Interactive Solid Geometry Via Partitioning Trees

Bruce F. Naylor

AT&T Bell Laboratories
Murray Hill, NJ 07974
naylor@research.att.com

Abstract

The extension from interactive 2D wireframe geometry to interactive solid geometry has been for some time one of the goals of Computer Graphics. Our approach to this objective is the utilization of a computational representation of geometric sets that we believe is better suited to geometric computation than alternatives inherited from mathematics. This representation is the binary space partitioning tree. Employing such a representation leads to simpler and faster algorithms and has enabled us to construct an elementary interactive solid geometry system which can execute effectively on workstations with no graphics acceleration hardware. Interactive manipulation of a collection of polyhedral objects is provided utilizing set operations, affine transformations, collision detection, picking and dragging, calculation of metric properties, rendering of transparent objects, and solid clipping to an arbitrary polygonal view volume. This is the first such system based on partitioning trees, and as a consequence, is the first interactive system that fully supports set operations, collision detection and transparency for arbitrary polyhedral objects.

Introduction

Real-time manipulation of 3D geometry has since its inception been one the objectives of computer graphics. But only limited success has been made in achieving this goal in the almost three decades that have passed since the creation by Ivan Sutherland of the 2D interactive design program SketchPad. We believe one of the crucial factors for this is the representation. Given any semantic domain, such as geometry, for which we wish to construct a computational object, the selection of the representation largely determines the algorithms required for implementing the operators of the domain. A good representation should yield simple and efficient algorithms, but this will happen only if the representation somehow captures the computational aspects of the semantic domain. While it is possible to invent new data structures optimized for each new problem, this approach, if used in the creation of an integrated software system, is quite impractical. It is also impractical if one is interested in hardware acceleration and/or parallelization. Rather support of only a few important representations is plausible. Such is the case for numbers, where we have only a few integer and floating point representations.

Boundary representations (b-reps), essentially inherited from mathematics, represent polyhedra in terms of topology: i.e. connected components and incidence. No where is the notion of a process, i.e. computation, overtly manifested, and the view is one of an empty space populated by objects. In contrast, the *binary space partitioning tree*, also *bsp tree* or *partitioning tree*, induces a structure on space that reflects the information density of the geometry in that space. It does this via recursive subdivision of space by hyperplanes which results in a hierarchy of convex regions. The combinatorial representation of this is a binary tree, which can also be interpreted as a computation graph or decision tree. Thus computation is intrinsic to the representation. Partitioning trees provide an efficient method of answering spatial relationship questions such as those needed for set operations and visibility determination; in contrast, topological representations have, by abstraction, removed this information from the representation since homeomorphisms do not change its structure.

Partitioning trees originated as a method of pre-processing a polygonal database to facilitate hidden surface removal (see [Schumacker et al 69] and [Fuchs, Kedem and Naylor 80]). Their first use for interactive viewing of solids was described in [Fuchs, Abrams and Grant 83]. In [Thibault and

Naylor 87] partitioning trees were extended to provide representations of polytopes, and algorithms were presented for converting a CSG expression on b-reps into a partitioning tree and for performing a set operation between a partitioning tree and a b-rep by modifying the partitioning tree. This later capability was the basis for an interactive solid modeling program in which the user sculpted a work-piece with a tool; the work-piece was a partitioning tree and the tool was a convex b-rep [Naylor 90]. Solid near-plane clipping was also incorporated. In [Chin and Feiner 89], a partitioning tree based shadow algorithm was presented, and its effectiveness was demonstrated through an interactive program.

The system described in this paper is a successor to [Naylor 90] and represents an entirely new software effort based on work presented in [Naylor, Amanatides and Thibault 90] which provided set operations between arbitrary polytopes via merging of partitioning trees. This new system allows any number of non-convex polyhedra each represented by a partitioning tree. Set operations can be performed between any pair; thus the restriction in [Naylor 90] to a single non-convex work-piece operated on by pre-defined convex tools is removed. In addition, arbitrary affine transformations can be applied to any object and metric properties, such as volume and center of mass, can be calculated.

The system described in this paper is the first interactive system using only partitioning trees for representing polyhedra. Two significant benefits, in addition to set operations, accrue from this: inexpensive collision detection and non-refractive transparency. We believe our system is the first interactive solid geometry system with such capabilities.

User View

The user is presented with a world composed of a set of polyhedral objects, with access to this world through a user-interface built upon a 3-button mouse and pop-up menus. Interactive control of objects and views for models comprised of several thousand faces is possible, even on workstations with no graphics drawing hardware.

The geometry of any object can be modified by applying affine transformations or by performing set operations between it and other objects. Visual attributes, such as color, opacity and texture, can be selected as well. Measurement of any object is also available: this includes computing its volume, mass, surface area, center of mass, moments of inertia, and axis-aligned extents. The object editing operations include choosing a new object from a pre-defined set, copying an object, deleting an object, reading an object from a file, and writing an object to a file. Finally, its combinatorial representation can be displayed (a binary tree).

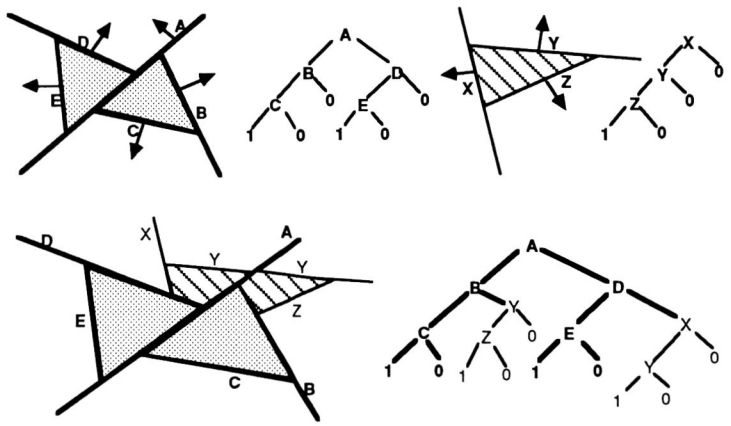
There is at any one time a single "current" object that is the operand of all unary operations on objects, and we refer to this current object as the *tool-object*. The user may at any time select any of the objects as the tool-object via picking (with the mouse). The tool-object is also one of the two operands to any set operation, the other operand being any object in which it is in contact. The tool-object is used as a tool to modify these "in-contact" objects, but the tool-object itself is unaffected. A special sweep mode is provided in which the tool-object is subtracted from each in-contact object as the user moves the tool-object.

Set operations

All polyhedra are represented by partitioning trees, and set operations are performed by merging trees [Naylor, Amanatides and Thibault 90]. Figure 1 gives a simple example suggesting how this could occur. Hyperplane normals point to the positive halfspace which is associated with the right-child of an internal node and the negative halfspace is associated with the left-child. At the leaf-nodes, a 1 indicates an in-cell and a 0 an out-cell. Set operations are achieved by merging two trees. This can be thought of in terms of inserting one tree into the other via a recursive algorithm. At each internal node **v**, the inserted tree **T** is split by the hyperplane associated with **v**, and the positive and negative "halves" of **T** are inserted into the respective subtrees of **v**. This continues recursively until a cell is reached. If the operation is, for example, union, then if the cell is an in-cell, the inserted tree is discarded; otherwise, being an out-cell, the cell is replaced with the subset of **T** that has reached/lies-in this cell.

Besides the traditional set operations of union, intersection and difference, we have included a symmetric difference corresponding to the boolean not-equal or exclusive-or. The result of this operation when viewed from the outside looks like union, but where the two objects intersect, a cavity is created which may be seen after subsequent set operations, or temporary cut-aways, or whenever the result is semi-transparent.

As suggested in Figure 1, each in-cell of a partitioning tree has an associated set of *attributes* defining the value of space within that cell (the polka-dots and cross-hatches). Since we are modeling physical objects, we use attributes describing those physical properties of the material being modeled that are relevant to our geometric operations. Thus color, opacity, density, etc., are specified for each in-cell, but not for out-cells. Polyhedra are then viewed as functions from 3-space to an attribute space in which the domain of



A union-operation using BSP Trees
Figure 1

the function is confined to the polyhedra. That is, for polyhedron **P**, point **X**, and attributes **A**, we have the function $\mathbf{f} : \mathbf{X} \in \mathbf{P} \rightarrow \mathbf{A}$, and for $\mathbf{X} \notin \mathbf{P}$, **f** is undefined (or maps to the NULL value).

When performing union or intersection, two possibly different attributes are defined wherever the two objects intersect, but the result must specify only one set of attributes in any region of space (i.e. we want a function). We currently have adopted the policy of attribute precedence: the resulting attribute is simply that of the operand with greater precedence, and we always give precedence to the first operand (it seems oddly convenient that the two operations for which this is an issue are commutative). An alternative policy would be to construct a new attribute from the blending of the two operand attributes.

We have also provided support for *transparency*; and by this we mean the thin-film variety, as with colored cellophane, that can be treated as an absorbing but not refracting medium. (This is also to be distinguished from "screen-door" transparency provided by some z-buffer based systems.) The requisite blending calculations are performed during polygon scan-conversion using an "alpha-channel" to hold the opacity of a point on those workstations having such capability. Since the in-cells of a partitioning tree may differ in their material attributes, and since reflection and refraction are considered to occur wherever the electromagnetic properties of space change discontinuously, visible boundaries can be present in the interior of a polyhedron, unlike opaque polyhedra. What is more, such a boundary can be visible from both sides, although with differing filtering properties, if the intervening space is sufficiently transparent. (The boundary between a red region and a green region will appear, if looking from the green region as red, but will appear as green if looking from the red region.)

The consequence of this is the need for explicitly creating *interior faces* with the appropriate attributes for both sides of each face; and this we do, although only for rendering environments that can support transparency. While the creation of interior faces would seem to be unnecessary whenever both sides of a boundary are fully opaque impling the boundary is never visible, any subsequent reduction in the opacity will reveal the interior boundaries. Thus we do not make this optimization.

Another geometric operation of considerable utility that we have incorporated is *collision detection*. While the temptation is to think of collision detection as merely intersection, we take the view that unlike intersection the returned value is essentially symbolic rather than geometric. Consider two "macro objects", molecules for instance, that are created from the union of many individual components, such as atoms and bonds. When we answer the collision question, we generate a list of all pairwise collisions between the components from the two macro objects. That is, if the component names (implemented, for example, as pointers to their representation) are considered to serve as the labels for nodes of a *collision graph*, then what we produce are the arcs of this graph. (The arcs are directed, going from a component in the first operand to a component in the second; thus the name space of the two operands need not be disjoint.)

The advantage of this symbolic approach is that it separates the mechanism of collision detection from the policy of what action to take as a consequence of any collisions. Below in the section Interactive Techniques, we describe two different policies for two different circumstances, neither of

which is concerned with the geometry of the intersection. Such a separation is made affordable because we perform collision detection as a small additional overhead to any set operation, and as we discuss below in the section Modeling, we use union to create the model each time the tool-object moves. The cost for collision detection is small because the algorithm for any set operation recurses until at least one operand is reduced to a cell. If this is an in-cell, then the additional work is no more than searching the tree of the other operand for other in-cells (containing identifiers) and the appropriate collision arcs are generated. We gain some efficiency by maintaining an identifier at any internal node whose region contains only one identifier (a list of identifiers at internal nodes is another possibility). The arcs are maintained as a set; that is, there are no duplicates.

Affine Transformations

Affine transformations are an elementary geometric operation whose implementation has customarily been treated as synonymous with 4x4 matrices. However, notable benefits can be gleaned if they are treated in a more sophisticated way. An important motivation for this is the need to provide an interactive user with intuitive ways to modify an object using affine transformations. Traditionally, we think of transformations being applied in the order in which they are specified; this is typically achieved by maintaining a composite matrix so that a new transform modifies the composite. While this policy is indeed the right one in many situations, it is not ideal for interactive object modification.

Instead, we have found from subjective experience that the primitive transformations should be applied in the following order: scaling, shearing, rotation and translation. Note that the transforms which modify distance, i.e. scaling and shearing, appear before the rigid body transforms. Consequently, we maintain each of the four primitive transformations separately and multiply them together each time their composite value is demanded, replacing the previous composite matrix. So for instance, a rotation operation modifies only the rotation matrix rather than the composite matrix. An additional consideration is the fact that all linear transforms have fixed points: the origin plus possibly some subset of coordinate axes. Changing these fixed points is achieved by employing an additional change-of-basis matrix. Thus the composite matrix resulting from the product of the primitive transforms is pre-multiplied by the change-of-basis matrix and then post-multiplied by its inverse. One then has the freedom to choose any local coordinate system for an object.

A second advantage of implementing an affine transformation class as something different from a matrix is the ease with which optimizations can be performed transparently. The greatest disparity among the primitive transforms is the much greater speed at which translation can be effected. Since translation appears in an interactive environment with considerably greater frequency (moving objects about), detecting when an affine transformation is no more than a translation is an effective and simple optimization. Similar considerations also lead to determining when transformed normals need re-normalization. Rigid body transforms do not require this and symmetric scaling necessitates only divisions, not the calculation of the length of the new normal which entails computing a square root (profiling confirmed that indiscriminate re-normalization resulted in significant overhead). A final optimization is the generation of composite and inverse matrices only on demand rather than after every change to the affine transformation. The routines that modify affine transformations mark them as not up-to-date; and routines that use them for transforming points, normals and hyperplanes first requests an up-to-date composite and inverse matrices, which are generated only the first time they are demanded (if the affine transformation is not utilizing the pre-defined ordering of primitive transforms, the composite will already be up-to-date, so only the inverse needs to be computed).

Picking

A classic operation dating to the days of random-scan vector-refresh displays is that of picking an on-screen object using the current cursor position (originally a light pen). We achieve this by ray-casting. In screen-space the cursor position provides the initial x and y values of the ray origin, and its z-value is the screen-space position of the near clipping-plane (0 in our system). The ray direction is that of a screen-space projector whose length is the distance between the near and far clipping-planes in screen-space ([0 0 1] in our system). The ray, with $t_{min} = 0$ and $t_{max} = 1$, is then mapped to model-space by the inverse of the model-to-screen space transformation. We can now use the ray-tracing algorithm for partitioning trees [Naylor and Thibault 86] that finds the first intersection point within t_{min} and t_{max}. This returns the model-space intersection point, the surface normal, the near and far attributes, the classification of the ray segment from the origin to the intersection point (in or out), and most importantly the identifier. Generating an equivalent "pick report" using standard methods usually entails clipping the entire model to a tiny

window. Our method reflects directly the nature of the query and is more efficient due to the partitioning tree's search structure.

Modeling

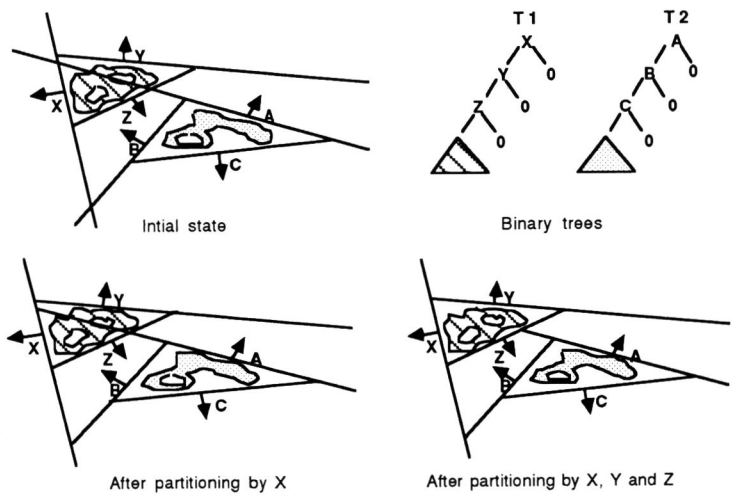
The representation of each user-level object entails three geometric sets, each represented by a partitioning tree. In particular, user-level objects are represented by an object-instance class containing the object definition, its value prior to the last set operation, an instance of the definition, and an affine transformation which determines the instance in terms of the definition (along with some minor state information).

As we mentioned in the user-view section, the model is a set of user-level objects one of which is distinguished as being the tool-object; the remaining objects we refer to as the *static-objects*. Any time a new tool-object is selected, a union of the static-objects is created as a distinct geometric set (represented by a single partitioning tree). A copy of each object's instance is used for this. The model is then the union of two sets: a copy of the static-objects and a copy of the tool-object. Any time the tool-object is modified, say by moving it, the previous model is discarded and a new one constructed; but note that the static-objects representation remains the same and so is reused as long as this set does not change (by set operations or picking a new tool-object).

Since we are using this in an interactive environment, these operations must be relatively quick. Two techniques are used to accelerate their execution. The first is the construction of good partitioning trees [Naylor 92]. This subject is somewhat complex, and so we will not address it here. But the essence is that we use expected case models to build partitioning trees that represent an object something like a sequence of approximations. An artifact of this process is the creation of a first level approximation that corresponds to the classic technique of bounding volumes. Figure 2 illustrates the effect of this when forming the union of two disjoint sets; in this case the computation is equivalent to forming the union of the bounding volumes, and so is very fast.

The second technique is a classic one: reference counts. Copying a tree has the effect initially of only incrementing the reference count to the root of the tree. Actual duplication of a tree node occurs only when it or its subtrees are modified. During set operations, any subtree of one tree lying within a cell of the other tree does not need to be duplicated in order for it to be included in the result.

Returning to figure 2, if the two trees were intended to be copies, only the first three nodes of each tree will be duplicated, while the subtrees indicated by filled triangles would remain untouched. Thus copying and discarding can require only sub-linear time. Reference counts are used as well for copying of polygons and attributes. One consequence of this for attributes is that the same allocated attributes will be shared by the object definition, its instance, and the copy of the instance used to form the model. Thus modifying the attributes of the object definition also achieves this modification for its instance and the copies of the instance present in the static-objects tree and in the model tree without the need for any additional work (other than rendering the new image).



Union of disjoint objects
Figure 2

Rendering

Given a partitioning tree representation of a collection of sets, for any viewing position a total visibility priority ordering of the sets can be generated by a single view-dependent traversal of the tree [Fuchs, Kedem and Naylor 81]. Since we have constructed a single partitioning tree representation of the entire model, we can solve the visible surface problem by performing the ordering operation on this tree.

There are several advantages to using the partitioning tree for visibility instead of the commonly used depth-buffer algorithm. First, the ordering is generated in model-space as opposed to the discrete, post-perspective screen-space. Thus, no depth-buffer is needed to represent this discrete space and surface continuity (coherence) is exploited to avoid performing ordering calculations for each pixel. This has contributed to the ability of our system to provide interactive solid geometry on workstations with no graphics hardware per se and requires only the presence of 2D convex polygon drawing routines. In addition, the information loss due to the perspective division using floating-point is avoided. Thus, for sets which intersect, or nearly intersect, their ordering will not vary incorrectly in the neighborhood of their intersection when small changes in the viewing position are made (this is especially a problem with polygons that are close and almost parallel). One consequence of this is that the edges of a polygon can be drawn "on top of" the polygon without any subsets of their discrete representation being occluded.

A second class of advantages of a visibility ordering arises from the desire to render non-refractive transparent objects and to perform anti-aliasing. Both of these operations can be implemented at the pixel level by blending the colors of two pixels. Given only two polygons (or pixels) $\mathbf{p_1}$ and $\mathbf{p_2}$, in which $\mathbf{p_1}$ has color $\mathbf{c_1}$ and opacity α_1 and occludes $\mathbf{p_2}$ which has color $\mathbf{c_2}$ and opacity α_2, then the resulting color is $\mathbf{c_{1,2}} = (\mathbf{c_1} * \alpha_1) + (\mathbf{c_2} * \alpha_2) * (1 - \alpha_1)$ and opacity is $\alpha_{1,2} = \alpha_1 + (1-\alpha_1) * \alpha_2$. Anti-aliasing using a box filter can be treated analogously if the polygon is modeled as covering the entire pixel but with opacity equal to the area of the pixel covered by that polygon (this is in lieu of computing the visible surface at the sub-pixel level using masks). The correct use of this technique requires that blending occur only between pixels that are consecutive in the priority order, which cannot be accomplished with the standard depth-buffer algorithm. However, a multi-pass two-buffer algorithm is known [Mammen 89] where the number of passes equals 1 + the maximum number of overlapping transparent polygons. While transparency can be performed with either a far-to-near or near-to-far priority ordering of the partitioning tree, anti-aliasing requires the near-to-far ordering in order to avoid blending in occluded surfaces. Such an ordering is also needed for maintaining sub-pixels masks, if available, that are required for higher fidelity anti-aliasing, and for avoiding the calculations at fully occluded pixels (when $\alpha_1 = 1$) required by texture mapping or Phong shading.

Another major component of the rendering operation that can be achieved simply and efficiently using partitioning trees is clipping. View-volume clipping is nothing more than forming the intersection of the view-volume with the model. Thus, we generate a partitioning tree representation of the view volume and execute the standard partitioning tree intersection algorithm, but with the following optimization. Since the results from clipping will not, in an interactive environment, be used in any subsequent set operations, there is no need to produce the 3D geometry of the intersection explicitly. Only the intersection of the model's faces with the view volume is required. Consequently, the operation does not modify the tree representing the model and so avoids all copying accept for those faces that intersect the boundary of the view volume.

There are three advantages to partitioning tree clipping. First, the efficiency of set operations is enhanced by the expected case performance of the partitioning tree when thought of as a search structure. Roughly speaking, one might expect for $\mathbf{n}$ faces $\mathbf{O(\log n)}$ operations instead of the $\mathbf{O(n)}$ of the standard implementation of clipping. For example, any object that is entirely inside or entirely outside the view-volume will be clipped by "visiting" only the top level nodes forming its bounding volume. Thus, culling is achieved without any explicit notion of culling just as there is no explicit notion of bounding volumes (and so no code to implement them); it arises from building good trees for efficient set operations. This contributes to the effectiveness of the system when used on standard workstations. Secondly, near-plane clipping does not reveal a polyhedron as an empty shell but rather maintains the semantics of solids. Not only are the faces of the view-volume lying within objects displayed, but they have the attributes of the region which they bound. (Note that b-reps do not typically have cells with attributes required by this.) This permits near-plane clipping to be used to provide cut-aways as a visualization aid. Thirdly, the partitioning tree clipping algorithm supports arbitrary polyhedral view volumes just as easily as the traditional truncated pyramid. This can be put to good use in an environment with multiple overlapping viewports with simultaneous renderings in each. Portions of a model occluded by another viewport can be clipped away by using the appropriate view-volume.

Interactive Techniques

Given the above capabilities, we have chosen several interactive techniques that enhance the usa-

bility of the system. One is the use of collision detection to provide highlighting via transparency. Forming the union of the tool-object with the static-objects, like any set operation, can generate a collision report. We can exploit this so that whenever the tool-object makes contact with any object in the scene, the contacted objects have their opacity reduced, and then subsequently restored when contact ceases. For rendering systems supporting transparency, this has the effect of not only indicating contact, but also allowing the user to see the tool-object as it moves through the interior of the contacted objects. For rendering systems without transparency, the reduction in opacity appears instead as a reduction in luminance, since the percentage of reflected light decreases with a decrease in opacity. In such environments, using the near-plane clipping as a cut-away can reveal the position of the tool-object as it moves through the interior. (Note that this visualization technique requires that of the tool-object's attributes take precedence over the static-objects when forming the model.) Cut-aways also provide a means of inspecting the otherwise occluded interior parts of objects.

A second use of collision detection is for operand selection in the execution of a user specified set operation. The set operation is performed between the tool-object and only those objects with which it is in contact. This method is, from the user's perspective, simple and intuitive. Not having some method of selection is unacceptable since, for example, intersection between the tool-object and objects with which it is *not* in contact would annihilate those objects.

Those objects which are modified by a set operation will also have their center of mass and principal axes recomputed, after which their object definition is translated and rotated so that the center of mass and the principle axes form the object's local coordinate system while the instance reamains stationary. Thus affine transformations are always with respect to an objects "natural" coordinate system. This is very important if one wishes to maintain an intuitive sense of the effects of affine transformations when applied to objects whose geometry has been significantly modified by a set operation.

Picking provides a direct geometric method of tool-object selection, and implementing this with ray-casting makes the selection precise. Dragging of the tool-object is achieved by mapping the translation vector defined in screen-space into model-space. This requires that the screen-space vector have a depth, which the mouse does not provide. For this, we use the screen-space depth value of the picked surface point. As a consequence the picked position always tracks the cursor exactly (as long as the view is not changed). A similar operation is placing, in which the user picks a source point on the tool-object and then picks a target point on any other object, while during the interim changing the view if necessary to make the target point visible. The source point is then translated to the target point.

Finally, we use a variety of other minor techniques to improve usability. Tool-object translation and view rotation via mouse translation is always with respect to screen-space; so, for example, left-right mouse motion maps to horizontal motion in a plane parallel to the screen. We also adjust the view rotation rate as a function of window size so that the subjective sense of the rate remains constant. Symmetric scaling of the tool-object increases as a constant percentage of the current scale factor, while differential scaling changes by a constant step size independent of the current scaling (empirically ascertained preference). Delta changes generated by holding down buttons have a slight acceleration and are adjusted to compensate for varying frame update rates; otherwise they would be too slow for long update times and too fast for short ones. And shearing preserves basis vector length rather than volume; otherwise surface area increases unboundedly even though the volume remains the same.

Comparison to Buffer Algorithms

The ability to transform and merge partitioning trees quickly has obviated the original restriction in which trees were only created during a pre-processing phase. Since affine transformations and perspective projections (or more generally d+1 dimensional linear transformations) do not change a tree's structure, off-line generation of good trees leads to efficient merging of instances of these trees, resulting in reasonably good new trees. Thus, we have only relaxed, not abandoned, the original pre-processing context, and this is a crucial point. For the original idea was to exploit time-invariant properties of the geometry to reduce computation. Thus complex but affinely invariant objects can still have their trees created once as a pre-processing step. Interactively created objects can also have their trees improved by off-line reconstruction. However, this investment can now be reaped in a much less restrictive environment, since these trees can be transformed and merged.

As the system described in this paper illustrates, the pre-computation yields efficient set operations, collision detection, clipping, visible surface determination, transparency, anti-aliasing and picking. Let us compare these capabilities to the screen-space approaches for polyhedral objects which are extensions in the spirit of the z-buffer algorithm (as opposed to scan-line algorithms). In these methods, typically no pre-computation is utilized and each object is sampled independently. Screen-space evaluation of set operations requires a representation of the geometric model in every 1D sub-domain defined by the projector for each pixel (or sub-pixel); this, of course, is the same methodology as ray-cast evaluation of CSG models. Assuming retention of the scan-conversion modus operandi, as opposed to ray-tracing, this requires

decomposing an object into a set of segments with attributes and identifier (for collision detection) instead of simply pixels. For convex primitives, this is not difficult, but for arbitrary polyhedra, no efficient scan-conversion type algorithm is known. The resulting segments must then be merged and evaluated as defined by the CSG tree (note that set operations require knowing the value of an operand in *every* projector, not just those intersecting the interior as would be discovered by scan-conversion). This will yield an ordered list of segments that can, for the purposes of transparency, be composited. The accuracy of this computation is affected not only by the sampling, but also by the rather considerable non-linear compression of the depth. This is in addition to the difference in accuracy of discrete screen space vs. continuous model space computations; a point of some importance in non-interactive geometric computation, such as interference detection and mass property calculations. Anti-aliasing requires all of this work to be performed at the sub-pixel level if a correct image is to be generated, adding a factor of 4 or 16 increase in the per pixel computation. Clipping is O(n), although this can be reduced by an initial culling of objects by comparing their bounding box to the view-volume. However, clipping unculled objects is still O(n) and the view volume is restricted to being convex, so that in a window system, subsets of the model occluded by another window would not be eliminated by clipping (however, solid clipping, i.e. cutaways, could be achieved with this proposed system). Picking has often been implement as an additional O(n) clipping step as opposed to our ~O(log n) ray-cast, although with this proposed schema, the necessary information would already be present in the buffer.

Having done all this, the result is an image identical to the one generated using partitioning trees. However, evaluation of set operations would be repeated for every frame at every sub-pixel, even though an object created using set operations may be affinely invariant for an arbitrarily large number of frames (not just seconds, but possibly days or years). So let us assume that screen-space algorithms are not used for set operations. Then for collision detection and transparency, we still must have the ordered set of segments for each sub-pixel, since we are effectively computing the union of the objects. The comparative cost of this operation suffers both from the absence of pre-processing acquired temporal-correlation and from the non-exploitation of spatial-coherence: instead of determining spatial relations between volumes, the same relation will be repeatedly computed for every ray that intersects those volumes (scan-line algorithms, of course, can exploit coherence). Parallelization of the buffer-based approaches is not necessarily a panacea, since partitioning tree algorithms can be parallelized as well. Only in circumstances where the presupposition of affine invariance over a sufficient period of time is violated, which is commonly the case for cartoon like animations, would the buffer-based approaches be arguably superior for polyhedral models.

References

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

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

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

[Mammen 89]
A. Mammen, "Transparency and Antialiasing Algorithms implemented with the Virtual Pixel Maps Technique," **IEEE CG&A** Vol. 9(4), pp. 43-55, (July 1989).

[Naylor and Thibault 86]
Bruce F. Naylor and William C. Thibault, "Application of BSP Trees to Ray-Tracing and CSG Evaluation," GIT-ICS 86/03, School of Information and Computer Science, Georgia Institute of Technology, (February 1986).

[Naylor 90]
Bruce F. Naylor, "SCULPT: an Interactive Solid Modeling Tool," Proceeding of *Graphics Interface* , pp. 138-148 (May 1990).

[Naylor, Amanatides and Thibault 90]
Bruce F. Naylor, John Amanatides and William C. Thibault, "Merging BSP Trees Yields Polyhedral Set Operations," **Computer Graphics** Vol. 24(4), pp. 115-124, (Aug. 1990).

[Naylor 92]
Bruce F. Naylor, "Constructing Good Partitioning Trees," manuscript in preparation.

[Schumacker 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), pp. 153-162 (July 1987).