Virtual Sculpting with Haptic Displacement Maps

by

Robert Jagnow
B.S., Computer Science (1998)
B.A., Mathematics (1998)
Texas Tech University

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

January 23, 2001

Signature of Author..
Department of Electrical Engineering and Computer Science

Certified by..
Julie Dorsey
Associate Professor of Computer Science and Engineering with Tenure
Thesis Supervisor

Accepted by...
Ain A. Sonin
Chairman, Department Committee on Graduate Students

Virtual Sculpting with Haptic Displacement Maps

by

Robert Jagnow

Submitted to the Department of Electrical Engineering and Computer Science
On January 23, 2001, in partial fulfillment of the
Requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis presents an efficient data structure that accommodates high-speed haptic (force feedback) interaction with detailed digital models. Complex models are partitioned into coarse *slabs*, which collectively define a piecewise continuous vector field over a thick volumetric region surrounding the surface of the model. Inside of each slab, the surface is represented as a displacement map, which uses the vector field to define a relationship between points in space and corresponding points on the model's surface. This representation provides a foundation for efficient haptic interaction without compromising the visual complexity of the scene. Furthermore, the data structure provides a basis for interactive local editing of a model's color and geometry using the haptic interface. I explain implementation details and demonstrate the use of the data structure with a variety of digital models.

Thesis Supervisor: Julie Dorsey
Title: Associate Professor of Computer Science and Engineering with Tenure

Acknowledgements

I'd like to thank Julie Dorsey and Leonard McMillan for having enough faith in my abilities to invite me into their lab and encourage me to pursue my interests. I'd like to thank my friends and officemates for supporting my endeavors and answering my incessant questions. I'd like to thank the MIT Outing Club, the runners@wind, and the Sloan Crew team for helping me to maintain my sanity. But mostly, I'd like to thank Mom, Dad, Beth, and Dan for always being there, and for putting up with me for all these years.

Contents

List of Figures

Chapter 1

Introduction

As digital models have become increasingly complex, the graphics community has struggled to develop efficient hardware and software capable of visually rendering geometrically demanding scenes at reasonable frame rates. But some of the more recent challenges to the research community are presented not by graphics hardware, but by haptic hardware – force-feedback devices capable of facilitating tactile interaction with digital models. The human visual system is satiated by animation sequences presented at 30 frames per second, but haptic systems have a more demanding refresh rate with less forgiving artifacts. The PHANToM® haptic device by SensAble Technologies, Inc.® requires updates at 1000 Hz – a constraint imposed by the inherent sensitivity of human tactile sensation. If this constraint is not met, unacceptable tactile artifacts, and possibly even hardware instability may result. In order to haptically render scenes with comparable complexity to the scenes that can be graphically rendered with current graphics hardware and software, new data structures are required for accelerated rendering. Ideally, we would like to haptically render intricate scenes without compromising their visual complexity.

The surge in model complexity has been driven largely by hardware vendors like CyberWare and Motorola, which have made 3D geometry scanners accessible in laboratory environments, giving graphics professionals unprecedented access to libraries of digital models with large polygon counts. Unfortunately, these models often do not exhibit a coherent topological structure – a condition commonly referred to as "polygon soup." Often, a user would like to make minor local edits to the model in a digital environment, but the lack of coherent topology makes this task very difficult. Editing can be tedious, often requiring substantial knowledge of complex modeling packages. Haptic interfaces offer a great deal of promise for intuitive modeling, but haptic systems are limited by the complexity of the data they can manage.

The data structure introduced in this thesis addresses the needs to haptically render complex scenes and to make local modifications to a model's surface characteristics, including, but not limited to its geometry and color. By decomposing a complex model into a collection of coherent regions of local geometry, I have been able to develop methods for implementing high-speed haptic interaction, including intuitive local editing.

1.1 Related Work

This thesis builds on a foundation of related research for improving the efficiency of both haptic and graphic rendering. Clearly, the simplest method for decreasing the computational burden on a haptic system is to decrease the complexity of the digital model. In some instances, this has been shown to be effective with minimal impact on the apparent complexity of the scene. [Morgenbesser95] demonstrated that in some situations, a coarse polygonal mesh can be used to effectively represent the tactile feedback of a more complex geometric surface. Morgenbesser uses a *force shading* algorithm to provide the tactile illusion of a smoothly curved surface in much the same way that Phong shading can provide a visual illusion of smoothness [Phong75]. Normal vectors are precomputed at each vertex in a coarse mesh so that a local normal can be calculated as a weighted average of the normals at the nearest three vertices. Morgenbesser limited his research to very simple polygonal models, and did not present a method for haptic rendering of models with arbitrary complexity.

Other researchers have turned away from the polygonal mesh data representation in search of more efficient data structures for haptic interaction. McNeely et. al. implemented a voxel-based system to accelerate haptic collision detection in complex environments, but their geometric models were not intended for interactive modification [McNeely99]. SensAble Technologies, Inc. commercially produces a volume-based modeling system that simulates a clay-like medium supporting non-physically-based methods of interaction [SensAble01]. Other alternative data representations include voxel-based systems with isosurfaces extraction [Ferley99], B-spline surfaces [Dachille99], and subdivision surfaces [Gregroy00], all of which are designed for interactive geometric editing of models with low to moderate visual complexity.

Here, I explore the use of displacement maps [Cook84] for representing visually complex surfaces in a manner that is amenable to high-speed haptic interaction with limited geometric modification. The most closely related data structure was introduced by [Dorsey99]. Dorsey et. al. decompose a complex model into a *volumetric surface* of extruded quadrilateral slabs that form a thick skin of varying depth around the original polygonal model. The work presented here preserves the notion of enclosing a model in a thick volumetric skin. I expand this representation for use with a haptic interface and present additional methods for interacting with the data.

Displacement maps offer a compact data structure for representing complex surfaces that can be expressed as a discrete function of two variables. They are typically stored as an array of scalar values that displace the surface of a polygon in the direction of that polygon's normal. The storage method is efficient because it requires only the coordinates of the original polygon, plus an array of uniformly spaced scalar displacement values.

Unfortunately, the class of displacement maps that offset a surface in the direction of the surface normal can only be used in limited situations. A single, large displacement map cannot represent undercut topology, and a collection of small, adjacent maps can exhibit problems with undefined or ambiguously defined surfaces, as shown in Figure 1-1. Spherical and cylindrical displacements help to address some of these concerns, but the problem of representing a large model as a collection of displacement maps is nontrivial.

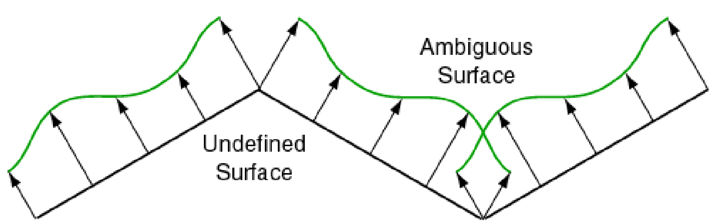

Figure 1-1 – Problems with adjacent planar-projective displacement maps

1.2 Goals and Contributions

In this thesis, I introduce a data structure that accommodates high-speed haptic interaction, even on visually complex models. This data structure is an extension of the volumetric surface, introduced by Dorsey et. al. [Dorsey99]. The representation is generated by simplifying the model into a collection of extruded triangular *slabs* – small volumetric regions where the local geometry is expressed as an array of scalar displacements. Each slab represents a planar-projective region on the surface of the original model – that is, a region in which the geometry can be projected without overlap onto an oriented plane. The fine details of the original model can then be represented by a scalar height field that is embedded between the inner and outer extents of the slab. Adjacent slabs can be seamlessly stitched together to provide both visual and tactile continuity. The resulting data structure supports high-speed haptic interaction, including the ability to locally edit the model's geometry and color.

The advantage of the representation lies in its simplicity and flexibility. The natural hierarchical division between coarse and fine features allows for rapid computation of local surface features, making the data structure ideal for rapid intersection detection for a haptic interface. Furthermore, since local features are represented by a simple scalar field, limited editing of the local geometry can be done extremely rapidly by modifying the values in the displacement map.

But the data structure need not be limited to a single scalar field. Additional scalar fields may be used to represent surface properties such as color, friction, hardness, or specularity. Other fields may represent the depth of various materials underneath or above the visible surface – materials that may be exposed or added by an edit operation. Any number of scalar arrays may be used, depending on the desired complexity of the model.

In the remainder of this thesis, I provide an overview of how slabs can be distributed to enclose the surface of a high-resolution model (Chapter 2). I then introduce a method for high-speed haptic collision detection (Chapter 3) and discuss methods for modifying the local geometry and color of the model (Chapters 4 and 5). Finally, I address graphic rendering concerns (Chapter 6) and discuss my results (Chapter 7), the limitations of the algorithm, and directions for future research (Chapter 8).

Chapter 2

Haptic Displacement Maps

The slab data structure provides an alternative to the common practice of representing complex geometry as an unorganized collection of polygons. A coarse collection of slabs can be a arranged to completely and unambiguously enclose features of a detailed mesh while maintaining the full detail of the model by representing local features as displacements from the surface of a coarse submesh. This hybrid data structure offers the detail of a surface representation with the flexibility and physical intuition of a volumetric representation. Many of the artifacts of voxel representations are avoided by orienting the slabs to coincide with the orientation of the local surface.

Figure 2-1 demonstrates an example object that appears to be highly tessellated with fine geometric details; however, the underlying representation is a simple icosahedron, with the detailed features stored as displacement maps at each surface.

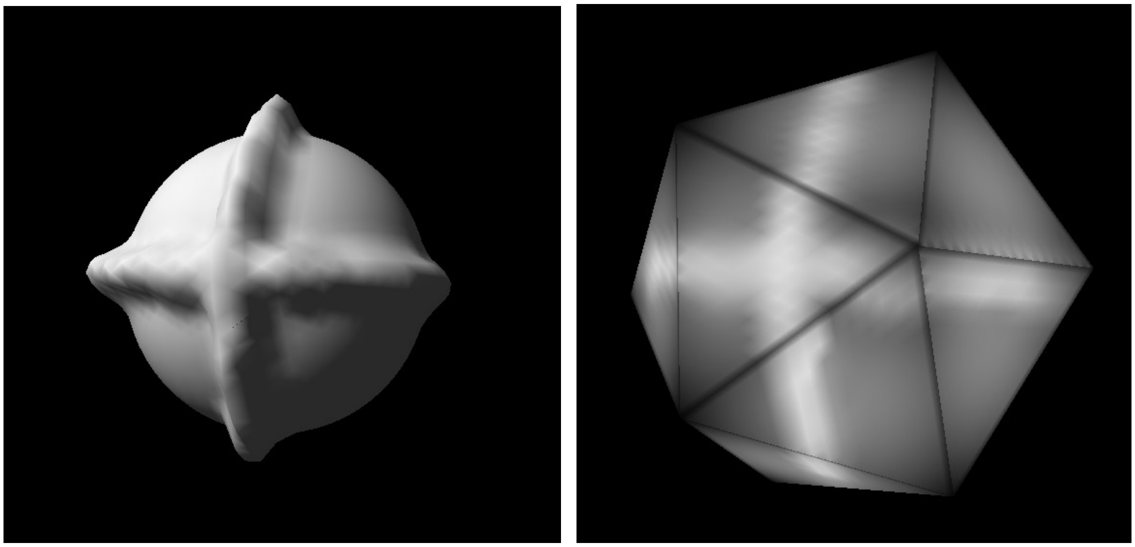

Figure 2-1 – A geometrically detailed object (a) and its equivalent representation with haptic displacement maps (b) Dark surfaces in the displacement maps represent the greatest displacements from the interior surfaces of the submesh.

2.1 Generating the Data Structure

In order to represent a complex digital model as a complete, unambiguous collection of displacement maps, we begin by partitioning the model into slabs, which collectively form a thick skin around the surface of the geometry we wish to represent. To generate the slabs, we start with a base mesh that coarsely approximates our desired model. Such a mesh can be modeled by hand, or by using an automated simplification technique such as Hoppe's *progressive meshes* [Hoppe96]. Each of the large polygons in this base mesh can then be extruded toward the interior and exterior of the model to form a slab. The polygons should ideally be approximately uniform in size, with each polygon representing an approximately planar-projective region of local geometry. To insure proper submesh alignment, the base mesh should contain no T-vertices – that is, vertices that lie on an edge, rather than a corner, of an adjacent polygon.

Given this coarse base mesh, we can partition the model into regions of space over which individual slabs are used to define the local surface geometry. Adjacent slabs are divided by carefully defined ruled surfaces that separate the polygons of the base mesh. For each vertex in the mesh, we can define a projection vector as an average of the normals of all adjacent triangles. A slab can be formed by extruding each base mesh polygon toward the interior and exterior of the mesh along the projection vectors defined at the vertices. This process is illustrated in Figure 2-2. The new slab is the region enclosed by extrusion.

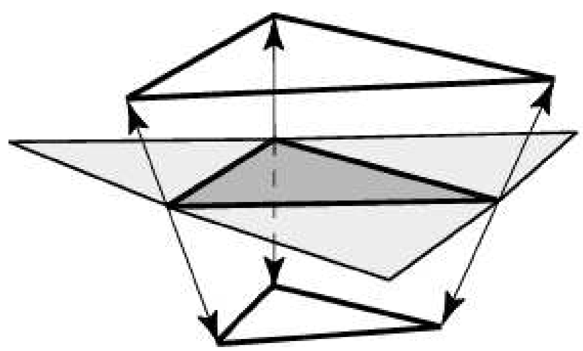

Figure 2-2: Constructing a slab by extruding triangles in the base mesh

The interior and exterior faces of a slab are defined by three points in space, and are thus planar – but since the points on the side of the slab may not be coplanar, I use bilinear patches, a type of ruled surface, to define these boundaries. The bilinear patches can be defined by sweeping a line segment from the interior to the exterior slab edges, as the endpoints are linearly interpolated between the extrema of the projection vectors. Equivalently, a line segment can be swept from one projection vector to the other as the endpoints are linearly interpolated along the interior and exterior edges of the slab. An example of a ruled surface between adjacent slabs is shown in Figure 2-3.

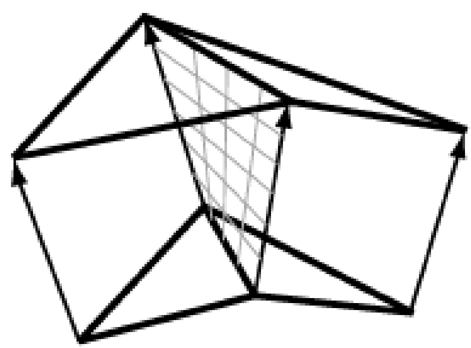

Figure 2-3: A ruled surface between adjacent slabs.

The full complexity of the detailed geometry is expressed as a displacement field that is projected from the interior slab face toward the exterior slab face, as shown in Figure 2-4. The direction of surface projection at each point in the slab is defined as a linear interpolant of the three projection vectors at the corners of the slab. Note that the slabs enclose the detailed geometry completely and without redundancy.

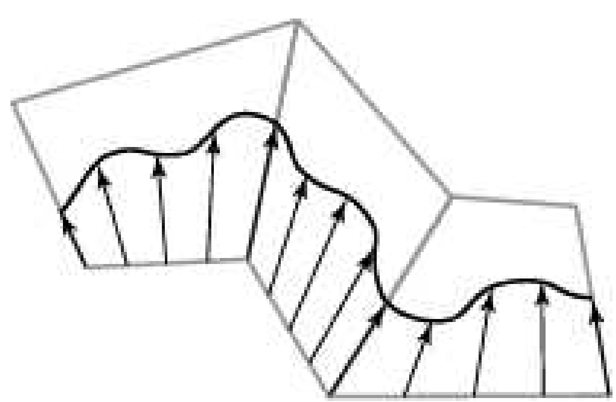

Figure 2-4 – Slabs are separated by bilinear patches that divide the polygons of the base mesh. The detailed geometry is completely enclosed by the slabs, and represented as displacements from the interior slab surfaces.

The displacement map for each slab is stored in a triangular grid of uniform dimensions. Since T-vertices are disallowed in the base mesh, the grid vertices on the boundary of a slab are guaranteed to correspond exactly with grid vertices on the adjacent slab. This constraint insures proper connectivity between the surfaces of adjacent slabs when the detailed model is haptically or graphically rendered. Each scalar value in the displacement map expresses the relative position of a vertex within the slab, with a value of zero representing a vertex on the interior slab face, and a value of one representing a vertex on the exterior slab face.

2.2 Intra-Slab Management Routines

Before I begin discussion of how the data structure is used for haptic interaction, it is important to introduce some of the basic elements of the data structure that contribute to its efficiency. The remainder of this chapter outlines a variety of methods that are used by the collision detection and data modification routines, which will be discussed in chapter three, four, and five. Here, I describe methods for efficient data storage, computation of interpolation weights, and detection of slab boundaries. These simple methods form a foundation for haptic interaction with the data structure.

2.2.1 Efficient Data Storage

Each slab has one or more arrays used to represent geometry and color information. Since slabs are triangular, it is convenient to map the coordinates to a

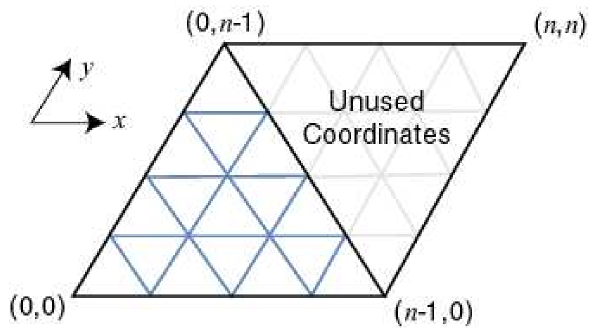

Figure 2-5 – Displacement map coordinate system. The coordinates of the upper-right triangle go unused.

skewed two-dimensional grid, as shown in Figure 2-5. However, a two-dimensional array proves inefficient for storage, as approximately half of the coordinates go unused. For a triangle with n coordinates along each edge, only $(n^2+n)/2$ of the n^2 total coordinates are required. To reduce this wasted space, we can use an additional level of indirection to map coordinates from the coordinate system shown in Figure 2-4 to a more compact one-dimensional array. All of the displacement maps have the same dimensions, so we can precomputed a single n by n array C to perform this coordinate-space conversion.

$$C[x][y] = (y * n) + x - \sum_{i=1}^{y} i$$

The intuition here is that we use a standard conversion from a 2D to a 1D array $(y*n)+x$, and then compact the unused regions by offsetting each row by the number of coordinates in the unused triangular region. Once generated, the coordinate-conversion array C can be used to convert every memory reference from the two-dimensional coordinate system to the tightly packed one-dimensional array. The one exception to this data-packing rule is the color data, which must be stored in a rectangular array to be rendered most efficiently by OpenGL. OpenGL rendering issues will be discussed in more detail in Chapter 6.

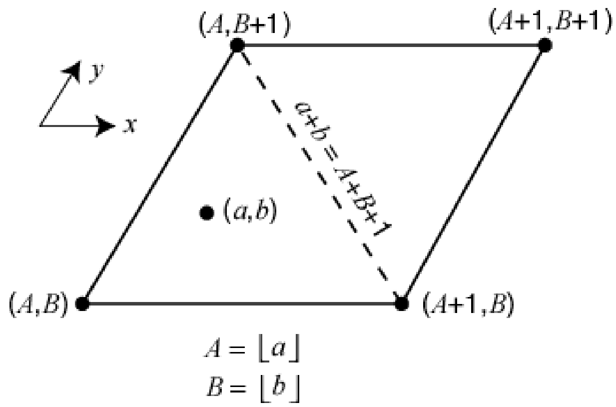

Figure 2-6 – Relationships used for calculating interpolation weights for the three nearest displacement map coordinates.

2.2.2 Interpolation Weights

The displacement data is stored as a discrete sampling of values, evenly distributed across the surface of each slab. However, the haptic system must be able to reconstruct a continuous surface representation based on this discrete sampling grid. Thus, the system relies on a procedure for linearly interpolating between the three vertices nearest to an arbitrary floating-point coordinate (a, b). A point on the surface of the mesh with displacement map coordinates (a,b) is bounded by the quadrilateral region with the coordinates (A,B), (A,B+1), (A+1,B+1), (A+1,B), where $A=\lfloor a \rfloor$ and $B=\lfloor b \rfloor$. If $a+b \leq A+B+1$, then the coordinate is in the lower triangular region. Otherwise, the coordinate is in the upper triangular region. These relationships are shown in Figure 2-6.

In each of the two triangular regions, linear, normalized interpolating weights for the three nearest vertices to the coordinate (a, b) can be computed as follows:

Lower-triangular region	
Weighted coordinate	**Weight**
(A,B)	$1-(a-A+b-B)$
$(A+1,B)$	$a-A$
$(A,B+1)$	$b-B$

Upper-triangular region	
Weighted coordinate	**Weight**
$(A+1,B)$	$1-b+B$
$(A,B+1)$	$1-a+A$
$(A+1,B+1)$	$a-A+b-B+1$

In each case, the normalized interpolation weights sum to one. Sections 3.1 and 3.3 discuss methods that use the interpolation weights to reconstruct the surface and to compute a local surface normal as a weighted average of the normals at the adjacent vertices.

2.3.3 Detecting Slab Boundaries

The routines for modifying the shape and color of a surface rely heavily on traversing the vertices within a slab while modifying the local data as necessary. Given a coordinate pair within a displacement map, these routines must use a method for distinguishing which of the adjacent coordinates are in the displacement map, and which are part of an adjacent slab.

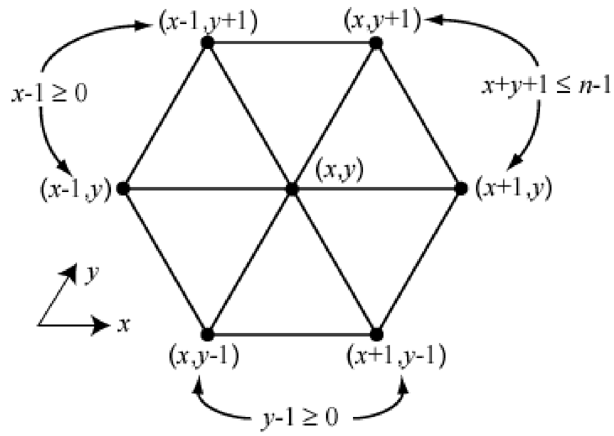

Figure 2-7 – Rules for safely traversing the displacement map coordinates adjacent to a given coordinate (*x*, *y*).

The edges of the displacement map coordinate system are bounded by the lines x=0, y=0, and $x+y$ = n-1, where n is the number of coordinates along any edge of the triangular grid. Under these constraints, we can safely traverse the neighboring coordinates of a vertex (x, y) upon satisfying the conditions shown in Figure 2-7. If the given condition is not satisfied, then the coordinate (x, y) lies on a slab boundary, and we must use a different technique for traversing to the neighboring slab.

2.4 Managing Adjacent Slabs

Each vertex at a slab boundary has a coincident vertex in the adjacent slab. As a surface traversal algorithm reaches the edge of a slab, it must quickly determine the corresponding coordinate on the adjacent slab, and continue the traversal there. This process of managing inter-slab adjacency is accelerated by precomputing the relationships between adjacent slabs and storing the coordinate adjacency relationships in a table.

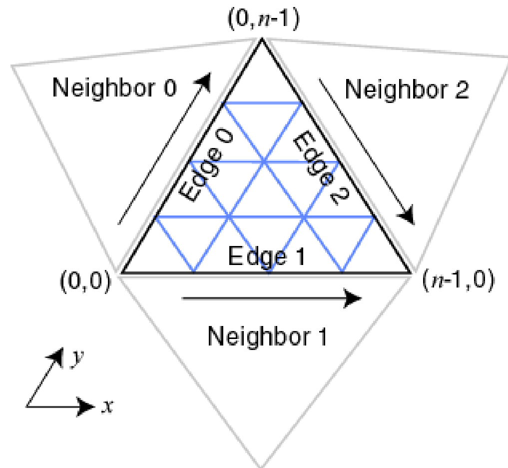

Figure 2-8 – Adjacency relationships used for precomputing slab adjacency information.

Assume that all slabs have a consistent orientation when viewed from the exterior of the model. This is a common requirement for graphic rendering, as a consistent orientation is used to detect and cull back-facing polygons. Under this constraint, there are only nine ways for any two slab edges to abut; that is, each of the three edges of a slab may be adjacent to any of the three edges of its neighbor. The specific adjacency relationship for each edge is stored with each slab. The adjacency relationship is used to index into a small set of global, precomputed tables that store the correspondences between adjacent displacement map coordinates for each of the nine possible relationships between abutting edges. The combined tables contain $9n$ entries, where n is the number of coordinates along any edge of a displacement map. The convention used here denotes the line x=0 as Edge 0, y=0 as Edge 1, and $x+y = n-1$ as Edge 2. Here is an excerpt from the resulting tables, based on the adjacency relationships shown in Figure 2-8:

Edge of this slab	Abutting edge of adjacent slab	Coordinate Correspondence				
		(0, 0)	**(0, 1)**	**…**	**(0, *n*-2)**	**(0, *n*-1)**
Edge 0	Edge 0	(0, *n*-1)	(0, *n*-2)	…	(0, 1)	(0, 0)
	Edge 1	(0, 0)	(1, 0)	…	(*n*-2, 0)	(*n*-1, 0)
	Edge2	(*n*-1, 0)	(*n*-2, 1)	…	(1, *n*-2)	(0, *n*-1)

Edge of this slab	Abutting edge of adjacent slab	Coordinate Correspondence				
		(0, 0)	**(1, 0)**	**…**	**(*n*-2, 0)**	**(*n*-1, 0)**
Edge 1	Edge 0	(0, 0)	(0, 1)	…	(0, *n*-2)	(0, *n*-1)
	Edge 1	(*n*-1, 0)	(*n*-2, 0)	…	(1, 0)	(0, 0)
	Edge2	(0, *n*-1)	(1, *n*-2)	…	(*n*-2, 1)	(*n*-1, 0)

Edge of this slab	Abutting edge of adjacent slab	Coordinate Correspondence				
		(0, *n*-1)	**(1, *n*-2)**	**…**	**(*n*-2, 1)**	**(*n*-1, 0)**
Edge 2	Edge 0	(0, *n*-1)	(0, *n*-2)	…	(0, 1)	(0, 0)
	Edge 1	(0, 0)	(1, 0)	…	(*n*-2, 0)	(*n*-1, 0)
	Edge 2	(*n*-1, 0)	(*n*-2, 1)	…	(1, *n*-2)	(0, *n*-1)

These adjacency tables are used for all surface-traversal routines, and for the precomputation step that calculates per-vertex normals as an average of the normals of the adjacent polygons.

2.5 Per-Vertex Normal Computation

The method used by the haptic collision detection algorithm to smoothly reconstruct a displaced surface relies on having local normal vectors at each point in the mesh. These normals are obtained by summing the normals of all of the submesh polygons adjacent to a given vertex and normalizing the result.

In the center of a slab, this process is relatively straightforward: the six faces that share the vertex are all enclosed by a common slab, so the normals for each face can be summed, and the result normalized. For vertices lying along an edge of a slab, three of the normals must be obtained from each of the adjacent slabs. For vertices at the corner of a slab, the process is more complex. Each slab stores a pointer to each of its three neighbors along its edges, but slabs do not store pointers to all of the neighbors that share a given corner. In fact, it is not even clear how many slabs may be adjacent to a given corner. Thus, the normal vectors at corner vertices are obtained using a recursive algorithm that walks through all of the slabs that share a corner, marking each slab as it is visited, and returning the sum of the normals from all visited triangles. The algorithm for summing the normals of all of the polygons adjacent to a given vertex (*x, y*) is a member function of each slab object. The procedure is expressed in the following pseudocode.

```
accumulateNormals(x, y) {
    total = (0, 0, 0)
    if (isMarked()) return total
    markThisSlab()

    if (x > 0 and x+y < meshSize-1)
        total = total + getNormal(x-1, y, UPPER)
    if (y > 0 and x+y < meshSize-1)
        total = total + getNormal(x, y-1, UPPER)
    if (x > 1 and y > 0)
        total = total + getNormal(x-1, y-1, UPPER)
    if (x ≠ 0)
        total = total + getNormal(x-1, y, LOWER)
    else {   //(x, y) is on the edge adjacent to Neighbor 0
        adjacentSlab = getNeighbor(EDGE0)
        seed = getAdjacentVertex(EDGE0, x, y)
        total += adjacentSlab->accumulateNormals(seed.x, seed.y)
    }
    if (y ≠ 0)
        total = total + getNormal(x, y-1, LOWER)
    else {   //(x, y) is on the edge adjacent to Neighbor 1
        adjacentSlab = getNeighbor(EDGE1)
        seed = getAdjacentVertex(EDGE1, getAdjacentEdge(EDGE1), x, 0)
        total += adjacentSlab->accumulateNormals(seed.x, seed.y)
    }
    if (x+y ≠ meshSize-1)
        total = total + getNormal(x, y, LOWER)
    else {   //(x, y) is on the edge adjacent to Neighbor 2
        adjacentSlab = getNeighbor(EDGE2)
        seed = getAdjacentVertex(EDGE2, getAdjacentEdge(EDGE2), x, y)
        total += adjacentSlab->accumulateNormals(seed.x, seed.y)
    }
}
```

The getNeighbor(*edge*) routine returns the slab adjacent to the given edge, based on the relationships shown in Figure 2-8. The getAdjacentVertex(*edge*, x, y) routine queries the tables given in Section 2.4 to obtain the vertex in the adjacent slab corresponding to the local displacement map coordinate (x, y). The getNormal(x, y, *region*) routine returns the normal of either the upper or lower triangular subregion of the quadrilateral region extending away from the origin, beginning at the coordinate (x, y). After summing the polygon normals, the resulting vector must be normalized before it can be used for graphic or haptic rendering.

As each slab is visited, it can be marked by setting a Boolean flag. The drawback to this method for marking slabs is that the Boolean flags for all of the slabs in the model must be reset to *false* before the recursive process is started. A preferable alternative marks surfaces by using a global counter that can be incremented by one each time a new marker value is needed. A slab can be marked by setting its corresponding marker to the value of the global counter.

It should be noted that the pseudocode does not account for variations in angles of the polygons adjacent to the vertex (x, y). A more precise method for averaging the normals would weight each normal by the angle or surface area of each adjacent polygon; but since the mesh is expected to be relatively uniform, this additional computation has proven unnecessary.

Each time the local geometry is modified, the normals in the region surrounding the modified geometry must be recomputed. This burden is placed on the graphic rendering process, as described in Section 4.1.

Chapter 3

Haptic Collision Detection

One of the primary advantages of the displacement map data structure described in Chapter 2 is its efficiency for use with haptic collision detection. Within a slab, a continuous vector field directed from the interior plane to the exterior plane can be defined by linearly interpolating between the rays at the corners of the slab. At slab boundaries, this vector field remains continuous due to a consistent definition of the bilinear patches the separate adjacent regions. In this section, I demonstrate how this continuous vector field is used to define a relationship between arbitrary points in space and corresponding points on the model's surface – a mapping that can be used for efficiently haptic response computation.

The interactive sculpting application is controlled by two separate processes that are independently forked when the program is started. The first process controls haptic interaction – performing collision detection, modifying geometry, and sending response forces to the haptic device at 1000 Hz. The second process controls graphic rendering – display list generation and execution. Having each as a separate process allows them to work at independent rates of execution, such that neither process is a burden to the other.

The haptic and graphic rendering processes share a common data structure for all slab data. As the haptic process modifies the data, the graphic process updates the corresponding display lists to reflect the changes. This section describes the operation of the basic haptic collision detection process. Chapters 4 and 5 discuss methods of modifying the data, and Chapter 6 explains the graphic rendering process.

The basic operations performed by the haptic process are shown in the flowchart in Figure 3-1.

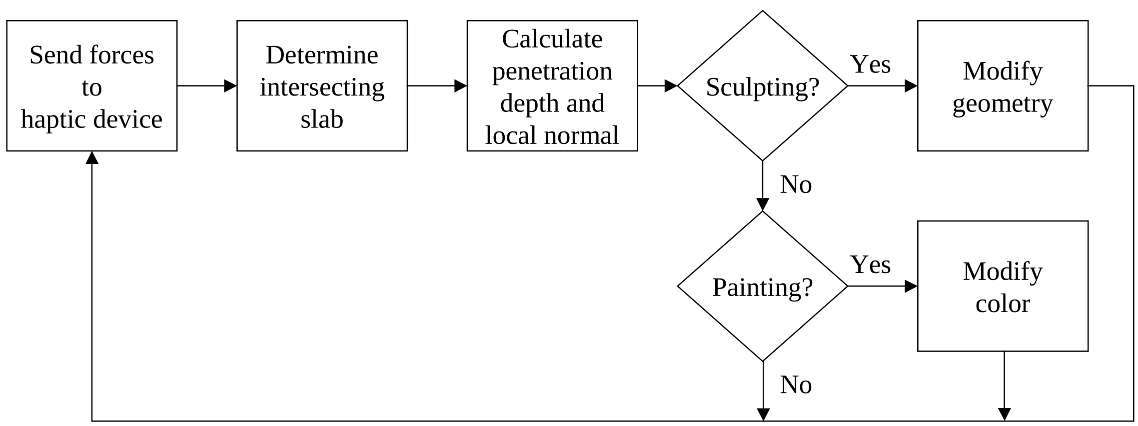

Figure 3-1 – Flowchart for the basic operation of the haptic rendering loop

3.1 Determining Slab Intersections

The first step in the haptic collision detection process is determining in which slab the haptic cursor is located. Although the cursor is visualized as a sphere, we treat it as a point for purposes of collision detection. The algorithm begins with a conservative check to see if the haptic cursor lies inside of an axis-aligned bounding box that encloses the slab. If this succeeds, the next step is to determine how far the cursor has penetrated into the interior of the slab.

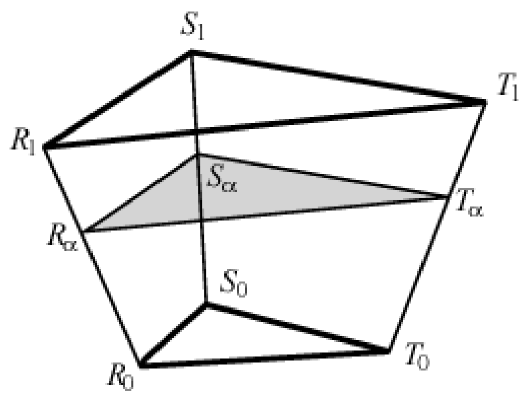

Figure 3-2: A plane with penetration depth α can be defined by interpolating between the triangles at the interior and exterior surfaces of the slab.

Each slab can be though of as the region of space swept out by a triangle whose vertices are linearly interpolated from the interior slab plane to the exterior slab plane. At any distance between the interior plane, which can be assigned an interpolation value of zero, and the exterior plane, which can be assigned an interpolation value of one, this triangle denotes a surface of constant penetration α between zero and one. An example alpha-plane is shown in Figure 3-2. To determine penetration, we can solve for an α interpolation value that defines the plane at the same penetration depth as the haptic cursor.

Each alpha-plane can be expressed in terms of its three corner vertices R_α, S_α, and T_α as follows:

$$A_\alpha x + B_\alpha y + C_\alpha z + D_\alpha = 0\,,$$

where

$$A_\alpha = \begin{vmatrix} 1 & R_{\alpha y} & R_{\alpha z} \\ 1 & S_{\alpha y} & S_{\alpha z} \\ 1 & T_{\alpha y} & T_{\alpha z} \end{vmatrix},\ B_\alpha = \begin{vmatrix} R_{\alpha x} & 1 & R_{\alpha z} \\ S_{\alpha x} & 1 & S_{\alpha z} \\ T_{\alpha x} & 1 & T_{\alpha z} \end{vmatrix},\ C_\alpha = \begin{vmatrix} R_{\alpha x} & R_{\alpha y} & 1 \\ S_{\alpha x} & S_{\alpha y} & 1 \\ T_{\alpha x} & T_{\alpha y} & 1 \end{vmatrix}, \text{ and } D_\alpha = -\begin{vmatrix} R_{\alpha x} & R_{\alpha y} & R_{\alpha z} \\ S_{\alpha x} & S_{\alpha y} & S_{\alpha z} \\ T_{\alpha x} & T_{\alpha y} & T_{\alpha z} \end{vmatrix}$$

Each of R_α, S_α, and T_α are, in turn, defined as interpolants between the slab's extrema R_0, S_0, T_0, R_1, S_1, and T_1 as follows:

$$
\begin{aligned}
R_\alpha &= \left((R_{1x} - R_{0x})\alpha + R_{0x}, (R_{1y} - R_{0y})\alpha + R_{0y}, (R_{1z} - R_{0z})\alpha + R_{0z}\right) \\
S_\alpha &= \left((S_{1x} - S_{0x})\alpha + S_{0x}, (S_{1y} - S_{0y})\alpha + S_{0y}, (S_{1z} - S_{0z})\alpha + S_{0z}\right) \\
T_\alpha &= \left((T_{1x} - T_{0x})\alpha + T_{0x}, (T_{1y} - T_{0y})\alpha + T_{0y}, (T_{1z} - T_{0z})\alpha + T_{0z}\right)
\end{aligned}
$$

When expanded, this series of equations yields quadratic expressions for A_α, B_α, and C_α, and a cubic expression for D_α:

$$
\begin{aligned}
A_a &= A''\alpha^2 + A'\alpha + A \\
B_a &= B''\alpha^2 + B'\alpha + B \\
C_a &= C''\alpha^2 + C'\alpha + C \\
D_a &= D'''\alpha^3 + D''\alpha^2 + D'\alpha + D
\end{aligned}
$$

Note that each of the term A, through D''' depend only on the six points that define the slab boundaries. Thus, these constants can be precomputed for each slab. Looking again at our original expression for the alpha-plane, we can redistribute the terms to express the plane as a cubic function in α:

$$
f(\alpha) = D'''\alpha^3 + \left[A''x + B''y + C''z + D''\right]\alpha^2 + \left[A'x + B'y + C'z + D'\right]\alpha + \left[Ax + By + Cz + D\right] = 0
$$

Thus, given the location of the haptic cursor $p=(p_x, p_y, p_z)^T$, we can solve for α by using the general solution to the cubic or, more simply, by using an iterative approach.

To begin the iterative solution process, we check if the point p lies between the inner and outer extents of the slab, which have α-values zero and one respectively. Plugging these values into the alpha-plane expression yields the following two values:

$$
\begin{aligned}
f(0) &= Ap_x + Bp_y + Cp_z + D \\
f(1) &= D''' + \left[A''p_x + B''p_y + C''p_z + D''\right] + \left[A'p_x + B'p_y + C'p_z + D'\right] + \left[Ap_x + Bp_y + Cp_z + D\right]
\end{aligned}
$$

If $f(0)$ and $f(1)$ have the same sign, then the haptic cursor lies outside of the slab. But if they have opposite signs, then p is bounded by the interior and exterior planes, in which case we can approximate a value for α by using a binary search algorithm. The search process repeatedly divides the search area in half by replacing either the upper or lower extremum of the search bounds, α_{upper} and α_{lower}, by the midpoint of the search region in a manner that insures that $f(\alpha_{upper})$ and $f(\alpha_{lower})$ always have opposite signs. The search is guaranteed to converge since it reduces the search region by half with each iteration. In practice, 15 iterations has been sufficient, yielding an answer for α within an error of $1/2^{15}$.

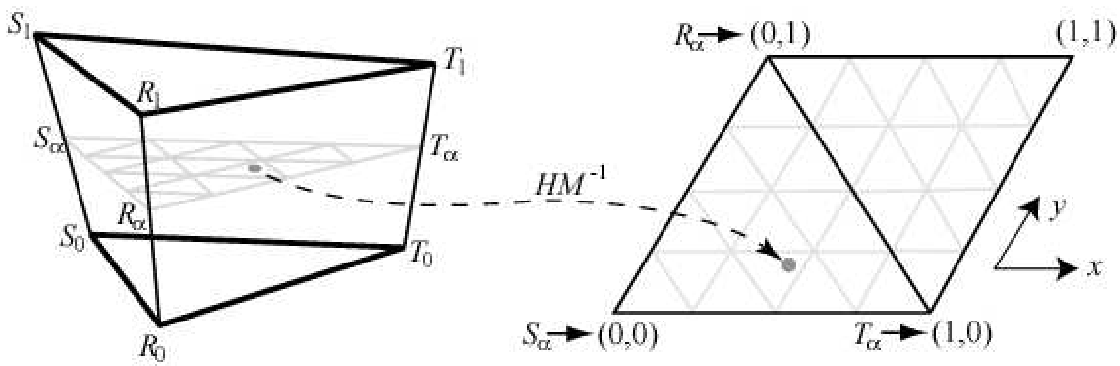

Figure 3-3: Coordinate conversion from the alpha-plane in slab space to the homogeneous displacement map coordinate system.

Now that we have determined where the haptic cursor lies between the inner and outer slab boundaries, we can check its position against the bilinear patches at the other three slab faces. This can be done by casting the point p into a homogeneous coordinate system where the boundary check becomes trivial. This is done by defining two matrices, M and H. The former transforms world coordinates into a system with basis R_α S_α T_α. The latter transforms coordinates into the homogeneous displacement map coordinate system shown in Figure 2-4. These matrices are defined as follows, where n is the width of the displacement map:

$$M = \begin{bmatrix} R_{\alpha x} & S_{\alpha x} & T_{\alpha x} & 0 \\ R_{\alpha y} & S_{\alpha y} & T_{\alpha y} & 0 \\ R_{\alpha z} & S_{\alpha z} & T_{\alpha z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } H = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ n & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can now use HM^{-1} to transform the point p from its position in the interpolated alpha-plane to the homogeneous coordinate system as illustrated in Figure 3-3. M is invertable if R_α, S_α, and T_α are noncollinear, which should always be the case with a well-formed slab.

In the homogeneous coordinate system, a point $h=(h_x, h_y)^T$ is known to lie on the interior of the slab if it satisfies the following three conditions:

$$\begin{gathered} h_x \geq 0 \\ h_y \geq 0 \\ h_x + h_y \leq n - 1 \end{gathered}$$

All other points lie outside of the slab boundaries.

In the current implementation of the haptic sculpting application, slabs are checked in sequence to determine the location of the haptic cursor, yielding a cost that scales linearly with the number of slabs in the model. Since slabs are designed to be nonoverlapping, the algorithm can terminate collision checks as soon as any intersecting slab is found. Potential methods for improving this search are discussed in Section 3.5.

The following sections describe how the homogeneous coordinates recovered by this process can be used to reconstruct the surface of the detailed mesh and to provide appropriate haptic feedback.

3.2 Local Surface Reconstruction

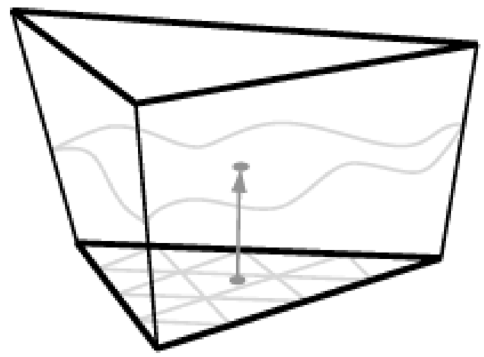

Figure 3-4 – Reconstructing the surface at a given texture index.

As part of the process for detecting slab intersections, we transform the haptic cursor position into a homogeneous coordinate system. We use this transformation to define a relationship between points in space and corresponding points on the surface of the detailed mesh. The displacement value corresponding to the haptic cursor location can be read from the array of displacement values. Since the homogeneous coordinates of the haptic cursor are unlikely to coincide directly with integral coordinate values in the displacement map, the algorithm calculates the displacement as a weighted average of the values at the three nearest coordinates, as indicated in Section 2.2.2. This simple interpolation scheme simulates piecewise linear connectivity between adjacent displacement map coordinates.

The complete displaced surface can be reconstructed by projecting points from the interior slab boundary toward the exterior slab boundary at a distance specified by the displacement value, and in a direction obtained by linearly interpolating the three rays at the corners of the slab, as shown in Figure 3-4. For a known homogeneous coordinate $h=(h_x, h_y)^T$ with a known displacement α, the world coordinate of the corresponding point on the surface of the mesh can be computed as follows: First, determine the vertices at the corners of the alpha-plane:

$$R_\alpha = \left((R_{1x} - R_{0x})\alpha + R_{0x}, (R_{1y} - R_{0y})\alpha + R_{0y}, (R_{1z} - R_{0z})\alpha + R_{0z}\right)$$
$$S_\alpha = \left((S_{1x} - S_{0x})\alpha + S_{0x}, (S_{1y} - S_{0y})\alpha + S_{0y}, (S_{1z} - S_{0z})\alpha + S_{0z}\right)$$
$$T_\alpha = \left((T_{1x} - T_{0x})\alpha + T_{0x}, (T_{1y} - T_{0y})\alpha + T_{0y}, (T_{1z} - T_{0z})\alpha + T_{0z}\right)$$

Then, interpolate between these three points using the interpolation weights:

$$T_{weight} = \frac{h_x}{n-1}$$

$$S_{weight} = \frac{h_y}{n-1}$$

$$R_{weight} = 1 - T_{weight} - S_{weight}$$

to yield the surface point:

$$R_{weight} * R_\alpha + S_{weight} * S_\alpha + T_{weight} * T_\alpha$$

3.3 Calculating Cursor Penetration

Recall from section 2.1 that displacement values are stored as normalized scalars that indicate the relative depth of the surface between the inner and outer extents of the slab. Thus, if the α value for the cursor position recovered in Section 3.1 is greater than the corresponding surface displacement value recovered in Section 3.2, then the haptic cursor has not penetrated the surface of the detailed mesh, and no force needs to be returned to the haptic device. If, however, the α value is less than the surface displacement value, then the cursor has penetrated the mesh by a magnitude equal to the distance between the cursor position and the corresponding point on the surface of the mesh. In this case, a response force should be applied proportional to the depth of penetration, as indicated in the following section.

3.4 Calculating a Response Force

If the haptic cursor has penetrated the surface of the displaced mesh, a response force is sent to the haptic device with the assistance of the GHOST API, provided by Sensable Technologies, Inc. Calculating the response force requires knowledge of the *surface contact point*, or *proxy position* – the point of interaction constrained to the surface of the model [Ruspini97] [SensAble99]. The GHOST API uses the surface contact point and the local surface normal to calculate a response force that accounts for surface spring and damping characteristics, as well as static and dynamic friction.

Before computing the surface contact point, the algorithm must determine the direction of the local surface normal. Every small triangle in the displaced surface of a particular slab can be considered to have its own local normal. But if the haptic feedback loop applies a response force in the direction of this local normal, at a magnitude proportional to the penetration distance, we encounter problematic behavior near major concavities and convexities in the surface of the model. Figure 3-5(a) shows five adjacent vertices of a displaced surface. As the position of the haptic cursor is pulled along the gray line, a force is applied in the direction of the local surface normal. The resulting surface feels as though it has a discontinuity at the peak. Furthermore, the ambiguous forces at the concavity can cause instabilities in the haptic device as opposing forces alternately attempt to achieve an unattainable equilibrium state.

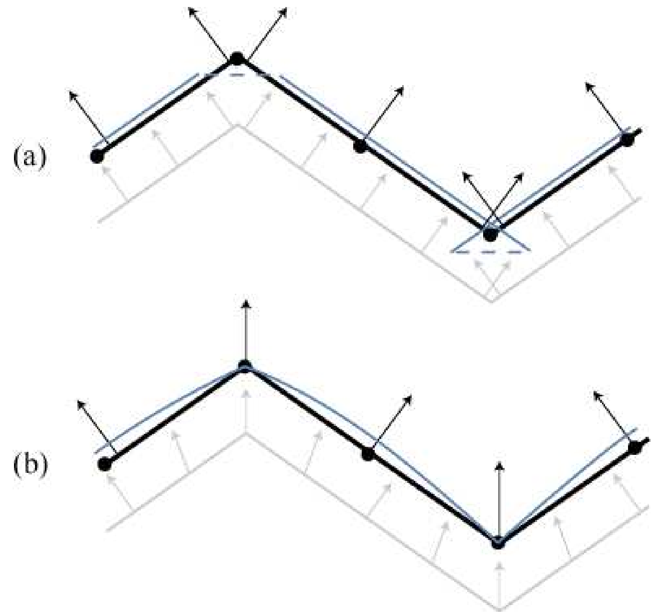

Figure 3-5 – Surface reconstruction using piecewise normals (a) and interpolated normals (b). Haptic cursor penetration through the model surface results in the reconstructed surface slightly above the polygons of the original mesh.

A better solution stores an additional normal with every vertex in the displaced surface. This supplementary vector is computed as the average of the normals of the surfaces adjacent to each vertex, as described in Section 2.5. At each point in the slab, a local normal can be calculated as a weighted average of the normals of the three nearest vertices. This calculation uses the same interpolation weights that were used for computing the local surface displacement.

Finally, using the penetration distance and the local surface normal, the algorithm computes the surface contact point using a method similar to Morgenbesser’s force shading algorithm. The surface contact point is offset from the haptic cursor position by a magnitude equal to the penetration distance, in the direction of the local surface normal. As can be seen in Figure 3-5(b), the resulting point may not lie directly on the surface of the displaced mesh, but the magnitude and direction of the resulting force provide a convincing representation of the local geometry, as verified by Morgenbesser's user experiments [Morgenbesser95].

There are a few anticipated problems with the surface reconstruction algorithm. Figure 3-6 illustrates the reconstructed surface when a sufficiently large force is applied to a displacement map with a fine level of subdivision. The resulting forces cause the reconstructed surface to appear ambiguous near the concavity. Nonetheless, the force field along the path of the haptic cursor does not exhibit any discontinuities, so this has not been a noticeable problem in the implemented system.

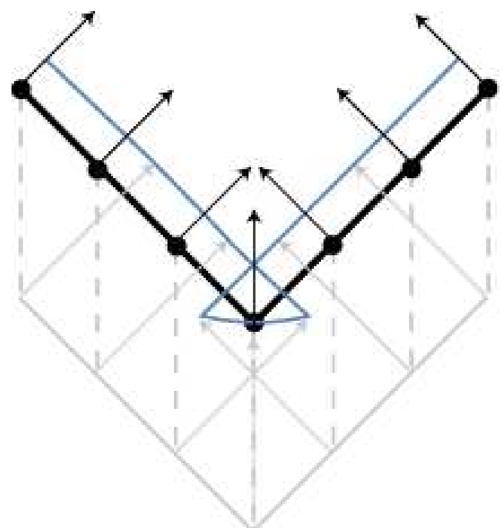

Figure 3-6 – Surface reconstruction ambiguity resulting from a fine mesh with large haptic forces

3.5 Improving Efficiency

Within a given slab, calculating the response force requires constant time, regardless of the complexity of the internal geometry. Assuming the slabs are relatively large with respect to the size of the model, the haptic cursor will very frequently lie inside of the same slab as the previous iteration of the haptic loop. Thus, if we begin the collision-detection routine by checking for collisions in the most recently intersected slab and its immediately adjacent neighbors, we can improve the typical system performance. The system could be further optimized by storing the slabs in a bounding sphere hierarchy, as suggested by Ruspini et. al. [Ruspini97], or by using any alternative spatial hierarchy. In practice, I have not found these optimizations to be necessary.

Chapter 4

Modifying Geometry

Since the coordinates of the displacement maps provide a uniform scaffolding that fully covers the surface of the model, they provide an ideal framework for locally modifying geometry or color attributes. The displacement maps within each slab have an inherent notion of adjacency and connectivity, and we can use our inter-slab management routines to seamlessly span adjacent slabs as operations are used to modify desired attributes.

4.1 Removing Volume with Surface Clipping

Using the adjacency tables to quickly traverse slabs, we can construct a flood-fill method that walks from vertex to vertex within a constrained region of the surface, modifying geometry or color according to a desired function. To begin the sculpting process, the system first positions a tool that will be used to modify the geometry. I have chosen to use a simple sphere that clips away the geometry that intersects with its surface. As part of the collision detection process, the haptics algorithm already computes the local surface normal $\vec{n}$, the penetration distance d, and the surface contact point, $S\dot{C}P$. To allow the tool to penetrate the surface by an amount proportional to the user-applied pressure, the sculpting routine positions the center of the tool as follows:

$$\dot{center} = \dot{SCP} + \vec{n} * (r - d),$$

where r is the radius of the desired tool. This is illustrated in Figure 4-1.

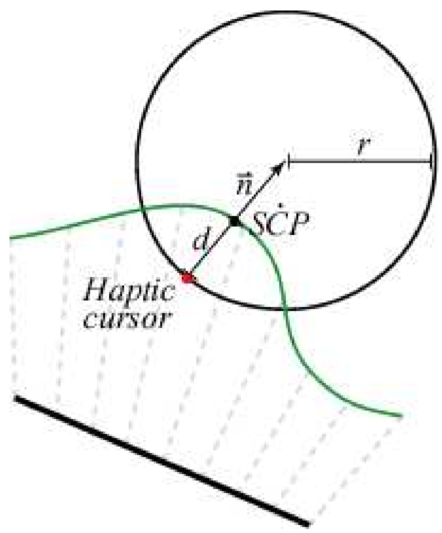

Figure 4-1 – Positioning a spherical sculpting tool at the surface of a slab.

Once the tool has been positioned, the sculpting algorithm clips the geometry that falls inside of the sphere. It begins by "seeding" a recursive process at the homogeneous coordinate nearest to the haptic cursor. At each visited coordinate, the surface projection line is calculated. The point at which this line intersects the interior slab face can be found by interpolating the R_0, S_0, and T_0 slab vertices using the interpolation weights given in Section 3.2. Likewise, the point at which the ray intersects the exterior slab face can be found by interpolating the R_1, S_1, and T_1 slab vertices. These two points are used to parametrically define the surface projection line. If this line intersects the spherical sculpting tool at a point on the interior of the mesh, then the surface is clipped at the sphere boundary.

The procedure continues by recursing to all adjacent displacement map coordinates that fall inside of the spherical tool. As homogeneous coordinates are visited, they are tagged using an array of status flags. The displacement map can only be modified within the confines of the slab, so displacements must be capped at appropriate minimum and maximum values. Pseudocode for the recursive sculpting algorithm is given below.

```
carveSurface(x, y, tool) {
  markThisSlab()
  if (vertexIsMarked(x, y)) return
  markVertex(x, y)

  if (not(toolIntersectsSurface(tool))) return
  clipSurfaceToIntersection(tool)

  if (x-1 ≥ 0) {
    carveSurface(x-1, y, tool)
    carveSurface(x-1, y+1, tool)
  }
  if (y-1 ≥ 0) {
    carveSurface(x, y-1, tool)
    carveSurface(x+1, y+1, tool)
  }
  if (x+y+1 ≤ meshSize-1) {
    carveSurface(x, y+1, tool)
    carveSurface(x+1, y, tool)
  }
  if (x==0) {   //(x, y) is on the edge adjacent to Neighbor 0
    adjacentSlab = getNeighbor(EDGE0)
    if (not(adjacentSlab->isMarked())) {
      seed = getAdjacentVertex(EDGE0, x, y)
      adjacentSlab->carveSurface(seed.x, seed.y, tool)
    }
  }
  if (y==0) {   //(x, y) is on the edge adjacent to Neighbor 1
    adjacentSlab = getNeighbor(EDGE1)
    if (not(adjacentSlab->isMarked())) {
      seed = getAdjacentVertex(EDGE1, x, y)
      adjacentSlab->carveSurface(seed.x, seed.y, tool)
    }
  }
  if (x+y==meshSize-1) { //(x, y) is on the edge adjacent to Neighbor 2
    adjacentSlab = getNeighbor(EDGE2)
    if (not(adjacentSlab->isMarked())) {
      seed = getAdjacentVertex(EDGE2, x, y)
      adjacentSlab->carveSurface(seed.x, seed.y, tool)
    }
  }
}
```

Like the accumulateNormals algorithm presented in Section 2.5, the carveSurface routine uses a series of checks to walk across of the surface of the mesh, spanning to adjacent slabs whenever an edge is reached. To maximize efficiency, the algorithm must maintain a record of which slabs have been visited, as well as tracking the individual vertices that have been touched. To mark slabs, the carveSurface routine uses the same marking algorithm outlined in Section 2.5. A similar algorithm is used to mark individual vertices. Each slab maintains a flag for each of its vertices, packed into an integer array. To mark a vertex, the corresponding flag is assigned the value of the global counter that is used by the slab-marking routines. A vertex is considered to be "marked" if its corresponding flag has the same value as the global counter. Thus, vertices can be "unmarked" by simply incrementing the counter value.

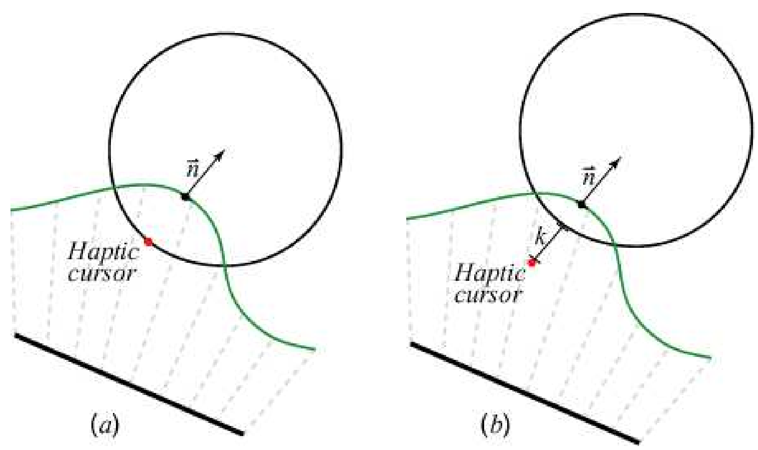

Figure 4-2: A narrow band of resilient material can be provided by offsetting the center of the haptic tool in the direction of the local surface normal. Here we see tool placement calculated without (*a*) and with (*b*) a resilient material band.

When placing the tool center as described at the beginning of this section, the user begins to sculpt the model as soon as the haptic cursor contacts the surface. A preferable alternative provides a narrow band of material around the surface of the model in which the user can receive tactile feedback from the geometry without modifying it. As illustrated in Figure 4-2, a resilient band of material with thickness *k* can be added by calculating the center of the tool as follows:

$$\dot{center} = S\dot{C}P + \vec{n} * (r - d + k)$$

Sculpting should occur only if $d > k$.

After the surface has been clipped against the sphere, the local normals for all affected slabs must be recomputed to insure proper graphic and haptic rendering. The process of reevaluating the surface normals is computationally expensive, so I have chosen to place this burden on the graphic rendering process. When the graphics routine encounters a slab with modified geometry, it recalculates the local normal vectors before displaying the slab. This means that the normal vectors used by the haptic routine may be incorrect for a fraction of a second. In practice, the tactile artifacts resulting from this inconsistency are negligible.

Each slab has an associated flag that is used to indicate to the graphic rendering routine if the surface has been modified. When the haptic rendering routine edits any surface characteristics, it sets the “modified” flag. The graphic rendering routine resets the flag when the display list and normal vectors have been updated to reflect the changes.

The sculpt routine adds substantial overhead to the basic collision-detection algorithm described in Section 3, so the routine is not executed with each iteration of the haptic collision detection process. Instead, sculpting is performed at 10Hz – approximately once per every 100 iterations of the haptic loop – with minimal impact on the running time of the basic haptic rendering routine.

4.2 Adding Volume

By centering a spherical tool at the surface contact point, a similar routine can be used for *adding* material to the surface of the model. In this case, as the recursive algorithm marches along the surface of the mesh, a thin layer of material is deposited with a thickness linearly proportional to the distance from each point on the mesh to the center of the spherical tool. Thus, material is added in a conical shape, at a constant speed, regardless of pressure.

4.3 Improving Tool Placement

As discussed in Section 3.3, the forces generated by the haptic system may not correspond exactly with the actual surface of the mesh. The force shading algorithm provides appropriate feedback forces, but having the surface contact point displaced from the actual surface can cause problems with the sculpting routine.

Figure 4-3 shows a surface with a local normal $\hat{n}$ that differs significantly from the direction $\hat{s}$ in which the surface mesh is projected. The surface contact point ($S\dot{C}P$) is placed at a distance d from the haptic cursor, in the direction of the local normal. However, when the sculpting tool is centered at this point, it does not contact the surface as expected. As increasing pressure is applied to the surface, the difference between the surface contact point and the actual surface is amplified. Thus, surfaces that are not oriented perpendicular to the direction of projection may appear to be substantially more resilient than other surfaces.

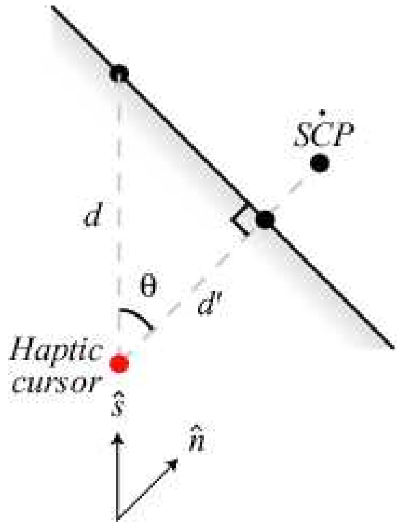

Figure 4-3 – Improving surface contact point (SCP) calculations for sculpting.

To remedy the problem, the algorithm can more closely estimate the actual surface position by determining the distance *d'* between the surface contact point and the expected surface location. We can express the cosine of the angle theta between $\hat{s}$ and $\hat{n}$ as the dot product of the two vectors. Thus, we can express the *d'* as follows:

$$d' = d * (\hat{n} \cdot \hat{s})$$

The estimated surface position can then be computed by displacing the position of the haptic cursor by a distance of d' in the direction of the local surface normal $\hat{n}$.

Chapter 5

Haptic Painting

The haptic interface can be used to paint color onto the surface of a model by using a straightforward modification of the sculpting algorithm described in the previous chapter. Here, I introduce a simple algorithm for "airbrushing" color onto the surface of the mesh.

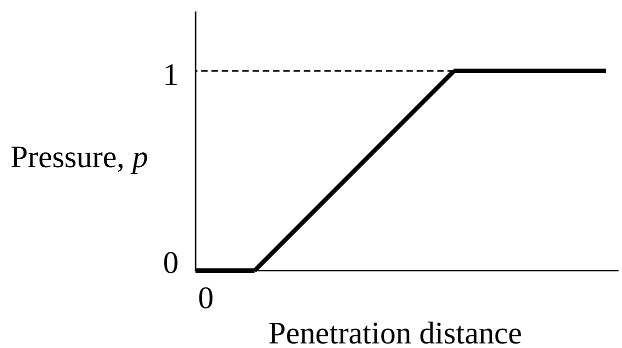

Figure 5-1 – Brush pressure, p, as a function of the penetration distance

In the implemented system, the virtual paintbrush is represented as a sphere, centered at the modified surface contact point described in Section 4.3. When the user touches the surface of the model, the paint process calculates the pressure p with which the paint should be applied. This is based on the penetration distance, normalized to a range between zero and one, as illustrated in Figure 5.1 Calculating the pressure as zero for small penetration distances allows the user to lightly touch the surface without modifying any surface properties.

As with the geometry modification routine, the recursive floodfill routine is “seeded” at the homogeneous coordinate nearest to the haptic cursor. At each iteration of the procedure, a new color c_n for the coordinate is calculated based on the old color c_o, the paintbrush color c_b, the pressure *p*, the brush radius *r*, and the distance *d* between the center of the brush and the coordinate position on the surface of the displaced mesh.

The color is applied with a normalized intensity *k* equal to $p * (1 - \frac{d}{r})$. Thus, the most intense color is applied with greater pressure, nearest to the center of the brush, linearly decreasing out toward the edges of the sphere. The new color is computed by blending the existing color and the brush color as follows:

$$c_n = c_p * k + c_o * (1 - k)$$

The procedure continues by recursing to all adjacent texture coordinates that lie within the sphere of the paintbrush, as is done in the haptic sculpting algorithm. As described in Section 4.1, the visited coordinates can be marked with an array of status flags to prevent redundant effort. Like the sculpting algorithm, the paint routine is only executed 10 times per second to keep the basic collision detection routine unencumbered by computationally intensive tasks.

Chapter 6

Graphic Rendering

Interactive visual feedback forms an important component of a effective haptic interface, so it vital that the graphic rendering algorithm for the slab data be carefully optimized. Since the graphic and haptic routines require different rates of execution, each task is run as a separate process, allowing the graphic process to render the scene at a frame rate independent of the haptic process. The flowchart for the basic operation of the graphic rendering algorithm is shown in Figure 6-1.

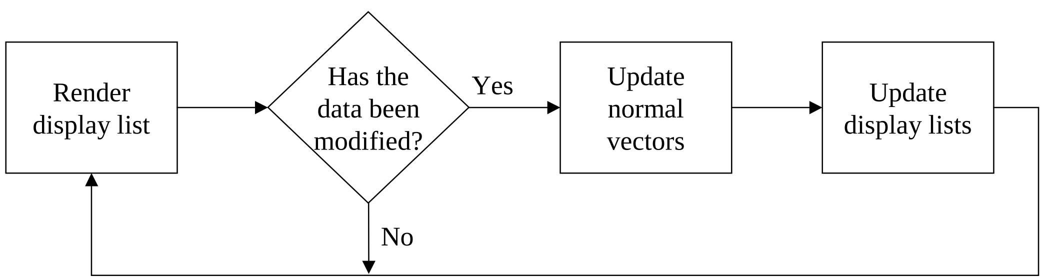

Figure 6-1 – Flowchart for the basic operation of the graphic rendering loop.

To take maximum advantage of the graphic rendering hardware, the detailed surface for slab is decomposed into a polygonal mesh that can be rendered with the assistance of the OpenGL graphics library [Woo96]. Whenever possible, the application uses OpenGL display lists to draw features that will be displayed for multiple frames in succession.

To construct the graphically rendered mesh for each slab, the display routine must quickly determine the displaced surface position for each vertex in the displacement map. This can be done as described in the local surface reconstruction process in Section 3.2. The surface vertices are stitched together in a triangular mesh by displaying each neighborhood of four coordinates as two triangles. For the quadrilateral regions of the mesh that are divided by the line $h_x+h_y=n-1$, only the triangular region nearest to the mesh origin is displayed. As the display lists are defined, each vertex is assigned a color and normal indicated by the appropriate data array.

Since assigning per-vertex color values is an expensive operation for the OpenGL rendering library, the process of coloring the model can be accelerated by instead treating the array of color values as a texture map. Using this method, texture coordinates, rather than color values, are associated with each vertex in the mesh. Each time the color values are changed, the texture map is reloaded into texture memory. This approach to coloring the model provides the added benefit of decoupling the resolutions of the displacement data and the color data. Thus, color can be mapped to the surface of a slab at a higher density than the displacement data with negligible impact on rendering speed.

Unfortunately, using the OpenGL interpolation scheme for texture data introduces its own set of concerns. First, for texture coordinates along the boundary of the slab, smooth texture interpolation relies on defining a border color for the texture data. The rules for defining color values along the texture border are illustrated in Figure 6-2. Recall that only the lower-left half of the texture space is utilized. Along the edges of the texture map, color values should be copied to the border pixel, as shown for pixels A, C, and D. Across the diagonal boundary, color should be painted into the adjacent pixels at half intensity, since these pixels inherit color values from two different neighbors. This is shown for pixels A and B.

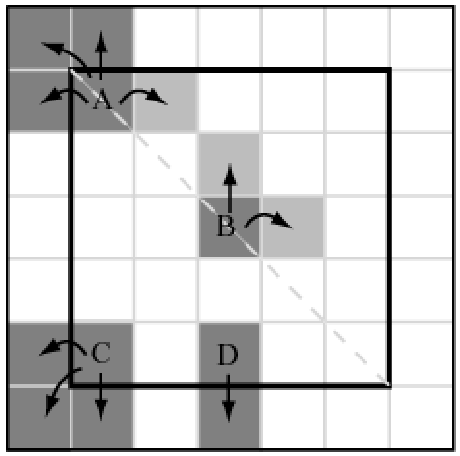

Figure 6-2: Applying color to border vertices. Edge vertices A, C, and D have their texture color copied to the adjacent border pixels. Diagonal boundary vertices A and B paint the pixels across the diagonal with half the original intensity.

The second concern with OpenGL texture rendering is the color interpolation scheme. For normal and displacement data, the surface reconstruction algorithm interpolates between the values at the three nearest homogeneous coordinates as indicated in Section 2.2.2. In contrast, OpenGL uses bilinear interpolation method that weights contributions from four adjacent vertices. The differences between these interpolation methods are illustrated in Figure 6-3. In most cases, the visual disparity between the methods is negligible, and well worth the extra speed gained by using the hardware to apply color. The artifacts can be reduced by increasing the resolution of the texture map.

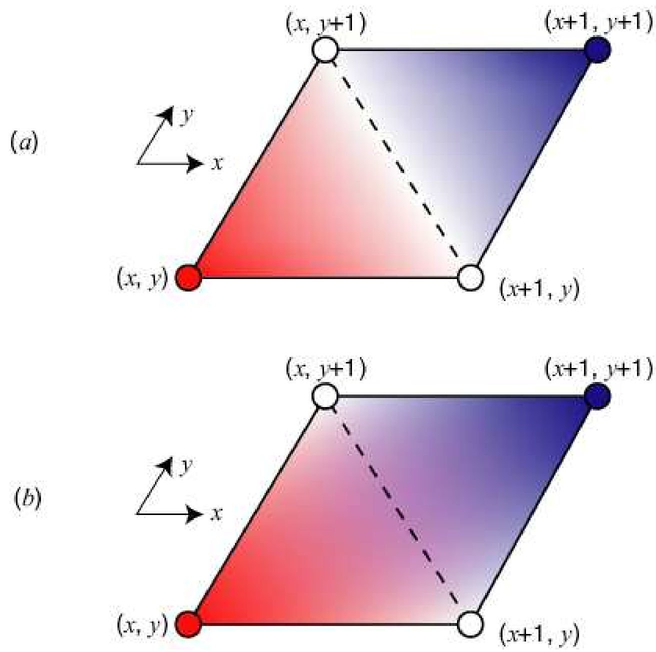

Figure 6-3: Color interpolation using a custom interpolation method (*a*), and OpenGL (*b*).

For each slab, the graphics routine maintains an OpenGL display list that can be quickly rendered in the display loop. Each time the haptic process modifies a slab, it sets a Boolean flag to indicate that the graphics process must update its corresponding display list. Thus, the graphics system is concerned only with local updates for modified slabs, while reusing display lists for unmodified regions of geometry. The displaced mesh for each slab is formed by $(n\text{-}1)^2$ triangles, where n is the number of coordinates along each dimension of the displacement map. Thus, the biggest rendering bottleneck is simply the high polygon count. If graphic rendering is unacceptably slow, one alternative is to generate the visible mesh from a coarse subset of the coordinates in the displacement map.

Chapter 7

Results

The system described above was implemented on an SGI Octane with dual 250 MHz MIPS R10000 processors and 1.5 Gb memory, running IRIX 6.5. For the haptic interface, I used the A-model PHANToM® Premium haptic device from SensAble Technologies, Inc.® with the *GHOST* Software Development Toolkit, Version 3.0.

Figures 7-1 and 7-2 present two polygonal models that were converted to the slab data structure and subsequently edited using the haptic sculpting system. For each model, a coarse base mesh was generated using the *qslim* application by Michael Garland, which is uses a quadric error metric to maintain general surface characteristics [Garland99]. As described in Section 2.1, the normals of the base mesh triangles were averaged at each vertex to identify slab edges. Slab thickness was initialized with a default value for each model, after which the default length and orientation of the slab edges were modified as necessary to properly enclose all of the features of the original mesh.

With the slabs in place, the displacement maps were composed by projecting rays for each homogeneous coordinate from the interior face to the exterior face of the slab. Displacement values were set to indicate the intersection points at which these interpolated rays first pierced the surface of the detailed mesh. In a few instances, malformed models caused the projection rays to miss the surface of the detailed mesh altogether. For these cases, after the remainder of the displacement values had been computed, the remaining holes were filled by averaging the values at the neighboring displacement map coordinates.

Figure 7-1(*a*) shows the polygonal mesh used to generate the teddy bear model. The initial model is composed of 3192 triangles. This was converted to a model with 200 slabs, illustrated in Figure 7-1(*b*). The texture and displacement maps for each slab have 16 components along each edge, resulting in a graphically reconstructed mesh with 45,000 triangles. An edited version of the model is shown in Figure 7-1(*c*). Note the OpenGL texture interpolation artifacts discussed in Chapter 6 along the high-contrast regions near the eyes and embossed characters.

The bunny model, provided courtesy of the Stanford Computer Graphics Laboratory, was used to construct the second model. The original mesh, shown in Figure 7-2(*a*) is composed of 18,268 triangles. This was converted to a model with 150 slabs, as shown in Figure 7-3(*b*). As with the bear model, the texture and displacement maps for the bunny have 16 components along each edge to yield a graphically reconstructed mesh with 33,750 triangles. An edited version is shown in Figure 7-2(*c*).

With the teddy bear model, the haptic system typically performs a single iteration of the collision detection and response process in under 0.65 milliseconds, thus utilizing 65% of the available processor time. For the bunny model, the haptic loop typically requires less than 50% of the available processor time. For either model, the additional burden of performing sculpting or painting operations at 10Hz appears to have negligible impact on overall computational cost. With either model, the graphic rendering system maintains a refresh rate of approximately 5-10 Hz, with minimal performance degradation during sculpting or painting operations.

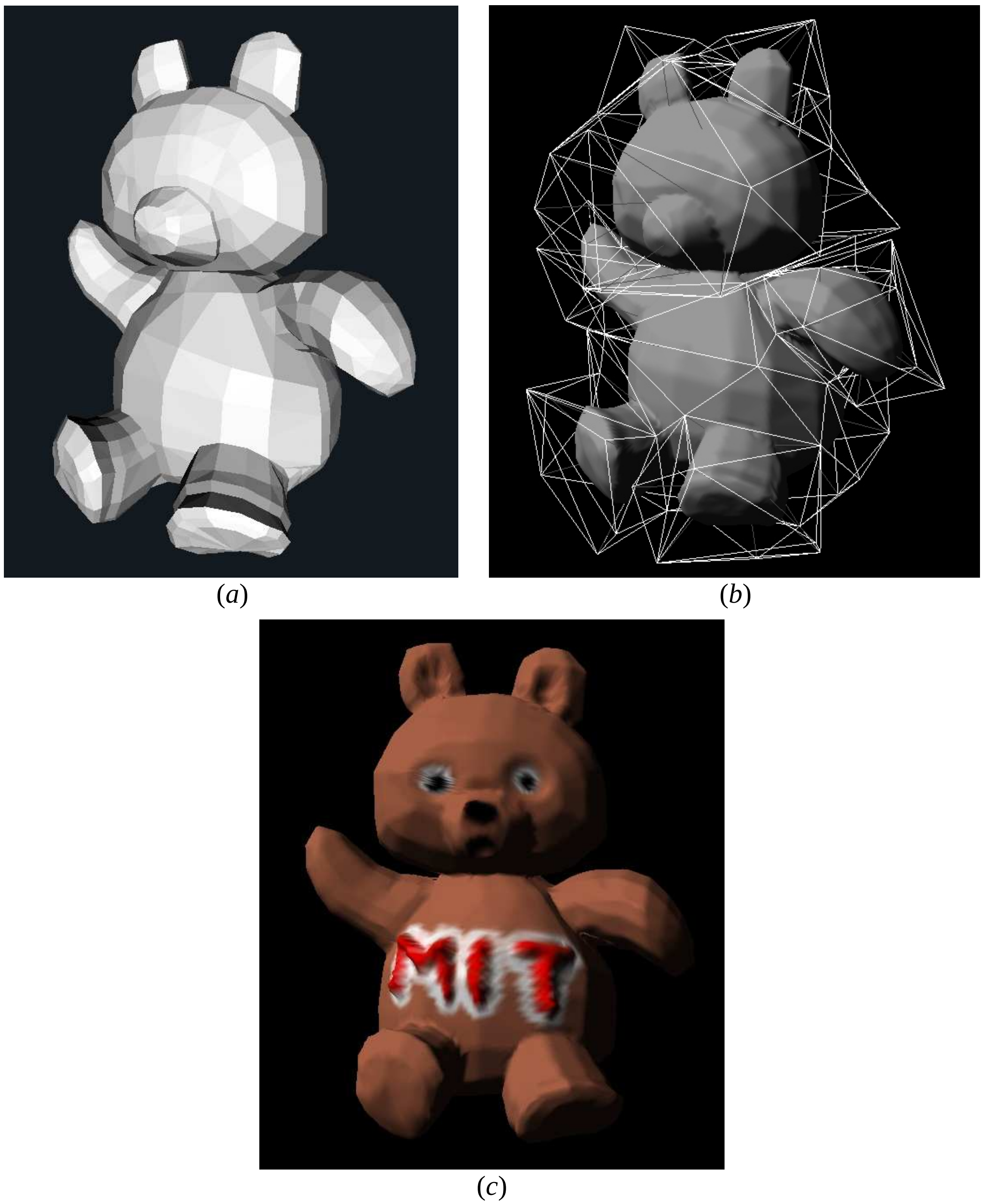

Figure 7-1: The original polygonal mesh for the bear model (*a*) is composed of 3192 triangles. The slab representation (*b* and *c*) is composed of 200 slabs, forming a visually rendered mesh with 45,000 triangles.

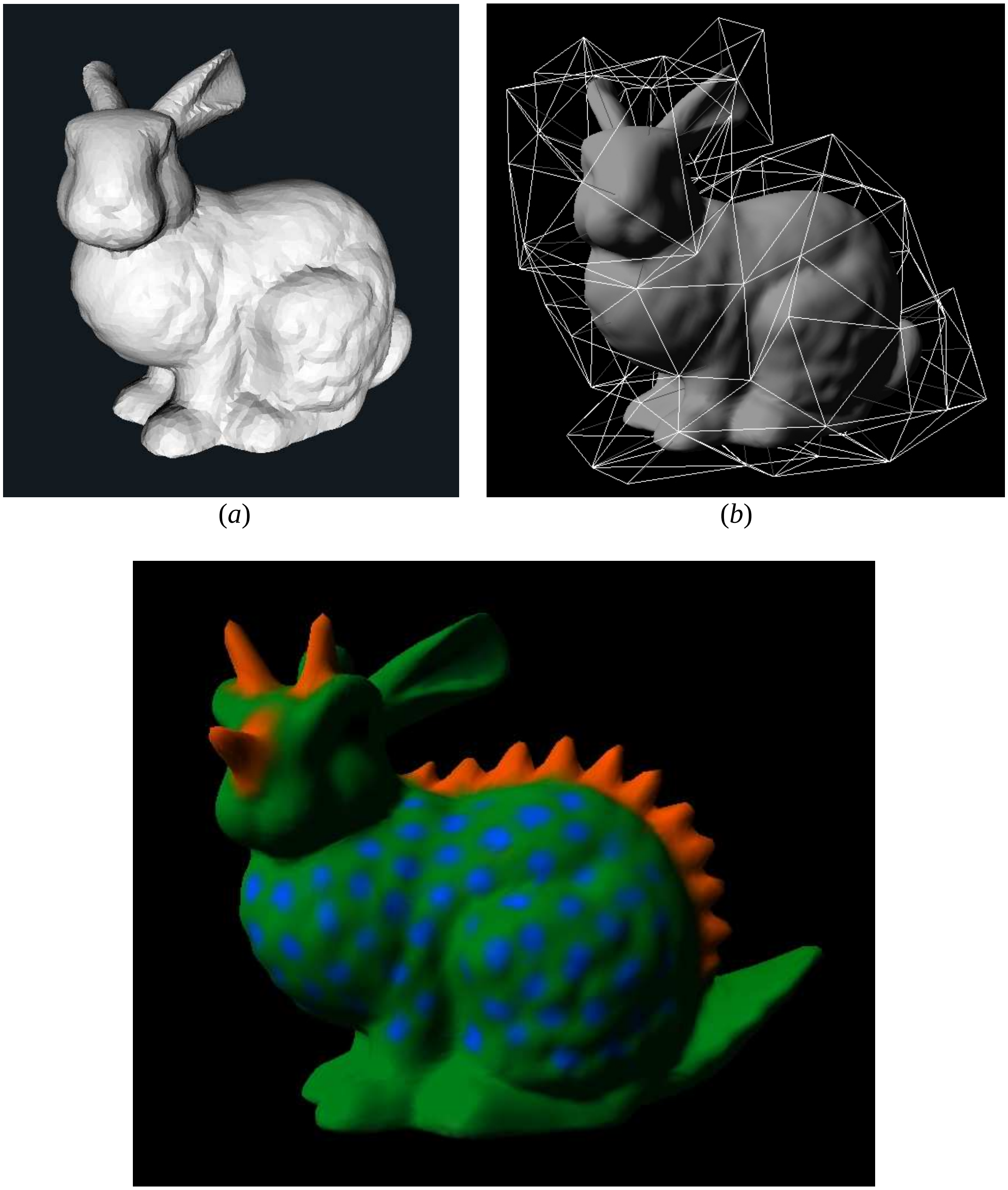

Figure 7-1: The original polygonal mesh for the bunny model (*a*) is composed of 18,268 triangles. The slab representation (*b*) is formed with 150 slabs. The spotted three-horned bunnysaur (*c*) is rendered with 33,750 triangles.

Chapter 8

Conclusions and Future Work

Haptic displacement maps fulfill much of the promise that has long been offered by haptic interfaces – Intuitive methods of interaction for rapid editing of complex models. Still, this data structure is certainly not a panacea to the difficulties of digital editing. Slabs are effective for performing local modifications to a model in a thin region surrounding the surface of the initial mesh, but the sculpting algorithms are not useful if dramatic changes need to be made to the underlying topology of the object. A torus will never be made into a sphere, and vice-versa.

There are additional limits on the types of models that can be represented using haptic displacement maps. Slabs are designed to be used with “thick” models, with clear interior and exterior regions, and without thin or overlapping topological features. Even with the bunny model shown in Figure 7-1, it is possible to “push through” thin regions of geometry like the ears, causing the haptic cursor to pop out the opposite side of the model.

Despite these restrictions, there are certain applications for which haptic displacement maps are particularly well suited. Many physical phenomena such as weathering, erosion, or biological growth limit their influence to a thin region near the surface of a model. This suggests potential for using the data structure for a variety of physically-based modeling applications.

Since the data structure provides a uniform scaffolding of points along the surface of a model, it naturally lends itself to supporting arbitrary surface characteristics that can be represented using one or more scalar arrays. Other supported characteristics could include material resilience, frictional coefficients, specularity, opacity, or other material properties. A single slab might even be used to store multiple layers of material, some of which could potentially be exposed by the sculpting process.

Haptic systems continue to offer a promising alternative to traditional methods of digital modeling. Hopefully, future research will continue to offer creative applications that exploit the potential of the intuitive nature of these powerful interfaces.

References

[Agarwala99] Aseem Agarwala. *Volumetric Surface Sculpting*. S.B. Thesis, Massachusetts Institute of Technology, 1999.

[Burdea96] Grogore C. Burdea. *Force and Touch Feedback for Virtual Reality*. New York: Wiley-Interscience, 1996.

[Dachille99] Frank Dachille IX, Hong Qin, Arie Kaufman, Jihad El-Sana. "Haptic Sculpting of Dynamic Surfaces," In the Proceedings of *Symposium on Interactive 3D Graphics 1999*, pp 103-110.

[Dorsey99] Julie Dorsey, Alan Edelman, Henrik Wann Jensen, Justin Legakis, Hans Kohling Pedersen. "Modeling and Rendering of Weathered Stone," Proc. of *SIGGRAPH 1999*, pp 225-234.

[Ferley99] Eric Ferley, Marie-Paule Cani, Jean-Dominique Gascuel. "Sculpture Virtuelle," Proc. of *Les journées AFIG 99*, Reims, November 1999.

[Garland99] Garland, Michael. *Quadric-Based Polygonal Surface Simplification*. Ph.D. Thesis, Carnegie Mellon University, 1999.

[Gregroy00] A. Gregroy, S. Ehmann and M. C. Lin. "*inTouch*: Interactive Multiresolution Modeling and 3D Painting with a Haptic Interface," In the Proceedings of *IEEE Virtual Reality Conference 2000*.

[Guskov00] Igor Guskov, Kiril Vidimče, Wim Sweldens, Peter Schröder. "Normal Meshes," In the Proceedings of *SIGGRAPH 2000*, pp 95-102.

[Hoppe96] H. Hoppe. "Progressive Meshes," In *Computer Graphics* Proceedings, Annual Conference Series, 1996 (ACM SIGGRAPH '96 Proceedings), pp 99-108, 1996.

[Lee00] Aaron Lee, Henry Moreton, Hugues Hoppe. "Displaced Subdivision Surfaces," In the Proceedings of *SIGGRAPH 2000*, pp 85-94.

[McNeely99] William A. Mcneely, Kevin D. Puterbaugh, James J. Troy. "Six Degree-of-Freedom haptic Rendering Using Voxel Sampling," In the Proceedings of *SIGGRAPH 1999*, pp 401-408.

[Morgenbesser95] Morgenbesser, H. B., "Force Shading for Haptic Shape Perception in Haptic Virtual Envornments." M.Eng. thesis, Massachusetts Institute of Technology, September 1995.

[Phong75] Phong, B. T., "Illumination for Computer Generated Pictures." *Communications of the ACM*, 18(6), pp 311-317, June 1975.

[Ruspini97] Diego C. Ruspini, Krasimir Kolarov and Oussama Khatib. “The Haptic Display of Complex Graphical Envoronments,” In the Proceedings of SIGGRAPH 1997, pp 345-352.

[SensAble99] SensAble Technologies, Inc. *GHOST SDK Version 3: General Haptic Open Software Toolkit*. January, 1999.

[SensAble01] SensAble Technologies, Inc. *Freeform Modeling System*. http://www.sensable.com. 2001.

[Woo96] Woo, Mason, Jackie Neider, Tom Davis. *OpenGL Programming Guide, 2nd ed*. Reading, Massachusetts: Addison-Wesley, 1996.