SCULPT
An Interactive Solid Modeling Tool

Bruce Naylor
Computing Systems Research Laboratory
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

Sculpt, an interactive polyhedral solid modeling system, combines the effectiveness of BSP trees for performing geometric search, set operations, and determination of visibility, with the rendering performance of the AT&T Pixel Machines to provide interactive sculpting of texture mapped solids. The paradigm presented to the user is one in which a tool is used to modify a workpiece repeatedly by set operations (union, difference, or intersection). The user may choose between performing one set operation at a time with repositioning of the tool between operations, or sweeping the tool during which either union or difference is performed "continuously" (at sampled tool positions). Solid near-plane clipping is also provided using a BSP tree clipping algorithm. The user interface is simple and was designed to allow portability to a variety of workstations. Update rates using a Sun 3/260 and a Pixel Machine 964 for texture mapped models of ~1000 polygons require ~0.2 secs per update when using a simple tool.

Keywords Solid Modeling, Interactive Design, BSP Trees

1. Introduction

A major goal of computer graphics since its inception has been to provide interactive design of 3D objects. Achieving the performance required for this has certainly proven to be difficult. However, we have recently reached a reasonable level of such capabilities through the combined performance gains of: a new representation of geometry admitting faster algorithms, non-geometric techniques such as texture mapping, specially designed processors with significant floating point capacity, and a parallel architecture specifically designed for 3D rendering. In particular, we have employed the BSP tree for representing objects in conjunction with an AT&T Pixel Machine to achieve interactive performance for designing 3D solids via a sequence of boolean set operations.

The earliest version of this system was described briefly in [Thibault and Naylor 87]. Developments since that time are explored in this paper, including user interface design, BSP tree clipping, numerical robustness, efficiency enhancements to the neighborhood operation, texture mapping, and porting to a variety of workstations, including the Pixel Machine (see [Potmesil and Hoffert 89] for a discussion of this architecture). Descriptions of other BSP tree based systems providing interactivity can be found in [Fuchs, Grant and Abrams 83] (basic viewing) and [Chin and Feiner 89] (shadow generation).

Sculpt currently runs in four environments: a Pixel Machine attached to a Sun workstation, any Silicon Graphics Iris 4D, a Sun workstation with a TAAC-1 graphics accelerator, and "stand alone" on any Sun workstation by using its bit-mapped screen for display. The Pixel Machine supports both color and texture maps, the TAAC-1 and Iris only color, and the Sun workstation either color or bi-level renderings.

2. User Model

The paradigm presented to the user is that of sculpting a workpiece with a tool. The user manipulates continuously the position of a tool with respect to a workpiece while being able to vary the view. Once positioned, the tool may be subtracted from the workpiece, or added, or the intersection between the tool and workpiece can be formed. Intersection and difference cause the tool to be made invisible to allow viewing of the result. In "sweep" mode, set operations are performed at as high a rate as possible. As soon as an operation is completed, the tool's position is sampled and the next operation is begun, thus resulting in a quantized sweep. The tool, selected from a palette of eight, may be scaled and rotated, and the user can choose its color to be solid or texture mapped.

The paradigm of sculpting a workpiece has been found to be intuitive, effective for the task, and simple to learn. Most users are able to achieve a reasonable level of proficiency in a few hours (including artists). Clearly, tapping into human beings' considerable familiarity with manipulating 3D objects is behind this. However, the simplicity of the user interface contributes as well. Since we wanted to run Sculpt on a variety of workstations, we built the interface out of the lowest common denominator: a single 3-button mouse and pop-up menus. Only three levels of menus are needed with most menu items not involving subsequent sub-menus. The number of items per menu is typically no more than ten, and the right button is used exclusively for menu selection. Consequently,

choosing menu items is relatively fast.

While the right button is reserved for menu selection, geometric input can utilize the mouse xy position as well as the left and middle buttons. However, given that the various demands for input exceed this facility, function overloading of the mouse is necessary. The highest level of functional partitioning distinguishes between controlling the view/model[1] and controlling the tool. In view mode, mouse motion typically controls horizontal and vertical rotations of the view/model, but occasionally it is used for xy positioning (centering). In tool mode, mouse position always affects xy tool position.

The remaining input functionality is provided by the left and middle buttons used in tandem to form a valuator; this provides all other geometric control such as scaling (more on this below). To change the input functionality entails selection of menu items. For instance, rapid toggling between view and tool modes involves selecting only one of two first level items. To remind the user of exactly which of the several input modalities is in effect, the cursor icon is changed to indicate the current state. Also, any time an action is selected from the menu, a "wait" icon is displayed until the action is completed.

3. The Tool

The tool is selected from a small set (8) of simple convex polyhedra, including a cube, cone, cylinder and sphere. Convexity of the tool arises from the implementation of the set operation (the solution for the non-convex case, which is given in [Thibault and Naylor 87], is noticeably more complex). However, users have not found tool convexity to be a significant limitation in the context of sculpting.

When in tool mode, the position of the tool tracks the xy position of the mouse. The tool's motion is always with respect to the screen: that is, left mouse motion always moves the tool to the left, etc. Consequently, changing the view will change the plane in model space in which the tool moves. This has proven to be quite effective as a means of positioning the tool, and results in an alternation between modifying the view and repositioning the tool. Translation perpendicular to the screen can be achieved using the mouse buttons, but this method is used typically for only small adjustments in the neighborhood of the target position. Scaling may be symmetric or differential along any principal axis of the tool; rotation about each principal axis is provided as well.

1. The model is the workpiece and tool together. Transforming the model admits the inverse interpretation of transforming the view, although only one affine transformation is involved, one that maps the model to the screen. So, for instance, rotating the model clockwise can be equivalently interpreted as rotating the view counter-clockwise. With Sculpt, the user typically has the subjective sense of transforming the model.

The buttons are used to control all of these.

The color attribute is defined for the entire tool, that is, all faces have the same attribute. The attribute may be either a single color or, in the case of the Pixel Machine, a texture. Texture coordinates for each vertex of the tool are defined as part of the tool definition and are not modifiable by the user. The current texture palette is selected from a predefined set of such palettes. Each texture map uses 8 bits/pixel along with its own 256 x 24 bit color table. The area available for storing texture maps is comprised of 4 channels, 8 bits/channel, with 256x256 pixels/channel. Thus each channel can contain one 256x256 texture or 4 128x128 or 16 64x64 textures. The current palettes provide various combinations of 256x256 and 128x128 textures, for up to 16 of the later. New textures and new palettes can be created using software tools external to Sculpt.

4. Mouse Input Issues

Restricting input to a single mouse provided by a windowing system required addressing a number of issues. The first arises from the difference between the desire for sampled input and the event driven and queued input commonly provided by window systems. The time required to draw a single image depends upon the number of polygons, the number of pixels generated during the sampling process, and the cost of determining the value of each pixel (texture mapped or constant color, etc.). Interactivity can tolerate up to maybe as much as 1 second per update; clearly much slower than the rate at which the mouse can generate events. One does not want, for instance, to see a new frame for every detectable change in mouse position. As a consequence, sampled input was deemed the preferred modality.

4.1 Sampled Input

To map an event driven model of input into a sampled model requires maintaining an input state, viz. position and buttons up/down, through the processing of mouse events. Whenever the state is needed, the entire event queue is processed to determine the current state, thus giving the effect of being sampled. However, this does not obviate completely the need for the notion of events. When no change has occurred in the state of the input, there is no need to regenerate an image; and in a multi-processing environment, avoiding this unnecessary computation is highly desirable. As a result, the boolean "geometric input has changed state" is determined to control generation of new frames.

When the rendering time is significant, as it is using a Sun workstation for rendering, detecting the starting and stopping of a mouse sweep becomes important. Detecting the start of a sweep is needed to avoid the first sample being an insignificant small change from the starting position, with the consequence that a noticeable amount of time may be spent rendering only a slightly different

model. Detecting the stopping of a sweep is need to avoid *follow-through*: continued change after the input has stopped. Consider that the input for a frame is sampled at time t0, the input stops at t1, and the current frame finishes at t2, where t0 < t1 < t2. As the input has changed since its last sampling at t0, a new frame will begin to be drawn at t2; thus, the user will experience two new frames after the sweep has stopped, one at t2 and one sometime later. When the time to draw a frame is long, this can be very disconcerting to the user; however, even with relative fast frame times, this follow-through is subliminally annoying. By using the time-stamps of events together with the rendering beginning and finishing times, small starts and old stops can be detected and discarded.

Another issue arose from using the mouse for both menu selection and geometric input. Menu selection requires changing the mouse position; once this is completed, the mouse's position reverts to providing geometric input. If the absolute position of the mouse is used, a discontinuity of position will have been introduced. Using absolute positions directly also encounters the problem of the mouse's area of excursion being rather small. Both of these limitations are overcome by using relative change as the logical input device with a reset capability. After a menu selection is completed, the relative change is reset to 0 by sampling the current mouse position. However, the resampling must be delayed until a new mouse motion event occurs; thus, the call to Reset() simply sets a flag to indicate the need for resetting upon the next event. This also solves the second problem: when a boundary is reached, the user presses the menu button, but re-positions the mouse outside of the menu area instead of choosing an item. This can be used when the cursor reaches the window boundary to reposition the cursor towards the center.

4.2 Mapping Input to Values

For each kind of geometric input (view rotation, tool translation, etc.), some constant must be specified that defines the mapping from input device coordinates to the desired geometric quantity. Clearly the button-down state needs mapping to a quantity, but scaling of mouse xy coordinates is also needed. The rate of change for each such input function is specified by an empirically derived table; and these rates are subsequently calibrated to each class of workstation by global calibration values for both motion and buttons.

Another issue arises from the non-linearity of the perspective projection. If a fixed quantity of translation is used for each translation-event, then the user will perceive larger step sizes the more one zooms in. This can be corrected by applying the view-to-model transformation to the mouse input (the inverse perspective transform). However, we found it useful to provide finer control the more one zooms in by using a negative exponent less than one. Thus, the user zooms out to perform coarse positioning and zooms in for higher precision placement. In addition, we use a traditional solution to the problem of providing exact input from a high resolution geometric input device: quantization of input values. For instance, integer*45-degree rotations can be easily achieved by a relatively slight quantizing of mouse input.

Varying rendering times posed another problem: a constant rate button controlled rotation would rotate much faster with a simple model than with a complicated one. Similarly for scaling or translation. Our solution is simply to modulate the quantity of change associated with each button sample by the time required to render the image; this then yields a fairly constant unit of change per second, at the expense of control resolution. However, zooming-in can be used to restore this resolution.

5. BSP Tree Review

As stated earlier, the representation of the workpiece is a binary space partitioning tree. A BSP tree is formed by using hyperplanes to recursively partition d-space (typically 3-space), resulting in its hierarchical decomposition into a binary tree of convex regions (represented combinatorially by a binary tree). The leaves of the BSP tree, called cells, form a convex partitioning of d-space. Figure 1a shows a BSP tree induced partitioning of the plane and 1b shows the corresponding binary tree. The root node represents the entire plane. A binary partitioning of the plane is formed by the line labeled u, resulting in a negative halfspace and a positive halfspace. These two halfspaces are represented respectively by the left and right children of the root. A binary partitioning of each of these two halfspaces may then be performed, as in the figure, and so on recursively. When, along any path of the tree, subdivision is terminated, the leaf node will correspond to an unpartitioned region (a cell). For any node of the tree, the corresponding region is defined by the intersection of the set of open halfspaces determined by each hyperplane associated with a node on the path to that region.

To define polytopes, we need the attribute *classification* ::= { *in*,*out* }. If, in figure 1, we assign to cells 1 and 5 the value *in*, and to the rest *out*, then we will have determined a concave polygon of six sides. Defining a polytope as the closure of a union of convex cells allows a single tree to represent any linear semi-algebraic set: they can have concavity, any genus, any number of connected components, non-manifold boundaries, and unbounded interiors. In addition, all sets are treated identically by the algorithms. The boundary of the set lies in the partitioning hyperplanes, or more precisely, in the *sub-hyperplanes*: that subset of a partitioning hyperlane that intersects the region it partitions. The boundary can be obtained either by classifying a sub-hyperplane by a neighborhood search operation, or it can be represented explicitly by polygons (assuming D=3), each stored at the internal node whose sub-hyperplane contains them.

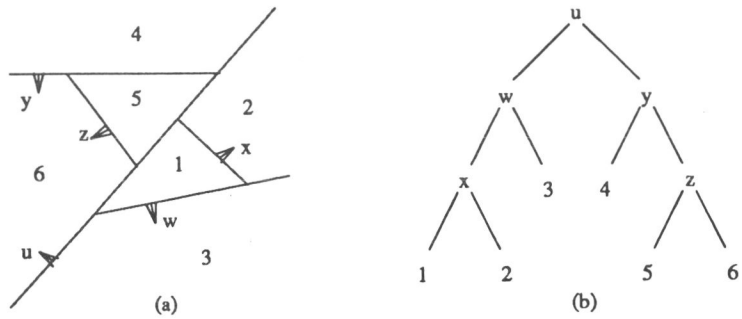


Figure 1. Partitioning of a 2D BSP Tree (a), and its binary tree (b).

6. Set Operations

Set operations are computed in 3-space and result in a modification to the 3-space model; therefore, they are view independent. This is in contrast to methods that evaluate the set operations only in a finite set of 1D spaces determined by viewing rays (e.g. ray casting). While computing set operations in 1D is simpler that 3D, the work must be repeated for every frame.

The workpiece is represented by a 3D BSP tree which explicitly includes the faces of the polyhedron. Each face is represented by a set of convex polygons, and these are represented by a list of vertices (i.e. as a B-rep). Polygonal convexity is required by all of the rendering systems that we use and this is generated automatically during neighborhood operations by the BSP tree's convex decomposition of space. The tool is represented simply as a list of convex polygons, where each polygon is once again a list of vertices.

To perform a set operation, the relative spatial relationship between the two objects must be determined by a geometric search. This can be accomplished by "inserting" the tool into the tree. Figure 2 shows a quadrilateral as the tool being subtracted from a triangular workpiece. At each partitioning node, a binary partitioning operation is performed that classifies each face of the tool with respect to the hyperplane at that node as {*InNegHs*, *InPosHs*, *InBoth*, *On*}. If there are no faces on one side, then the subtree on that side is either entirely inside or outside the tool. Determining which is the case is achieved by locating a point in the sub-hyperplane with respect to the planes of the tool faces. This results in classifying an entire subtree with respect to the tool without visiting any of its nodes. After this, faces are then "sent" to the respective subtrees which contain them and this partitioning process continues until a cell is reached. At this point, the faces are either discarded or used to extend the tree, depending upon the set operation and the classification of the cell. (See [Thibault and Naylor 87] or [Thibault 87] for a complete description).

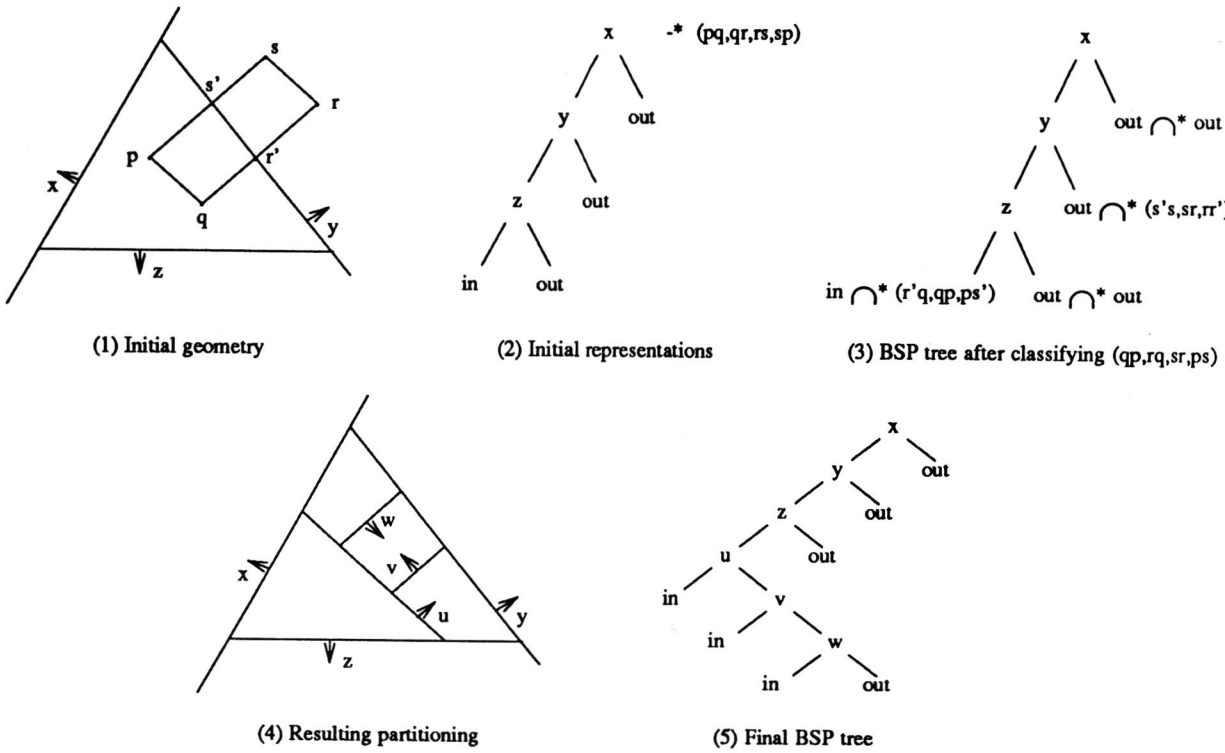
When manipulating the tool, the user views a model that is the union of the workpiece and the tool. Each time the tool is changed, it is inserted into the tree; that is, a union is performed, except that the workpiece faces are not modified, since tool faces occlude any workpiece faces inside the tool. When the tool is changed, the previous instance of the tool is first stripped from the tree, the modifications to the tree having been noted in a table.

For those wishing to "see" the underlying representation, the user may choose to display the BSP tree's binary tree structure. This simple drawing of the binary tree is updated in every frame in which the tool changes or a set operation is performed. This, together with a display mode showing polygon edges, provides a means to gain some understanding of the space partitioning by moving the tool around and observing the changes to the tree and to the edges resulting from partitioning the tool.

6.1 The Neighborhood Operation

User selected set operations result in modifications to the BSP tree representing the workpiece, including updating of faces via a neighborhood operation to reflect changes to the boundary. This will, in effect, compute the closure of the interior of the new workpiece. For example, in figure 2 the face of the workpiece lying in the sub-hyperplane at y needs to change to reflect the hole created by subtracting the tool. This requires examing the new neighborhood of the face. By inserting the face into y's left subtree, we can discover which subsets are *in* and which are *out*, and then retain only those that are *in*. Since the set operation is subtraction, the neighborhood on the exterior side of the face, represented by y's right subtree, could not have changed (0-S = 0) and so does not need to be examined. This would also be true had the operation been intersection; for union it is the opposite neighborhood, i.e. the interior neighborhood, that cannot change.

It is important to make this closure operation as efficient as possible, and at the time of [Thibault and Naylor 87] this operation was not well understood. Whenever the tool is entirely in one halfspace of a node's hyperplane, the correct values for the faces in the sub-hyperplane as well as those in the opposite halfspace are



(1) Initial geometry

(2) Initial representations

(3) BSP tree after classifying (qp,rq,sr,ps)

(4) Resulting partitioning

(5) Final BSP tree

Figure 2. Workpiece - Tool.

determined directly by the set operation: they are either unchanged or discarded along with their tree nodes. Therefore only those workpiece faces at nodes in which the tool faces were found to lie in both halfspaces require the neighborhood operation. Even this can be avoided if no change has occurred in the relevant subtree; consequently, the recursive call performing the set operation on a subtree returns a boolean "subtree has changed" which conditions the neighborhood operation.

The neighborhood operation generates a convex decomposition of the relevant neighborhood of a face, this resulting from the subtree's hyperplanes that intersect the face. It is the retention of only the *in* faces for an interior neighborhood, and only the *out* faces for an exterior neighborhood that results in the correct determination of the boundary. However, this process may produce more fragments that necessary. If two face fragments created by some partitioning hyperplane have the same classification, one can replace these with their union. Since these fragments were created by partitioning some polygon, this original polygon is in fact the desired union. Thus, whenever it is necessary to partition a polygon, a copy of it is saved so that it may be used as the return value whenever appropriate.

Another issue concerns the problem of faces that are both coplanar and overlapping. If a tool face is found to be *On*, it is retained at the node at which this occurs, since it lies in that node's sub-hyperplane. If this face overlaps (intersects) a workpiece face, this fact cannot be discovered by the neighborhood operation, since the region of overlap will have the same neighborhood for either face. The solution presented in [Thibault and Naylor 87] involves recursing in dimension, which requires procedures and data structures specific to this task. A simple alternative is to classify tool *On* faces *before* performing the set operations on the subtrees. If the operation is union, then classify the interior neighborhood and retain *out* fragments, since the *in* fragments must overlap workpiece faces. For difference, retain *in* fragments, and for intersection no tool faces are needed.

6.2 Numerical Robustness

Relying on finite precision to support semantics developed for the continuum is a bane of geometric computation. We have a simple scheme for dealing with this problem, however. To begin with, the semantics of the BSP tree spatial partitioning is based on open sets: the cells are open sets, set operations are on open sets, and partitioning a face is in terms of open sets in the plane of the face. As a result, regularization is not an issue per se, since union and intersection of opens set yields open sets. Instead, the boundary is created by explicitly executing the closure operation. In addition, BSP trees, unlike boundary representations, use no topological information. Consequently, sensitivity of the algorithms to correct and consistent answers in determining co-incidence is greatly reduced. In particular, there is no use of transitivity in determining co-incidence so that one of the major problem with "epsilon" based numerical methods is avoided.

Our numerical problems arise because we need the semantics of planarity for polygons. In particular, we need to guarantee that all vertices of a polygon lie *On* its supporting hyperplane and that the vertices do not oscillate from one side of a partitioning hyperplane to the other. The simple solution we use is to treat hyperplanes as having a thickness so as to "dampen" numerically created noise. This is in keeping with BSP trees' space partitioning nature, since it is equivalent to treating hyperplanes as D-dimensional open sets. For partitioning polygons, this gives us a polygon being *InNegHs* if and only if all its vertices are *InNegHs* or *On*, that is, if and only if its interior lies entirely *InNegHs*.

Unfortunately, this alone is insufficient to provide the semantics needed by the neighborhood operation. This semantics requires that along any path in the tree, all hyperplanes are distinct. However, numerical error can cause a face fragment, which of course is *On* its node's hyperplane, to be classified during the neighborhood operation as *On* a descendant's hyperplane. Our solution is to use two different hyperplane thicknesses: one for set operations and a lesser one ($\tilde{}1/5$) for neighborhood operations. This creates a "numerical space" between any two sub-hyperplanes on a path. While this method is not provably robust (like all other methods), in practice it has provided approximately four orders of magnitude in size using only single precision. In our interactive domain, where input is already quantized relatively coarsely, this is sufficient to avoid experiencing problems arising from numerical errors.

7. Rendering

The BSP tree was originally developed in the context of determining visibility priority in 3D scenes [Schumacker et al] [Fuchs, Kedem, Naylor 80], although in [Naylor 81] it is defined as a dimension independent and application independent spatial partitioning scheme akin to k-d trees and linear decision trees. Visibility priority arises from the following observations. Since both a plane and a view-ray are linear, they can intersect at only one point. This intersection point partitions the ray into a *Near*, *On* and *Far* subsets. Thus all points on the side of the plane containing the viewing position, the *Near* subset, have higher visibility priority than those in the plane, the *On* subset, which in turn have higher priority than those in the other halfspace, the *Far* subset. As a result, a total priority ordering of the cells and sub-hyperplanes can be generated by a recursive inorder-like traversal in which the location of the view with respect to a node's partitioning plane determines the priority ordering of its subtrees. In [Schumacker et al], faces where contained in the cells (at the leaves) and in [Fuchs, Kedem and Naylor 80] they are contained in the sub-hyperplanes. Sculpt employs this later methodology, and therefore can generate either near-to-far or far-to-near orderings.

7.1 BSP Tree Polyhedral Clipping Algorithm

The capacity to easily perform set operations on BSP trees can be exploited to provide polyhedral/solid clipping. In Sculpt, the user is provided with solid clipping to the near clipping-plane, thus creating another form of continuous set operations. This operation, calculated for each view, yields the intersection of the model with the halfspace determined by the near clipping-plane whose position can be controlled continuously by the mouse buttons. Rather than make a new copy of the tree and then intersect it with the halfspace, the operation is interleaved with the display traversal of the BSP tree that generates the visibility priority ordering of the polygons. No new tree nodes are allocated. It serves to eliminate entire subtrees on the viewer's side of the near clipping-plane without traversing those subtrees, and it creates the faces resulting from the intersection of the clipping-plane and the model. In figure 3, the algorithm for generating a lowest-to-highest priority ordering with solid near plane-clipping is given.

This algorithm can be easily extended to perform clipping to the entire view-volume, that is, to perform intersection between the view volume and the model. The view volume is simply represented the same as the tool, i.e. as a list of faces, and the algorithm becomes almost the same as that used to perform intersection with the tool. The differences are that the model's tree is not modified and that the neighborhood operation needed to form the boundary of the clipped model is replaced by clipping of model faces in the *InBoth* case to the planes of the view volume, and by display of near-plane faces falling into *in* cells of the model. If one substitutes in figure 3 "view_volume" for "near_plane", the BSP tree clipping algorithm will result. This requires that Partition_Polygons() and Locate_Point() handle a list of polygons, as in fact they do for set operations with the tool, and also that partitioned_polygons.location == *On* whenever there is a single face co-incident with the partitioning hyperplane. Otherwise, they are identical.

Only the intersection of the model with the near clipping-plane will actually lead to a visual difference between this polyhedral clipping and the usual polygon clipping method, since faces lying in the side/window planes and far plane are not visible. However, the BSP tree clipping can be much more efficient than transmitting the entire model through the viewing pipeline, an O(n) operation. Subtrees found to be completely outside or completely inside the view volume are not visited, and therefore are "clipped" atomically. This savings will be most dramatic when most of the model is inside the view volume as is an overview, or mostly outside as when one is zoomed-in viewing a small portion of the model. As a consequence, sub-linear performance for clipping is usually attained; and for a relatively balanced tree, this can result in an O(log n) expected case.

7.2 Visibility Priority

Using a far-to-near priority ordering in leu of the depth buffer algorithm is particularly attractive when using the Sun workstation as the display device since it obviates the need for the space and time required by the depth buffer. While the other workstations can provide depth buffers more easily for some additional cost in hardware (e.g 2-4M of memory), the computation time for the z calculations are eliminated. (A casual comparison on a Personal Iris indicated an averaged speed up of 2x; but this was much less on bigger machines.) Numerical errors resulting from the perspective projection (1/z) are eliminated as well. These errors are very noticeable when rotating an object which has nearly co-planar faces: the ordering of the faces oscillates.

Additional performance gains can be obtained using instead a near-to-far priority ordering in which only visible pixels are rendered. The savings accrued from this grows as the cost of pixel rendering grows as a result of texture mapping, phong shading, transparency and/or anti-aliasing calculations, and also as the depth complexity increases, due to the avoidance of rendering occluded pixels. To achieve this, a one-bit mask is maintained at each pixel which is initially unset. Analogous to the depth buffer algorithm, the color of each polygon sample is computed and the corresponding pixel updated if and only if the mask in unset. The mask is then set to preclude subsequence modification. We have experienced noticeable improvements from this when using texture mapping (with intensity scaling), even with relatively low depth complexity (we are not currently using any of the other rendering technics).

Even more gains can be obtained by using sub-pixel masks to perform anti-aliasing [Fiume and Fournier 83]. The 1-bit/pixel mask above can be extended to a 4x4 mask, for instance, thus requiring 16-bits/pixel, or to a 2x4 which can fit into the alpha channel. If all the bits of a mask are set, the pixel has been completely rendered (using Set == 0 makes this an == 0 test). Otherwise, a bit mask representing the intersection of the current pixel with the polygon is generated. The current pixel's mask is subtracted from the polygons mask to determine the visible sub-pixels (A - B -> A & ~B). This can then be used with a convolution table [Abrams, Westover and Whitted 85] to determine the filtered value that is added to the current pixel's color. The pixel mask is then updated by or-ing the two mask together. Since this process is guaranteed to filter only the visible surfaces, it will produce the correct pixel value up to the error introduced by quantization. This is not the case for the method in [Fiume and Fournier 83] which relies on the depth buffer and is susceptible to "color bleeding" from occluded pixels, or the A-buffer [Carpenter 84] which uses sorting of micro-polygons by their z-value sampled at the pixel's center and so may generate ordering errors. We are currently implementing this technique on the Pixel Machine.

In closing, we note that it is conceptually a simple matter to construct an analytic representation of the visible surfaces instead of the discrete bit-masked representation such discussed. Given a near-to-far ordering on the faces, one needs only to form their union in priority order. The Bspt <union> B-rep -> Bspt algorithm described in section 6, when applied in 2D to ordered and projected faces, generates a 2D BSP tree representing the visible surfaces, i.e representing the image/picture. As with any visible surface algorithm (especially analytic ones), this can be used to generate shadows as well. The resulting shadow algorithm is, in fact, almost identical to the method described in [Chin and Feiner 89].

8. Performance

In figures 4-9, we show images generated by sculpt on a Pixel Machines 964d connected to a Sun 3/260. This machine has a 1024x1280 screen, 64 pixel processors and 18 transformation processors. Figure 4 shows a simple object designed with sculpt employing 4 texture maps, each 256x256. Figure 5 demonstrates a sweep of the tool with union performed at each sampled position. Figure 6 shows a head and its corresponding binary tree, while figure 7 demonstrates the effect of solid near-plane clipping and figure 8 sweeping with subtraction. Finally, figure 9 demonstrates the use of 16 textures, each 128x128. Performance data for these images is given in Table 1. All numbers have been rounded to some approximate value.

9. Next Generation

The most overt deficiency of the current system results from a technological limitation at the time of its design: the absence of a closed algebra on BSP trees for set operations. However, this deficiency has since been remedied. We have just completed implementation of a new BSP tree library/module that provides such a capability. The current user interface to this library is language based: objects are defined as C programs or as shell scripts. In the near future, we will be constructing a new version of Sculpt using this library. This will then remove the current limitation on the tool of being convex and predefined and will allow the user to perform general CSG modeling. We envision retaining the sculpting paradigm, however, in order to sculpt components which are subsequently added to other components to form objects, and also to sculpt new tools, rather than being confined to the predefined set. We also intend to construct a much more sophisticated user interface that will support a variety of input devices using user interface building tools currently under development at Columbia University.

Acknowledgements

Thanks to Bill Thibault, whose did the original implementation, and whose basic design is still preserved throughout several rewrites, and to John Amanatides who

object	cpu/frame (seconds)	tree size (nodes)	number of polygons	average depth
lamp stand	0.1	125	160	13
+ sweep	0.2	850	950	15
head	0.3	1500	1500	20
clipped head	0.3			
head - sweep	0.3	1500	1600	20
16 cubes	0.15			

Table 1. Statistics for figures 4-9.

ported Sculpt to the Pixel Machine, TAAC-1 and the Sun.

References

[Abrams, Westover and Whitted 85]
Greg Abrams, Lee Westover and Turner Whitted, "Efficient Alias-free Rendering using Bit-masks and Look-up Tables", *Computer Graphics, Vol 19(3)*, pp. 53-59, (July 1985).

[Carpenter 84]
Loren Carpenter, "The A-buffer, an Antialiased Hidden Surface Method", *Computer Graphics Vol 18(3)*, pp. 103-108, (July 1984).

[Chin and Feiner 89]
N. Chin and S. Feiner, "Near Real-Time Shadow Generation Using BSP Trees", *Computer Graphics Vol. 23(3)*, pp. 99-106, (June 1980).

[Fiume and Fournier 83]
"A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer", *Computer Graphics Vol. 17(3)*, pp. 141-150, (July 1983).

[Fuchs, Kedem, and Naylor 80]
H. Fuchs, Z. Kedem, and B. Naylor, "On Visible Surface Generation by a Priori Tree Structures," *Computer Graphics Vol. 14(3)*, pp. 124-133, (June 1980).

[Fuchs, Abram and Grant 83]
Henry Fuchs, Gregory D. Abram, and Eric D. Grant, "Near Real-Time Shaded Display of Rigid Objects," *Computer Graphics Vol. 17(3)* pp. 65-72 (July 1983).

[Naylor 81]
Bruce F. Naylor, "A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes," Ph.D. Thesis, University of Texas at Dallas (May 1981).

[Potmesil and Hoffert 89]
Michael Potmesil and Eric Hoffert, "The Pixel Machine: A Parallel Image Computer", *Computer Graphics, Vol. 23(3)*, pp. 69-78, (July 1989).

[Schumaker et al 69]
R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp, "Study for Applying Computer-Generated Images to Visual Simulation," AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory (1969).

[Thibault and Naylor 87]
W. Thibault and B. Naylor, "Set Operations On Polyhedra Using Binary Space Partitioning Trees," *Computer Graphics Vol. 21(4)*, (July 1987).

[Thibault 87]
William C. Thibault, "Application of Binary Space Partitioning Trees to Geometric Modeling and Ray-Tracing", Ph.D. Dissertation, Georgia Institute of Technology, Atlanta, Georgia, (1987).

```
Display_Bspt( T : Bspt ; near_plane : ( polygon : Brep; hp : plane ) )

hp ::= hyperplane bp ::= binary partitioner : ( hp, faces, interior_pt )

IF T.is_a_cell
 THEN
   IF T.membership == IN
    THEN
      Render_Polygons( near_plane.polygon )
    ELSE
      Discard_Polygons( near_plane.polygon )
    END
 ELSE
   partitioned_polygons = Partition_Polygons( near_plane.polygon, T.bp.hp )
   CASE partitioned_polygons.location
      InNegHs :
          bp_location := Locate_Point( T.bp.interior_pt, near_plane.hp )
          IF bp_location == InNegHs
           THEN
              Display_Bspt( T.posHs_subtree, NULL )
              Render_Polygons( T.bp.faces )
              Display_Bspt( T.negHs_subtree, near_plane )
           ELSE
              /* Clip away near subtree and T.bp.faces */
              Display_Bspt( T.negHs_subtree, near_plane )
           END
      InPosHs :
          bp_location := Locate_Point( T.bp.interior_pt, near_plane.plane )
          IF bp_location == InNegHs
           THEN
              Display_Bspt( T.negHs_subtree, NULL )
              Render_Polygons( T.bp.faces )
              Display_Bspt( T.posHs_subtree, near_plane )
           ELSE
              /* Clip away near subtree and T.bp.faces */
              Display_Bspt( T.posHs_subtree, near_plane )
           END
      On :
          eye_location := Locate_Point( eye, T.root.hp )
          IF eye_location == InNegHs
           THEN
              Display_Bspt( T.posHs_subtree, partitioned_polygons.inPosHs )
              Render_Polygons( T.bp.faces OR near_plane.polygon )
              /* Clip away near subtree */
           ELSE
              Display_Bspt( T.negHs_subtree, partitioned_polygons.inNegHs )
              Render_Polygons( T.bp.faces OR near_plane.polygon )
              /* Clip away near subtree */
           END
      InBoth :
          eye_location := Locate_Point( eye, T.root.hp )
          IF eye_location == InNegHs
           THEN
              Display_Bspt( T.posHs_subtree, partitioned_polygons.inPosHs )
              Render_Polygons( Clip_Faces( T.bp.faces, near_plane.hp )
              Display_Bspt( T.negHs_subtree, partitioned_polygons.inNegHs )
           ELSE
              Display_Bspt( T.negHs_subtree, partitioned_polygons.inNegHs )
              Render_Polygons( Clip_Faces( T.bp.faces, near_plane.hp )
              Display_Bspt( T.posHs_subtree, partitioned_polygons.inPosHs )
           END
   END Case END Display_Bspt
```

Figure 3. Displaying BSP tree faces in far-to-near priority ordering with near-plane clipping.

Figure 4

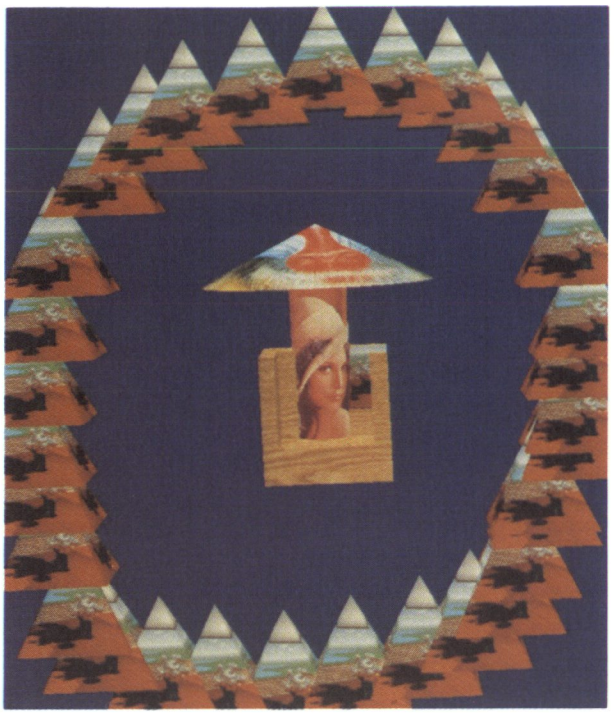


Figure 5

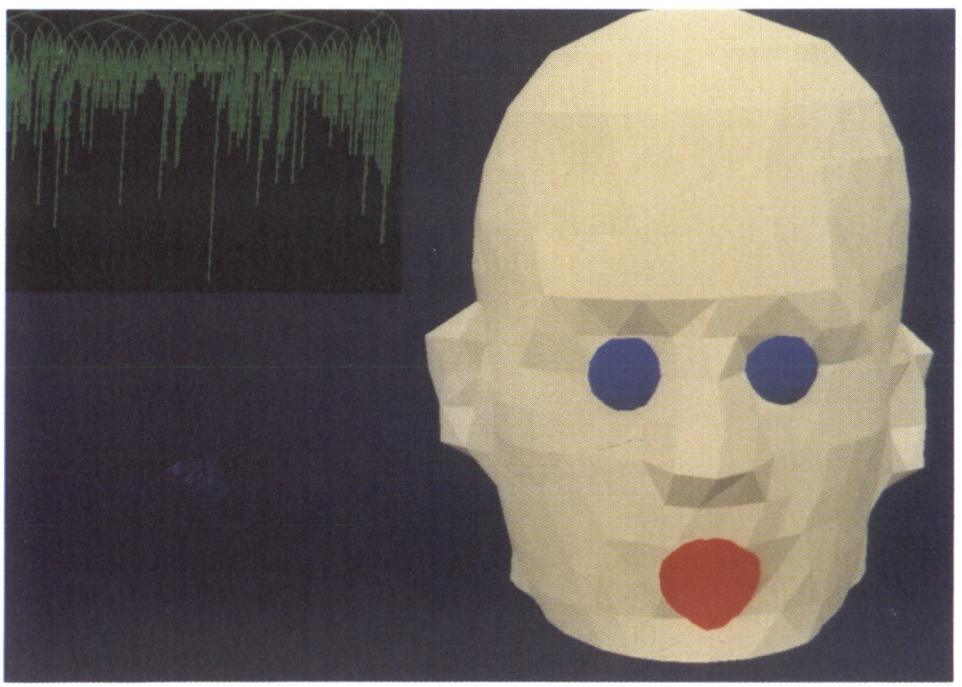


Figure 6

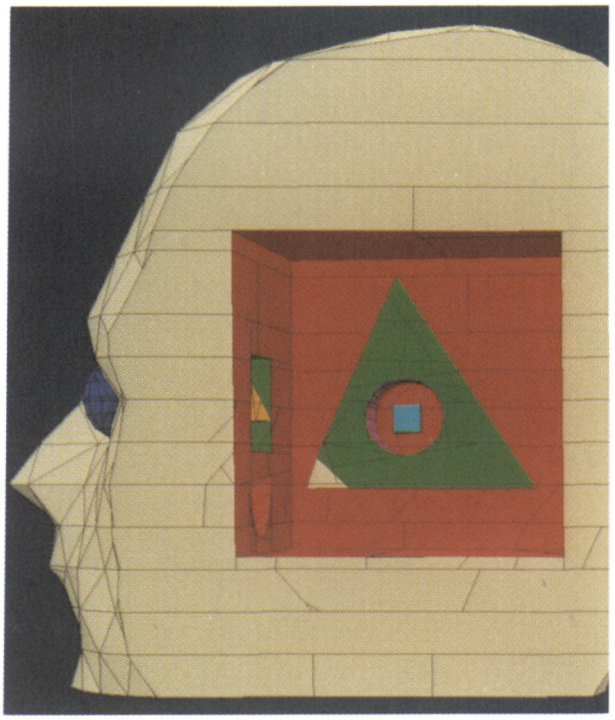


Figure 7

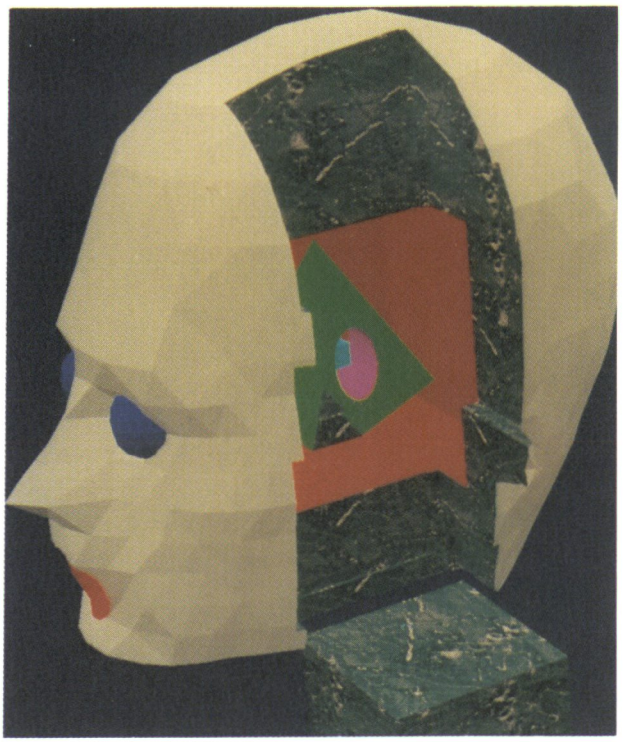


Figure 8

Figure 9