University of Nevada, Reno

Conversion of Thin Surface Solids to BSP Solid Sets

With Visualization and Simulation Applications

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Science.

by

Jeremy W. Murray

Dr. Frederick C. Harris, Jr., Thesis Advisor

August 2008

University of Nevada, Reno
Statewide • Worldwide

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

JEREMY W. MURRAY

entitled

Conversion Of Thin Surface Solids To BSP Solid Sets

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Frederick C. Harris, Jr., Ph. D., Advisor

Yaakov L. Varol, Ph. D., Committee Member

Danny L. Taylor, Ph. D., Graduate School Representative

Marsha H. Read, Ph. D., Associate Dean, Graduate School

August, 2008

Abstract

The advantages of modern game engines in the simulation field are numerous, but simulations need to accurately represent physical data. Real world objects can be scanned and recorded as sets of point data. Surface reconstruction from point data is only one step needed to visualize it in a virtual environment - static objects can take advantage of significant optimizations by converting the surface faces into convex solids and storing them in a binary space partition tree. During this conversion, other physical characteristics of the data can be determined for use in the simulation. This thesis presents methods for conversion and data extraction, algorithm backgrounds and step-by-step procedures from spatial data through active simulation.

Acknowledgements

Thanks are due to Dr. Frederick Harris for his long-term support and direction, Dr. John Frederick for fostering creative solutions, Danny Taylor for providing complex input data, John Jusayan for his innovative encouragement, and several friends for allowing long running processes on their expensive hardware.

Thank you to my family, especially my mother for her resolve and giving nature and my grandmother for her continued displeasure of my procrastination. Finally, thank you to my wife, Arianne, for her patience, love and for our little hard deadline, Aelliana, who made corrections on her very own keyboard.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Virtual reality (VR) is most commonly experienced in commercial game engines. Today, games are powered by tool sets using highly optimized routines for both virtual physics and display of the virtual world. The power of these engines can be extended to more practical use in fields currently dominated by physical modeling, specialty hardware training, hands-on experience and specialty simulations. This would allow safer, less expensive and easier to distribute solutions to problems like flight training, mining safety, military training, and driving instruction.

Many current simulations are written from scratch for each application. The use of a commercial game engine can shorten development time significantly. Routines for rendering and physics are provided, allowing the software developer to reduce their scope. In general, simulation developers are most concerned with a finite set of specific requirements. A flight simulator, for example, must provide realistic aircraft, flight dynamics, cockpit layout, radio communication, air traffic, and ground landmarks. These requirements can be coded with as much detail as needed while the game engine handles the rest of the application.

The benefits of a framework are especially evident when all of the project developers are well versed in one discipline, such as a team of hydrologists who want to create a dam simulation but are not experienced in rendering effects, user interfaces, network communication, or hardware compatibility. By using a game engine, a development team can be much smaller and undiversified while still producing a strong final product.

A game engine is a toolset providing the complete back-end to an interactive game. It is usually comprised of a graphical output or renderer, a physics engine, a sound engine, a networking layer, and an input handler, along with a set of logic routines defining the game parameters and rules.

The term game engine was first coined in the 1990s and referred to the original first-person shooter (FPS) games, like *Doom* and *Quake*, but the concept of a reusable library is an old one. A company could spend the resources needed to code a single game and would be able to reuse most of the programming to create a sequel. The cost and time-to-market of any follow-up projects is reduced significantly. Studios that invest in creating engines see an increase on their return through licensing to other companies - an attractive alternative for many smaller design houses, where hiring the talent needed to make a competitive engine might not be feasible.

By using a game engine for a project, the developer begins far ahead of from-scratch development. They will have a full working example program that they can learn from and edit. They will have documentation to follow and developers to query with issues. They will not need to write code in a completely foreign discipline, keeping graphics people from writing distributed networking code and the like.

Game engines are also designed with one or more target platforms. They are designed to work consistently on that platform, and will already have coded in the necessary user interface considerations needed for interoperability. They will usually include ways to modify their detail so as to scale to different levels of hardware, allowing a much larger audience to use the final program.

Developers using game engines also begin development using a base that is near the level of the existing standard of the day. With the shortened development time, this lessens the potential for release of a product that is already considered dated at release. Many engines also are constantly refining to stay even with competitor advances, providing updates at little or no cost to current licensees. While this leaves the game makers the freedom to create their games with as little overhead as possible, it has also accelerated the technology advances in the field. The studios creating the engines are hiring specialized programmers to focus on improvements and the results of their developments have been spectacular.

Modern game engines have come far since their original inception. Current engines can handle many thousands of objects on screen, deliver believable physics and world interactions, as well as handle the networking tasks to bring hundreds of people into a shared space. Scenes can display almost a gigabyte of texture and vertex data and users can communicate in real-time by voice and text. Environments can be constructed in startling accuracy, and can furthermore be designated by material type and behave as such. Modern engines know wood floats and burns, glass breaks, and water puts out fire (while producing steam). Liquids fill their containers and the wind moves leaves and grass. Underneath is complex math, but the interface to the programmer or artist is highly simplified.

Physical interaction between objects can extend from simple collisions to programmatic destruction. Liquids will move when touched, stick to objects that touch them and leave traces as the object continues to move. Metal can be dented. Paint can be scraped off. Fires can start in a wastebasket and extend to an entire structure while providing light to the area around it and sending sparks and smoke into the air. The level of detail in calculating interactions between objects has never been greater. Once implemented, a single physics model can be reused in many situations to provide new game mechanics. The addition of friction, for instance, could be used to interact with sliding objects like in hockey and more realistic driving by emphasizing the interaction between the tires and a surface. The addition of fluid dynamics gives way to splashing water, realistic waves and rainfall, and possibly boating and swimming. As the engines grow more complex, the possible interactions continue to grow.

An area where game engines lack, though headway has recently been made, is in the area of simulation. A game engine still strives for a smooth user experience over an accurate one. Calculations are not designed to be exact, but fast. Interactions may be done to a lesser degree of detail to reduce calculation time. In many cases, the extra accuracy would not be noticed or missed. While this is acceptable in a game environment, when using an engine for simulation it may not be. In order to allow options for correcting this, many engines have adopted a structured approach to this problem. Many will allow licensees to code modules and attach them to the engine. This allows simulation developers to add realism and structure in areas where they feel it is needed while allowing the engine to continue to handle all other parts of the application. This method again leaves the programmers to concentrate only on their specific areas of expertise and leave the rest to others who know better.

And while game engines may provide the toolset to create an application, they do not provide much in the way of objects to use in that application. When developing a new simulation, the largest barrier after coding the required accurate physics rules is the creation of objects to fill the simulation. This is why many companies using game engines will hire many more artists than programmers.

What simulation developers lack in art they can make up for with data. The display of accurate physical data in VR is one of the most important factors in the acceptance of that VR by a participant. A simple method for importing real environmental data into the virtual space can speed simulation development while increasing its realism. This project is a step toward this last point and has three main goals: the conversion of spatial data as a point cloud into a tessellated thin-surface solid, the conversion of a thin-surface solid into a binary space partition (BSP) solid set, and the display and evaluation of the BSP solid set in commercial game engines.

Chapter 2 gives a background on the research used in this project. It begins with data acquisition, where current methods of spatial sampling are examined. It discusses advantages to the individual methods with respect to this project, along with theory for merging data from multiple data sets. Once a point cloud is available, the conversion to a thin-surface solid can be done using any of many published methods. Tessellation and related algorithms are presented, and the two methods used in this project are examined in detail. Procedures for evaluating the final mesh and for extracting other useful information are discussed. BSP trees - a data structure with strong ties to current game engines - are discussed. The input format is also presented as related to a thin-surface solid.

Chapter 3 covers the original work of the project. The reduced medial axis transform is introduced as a modification of the medial axis transform. Volumetric thresholding is introduced and its output is examined. Finally, the thin-surface solid to BSP solid set conversion is discussed.

Chapter 4 describes the conversion process from start to finish. Individual game engines are discussed, along with their specific input requirements. BSP compiler options, errors, and resolutions are demonstrated. The final output as an environment in the engine is shown and evaluated in multiple engines, while the individual engines are discussed for their potential application to simulation issues. This chapter also presents common conversion issues and their resolutions as well as optimizations and parallelization opportunities.

Chapter 5 contains several case studies of conversions and Chapter 6 discusses the overall project strengths, opportunities and possible paths to continue this work.

Chapter 2 Background

The creation of a mesh from a point cloud is a long-standing problem. Several different approaches have been made in this area, most notably Delaunay tessellation. While this is the classic, verifiable approach, there have been advances in recent years that can speed the mesh construction. In this chapter, different input methods are described along with methods for registration of their point sets. Tessellation output post-processing is shown, both for mesh refinement and data extraction. Binary space partition trees and their application in graphics and physics are also discussed.

2.1 Input Data Acquisition

There are currently many different methods for sampling the true space coordinates of a single point in space. These include direct sampling with a laser rangefinder or global positioning system and indirect sampling using multiple 2D representations from different viewpoints (recently discussed in [24] and [38]). While these methods to find individual points are very accurate, the process of creating a 3D mesh from the sampled points can be complex.

2.1.1 3D Scanning

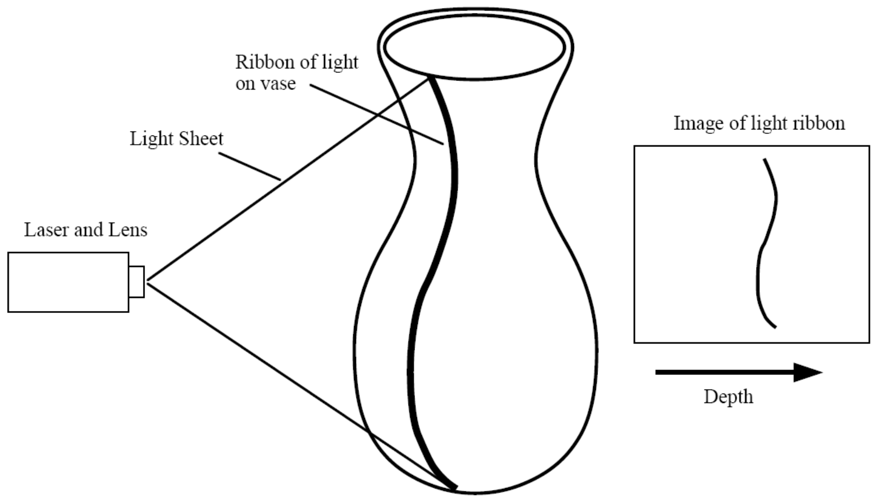

Figure 2.1: 3D scanning example [44].

Currently, the easiest method for creating a solid from sampled coordinates is to take the samples in a set order that implies the overall surface. This is best represented by a 3D scanner, as in Figure 2.1. Using a laser rangefinder, a static object is scanned as a set of connected points. The scanner circles the object, keeping the laser parallel to the vertical axis. Starting at a low height, the scanner moves the laser vertically across the object taking individual samples at set intervals. Since the samples are taken in order, they can be connected by edges. In the example, one full rotation of these points and edges comprise a single longitudinal set. Once a set has been completed, the scanner rotates a small amount and repeats the scanning process. By sampling the object at an interval independent of the object, longitudinal sets can be connected by edges across adjacent points taken at similar intervals to form latitudinal sets. This creates an irregular mesh of quads over the entire model with inconsistent resolution.

Problems with this scanning method arise when the object has irregularities, such as occlusion. Occlusion is where part of the object obscures itself from a certain perspective. Since the laser cannot penetrate the object, the surface that is hidden is left unsampled. If the mesh were to uniformly cover every recorded point, the mesh would incorrectly span this occluded area. Some occlusions can be detected by setting a segment length threshold as a function of the scanner resolution. If exceeded, the point is still recorded but the segment is discarded as in Figure 2.2.

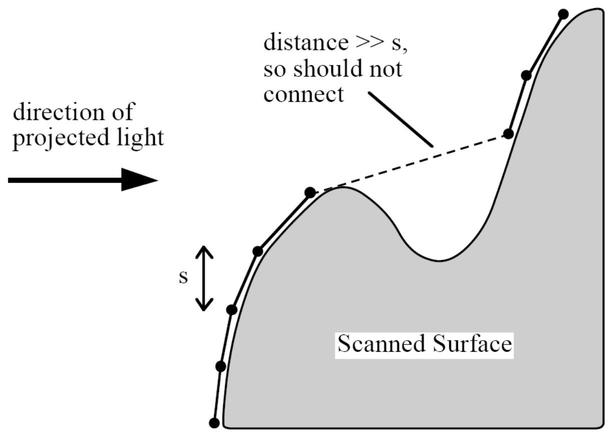

Figure 2.2: Occlusion example - note the excluded segment [44].

While this helps to prevent erroneous meshes, the occluded area still has not been scanned. The solution to this problem is to rescan the object from a different perspective to produce a second irregular mesh. Use of a calibrated beam splitter allows sampling of several points from many angles at each interval point, and is significantly faster than multiple longitudinal rotations. Newer 3D scanners use a combination of object rotation and beam-splitting, producing a set of meshes - one for each perspective. Unfortunately, these individual data sets cannot be combined easily. The meshes must be related to each other spatially and in complex areas the individual meshes may overlap and show inconsistent data.

2.1.2 Registration

First the spatial relationships between the various data must be determined. This process is called *registration*, and will produce a set of spatial data that can transform the coordinate data of all inputs into a common space. This is a classic problem - [13, 14, 16, 36] are good sources for background and methods.

Reference points are a set of paired coordinates shared between two sets of input data. Each pair of points specifies a common point in the two independent sets. Any number of pairs helps to speed the registration process, but a full relation requires a minimum of four non-coplanar pairs with no more than two collinear points. If the spaces are assumed not to be skewed (the axes are perpendicular), this can be reduced to three pairs. When registration is complete, the output is a set of reference points showing the input meshes relationship. See Figure 2.3 for an example of registration.

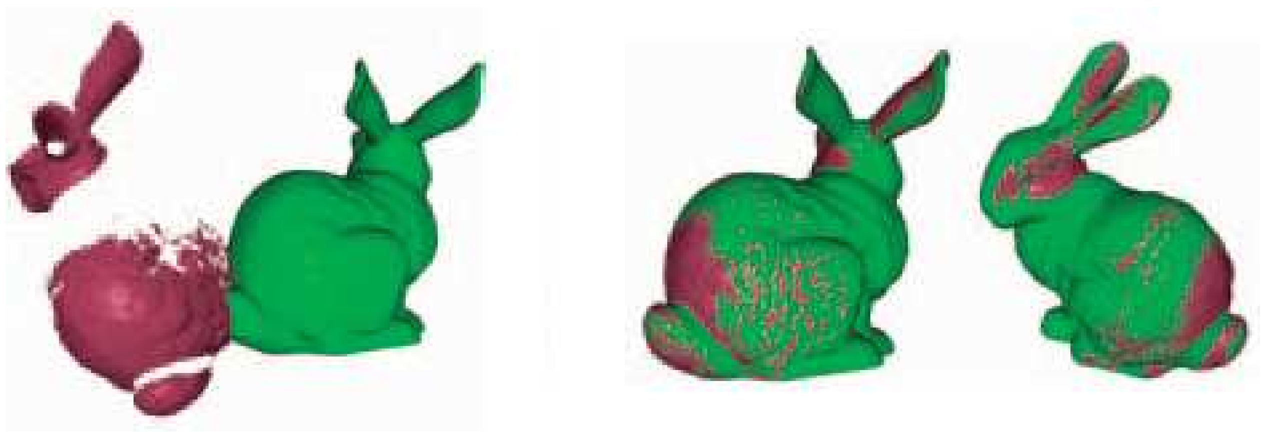

Figure 2.3: Aligning related spatial data is called registration [36]. (Stanford Bunny courtesy of the Stanford Computer Graphics Laboratory)

2.1.3 Other Data Sources

The data set can be made of mixtures from different input sets, as long as the data has been registered. For simulations needing to display accurate depictions of places, buildings, and objects, data can be imported from survey data, blue print data, elevation data, and other readily attainable documents. This shortens the data acquisition phase of modeling, as most structural data requires highly accurate plans during construction and many require as-built plans from post-construction surveys. Reuse of these typically available materials is an advantage to this method. A disadvantage is that these data sets tend to be very large, with many thousands of points for even simple plans. There is also no current industry standard for the plans other than paper, so it may not be feasible to use them. While topographies are normally 3D (or 2D with contour lines depicting elevation), only very recent construction might have 3D data and it may not be easily obtainable. Debevec *et al.* points out how architecture specifically has several optimizations when forming meshes from photographs, and the results are quite detailed [24].

2.2 Mesh Combining Algorithms

Once registered, combining the meshes presents several challenges. Different methods will question the validity of individual points, expecting possible error in the sampling process. Others expect that every point is guaranteed to lie on the surface of original object.

If range data was used for the individual meshes, the addition of an extra piece of information can create a very well structured final mesh. By adding knowledge of the perspective point with respect to the sampled data, the algorithm proposed by Curless *et al.* in [23] (based on Turk and Levoy [44]) can be used. By creating an implicit function that represents the weighted signed distance of a calculated point to the nearest ranged surface along the line of sight to the sensor, a new surface point can be calculated (see Figure 2.4). Weights are determined based on input method, and are usually a function of sampling density or sampling error, such as the reduced resolution and increased optical error as the incident angle of a laser rangefinder approaches tangential intersection.

Without the knowledge of the perspective point, this method can still be used to combine mesh points. The average of the neighbor normals can be used to determine the perspective

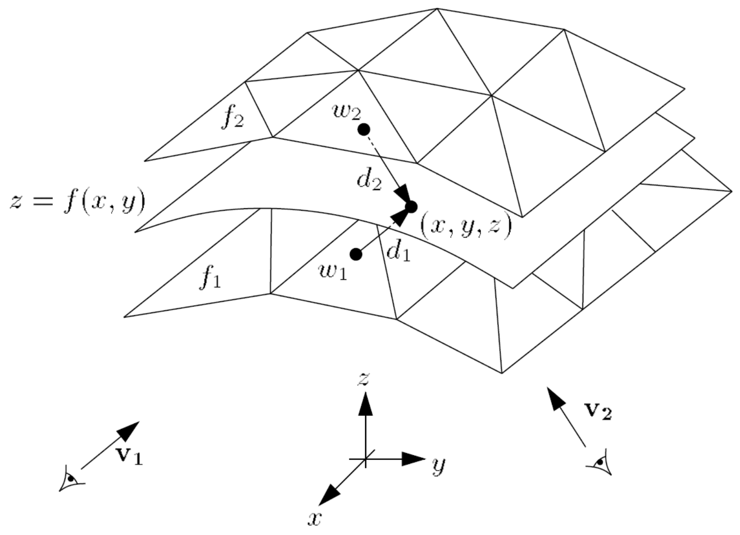

Figure 2.4: Two range surfaces, f_1 and f_2, are tessellated range images acquired from directions v_1 and v_2. The possible range surface, $z = f(x, y)$, is evaluated in terms of the weighted squared distances to points on the range surfaces taken along the lines of sight to the sensor. A point, (x, y, z), is shown here being evaluated to find its corresponding signed distances, d_1 and d_2, and weights, w_1 and w_2 [23].

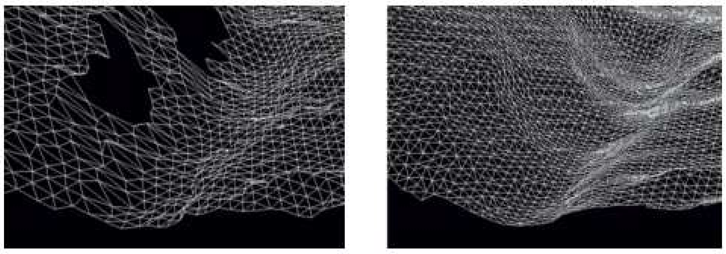

Figure 2.5: Left, a single range surface with occlusions. Right, a merge of multiple scans. Note the greatly increased resolution [33].

vector and the weights can either be distance based or mesh-position based.

Another method for combining multiple meshes is called zippering, first described in [44] and illustrated in Figure 2.6. A border is chosen, usually either the extreme edge of one mesh or the average of their overlap, as a series of edges in an existing mesh. Triangles that are fully over this border are discarded from both meshes. The triangles that intersect the border are cut at the border and new triangles are created using the intersection points and vertices of the border. Zippering may not be ideal, as overlapping range information is discarded when it could be used for refinement, but in many sampling methods the confidence is lowest in the data recorded near extremities. This factor should be considered when choosing the zippering border selection method.

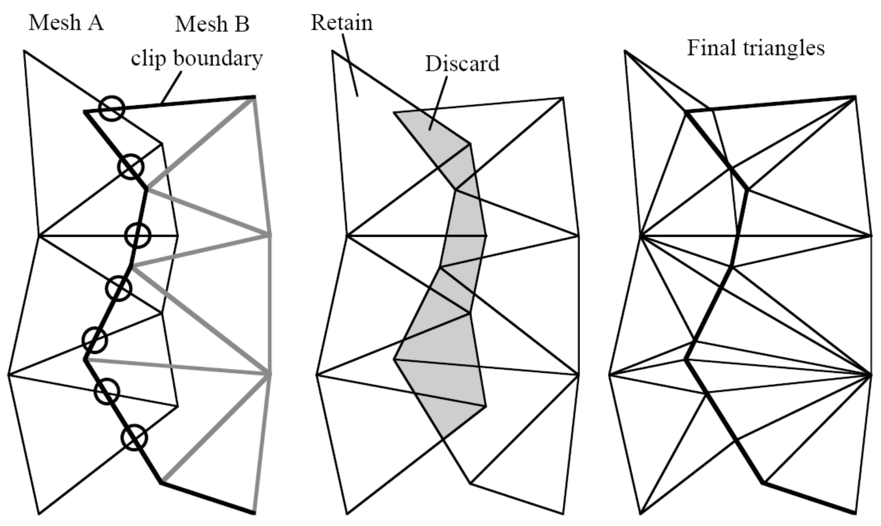

Figure 2.6: To zipper the meshes, the boundary of Mesh B is used as a clip boundary. Intersections of A with this boundary are used as new intersection points. The portion of A inside B is discarded, and new polygons are created in B using the new intersection points and new polygons are created in A using the new intersection points and the vertices of the clip boundary [44].

Turk and Levoy's consensus geometry [44] can be used to resolve errors in these merging methods. Especially when new points are created, deviation from the original sampling can be seen as errors in the geometry. Using a method similar to Figure 2.4, each point is re-evaluated compared to the original mesh inputs. A normal is calculated based on the average of the neighboring vertex normals (in a specified range) and a weighted average of the intersection of that normal with each of the input meshes is formed. The weighting can be influenced by the measure of confidence in each input set, if available.

Finally, another method to resolve multiple meshes is to retain only the individual 3D points, combine the points of all sampled meshes and rebuild a single mesh from these points. This process of mesh creation from points is called tessellation. Since it requires only points as input, we can obtain suitable data from a wide variety of sources. It also prevents any erroneous edge data in input meshes, but without any edge information the calculation takes much more time, comparatively.

2.3 Surface Tessellation

Finding the surface defined by a point set is a classic problem, with solutions first appearing in the late 1800s. Delaunay tessellation is a commonly accepted solution. At its heart, it is based on a variation of the nearest neighbor approach. All space is evaluated to determine the closest point, forming a set of regions known as a Voronoi diagram. Neighboring regions are connected by an edge, and this set of edges forms a Delaunay triangulation. For a set of points in general position (in 2D, no three points are collinear and no four points lie on the same circle), this method produces a unique mesh. It also follows several mathematical traits, such as the circumcircle defined by the three points of a face encompasses no other points. This finds discrete faces that do not overlap while maintaining a near fit to the ideal surface.

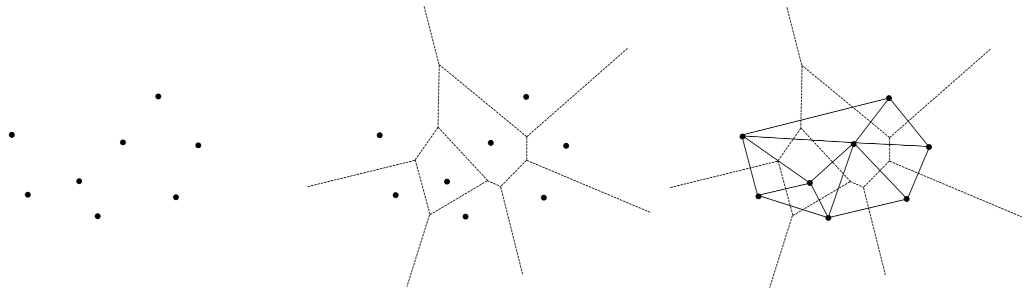

Figure 2.7: A point set, the Voronoi diagram and the complementary Delaunay triangulation in 2D [8].

While simply illustrated in 2D as in Figure 2.7, this method can be extended to any number of dimensions. The computation time required for 3D triangulation can be significant for many input points, as the entire set is evaluated for each face. On larger data sets (above one million points), the complexity is too large for timely execution.

2.4 Regioning and Parallel Surface Tessellation

As data sets become larger, memory requirements for classical Delaunay surface reconstruction become an obstacle. As with any other data-intensive applications, swapping to disk drastically increases runtime. Since Delaunay tessellation requires calculations on all points in the data set to guarantee the fit of the surface, and since the complexity of the surface calculation is quadratic (Attali *et al.* shows a best case N log N for even distribution over smooth surfaces [7]), extremely large data sets cannot be evaluated.

Subdivision of the original point set can reduce the number of points evaluated at all steps during surface reconstruction, explored in depth by Dey *et al.* [25]. By splitting the data set into smaller pieces, two advantages can be realized. First, the calculation can proceed much quicker, as the split reduces the computational complexity. Second, the individual areas can be calculated without full knowledge of the points contained in other areas. This leads to a natural parallel implementation.

The main data set is split into pieces following a similar setup to binary space partition trees, discussed in Section 2.7. The split is done orthogonally and in such a way that the boxes formed each hold approximately the same amount of total space. Each of these new boxes is further sub-divided until each box contains a number of points beneath a threshold. The resulting structure is an octree.

These boxes will be individually meshed, but each box does not contain a full data set. In order to accurately form faces on all points inside one of these boxed regions, the algorithm must use data from points within boxes that neighbor the current box. It does not need to use all neighbor box points, but can instead use a subset. Depending on the uniformity of the sample data, this neighbor threshold can be adjusted. The neighbor points are added to the smaller data set and labeled as border points.

When meshing, the data set is treated as a standard Delaunay set with two exceptions: the points are not guaranteed to be all on the same surface within a box, and no faces are kept that are comprised of three border points. Faces that are comprised solely of border points will also be produced in other box calculations and should not be duplicated.

An additional step is required when merging the individual meshes, involving comparison and reconciliation of any discrepancies in the border faces of adjacent boxes. With large enough border regions, this is a small calculation and any errors are discovered and corrected during manifold tests. Figure 2.8 shows the final results of this process, colorized to illustrate the individual partition volumes.

This simple regional division allows for parallelization of larger data sets and reduces the hardware requirements (especially RAM) to workstation levels. Adjusting the point

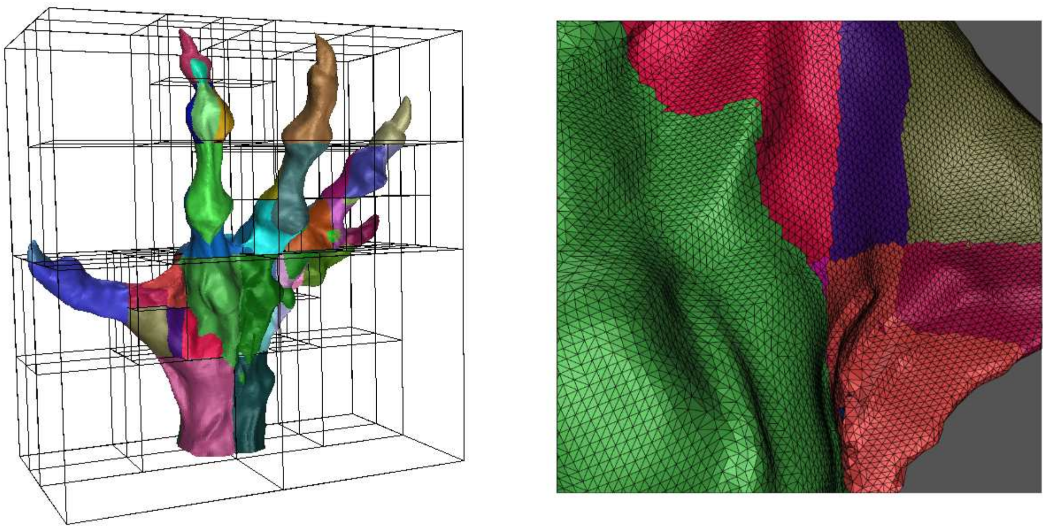

Figure 2.8: By partitioning the data into smaller pieces, meshing is much faster [25].

threshold and neighbor point thresholds can further reduce the computation, with the optimal settings using most of the workstation RAM and maintaining enough neighbor points to create accurate border faces. Output time is drastically less when compared to meshing the full data set. Run-times on standard desktop hardware using data sets in the area of a million points are reduced from three or more hours to under a half-hour with identical output meshes.

The main variance in this method from Dey is in the subdivision. After numerous meshing issues when using standard octree divisions (a side-effect of irregular density in the input sets), the method was modified to subdivide while taking into account the number of points in each subdivision. The final data structure used is a three dimensional k-d tree (see Figure 2.9). The k-d tree is introduced in [11] and discussed later in [12] by Bentley. Changing the splitting plane in this way makes the tree more balanced and the operations on it are faster, but otherwise calculation methods are similar.

2.5 Tessellation Post-processing

Once the mesh has been formed from the input data, some post-processing will insure that it is valid and well-formed. This section discusses methods for evaluation and correction of errors in the mesh.

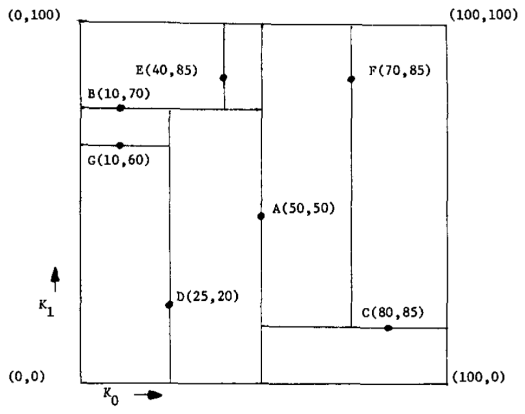

Figure 2.9: For non-uniform mesh density, a k-d tree provides better partitioning than an octree [11, 12].

2.5.1 Mesh Holes

While Delaunay tessellation will not leave holes in the geometry, other methods might. It is especially important when using range scans exclusively to check the final mesh for holes, especially if occluded areas are not adequately sampled. The method does no harm on a well-formed mesh, so it also acts as a sanity check for all methods.

For all edges in the final mesh, check to see if they are members of exactly one triangle. These edges are labeled as borders. Once all edges have been classified, search the set of border edges for cycles. For each set of cycles, fill the interior with a rough triangulation, then rework the triangulation to best fit the neighboring triangles (see Figure 2.10). This is discussed in more detail by Liepa in [34]. Other potential algorithms are surveyed by Botsch in [17], which includes information about other degenerate cases, such as isles or islands.

2.5.2 Topological Evaluation

Adamy *et al.* discuses a method for testing for topologically-correct surface reconstruction in [1]. This addresses many of the issues where mesh formation on close-proximity surfaces with inconsistent resolution incorrectly link the two surfaces. It does so by enforcing the idea of an umbrella condition - each vertex has a set of vertices that share an edge with itself (called neighbor points), and these vertices must connect to each other to form a cycle. The effect of this in the normal case is a triangle fan, shaped like a parasol or umbrella.

There are three possible errors to be detected, as illustrated in Figure 2.11: type 1 errors

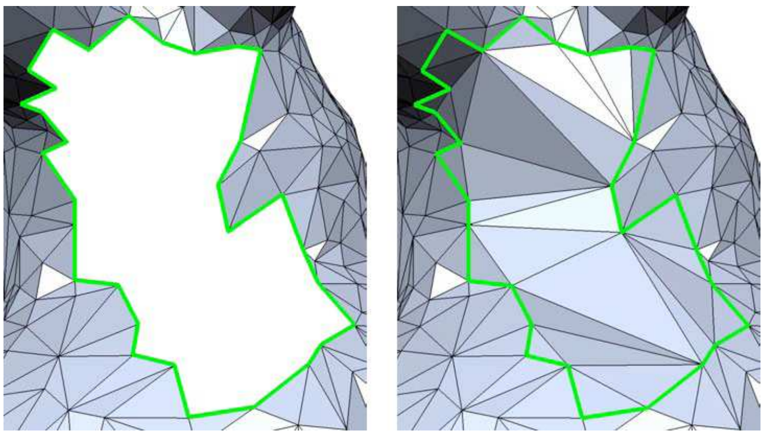

Figure 2.10: After detecting a border edge cycle, the inferred hole can be filled [17].

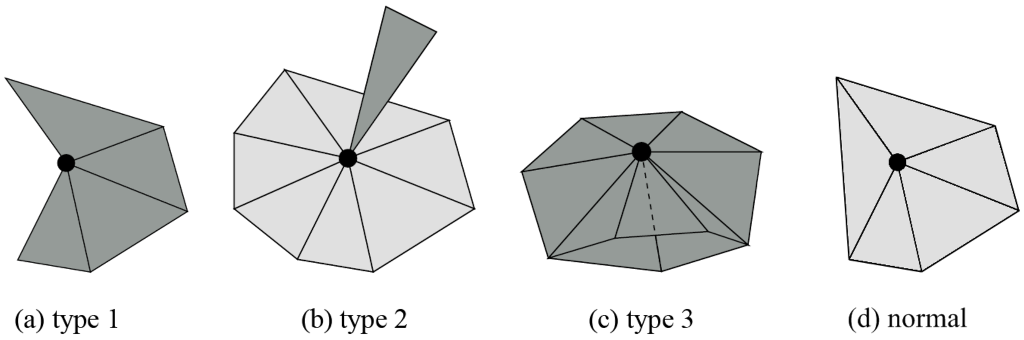

Figure 2.11: Four cases possible in umbrella topology evaluation [1].

are where the neighbor points do not form a cycle, type 2 errors contain a neighbor cycle where some neighbor points are not included, and type 3 errors have more than one cycle. After identifying errors in the mesh, the offending triangles (shown darker in Figure 2.11) are removed and the mesh is re-evaluated to fill holes. In practice, type 1 errors can be left for the hole filling process without ill effect. Removal of topological errors using this method can find errors in water-tight meshes, preventing issues in manifold testing, such as in Figure 2.12.

2.6 Alternatives and Other Data Processing

2.6.1 Higher Dimension Convex Hull

An alternative method to regioning was introduced by Brown in [18] and involves adding a fourth dimension to every point in the set and finding the convex hull. Once found, the fourth

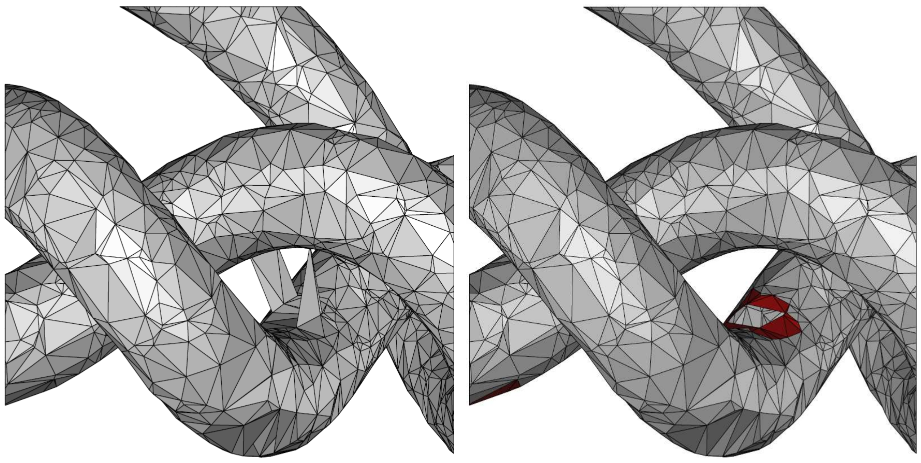

Figure 2.12: Before and after topological clean up [1].

dimension is discarded and the bottom of the new convex hull is equivalent to the Delaunay triangulation. This is discussed in more detail, along with other calculation methods, in the overviews by Aurenhammer [8] and Boissonnat [16]. More current research and some alternatives to strict Delaunay triangulation can be found in [6, 7, 22, 25, 32, 37]. While faster triangulation is desirable, discussion of the method is outside this project's scope.

2.6.2 Evening Resolution by Interpolation

Some of the data processing requires the mesh density throughout the model to be somewhat uniform. Once the set of faces has been found, they are reordered according to size. A standard curve is obtained, as well as the average size. If any face areas lie outside of the 95 percent confidence interval, adjusting them will help to prevent errors. Since merging faces that are too small will result in a loss of detail, faces that are too large will be broken up. The largest faces in the mesh are repeatedly broken into pieces until their area is equal or less than the average face area of the original set. Interpolation stops once no face lies outside two standard deviations from the average on the large side. Where possible, triangle reductions will consist of the four triangle set formed by the contact triangle so as to prevent small incident angles from becoming smaller, as illustrated in Figure 2.13.

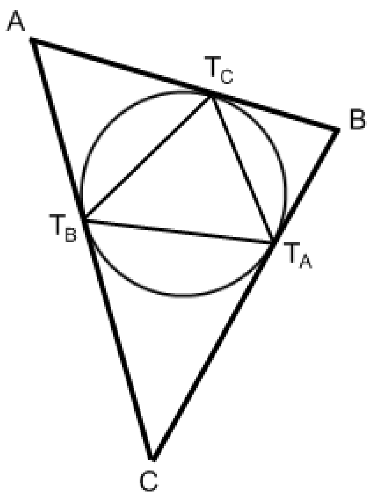

Figure 2.13: The contact triangle (or intriangle) is formed by connecting the three semiperimeter points of a triangle.

2.6.3 Medial Axis Transform

The medial axis transform (MAT) is a way to find topological data of an object. This data can be used to relate multiple objects to one another or to classify the object, partition the object, smooth the object's geometry or to recreate similar objects. The structure of the MAT describes the original surface in ways that are otherwise difficult to quantify. The MAT shows areas of uniformity as well as areas of noise. The structure of the MAT can show symmetry in the model, as well as areas where the model might naturally be split into subcomponents. See Figures 2.14 and 2.15 for examples.

The MAT is defined as the volume specified by the centers of all maximal interior balls. The balls will be largest in open volumes of the object, and will become smaller in more detailed areas of the boundary. The MAT is formed by connecting the centers of the balls. Sheehy *et al.* provides a good overview and an algorithm for triangulated solids in [41]. Dey [26] and Amenta [5] both describe approximate MAT methods.

2.6.4 Simplified Medial Axis Transform

Finding the MAT is calculation intensive. Foskey *et al.* describes an alternative that is faster called a simplified medial axis transform (SMAT) [28]. The SMAT also tries to eliminate some of the instability of the MAT by reducing the effects of small changes in the mesh. This

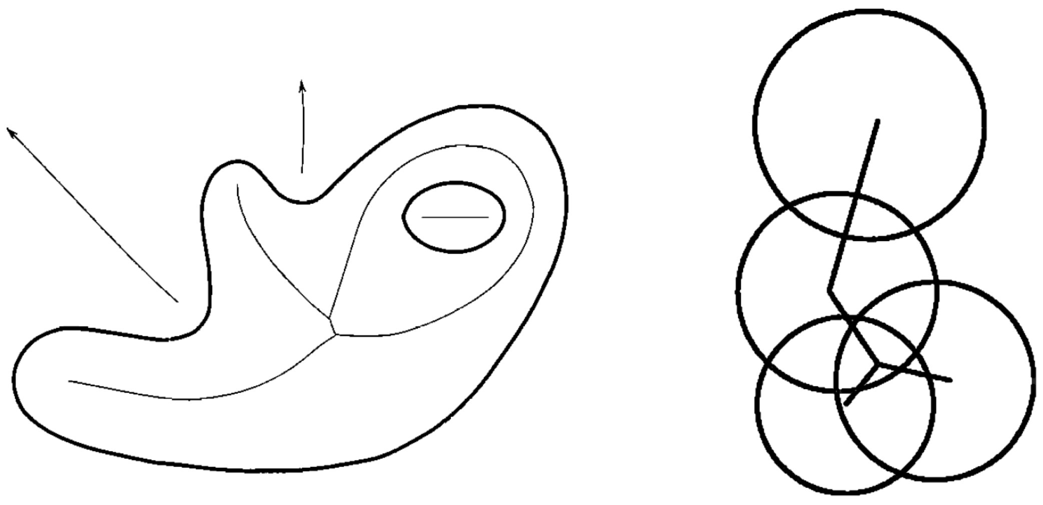

Figure 2.14: Left, the medial axis transform of a smooth body [3]. Right, the medial axis of a union of balls [5].

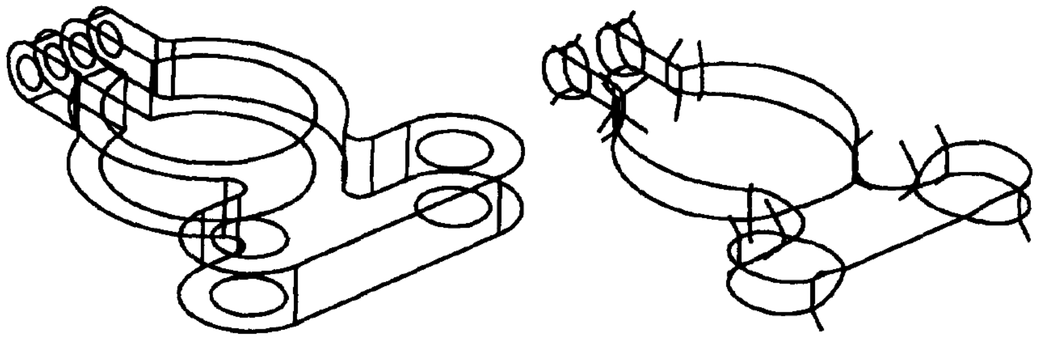

Figure 2.15: The medial axis transform of a 3D object [41].

provides a usable MAT alternative while actively changing the mesh during operations such as simplification, morphing or deformation (see Figure 2.16).

The SMAT reduces the number of points in the MAT by enforcing a angle restriction, forming a volume called the θ–simplified medial axis. When evaluating an incident circle or sphere for the MAT, the angle centered on the circle between the two points of intersection must be greater than θ for it to be included in the SMAT. This removes most of the spheres on the boundaries of the mesh where the faces are contiguous and smooth.

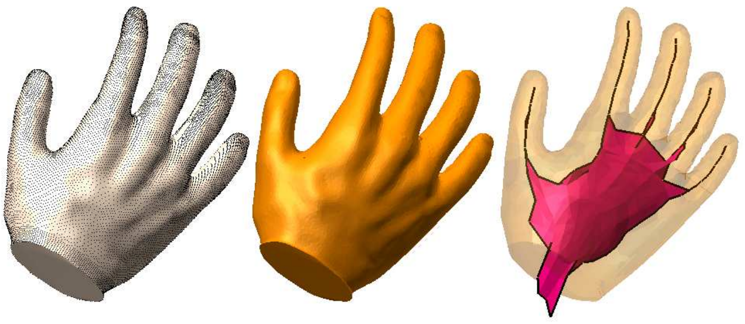

Figure 2.16: The range data, calculated surface, and simplified medial axis transform of a hand [4].

2.6.5 Power Crust

A more recent project that has had impact on the project's implementation is the research by Amenta *et al.*, notably in the research involving the Power Crust [5, 4]. The Power Crust is a combination of two methods for surface reconstruction - a simplified medial axis transform and a power diagram. The power diagram selects and changes the radius of the balls in the SMAT by a function of their distance from the original input data. This process is much faster than exact Delaunay triangulation, and produces a good mesh with far fewer triangles. This project makes use of the Power Crust in comparison to the Delaunay triangulation and as a faster evaluator of new input data.

2.6.6 Variational Shape Approximation

Depending on the density of the data, geometry simplification may reduce the number of faces significantly while retaining most of the contour data of the original data. This savings in polygons will speed all subsequent calculations considerably, as most of the calculations are not linear. Cohen-Steiner introduces variational shape approximation in [21], and the results are encouraging on some data sets. This can also be performed before the Delaunay tessellation, providing reduced complexity regions that are fast to triangulate. This is most beneficial on extremely large and dense data sets.

The method uses simple mathematical surfaces called shape proxies to cover the partitions on the original set of points. Fitness is measured by an optimization function. Once the error metric is below a set threshold, the partitions are split at the intersections of the shape proxies and a simplified mesh is created from the proxy surfaces, as in Figure 2.17.

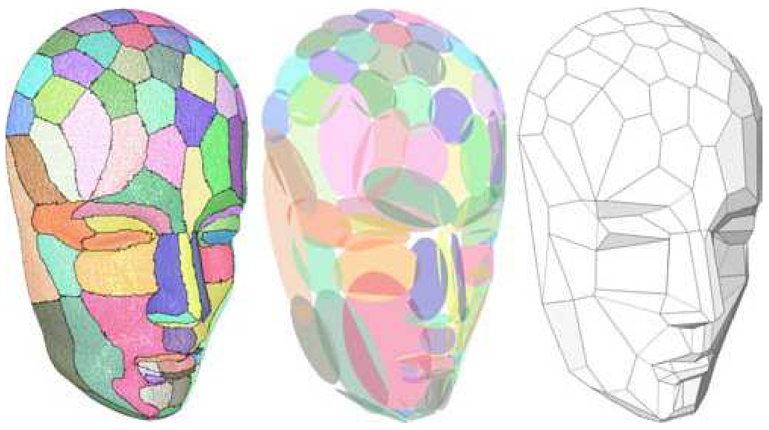

Figure 2.17: Variational shape approximation can quickly region mesh areas, providing a simplified mesh and partitioned sets of points to mesh separately [21].

2.6.7 Tetrahedral Meshing

Once a surface has been obtained, it can be used to find a tetrahedral mesh of the entire model (see Figure 2.18). Tetrahedral meshing fills the object with tetrahedra, a guaranteed convex solid. Once this set of tetrahedra has been found, it can be used for many more applications than a simple mesh. The solids can be used to calculate exact volume of the object, as well as the approximate volumes of smaller portions of the object.

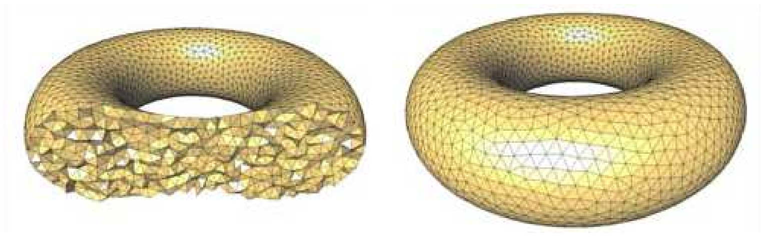

Figure 2.18: Tetrahedral mesh for a torus with 1000 vertices [2].

Alliez *et al.* describes one approach for this meshing called variational tetrahedral meshing that has shown accurate results [2]. It is a hybrid of optimal Delaunay triangulation and local vertex relocation that seeks to produce a high quality mesh while avoiding exhaustive calculations, producing well formed tetrahedra, and minimizing the global energy over the domain. The modifications to provide optimal and near-optimal Delaunay triangulations by Chen and Xu [20] help to keep the accuracy high without drastically increasing run-times.

In use, the algorithm produces tetra that vary in size with the density of the faces in the mesh. The tetrahedra formed are, in general, shaped well and without slivers. A sliver is a tetra is where any interior angle approaches zero and is considered a degenerate case in many calculation methods. Calculating the volume of a tetra is relatively simple (Equations 2.1, 2.2, 2.3, 2.4, 2.5), and by using this meshing algorithm we can obtain an accurate volume of our object. Furthermore, though outside the scope of this project, the tetra are essential in performing any geometric modifications to the model. Once the tetra have been calculated, it is much easier to deform the object, split it into pieces and add objects together. The tetra limit the volume affected by such operations. A repeat of the meshing algorithm on the affected subset can be done quickly, and the merge of the output to the remaining unaffected area is trivial.

Semiperimeter of a triangle with sides length abc

$$s = \frac{a+b+c}{2} \tag{2.1}$$

Area of a triangle with sides length abc using Heron's formula

$$A_T = \sqrt{s * (s-a) * (s-b) * (s-c)} \tag{2.2}$$

Planer normal vector $\vec{n}$ from three points $p_1p_2p_3$

$$\vec{n} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} = (p_2 - p_1) \times (p_3 - p_1) \tag{2.3}$$

Distance D between point P and the plane defined by point p and $\vec{n}$

$$D = \frac{|aP_x + bP_y + cP_z - (\vec{n} \cdot p)|}{\sqrt{a^2 + b^2 + c^2}} \tag{2.4}$$

Volume V of irregular tetrahedron with points $p_1p_2p_3p_4$

$$V = A_T(p_1, p_2, p_3) * D(p_1, \vec{n}(p_1, p_2, p_3), p_4) * \frac{1}{3} \tag{2.5}$$

2.7 Binary Space Partition Trees

Game engines commonly use a pre-compiled geometric data structure to speed the detection of collisions between objects and avoid calling computationally expensive collision operations on every pair of faces in a scene. Binary Space Partition trees, or BSP trees, store static objects as collections of planes divided into convex groups. Planar collisions are a quick operation, and the tree can reduce the number of objects needed to test for collisions by several orders of magnitude. It can also quickly identify objects that are not inside the camera view to save drawing operations in the renderer and sort planes in order of depth. BSP trees are also used for radiosity calculations and network optimizations. Ranta-Eskola gives an overview of the most common uses of BSP trees with pseudocode in [39].

2.7.1 BSP Tree History

The concept was originally introduced by Schumaker *et al.* in [40] and used arbitrary planes to help with the issue of correctly drawing cyclic-overlap (see Figure 2.19). The drawing routines of the day used the painter's algorithm - sorting polygons based on depth and drawing from bottom up. In this order, objects that are closer are drawn last, providing a simple way of hiding parts of objects that are occluded by nearer objects. The painter's algorithm has two major issues - all objects in the view must be drawn, and complex object interactions are not drawn correctly. One example of an unhandled complex interaction is cyclic overlap, where two or more objects cover each other. The painter's algorithm will incorrectly cover whichever object is thought to be farther away. By splitting the scene with a plane, cyclic overlap could be prevented by splitting one of the polygons into two pieces with one in front of the plane and one behind. Note that this is not a issue of complex polygons, as a cyclic overlap of three convex polygons is also possible.

Sutherland *et al.* (including Schumaker) described it in more depth later, with special regard for hidden surface removal and culling faces not within the viewing frustum, in [42]. Fuchs *et al.* added a refinement using cutting planes incident with surfaces in the figure [29] to avoid the expensive plane discovery step but possibly causing uneven tree balance and forming a constrained tree. While the effort is somewhat intensive to create the tree, the intermediate structure is negligibly larger than the original geometry and preprocessing

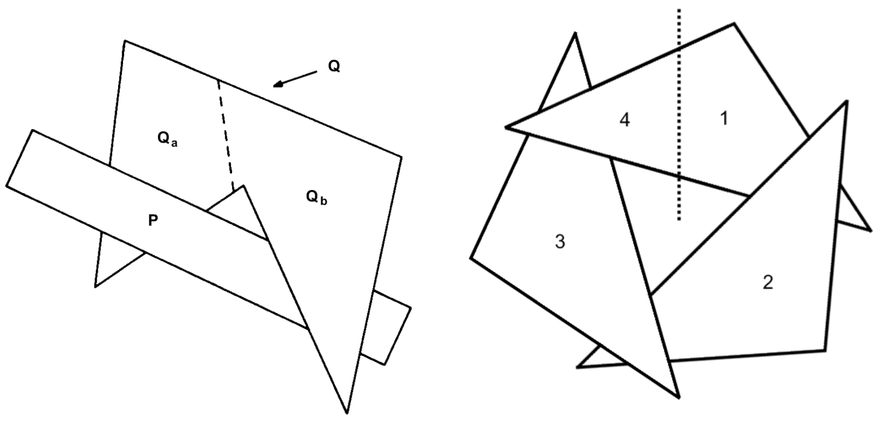

Figure 2.19: Left, Cyclic overlap occurs when two or more object cannot be sorted accurately according to depth - Q covers P and P covers Q. By splitting one of the objects, the depth priority can be determined - Qb covers P and P covers Qa [42]. Right, cyclic overlap of three convex primitives, a single cutting plane intersection and the corrected draw order of the pieces.

static objects provides much faster run-time calculations. This is the first definition of the BSP tree.

Gordon and Chen later used BSP trees to order polygons but reversed the draw order to front-to-back. By noting which parts of the screen had already be drawn, the renderer could omit drawing of occluded polygons for another large increase in rendering performance [30]. A variation on this is the Z-buffer, usually attributed to Edwin Catmull but described previously by Wolfgang Strasser, as shown by Green *et al.* in [31]. The Z-buffer records not only that a pixel has been drawn, but also notes the depth of that pixel. By using Z-buffering, the objects no longer need to be drawn in order of depth, though ordering by depth can prevent wasted re-draw. Z-buffering and z-culling show the highest speed increase when the processing to draw a single pixel is expensive, such as when complex shaders or lighting are used. Z-buffering is an accepted standard and is implemented in hardware on nearly all modern graphics cards.

2.7.2 BSP Tree Structure

BSP trees are formed by partitioning an input set of primitives - usually edges or planes. Partitions are made along a plane. Any plane can be used, though some implementations constrain to a plane defined by a primitive in the set to reduce calculation time. Any primitives that intersect this plane are split into two pieces. At each split, the new partitions are evaluated to determine if the primitives they contain are convex. If not, the non-convex set is further partitioned.

Choosing a partition plane can be difficult. When unconstrained, plane selection can take a long time due to the very large number of potentials. Constraining to primitives makes this selection set much smaller, but at the cost of not necessarily selecting the optimal plane. In practice, constrained plane selection is almost always preferable due to speed issues when forming the tree. An example of both constrained and unconstrained plane selection is shown in Figure 2.20. If constrained to primitives, any primitives that lie on the convex hull of the set cannot be chosen, as they will not divide the geometry and will add useless depth to the tree and an empty node. Depending on the object, calculating the convex hull and removing these primitives from the cutting plane potentials may be faster than testing the planes.

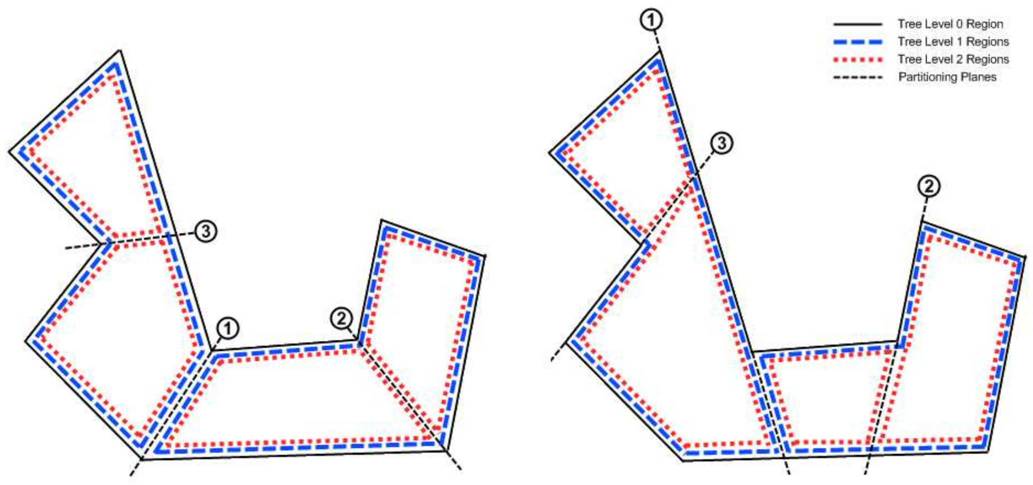

Figure 2.20: Left, unconstrained (1 split edge) and Right, constrained (3 split edges) BSP partitioning.

Partitioning can be done in one of three ways: such that the two formed partitions are of close to equal number of primitives, creating the most balanced tree; such that the fewest number of primitives would need to be split, avoiding the creation of new vertex data; or such that a new convex set is always formed in one of the partitions, which spends the least time finding a plane. All will produce valid trees and will have different performance traits depending on input set and application use. Partitions by primitive count is the most widely used, though combinations of the three methods are also common. Once all partitions contain convex sets, the tree is complete.

2.7.3 BSP Trees and Collision Detection

When an object is encoded in a BSP tree, it is reduced to a set of convex pieces. For each of these convex pieces, it is no longer necessary to retain the exact point data of the individual faces. Instead, the BSP tree stores a collection of planes that form a convex hull around the object. In order to test if the object is colliding, the query is tested against each plane in the BSP tree. If the query is fully outside (on the same side as the normal) any single plane, it is not colliding with that section of the tree. Assuming the query volume is usually small compared to the volume of the BSP tree, the majority of collision checks will be false. Pruning at the first negative test saves many calculations. Planar comparisons are much faster than triangle intersections and can be applied to a number of query simplifications, such as bounding spheres, to further speed the tests.

When collisions are detected, the result is a sub-set of the BSP tree. If simplifications were previously used, a less general simplification is retested against the BSP sub-set. If collision is again detected (usually returning a smaller BSP sub-set), the actual geometry of the query is used and retested. If collision is detected again, the object is in collision and the collision must be handled. If needed, the individual faces of the query can be tested against the BSP sub-set to determine the specific geometry that collided.

Chapter 3 Original Work

3.1 Reduced Medial Axis Transform

The reduced medial axis transform (RMAT) is the MAT reduced to segments and vertices. In 2D, a similar figure is referred to as the straight skeleton or straight-line skeleton (SLS), though it is computed differently than the MAT and RMAT (see Figure 3.1). Using the MAT calculated in Section 2.6.3, any curve data is interpolated into straight segments. Surfaces are reduced to segments by performing a constrained 2D MAT and interpolating any curves.

If using a SMAT, all solids are reduced to points at their centroids and segments are formed between neighbors. The SMAT is not guaranteed to be contiguous, so some post processing may be required. In general, the results are not comparable to the MAT unless θ is small (see Section 2.6.4).

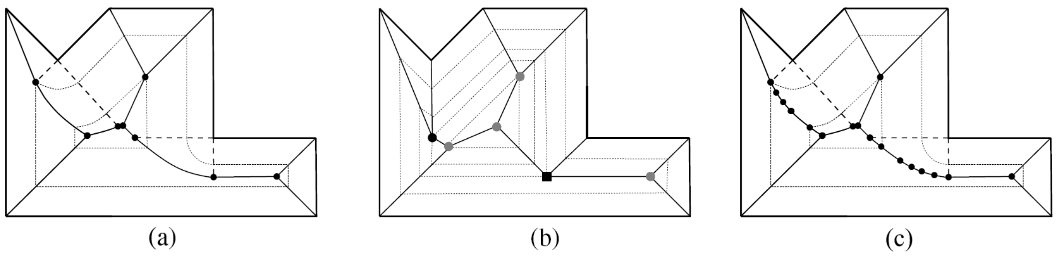

Figure 3.1: (a) The medial axis, (b) the straight-line skeleton, and (c) the reduced medial axis (modified from [43]).

The RMAT can be used to perform graph related functions on a generalization of the model's structure, including finding the volumetric thresholds and skeleton of the solid. It generally has less perturbations than an SLS and is more stable when the surface changes. It is also faster to calculate the MAT than the SLS for highly detailed input.

3.2 Volumetric Thresholds

After the tetrahedral mesh and RMAT have been calculated, they can be used to find the volumetric thresholds of the model. Volumetric thresholds are the dividing space between continuous volumes within an object. By finding the thresholds, regional volumetric data can be found. A planar representation of the threshold can also be used as a portal for visual culling. The threshold boundary can be used to partition the exterior mesh for use in dynamic deformations based on the determined skeletal structure.

The RMAT provides a skeletonization of the object - a set of points, either endpoints or intersections, and their connectivity as a set of segments. For each segment, find the set of spheres internal to the mesh and centered on the RMAT. Find any overlap between the segment sphere sets and mark that volume as an intersection. Record the position and size of the smallest sphere per segment that is not in the overlap. After all intersection volumes have been marked, find the interfaces between each segment sphere set and any intersections. Assign tetrahedra to either a segment or an intersection. For volumetric accuracy, the tetrahedra can be divided on the intersection borders, but in practice the error produced by leaving the tetrahedra whole and assigning them based on their centroid is negligible for well-formed tetrahedra. Finally, find all RMAT intersection points and assign them to the intersection volume that contains them, allowing multiple intersection points to share a single intersection volume if needed. Alternatively, the methods described by Edelsbrunner in [27] can be used to calculate some of this information by directly operating on the set of spheres generated, possibly saving time for highly detailed meshes.

This set of operations can provide the following data about the object: total volume, total surface area, simplified skeleton segment list (with intersections that share a volume reduced to a single intersection), low approximate smallest cross section of each segment, approximate volume of each segment and intersection, and a set of mesh regions and the portals between them. This data can be used as input to a number of different calculations, including MFIRE (see Appendix A.2.1) for air flow.

3.3 Thin-Surface Solid to BSP Solid Set Conversion

The input for the BSP tree must be made up of solid 3D objects, and each object must be made up of a convex hull of flat planes. The smallest valid object would be a pyramid with a total of four sides - a tetrahedra. Triangle meshes are collections of triangles, which are not 3D objects - they are 2D objects in a 3D space. A mesh of triangles forming a manifold is called a thin-surface solid (TSS). To be used in a BSP tree, the individual triangles must become objects that have some amount of volume, creating a new thick-surface solid or BSP solid set.

3.3.1 Interior/Exterior Detection

To convert the TSS, we must first identify the interior and exterior of the object. This can be done by selecting a single triangle at an extremity, designating the surface such that the face pointing away from the center of the solid is considered the outside, and propagating the designation of outside to that triangle's immediate neighbors. Repeat for all neighbors of neighbors until all triangles have been touched. When finished, we have a clear designation of interior and exterior of the solid. Reversing the designation is quick, if needed. The triangle points are reordered such that their winding orients their normals in the direction of exterior.

3.3.2 Intrusion/Extrusion

To create solids, each face must be intruded (e.g. open-pit mine) or extruded (e.g. underground mine) - in general, penetrating the side where the original data was sampled. The normal of each triangle is used to determine the correct offset direction. The normal is a vector from the center of the triangle extending perpendicular to the surface. For each pair of neighbors, the average of their normals and their neighbor edge are used to determine the plane of the new neighboring face between them. Once the average has been calculated for all three neighbors of a surface, the vectors can be used to create a set of five planes. These planes include the original face, the three border faces, and a containing face in the direction of intrusion or extrusion, as illustrated in Figure 3.2. The containing face is parallel to the original face, moved along the normal by an offset distance. This is a single addition, which allows the corollary surface to be generated with minimal overhead or recalculation. The offset distance is determined by the needs of the specific renderer.

The containing face should be omitted if the border faces converge, though the resulting solid may produce errors later (see Section 3.3.4). Border faces will be reused for the neighboring face. Compared to a full tetrahedra mesh, intrusion produces a set number of solids - only enough to provide proper collision detection.

If the goal was to form quads between neighboring faces, it would be tempting to use the vertex normals as the edges. Unfortunately, using vertex normals to describe the neighbor

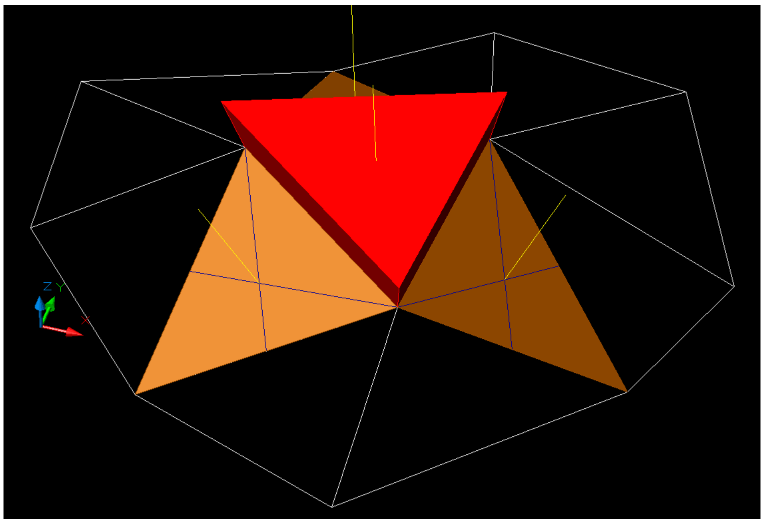

Figure 3.2: Example of extrusion - the containing and border faces are red while the neighbor faces are brown. Normals are yellow and face angle bisectors are blue.

plane in not a consideration, as they are not guaranteed to be coplanar. The average of the face normals yields the desired plane, though to form quads extra work must be done to account for the three intersecting planes.

3.3.3 Brushes and Visualization

The set of five planes that make up the individual extruded mesh triangles are all that is needed to create a BSP tree. In applications used for creating geometry specifically for game levels, the user manipulates convex objects called brushes. The brush contains the set of planes, as well as other information including texture. Brushes can be extended to denote game-specific objects, such as starting positions and goals.

Since it is difficult to visualize many planes simultaneously, it would be helpful to be able to instead visualize the volume formed by the planes. To convert a brush into a set of faces, each plane is intersected with all other planes. The area on the inside (opposite the normal) of the incident plane is kept, and the area left after all intersections is a face. The set of faces covers the interior volume of the brush and is easier to draw.

3.3.4 Offset Overlap

In narrow regions of a mesh, faces may be extruding into the same space. Overlap does not affect the viewing quality of the model from the other side of the mesh face, but if the overlap extends so far as to touch or cross another mesh face, it can cause issues. By calculating the border plane based on the neighboring faces' normals, overlap is prevented for immediate neighbors. Unfortunately, this does not apply for neighbors of neighbors or beyond.

When the offset volume of a face intersects another face, the two faces are a pair of offset overlap faces. Overlap faces occur in two types of degeneracy: sharp features and channels. Sharp features are where the mesh forms a peak, dimple, edge, or corner causing neighboring faces to form small incident angles to each other. The angle threshold is usually less than sixty degrees, but can vary depending on mesh density. Channels are where two faces are spatially close but topologically far from one another.

It is possible to identify geometry that is going to cause overlap issues before extrusion and change/simplify the geometry. This can be done through a modification to the previous MAT method. So far this paper has only shown the MAT interior to the mesh geometry, but the same calculations can be used to find the intersection of maximal balls on the outside of the mesh to form an exterior MAT (see Figure 3.3). Areas where overlap issues can occur will be where the balls have diameter equal or less than the offset distance, though extending the area to a slightly larger diameter can help identify bordering faces that will later be affected by any mesh changes. This allows the MAT calculation to ignore the degeneration of the balls as they become infinitely large and preventing some of the longest MAT calculations. Individual pairs of non-neighboring faces in the overlap areas will create issues if their nearest points are equal or less than the offset distance and are labeled as offset overlap faces. It is also possible to calculate the offset and identify problem areas after using collision checks between the mesh and each offset volume.

An alternative use of the exterior MAT is to identify the smallest exterior ball in a channel. The radius of this ball is the largest offset value that can be used without the need to post process the offsets. Channels are identified by a decrease in ball radius followed by an increase.

Figure 3.4 shows three solutions to this issue. The mesh can be modified to compensate

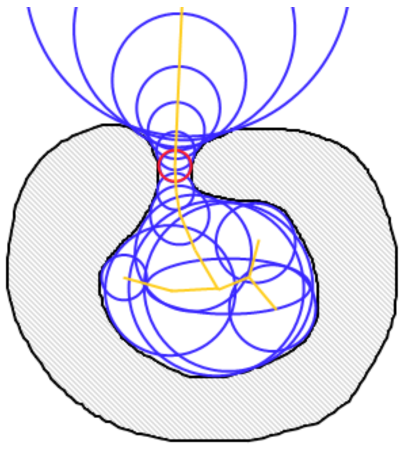

Figure 3.3: The exterior MAT of a solid, where blue shows maximal balls (excluding degenerate half-spaces), yellow shows the MAT, and red denotes the smallest ball in a channel. If the offset distance is greater than the radius of the ball, there will be offset overlap and the calculation must compensate. Note that if merging were to be used, the center space would become an island.

for offset overlap faces. The regions of the mesh surrounding each face can be smoothed, removing sharp features. Another option is mesh deformation. Overlap faces can be moved in the direction of their normal until their offsets no longer overlap any faces. The mesh can also be merged at the offset faces. An irregular cylinder is formed between the outside borders of the overlapping faces in the surface. In practice, the faces are not separated enough to require tessellation along the cylinder surface, but for large offset values this may be necessary. The zippering algorithm is applied at the intersection. The zipper will delete any faces interior to the border. Once fixed, localized smoothing algorithms are applied to the faces neighboring the changed region and the mesh is re-evaluated to find offset overlap faces, repeating the algorithm if needed.

Another alternative for preventing offset overlap is called convergence propagation. When neighboring faces intersect at a sharp angle, their offset areas will become long and narrow. Overlap is prevented for neighbors, but the small offset greatly increases the probability that the next face will cause overlap faces. When neighbors have sharp incident angles, their containing faces are usually omitted as their border faces converge before the offset distance. To prevent further neighbor offsets from overlapping, the neighbors can adjust their offset distance to the implied offset distance produced by the convergence, either directly inheriting

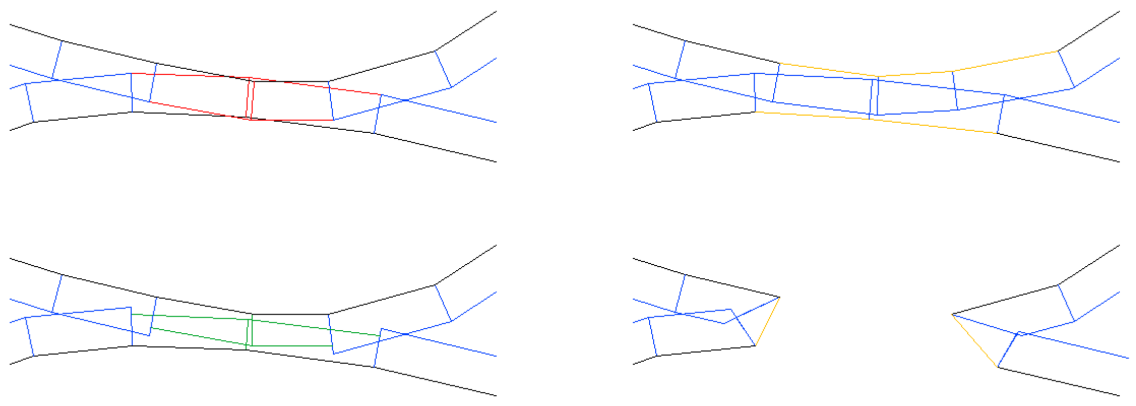

Figure 3.4: Top Left, an example of a channel offset overlap. White is original mesh, blue is calculated offset, red are offsets that overlap geometry. Three resolutions are presented: Top Right, mesh deformation; Bottom Left, dynamic offsets; Bottom Right, merging. Yellow shows altered original geometry and green shows altered offsets.

the distance or by using the sharp neighbors' border face as a containing face.

Lastly, the offset can become dynamic, with the original offset distance defining the maximum length. Each pair of overlap faces has their offset reduced until the offset volume no longer intersects with any face. This requires first finding all offset overlap face pairs, but each pair can be evaluated separately. By enforcing that the containing face must be parallel to the mesh face, there are no unintentional interactions between different sets of pairs.

In general, convergence propagation is adequate to deal with sharp features, but cannot handle channels. Dynamic offsets can account for both issues, but verifying all offset volumes takes calculation time. For both of these methods, some game engines may not handle collisions correctly with the new, thin brushes. The exterior MAT can account for both issues, but changes the model. Deformation reduces interior volume. Merging increases it. Channel removal can drastically change the mesh geometry, also changing the MAT and volumetric threshold. It can also create new portals and islands. Smoothing reduces the accuracy of the mesh.

Overlap faces are usually a sign of undersampled meshes - as the mesh density goes up, the offset distance can go down. In any case, final geometry should always be evaluated for fitness when the original mesh has overlap faces.

Chapter 4 Conversion Process

4.1 Utility Conventions

All conversion and extraction utilities written for this project are run via the command line. Some general Unix conventions are assumed. All utilities can take input either as a command line argument or via STDIN. All utilities will write to STDOUT unless the *–output* flag is passed specifying a file to write to. Status and information output is written to STDERR.

4.2 Input Conversion

The first step for tessellation is to extract raw 3D point data. These points can come from a variety of sources, but among the most common are from ASCII data or from AutoCAD proprietary DXF files [9]. Current modeling programs can usually convert their data into one of these two formats, and both are easily parsable.

Two conversion routines were written to format this data: *dxf2points* and *ascii2points*. Note that the input can be either point or surface data. If surface data is given, individual points are extracted and the surface is discarded. If specified, the *–interpolate* flag will cause faces larger than one standard deviation above average face area to be broken into smaller pieces. A triangle is divided into four using the triangle defined by the interior contact circle (see Figure 2.13). Curved data and more complex figures from DXF files are currently ignored, but they could be replaced at this step with interpolated data derived from user defined granularity parameters.

For multiple input data sets, registration is defined by four points for each set. The input data is preceded by the local coordinate equivalent for the origin and the XYZ unit segment terminators. This allows skewed, non-uniformly scaled and rotated input data which is translated to the global coordinate system at read time. It also allows the inclusion of 2D data, with the spatial relationship determining the constricting plane. If no parameters are given, the input conversion routine assumes the standard origin and scale are common to the local data.

Once extracted, any duplicate points are removed. The remaining points are ordered by distance from the origin and form the final point cloud. The points are written in a universal ASCII format for use as input to the tessellation routine, *points2tss*. Multiple data sets can be combined by passing the sets to either routine using the *–combine* flag.

```
INPUT
./dxf2points --interpolate --output dataset1.points dataset1.dxf
./ascii2points --combine dataset1.points --output dataset.points dataset2.txt

OUTPUT
dxf2points - Interpolation activated
dxf2points - Reading dataset1.dxf...
dxf2points - dataset1.dxf contains 0 points, 0 segments, 82852 faces, 0 curves
dxf2points - dataset1.dxf creates 47216 points from 82852 faces
dxf2points - dataset1.points contains 47216 points

ascii2points - Reading dataset2.txt...
ascii2points - dataset2.txt contains 1626 points, 0 faces
ascii2points - Reading dataset1.points...
ascii2points - dataset1.points contains 47216 points
ascii2points - Removed 21 repeated points
ascii2points - dataset.points contains 48821 points
```

4.3 Mesh Creation

The combined point cloud is input to *points2tss*, which has multiple paths for mesh creation. The *–qhull* flag will use the Qhull library [10]. The *–cgal* flag will use the CGAL library [15]. The *–jwm* flagwill use the author's original mesh implementation. The *–parallel* option expects a set of comma separated IP addresses or machine names with an optional colon separated port number. Unspecified port numbers will be assumed as 8332. The routine will attempt to contact all workers, but will continue even if it cannot connect to one or more of them. The *–notify* option takes an e-mail address to send results to. The amount of notifications depends on verbosity. The *-v* or *–verbose* flags increase the amount of output - up to three levels of verbosity are available. All connection information is written to STDERR.

The parallel meshing routine *points2tssworker* listens on a port for connections from a *points2tss* routine. It takes *–port* as a parameter but will use port 8332 if it is not specified. It will run as a background process and must be killed by PID.

```
INPUT
./points2tss --cgal --notify root@localhost \
  --parallel 192.168.0.10:6000,192.168.0.11:10000,192.168.0.12,clusterslave3 \
  --output dataset.tss dataset.points

OUTPUT
points2tss - Using CGAL meshing support
points2tss - Notifications will be sent to root@localhost
points2tss - Contacting 192.168.0.10:6000... (Failure)
points2tss - 192.168.0.10 - ping successful, worker not running?
points2tss - Contacting 192.168.0.11:10000... (Success)
points2tss - Contacting 192.168.0.12:8332... (Success)
points2tss - Contacting clusterslave3[192.168.0.13]:8332... (Success)
points2tss - Reading dataset.points...
points2tss - NOTIFY - Started meshing 48821 points with 3 workers Fri Aug  4
01:03:20 PDT 2006
points2tss - NOTIFY - Finished meshing Fri Aug  4 01:16:01 PDT 2006
points2tss - dataset.tss contains 85611 faces
```

4.4 Extrusion

The conversion program that creates the BSP Solid Set is called *tss2map*. By default, it produces a map file for input to Quake III. The *–offset x* option sets the offset value to x. x can be specified as a float or integer. If x is preceded by the @ character, all faces will be examined and the average length of an edge will be calculated. The offset will be x times the average length. If an offset is not provided, it will be assumed to be *@0.1*. The *–intrude* flag specifies that the program should intrude instead of extrude. This also adds an orthogonal surrounding mesh with minimal density to act as a bounding box. The *–precalc* flag will perform exterior MAT on the surface before extruding, looking for overlap based on the offset value. The *–overlap* option specifies the overlap handling routine, if needed. If unspecified, the program will only write output if there are no overlap faces. Possible overlap specification values are *deform*, *dynamicoffset*, *merge*, and *any*.

```
INPUT
./tss2map --output dataset.map dataset.tss
./tss2map --overlap any --output dataset.map dataset.tss

OUTPUT
tss2map - Reading dataset.tss...
tss2map - dataset.tss contains 83611 faces
tss2map - Offset is 0.0239
tss2map - ERROR - Found 6 overlap faces.  (Minimum separation: 0.0155)

tss2map - Overlap will be handled by: any
tss2map - Reading dataset.tss...
tss2map - dataset.tss contains 83611 faces
tss2map - Offset is 0.0239
tss2map - WARNING - Found 6 overlap faces.  (Minimum separation: 0.0155)
```

```
tss2map - Evaluating possible changes... chosing deform
tss2map - Applying deform to overlap faces...
tss2map - Starting over...
tss2map - dataset.map is complete
tss2map - WARNING - dataset.map has altered geometry
```

4.5 Other Data

Once the mesh has been created, processing can be done to extract more data useful in optimization. The RMAT extraction program, *tss2rmat*, produces a geometry file consisting of vertices and a connectivity graph of those vertices. The *tss2tets* program creates a tetrahedral mesh from a thin-surface solid. The output is a geometry file of vertices and an index table of vertices for individual tetrahedra.

Once the RMAT and tetrahedral mesh are available, they can be simultaneously evaluated to find various information about the original geometry. After splitting the geometry into regions, each region can be evaluated to find its volume, surface area, and smallest cross-section. The *geometryinfo* utility outputs this information in a human-readable format to a text file. The *maphints* program performs a subset of these calculations and attempts to find optimal portal placement for a map. Brushes are added to an existing map file and textured with the special *hint* texture (see Section 4.7) by finding the volumetric thresholds of the geometry. Both of these utilities specify the RMAT file with the *–rmat* option and the tetrahedral mesh with the *–tets* option.

```
INPUT
./tss2rmat --output dataset.rmat dataset.tss
./tss2tets --output dataset.tets dataset.tss
./geometryinfo --output dataset.txt --tets dataset.tets --rmat dataset.rmat
./maphints --output dataset2.map --tets dataset.tets --rmat dataset.rmat dataset.map
```

4.6 Input Requirements for Engine Limitations

The planes of the BSP solid set or MAP file are used as the input to the specific game engine BSP compiler. As each engine has its own set of input parameters and data requirements, advanced knowledge of the target platform can allow for integration and testing at the TSS conversion step. The current project has targeted the Quake III and Torque game engines, providing compatibility with the many derivatives of each that use similar input formats.

Some earlier versions of Torque have a limitation of +/-4096 integer-only coordinates for vertices. This usually means that the coordinates of the calculated brushes must be truncated or scaled to fit in this region. Scaling need not be uniform, allowing better resolution in the smaller displaced dimensions. Support for float-based coordinates is mentioned in the engine source, but in practice the coordinates are truncated. To mitigate this, Torque allows multiple compiled BSP objects to be combined in a larger play area defined by a Torque mission file. This requires some interesting splitting calculations to be done, even occasionally requiring alteration of the MAP file to split up larger brushes, but it can result in a full-detail version of the original map file. Extra calculations must be done to orient and arrange the individual pieces of the map file into the Torque mission file.

Compared to Quake III, where the lighting is pre-calculated during the BSP compile step, Torque must calculate the lighting for the combination of all these interior pieces for the mission. This is done within the engine by calling a *relight scene* command. For especially complex geometry, like the output of the examples in this paper, the lighting step can take several minutes even when only a single omnidirectional light source is used. Fortunately this lighting data is cached with the mission and will not need to be repeated unless the geometry changes or moves.

The Quake III engine has an upper limit of 2048 individual drawn entities, which begins to be an issue when approaching 10,000 brushes. When the limit is exceeded, depending on which version of the engine being run, errors manifest as either missing geometry, blinking geometry, or renderer crashes. This can be mitigated by proper regioning using portals. There are also restrictions on the number of textures that can be applied and the dimensions of input textures.

4.7 MAP and BSP Optimizations

While the MAP file so far has been a relatively ubiquitous file type, different game engines provide optimizations that may not be generally supported. Render speed is not a main consideration of this project, but a couple of easily implemented optimizations should be mentioned due to their performance gains and low cost.

During the BSP compile process, a map is split into regions by portals. Unfortunately, the automatic detection of portal placement is sometimes very inefficient, causing the engine to attempt to draw many surfaces that end up obscured by closer geometry. To combat this, a special texture named *hint* is provided. When deciding on portal planes, the BSP compiler will first try to region the map using the hint faces, then follow-up as needed with calculated portal faces. If even a few hint planes are added, the overdraw of complex regions can be reduced greatly. Note that the map file has no way of specifying a single plane, only full brushes. Since the hint is only a single plane, the other planes of the brush can be textured as *hint_skip*, effectively discarding them in any BSP calculations. Both hint and hint_skip textured faces are not drawn by the renderer.

For this project, the volumetric thresholds can have a second benefit by also acting as portal hint planes. This requires an additional calculation to create a brush at the threshold, which may be irregular and concave, but the work required is relatively minor and can be completed at any time - even as post-processing to a finished map. Depending on the engine used, it may be possible to pre-calculate which regions of the map will have the largest face draw count and specifically target these areas and the adjacent thresholds for this optimization. Since the hint planes can be added and removed easily, it only requires a recompile of the BSP tree to test different portal choices.

In some later games based on the Quake III engine, such as *Enemy Territory*, a special texture type was created named *caulk*. Applying this special texture to known hidden surfaces (such as the neighbor face shared by two adjoined brushes) would tell the renderer that the face would never be seen. Since it can't be seen, the renderer will never attempt to draw it and it will not be used in any visibility calculations during the BSP compile. This is a kind of tag culling and is commonly a large speed improvement. The mesh intrusion/extrusion output benefits greatly, as the number of faces that will never been seen approaches 83 percent of the total. In the examples presented in this paper, there was only a 0.2 percent increase in the number of drawn faces over the original tessellations. On average, this reduced the BSP compile time by about 40 percent and increased the frame rate of the renderer by over 50 percent.

Though not used in this conversion, brushes can be marked with a tag called *detail*. This tag is used for geometry that should not be considered when calculating portals but should still be drawn. In general, any non-wall static object is marked with this tag, leaving only the wall and structure geometry to be used during the visibility calculations in the BSP compile. As the current conversion process does not add any visible, non-wall geometry, this is not necessary, but when starting to add any sort of visual enhancement to a map it should be considered and used appropriately.

Chapter 5 Case Studies

This chapter follows the process from input data through the conversion process and finally to the display of the data inside a game engine. Statistics and run times are provided for various options. Where possible, colors are used to illustrate relative height.

5.1 Cove Pit Mine

The Cove Pit is part of a joint operation of the Cove/McCoy mining operation located near Battle Mountain, Nevada. It was the largest silver producing mine in North America with an estimated total capacity of 1.9 million ounces of gold and 83 million ounces of silver [35]. The coordinate data for this mine comes from a survey in 2004 and is a combination of optical and GPS data in an AutoCAD DXF file.

Table 5.1 lists the statistics and run times for the Cove Pit input file. Note that as the pit is not a closed figure, volume-based calculations, including tetrahedral meshing, are not possible. This also prevents the use of maphints, but as visibility is continuous throughout the pit, this is not much of a concern.

Cove Pit Mine			
Statistics		Run Times (sec)	
Points	78448	dxf2points	8
Faces	156909	points2tss (CGAL)	2995
Dimensions	(5148.5, 4065.9, 762.5)	points2tss (QHull)	5441
Surface Area	80565486.3	points2tss (JWM)	17302
Volume	N/A	points2tss (CGAL) (3 Workers)	1973
		tss2map	51
		tss2rmat	255
		tss2tets	N/A
		maphints	N/A
		q3map2	4642

Table 5.1: Cove Pit Mine Statistics and Run Times

Figure 5.1 shows the calculated crust using the points of the input file. Note the perturbations and long, thin triangles on the outside border. The crust considers the closest points to any point, so outlying points will still form triangles with their closest neighbors.

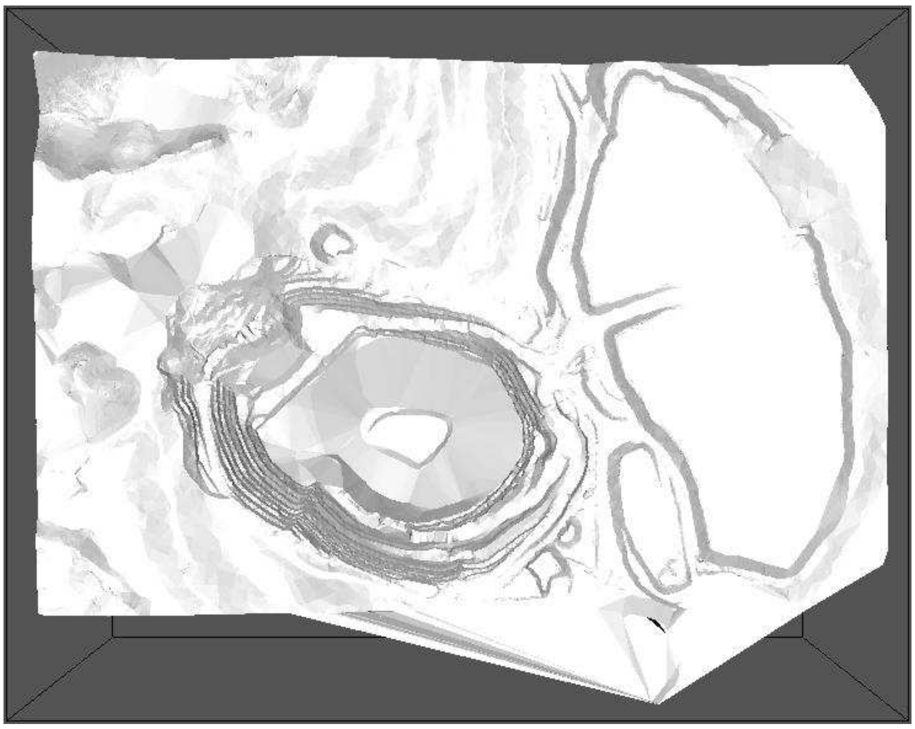

Figure 5.1: Cove pit crust.

Figure 5.2 shows the full point set of the Cove Pit. Figure 5.3 shows the triangulation calulated from the input data. Figure 5.4 shows the edges calulated from the input data.

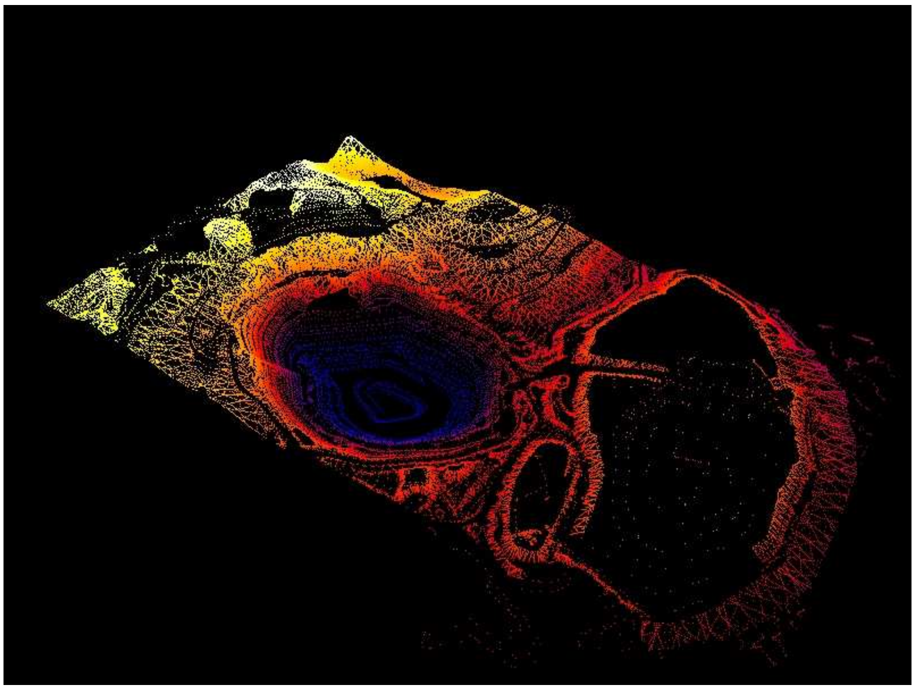

Figure 5.2: Cove pit point set.

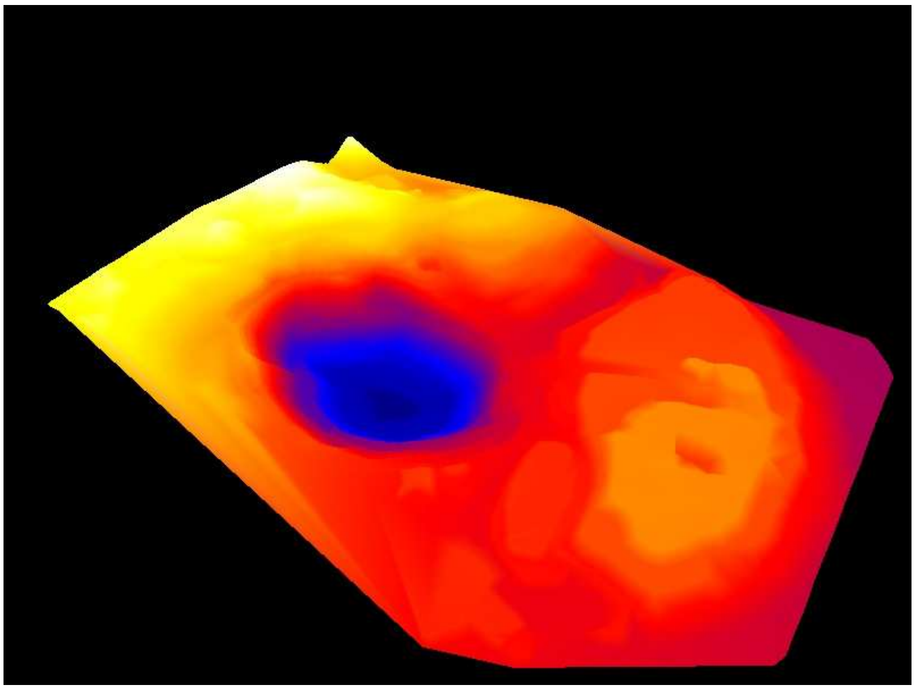

Figure 5.3: Cove pit Delaunay triangulation.

5.2 Total Solid Mine

The Total Solid mine is a conceptual mine from the University of Nevada, Reno Department of Mining Engineering. It is an underground mine design for excavating a 50 meter wide 300

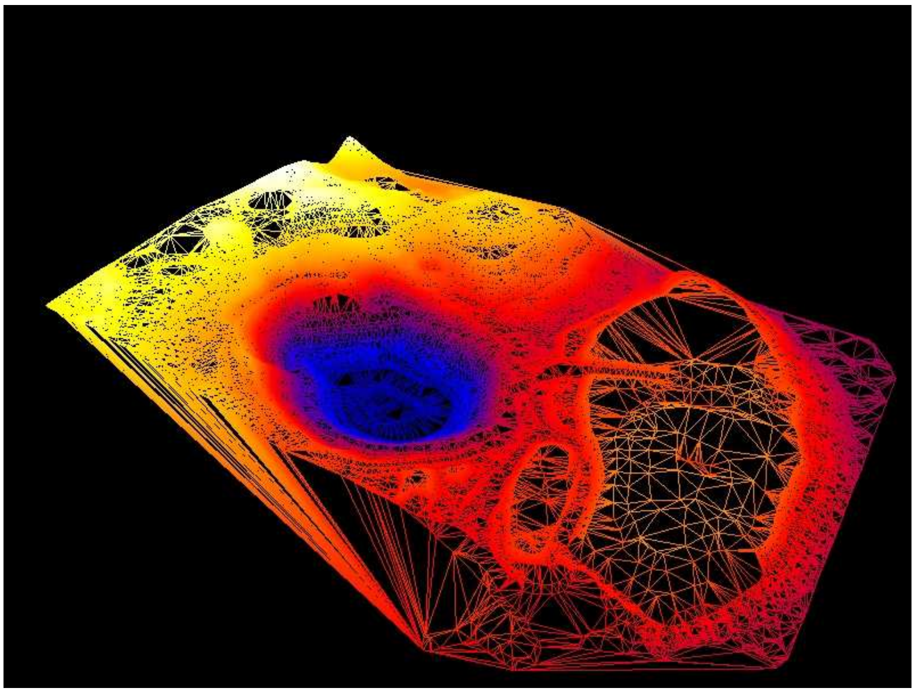

Figure 5.4: Cove pit triangulation edge set.

meter deep ore deposit. The coordinate data is from an AutoCAD DXF file.

Table 5.2 lists the statistics and run times for the Cove Pit input file.

Total Solid Mine			
Statistics		Run Times (sec)	
Points	6813	dxf2points	3
Faces	13470	points2tss (CGAL)	619
Dimensions	(99.9, 120.4, 99.3)	points2tss (QHull)	799
Surface Area	103455.7	points2tss (JWM)	1813
Volume	25219.4	points2tss (CGAL) (3 Workers)	408
		tss2map	18
		tss2rmat	474
		tss2tets	1563
		maphints	204
		q3map2	1925

Table 5.2: Total Solid Mine Statistics and Run Times

Figure 5.5 shows the calculated crust using only the points of the input file. Note the effect of uneven sampling. With so much distance between the indivual points, the crust treats smaller portions of the model as individual models.

Figure 5.6 shows the calculated crust using resampled points from the input file. The

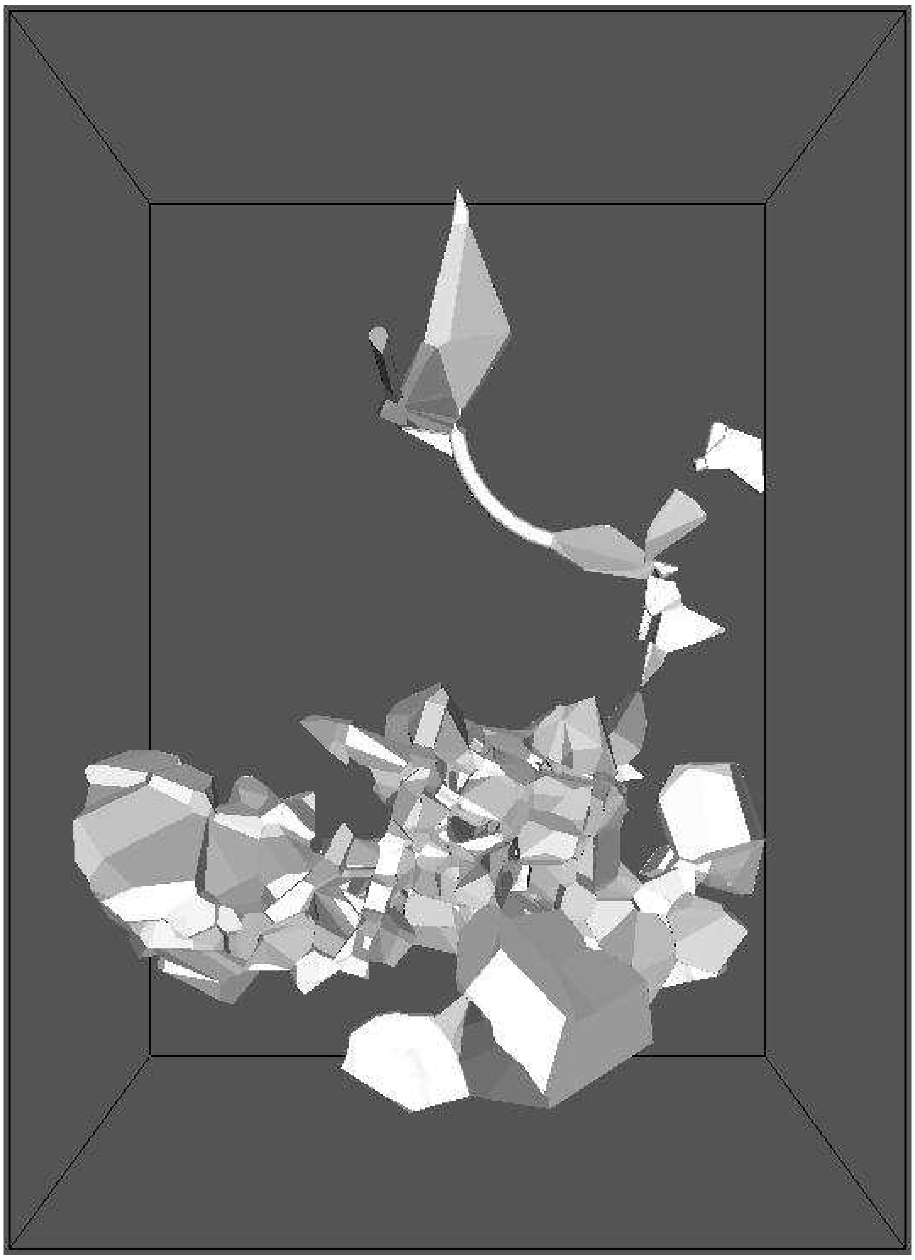

Figure 5.5: Total Solid crust - note the effect of uneven sampling.

crust is a much better fit.

Figure 5.7 shows the full point set of the Total Solid mine. Figure 5.8 shows the triangulation calulated from the input data. Figure 5.9 shows the edges calulated from the input data.

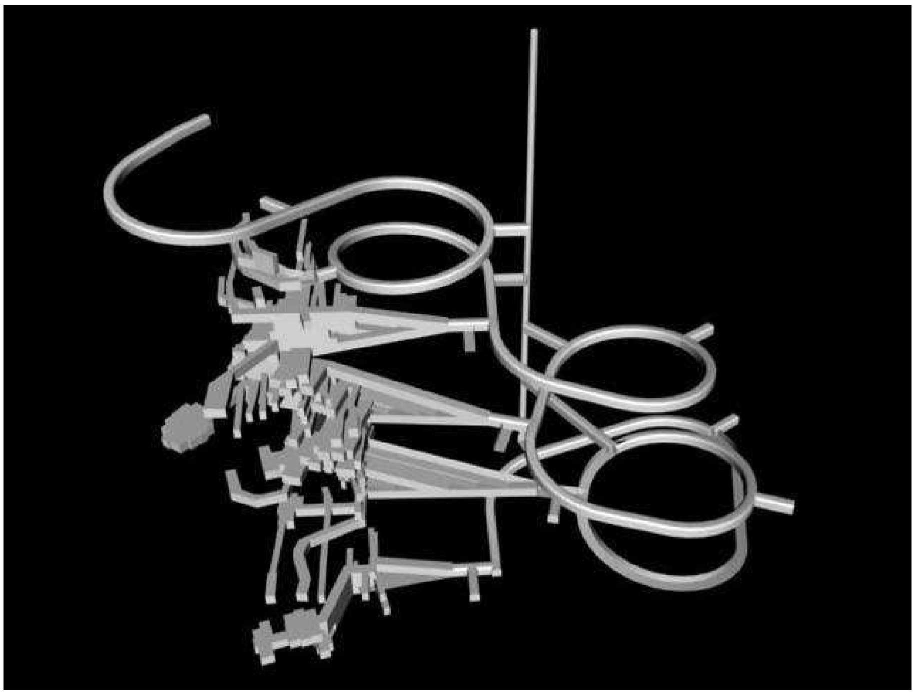

Figure 5.6: Total Solid resampled crust.

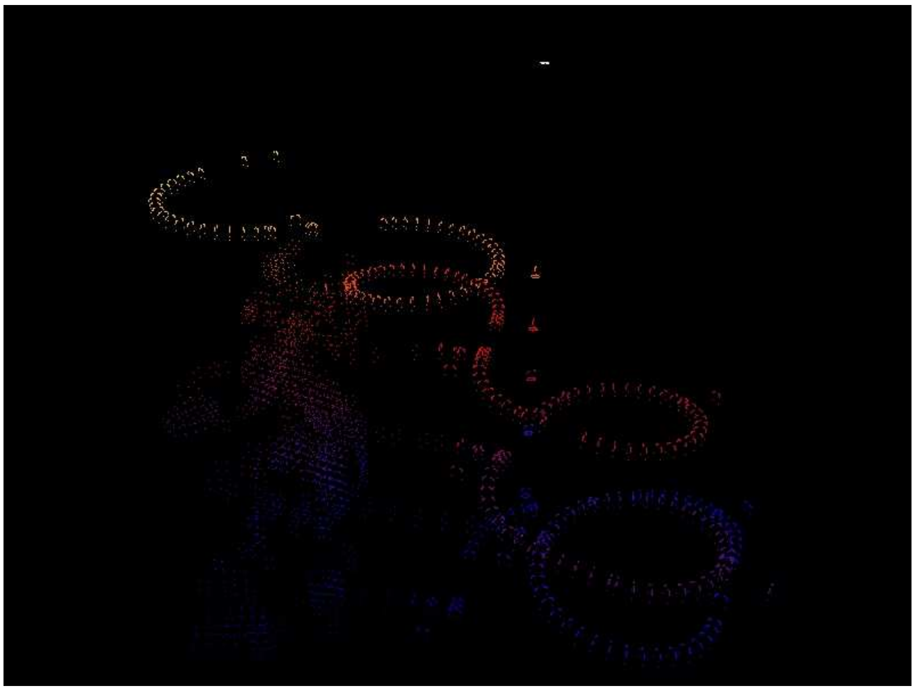

Figure 5.7: Total Solid point set.

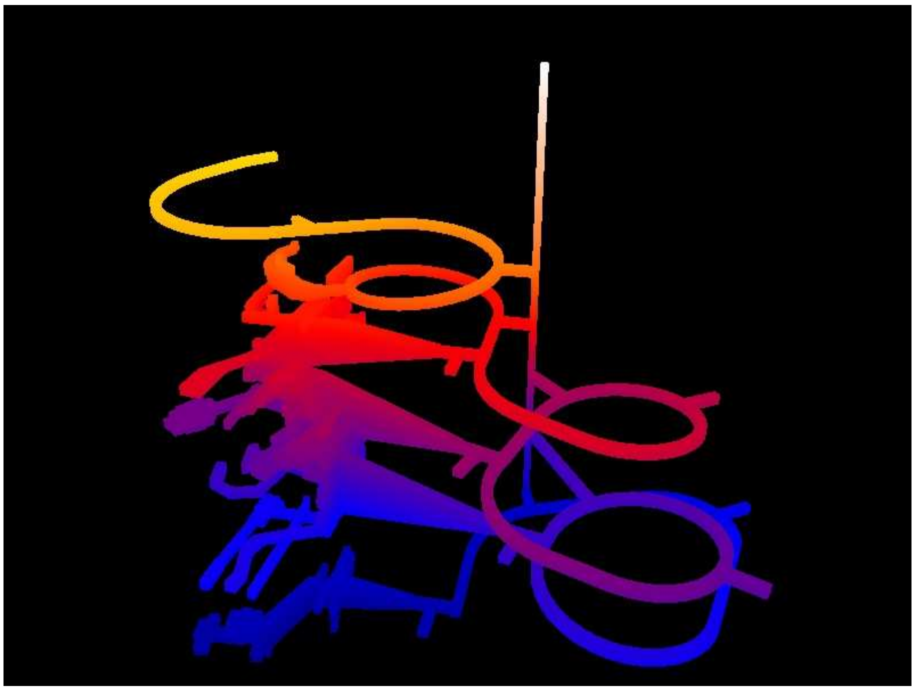

Figure 5.8: Total Solid Delaunay triangulation.

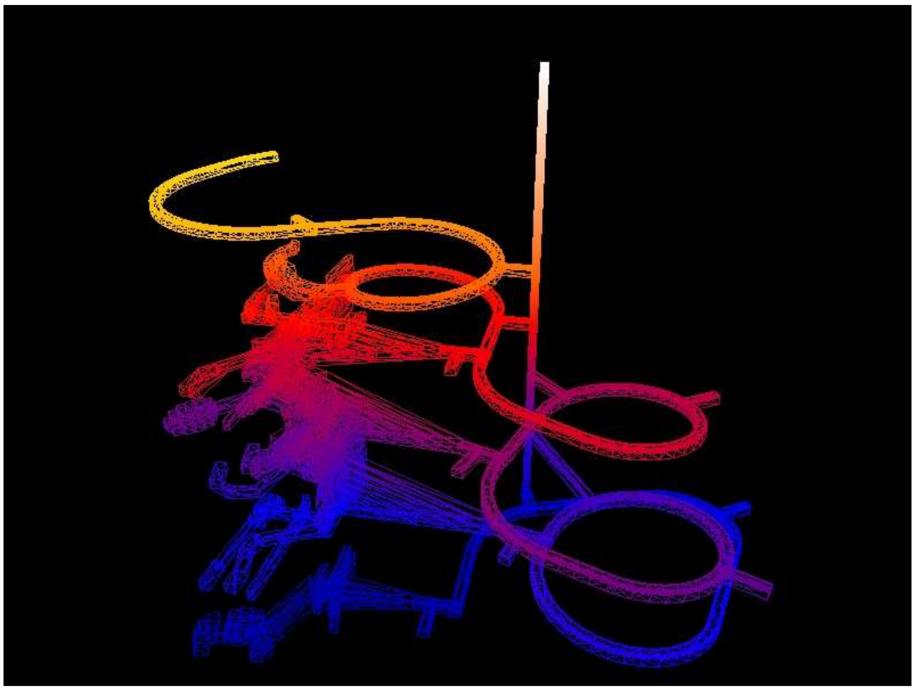

Figure 5.9: Total Solid triangulation edge set.

Chapter 6 Conclusion and Future Work

6.1 Conclusion

Volumetric thresholding provides two major benefits: information for placement of portals in the MAP file and volumetric data on the individual pieces of the skeleton. In another field, volumetric segmentation of the model can also be used to localize mesh deformation.

Conversion of a thin-surface solid to a BSP solid set is more complex than expected. Directly forming tetrahedra from the individual faces is not a robust solution, showing numerous manifold and visual errors. By using neighboring faces and calculating a shared border plane, most of these anomalies are prevented. The intrusion/extrusion step is timely and most errors can be caught early on.

The MAT is a surprisingly useful data model. The use of the reduced medial axis transform provides quick solutions in both the skeletonization and in offset overlap detection. More investigation in how to apply the MAT to other problems is definitely needed.

One of the largest difficulties in this project was the smaller modifications needed to produce adequate output for each individual game engine. While the basic idea of the BSP compilation is unchanged, differences in MAP limitations from engine to engine can be quite extreme. This limits the scope of the project, and might be a time-intensive area for any modifications targeting a new engine.

The computation power of modern machines has increased to a point where static BSP compilation of geometry is not necessarily needed for performance. The main benefit from using BSP trees is the ability to do a large amount of optimization on objects that cannot change during run-time. For future applications where all geometry is alterable, this will no longer be usable. While the MAP creation step may not be long-lived, the rest of this project still has relevance in other geometric applications.

Conversion of real-world spatial data into a format that can be read by modern game engines is not only possible, but works with very few issues. The conversion can tolerate inconsistent resolution in input sets, identify and overcome issues when storing that geometry, combine multiple input sets, and format output to work with multiple game engines. As the method matures, vast amounts of existing data can be brought into a virtual environment with minimal hassle. This conversion will help to make the power and simplicity of modern game engines available to simulation applications and to bring more realism and variety to future games.

6.2 Future Work

There are several areas to expand on the work in this project. Support for more input formats would allow more diverse input sets to be considered. Improving the interpolation options for input data, especially complex and curved data, would also allow more diverse input and possibly faster meshing by retaining information where possible. Support for more output formats, especially non-MAP files, would greatly increase the number of engines that could be targeted.

Support could be added to provide cosmetic enhancements to the output MAP files via more complex texturing, possibly basing which textures to use on the geometry data. More realistic textures could be added, such as satellite imagery for the open-pit mines. Extra detail geometry and lighting could also be added.

The additional data gathered can be used as input to related simulation calculations, such as using MFIRE for heat and smoke movement. The geometric calculations could be improved on as research progresses in the field, especially to speed the individual conversions and identify the problem areas discussed. Finally, packaging these applications and providing them in a single application would expand the target audience and make the overall project more useful.

Appendix A Applications and Tools

A.1 Libraries

A.1.1 Qhull

http://www.qhull.org

Qhull is a general dimension convex hull program that reads a set of points from stdin, and outputs the smallest convex set that contains the points to stdout. It also generates Delaunay triangulations, Voronoi diagrams, furthest-site Voronoi diagrams, and half-space intersections about a point [10]. Qhull is open-source with no specified license.

A.1.2 CGAL - Computational Geometry Algorithms Library

http://www.cgal.org

The goal of CGAL is to provide easy access to efficient and reliable geometric algorithms to users in industry and academia [15]. CGAL has algorithms to compute n-dimensional convex hulls, graph traversal, mesh simplification, and more. CGAL is open-source and free to use in non-commercial projects.

A.2 Related Applications

A.2.1 MFIRE

Originally developed at Michigan Tech, MFIRE is a computer simulation program that performs normal ventilation network planning calculations, and dynamic transient state simulation of ventilation networks under a variety of conditions. The program is useful for the analysis of ventilation networks under thermal and mechanical influence. MFIRE simulates a mine's ventilation system and its response to altered ventilation parameters, external influences such as temperatures, and internal influences such as fires [19].

A.3 MAP File Editors

A.3.1 Radiant

http://www.idsoftware.com/business/techdownloads

Radiant is the original editor for Quake-based games and is available from id Software. Several versions exist, including GtkRadiant, Q3Radiant, and the upcoming ZeroRadiant.

A.3.2 QuArK - the Quake Army Knife

http://quark.planetquake.gamespy.com

QuArK is an independently developed map editor. Originally called Quakemap, QuArK has grown to support a large range of games. It is written in Dephi and expanded with python. QuArK is open-source and still developed.

A.4 Game Engines

A.4.1 Quake III

http://www.idsoftware.com/business/techdownloads

Quake III is the third installment in the venerable Quake franchise, originally touted for using full 3D level design. Released in 1999, it uses the id Tech 3 engine. The full source code of the engine was open-sourced in 2005 and is available from id Software. Note that the models, maps, and textures were not open-sourced.

Content for Quake III-based games is commonly supported in various modeling programs, especially the MD3 model format. MD3 is a generic container for models and animations It allows segmentation of the model into individually addressed pieces while still performing dynamic mesh deformation to keep the outer mesh consistent. It also allows dynamic keyframing for animations.

Many games after Quake III have utilized the same engine, including Call of Duty, Medal of Honor - Allied Assault, and Star Wars Jedi Knight II: Jedi Outcast. Individual games have made modifications to the engine, so while the file formats are similar, certain extensions may require resources to be built with tools specific to the game.

A.4.2 Torque

http://www.garagegames.com

The Torque engine is the product of GarageGames, Inc. It was originally developed by the company Dynamix for the game Tribes 2. After release of the game, some Dynamix employees negotiated the purchase of the engine from Tribes 2 publisher, Vivendi Universal, and began to market the engine to independent developers. It is considered to be one of the lowest-cost commercial engines available and has been used in many games, including Marble Blast and Penny Arcade - On the Rain-Slick Precipice of Darkness.

The developer package contains more than just the engine - a full content pipeline is provided, with support for several modeling programs and audio formats. The engine itself provides rich content tools, allowing in-game terrain editing and placement of world objects. The full source code of the engine is provided, along with support for compiling under Linux, XCode on OSX, and Visual Studio on Windows. The engine also supports a rich scripting language, allowing game logic to be updated without recompiling the engine.

Bibliography

[1] Udo Adamy, Joachim Giesen, and Matthias John. New techniques for topologically correct surface reconstruction. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 373–380, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[2] Pierre Alliez, David Cohen-Steiner, Mariette Yvinec, and Mathieu Desbrun. Variational tetrahedral meshing. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 617–625, New York, NY, USA, 2005. ACM Press.

[3] Nina Amenta and Marshall Bern. Surface reconstruction by voronoi filtering. In *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*, pages 39–48, New York, NY, USA, 1998. ACM Press.

[4] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. The power crust. In *SMA '01: Proceedings of the sixth ACM symposium on Solid modeling and applications*, pages 249–266, New York, NY, USA, 2001. ACM Press.

[5] Nina Amenta and Ravi Krishna Kolluri. Accurate and efficient unions of balls. In *SCG '00: Proceedings of the sixteenth annual symposium on Computational geometry*, pages 119–128, New York, NY, USA, 2000. ACM Press.

[6] Dominique Attali and Jean-Daniel Boissonnat. A linear bound on the complexity of the delaunay triangulation of points on polyhedral surfaces. In *SMA '02: Proceedings of the seventh ACM symposium on Solid modeling and applications*, pages 139–146, New York, NY, USA, 2002. ACM Press.

[7] Dominique Attali, Jean-Daniel Boissonnat, and André Lieutier. Complexity of the delaunay triangulation of points on surfaces the smooth case. In *SCG '03: Proceedings of the nineteenth annual symposium on Computational geometry*, pages 201–210, New York, NY, USA, 2003. ACM Press.

[8] Franz Aurenhammer. Voronoi diagramsa survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, 1991.

[9] AutoDesk, Inc. *DXF Reference*, autocad 2008 edition.

[10] C.B. Barber, D.P. Dobkin, and H.T. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. on Mathematical Software*, 22(4):469–483, Dec 1996.

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

[12] Jon Louis Bentley. K-d trees for semidynamic point sets. In *SCG '90: Proceedings of the sixth annual symposium on Computational geometry*, pages 187–197, New York, NY, USA, 1990. ACM Press.

[13] Paul J. Besl and Ramesh C. Jain. Three-dimensional object recognition. *ACM Comput. Surv.*, 17(1):75–145, 1985.

[14] Paul J. Besl and Neil D. McKay. A method for registration of 3-d shapes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(2):239–256, 1992.

[15] CGAL Editorial Board. CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[16] Jean-Daniel Boissonnat. Geometric structures for three-dimensional shape representation. *ACM Trans. Graph.*, 3(4):266–286, 1984.

[17] Mario Botsch, Mark Pauly, Christian Rossl, Stephan Bischoff, and Leif Kobbelt. Geometric modeling based on triangle meshes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, page 1, New York, NY, USA, 2006. ACM Press.

[18] K. Q. Brown. Voronoi diagrams from convex hulls. *Inf. Process. Lett.*, 9(5):223–228, 1979.

[19] Xintan Chang, Linneas W. Laage, and Rudolf E. Greuer. *A user's manual for MFIRE: a computer simulation program for mine ventilation and fire modeling*, 1990.

[20] Long Chen and Jin chao Xu. Optimal deaunay triangulations. *Journal of Computational Mathematics*, 22(2):299–308, 2004.

[21] David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. Variational shape approximation. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 905–914, New York, NY, USA, 2004. ACM Press.

[22] David Cohen-Steiner and Jean-Marie Morvan. Restricted delaunay triangulations and normal cycle. In *SCG '03: Proceedings of the nineteenth annual symposium on Computational geometry*, pages 312–321, New York, NY, USA, 2003. ACM Press.

[23] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, New York, NY, USA, 1996. ACM Press.

[24] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 11–20, New York, NY, USA, 1996. ACM Press.

[25] Tamal K. Dey, Joachim Giesen, and James Hudson. Delaunay based shape reconstruction from large data. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 19–27, Piscataway, NJ, USA, 2001. IEEE Press.

[26] Tamal K. Dey and Wulue Zhao. Approximate medial axis as a voronoi subcomplex. In *SMA '02: Proceedings of the seventh ACM symposium on Solid modeling and applications*, pages 356–366, New York, NY, USA, 2002. ACM Press.

[27] Herbert Edelsbrunner. The union of balls and its dual shape. In *SCG '93: Proceedings of the ninth annual symposium on Computational geometry*, pages 218–231, New York, NY, USA, 1993. ACM Press.

[28] Mark Foskey, Ming C. Lin, and Dinesh Manocha. Efficient computation of a simplified medial axis. In *SM '03: Proceedings of the eighth ACM symposium on Solid modeling and applications*, pages 96–107, New York, NY, USA, 2003. ACM Press.

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

[30] Dan Gordon and Shuhong Chen. Front-to-back display of bsp trees. *IEEE Comput. Graph. Appl.*, 11(5):79–85, 1991.

[31] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238, New York, NY, USA, 1993. ACM Press.

[32] Greg Leibon and David Letscher. Delaunay triangulations and voronoi diagrams for riemannian manifolds. In *SCG '00: Proceedings of the sixteenth annual symposium on Computational geometry*, pages 341–349, New York, NY, USA, 2000. ACM Press.

[33] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 131–144, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[34] Peter Liepa. Filling holes in meshes. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 200–205, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[35] Dick Meeuwig. The mccoy mining district. *Nevada Bureau of Mines and Geology Quarterly Newsletter*, 1, 1995.

[36] Niloy J. Mitra, Natasha Gelfand, Helmut Pottmann, and Leonidas Guibas. Registration of point cloud data from a geometric optimization perspective. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 22–31, New York, NY, USA, 2004. ACM Press.

[37] Yutaka Ohtake, Alexander Belyaev, and Hans-Peter Seidel. An integrating approach to meshing scattered point data. In *SPM '05: Proceedings of the 2005 ACM symposium on Solid and physical modeling*, pages 61–69, New York, NY, USA, 2005. ACM Press.

[38] Marc Pollefeys and Luc Van Gool. From images to 3d models. *Commun. ACM*, 45(7):50–55, 2002.

[39] Samuel Ranta-Eskola. Binary space partitioning trees and polygon removal in real time 3d rendering. Master's thesis, Uppsala University, 2003.

[40] Robert A. Schumacker, B. Brand, M. Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, US Airforce Human Resources Laboratory, September 1969.

[41] D. J. Sheehy, C. G. Armstrong, and D. J. Robinson. Computing the medial surface of a solid from a domain delaunay triangulation. In *SMA '95: Proceedings of the third ACM symposium on Solid modeling and applications*, pages 201–212, New York, NY, USA, 1995. ACM Press.

[42] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, 1974.

[43] Mirela Tanase and Remco C. Veltkamp. Polygon decomposition based on the straight line skeleton. In *SCG '03: Proceedings of the nineteenth annual symposium on Computational geometry*, pages 58–67, New York, NY, USA, 2003. ACM Press.

[44] Greg Turk and Marc Levoy. Zippered polygon meshes from range images. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 311–318, New York, NY, USA, 1994. ACM Press.