Sculpting: An Interactive Volumetric Modeling Technique*

Tinsley A. Galyean
The Media Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

John F. Hughes
Department of Computer Science
Box 1910
Brown University
Providence, RI 02906

ABSTRACT

We present a new interactive modeling technique based on the notion of sculpting a solid material. A sculpting tool is controlled by a 3D input device and the material is represented by voxel data; the tool acts by modifying the values in the voxel array, much as a "paint" program's "paintbrush" modifies bitmap values. The voxel data is converted to a polygonal surface using a "marching-cubes" algorithm; since the modifications to the voxel data are local, we accelerate this computation by an incremental algorithm and accelerate the display by using a special data structure for determining which polygons must be redrawn in a particular screen region. We provide a variety of tools: one that cuts away material, one that adds material, a "sandpaper" tool, a "heat gun," etc. The technique provides an intuitive direct interaction, as if the user were working with clay or wax. The models created are free-form and may have complex topology; however, they are not precise, so the technique is appropriate for modeling a boulder or a tooth but not for modeling a crankshaft.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling; Curve, surface, solid, and object representations; I.3.3 [Computer Graphics]: Picture/Image Generation; Display algorithms; I.3.6 [Computer Graphics]: Methodologies and Techniques; Interaction techniques.

Additional Keywords: Sculpting, volumetric data, 3D interaction, antialiasing, free-form modeling.

1 INTRODUCTION

We present a new modeling technique for computer graphics based on the notion of sculpting a solid material with a tool. This technique is derived from traditional 2D paint systems, from Blinn's blobby objects [1], and from the soft objects of Wyvill et al. [17]. The term "sculpting" has been used by others: Naylor [9] uses it to describe a polyhedral CSG system that is capable of interactive performance when implemented on a Pixel Machine; Coquillart [3] uses it to describe her interface to free-form deformations [12], that edit the geometry of an object but not its topology; Pentland et al. [10] use it to describe the altering of shapes by modal forces. In related work, Williams 3D paint system [15] lets the user edit the z-depths of points on an object that is a union of two topological disks, using color as a proxy for height. Bloomenthal et al. [2] describe an object by forming a geometric skeleton, associating a potential function with it, and drawing isosurfaces of the potential function; editing the skeleton then modifies the surface. We prefer to avoid the fixed structure of these last four systems and the potential proliferation of polygons of the first, and aim instead for a system that gives the user control of both the geometry and topology of an object and at the same time provides an extremely intuitive interface. Our sculpting may appear similar to work of Van Hook on milling machines [6], but his system only removes material from an object (i.e., no additive tools are allowed), and retains an *image* of the object but not the structure of the object itself. The resulting object cannot be viewed from any direction other than the one in which it was constructed.

Our fundamental notion is to describe the shape of a piece of clay by its *characteristic function*, whose value is 1.0 at any point in space where there is clay and 0.0 elsewhere.[1] Modifying the shape of the clay is therefore equivalent to modifying this function. The same idea applies in a traditional 2D paint system: we think of the canvas as the cartesian plane and assign values to the pixels of the canvas; certain values indicate the presence of ink and others indicate its absence. Painting is done in such systems by moving a brush across the canvas, and data associated with the brush edits the data in the pixmap; for example, moving the tip of a pen across the canvas changes the values of all pixels underneath it to the current color. Our system modifies the values of volumetric data by moving a 3D tool through space, in exact analogy to the brush. Our standard tool is the opposite of the pen in a paint system, however, in that we start with a block of material and remove it bit-by-bit, and hence we refer to the process as *sculpting*. Just as traditional paint systems offer many ways to apply and remove paint, we provide several different tools with which to edit the volumetric data, including ones that add material, smooth a surface, or melt away material as a heat gun melts styrofoam. Many users are familiar with paint programs, and therefore readily accept this variety of tools.

The models created by our technique are free-form and often lack fine detail, but they can have complex topology. They are *not* precise geometric models of the sort traditionally generated by CAD systems. However the technique opens the door to modeling that would otherwise be difficult; it is also well-suited to a free-form design process, in which the user starts with no particular goal, and just plays with the material in an intuitive fashion.

While the underlying idea of our technique is simple and attractive, making it work in practice is not trivial. Paint systems have the advantage that they work by modifying the data in a framebuffer or an offscreen copy of the screen canvas; this memory is typically easy to read and write, and the image can be transferred to the screen

*This work was supported in part by grants from IBM, NCR, and Sun Microsystems.

[1]We represent volumetric functions by giving their values at the vertices of a rectangular lattice in 3-space, and call this *voxel data*, or a *voxmap* for short.

extremely quickly with most current hardware. This is partly because we always look at a 2D painting from the same point of view. By contrast, we may wish to view volumetric data from any angle. Also, since we actually want to see the *boundary* between the material and the empty space, this boundary must be computed by some thresholding algorithm. The computation of an isosurface from the volumetric data is $O(n^3)$, using the marching-cubes algorithm [7], for an $n \times n \times n$ data array. Clearly, for interactivity, we must improve the algorithm. Fortunately, sculpting the data modifies it only *locally*, so one need not recompute an entire isosurface. Of course, the isosurface for such data may contain many polygons, and even if we recompute only the local data and replace some polygons with others, we must find a way to redisplay only the local area or the process becomes polygon bound. Even with only local updates, over 50% of the time is spent rendering polygons on an HP835 Turbo SRX.

Simply sampling the characteristic function of a solid will lead to aliasing. We avoid this by using a low-pass filtered version of the characteristic function. This means that certain samples may be neither 1.0 nor 0.0, but rather some intermediate value, indicating transition from material to empty space. To avoid the introduction of aliases, the values written by the tool must also be band-limited, as discussed in Section 3.1.

Before giving a detailed explanation of the technique and the associated algorithms, we make two important remarks:

1. The success of the program is greatly enhanced by the use of 3D interaction devices. We use the Polhemus Isotrack device, and have begun to experiment with the Ascension 'Bird' and a 3D force-feedback joystick with good results [8].

2. User response indicates that this method of modeling has substantial initial appeal and is extremely easy to learn. Although we have not performed any perceptual studies on the system, we have found that many users of the system say one of two things: "Can I come back and use this again later?" or, from the more experienced users, "This is what I *thought* that 3D modeling would be like when I first started learning about computers."

The remainder of this paper describes the modeling technique at two levels: the user's view of the system and the internal implementation. We include throughout ideas for future work. While the system is a full-fledged modeling system, we view it as comparable to early painting programs like MacPaint; the future work is what will make it more like the painting programs of today.

2 THE USER'S VIEW

In calling our modeling technique "sculpting," we hope to connote a very free-form interaction; a sculptor can carve away bits of material, stick on new pieces of clay, change the topology of a sculpture, etc. (A sculptor using physical clay can also squeeze or flex the clay; our system does not yet provide this functionality.)

To present this free-form modeling technique, we provide the user with a cubical "lump of clay" (called the *object*) and a small *tool*. The tool, displayed as a sphere or cube, is directly controlled by a 3D interaction device such as the Polhemus Isotrack. In a separate window, the user has a traditional user interface, consisting of menus for file management and buttons for selecting how the tool should act on the object and for resetting the system. Interaction with this part of the interface is done with a mouse and keyboard. A typical session begins with the default (a cube of clay) or with the selection of a previous sculpture, and continues as the user holds the Polhemus device and sculpts away material. When the user wishes to change the effect of the tool, she uses the mouse to select a new tool type (see Figure 1).

2.1 Types of Tools

The tool, in its simplest form, is analogous to the eraser provided in many paint programs: wherever the tool moves, it cuts away the object. In 3D terms, the tool acts like a milling head or a router, but unlike these, the tool leaves no chips. We call this a *routing tool* or *subtractive tool.*

The analogy between the subtractive tool and a paint program's eraser is of considerable value. Most users are familiar with 2D paint programs, and are used to the notion that the mouse can have different effects. This makes it simple to give the user a variety of sculpting tools and to invent new types of tools. Here are the tools we have implemented:

Additive Tool or *Toothpaste Tube*. This tool leaves a trail of material wherever it moves, much like a tube of toothpaste that is squeezed as it is moved.

Heat Gun. This tool "melts away" material much as a heat gun melts styrofoam. If held in one place for a while, it removes all the material, like the routing tool; if moved quickly past a region, it melts the material there slightly.

Sandpaper. This "smoothing" tool alters the object by wearing away the ridges and filling the valleys. (This is analogous to the low-pass filter brushes available in some sophisticated 2D paint systems.)

Other possible tools include a filleting tool, to smooth the joins between adjacent surfaces, and geometric construction tools, which would allow the user to create a cylindrical tube between two points, or create a torus with a certain center and radii, much the way that painting programs allow one to draw straight lines and circles. We also envision adding tools for deforming the object as a clay model, squeezing or bending it, as described in [13].

We have also implemented a primitive color tool, which assigns a chosen RGB color triple to each vertex of the data array that lies within the current tool region. We currently apply a low-pass filter to these assigned color values to create a smoother appearance. There is much more work to be done in this area.

2.2 Interaction

2.2.1 Low and high resolution modes

To make a good sculpture the user must be allowed to view it from different perspectives and work on the back as well as the front. Furthermore, it is often desirable to rough out the coarse shape of a sculpture first, and work on finer detail afterwards. Making this coarse sculpting efficient requires a larger tool for the initial shaping. Thus we provide both low-resolution (*low-res*) and high-resolution (*hi-res*) modes. In low-res mode, we provide full control of the view and all tool functions, but with coarse tools. In hi-res mode, view control is unavailable, but much greater resolution is provided.

In low-res mode, the object is displayed by applying the marching cubes algorithm to a subsample of the data that represents the object. In hi-res mode we use a $30 \times 30 \times 30$ voxel array; in low-res mode the array is $10 \times 10 \times 10$. The visual effect is that the low-res view of an object is very coarse and shows only its general form. Of course, subsampling is not ideal because of the aliasing implicit in the process: small details may disappear completely. The correct approach is to filter the large voxel array into the smaller array; we will do this in the future, but have not found it to be a significant problem in the current implementation.

In low-res mode, the $10 \times 10 \times 10$ array *might* give rise to an object with as many as 5000 polygons (five polygons from each cube in the array). This would be extremely unusual, and 500 polygons is more likely. On the HP835 Turbo SRX, 5000 polygons can be displayed with a refresh rate of 7 per second, allowing the user to rotate the view of the low-res representation of the object in real time.

The conversion from low-res to hi-res mode requires a substantial computation, since the full marching-cubes algorithm must be

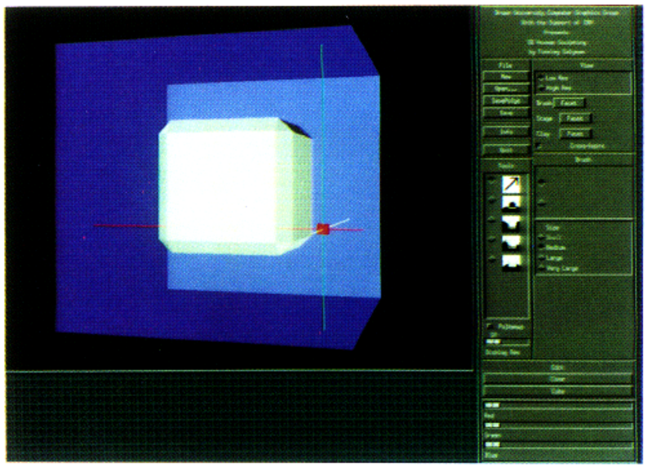
Figure 1: The interface to the system

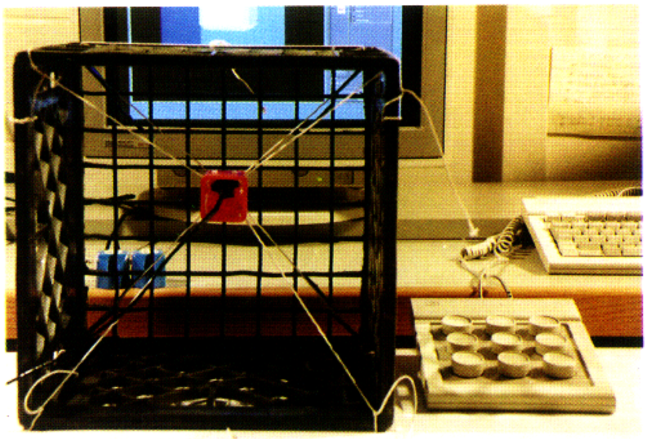
Figure 2: The poor man's force feedback unit

executed and the hashgrid data structure (see Section 3.3) must be recreated; furthermore, all the polygons created in the hi-res model (perhaps several hundred thousand) must be displayed. This conversion causes a slight delay; we are attempting to reduce this, but have not yet found a method to do so. The length of this delay is the prime determinant of the high-res data array size of $30 \times 30 \times 30$.

2.2.2 3D control of the tool

We use the Polhemus Isotrack device to control the position of the sculpting tool. This device provides a constant stream of samples indicating the xyz-position of a pen-shaped pointer; it also provides data describing the orientation of the pointer, which we do not currently use.

There are two problems with using the Polhemus. The data stream is somewhat noisy (this seems not to be as significant a problem with the Ascension 'Bird'), and the natural mapping of the physical space of the Polhemus pointer to the screen space representation of the tool makes the pointer an absolute device instead of a relative one. We have addressed these as follows:

To smooth the Polhemus data, we use a limited predictive tracking process: we use previous samples to predict (by linear interpolation) where we expect the tool to be at the next sample time. We then average the actual sample with the predicted value, assigning a weight of 7 to the actual sample and a weight of 1 to the predicted value. To prevent overshoot, we limit the distance of the predicted position from the previous position; if the predicted position is more than a certain distance from the previous position, the difference vector is truncated. This provides a compromise between the lag of a moving average and the overshoot of predictive tracking.

Using a simple linear transformation to map the coordinates provided by the Polhemus to the modeling coordinates is adequate but not particularly satisfactory. Our initial mapping made the region in front of the monitor (about a $2' \times 2' \times 2'$ cube) correspond to the region occupied by the object (the *sculpting space*). This gives good large-scale control, but sculpting fine details is difficult because of the noise in the Polhemus. We therefore introduce a relative mode, that lets the user move the Polhemus much as one moves a cursor a long distance by repeatedly lifting and replacing a mouse on a mouse pad. (Instead of "lifting," the user presses a button.) The $2'$ cube is then mapped to a small portion of the sculpting space, giving fine control.

Controlling the tool position is not easy. Even though the Polhemus pointer is held in a well-defined region, it is often difficult to correlate the position of the pointer in space with the position of the tool on the screen. To assist in this, we draw a box around the object being sculpted and do front-face culling on the box, so that no matter what the orientation of the sculpture, three walls behind it form a "stage." We then show the tool position by drawing three crosshairs; the intersections of the crosshairs with the stage walls help the user determine the tool's position.

We have also begun to experiment with a force-feedback joystick to help the user position the tool. The joystick generates a small force as the tool approaches the surface, allowing us to "feel" that we are close to it. Using the joystick also helps relieve the user from holding up the Polhemus pointer. We have also implemented a "poor man's force-feedback Polhemus," which alleviates this problem by providing a certain resistance to motion (although there is no real feedback involved) (see Figure 2). The device consists of a Polhemus suspended in a cube by eight elastic cords attached to the corners of the cube.

2.3 Sample Sculptures

Figure 3-Figure 9 show the kinds of sculptures possible with this system; Figure 8 was made using a dial box as input device instead of the Polhemus, which accounts for the orthogonal appearance of the object.

2.4 The Object as a Voxmap

As we have said, the *object* is described by a data array, which we call a *voxmap* in analogy with the 2D notions of bitmaps and pixmaps. (Recall that we think of the values in the array as giving values at the vertices of the lattice, not the centers of cubes in the lattice.) Because the object's characteristic function is discontinuous at its boundary, it has an infinite frequency spectrum, and hence sampling it yields aliases. We therefore store the values of the *low-pass-filtered* (or *band-limited*) characteristic function of the object, which may be between 0 and 1. They are analogous to the grey pixels drawn near a sloping line to reduce the jaggies [4].[2]

We use the values in the voxmap to determine the values of the band-limited characteristic function at intermediate points. To do this precisely, we would have to apply a perfect reconstruction filter; instead, we simply interpolate the values linearly, which amounts to reconstructing with a triangle filter.

[2]We actually use 8-bit integers to represent the function: 0 and 255 correspond to 0.0 and 1.0, respectively.

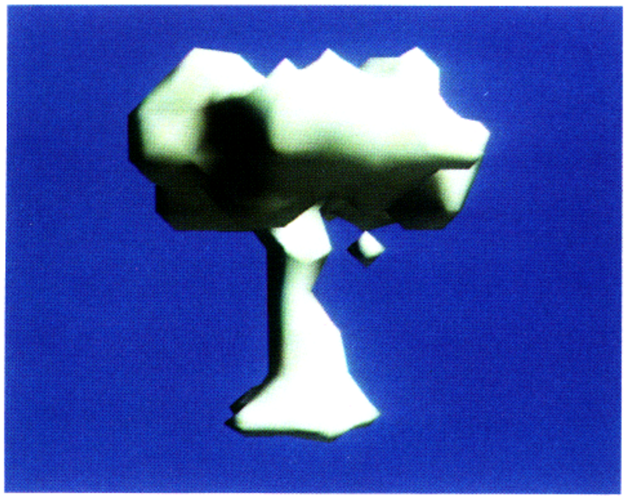

Figure 3: Low-res model of a tree: 1138 polygons

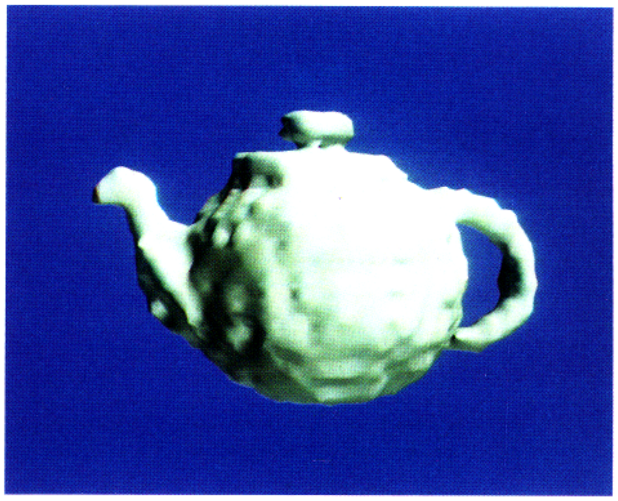

Figure 6: Teapot, chiseled from stone: 9244 polygons

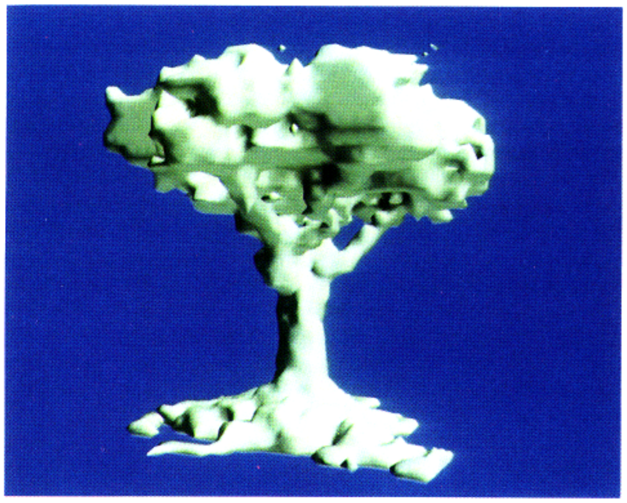

Figure 4: Hi-res model of a tree: 17216 polygons

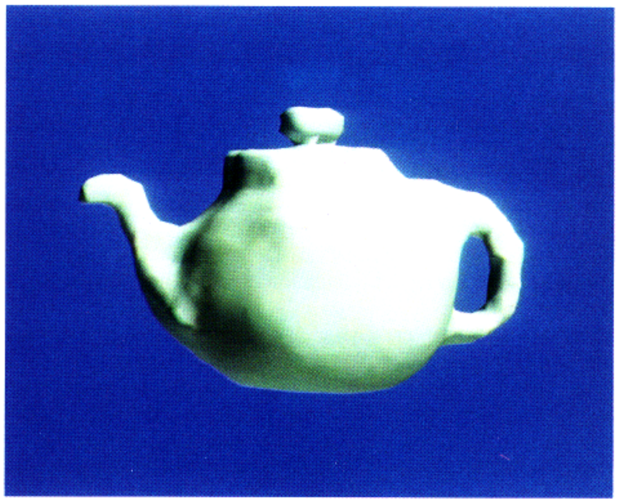

Figure 7: Teapot, after application of sanding tool: 8374 polygons

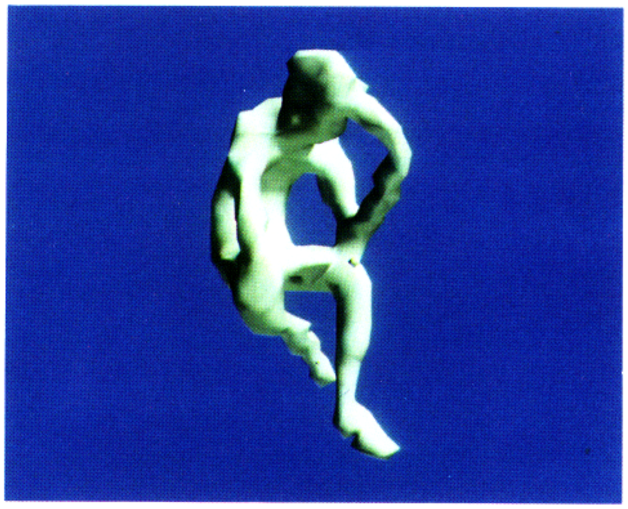

Figure 5: The thinker: only 2118 polygons!

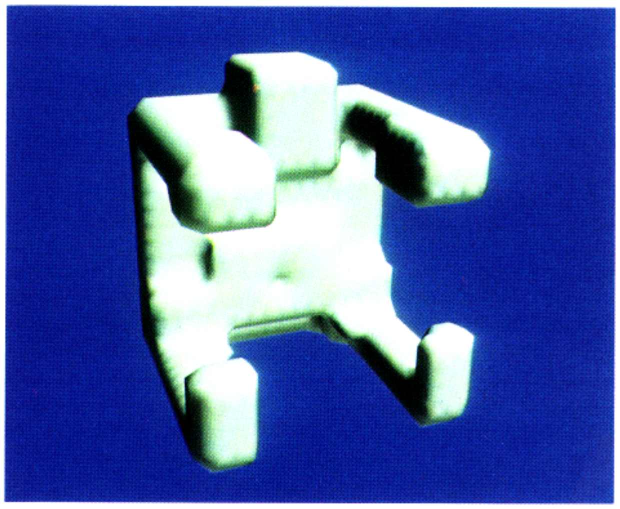

Figure 8: Sculpture made with dial box control

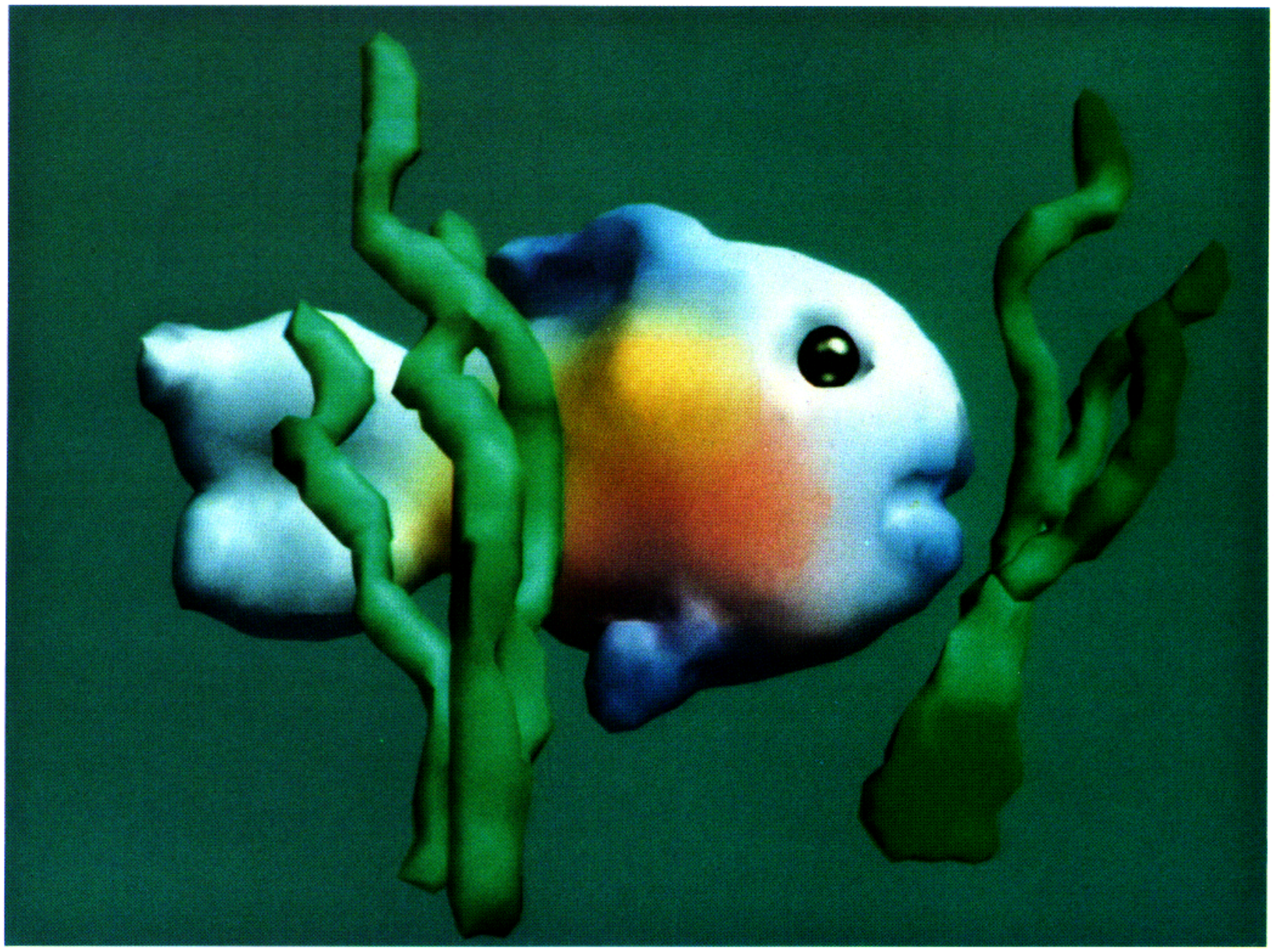

Figure 9: A fish smoothed out using sandpaper: 5066 polygons. The eye (a specular black sphere) was added after the sculpture was completed.

We make an initial image of the object described by the voxmap using a slight modification of the standard "marching-cubes" algorithm: Following the work in [16], we generate all the intersections of the isosurface with the edge of each cube, and if there are more than three we find the center of mass of these points, and then join each point to this center. (An improved algorithm, like the one in [16], might help performance somewhat, particularly as the models become more complex.) Our space is composed of an array of cubes; the voxmap gives the value of the band-limited characteristic function at the corners of the cubes. For each such cube, we use the values at the corners to estimate the intersection of the isosurface at level 0.5 with the cube's edges. These vertices are then connected to fill in a collection of polygons that approximate the intersection of the isosurface at level 0.5 with the interior of the cube (see Figure 10).

We use the gradient technique of [7] to estimate the normal to the isosurface at each polygon vertex, thus allowing smooth shading computations.

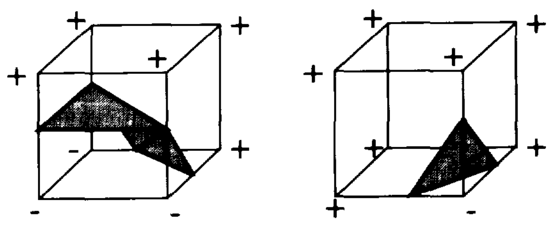

Figure 10: Determining an isosurface within a cube in the lattice. The vertices marked "+" values above the threshold, those marked "−" have values below.

3 IMPLEMENTATION DETAILS

3.1 The Tool as a Voxmap

The sculpting tool is also represented by a voxmap. One might suppose that the subtractive tool, for example, would be represented

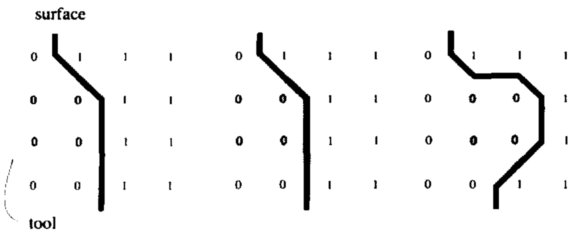

Figure 11: With a 0/1 tool, the surface of the object jumps as the tool approaches. This figure shows the analogous behavior in 2D: the gray square is the tool and the solid black line is the isocurve.

by a voxmap filled with 0s, and that the act of cutting away material would consist of copying data values from the tool voxmap to the object voxmap. This is, however, only approximately correct. Values in the tool voxmap *are* combined with those in the object map; the combination rules are described in Section 3.2.1. But the actual values in the tool voxmap are not as simple as they might seem.

If values of 0 are copied directly from the tool voxmap to the object voxmap, the results are jumpy. As the tool moves towards the object, nothing happens for a while; then, when it is sufficiently close, the object voxmap values change all at once, and the surface of the object moves rapidly. This is a consequence of the low sampling rate used to represent the characteristic function of the object (see Figure 11 for the analogous behavior in a 2D system).

We compensate for this with two tricks based on the notion of antialiased brushes [14]. First, we create a voxmap for the tool that is sampled at a higher rate than is the object (four times as many samples in each direction). The tool voxmap, for a spherical subtractive tool, is filled with 0s on the inside of a sphere, and 1s elsewhere, i.e., with the characteristic function of the sphere. We then remove most of the high-frequency components of this voxmap by filtering it twice with a $2 \times 2 \times 2$ box filter [5].[3] Second, we apply the tool to the object voxmap by determining its sub-voxel location and then selecting particular values from the tool voxmap to combine with the object voxel values. The details are presented in the following section.

3.2 Tool-Object Interaction

The central loop of the program is essentially

1. Poll repeatedly until the tool has an effect.[4]
2. Modify values in the object voxmap.
3. Recompute isosurface.
4. Redisplay isosurface and tool.
5. Return to step 1.

We will describe these steps in order: step 2 in the following subsection, and steps 3 and 4 in Section 3.3.

3.2.1 Applying the tool to the object voxmap

Points in the voxmap are identified by three indices, so that a typical voxel value is referenced as $v[i][j][k]$, where each of i, j, and k is an integer between 0 and 29. We can think of *any* point in the object region as having ijk-coordinates: a point whose coordinates are $(i, j, k) = (1.5, 2, 2)$ is on the line segment between the points represented by the voxels $v[1][2][2]$ and $v[2][2][2]$. We compute the ijk-coordinates of the tool's location, and then round

[3]Our choice of this filter size and number of iterations was determined by experimentation.

[4]For most tools, this means "until the tool has moved." The melting tool, however, has an effect at every instant.

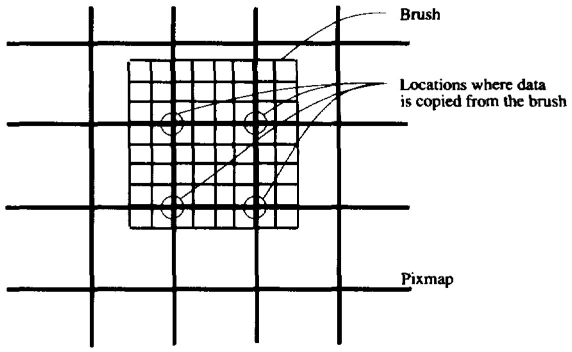

Figure 12: The 2D paintbrush location is determined to a finer resolution than that of the canvas. Values that correspond to pixel centers are actually used in applying the brush to the canvas.

each coordinate to the nearest 1/4. We then imagine the tool's voxmap as superimposed on the object voxmap at that position; 1/64th of the tool's voxel locations correspond exactly to the object voxel locations, and it is this subsample of the tool's voxmap that is combined with the object voxmap. Figure 12 shows the analogous situation in 2D.

How are object voxel values and tool voxel values combined? We use the *min* operator on each voxel:

$$\text{OBJECT} \leftarrow \min(\text{OBJECT}, \text{TOOL});$$

this prevents the 1s in the tool's voxmap from depositing material in empty space.

The additive tool is created by filtering a sphere full of 1s (with 0s outside), and applied by using the *max* operator. Two other tools use this tool data as well. The "heat gun" is applied by the rule

$$\text{OBJECT} \leftarrow \max(0, \text{OBJECT} - \text{TOOL}),$$

and the "building tool," which gradually pastes new material on in the same way that the heat gun removes it, is applied by the rule

$$\text{OBJECT} \leftarrow \min(1, \text{OBJECT} + \text{TOOL}).$$

The "sandpaper" tool is anomalous, in that it has no associated data. It is applied as follows: each voxmap value within the tool's extent is replaced by a weighted average of its current value and those of its six adjacent voxels. The central voxel is given a weight between 4 and 24, and the adjacent voxels are given weight 1. The user then adjusts the rate of "sanding" by varying the weight of the central voxel.

The tool voxel data and the object voxel data arrays must both have the same axes in this model. We would like to add other tools in the future, and allow the tool orientation to be controlled by the orientation of the Polhemus pointer; however, this requires resampling the tool voxel data to get a rotated sample, and at present this is not feasible at a reasonable refresh rate.

3.3 Regenerating the Isosurface

When the tool voxmap is applied to the object voxmap, the object data is modified only in a small region. Thus we need not recompute the entire level 0.5 isosurface — only certain polygons change, namely those that arose from cubes the values of whose vertices have been modified. Since we know exactly which vertices these are, we can readily compute the new polygons to be displayed. We call this the *incremental marching-cubes algorithm.*

To redisplay the isosurface, we wish to display the newly computed polygons and remove the polygons formerly associated with the modified regions. The removal of these defunct polygons might

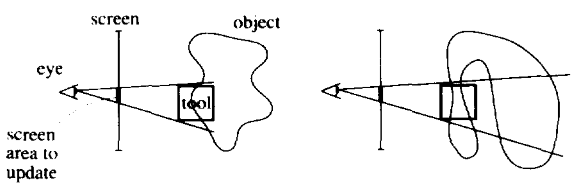

Figure 13: When the square tool cuts away material, polygons that were formerly obscured may be revealed, as shown in the right hand case.

expose certain other polygons. These obscured polygons are no longer in the z-buffer, having been overwritten by the now-defunct polygons, so we must redisplay the formerly obscured polygons too (see Figure 13 for a 2D slice of this situation). To facilitate this, we use a data structure we call a *hashgrid.*

The hashgrid makes it easy to determine which cubes in the object voxel array contribute polygons to a specific screen region. We divide the screen into a grid of squares, which we call *cells*, and associate a bucket (implemented as a linked list) to each. Whenever a cube that contributes polygons has a screen projection overlapping a cell, that cube is added to the cell's bucket. Once this array of cells, the hashgrid, has been computed, it is easy to determine which cubes' polygons must be redrawn to update a region of the screen.

To make the hashgrid efficient, we make two requirements: (1) a hashgrid cell must be at least as large as the projection of any cube to the screen; (2) a hashgrid cell must be at least as large as the largest screen extent of the projection of the tool from any point in the sculpting space. Since our tools are always at least as large as a single cube, it suffices to satisfy the second condition.

We compute the bounding rectangle of this maximum-size tool projection analytically. This is possible because the tool is constrained to lie in the sculpting space and the camera's field of view is fixed. We also compute the bounding rectangle for the projection of the entire sculpting-space cube. The larger dimension of the first rectangle is what we choose for the size of each cell edge in the hashgrid; the larger dimension of the second rectangle, divided by this cell edge length, determines the size of the array of cells.

We initialize the hashgrid during the marching-cubes algorithm. Each cube in the voxmap is examined to see whether it contributes polygons to the object; if so, it is flagged (the *contribution flag* of the cube is set), we determine which cells the cube's projection overlaps, and add the cube to the bucket for each such cell.

To determine which cells a cube hits, we project the eight vertices of the cube, and note which cells these projected vertices lie in (because of the first requirement on the size of grid cells, the projected vertices can lie in at most four different cells). Then, if the projected vertices lie in exactly three cells, we add a fourth cell to the list, as shown in Figure 14, which indicates why we must do this: it is possible for the screen extent of a cube's projection to intersect a cell in which no projected vertex lies. This *L-shape anomaly* will arise again when we discuss the effect of tool motion.

The obvious way to project the vertices of a cube to the screen is to take the coordinates of each vertex, multiply them by the current (4 x 4) transformation matrix, and then project to screen space via the perspective projection transformation. Both of these operations are linear functions, except for the homogeneous division in the perspective transformation. We use this linearity as follows: We project the corners of the entire object voxmap to homogeneous coordinates (just before the perspective divide), and use the resulting coordinates to infer the locations of each small cube's corners in this space via linear interpolation. We then perform the homogeneous division; this interpolated computation of the projected vertices reduces the per-cube computation of associated hashgrid cells from 128 multiplies, 96 additions, and 16 divisions (the cost of the two

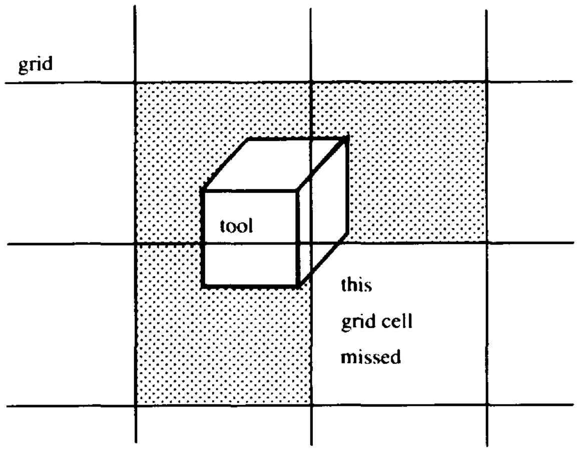

Figure 14: A projected cube may intersect four grid cells even though the projected vertices lie in only three cells. Because of this, we always add the cube to the bucket of the fourth cell.

matrix multiplies and the perspective divide of the naive approach) to just 9 multiplies, 30 additions, and 16 divisions.

To update the hashgrid when the tool is moved, each cube in the voxmap whose vertices have been modified by the tool is examined. If the cube's contribution flag is not set but the cube now contributes polygons, the flag is set and the cube is added to the hashgrid data structure. If the cube's contribution flag is set but the cube no longer contributes polygons, the flag is cleared and the cube is removed from the hashgrid.

The display is updated by determining which cells in the hashgrid might have been affected by the tool. By the second condition on hashgrid cell size, the screen projection of the tool intersects at most four cells. As before, in the event of an L-shape anomaly, the fourth cell is added to the list of affected cells. This cell list is merged with the list of cells associated with the previous tool position (these two lists often overlap, especially when the tool is being moved slowly). The screen area and z-buffer area associated with these cells are cleared. Then, for each cube in the bucket of each grid cell on the list, the polygons are regenerated and drawn. (We regenerate the polygons to save the prohibitively large space required to store them).

When we alter our view of the object, the hashgrid must be recomputed. Rather than recompute it from scratch, we use the old hashgrid to compute the new one. By examining only those cubes that appear in some bucket in the old hashgrid, we avoid performing marching-cubes computations on those parts of the object voxmap that will not contribute polygons. The process for creating the new hashgrid is therefore: (1) for each cell of the old hashgrid, look at the cubes in its bucket; (2) for each such cube, check a flag (the *already-processed* flag); if it is not set, set it, perform the marching-cubes algorithm on that cube, and insert the cube into the new hashgrid; (3) once the whole old hashgrid has been processed, clear the already-processed flags on all cubes. Using the already-processed flag is necessary because a single cube is likely to be in the buckets of more than one hashgrid cell (but no more than four).

4 FUTURE WORK

We have many plans for future work in extending this modeling paradigm. We are in the process of improving the color editing in the system, and look forward to adding patterning or even solid textures like those described in [11]. We would like to develop a voxmap with so many cubes that the screen projection of a typical cube is about the size of a screen pixel. This actually simplifies some of the algorithms (polygon rendering in the display device is no longer needed, for example); unfortunately, the memory requirements are still prohibitive. We are eager to experiment further with force feedback, as we feel that this will provide an extremely intuitive user interface. We want to add tools that have orientation, so that we can use the full range of data from the Polhemus device. We want to add various high-level operations on the sculpture such as scaling, translation, cutting, copying, and pasting regions of the sculpture, reflecting and rotating portions of the sculpture; in general, we would like to make available as many as possible of the other operations available in traditional paint programs. We would like to add hierarchy to the system, so that in regions where more detail is needed, we could locally increase the resolution of the voxel lattice. Finally, we want to gain experience with a wide selection of users so that we know better how to make sculpting a natural inclusion in the standard repertoire of modeling techniques. This should include further study of the user interface and its ease of use; our current experience involves no rigorous perceptual studies, and we feel that these may considerably enhance the interface.

5 ACKNOWLEDGMENTS

Much of this work was done by the first author as his Master's project under the direction of the second author. We appreciate the support of the Brown Computer Graphics Group, in particular Bob Zeleznik and Dan Robbins, and of the MIT Media Lab, especially the Computer Graphics and Animation Group. The first author thanks Sheri Galyean for her support and encouragement.

References

[1] J.F. Blinn. A generalization of algebraic surface drawing. *ACM TOG*, 1(3):235–256, 1982.

[2] J. Bloomenthal and B. Wyvill. Interactive techniques for implicit modeling. *Computer Graphics*, 24(2):109–116, March 1990.

[3] S. Coquillart. Extended free-form deformation: A sculpting tool for 3d geometric modeling. *Computer Graphics*, 24(4):187–196, August 1990.

[4] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition, 1990.

[5] P.S. Heckbert. Filtering by repeated integration. *Computer Graphics*, 20(4):315–321, August 1986.

[6] T. Van Hook. Real-time shaded nc milling display. *Computer Graphics*, 20(4):15–20, August 1986.

[7] W.E. Lorenson and H.E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[8] M. Minsky, M. Ouh-young, O. Steele, and F. Brooks. Feeling and seeing: Issues in force display. *Computer Graphics*, 24(2):235–243, March 1990.

[9] B. F. Naylor. Sculpt: An interactive solid modeling tool. In *Proceedings of Graphics Interface '90*, pages 138–148, May 1990.

[10] A. Pentland, I. Essa, M. Friendmann, B. Horowitz, and S. Sclaroff. The thingworld modeling system: Virtual sculpting by modal forces. *Computer Graphics*, 24(2):143–146, March 1990.

[11] K. Perlin. An image synthesizer. *Computer Graphics*, 19(3):287–296, July 1985.

[12] T.W. Sederberg and S.R. Parry. Free-form deformation of solid geometric models. *Computer Graphics*, 20(4):151–160, August 1986.

[13] D. Terzopoulos and K. Fleischer. Modeling inelastic deformation: Viscoelasticity, plasticity, fracture. *Computer Graphics*, 22(4):269–278, August 1988.

[14] T. Whitted. Anti-aliased line drawing using brush extrusion. *Computer Graphics*, 17(3):151–156, July 1983.

[15] L. Williams. 3d paint. *Computer Graphics*, 24(2):225–233, March 1990.

[16] B. Wyvill and D. Jevans. Table driven polygonization. In *SIGGRAPH '90 Course Notes, Modeling and Animation with Implicit Surfaces*, pages 7/1–7/6, August 1990.

[17] B. Wyvill, C. McPheeters, and G. Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4), 1986.