BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M5

# "Using Texture Mapping and WYSIWYG Painting to Create Geometric Tools"

by

Oren J. Tversky

# Using Texture Mapping and WYSIWYG Painting to Create Geometric Tools [*]

Oren J. Tversky

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the
degree of Master of Science in the Department of Computer Science
at Brown University

May 1993

[*]This work was supported in part by grants from NSF/DARPA Science and Technology Center for Computer Graphics and Scientific Visualization, IBM, NCR, Sun Microsystems, DEC, HP, and ONR grant N00014-91-J-4052 ARPA order 8225.

This research project by Oren J. Tversky is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]

Professor John F. Hughes
Advisor

3/30/93

Date

## Abstract

The process of modeling geometric objects often entails modification of a portion of a surface—a *subsurface*—which is part of a larger geometric shape. Most geometric modeling applications do not provide an interface that allows for user specification of arbitrarily shaped subsurfaces. We have developed two polygonal modeling applications which overcome this limitation. They employ an interaction technique based on 3D WYSIWYG painting which allows fine control over the precise shape of a subsurface; the painted region determines the subsurface that is to undergo geometric alteration. This correspondence between painting and geometric properties constitutes the basis of our modeling systems. And, while we operate on polygonal models exclusively, we believe the technique may be generalized to higher-level models.

Furthermore, this new type of modeling system has unexpected benefits. For instance, modeling often necessitates some form of refinement prior to geometric alteration. Our modeling systems accomplish this by using a variety of common image processing algorithms applied to texture maps which are created as a bi-product of 3D WYSIWYG painting. Also, our established correspondence between paint and geometry, enables us to create new modeling paradigms by associating painting properties with geometric properties. Finally, we offer a range of applications that can benefit from such a technique.

**CR Categories:** I.3.5 [Computer Graphics]: Geometric Transformations and Object Modeling I.3.6 [Computer Graphics]: Interaction Techniques I.3.7 [Computer Graphics]: Texture

**Additional Keywords and Phrases:** Texture Mapping, Painting, Direct Manipulation, Geometric Modeling, Image Processing.

## Contents

# List of Figures

## 1 Introduction

There are several components involved in developing geometric modeling systems, some of which are:

- an underlying model,
- operations to perform on the model,
- a user interface

By an *underlying model*, we refer to the characteristics of the geometry that is to be operated on. Underlying models include such disparate models as polygons, spline based surfaces, volumetric data, and particle systems.

The underlying model, however, constitutes only a portion of a modeling system. In the process of creating a desirable shape, many geometric *operations* are performed on the underlying model. For example, in a spline based system, one operation might be moving the control points. The spline surface is the underlying model and the movement of control points is the operation.

Finally, a modeling system usually provides a *user interface*. Interfaces can range from arcane numerical input text editors (used with a scripting language) to sophisticated direct manipulation constraint-solving systems with multi-dimensional input devices.

It is this user-interface portion of modeling systems which we wish to address. The end result of a modeling system is a geometric shape of some kind, which can usually be approximated, for visualization purposes, as a polygonal surface. We shall refer to this polygonal surface as the *surface representation*. Furthermore, we will speak in general terms about the underlying model of the surface representation whose characteristics will remain unspecified. While the work we have done deals entirely with polygons, we discuss schemes for extending the concepts we develop to more abstract models in Section 7.

In the process of modeling a geometric shape, a user typically performs operations on the underlying model of the system. At each such stage, a portion of the underlying model undergoes alteration, and a new polygonal surface representation of the underlying model is generated. We shall call that portion of the surface representation of the underlying model which is operated on at each stage the *subsurface*.

Unfortunately, most modeling systems do not provide a mechanism for specifying the precise size and shape of the subsurface. It is this problem that we hope to remedy.

Towards this end, we offer a new modeling system. The system uses 3D WYSIWYG painting to allow for selection of a subsurface to be manipulated. Geometric operations are performed only on the subsurface—the painted region of the surface representation of the model. This relation between paint and geometric operations constitutes the basis of our system. This correspondence yields several positive bi-products including adaptive refinement of the subsurface using texture maps created during WYSIWYG painting, and the ability to create new modeling paradigms by associating painting properties with geometric ones.

## 1.1 Previous Work

Recent geometric modeling systems vary significantly in their approaches towards shape design. For example, Fowler [Fow92] provides tools for sophisticated direct manipulation geometric operations on tensor product surfaces. Users modify high level geometric properties of a surface, such as normals, and a sophisticated constraint solving system alters the surface to reflect those changes. Fowler's success lies in hiding control points from the user. However, Fowler does not provide a means by which a subsurface may be manipulated. Hsu *et al.* [HHK92] and Borrell and Bechmann [BB91] both describe methods of modifying spline-based objects based on targets, employing the pseudo-inverse method to compute solutions. Hsu's interface lets a user drag a point of a surface and use a constraint solver to determine changes to an FFD lattice that will deform the surface accordingly. Hsu's alteration of the FFD lattice has its advantages. For one thing, the surface that is being deformed can be represented at any resolution, since the FFD lattice deforms the space the surface sits in. Unfortunately, while Hsu's system does allow a user to drag a point of a surface to an arbitrary location, it does not provide a mechanism for dragging a subsurface. Welch and Witkin [WW92] advance an approach based on eliminating control point interfaces in favor of an interface based on constraints on B-spline surfaces. Among the constraints they offer is the manipulation of a curve on a surface. While the authors approach a subsurface manipulation scheme, arbitrary areas on the B-spline surface still cannot be constrained.

## 1.2 Overview

While the aforementioned systems feature disparate paradigms and interaction techniques, they share one goal: the end result is a shape. Typically, this shape is or can be represented as a collection of polygons—a polygonal surface—at some stage in the modeling process. At each such stage, some or all of the underlying model is modified and a new surface representation is generated which reflects these changes. Unfortunately, none of the systems described in Section 1.1 allow for explicit control of the subsurface that is to be modified at each stage. A polygonal modeler might allow the user to interactively edit a vertex or an aggregate of vertices, but not an arbitrarily shaped portion of the polygonal surface. This deficiency can create problems. When we move a control point of a spline, or a vertex of a polyhedron, or elastically stretch a portion of a physically based object, it is difficult to predict the final outcome of the operation, in part because the user has little control over or knowledge of the exact region the operation will affect.

And, even when we do know the precise nature of the subsurface, it is often not enough. Indeed, knowledge of the subsurface is available in many systems. For instance, in CSG systems, the user knows the shape of the region that is being affected. Likewise, when a user moves a control point of a spline surface, a precise mathematically defined region is affected. However, knowledge of and control of the affected region are different: while the user may know the exact region which will undergo alteration, there may not be an interface for controlling it. For example, dragging a point of a surface in Hsu's FFD modeling system produces results that depend on the resolution of the underlying FFD lattice, which is invisible to the user. Even if the user is aware of what region of the surface will be affected, there is no means of controlling the shape of the region that will be affected.

We provide an interface to remedy this problem. We utilize 3D WYSIWYG painting as a metaphor for selection of a portion of a surface; the painted portion of the surface is selected. This interaction technique offers many benefits. For example, the technique is easily implemented and useful in a wide array of applications. Furthermore, in the process of WYSIWYG painting we create 2D texture maps, representing the image painted on a 3D surface. Every point on the 3D surfaces we use corresponds to a pixel in a 2D texture map, and vice versa. Clever use of image processing routines on these texture maps enables us to refine our 3D surface where necessary. We use the image processing routines to reveal where detail needs to be added.

Our prototype application is one in which the user first paints on a geometric shape to select the subsurface that is to be manipulated. Then, a geometric operation of some sort is chosen, and image processing routines, performed on the texture maps, refine the selected subsurface. Finally, the subsurface is operated on accordingly. As we shall see, various painting attributes can be used to select different geometric modeling attributes, creating novel modeling paradigms. That is, by establishing a correspondence between 2D texture maps and 3D geometry, we have engineered a new type of model, distinct from the traditional parametric model. The new model has its advantages (e.g., refinement through image processing routines) and its drawbacks (e.g., finite resolution limited by texture maps). Moreover, while this paradigm might in the future be applied to non-polygonal surfaces, we have not yet demonstrated that this is possible.

We illustrate the usefulness of our interaction technique through two applications—a polygonal modeler and a trimming curve generator. Both run under the same application shell and incorporate many of the same implementation techniques. The polygonal modeler allows a user to paint an image on a polygonal model and then to perform deformations on the painted portion of the 3D shape. Image processing techniques are applied to the texture maps generated by the 3D WYSIWYG painting to add detail to the polygonal model where necessary. We also illustrate the idea of mapping geometric properties to painting properties by establishing a correspondence between the intensity of a point of a painted surface and the extent to which that point is moved during a deformation. Our second application, a trimming curve generator, allows a user to paint on a polygonal representation of a patch. The application uses this information to create a 2D domain space curve at the boundary of the painted region. This curve is then converted to a trimming curve and used on the original patch object.

## 2 WYSIWYG Painting

Both of these applications make extensive use of the 3D WYSIWYG painting technique developed by Hanrahan and Haberli [HH90] as a direct manipulation interface for creating texture maps on 3D shapes. This technique enables a user to paint with a 2D input device directly on a screen depicting a 3D shape. The painting resembles a 2D painting program, where the user's brush strokes are converted into painted regions in 2D texture maps. The technique typically uses a graphics workstation's hardware texture mapping to apply the texture maps to the 3D shape. Brushes are 2D pixmaps overlayed onto the screen, with each brush pixel having a unique ray associated with it. Essentially, the technique approximates painting onto a 3D surface by ray tracing the polygonal representation of a 3D shape. By casting a ray for every brush pixel, 2D brushes mimic the effect of 3D painting.

Indeed, WYSIWYG painting succeeds largely because of its similarity to 2D painting. In exploiting the 2D metaphor, we use a 2D "screen-space brush" (as describe in [HH90]) to do our painting. The notion of 3D paint brush sizes, styles, shapes and colors is rooted in analogs from 2D painting. New brushes are pixmaps and are thus easily created. Moreover, users may "paint" and "unpaint" a surface until a desirable result is obtained. Brush intensity is a parameter in the system. An eraser brush is just a zero intensity pixmap. This is advantageous because we use the texture maps created by WYSIWYG painting to determine where the original 3D shape will be refined. Hence, because only the final painted image is used, unnecessary refinement does not occur.

In order to simulate painting directly on to the screen with the mouse, we must know for each pixel of our brush what portion of a 3D object is in its view, since ultimately each pixel on the screen corresponds to some point on a 3D surface, and thus some point in a 2D texture map. To accomplish this in real-time, we use a speedup mechanism. This speedup involves the addition of two windows to the application shell. The first is identical in size to the window which displays the 3D object. This window contains the same 3D shape as the original window, but each polygon of the surface representation (in this case we use triangles) in the speedup window is colored according to a unique identification number. The second window contains a texture map image which is scanned periodically and then wrapped around the 3D object using the graphics workstation's hardware texture mapping. Figure 19 shows the application shell with these windows.

When the user paints on an object in the original window, the coordinates of each pixel of the paint brush are recorded. Then, the corresponding pixel's color values in the speedup window are recorded. Then, these color values are converted into triangle identification numbers. Now we know automatically, for each pixel in our brush, which triangle in the 3D shape, if any, is visible and should therefore be painted. (In essence we are using the $Z$-buffer capabilities of the frame buffer to do some work for us). Figure 1 illustrates this. If a brush pixel lies within a triangle, the ray corresponding to that pixel is traced and intersected with the triangle in its path. This yields a point of the triangle. This point has associated with it the coordinates ($u$ and $v$) of a point in the texture map. These coordinates are computed by interpolating the $u$ and $v$ values at the vertices of the triangle. So, for every pixel the user paints on the 3D shape, the program can generate the corresponding $uv$-point in a texture map in real time (see Figure 2).

As the user paints on the 3D shape, every triplet of adjoining pixels in which all three pixels hit a triangle, is converted to a triplet of $uv$-points. The new points define a triangle in 2D. This triangle is then drawn in the texture map window (see Figure 3). After painting, the texture map is rescanned and the 3D shape reflects the changes painted on it. However, scanning occurs only after the user changes the camera position. In the meantime, we paint directly onto the 2D screen representing the 3D shape. This is due to the fact that scanning the texture map in is relatively slow.

Despite, the lack of speed due to rescanning the texture maps, the technique proves effective. Our 3D WYSIWYG painting framework is enhanced by a user interface that allows for varying brush size, styles and intensities.

## 3 Image Processing

Texture maps are traditionally used to add detail to objects without adding geometry. There is no reason we cannot reverse this process. In other words, why not use texture maps to determine where to add geometric complexity? If the color at each pixel in the texture map corresponds to some geometric property in that pixel's corresponding point on the 3D shape, we will assume a change in intensity in the texture map indicates a change in the original shape. Essentially, we are treating a texture map as a *detail map*. By maintaining not only a mapping from the object in $\Re^3$ to the texture map in $\Re^2$ (we need this to do the WYSIWYG painting), but also the inverse map, we can add detail to our 3D object in places determined by the texture map.

We want changes in the texture map to reflect changes in the painted object. One method of determining where change is occurring is to compute the gradient of the texture map. Figure 4 illustrates a painted region in the

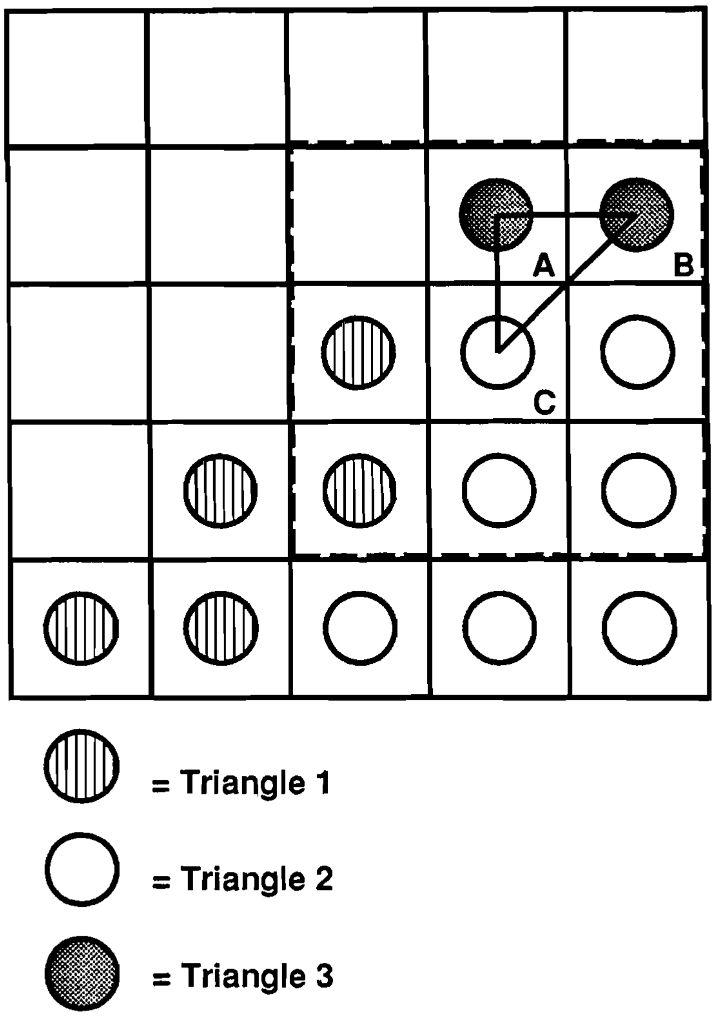

Figure 1: Enlargement of pixels of a paint brush over speedup window. Squares represent pixels of the $Z$-buffered speedup window, the dashed-line square is the brush, the circles in the pixels represent the encoded triangle colors of the pixel, and A, B and C denote pixels whose corresponding $uv$-points will be converted into a polygon in a texture map. By scanning the frame buffer, and reading in color values, we can predetermine, for each pixel, which triangle, if any is visible.

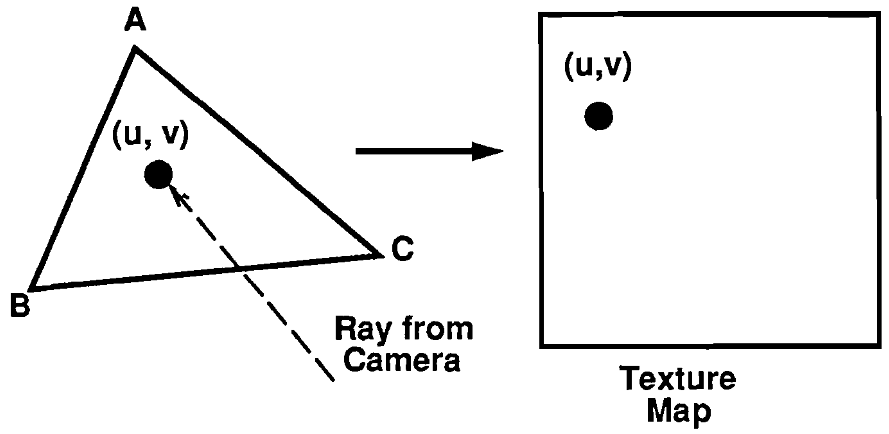

Figure 2: Left: The intersection point of a ray and a triangle produces a *uv*-point. This point is obtained through interpolation of the *uv*-values at the vertices of the triangle. Right: The *uv*-point in the texture map.

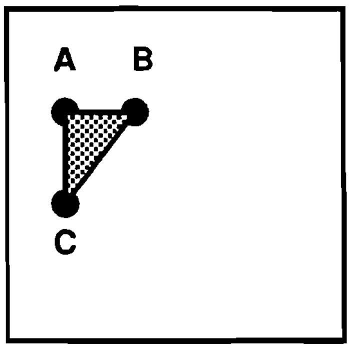

Figure 3: Triplets of *uv*-points from the intersected triangles are converted to a region in the texture map. The points A, B and C shown in the speedup window are ultimately converted to a triangle in the texture map. Subsequently, the texture map is rescanned and applied to the object. The end result is the illusion that the user is actually painting on the 3D shape.

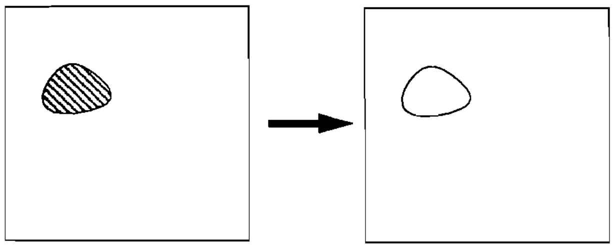

Figure 4: Left: The painted image in a texture map. Right: The texture map after gradient computation.

texture map, and the subsequent image after gradient calculation. Equation 1 approximates the magnitude of the gradient where $t(x, y)$ denotes the intensity of the texture map at pixel $(x, y)$:

$$\begin{aligned} D(t(x,y)) &= \|t(x,y) - t(x+1,y+1)\| + \\ &\quad \|t(x+1,y) - t(x,y+1)\| \end{aligned} \tag{1}$$

The calculation of the gradient yields a new map, which we will refer to as our detail map (see the right side of Figure 4). What we do with this map depends on the application.

One useful operation, used by both of our applications, is the calculation of all of the contours of the detail map. This operation is useful because the 3D analog of a 2D contour is a 3D curve which bounds the subsurface selected by the user. With the 3D curve in hand, we can, for instance, retessellate the surface along the curve bounding the selected subsurface. This is what we do in both our polygonal modeler, and the trimming curve generator.

By setting a threshold value, $T_{val}$, for the gradient, as described by Equation 2, we obtain a binary image of pixels on which we can use contouring algorithms.

$$Thresh(D(t(x,y))) = \begin{cases} 1 & \text{if } \nabla(t(x,y)) > T_{val} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Many sophisticated contouring algorithms exist [KWT88], but we are dealing with simple texture images, which do not require powerful algorithms. A contouring algorithm yields an ordering of points defining a curve. In our case, we generate a contour of pixel coordinates whose pixels bound a painted

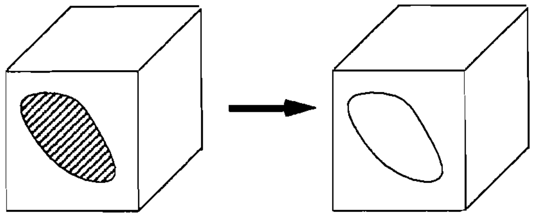

Figure 5: The original painted region is converted to a bounding curve of the region, using our image processing techniques.

region. The 2D contour is converted to a 3D contour (using the $\Re^2$ to $\Re^3$ map), which represents a bounding curve on the 3D shape of the area of interest, as illustrated in Figure 5.

There are, however, many other schemes for adding detail to a 3D shape. Most involve the gradient operation. However, not all involve computing contours. For instance, we can add vertices to a 3D polyhedron, or rows of control points to a spline surface, wherever the corresponding texture map gradient exceeds some threshold value. The choice of which image processing routines ought to be used and in what manner they ought to be utilized are dictated by an individual application's needs.

## 4 Polygonal Modeling

Our first application allows a user to paint an image on a 3D polygonal shape and then translate the region that is painted on. This application features two stages of operation. The first stage is a *WYSIWYG paint mode* in which the user is free to move the camera about and paint on an object with several different brush shapes and styles.

Once the user is satisfied with the image on the shape, the application switches to *deform mode*. We first refine our polygonal object using the techniques in Section 3 to create a 3D curve on the surface of the object, based on the previously described contouring scheme.

The resulting curve is actually an ordered collection of 3D points on the surface of the 3D shape. Figure 6 depicts a cross section of this shape, with the points of the generated curve and the corresponding surface normal at each point. For each point on the shape we create two corresponding points:

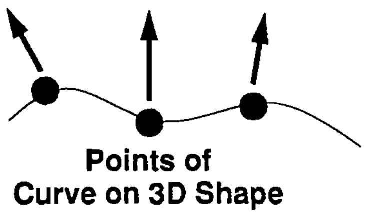

Figure 6: The points of the generated 3D curve and the corresponding normal at each point.

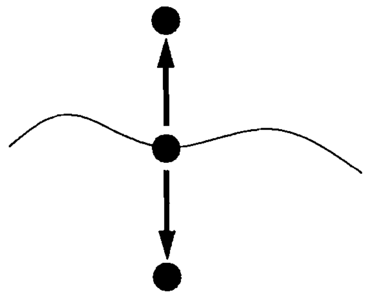

Figure 7: Two points are added for each curve point by traversing the surface normal.

one below the surface of the object and one above (see Figure 7). This is accomplished by traversing the normal of the 3D surface at each point on the curve. Attaching the ordered points together as illustrated in Figure 8 produces a polygonal object.

We shall call this polygonal object a *cutter* (see Figure 9). The cutter object is so called because it is used much like a cookie cutter. We use the cutter to cut the 3D object (the dough), and then return the cut dough to its place. This operation is accomplished using standard CSG routines which have been slightly altered. The operation is similar to a boolean subtraction operation in that the cutter object is subtracted from the 3D shape. In our case, what is cut out is then added back in. The resulting tessellation has edges and vertices along the boundary of the painted region on the shape. These vertices will eventually be translated, producing an extrusion in the shape of the boundary of the painted region. For this reason, this polygonal operation yields the correct tessellation of the object that enables us to extrude the shape we painted. The upper left hand portion of Figure 19 shows

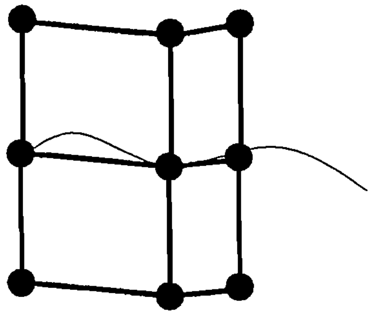

Figure 8: Connecting the curve points yields a polygonal object.

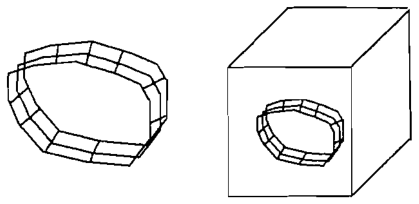

Figure 9: The cutter object and its placement on the 3D shape.

the speedup window with a retessellation of the object in accordance with the painted region.

Next, each vertex on the 3D surface is assigned a weight according to its corresponding texture map color [Bry92]. By *weight*, we mean a numerical value between 0 and 1 which is used as multiplication factor in applying deformations to the vertex. In this example, vertices whose corresponding texture map color is white are selected and thus receive a weight of 1. The rest are not selected and thus carry a weight of 0.

At this stage, the application behaves much like a conventional polygonal modeler. When the user wishes to deform the 3D shape, the application applies the desired deformation (in this case translation) to each vertex, multiplying by each vertex's weight. Hence, vertices with weight 0, those whose corresponding texture map pixels remain unpainted, will not be deformed. Those vertices which *are* painted will feel the effects of the deformation. The user may interactively translate the selected vertices. The result of this is an extrusion of the image painted on the polygonal object.

Figure 12 depicts a "squiggly" painted on a cylinder. Figure 13 shows the extruded squiggly after refinement of the cylinder. The original cylinder had approximately 20 faces. The refined version has approximately 700. There was not enough detail in the original cylinder to extrude the squiggly, without prior refinement. Similarly, Figure 14 depicts a ring painted on a cube, while Figure 15 shows the ring after the translation deformation. Figure 16 illustrates the differences between a solid colored brush and a brush with tapering intensity. In this case, the color of the paint is used as a weight for the extrusion—the whiter the color of a vertex, the more it gets extruded. This is an example of associating paint properties with geometric ones. We can think of many more such paradigms. For instance, we could associate different paint colors with different translational directions. Under this paradigm, blue could be mapped to the $X$ direction, green to the $Y$ direction and red to the $Z$ direction. Users could paint with any color paint brush, and vertex weights would be assigned for each translational direction according to colors in the texture map.

Similarly, this application need not be restricted to translation operations. Indeed, we could just as easily implement scales, twists, or any other polygonal deformations using the vertex weights [Bar84] (we could even map different paint colors to different types of deformations).

Finally, we should note a way that our refinement scheme might be improved. A simpler form of the application omits the contouring and cutter objects and simply adds vertices where the gradient of the texture map is high. This succeeds, provided the original object was tessellated sufficiently finely. Perhaps more sophisticated schemes would yield better results.

## 5 Trimming Curves

A similar application, using many of the techniques described in Section 4, generates trimming curves for patches. In this application, an image is painted onto a patch by the user. Once the user is satisfied with the image on the patch, the program calculates a contour curve for the area on the patch. By *curve* we refer to an ordered collection of points. For modeling, the points may be converted to a $C^0$, $C^1$ or $C^2$ continuous curve. This curve is then use as a trimming curve. In Figure 17, we see an image painted onto the patch. Figure 18 illustrates the trimmed patch. Naturally, the painted area may act as either a cutout or a mask. Smoothing of the trimming curve may be performed as well.

## 6 Limitations

Despite the strides we have made in developing a practical, useful interface for selecting subsurfaces, the techniques we describe suffer from many drawbacks—some practical and some theoretical. The most glaring deficiencies in the applications stem from the polygonal nature of our models. After performing the cutter operations on a polygonal object, the newly retessellated object often contains many ragged edges which are due to the fact that our polygonal cutter object is not necessarily smooth. The upper left hand portion of Figure 19 illustrates the new tessellation and its rough character. Perhaps a triangle smoothing algorithm like Turk's [Tur92] applied to the a retessellated object would remedy the situation.

Similarly, in implementing tapered brushes, we found the cutter method was less effective than other techniques in producing a smooth retessellation of the polygonal object. We employed several strategies for retessellation of a surface painted on with a tapered brush. Most suffered from uneven edges as well, although different techniques met with varying degrees of success. In one strategy we omitted the contouring and cutter steps of the retessellation altogether. Instead, we mapped every texture map pixel whose gradient value surpassed a certain threshold value, back onto the original surface, just as we did with the contour points. Then every such point was used to split the triangle in which it lay into three new triangles. Unfortunately, performing this task for every pixel whose gradient exceeded the threshold resulted in numerical errors due to degenerate triangles. When we restricted the operation to some random subset of potential pixels, the technique fared better. We then formulated another technique using the standard contouring and cutter methods to retessellate the object. Once the standard retessellation was performed, every triangle inside the painted region was further retessellated (the CSG cutter operations mark every triangle as being inside or outside the cutter object). This worked well, but again suffered from ragged edges. We can devise many other schemes to retessellate polygonal shapes for tapered brush use. While we have met with some success, we believe there is still room for improvement through research.

Finally, we wish to point out a problem that is theoretical as well as practical. After we deform a polygonal object, certain polygons may expand, while others may shrink. When the user paints on these polygons to select a new subsurface, the ratio of the size of a polygon's area to its corresponding area in the texture map may have changed. This causes resolution problems. As a solution, we could continually increase the size of the texture map after every deformation step. Obviously this has a practical limit, but it might work in some cases. Indeed, we attempted this solution with reasonable results. We also tried to couple it with *relaxation techniques* applied to the texture map. Under this system, we treat the polygons in the texture map as a system of springs, where each edge of each polygon behaves as a spring. The natural rest length of every edge is proportional to its corresponding edge length in the 3D polygonal object. We let the system go, and allow the simulation to run until the system is reasonably static. We found this solution to improve the situation, but it was not a complete solution.

Our relaxation technique is rather simple, so perhaps a more sophisticated algorithm could yield more useful results [BVI91]. The correct solution of course may be to use manifold structures to cover our polygonal object with a collection of texture maps, one per chart, and associated correspondence functions [HSBB93]. Ideally, we could generate a new set of charts every time the object is deformed, allowing painting in virtually limitless detail. Manifold structures might also help address the "seam problem" encountered when using 3D WYSIWYG painting with a single texture map per polygonal object. When the user paints onto an object at a polygon or polygons whose *uv*-values straddle the border of the texture map, we encounter an ambiguity when painting a polygon in the texture map (see Figure 10). Given points $A$, $B$ and $C$ as *uv*-points forming a triangle, do we paint the triangle in the middle picture or the one on the right? As it currently stands, the program opts for the solution shown on the right.

## 7 Future Work

We have developed an interaction technique that can be used in a variety of modeling paradigms. We feel WYSIWYG painting is a very natural, intuitive means by which users can select portions of shapes with a great degree of accuracy. The technique may be used with many underlying models that can take the form of a polygonal representation. For instance, we can envision a

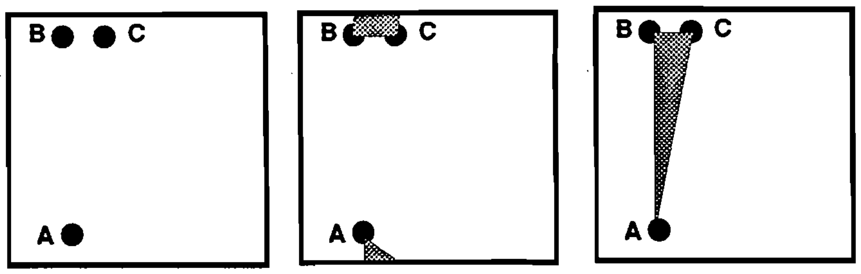

Figure 10: An ambiguity arises when painting across the seam of a texture map. When painting into the texture maps, what is the polygon defined by the points $A$, $B$ and $C$? Is it the polygon in the middle or the triangle on the right?

spline-based version of our polygonal modeler, in which control points instead of vertices are added to a spline surface. Just as in the polygonal modeler, control points are assigned weights according to corresponding color values in a texture map. They are then manipulated interactively. A clever use of hierarchical B-splines [FB88] might avoid the explosion of control points often encountered when adding detail to patches.

It might also be possible to use our interaction technique to provide a user-friendly interface for generating EFFD's [Coq90]. EFFD's are a useful modeling tool for creating bumps on surfaces. Unfortunately creating the lattices necessary for EFFD modeling requires expert knowledge. By expanding on our cutter notion, we can generate EFFD's by painting. After painting has occurred, contouring of the texture maps is performed, and cutter objects are generated. By scaling the cutter objects (about the center point of the object) up and down slightly, we can form a series of cutter objects in the shape of the contour of the painted region. We join the scaled cutters together connecting corresponding vertices. Instead, we connect corresponding points. In other words, each vertex of the cutter is connected to its scaled-up and scaled-down analogs. We scale by traversing the normal at each point. The newly connected structure is the EFFD lattice, with the vertices of the cutter object acting as the lattice's control points. Figure 11 illustrates what an EFFD generated by painting might look like. Finally, the user can interactively move the lattice control points, while the original texture map color values act as weights for the control points.

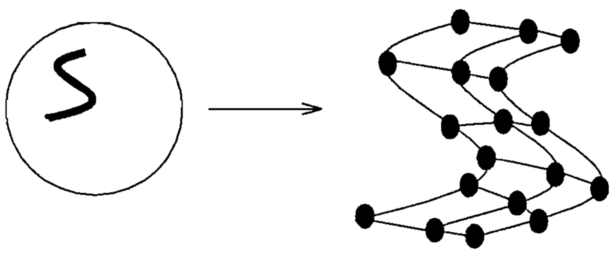

Figure 11: A painted S on a sphere is converted into its corresponding EFFD lattice.

Other applications that might fit well into our scheme involve specifying constraints and snapping using WYSIWYG painting. Joint specification by painting axes of freedom on objects might be a possibility, as might snapping shapes to each other based on the color of their surfaces. We can also foresee an updated version of Williams' 3D painting program [Wil90] in which each paint color connotes a different direction of deformation.

## 8 Acknowledgments

I would like to thank John Hughes, Nate Huang, Phillip Hubbard, Dan Robbins, Andy van Dam and the rest of the Brown Computer Graphics Group for all of their invaluable assistance and support.

## References

[Bar84] A.H. Barr. Global and Local Deformations of Solid Primitives. *Computer Graphics*, 18:21–30, July 1984. Proceedings of SIGGRAPH '84.

[BB91] P. Borrell and D. Bechmann. Deformation of N-Dimensional Objects. *International Journal of Computational Geomatry & Applications*, 1(4):427–453, 1991.

[Bry92] S. Bryson. Paradigms for the Shaping of Surfaces in a Virtual Environment. *Proceedings of the Twenty-Fifth Hawaii International Convference on System Sciences*, 2:649–658, 1992.

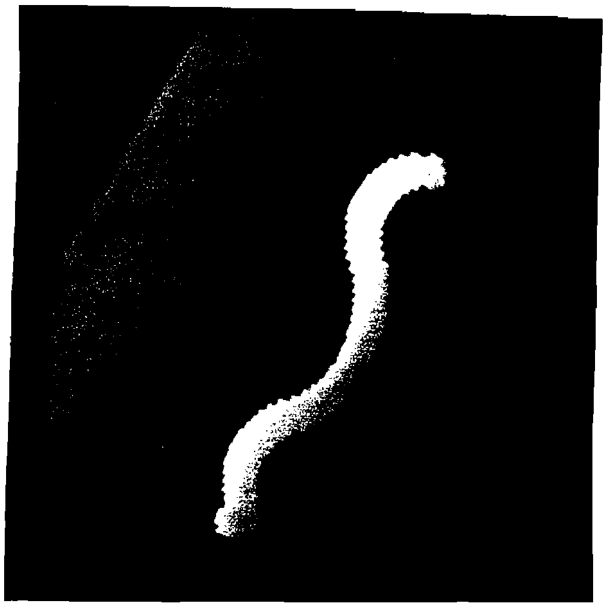
Figure 12: A cylinder with a squiggly painted on it.

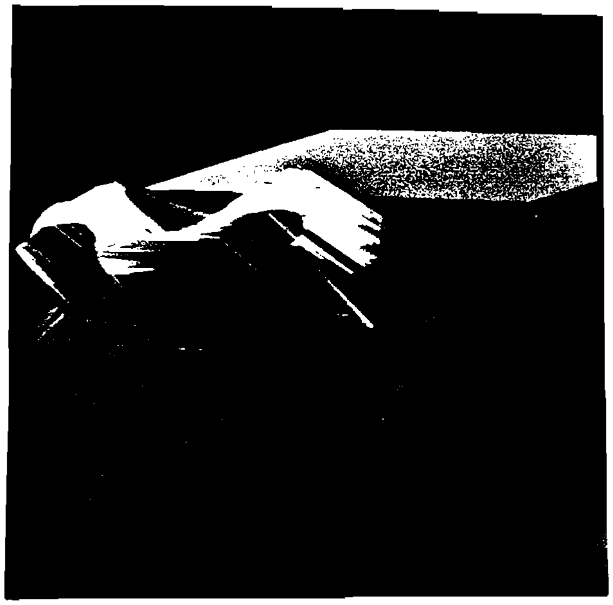
Figure 13: A cylinder with the squiggly extruded.

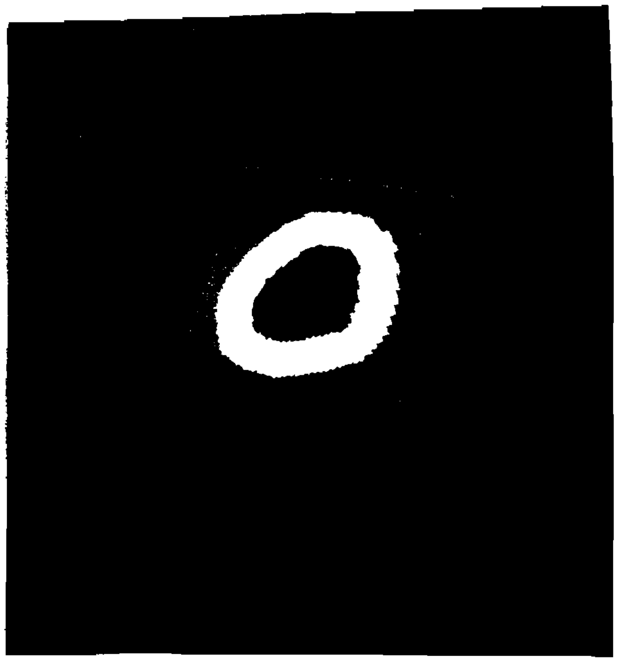
Figure 14: A cube with a ring painted on it.

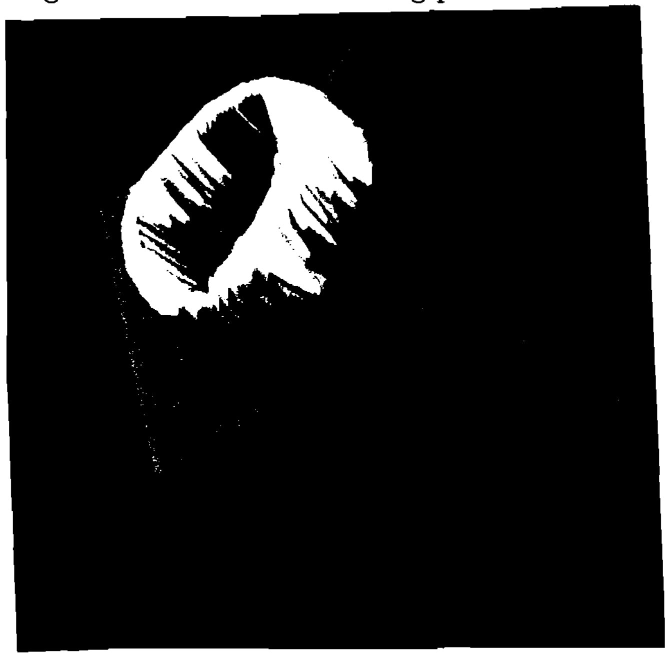

Figure 15: A cube with the ring extruded.

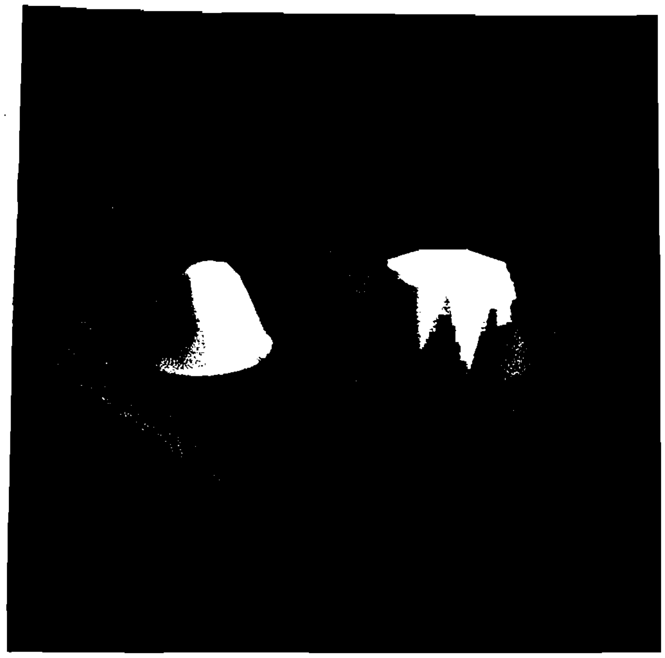

Figure 16: Two different brushes yield different results when extruded from the sheet.

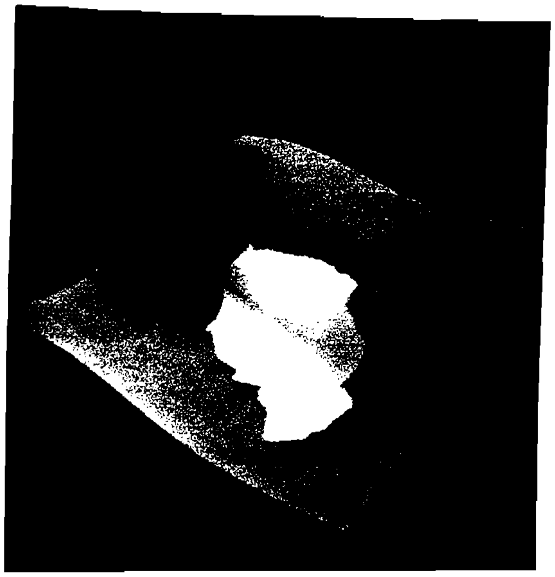

Figure 17: A patch with a trimming region painted on it.

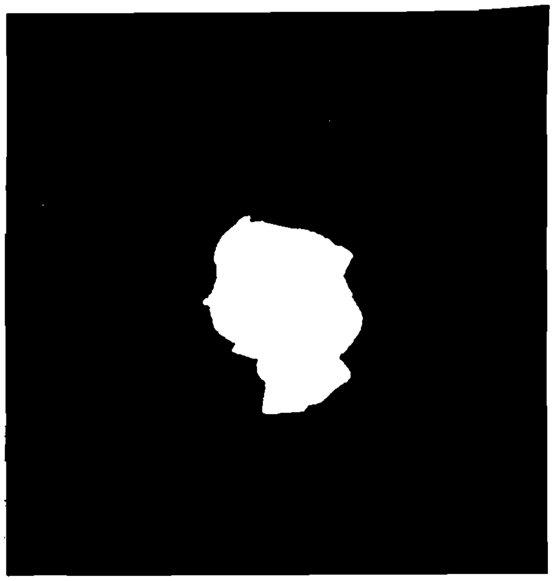

Figure 18: The newly trimmed patch.

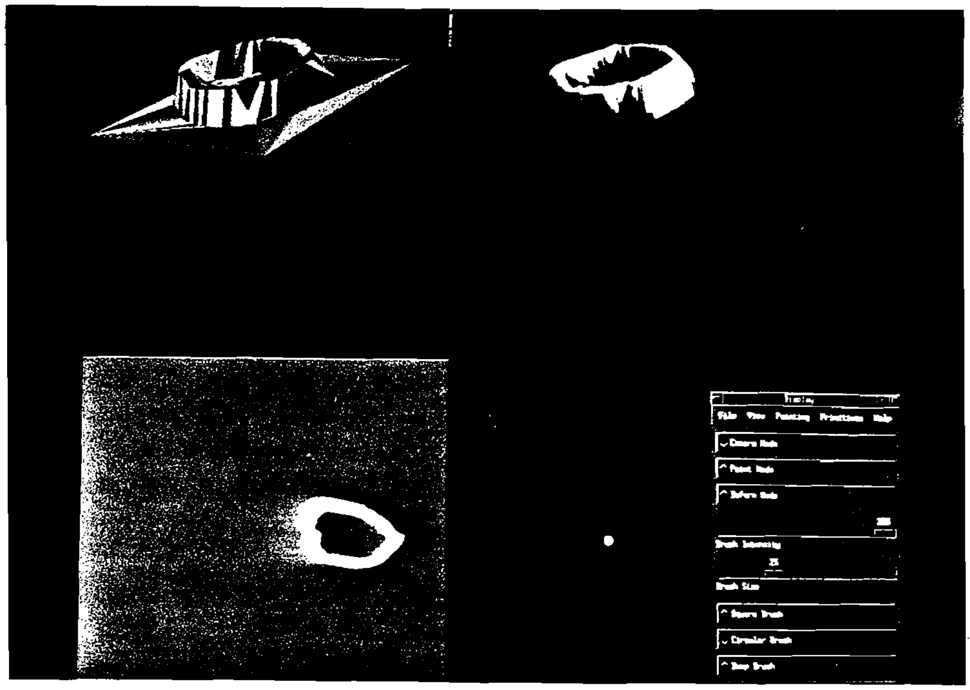

Figure 19: The upper left hand window shows the polygonal object shaded according to a face identification number. The upper right hand window is the main window. The user interface is to the lower right, and the texture map which is scanned and wraps around the cube is shown in the lower left.

[BVI91] C. Bennis, J-M Vezien, and G. Iglesias. Piecewise Surface Flattening for Non-Distorted Texture Mapping. *Computer Graphics*, 25(4):237–246, July 1991. Proceedings of SIGGRAPH '91.

[Coq90] S. Coquillart. Extended Free-Form Deformation: A Sculpting Tool for 3D Geometric Modeling. *Computer Graphics*, 24(4):187–196, 1990. Proceedings of SIGGRAPH '90.

[FB88] D. Forsey and R. Bartels. Hierarchical B-Spline Refinement. *Computer Graphics*, 22(4):205–212, August 1988. Proceedings of SIGGRAPH '88.

[Fow92] B. Fowler. Geometric Manipulation of Tensor Product Surfaces. *Computer Graphics*, 25(2):101–108, March 1992. Proceedings of 1992 Symposium on Interactive 3D Graphics.

[HH90] P. Hanrahan and P. Haberli. Direct WYSIWYG Painting and Texturing on 3D Shapes. *Computer Graphics*, 24(4):215–223, 1990. Proceedings of SIGGRAPH '90.

[HHK92] W. M. Hsu, J. Hughes, and H. Kaufman. Direct Manipulation of Free-Form Deformations. *Computer Graphics*, 26(2):177–184, July 1992. Proceedings of SIGGRAPH '92.

[HSBB93] J. Hughes, J. Snyder, R. Barzel, and A. Barr. Using Manifolds in Computer Graphics. *Computer Graphics*, 1993. Submitted to SIGGRAPH '93.

[KWT88] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active Contour Models. *International Journal of Computer Vision*, pages 321–331, 1988.

[Tur92] G. Turk. Direct WYSIWYG Painting and Texturing on 3D Shapes. *Computer Graphics*, 26(2):55–64, 1992. Proceedings of SIGGRAPH '92.

[Wil90] L. Williams. 3D Paint. *Computer Graphics*, 24(2):225–233, March 1990. Proceedings of 1990 Symposium on Interactive 3D Techniques.

[WW92] W. Welch and A. Witkin. Variational Surface Modeling. *Computer Graphics*, 26(2):157–165, July 1992. Proceedings of SIGGRAPH '92.