocked at 2.16GHz, 2GB of DDR2 RAM at 333MHz, and an nVidia GeForce Go 7950 GTX graphics card with 512MB of VRAM.

		Mean	Std. Dev.	Min	Max
Fast machine	Model 1	60.98	19.37	5.99	429.36
	Model 2	60.97	14.61	18.47	313.31
	Overall	60.98	20.23	2.29	1000
Slow machine	Model 1	51.21	16.11	0.23	100.49
	Model 2	54.21	11.19	5.28	121.78
	Overall	52.74	15.23	0.1	200
Overall		53.19	15.6	0.1	1000

Table 7.1: Frame rate across two machines and two test models

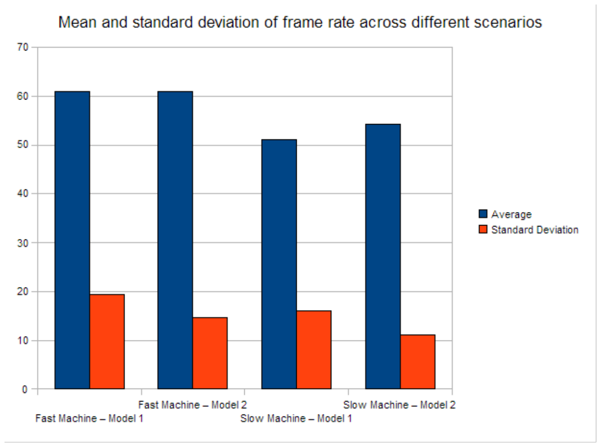

Figure 7.12: Average frame rate, using two meshes on two different machines

Table 7.1 shows a summary of the testing results. For each test, the mean, standard deviation, and minimum and maximum frame rates were calculated. The rows labelled “Model 1” and “Model 2” show the averages of these results. The rows labelled “Overall” show the mean, standard deviation, minimum and maximum values over the entire dataset from each machine and both machines, respectively.

Since both machines ran the application with average frame rates well above the minimum of 20, the application was running at a rate suitable for interactive simulation for a large part of the testing procedure. However, the minimum frame rates indicate that some very long frames occured, with the longest incurring a delay of approximately 10 seconds. We expect this is because of time taken to prepare the cutting algorithm initially, by traversing the relevant part of the mesh and performing a planar slice.

In a normally-distributed dataset, we would expect over 95% of the frame rates to lie within 2 standard deviations of the mean. Therefore, assuming the data is of a symmetrical normal distribution, approximately 98% of the frame rates should lie at or above the mean minus 2 standard deviations. Table 7.2 shows that for all but one of the sample data sets, this value lies at or above 20 frames-

		Mean	-1 S.D.	-2 S.D.	-3 S.D.
Fast machine	Model 1	60.98	41.61	22.24	2.87
	Model 2	60.97	46.36	31.75	17.14
	Overall	60.98	40.75	20.52	0.28
Slow machine	Model 1	51.21	35.1	18.99	2.89
	Model 2	54.21	43.02	31.83	20.63
	Overall	52.74	37.51	22.28	7.06
Overall		53.19	37.59	21.99	6.39

Table 7.2: Mean frame rates minus 0, 1, 2 and 3 standard deviations

per-second. However, by manually counting, we find that the total percentage of frame rates above 20 frames-per-second to be 99.78% for the faster machine, 95.77% for the slower machine and 96.02% overall. This suggests that the sample data is not normally distributed, which may be due to the initial spikes in the frame time from the initialisation of the cutting process.

Overall, the planar slice cutting method appears to maintain high enough frame rates to be acceptable for use in an interactive manner at least 95% of the time. There may be some initial stutter as the cutting method initialises itself, but after this, the application operates within accepatable frame rates.

Chapter 8

Conclusions and Further Work

8.1 Research Aims

The main aim of this research was to find or develop an algorithm capable of simulating cutting through bone, suitable for use at the refresh rates required for an interactive podiatric surgical simulator. This goal has been achieved by researching conventional geometry cutting techniques, and developing a cutting algorithm based on the Boolean subtraction technique for polygonal meshes. We have implemented and tested this algorithm to determine its performance in different situations.

8.2 Algorithm Development

We have shown that none of the standard methods of cutting into geometery (mesh morphing, metaballs, CSG trees, Boolean subtraction on polyhedra) are suitable to simulate real-time interactive bone surgery. The most promising candidate, Boolean subtraction performed on two polyhedral meshes, has two major performance bottlenecks that prevent it from being suitable. These are its high degree of computational complexity and tendency to generate large number of irrelevant triangles, resulting in exponential slowdown.

We proposed a method of simulating real-time cutting designed to gain a similar end-result to a series of Boolean subtractions, but capable of operating in real-time interactive conditions. Our method uses a combination of planar slicing and controlled vertex separation. It performs a single initial slice along the plane defined by the path of the blade, and gradually reveals space either side of the slice over a series of frames.

In our algorithm, the space created by cutting into the mesh is revealed simply by moving pairs of vertices a fixed distance along the normal of the slice plane, in opposite directions. The pairs of vertices are moved apart as the user moves the blade through them. The pairs of vertices separated in this way are part of the cross-section of each side of the slice, and initially have identical position vectors. Since updating a vertex's position is a relatively computationally cheap operation, a large number of vertices can be updated per frame without any significant slowdown of the application.

8.3 Testing Results

In our initial phase of testing, we determined the result of several mesh modification optimisations. We demonstrated a clear benefit from pre-allocating extra space in vertex and index buffers. The benefit is much more pronounced in vertex buffers, which is likely due to the much larger quantity of data that is stored in them compared with index buffers. Increasing the percentage of buffer space allocated from 100% to 103% of the required space results in a large performance gain. Increasing the allocation percentage has been shown to result in further (but less substantial) gains for each percentage point up to 120%.

Using the index-buffer removal optimisation has been shown to sometimes result in a performance gain. For meshes with small numbers of polygons, a clear performance gain is seen, but for higher numbers of polygons, the performance with the optimisation actually decreases. This may be because selectively removing indices requires extra computations to determine the positions in the index buffer that correspond with a given triangle. Further research is needed to identify cutoff points for the total number of polygons in a mesh and the number of polygons being removed at which the optimisation becomes ineffective.

Testing the planar slice's performance determined that very little slowdown occurs when cutting through meshes with low numbers of polygons. For meshes with higher numbers of polygons, a number of frames from the initial instant of the cut are slowed down due to the planar slice taking place, but subsequent frames take place at full speed. As the planar slicing algorithm is designed to operate in chunks over a series of rendered frames, this slowdown is limited per frame and does not cause the application to freeze. A better solution would be to run the slicing algorithm on a separate thread where possible.

8.4 Future work

8.4.1 Improved Capping

The algorithm presented in this work has a number of drawbacks that, with further research, could be addressed to make it more suitable for simulating orthopaedic podiatric surgery. First, the capping algorithm (used to create a seal across the cross-section of geometry sliced into) could be significantly improved. Currently, the algorithm simply creates a 'triangle fan' containing all vertices on the cross-section, linked to a vertex generated at the cross-section's mid-point.

This approach works well with convex cross-sections, but does not cope well with nested surfaces — for instance, when cutting through a hollow tube, the cross-section will contain two concentric rings of vertices. For this to be capped correctly, only the area between the two rings of vertices should be filled with polygons. However, the current algorithm will typically create a 'spikey' surface that encloses the hollow section inside the innermost ring of vertices, and partially covers the space between the two rings. Furthermore, the current algorithm does not correctly cap concave cross-sections with vertices which do not have a direct line-of-sight to the mid-point vertex.

8.4.2 Improved use of Octrees

Second, all parts of the algorithm that involve searching for vertices or polygons meeting a given criteria (such as traversing the mesh to find adjacent polygons lying on a plane) use an octree to efficiently partition the search space. Although it is known that using an octree speeds up the search process somewhat, is unknown to what degree this is the case.

In addition, the octree has a number of parameters (maximum depth of the tree, maximum number of entries for leaf nodes, etc.), which can be used to fine-tune the performance of the search operation. It is unknown what values are most suitable for these parameters, which could be investigate in a future study. The use of other spatial partitioning schemes (such as using quadtrees or BSP trees) could be investigated to determine the most suitable scheme for optimising mesh cutting.

Furthermore, the current cutting implementation treats the octree as a 'black box' and simply uses it as an efficient way of searching for vertices and triangles in given areas, when required by part of the cutting algorithm. However, it may be possible to adapt parts of the planar slice cutting algorithm to use octrees (or other spatial partitioning schemes) more efficiently. For instance, regarding traversing a mesh along a given plane (see section 4.5.4), the current algorithm uses an octree to find triangles adjacent to the current triangle in the search and tests them for intersection with the plane. A better use of an octree in this case may be to test all the triangles in a given octant in the octree, and traverse to adjacent octants that lie on the plane.

8.4.3 Complete Simulator

Although enhancing the mesh cutting algorithm would be of some benefit, we feel that the most important future research to build on this work would be to progress to the next stage of developing an actual orthopaedic podiatric surgery simulator. This research would entail simulating actual surgical operations, with the simulator allowing the user to carry out all relevant steps in the operation. It would be necessary to obtain an accurate 3D foot model as a starting point for each operation being simulated, complete with the defects that the corrective surgery is intended to repair. All the appropriate actions required for correct performance of the procedure should be planned with the aid of a qualified surgeon, and a method of storing these actions in the simulator and comparing them with the user's actual performance should be devised.

In the finished simulator, users would be scored on their accuracy and speed. Facilities should be present to allow users to review their performance, compare it with others', see an explanation of how they can improve, and see the change in their performance over the course of many operations. Ideally, the system should support multiple trainees and an instructor, allowing the instructor to view the trainee's performance and provide feedback, and allowing the trainee to highlight areas to their instructor with which they require assistance.

References

Aftosmis, M., Berger, M. and Melton, J. (1998) **Robust and efficient Cartesian mesh generation for component-based geometry**. AIAA journal, 36(6):952–960. ISSN 0001-1452.

Agus, M., Giachetti, A., Gobbetti, E., Zanetti, G. and Zorcolo, A. (2002) **A multiprocessor decoupled system for the simulation of temporal bone surgery**. Computing and Visualization in Science, 5(1):35–43. ISSN 1432-9360.

Agus, M., Giachetti, A., Gobbetti, E., Zanetti, G. and Zorcolo, A. (2003) **Real-time haptic and visual simulation of bone dissection**. Presence: Teleoperators & Virtual Environments, 12(1):110–122. ISSN 1054-7460.

Allan, J., Wyvill, B. and Witten, I. (1989) **A methodology for direct manipulation of polygon meshes**. In Proceedings of CG International. pp. 451–469.

Bell, P. C. and O'Keefe, R. M. (1987) **Visual Interactive Simulation — History, recent developments, and major issues**. SIMULATION, 49(3):109 –116.

Bill, J. (1994) **Computer sculpting of polygonal models using virtual tools**. Ph.D. thesis, University of California.

Blackman, S. (2005) **Serious games... and less!** ACM Siggraph Computer Graphics, 39(1):12–16. ISSN 0097-8930.

Blinn, J. (1982) **A generalization of algebraic surface drawing**. ACM Transactions on Graphics (TOG), 1(3):235–256. ISSN 0730-0301.

Bloomenthal, J. (1988) **Polygonization of implicit surfaces**. Computer Aided Geometric Design, 5(4):341–355. ISSN 0167-8396.

Blythe, D., Grantham, B., Kilgard, M., McReynolds, T. and Nelson, S. (1999) **Advanced graphics programming techniques using OpenGL**. Course Note# 29 of SIGGRAPH, 99.

Chen, P., Barner, K. and Steiner, K. (2005) **A spring-net deformable model for surgery simulation with haptic feedback**. In ACM SIGGRAPH 2005 Posters. ACM, p. 111.

Choi, K., Sun, H., Heng, P. and Cheng, J. (2002) **A scalable force propagation approach for web-based deformable simulation of soft tissues**. In Proceedings of the Seventh International Conference on 3D Web Technology. ACM, pp. 185–193. ISBN 1581134681.

De, S. and Lim, Y. (2006) **A meshfree computational methodology for surgical simulation**. Journal of Biomechanics, 39(1):214. ISSN 0021-9290.

Delingette, H. (1998) **Toward realistic soft-tissue modeling in medical simulation**. Proceedings of the IEEE, 86(3):512 –523. ISSN 0018-9219.

Delp, S., Loan, J., Hoy, M., Zajac, F., Topp, E. and Rosen, J. (2002) **An interactive graphics-based model of the lower extremity to study orthopaedic surgical procedures**. Biomedical Engineering, IEEE Transactions on, 37(8):757–767. ISSN 0018-9294.

Delp, S., Loan, P., Basdogan, C. and Rosen, J. (1997) **Surgical simulation: An emerging technology for training in emergency medicine**. Presence: Teleoperators and Virtual Environments, 6(2):147–159. ISSN 1054-7460.

Eberly, D. (1998) **Triangulation by ear clipping**. Geometric Tools, LLC. Available from: <`http://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf`>.

Eberly, D. H. (2007) **3D game engine design**. Gulf Professional Publishing. ISBN 9780122290633.

Edmond, J., Wiet, G. J. and Bolger, L. B. (1998) **Virtual Environments: Surgical Simulation in Otolaryngology**. Otolaryngologic Clinics of North America, 31(2):369–381. ISSN 0030-6665.

Foley, J. D. (1995) **Computer graphics: principles and practice**. Addison-Wesley. ISBN 9780201848403.

Fosse, E., Rtnes, J. S., Kaasa, J., Westgaard, G., Eriksen, E. M., yvind Hvidsten, P., Strm, K., Srhus, V., Halbwachs, Y. and Elle, O. J. (2001) **Realism in surgical simulators with free-form geometric modeling**. International Congress Series, 1230:1032–1037. ISSN 0531-5131.

France, L., Lenoir, J., Angelidis, A., Meseure, P., Cani, M., Faure, F. and Chaillou, C. (2005) **A layered model of a virtual human intestine for surgery simulation**. Medical Image Analysis, 9(2):123–132. ISSN 1361-8415.

Fuchs, H., Kedem, Z. and Naylor, B. (1980) **On visible surface generation by a priori tree structures**. ACM Siggraph Computer Graphics, 14(3):124–133.

Gouraud, H. (1971) **Continuous shading of curved surfaces**. IEEE transactions on computers, pp. 623–629. ISSN 0018-9340.

Haluck, R. S., Marshall, R. L., Krummel, T. M. and Melkonian, M. G. (2001) **Are surgery training programs ready for virtual reality? a survey of program directors in general surgery**. Journal of the American College of Surgeons, 193(6):660–665. ISSN 1072-7515.

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

Jansen, F. (1987) **Solid modelling with faceted primitives**. Delft University Press.

Kalvin, A. and Taylor, R. (2002) **Superfaces: Polygonal mesh simplification with bounded error**. Computer Graphics and Applications, IEEE, 16(3):64–77. ISSN 0272-1716.

Lane, J., Carpenter, L., Whitted, T. and Blinn, J. (1980) **Scan line methods for displaying parametrically defined surfaces**. Communications of the ACM, 23(1):23–34. ISSN 0001-0782.

LeBlanc, A., Kalra, P., Magnenat-Thalmann, N. and Thalmann, D. (1991) **Sculpting with the ball and mousemetaphor**. In Proc. Graphics Interface. Citeseer, vol. 91, pp. 152–159.

Li, Z. (2004) **The potential of America's Army, the video game as civilian-military public sphere**. Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Comparative Media Studies.

McEachern, M. (2008) **Force To Be Reckoned With**. Computer Graphics World, 31(9):20–26. ISSN 0271-4159.

Meier, L., Sigal, E. and Vitale, F. R. (1985) **Proceedings of the 17th Conference on Winter Simulation**. San Francisco, California, United States: ACM, pp. 558–564. ISBN 0-911801-07-3.

Mitchell, C. L., Fleming, J. L., Allen, R., Glenny, C. and Sanford, G. A. (1958) **Osteotomy-Bunionectomy for Hallux Valgus**. J Bone Joint Surg Am, 40(1):41–60.

Morris, D., Sewell, C., Barbagli, F., Salisbury, K., Blevins, N. and Girod, S. (2006) **Visuohaptic simulation of bone surgery for training and evaluation**. Computer Graphics and Applications, IEEE, 26(6):48–57. ISSN 0272-1716.

Morris, D., Sewell, C., Blevins, N., Barbagli, F. and Salisbury, K. (2004) **A collaborative virtual environment for the simulation of temporal bone surgery**. Medical Image Computing and Computer-Assisted Intervention–MICCAI 2004, pp. 319–327.

Neubauer, A., Wolfsberger, S., Forster, M., Mroz, L., Wegenkittl, R. and Buhler, K. (2004) **Proceedings of the conference on Visualization '04**. IEEE Computer Society, pp. 513–520. ISBN 0-7803-8788-0.

Occena, L. and Tanchoco, J. (1988) **Computer graphics simulation of hardwood log sawing.** Forest Products Journal, 38(9):72–76. ISSN 0015-7473.

O'Kane, C. and Kilmartin, T. E. (2002) **The rotation Scarf and Akin osteotomy for the correction of severe hallux valgus**. The Foot, 12(4):203–212. ISSN 0958-2592.

O'Kane, C. C. and Kilmartin, T. E. (2007) **Orthopaedic surgery and podiatric surgery: will you get the same operation?** Podiatry Now. Available from: <`http://findarticles.com/p/articles/mi_6857/is_8_10/ai_n28496709/`>.

Okoshi, T. (1980) **Three-dimensional displays**. Proceedings of the IEEE, 68(5):548–564.

Parent, R. E. (1977) **A system for sculpting 3-D data**. In Proceedings of the 4th annual conference on Computer graphics and interactive techniques. New York, NY, USA: ACM, SIGGRAPH '77, pp. 138–147.

Petersik, A., Pflesser, B., Tiede, U., Höhne, K. and Leuwer, R. (2002) **Realistic haptic volume interaction for petrous bone surgery simulation**. Computer assisted radiology and surgery, Proc. CARS 2002, pp. 252–257.

Pflesser, B., Tiede, U., Höhne, K. and Leuwer, R. (2000) **Volume based planning and rehearsal of surgical interventions**. In Computer assisted radiology and surgery, proc. CARS. Citeseer, vol. 1214, pp. 607–612.

Phong, B. (1975) **Illumination for computer generated pictures**. Communications of the ACM, 18(6):311–317. ISSN 0001-0782.

Pool, J., Lastra, A. and Singh, M. (2009) **Energy-precision tradeoffs in mobile Graphics Processing Units**. In Computer Design, 2008. ICCD 2008. IEEE International Conference on. IEEE, pp. 60–67. ISSN 1063-6404.

Ramis, F. J., Palma, J. L. and Baesler, F. F. (2001) **Proceedings of the 33nd Conference on Winter Simulation**. Arlington, Virginia: IEEE Computer Society, pp. 1401–1404. ISBN 0-7803-7309-X.

Reeves, W. (1983) **Particle SystemsmA Technique for Modeling a Class of Fuzzy Objects**. ACM Transactions on graphics, 2(2):91–108.

Requicha, A. (1980) **Representations for rigid solids: Theory, methods, and systems**. ACM Computing Surveys (CSUR), 12(4):437–464. ISSN 0360-0300.

Requicha, A. and Tilove, R. (1978) **Mathematical foundations of constructive solid geometry: General topology of closed regular sets**. Production Automation Project Technical Memorandum TM-27a, University of Rochester, New York.

Sørensen, T. S. (2005) **ACM SIGGRAPH 2005 Electronic Art and Animation Catalog**. Los Angeles, California: ACM, pp. 295–295. ISBN 1-59593-101-5.

Squire, K. and Jenkins, H. (2003) **Harnessing the power of games in education**. Insight, 3(1):5–33.

Stapleton, A. J. (2004) **Serious Games: Serious Opportunities**. Paper presented at the Australian Game Developers' Conference, Academic Summit, Melbourne, VIC. Available from: <`http://andrewstapleton.com/?page_id=21`>.

Szirmay-Kalos, L. and Márton, G. (1998) **Worst-case versus average case complexity of ray-shooting**. Computing, 61:103–131. ISSN 0010-485X. 10.1007/BF02684409.

Turk, G. (1992) **Re-tiling polygonal surfaces**. Computer graphics, 26(2):55–64. ISSN 0097-8930.

Vidal, F. P., Chalmers, N., Gould, D. A., Healey, A. E. and John, N. W. (2005) **Developing a needle guidance virtual environment with patient-specific data and force feedback**. International Congress Series, 1281:418–423. ISSN 0531-5131.

Ward, G. (1992) **Measuring and modeling anisotropic reflection**. In Proceedings of the 19th annual conference on Computer graphics and interactive techniques. ACM, pp. 265–272. ISBN 0897914791.

Wickens, C. (1989) **Three-dimensional displays: Perception, implementation, and applications**. Tech. rep., CREW SYSTEM ERGONOMICS INFORMATION ANALYSIS CENTER WRIGHT-PATTERSON AFB OH.

Wyvill, B. and van Overveld, K. (1996) **Polygonization of implicit surfaces with constructive solid geometry**. International Journal of Shape Modeling, 2:257–274. ISSN 0218-6543.

Wyvill, B. and van Overveld, K. (1997) **Visualizing and Animating Implicit and Solid Models**. Visualization and modeling, p. 403.