1-1-2012

An Octree Surface Wrapping Algorithm to Recover Building Structures

Michal Trcalek

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation
https://scholarsjunction.msstate.edu/td/611

This Graduate Thesis is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.
An octree surface wrapping algorithm to recover building structures

By

Michal Trcalek

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Mechanical Engineering
in the Department of Mechanical Engineering

Mississippi State, Mississippi
December 2012
Copyright 2012

By

Michal Trcalek
An octree surface wrapping algorithm to recover building structures

By

Michal Trcalek

Approved:

Michael G. Remotigue
Associate Research Professor of Mechanical Engineering
(Thesis Director)

David McLaurin
Associate Research Professor of Aerospace Engineering
(Committee Member)

David Marcum
Endowed Professor of Mechanical Engineering
(Committee Member)

Steven R. Daniewicz
Professor of Mechanical Engineering
(Graduate Coordinator)

Sarah A. Rajala
Dean, Bagley College of Engineering
In the past twenty years, our world has experienced a number of disasters, ranging from hurricanes to acts of terrorism. While both natural and anthropogenic disasters are inevitable, being able to simulate their progression and impact can provide guidance for preemptive measures to mitigate casualties and property damage. Specifically, computational fluid dynamics (CFD) simulations can be used to simulate wind forces associated with hurricanes or pressure waves associated with explosions. However, even a simple CFD simulation is time consuming and requires highly-specialized expertise. This paper aims to reduce the processing time by utilizing readily available geometry models, and a surface wrapping algorithm that allows for fast and seamless way of repairing geometry. Both of these processes are automated which also reduces the amount of training in various software packages.
DEDICATION

This thesis is dedicated to my mom Jana Hurley and grandmother Hilda Filipcikova that have pushed me and encouraged me throughout my graduate career. They have supported me no matter what I needed, and I am very thankful for them.
ACKNOWLEDGEMENTS

I’d like to thank Dr. Remotigue for all his patience and support that he has provided to me. I’d also like to thank my committee members, Dr. Marcum and Dr. McLaurin for giving me valuable input on my work.
# TABLE OF CONTENTS

DEDICATION .................................................................................................................................................................................... ii

ACKNOWLEDGEMENTS ........................................................................................................................................................................ iii

LIST OF FIGURES .................................................................................................................................................................................... vi

CHAPTER

I.  INTRODUCTION ............................................................................................................................................................................. 1

II. PREVIOUS WORK .............................................................................................................................................................................. 4

  2.1 Marching cube algorithm .......................................................................................................................................................... 4
  2.2 Marching tetrahedral algorithm ............................................................................................................................................. 5
  2.3 Adaptive Cartesian grid generation ............................................................................................................................................ 6
  2.4 Summary ..................................................................................................................................................................................... 6

III. DATA TRANSLATION ......................................................................................................................................................................... 7

  3.1 Google Earth Data ....................................................................................................................................................................... 7
  3.2 Available File Formats ............................................................................................................................................................... 9
  3.3 Collada format ................................................................................................................................................................................ 9
      3.3.1 SketchUp format ............................................................................................................................................................ 10
      3.3.2 SURF file format ........................................................................................................................................................ 10
  3.4 Wavefront Object file translation ......................................................................................................................................... 11
      3.4.1 Quad Splitting ............................................................................................................................................................ 12

IV. WRAPPING ALGORITHM ................................................................................................................................................................. 14

  4.1 Pre-Processing ............................................................................................................................................................................ 14
  4.2 Octree generation ....................................................................................................................................................................... 15
  4.3 Geometry Integration ................................................................................................................................................................. 15
  4.4 Outer Boundary Recovery ......................................................................................................................................................... 17
  4.5 Surface Smoothing .................................................................................................................................................................... 17
  4.6 Projection .................................................................................................................................................................................... 19
      4.6.1 GridRx data structure .................................................................................................................................................... 20
      4.6.2 Calculating node normal vectors .................................................................................................................................. 20
4.6.3 Projection of nodes .................................................................20

V. COMPUTATIONAL EFFICIENCY .....................................................22

5.1 CPU usage .................................................................................22
  5.1.1 Node and triangle insertion ....................................................23
  5.1.2 Outer quad face extraction ....................................................23
  5.1.3 Smoothing .............................................................................23
  5.1.4 Projection ...............................................................................24
  5.1.5 Experimental data .................................................................24
5.2 RAM usage ..................................................................................25

VI. RESULTS ..........................................................................................27

6.1 File conversion .............................................................................27
6.2 Outer Surface Extraction ..............................................................28
6.3 Smoothing and Projection ..............................................................29
6.4 Resolved Issues ............................................................................30
6.5 Known problems ..........................................................................31

VII. CONCLUSIONS .............................................................................34

7.1 Future Work .................................................................................34

REFERENCES ......................................................................................36
<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Downtown New York in 3D from Google Earth [1]</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Intersection configurations [8]</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Intersection cases per Marching tetrahedra [6]</td>
<td>6</td>
</tr>
<tr>
<td>3.2</td>
<td>Node reconnection required for AFLR a) triangle b) quad</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>Triangle Insertion by point sampling</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>Effects of octant smoothing on a skull model acquired from Google’s 3D Warehouse</td>
<td>19</td>
</tr>
<tr>
<td>4.3</td>
<td>Projection Decision Tree</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>CPU run time for a) geometry Insertion and b) quad face extraction</td>
<td>25</td>
</tr>
<tr>
<td>6.1</td>
<td>Casino model a) in SketchUp and b) in SolidMesh</td>
<td>28</td>
</tr>
<tr>
<td>6.2</td>
<td>a) Stair stepped surface and b) Error due to point insertion</td>
<td>29</td>
</tr>
<tr>
<td>6.3</td>
<td>a) Smoothing after 200 iterations b) Smoothing with projection</td>
<td>30</td>
</tr>
<tr>
<td>6.4</td>
<td>a) Original geometry b) New geometry representation</td>
<td>31</td>
</tr>
<tr>
<td>6.5</td>
<td>a) Original geometry and b) new geometry representations of casino model</td>
<td>32</td>
</tr>
<tr>
<td>6.6</td>
<td>a) Failed corner projection and b) cross projected corner on casino model</td>
<td>33</td>
</tr>
<tr>
<td>6.7</td>
<td>Surface projected a) without smoothing b) with smoothing</td>
<td>33</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

In the past twenty years, our world has experienced a number of disasters, ranging from hurricanes to acts of terrorism. While both natural and anthropogenic disasters are inevitable, being able to simulate their progression and impact can provide guidance for preemptive measures to mitigate casualties and property damage. Specifically, computational fluid dynamics (CFD) simulations can be used to simulate wind forces associated with hurricanes or pressure waves associated with explosions. However, even a simple CFD simulation is time consuming and requires highly-specialized expertise. This paper aims to reduce the processing time, as well as simplify the simulation process, to make CFD simulations easier for first response teams to use when dealing with disasters.

The first step in a CFD simulation is to generate a usable grid that surrounds the geometry of the simulated domain. Although computer aided design (CAD) models are primarily used in product development, they are also useful in three-dimensional (3D) computer graphics (CG). Therefore, 3D CAD models can be used to represent the geometry in a CFD simulation. However, because CAD models are time consuming to build and require a working knowledge of the design software, utilizing already available models would reduce processing and user training times. For example, software like Google Earth [1] utilizes CAD models to render buildings within its graphical user interface. All of the models in Google Earth are available online in a directory named
Google 3D Warehouse [2], which includes models of everything from human body parts to vehicles and buildings available for download, most at no cost. Google Earth also includes topological locations of each building, which can be used to reconstruct a city block for a simulation. Figure 1.1 shows downtown New York as displayed in Google Earth.

The CAD model has to meet a number of criteria before a grid can be generated for the CFD simulation. The geometry is required to be watertight and free of any interior points, edges, or surfaces in order to successfully create a 3D grid. Grid generation is done using SolidMesh [3] and AFLR [4]. Geometry repair is often done manually and requires many man-hours, therefore increasing the cost of the simulation. This paper centers on automating the geometry cleanup process by using an octree algorithm that wraps around the geometry, thus mimicking only the exterior surface. The octree algorithm was selected to explore its flexibility of utilization.

An algorithm is devised to combine the benefits from utilizing both the CG CAD models and the octree algorithm to extract the outer surface. This algorithm is to be a robust tool to quickly produce usable geometries for CFD simulations. It will be a standalone function or incorporated into SolidMesh, which is needed to create the final grid.
Figure 1.1  Downtown New York in 3D from Google Earth [1]
CHAPTER II
PREVIOUS WORK

Using a Cartesian grid to extract the outer surface from geometry is not a new concept. The common known approaches to this problem are marching cube algorithm by Lorensen and Cline [5] and marching tetrahedral algorithm by Gueziec and Hummel [6]. Another algorithm discussed by Wang and Srinivasan [7] utilizes a painting algorithm on an adaptive Cartesian grid to extract the outer geometries.

2.1 Marching cube algorithm

Lorensen and Cline developed the original marching cubes algorithm in 1987 to create a polygonal mesh from an isosurface. Similarly, the algorithm can be used to extract an outer surface of geometry. Information about marching cube algorithms has been compiled into a survey by Newman and Yi [8]. Marching cube algorithm utilizes a lattice data structure that stores data at the vertices. Edges of the lattice are checked for intersection with the geometry. Vertices on intersected edges are marked in or out. Cubes that contain an intersected edge are activated. Each cube has 256 ($2^8$) different scenarios in which it can be intersected, based on eight vertices that can be marked in two ways. The 256 scenarios can be reduced down to 15 by assuming rotational symmetry of the cubes. The final 15 scenarios are displayed in Figure 2.1. Standard marching algorithm relies on linear interpolation to determine the intersection of the surface and edge. Once the intersection point is calculated it can be applied to the four cubes surrounding the
containing edge. The last step in the marching cube algorithm is to reconnect the points in each cube to extract the new surface.

![Intersection configurations](image)

Figure 2.1  Intersection configurations [8]

### 2.2 Marching tetrahedral algorithm

This algorithm, developed by Gueziec and Hummel, utilizes tetrahedral elements to recover the exterior boundary of surfaces in medical imagery. Surfaces in this algorithm are represented by an intensity function \( I(x,y,z) \). Similar to the marching cube algorithm, a lattice data structure is formed. Each cube within the lattice is split into 5 tetrahedra elements. Points of each tetrahedron are submitted to the intensity function, point is declared as in or out based on the result. Tetrahedra containing both interior and exterior vertices are intersected by the surface. Bilinear interpolation is used to find the intersection point on an edge containing mixed vertices. Based on cases shown in Figure 2.2, a triangle or two are created in each intersected tetrahedron and reconnected with the others to form a representation of the exterior surface.
2.3 Adaptive Cartesian grid generation

Method developed by Wang and Srinivasan seeks to eliminate dirty geometry by extracting the outer boundary, and generate usable grids for CFD applications. Initially the Cartesian grid is developed and the geometry is integrated by an intersection operation. Intersected cells are determined within the Cartesian grid. Cells are refined recursively until cells are smaller than a specified threshold; this guarantees a minimum grid resolution at the geometry. A cell is selected on the interior of the geometry and a neighbor-painting algorithm is to identify all cells in the computational domain. Cells that are intersected and unpainted cells are removed. Exposed Cartesian faces are projected onto the original geometry.

2.4 Summary

While marching cube and marching tetrahedra algorithms are excellent in surface recovery, processing time and memory requirements pose a problem. The adaptive Cartesian algorithm is similar to the proposed algorithm; however, the details of the boundary recovery, as well as the smoothing approach are unclear. This paper aims to clear up the boundary recovery by using an octree, and implement a modified Gaussian smoothing algorithm to eliminate undercutting.
3.1 **Google Earth Data**

Google Earth is an Internet tool available worldwide that allows the general public to view complete earth topography in 3D from any angle. In the past few years, Google Earth has incorporated support for storing and displaying 3D models, including buildings, within its interface. Most of the 3D models utilized are built using SketchUp/BuildingMaker [9]. Once a model is generated, it can be incorporated into the Google Earth interface. Most of the models are also stored in Google Earth’s online 3D warehouse and available to download for free. Concentration of this paper is mainly on building cases; however, the online storage contains a wide variety of modeled objects, including vehicles, household items, and even people.

Google Earth does provide online the minimal requirements [10] at which models will be accepted for use in its interface. These standards, however, do not specify how the model should be built; instead, they merely indicate that the model should extend all the way to but not protrude under the ground. It is important to note that Google Earth also includes topography, including land elevation; therefore, many of the buildings do not have an even ground plane but one that resembles the local topography. This phenomenon can be seen in the Figure 3.1 below, which shows that the ground plane is not flat as the elevation increases toward the rear of the building.
Virtually anyone can create the models for Google Earth because the tools are designed to be user-friendly and free to access. However, because unfunded parties generally create the data, a high degree of accuracy should not be expected. Furthermore, Google Earth is not concerned with geometry in so much as the visualization of buildings. Thus, Google Earth prefers the geometrical data to be simpler and use textures for representing the detail on buildings. Accordingly, their models will require less memory and render faster. From a simulation point of view, this data may not be ideal, for an accurate representation of a model is crucial to any simulation. Nonetheless, when the goal is to produce a quick and easy approximation of a simulation, the Google Earth geometry becomes attractive for reducing the amount of time and cost needed to develop models.

Figure 3.1  US Capitol building [11] model in SketchUp [9]

Note that in SketchUp the green area represents the ground plane and the figure shows how the building extends below it.
3.2 Available File Formats

Note that not all buildings are available for download in Google Earth, but those that are not available usually have other versions stored in the 3D warehouse that are available for download. Downloading is simple, just requires specification of a desired format. The supported formats are SketchUp file format (.skp) [9], actual Google Earth format (.kmz) [1], and Collada format (.dae) [12], most common is the SketchUp format. Two of the formats, .skp and .kmz, are binary and cannot be directly accessed through any text editor. The Collada .dae file is an XML [13] style format that can be read within a scope of any text editor.

For the scope of this paper, a format had to be selected and translated to a SURF [14] file format suitable for use in SolidMesh. SURF file format was chosen, because it provides the necessary information and it is closely related to the Wavefront object format [15]. The object format is a necessary stepping-stone in the translation to the SURF file format.

3.3 Collada format

The original idea was to make the translation completely seamless and without any interaction from the user. The Collada format was chosen because it was the only available non-binary format. While writing the translator for this format, many inconsistencies were discovered among the different files tested. Some files contained multiple groups, while others had just one; some contained normal vector information, while others did not; and the reconnection of the faces was difficult to decipher.

Because just parsing these files as text formats proved to be highly cumbersome, use of free online available libraries was attempted. The libraries chosen were ColladaDOM [12], which contained some useful functions but unfortunately no direct
function to get just nodes and triangular reconnection as needed. Because the parsing of this file format was not successful, an easier solution was found in a software named Blender [16], which allows the user to import .dae Collada format and translate it into an object file.

3.3.1 SketchUp format

One of the Google Earth’s options is to download a .skp, SketchUp file format. This is a binary format that cannot be directly translated; however, an object file can be created using SketchUp. Exporting the geometry as .3ds format [17], before the file conversion creates an extra step in the operation; however, the resulting format will be strictly triangulated. Triangulated mesh geometry representations are desirable because they do not have to be further split for the wrapping algorithm. Octant splitting is a feature of the object translation.

3.3.2 SURF file format

The algorithm herein described is to be a tool for the SolidMesh software package, and therefore the file formats must be compatible. SolidMesh accepts many formats, but none are available from Google Earth. The standard file format to represent surface geometry is the SURF file format. This format is a non-binary representation of geometry, where the first line consists of the total number of triangles, total number of quads, and total number of nodes.

The initial declaration of quantities of all the components in the SURF file format is followed by a declaration of all the nodes. Nodes are defined as X, Y, and Z coordinates; furthermore, initial normal spacing can be specified per node. After all of the nodes are declared, the actual faces can be constructed. The triangles are constructed first,
using numerical indices to specify which nodes govern each triangle. Quad faces are represented in a similar fashion. The SURF file format also provides an option to specify a group tag, which is useful if multiple groups of data are represented. The reconnection required for the AFLR is shown in Figure 3.2 below. This reconnection also helps by specifying a direction in which to calculate the normal directions.

Figure 3.2 Node reconnection required for AFLR a) triangle b) quad

Note: Figure acquired from:
http://www.simcenter.msstate.edu/docs/solidmesh/ugridconnectivity.html

3.4 Wavefront Object file translation

Using the object format as a stepping-stone between file formats downloaded from Google Earth and files required for SolidMesh proved to be the most robust approach. Object files (.obj) can be created using SketchUp or Blender, and they are an easily understandable file format that clearly specifies vertices, faces, and normals.

The object file format is a non-binary format that consists of easy-to-read lines, each specifying an entity or a property of the geometry. In addition to the specified entities, the texture of the face or the group assignment can be specified. The texture property is not essential to the geometry or our application, and therefore it is discarded.
Geometry files tend to be very large according to the level of detail incorporated; therefore the translator should be very efficient in the way it parses through the file. The first step is to take each line and determine what entity or property it describes. Once the type is determined, a simple logic is determined to store the entity in an appropriate standard template library (STL) vector [18]. The STL is a software library that defines containers and data structures. An STL vector container is used because it can be dynamically allocated and randomly accessed, which is needed because the object file does not specify a total number of each entity at the top of the file. The translator stores a container for vertices, triangles and quads. Normal vectors for each of the nodes of the original geometry can also be stored during the file parse. By default the normals are not stored but could be used later to determine the orientation of elements occupying the same octant. The algorithm is fast due to the fact that all of the containers are filled after a single parse through the object file. The entities are written into the SURF file format, which consists of vertices and element connectivity.

3.4.1 Quad Splitting

The surface-wrapping algorithm assumes a discrete representation that is composed of strictly triangular elements. Splitting quads was simpler than incorporating a quad logic into the wrapper algorithm. For future use, the quad logic should be incorporated into the wrapper algorithm to be able to accept a more general type of geometry.

Each quad can be split in no more than two different ways. The quality is already poor, but maximizing the minimum angle is a good approach to get decent triangulation from the quad. There are more sophisticated algorithms for quad splitting, such as the
Delaunay algorithm. This geometry, however, would not benefit from the Delaunay algorithm, as it is for representation purposes only and will not serve as a grid.

Maximizing the minimum angle is the implemented technique, where each quad is split in two different configurations, and the angles are calculated in each triangle. The minimum angles for both configurations are compared, and whichever configuration offers the highest minimum angle is chosen. Each new triangle adopts the group tag from the quad surface and is pushed onto the triangle STL vector. Degenerate quads contain only a single reconnection option, and they are not considered within the scope of the quad splitting.
CHAPTER IV
WRAPPING ALGORITHM

An algorithm is developed utilizing an octree to locate and extract the outer surface of a prescribed geometry. After the initial preprocessing, an octree is built around the existing geometry. The original geometry is located within the octree by point insertion. A voxel map is built at the terminal octant level, and a painting algorithm is used to extract the outer boundary. Smoothing and projection algorithms are applied to the new surface to eliminate stair stepping and recover details, respectively. The resulting surface represents a grid suitable for volume grid generation using AFLR.

4.1 Pre-Processing

Geometry is introduced to the algorithm in the form of the SURF file. The surf file format is described in detail in section 3.3.2. GridRx [19] is used to eliminate invalid triangles with area equal to zero, as well as glue triangles based on a specified tolerance. Some models contain separate triangles rather than a continuous surface; in these cases, GridRx is utilized for gluing. The SolidMesh++ [20] library is used to generate node-around-node, element-around-node, and element-around-element maps. Elements are formed into groups by traversing the element around element map. Grouping is useful upstream if an analysis is to be done only for a specific group.
4.2 Octree generation

The octree is a structured and robust algorithm. Octants are readily accessible due to the recursive parent/child relations. The octree is relatively inexpensive computationally; however, the memory requirements grow quickly. The memory requirements rise with a factor of eight at each new level, as each octant splits into eight.

The first step is to create a box to fully enclose the given geometry; this bounding box will stretch from the minimum value to the maximum value in each dimension. To ensure that all of the geometry is included in the bounding box, the box is scaled up by one percent. Most models that are considered in this project are buildings with a nominal ground plane; therefore, the bounding box is shifted in the z direction. Shifting the bounding box will ensure that the algorithm will not advance into the building and mark any unwanted cells. Some geometry does not have a flat surface for a ground plane, and for these special cases, a different approach should be used to eliminate the interior cells.

The bounding box is scaled to be divisible by the cell size and to ensure it forms a cube. Cell size is input by the user, or by default it is based on an average edge length of an element in the geometry. Each octant beginning with the root is split recursively until the desired octant size is reached if it is occupied by any element of the geometry. Each octant stores vital information, such as position, level, neighbors, etc., within the octree.

4.3 Geometry Integration

The geometry is integrated into the octree by point insertion. Triangle edges are discretized using 30 percent of element size as a criterion. Higher percentages were tested, but they yielded poor boundary recovery, due to insufficient point insertion. New points are connected to the centroid of the triangle by line segments, which are also discretized with the new criterion. Each new point is added to an STL stack container
along with the geometry nodes. Each node from the stack is inserted into the octree. The octree splits octants that contain the inserted node to the desired resolution. Octants at the highest level are terminal octants and always contain an instance of the geometry. Figure 4.1 below shows a 2D version of the point insertion. This figure shows points that have been inserted along with how the quad tree is split. The solid blue squares represent the terminal octants. Another thing to notice about the figure is the undercutting that occurs along the edges. Reducing the discretization criterion can eliminate undercutting.

Figure 4.1  Triangle Insertion by point sampling
4.4 Outer Boundary Recovery

The outer boundary is recovered by a painting algorithm, which requires all octants to be at the same level. As mentioned in section 4.2, due to memory requirements of the octree a simple terminal voxel map is used. To further reduce memory requirements of the voxel map, a bounding box around the terminal octants is found. The voxels are imprinted with the corresponding terminal octant if it contains data. The painting algorithm advances from the corners of the voxel map through the face neighbors. When the painting algorithm intercepts a voxel that contains geometry, the face between them is ‘wetted’ and the voxel is marked as outer. All ‘wetted’ faces are combined to the new outer surface. Nodes and quads that compose the new surface are stored in an STL vector for easy access. Neighboring nodes are searched for saved nodes to ensure connectivity.

4.5 Surface Smoothing

A modified Gaussian smoothing algorithm is implemented to eliminate the stair stepping or faceting associated with the surface that is recovered from the voxel representation. The smoothing algorithm is described by Taubin [20]. For a given node, the first step is to calculate vectors to all neighboring nodes. Then, a weighted average is calculated from the vectors. This principle is shown in equation 4.1 below, where $w_{ij}$ are the weights calculated based on the surrounding geometry and sum of which is one.

$$\Delta v_i = \sum_{j \in \ell} w_{ij} (v_j - v_i)$$

A displacement vector for each node is calculated once all the vector averages have been determined. The displacement vector is calculated as a product of itself and a weight. The weight is based on the number of surrounding nodes in this implementation. The equation for the displacement vector is shown in equation 4.2 below. Taubin
suggests using alternating scale factors between iterations based on low pass filtering to eliminate undercutting. A low pass filter can be designed to determine the scale factors $\lambda$ and $\mu$. At this time the factors are chosen with respect to the prescribed conditions, which are shown in equations 4.2 and 4.3.

\[ v_i' = v_i + \lambda \Delta v_i \text{ with } 0 < \lambda < 1 \quad (4.2) \]
\[ v_i' = v_i + \mu \Delta v_i \text{ with } 0 < \lambda < -\mu \text{ & } 0 < -\mu < 1 \quad (4.3) \]

Surface smoothing is utilized to produce a surface representation for the model or to aid in projection. Figure 4.2 below shows the effects of smoothing on a skull [21] model acquired from Google’s 3D Warehouse. Smoothing behavior is highly affected by the number of iterations used. There were 0, 50, 500, and 5000 iterations used in a, b, c, and d respectively and the scale factors $\lambda$ and $\mu$ were 0.33 and -0.34 respectively in all cases.
Figure 4.2 Effects of octant smoothing on a skull model acquired from Google’s 3D Warehouse

Note the edge cutting as the iterations reach 5000.

4.6 Projection

The smoothing algorithm eliminates stair-stepping and provides a smooth surface; however, the surface is offset from the original geometry, and sharp edges cannot be recovered. Projection of the nodes can be utilized to produce an accurate representation of the original geometry. The projection is done by tools provided within GridRx [19], which uses a directional ray-casting algorithm to project each point.
4.6.1 **GridRx data structure**

*GridRx* requires a unique data structure that builds a separate octree from the one used previously to generate the quad surface. This octree data structure is used for efficient searching and uses a balanced distribution of geometric information across the octree. The original geometry is integrated into the octree by inserting a *grx* surface mesh. Inserting nodes and triangles from the original geometry generates the surface mesh to be inserted into the octree. The octree has an option to split to a prescribed level, or it can logically split only those octants that contain more than one data point. Memory requirements are reduced by choosing the second option.

4.6.2 **Calculating node normal vectors**

The projection algorithm requires a direction in which to project the node. The direction of projection is determined by calculating the average of the normal vectors that belong to the faces surrounding the node in question. *SolidMesh++* generates maps of faces around each node, and therefore acquiring information about the faces surrounding a particular node is simple and easily accessible.

The normal vector for each face is calculated by a cross product of the diagonals of each quad face. This calculation is executed under the assumption that each quad is planar, which is correct if no smoothing is applied. Furthermore, the orientation of the quad can be obtained from the vertex numbering order and the right hand rule.

4.6.3 **Projection of nodes**

The *GridRx* uses a ray-casting algorithm coupled with the octree data structure to quickly determine the coordinates of the projected node. Correct direction is essential to the projection function. Accordingly, if the node lies within the original geometry the
direction must be reversed. GridRx will return an error if there are no surfaces in the specified direction. When an error occurs or if the distance between the original node and the projected node is larger than a diagonal of the terminal octant, the opposite direction is tested for better results. If the distance criterion fails in both directions, the node is not modified. A projection decision tree is devised, as shown in Figure 4.3. Projected points are stored separately from the original quad points and are added once all of the projections have been completed.

Figure 4.3 Projection Decision Tree
CHAPTER V
COMPUTATIONAL EFFICIENCY

Computational efficiency is necessary for this algorithm to function as it is intended: as a fast and seamless way of obtaining usable geometry. The data acquired from the 3D Warehouse is often simple in order to aid rendering inside Google Earth; however, some of the geometry does contain large amounts of data and complexity, and thus requires small element size to resolve it. Small elements are achieved by subdividing the octree to the highest level of resolution. It is this level of resolution that proved to be a computational bottleneck for the algorithm. In fact, only the 9th level could be achieved, after which available memory limits were exceeded. Furthermore, in respect to processing power, the number of points on the quad surfaces increases at least four times with the addition of each new level, thus the projection of these points is the major contributor to required CPU time.

5.1 CPU usage

Portability of this algorithm is important; therefore, the algorithm should have low CPU requirements and execute quickly. Major contributors to the computational time are node and triangle insertion, outer quad face extraction, smoothing, and projection. Timing of each step was obtained with a CPU ticker.
5.1.1 Node and triangle insertion

Node and triangle insertion are the most significant contributors to the final computation time. Nodes are inserted one at a time, ensuring that the octree is split to the terminal level at the node location. Triangles are inserted using the sampling algorithm, which discretizes each triangle into a number of points. These points are then inserted into the octree. High resolution of each triangle requires more points to be sampled and, in turn, requires additional CPU time. Therefore, this aspect of algorithm is highly dependent on the physical size of the geometry as well as level of detail to be captured. Furthermore each additional level of octree resolution adds an additional step to the searching algorithm, which occurs for every node that is to be sampled.

5.1.2 Outer quad face extraction

This step includes the voxel creation and the painting algorithm. Terminal octants have to imprint their information onto the overlaying voxels. Imprinting consumes time and CPU power with respect to the number of terminal octants. Voxels are also created in space that does not contain any data; reducing the voxel map region as described in section 4.4, has minimized this.

The painting algorithm traverses the voxel map in search of the geometry. The algorithm advances from each corner to ensure complete painting of the exterior. Although necessary, this process is time consuming and memory extensive with respect to resolution.

5.1.3 Smoothing

This algorithm uses Gaussian smoothing to approximate each point location based on the surrounding faces. The implemented algorithm requires many iterations as it
alternates between positive and negative coefficients to execute the smoothing. Therefore, the time required for the smoothing algorithm will be based not only on the number of points to be smoothed but also, more importantly, on the number of iterations.

5.1.4 Projection

The implemented projection algorithm requires a new data structure to store the geometry, which includes a new octree specifically designed for efficient searches. This algorithm splits the tree only when there are more than a single data point within the octant, which drastically reduces the computational time needed. It should be noted that the previous octree has been deleted at this point. Once the new data structure is formed, each of the nodes on the new surface has to be projected. The projection itself involves a ray-casting algorithm, which requires searching through the generated octree until an intersection of the mesh is reached. The projection algorithm may also be called twice because it cannot decipher whether the node is on the inside or outside of the geometry. Because of the relatively high number of nodes or operations to project, this process contributes the most computational time.

5.1.5 Experimental data

The machine used for the development was a MacBook 2,1 with 3GB total RAM, 2GHz Intel Core 2 Duo processors, and Mac OS X 10.6.8. 32bit version operating system. The algorithm was also implemented on Linux computer to explore higher resolution level due to more available memory. Scalability of the algorithm was also observed from the implementation. The following charts in Figure 5.1 have been constructed using the Skull model from Figure 4.2. Various levels of resolution were tested on the model, and the results were combined in a bar chart for both mac and linux
systems. Times for each operation were obtained using a CPU ticker. Charts in Figure 5.1 show that the time required for each operation grows exponentially with the increase of the level of resolution. The memory limit was exceeded with an attempt to run level 10 on the Linux machine and with level 9 on the mac.

![Geometry Insertion](image1)

![Quad face extraction](image2)

Figure 5.1 CPU run time for a) geometry Insertion and b) quad face extraction

Note that for this experiment, chosen resolutions were 0.002, 0.001, and 0.0005 for levels 7, 8, and 9, respectively. It is also notable that the mac crashed during the quad face extraction at level 9 due to insufficient memory.

### 5.2 RAM usage

Memory consumption was the limiting factor in the development of this algorithm. The octree is very expensive in terms of memory because it stores information of each level. The following steps reduced the memory requirements: (1) Generate a higher-level octant only if it contains data, and (2) once terminal octants are established, the octree is replaced with a Cartesian voxel map. At this point, the number of voxel
elements becomes the bottleneck; this number was then reduced by creating a new bounding box around the geometry as explained in section 4.4. These improvements helped achieve higher resolutions.
CHAPTER VI
RESULTS

The goal of the proposed algorithm is to extract a usable geometry representation of the outer surface of building structures to be used for quick visualization. A known casino [22] building has been downloaded as an example to show the performance of the algorithm.

6.1 File conversion

The casino CAD model was acquired in SketchUp format from 3D Warehouse by Google. SketchUp software was used to export the file to a Wavefront .obj object file format. The object file provided input for the file converter to produce a SURF file that can be used for the wrapping. Note that models are converted from Y-up to Z-up orientation during the file conversion process, and respective groups are maintained within the model. Figure 6.1 below shows a model of the casino in a) SketchUp and b) SolidMesh.
The model is submitted to the wrapping algorithm for outer surface extraction. The outer stair-stepped surface is generated with respect to a specified terminal octant size, which, in this case, was chosen to be 58 in order to reach level eight. Although the specified grid resolution at level 8 may not be enough to resolve all the details involved in this model, higher levels require unreasonably large run times. A shift in the Z direction is applied to signify a common ground plane for the model and the octree.

Point insertion by sampling the geometric triangulation does not guarantee that all the points on the wetted surface will lie outside of the geometry. Figure 6.2 below shows the generated quad surface and also shows the protrusion of the original geometry through the new surface, which occurs due to the sampling. Figure 6.2 below was generated using triangle sampling at 50 percent of the edge length and a level 8 octree resolution.
6.3 **Smoothing and Projection**

Smoothing involves a modified Gaussian algorithm to relieve the stair stepping within the new surface. Smoothing is beneficial to objects that containing continuous curvature; however, smoothing can pose problems for geometries with sharp corners and edges. The casino model contains many sharp edges, which is typical for building structures; however, a low level of smoothing (approximately 200 iterations) is utilized to aid in projection by orienting faces around each node.

The projection occurs to recover the outer boundary of the geometry. Nodes on the new quad surface are projected onto the original geometry using a directional ray-casting algorithm. Because the new surface is not guaranteed to be on the outside of the geometry, projection should occur in both directions and results should be compared. Resolution is crucial to the projection algorithm to recover detailed geometry. Figure 6.3 shows the new surface after smoothing and projection algorithms.
Figure 6.3  a) Smoothing after 200 iterations b) Smoothing with projection

6.4  Resolved Issues

As a result of the wrapping algorithm, the interior geometry has been eliminated, as shown in Figure 6.4. Overlapping regions in the geometry have also been resolved, and the result is a single, continuous representation of the original geometry. The new geometry is watertight with the exception of the ground plane.
6.5 **Known problems**

The major problem is recovering details within the geometry. Higher resolution could be used but becomes expensive computationally if used across the entire model. Figure 6.5 shows how details are treated within the model. The ledges of the model are not recovered accurately due to their small size relative to the resolution.

Corners and sharp edges pose a problem for the smoothing algorithm and the projection. The smoothing algorithm eliminates corners and sharp edges and therefore fails to recover them. The projection algorithm fails to capture corners and sharp edges in two different ways. First, the projection could miss the corner based on the specified normal and project to another part of the surface, which would lead to projection failure. Second, the corner could be cross-projected, which would result in invalid quads on the surface. Corner projection failures are shown in Figure 6.6.
Another problem has been experienced when using the model of a human skull. Pimples, cells that have five ‘wetted’ faces, pose a problem during projection. Pimples often cross project and form invalid quads on the surface. Smoothing can eliminate pimples. Figure 6.7 below shows cross-projected quads if no smoothing is applied, and also the same region with smoothing implemented before the projection.

Figure 6.5 a) Original geometry and b) new geometry representations of casino model
Figure 6.6  a) Failed corner projection and b) cross projected corner on casino model

Figure 6.7  Surface projected a) without smoothing b) with smoothing

Note the black spots in a) are unresolved quads, where the normal can not be determined
CHAPTER VII
CONCLUSIONS

The goal was to reduce time requirements and complexity associated with preparing a model for CFD analysis. Time was reduced by a procedure that allows for use of readily available 3D models in CFD simulation. In this procedure, files were translated into the appropriate format and the geometry was cleaned up using a surface-wrapping algorithm. The procedure was tested on multiple geometry models and the results were presented. Efficiency and scalability of the process were also presented. The resulting procedure could be utilized to quickly obtain useable geometries for CFD simulations.

7.1 Future Work

Utilizing an adaptive voxel map would allow for increased resolution near the geometry boundary. Increased resolution could be used to capture detailed features of the geometry for a more accurate representation of the geometry. Allowing for adaptation will allow for the maximum resolution in large expanses and thus reducing the memory requirements but increasing the complexity of the surface recovery.

Consider an intersection algorithm rather than point sampling to locate the geometry within the octree. The two algorithms should be compared in terms of run time and efficiency. An intersection algorithm would allow for faster determination of terminal octants for cases with high resolution as well as eliminate undercutting associated with the sampling algorithm.
Many current problems arise when projecting the new nodes onto the original surface, and therefore the projection methods should be further studied and refined. Sharp corners and edges are problem areas for projection due to the normal vector failing to intersect with the edge or a corner. Sharp corners and edges could be identified using feature detection, followed by a corrective action such as cell elimination or addition, face collapse or cell splitting.

Time studies should be done comparing the proposed algorithm with the previously studied approaches of the marching cube algorithm and the tetrahedral intersection. Projection in the current algorithm can be quite time consuming depending on the number of nodes to project, and therefore the marching cube algorithm could possibly provide faster results with higher accuracy.

Another way the algorithm could be refined is to add ability handle meshes containing quad elements. As of right now the quad elements are being split during the translation from the object format, but perhaps a SURF file that contains quad faces should also have the ability to be inserted into this algorithm.
REFERENCES


