"Teddy" Extended:
A New Method For 3D Content Creation

by

Michael Bernard Rudolph

B.S., University of Colorado at Colorado Springs, 2004

A project submitted to the Graduate Faculty of the
University of Colorado at Colorado Springs
in partial fulfillment of the
requirements for the degree of
Master of Science
Department of Computer Science

2010
This project for the Master of Science degree by

Michael Bernard Rudolph

has been approved for the

Department of Computer Science

by

______________________________
Sudhanshu K. Semwal, Chair

______________________________
Bonnie Snyder

______________________________
C. H. Edward Chow

______________________________
Date
Abstract

This thesis project proposes a software solution that provides an interface more akin to traditional artistry. With the system proposed, a user will be able draw a simple two-dimensional sketch of a figure or object and convert it into a 3D object without having to rebuild the figure from scratch in a 3D development environment. Because such sketches are crude in nature, the newly created figure can be refined by carving details from it using a haptic device, much as a sculptor does with stone or clay. This thesis project details the problems in developing such a system, and how their solutions were addressed.
CONTENTS

INTRODUCTION ................................................................................................................. 1

PREVIOUS WORK ........................................................................................................ 3

DESIGN ............................................................................................................................. 19

IMPLEMENTATION ...................................................................................................... 23

FUTURE WORK .......................................................................................................... 32

CONCLUSION ............................................................................................................. 35

BIBLIOGRAPHY ........................................................................................................... 36
FIGURES

Illustration Index

Simple 10-point shape...................................................................................................................... 7
Splits of 10-point figure.................................................................................................................... 7
Example from Teddy.......................................................................................................................... 7
Splits in Teddy example..................................................................................................................... 7
Original convex polygon.................................................................................................................. 8
Splitting points into LR subsets....................................................................................................... 8
Draw edges and begin merging low-level subsets........................................................................... 8
Base edge and initial candidates....................................................................................................... 9
Circumcircle of base edge contains next left candidate.................................................................... 9
Left edge deleted from triangulation............................................................................................... 9
Selecting a new base point.............................................................................................................. 10
New base point drawn...................................................................................................................... 10
Left and right subsets merged.......................................................................................................... 10
CDT of sub-polygon.......................................................................................................................... 10
Completed Triangulation................................................................................................................... 11
Initial Chordal Axis.......................................................................................................................... 11
Pruning algorithm.............................................................................................................................. 12
Final triangulation after fanning extrema.......................................................................................... 12
Polygonal Mesh Construction........................................................................................................... 13
"Humps" form with increasing size factors......................................................................................... 14
Igarashi triangulation....................................................................................................................... 17
Before B-Spline smoothing.............................................................................................................. 17
After B-Spline Smoothing.................................................................................................................. 17
"Bean" triangulation......................................................................................................................... 17
Before B-Spline smoothing.............................................................................................................. 17
After B-Spline smoothing.................................................................................................................. 17
Optimal vertex alignment.................................................................................................................. 19
Typical vertex alignment.................................................................................................................. 19
SECTION I
INTRODUCTION

Current tools for three-dimensional content creation, such as Maya and 3dsMax, are capable of creating 3D models and environments with stunning realism. Animated models have become more lifelike over the years as techniques modeling flesh, fur, and cloth have improved. Improved techniques in terrain generation have allowed for the creation of outdoor landscapes that seem as real as any forest. While these software solutions are indeed powerful, their interfaces can be overly complicated. Modelers generally must undergo years of training to become proficient enough with these programs. These tools usually have interfaces based on those of CAD programs. While such interfaces lend a great deal of precision to creation, they are often stifling to the creative process.

The interface proposed here is one more akin to traditional artistry. With it, the user will be able draw a simple two-dimensional sketch of a figure or object and convert it into a 3D object without having to rebuild the figure from scratch in the 3D development tool. Because such sketches are crude in nature, the newly created figure can be refined by carving and extruding details from it using a haptic device, much as a sculptor does with stone or clay. For this particular project, the interface will make use of the Novint Falcon, an inexpensive and easy to use haptic controller that provides good haptic realism in three degrees of freedom.
Creating such a tool presents many challenges. To present a three-dimensional model for a convincing clay modeling simulation, the initial mesh should be smooth enough to appear to be a typical clay figure. Artifacts in the 2D/3D conversion process should not create unrealistically jagged edges. Once this is completed, the haptic model will have to be rendered in parallel with the graphical rendition. A significant observation in [1] is that if there is inconsistency between what the user sees in the graphical view and the forces being sent back through the device in the haptic model, the user will tend to give more credence to what they see in the graphical representation. In their example depicting a sword striking a shield, they say that the user's perception will tolerate a split-second difference between when the sword hits in the graphical view and when the force is felt. But it is important for the correct force to be sent well before the user sees the sword travel through the shield. While the graphical view only needs to be refreshed 30 times per second to give the user smooth animation in rendering, the haptic model will have to refresh itself about 1,000 times per second to provide the user with a convincing force feedback experience. Thus the haptic model will have to be efficient enough to perform its calculations at close to this rate.

The next section will describe the history and theory of haptic rendering, along with previous work leading up to this thesis project. Section III will cover the design of the sketchpad and haptic interfaces. Section IV discusses the implementation details of integrating the Novint Falcon with Java using the Java Native Interface. Section V contains ideas for possible future work. Section VI is the final summary and conclusion.
SECTION II
PREVIOUS WORK

Sketchpad interfaces have been researched extensively over the past decade. Systems such as Virtual Sculpture [2] and inTouch [3] have allowed for the creation of 3D models by deforming them with tools of different shapes, much as one would with fingers or a crafting tool. There are also many systems which allow the use of the haptic device to paint both color and haptic texture data on the object in question. One of the demonstrations for ArtNova [4] allowed the creation of different textures on a cow figure, denoting the different cuts of beef. Currently, the only commercial applications created for haptic 3D modeling are FreeForm and ClayTools, both developed by Sensable Technologies, for use with their PHANTOM force feedback devices. FreeForm is the software suite oriented toward modeling in CAD, while ClayTools is a utility geared toward use for 3D modeling apps such as Maya and 3dsMax.

The main inspiration for this project is the work done by Igarashi on Teddy [5]. In Teddy, the user drew a single non-intersecting stroke on a pad. A Constrained Delaunay Triangulation of the 2D sketch would then be calculated, and then the program would construct a chordal axis, or "spine" from the midpoints of the resulting triangles. Two sets of points on the chordal axis would then be raised and lowered a distance proportional to the distance from the sketch boundary. Finally, the 3D mesh of the sketch is constructed by connecting the chordal axis points to the boundary points, and stitching adjacent edges together. The resulting mesh, while
aesthetically true to the shape of the polygon drawn, is often rough, particularly after detail is added by carving from the mesh. Research has been conducted to refine the 3D meshes of models generated by using techniques such as least squares and quadratic fitting to smooth out artifacts generated by the object's creation and subsequent modifications.

The sketchpad window captures user input by recording the xy-coordinates of mouse events captured on it (specifically, when the user clicks the mouse, drags it, and when they release the mouse button). The program then first analyzes whether or not the stroke intersects itself. If it does, the polygon is not drawn, and an “invalid stroke” message is displayed to the user. If a valid closed loop stroke is drawn, the set of points collected is filtered, so that successive points in the polygon are a minimum distance apart from each other. Once the points to be included in the polygon are processed, the polygon is drawn and ready for the next step, which is triangulation.

A Delaunay triangulation is often described to be the most aesthetically optimal triangulation of a set of points. An edge between two points in a set can be included in a Delaunay triangulation if and only if a circle can be drawn such that the two points of the edge lie on the edge of the circle, and no other points in the set lie within the circle's boundaries. The geometric double of a Delaunay triangulation for a set of points is a Voronoi diagram, which is the set of edges defining regions which only contain one point each [6]. Each region in a Voronoi diagram is as large as it possibly can be while accommodating the previous assertion. Coincidentally, the set of edges of a Voronoi diagram are perpendicular to those found in a Delaunay triangulation.
The Delaunay triangulation of a set of points has the property of having a minimal number of acute triangles in the point set for the polygon, thus giving a very uniform looking triangulation. A constrained Delaunay triangulation (CDT), is a specialized instance of this, in which a set of edges that must be included are added to the polygons point set. There is also the stipulation that no edges may intersect each other, except at the points in the set. Because these constraining edges are not necessarily Delaunay edges themselves, the CDT of a polygon is not a perfect Delaunay triangulation, but the closest one possible given its constraint edges.

Most Delaunay triangulation algorithms work well for a set of points that are meant to define a convex polygon. Without any edges to define the outer perimeter of the polygon, these algorithms will create one with the points they receive as input, often with some of the points lying inside the polygon boundary. However, the set of points defined by the user of this interface will always define the outer boundary of a polygon, so these algorithms need to be modified to create an approximate CDT of the input shape rather than the best triangulation of a convex polygon. Given that this software will be used as a quick sketchpad, there is no reason to believe that the user will be constrained to drawing only convex polygons. On the contrary, there will often be a degree of concavity in the polygon, despite the user’s best intentions to draw a convex polygon similar to a circle. The scheme devised for this software project obtains a CDT of user input by first splitting the likely concave polygon into a set of convex polygons. The CDT’s of these sub-polygons are then computed and joined together in the final triangulation of the polygon.
A simple method was devised to determine how to split a concave input polygon into a series of convex sub-polygons. Given a sequence of \( i \) points that define the outer boundary of a polygon, \( p_1, p_2, \ldots, p_n \), the determinant about each point \( p_k \) is calculated in the following matrix:

\[
\begin{vmatrix}
 p_{(k-1)x} & p_{(k-1)y} & 1 \\
 p_{kx} & p_{ky} & 1 \\
 p_{(k+1)x} & p_{(k+1)y} & 1
\end{vmatrix}
= p_{(k-1)x}(p_{ky} - p_{(k+1)y}) - p_{(k-1)y}(p_{kx} - p_{(k+1)x}) + (p_{kx}p_{(k+1)y} - p_{(k+1)x}p_{ky})
\]

For a series of points defining a convex polygon, the sign of all the determinants about every point will be the same. In Java’s graphics coordinate system, where the x-coordinate increases going to the right and the y-coordinate increases going downward, this means that the determinants will all be positive when traversing a convex polygon clockwise, and they will all be negative when traversing the outer edges in the opposite direction. If there is a single point in the set whose determinant is different from the others, then the polygon is concave, and therefore must be split into concave sub-polygons before triangulation. When a polygon is drawn, all of the positive and negative points are counted. Points that lie between collinear edges will have a determinant of 0, so these will have to be set to the same sign as the point that precedes them. If most of the points are positive, then the sequence defines a clockwise traversal, and if the determinants are mostly negative, then the sequence traverses the polygon in a counterclockwise direction. Once it is determined which direction the polygon is drawn in, the points are traversed from the beginning, finding the first point in the sequence that has a determinant opposing the others. When such a point \( p_k \) is found, we continue traversing through the path starting from point \( p_{k+1} \) until a point \( p_n \) is found such that the determinant about the three points \( p_{k-1}, p_k, \) and \( p_n \) is
the same as the majority of points in the polygon. An edge is drawn between point \( p_n \) and \( p_k \), which becomes another constraint in the triangulation. The polygon is split into two sub-polygons at this edge, which themselves are checked for concavity, splitting them again as necessary. This algorithm is guaranteed to eventually terminate, as the smallest possible subset of points that will result from continued splitting is three points defining a triangle, which by definition is always convex. The figures on the following page illustrate the results of this splitting algorithm on two test polygons.
Once the polygon has been split, the CDT for each sub-polygon is calculated using the following algorithm described in [7]. The points describing the convex polygon are ordered first by x-coordinate, then by y-coordinate. The points are then split into left and right subsets of equal sides, with odd points being put into the left subset. This splitting continues until each subset contains either two or three points. Edges are then drawn between the points in each subset, and then each left side subset is merged with its neighboring right subset with the following procedure. The following pictures illustrate this process with the largest sub-polygon in the second example earlier.

The lowest point for each of the L and R subsets are selected as points for the base edge of the merged polygon. This base edge is the first edge of the merged subsets. The next edge in the sequence is selected by computing a list of potential candidates in each of the subsets. A point is a candidate for the next edge if it meets both of the following criteria:
• There exists an edge connecting the point to the current base point in the corresponding set (e.g. left candidates are connected to the left base point and right candidates to the right base point)

• The angle formed by the base edge and the edge with endpoints between the possible candidate and the base point is less than 180 degrees.

If no points satisfy either of the criteria in either subset, then no points are submitted as candidates for the subset, and merging of the subsets is complete.

Illustration 8: Base edge and initial candidates
Illustration 9: Circumcircle of base edge contains next left candidate
Illustration 10: Left edge deleted from triangulation

Candidates are then ordered by the angle that they form with the base edge. If the circumcircle defined by the base edge points and the first candidate point in a subset contains the next candidate, then it can be concluded that the edge between the first candidate and the base point is not Delaunay. This edge is then removed, and the next candidate is submitted as the candidate point for the subset.
One of two edges will be selected as the next base edge in the triangulation – the edge defined by the left candidate and the right base point, or the edge defined by the right candidate and the left base point. If only one subset contains a candidate point, the single point is the obvious choice chosen as the next base point in the triangulation. If both sides submit a candidate point, then the next point is selected with a simple test – if the circumcircle defined by the base edge points and the right candidate contain the left candidate, then the left candidate is chosen as the new left base point. Otherwise, the right candidate becomes the new base point.

If neither side submits candidate points, then the triangulation for these two subsets is complete. Merging continues until all the subsections have been joined to form the original convex polygon.
Once the CDT for the user’s input polygon has been found, the triangles generated are separated into three categories: terminal triangles, which have two external edges, sleeve triangles which have only one external edge, and junction triangles that have no external edges. The chordal axis for the polygon is constructed by traversing through the triangles, starting at the leftmost terminal triangle. The first point in the chordal axis is the point where its two external edges meet, and the second is the midpoint of the edge opposite of this point. Successive midpoints are added as the triangles in the polygon are traversed. Midpoints of junction triangles are joined at the triangle’s centroid, and when new terminal triangles are encountered, their outermost points are added as endpoints to the chordal axis. Once this initial chordal axis is created, the triangulation must be slightly altered as the chordal axis is pruned of unnecessary points.
The arrangement of the points lend themselves quite easily to a tree representation. The tree is iterated, starting at the point associated with the first terminal triangle found in the triangulation. Chordal points, along with their corresponding edges and triangles, are pruned from the tree if the circle centered at the current point with diameter equal to the length of its containing edge contains all the points of the triangles associated with the point and its parents. Merging and pruning terminate when either some point in the terminal polygon lies outside of this circle, or when an edge of a junction triangle is merged. In the former case, the terminal polygon is split into a fan of triangles radiating from the midpoint of the last merged edge. This midpoint then becomes a terminal point of the chordal axis. In the latter case of a junction edge being merged, the edge is removed from the triangulation, with new edges being drawn from these two endpoints to the centroid of the junction triangle, and the final terminal polygon being fanned about this point. The resulting junction quadrilateral is split into two triangles. The triangles intersected by the pruned chordal axis are then split along the axis and triangulated to
obtain the final triangulation of the drawn polygon, which is then exported to the 3D space for extrusion into a 3D object.

The points along the chordal axis are duplicated in 3D space, with one set of points being elevated a distance proportional to their distance from the edge of the polygon, and the other set being lowered by the same distance. Arcs are then drawn over the edges between the chordal axis points and the polygon points, forming a series of circles along the polygon. The reason for pruning the chordal axis and re-triangulation become apparent at this stage, as the edges drawn between the fanned triangles and the terminal point of the chordal axis fold out like an opening umbrella. Adjacent arcs are then stitched together to form the 3D representation of the original polygon.

Illustration 19: Polygonal Mesh Construction

Two parameters control how the 3D mesh is shaped. First, an elevation parameter specifies the proportion that chordal points extend from the input plane. By default, this parameter is 1, meaning that each chordal point is raised and lowered by the average length of all connecting edges. The second is a roundness parameter, which allows the user to specify how many detail points are added in each arc. Because the arcs in the 3D mesh can’t be rendered as
perfect curves, they are instead rendered as a series of arbitrarily small line segments bounded behind the arc. The roundness parameter would be the number of divisions in these arcs. A roundness of 0 would result in no subdivisions in the arcs, forcing edges between boundary and chordal points to be straight lines, and creating a crystalline kind of structure. Higher roundness values would result in more subdivisions, and therefore a rounder, more plush-like figure.

Even with these parameters, the resulting shapes still have an overly rough appearance. Granted, with a high enough roundness parameter, the sleeve rings centered around each choral point will seem to be perfectly round. With an elevation factor of 1, the meshes appear ready for haptic editing. However, for higher values of the size factor, the flaws in this system become readily apparent: "humps" begin to form in the longer extensions of the mesh.
To visualize how this occurs, consider a subset of stroke points in the triangulation that form a long rectangle-like shape. The chordal points lying on the diagonals of each of the smaller rectangles in this section are connected to more longer edges than those that lie on the edges of each rectangle. As a result, the diagonal points are raised higher than normal, a condition that is further magnified with an increasing size factor.

Even without haptic integration, the need for smooth 3D meshes from Teddy cannot be underestimated. An effective solution to this problem is to fit the chordal points along a parametric cubic curve. In this implementation, it has been found that fitting the chordal points to localized B-Spline curves yields a remarkably smooth mesh.

The B-spline method borrows from the drafting practice of attaching weights to metal strips called splines. Provided that the splines weren't excessively stressed, they formed a smooth continuous curve. In this case, the chordal points act as the weights, and are taken as the guiding points of B-Spline curves. The resulting initial curves usually do not contain any of the points used to described them. The chordal points are then moved to the corresponding points on the newly derived curves. The process is repeated again, using the new point locations as input parameters for the next curve. After a few iterations, the chordal points begin to converge on the curve that they describe, thus smoothing out the parts of the mesh that they lie on.

For each figure, the chordal axis must be split into curves separated by the junction points of the 2D triangulation. It is at these points that the orientation of the path along the chordal axis changes to make fitting a curve through the points difficult. The curves are defined by the following matrix equation from [8]:

15
\[ T \times M \times G = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \times \begin{bmatrix} -1 & 1 & -1 & 1 \\ \frac{1}{6} & \frac{2}{2} & \frac{2}{6} \\ \frac{1}{2} & -1 & \frac{1}{2} & 0 \\ -\frac{1}{2} & \frac{2}{2} & 0 & 0 \\ -\frac{1}{6} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{bmatrix} = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} \]

where \( T \) is a 1x4 matrix containing the parameters \([t^3 \ t^2 \ t \ 1]\), with \(0 < t < 1\), \( M \) is a 4x4 basis matrix, and \( G \) is a 4x3 geometry matrix containing the \(x\), \(y\), and \(z\)-coordinates of four consecutive input points.

In this method every point is processed along with the three points ahead of the sequence. This gives the B-Spline the unique advantage that modifying one point only modifies a subsection of the curve described by all the points. B-spline curves are also easily transformable, and can be subjected to any number of scaling, translation, and rotation operations without compromise in the basic geometry. The following figures present some test meshes before and after refinement with the B-Spline method described.
Illustration 21: Igarashi triangulation

Illustration 22: Before B-Spline smoothing
Illustration 23: After B-Spline Smoothing

Illustration 24: "Bean" triangulation

Illustration 25: Before B-Spline smoothing
Illustration 26: After B-Spline smoothing
Given the relative ease of implementation and quality of meshes produced by B-Spline curve fitting, it appears to be the fastest, most reliable method. However, once the initial mesh is complete, such methods cannot be used to refine the mesh after it has been altered. Existing points can no longer be considered chordal or not.

In the original Teddy [9], this problem was confronted by using a modified version of the Skin algorithm to smooth out their models. Skin takes an unrefined 3D mesh produced by Teddy as input. This is considered to be a "skeleton mesh. Skin approximates a quadric surface around the skeleton mesh, and dubs it as the "skin mesh". Each point in the skin mesh is associated with a point on the skeleton mesh, and with each refinement of the algorithm, each skin point is moved to form as many equilateral triangles as possible in the skin mesh, while still adhering to the approximated surface. A "beautified mesh," containing even more detail points interpolated on the skin mesh is presented to the user. "Teddy" uses this algorithm to refine the mesh upon creation and subsequent editing operations to the underlying skeleton mesh.
SECTION III
DESIGN

In this project, the B-Spline method is first executed to provide a starting point for the mesh. After that, each vertex is associated with a localized quadric surface. Assuming that in an optimal mesh arrangement, the vertices are spaced as equally apart as possible while lying on the quadric surface they describe. If the surface were projected in 2D, it would appear to be a perfect triangular grid. In the Skin algorithm [10], the vertices used to calculate local quadric patches would be the six neighboring vertices, along with the six points in the triangles adjacent to the first set. This would be more than enough data to calculate the 10 coefficients of the quadric surface about the original point. The remaining three points are used as error reduction data. The diagrams below illustrate an optimal vertex configuration, and a typical configuration on a straight mesh section produced by the triangulation algorithm.

Drawing 1: Optimal vertex alignment
Drawing 2: Typical vertex alignment
In order to determine which points to use for the quadric surface calculations, the system must have a way to identify which vertices are adjacent to each other. This can be accomplished by embedding a doubly-connected edge graph in the mesh object. One that has proven to be easy to implement and navigate quickly is given in [11]. In this implementation, the structure maintains a list of half-edges. Each half-edge is a one way line segment starting from a single vertex on the mesh and ending on another. Each half-edge keeps a link to these vertices, and a link to their twin edge, which is defined as the edge that originates at the half-edge's endpoint, and ends at its origin. Half-edges also have a link to a face in which it belongs – in the context of a triangular mesh, it would be the triangle to which it belongs. Finally, half-edges have links to the next edges in the face (or triangle) that they are contained in, assuming a counterclockwise traversal about the face.

With this edge list generated for each mesh, the software can find the correct points for a quadric surface about a vertex with the following steps:

1. Find all edges originating from the current vertex. The endpoints of these edges are the first points to be added to the vertex list.
2. Find the next edge of each of these initial edges. Because these edges belong to triangles, these edges connect the first set of points to each other, and do not need to be repeated on the vertex list.
3. Find the twin edges of each of the next edges. These edges will belong to a new set of mesh triangles.
4. Find the next edges of each of the twin edges. The endpoints of these edges are new points that need to be added to the vertex list.

5. Repeat steps 3 and 4 as long as the vertex list contains fewer than 10 vertices, adding any new points that are encountered. This step is necessary, because the initial mesh will not be optimally spaced, and therefore is not guaranteed to be connected to six points.

Because the meshes in Teddy are meant to be edited, exceptions can occur for a mesh that has undergone an extensive operation, such as cutting it in half along a plane. Vertices lying on the edge of the plane would generate irregular quadric surfaces that would improperly round out the cutting plane with repeated iterations. Thus, points lying on these sharp edges should be considered to be part of two separate but intersecting quadric surfaces. This brings up the question of how a sharp edge is defined. It can be logically deduced that on any closed triangular surface mesh, a vertex will be connected to at least three other vertices in the mesh. Such a case may arise if the user were to draw an extremely long and slender extrusion, like a pointed horn. The four points in question would then form a regular tetrahedron on their own. To allow for this case, a sharp edge is defined as an edge between two faces with an angle less than the angle of two faces of a normal tetrahedron, which would be equal to \( \arctan(2\sqrt{2}) \), or about 70.5288°. If the edge in question is connected to the vertex currently being processed, then two quadric surfaces are calculated at that vertex, with neighboring vertices being assigned to their respective surfaces. Vertices lying on the sharp edge are assigned to both quadric functions. After that, processing continues as normal, gathering new points for each surface, but not adding points that cross the sharp boundary.
Haptics are handled in a relatively simple manner. The Novint Falcon is a haptic device capable of only three degrees of motion – movement to the left, right, up, down, towards the device, and away from it. The Falcon isn't able to rotate in 3D space. Given these limitations, it was decided to keep the virtual tool in the application simple – one which the rotational orientation didn't matter in determining collision with the mesh. A sphere would be the obvious choice with the limitations being faced.

The collision detection model follows a two-tiered approach. First, axis-aligned bounding boxes (AABBs) are calculated for both the tool sphere and the mesh. An AABB simply specifies the minimum and maximum x, y, and z-coordinates of the smallest possible cube enclosing all the points of a 3D object. The collision algorithm first checks to see if the AABBs of the tool sphere and mesh intersect each other. In this way, a simple test is performed to check if there is an intersection. If the AABBs don't even intersect, then no further processing is performed. If, however, they do intersect, the contact points of the AABBs serve as a way to determine which triangles in the mesh are closest to the tool sphere. Because only the simple case of a sphere intersection is being considered, all that needs to be generated is a list of mesh triangles that are closer than the radius of the tool. A simplistic mass-damper equation is then used to determine the corresponding force to send as feedback. The closer the tool is to any given triangle in the mesh, the more force it exerts in the direction of its normal vector. These forces are summed together and then normalized to provide the correct force through the haptic controller.
SECTION IV
IMPLEMENTATION

The code developed for this thesis project is meant to provide a simple haptic and 3D graphic interface for viewing triangular meshes. No extensive error checking or code optimization was implemented, except to ensure that haptic processing methods were executed in a timely manner for haptic fidelity.

Java and JOGL

Java was chosen language for development of this thesis project. At the time of the development of the 2D sketchpad interface, the Swing GUI API provided by Java was considered to be the easiest API to work with. Using a Swing based sketchpad window, the software allows the user to draw a single stroke on a blank canvas, using a mouse. This drawing operation closely resembles the way a user would draw a stroke in many 2D graphics applications, like Microsoft Paint, or Adobe Photoshop.

To provide 3D rendering, the Java OpenGL (JOGL) API was used in the project [12]. JOGL provides Java bindings to C-based OpenGL library functions. Because there are concepts in the C programming language that do not translate well into Java, JOGL must provide alternative approaches to traditional OpenGL programming. Thankfully, JOGL presents a minimal learning curve to those already familiar with OpenGL.
The software uses OpenGL to implement a simple viewer for 3D mesh files conforming to the open Wavefront OBJ standard [13]. Any mesh stored in an OBJ text file containing a list of vertices and faces can be rendered. Use of vertex normal data is also supported, but not necessary, as the software will calculate default normals for faces without them. Rendering is done by loading all vertex, normal, and face index data into buffer arrays and using the glDrawArrays() function to render the mesh – an efficient method that minimizes overhead from calling multiple draw functions and improves performance overall [14]. These advantages are complimentary to ensuring that haptic operations can be executed in a timely manner as well. Implementation of other features of the OBJ format were reserved for future work.

Novint HDAL and JNI

The Novint Falcon was chosen as the haptic controller for this project because of it's low price and portability. However, because Novint's SDK was made for development in C/C++, communication with existing Java applications presented a challenge. At first, open-source third party applications were investigated for generating Java bindings to the Novint API. Gluegen, the application used to generate the bindings for JOGL was considered to be an obvious choice [15]. SWIG was also considered for its ability to generate bindings for a vast multitude of programming languages [16]. Upon using these applications, however, neither one was able to produce a suitable Java binding.

At this point it was decided that a binding would be coded from scratch using the Java Native Interface (JNI) [17]. JNI is a an interface that allows programmers to use libraries...
developed in C/C++ in Java and vice versa, and as of Java version 1.5, it is bundled along with the standard Java SDK. Implementing this approach would entail creating a Java class able to load the Novint device driver and call its functions. It would also be necessary to implement a set of C functions that would accept calls from the Java layer, perform conversions on the parameters received, and call the corresponding driver function.

The Novint Hardware Device Access Layer (HDAL) API lists 22 functions [18]. Most of these functions are very straightforward – they return state data of connected haptic devices through primitives (ex. version numbers, button states) and primitive arrays (ex. transformation matrices, haptic workspace boundaries, controller position). The Java side of the JNI binding mimics the function prototypes given in the header file for the HDAL for all of the functions except for one that will be discussed shortly. Converting input parameters and return values from these functions was a trivial matter. The real challenge of the JNI implementation would be in the handling of the haptic servo thread. When the Novint Falcon is initialized, a new thread is started in parallel with the running application called the servo thread. The purpose of this thread is simple – it calls a user-defined callback function at a rate of about 1 Khz. In a C/C++ application, the user would set this callback function using the hdlCreateServoOp() function, whose prototype is given below:

```
HDLAPI HDLOpHandle HDLAPIENTRY hdlCreateServoOp (HDLServoOp pServoOp, void * pParam, bool bBlocking)
```

This function returns an integer associated with the user-defined callback function, allowing the user to access it in the servo loop when the callback is no longer needed with the hdlDestroyServoOp() function. The first parameter pServoOp is a void pointer to a user-defined
function, which takes another void pointer to input data as a parameter, and returns one of two integer status codes: HDL_SERVOOP_EXIT (0) or HDL_SERVOOP_CONTINUE (1). A function returning 0 notifies the servo thread that the function is not to be called again on the next servo tick. This would be useful for handling events that only happen sporadically, such as button presses. A function returning 1 signals the servo thread to repeat the call in the next servo tick, and is useful for passing data that is accessed frequently between the graphic and haptic model. The second parameter pParam is a void pointer to some data that the callback function will use as input. It is up to the implementor to cast this pointer to the appropriate data type in the callback function. The third parameter bBlocking is a boolean flag indicating whether the application should wait until the servo thread returns from the callback function. Blocking is necessary for functions that need to synchronize data between the haptic and graphic models of the application, but the programmer needs to ensure that such a function returns quickly, or else risk crashing the application from thread deadlock.

The problem here is that Java has no support for explicit function pointers. To get around this issue, an abstract ServoOp class was defined on the Java side of the binding, which is listed below.

```java
public abstract class ServoOp
{
    public int handle;
    public Object params;

    public abstract int doIt();
}
```

This simple class provides everything needed for the servo thread to run – an integer field to place the returned handle, an Object field to store input data, and a method to act as a callback.
The programmer can create new separate classes that extend ServoOp and implement the doIt() operation as appropriate (or in the case of this project, define an anonymous inner class that does all of this). Because the ServoOp defines a field to store parameter data, there is no need to specify such a parameter in the Java prototype – instead, the Java method only requires a reference to an instance of ServoOp and a boolean flag to indicate whether or not the callback method blocks for the servo thread.

The issue of providing a void function pointer to the HDAL on the C side of the binding has not yet been addressed. Java lacks support for pointers to functions, and so it isn't as simple as getting the pointer to the doIt() function and passing it to the HDAL. To address this need, the C side implementation includes a callback function. Every time the hdlCreateServoOp() method is called on the Java side, the HDAL is provided with a pointer to this function on the C side, and a global reference to the ServoOp instance is established, protecting it from erasure by Java's garbage collector. The HDAL is then given this reference as its second parameter. Finally, the boolean flag from the Java method call is converted to its C equivalent and passed as the third parameter. When the HDAL function finishes, the ServoOp global reference is stored in a map internal to the C side, using the returned handle as the key. The C side then converts and returns the handle to the calling Java method.

Now that the issue of sending the servo thread callback data has been addressed, it's time to illustrate what happens when the servo thread calls the supplied method. As stated before, the servo thread calls the static C side callback function, with the global reference as a parameter. Because the servo thread is executing outside of the scope of the Java application and the Java
Virtual Machine in which it's running, the servo thread will have to attach itself to the JVM thread in order for the global reference to have any useful meaning. When the Java application starts and loads the JNI driver, the C side saves the reference to the calling JVM in a static pointer. With this reference, the standard JNI function AttachCurrentThread() can be called to attach the servo thread. Once attached, the callback function uses the object reflection capabilities provided by the JNI to get the class type of the callback's global reference, and call the doIt() method from there. The ServoOp method then executes on the Java side, doing all of the haptic processing there before returning. If doIt() returns HDL_SERVOOP_CONTINUE (1), the flow of execution returns to the servo thread, where the call is repeated. If the method returns HDL_SERVOOP_EXIT (0), then the global reference is removed from the servo op map and deallocated, so that the memory it occupies can be recovered through Java garbage collection.

**Open Dynamics Engine**

Collision detection is implemented using an open source library called the Open Dynamics Engine (ODE) [19]. ODE is a framework which allows the programmer to create models for rigid-body dynamics, including robust collision detection. ODE has been used extensively in many commercially available games, game engines, 3D development suites, and virtual reality simulations. This project makes use of the Java port of ODE called ODE4J [20]. Because ODE4J is a pure Java port, there was no need to develop a Java binding to ODE libraries. Much like its C counterpart, ODE4J organizes its methods in static classes, instantiating
ODE classes as necessary through static factory methods, thus making the learning curve minimal for those experienced in ODE programming.

Setting up a dynamic model in ODE4J has many parallels to setting up the graphical model. In ODE4J, this starts with creating a space in which dynamics can be modeled. Rigid body instances are then created for each object in the graphical representation. In the case of this project, the bodies of interest are the tool sphere and the rendered triangle mesh. Once the bodies are instantiated, their shapes must be defined by associating them with instances of ODE Geom objects. These are instantiated in a manner similar to how geometric objects in OpenGL are rendered. For this project, the tool sphere Geom is instantiated with a function as simple to use as the glutSolidSphere() function in OpenGL. In ODE4J, this is accomplished through the OdeHelper.createSphere() method, which takes as input a reference to a dynamics space, and a radius for the sphere. The triangle mesh Geom is a bit more involved, but strikingly similar to setting up buffer objects in OpenGL for use with the glDrawArrays() function – arrays containing the coordinates of each vertex and their respective indices are stored in a TriMeshData object, which is then passed to the OdeHelper.createTriMesh() method to create its dynamic model.

With the rigid bodies defined along with their respective geometries, collision detection can now be implemented. To do this, ODE4J provides an interface called DNearCallback. The only thing specified in this interface is a void call() method, which takes two references to potentially colliding bodies, and a reference to an object that contains any supplemental data needed. Much like the ServoOp abstract class in the binding to the HDAL, programmers can
implement the DNearCallback interface to handle collisions between rigid bodies. The project defines a class called Collider to implement the interface.

At every tick of the servo thread, the ODE model is updated with the current position of the cursor and orientation of the mesh. Once complete, the OdeHelper.spaceCollide() method is called on the Collider. If the AABBs of the sphere and mesh intersect, the call() method is invoked with the sphere and mesh Geoms as parameters. In the call method, the Geoms are passed to the static OdeHelper.collide() method, which performs more precise collision calculations on the Geoms. If the tool sphere and mesh truly intersect each other, then a list of contact points is returned by the collide method. Each contact point object contains the following:

- a reference to each of the colliding Geoms
- indices of intersecting faces from each Geom (for the mesh only – the sphere doesn't have any sides, so it always returns -1 as its index)
- a location in xyz-coordinates where the collision was found
- a normal vector in the direction of contact along the surface
- a depth attribute indicating how far the Geoms intersect each other.

In initial implementations of the Collider object, the normal vectors reported from each contact point were added together to find the total force acting on the haptic sphere. While this doesn't present any problems when only one mesh triangle is intersected, this would cause some haptic “stickiness” when the sphere intersected multiple faces, especially at the edges where triangles met. In these cases, the normal vector returned for each contact point would not
necessarily be equal to the true normal vector of the face. Because of this, a user sliding the
haptic cursor across a flat surface would feel bumps as edges of the mesh were crossed.
Compounding more to the problem was the inability to determine parallel adjacent faces, adding
some forces multiple times in a sum where only one is necessary. To get around this, the face
index was used to access the corresponding mesh face in the graphical model. The normal is then
calculated by the three vertex indices referenced by the side. Duplicate normal vectors are
filtered out, with the Collider only accepting the face normal with the deepest face of
intersection. The Collider object adds up the normal vectors of all contact mesh faces, scaling
each one by their associated depth, and multiplies a stiffness parameter based on the size of the
cursor. The final sum vector is then returned as a force to send back to the servo thread.
SECTION V
FUTURE WORK

The current project can serve as a good basis for future projects. The main focus of the project was to develop a software application that could not only graphically render 3D objects, but present the user with a haptic rendering of them as well. Editing meshes in this way would at the very least require a reliable method of altering mesh vertex data in both the buffers for the graphic and haptic models to allow the user to push into the figure with the haptic tool, similar to pushing into clay with fingers. A haptic paintbrush could also be implemented, not only to paint color onto figures, but also apply haptic textures to them as well. An example of haptic painting is the DAB system described in [21]. Because the Novint Falcon only provides three degrees of haptic motion, some modifications will have to be added for more complex operations.

According to [1], the Novint HDAL can support up to four Novint Falcons on the same machine. A possible application of this could be a simulation of a cutting wire tool. The potential variety of haptic tools expands with the possibility of adding support for six degree of freedom haptic devices. Carving tools such as knives and awls, as well as the varying strokes of brushes in DAB could be utilized with the ability to rotate haptic tools. Implementing such tools, however, will require more sophisticated haptic calculations than the simple spring damper models used in this project. A deformable buffer model that could help in achieving this goal is presented in [22]. This model takes elastic forces due to surface pressure of a deformable object into account,
which would lend itself well to a clay modeling simulation. Other potentially useful haptic models include the hybrid surface representation of 3D figures presented in [23], and the Lead Lag and Linear Quadratic Gaussian haptic control methods in [24], which include friction forces from sliding along a model's surface.

Another potential addition to the project would be to allow the user to place the refined figures in a virtual setting. One way of accomplishing this could be using images captured from multiple camera inputs to create a virtual environment. Efficient methods of stereo image processing have been a topic of interest in robot vision for years, and could prove useful in providing such a capability. One recent algorithm given in [28] accurately derives a depth map from two images using image gradient guided cost calculation scheme. Another approach to obtaining 3D information efficiently given in [29] divides input images into "patchlets" - surface element data structures with size, orientation, and probabilistic confidence parameters. By analyzing observed surfaces as groups of pixels rather than each pixel in multiple reference images, a 3D environment can be more efficiently represented.

There is also room for improvements to the sketchpad interface. Rather than use camera image input to create virtual scenery, the user could sketch settings in a manner similar to VR Sketchpad [30]. It could also be extended to give the user a way to give the 3D figures a skeleton to serve as a basis for animation for the figure. An example of this is demonstrated in Motion Doodles [31].
These are only a few of the potential areas of continued research for content creation. Truly, the possibilities are far too many to list here, and are only limited to the vast number of traditional artistic techniques through history.
SECTION VI
CONCLUSION

This thesis project proposed the idea of a three-dimensional content creation system with an easy to use interface that resembled more traditional and intuitive artistic methods. In this system, a user with little to no experience in modeling would be able to create 3D models with a simple sketch of a figure. The user would then be able to make refinements to the initial figure by sculpting it using a haptic controller, much as an artist would sculpt a figure out of clay. To make the interface more accessible to novice 3D modelers, the system proposed makes use of the Novint Falcon, an inexpensive and easy to use haptic controller.

The challenges encountered in the design of such a system were also presented, along with solutions inspired by those who have previously researched graphical and haptic modeling, and original solutions devised by the author. These challenges included extrapolating a 3D model from a 2D sketch, refining it into a realistic basis for sculpting, and implementing a solid collision detection algorithm to provide lifelike force feedback. The current system can only allow a user to feel the surface of any 3D mesh, but it serves as a good basis for the proposed modeling system, and shows potential for projects beyond the author's scope, including applications for the visually impaired and as a learning tool for children. It is hoped that future work will be as personally rewarding as this project has been.
BIBLIOGRAPHY

[18] Haptic Device Abstraction Layer (HDAL) API Reference VERSION 2.1.3 August 14, 2008 Novint Technologies Incorporated Albuquerque, NM USA


[23] “A haptic-rendering technique based on hybrid surface representation” L. Kim, G. Sukhatme, M. Desbrun in something by IEEE, 2004, p.66-75*


