A Methodology for Geometric Algorithm Development

Seth J. Teller

ABSTRACT

Much current development of geometric algorithms is performed in a batch fashion, even though the algorithms themselves are often intended for highly interactive and visual applications. This predominance of batch techniques has resulted in lessened intuition about, experimentation with, and productivity of geometric algorithms.

We describe a methodology for the development of robust geometric algorithms. First, we propose that algorithm development should be, and can be, performed in a visually and interactively rich environment, and that this environment facilitates robustness, intuition, and pedagogy. We relate some experience that bolsters our claims of increased robustness. Second, we argue that geometric computations should be designed to "prove" the validity of their conclusions to the practitioner via data objects called *geometric witnesses* that have direct visual representations. Third, we show that an interactive "template" program for algorithm development is easy to construct on currently available workstations, and that it is well worth it to do so. Finally, we enumerate several other development principles, touching on interaction techniques and the use of temporal coherence to detect subtle algorithmic flaws.

Key Words: Algorithm visualization, geometric computation, robustness, interaction techniques, geometric witnesses.

1 INTRODUCTION

Practitioners of computer graphics, computational geometry, computer-aided geometric design, and related fields routinely design, implement and use sophisticated geometric algorithms. In contrast to the eventual ways in which these algorithms are *used* (typically in complex modeling or geometric visualization or simulation applications), the environments in which such algorithms are *implemented* are often relatively unsophisticated. One reason for this phenomenon may be that interaction techniques are widely (and erroneously) thought to be "expensive;" that is, to require many person-hours of implementation effort and an expensive hardware platform. Thus the activities of implementing the components of the system, and of using the system, tend to remain separate in practice.

Geometric Algorithms Are Often Developed in Batch Mode

A specific example of the problem might be a computational geometry practitioner developing a convex hull algorithm. The practitioner writes an algorithm that reads a file of input points, computes the convex hull, and writes a textual representation of it to another file. This file is then read by another program that displays the points and the hull described in the file. If a problem is observed, the code is modified, and another iteration of execution and inspection begun.

This procedure is essentially a batch computation, since it runs from start to finish with no

interaction. The convex hull algorithm has no access to the graphical environment in which the algorithm is "used" (i.e., its output displayed). This separation tends to have at least two deleterious effects. First, the slowness and rigidity of the development process tends to discourage experimentation and intuition. Second, algorithms developed in this manner tend to have bugs that show up "later" than is desirable (e.g., after the algorithm is subsumed into some larger application), for the simple reasons that relatively few test inputs to the algorithms are generated and exercised, and the algorithm is not tested in the fashion in which it will be used (i.e., repeatedly). Since exhaustive enumeration of all inputs is computationally infeasible, even if systematic or randomized testing is performed, there is no reason to suppose that it will find troublesome input cases. In short, batch computations discourage intuition and experimentation.

A New Way of Developing and Testing Geometric Algorithms

In this paper we propose that the design and use of geometric algorithms be combined in a single interactive environment. By this, we do not mean that development and application tools be combined into one monolithic application, but rather that algorithm development should be, and can be, performed in a visually and interactively rich environment. Moreover, the availability of such an environment makes possible a novel methodology for algorithm development. We have constructed an algorithm visualization system in which the practitioner interacts with visual representations of the input and output of a geometric algorithm, which is re-invoked whenever the input is modified (Figure 1). The environment was straightforwardly constructed on an existing workstation, and is a "template"; it can be easily reused whenever a new geometric algorithm is to be developed.

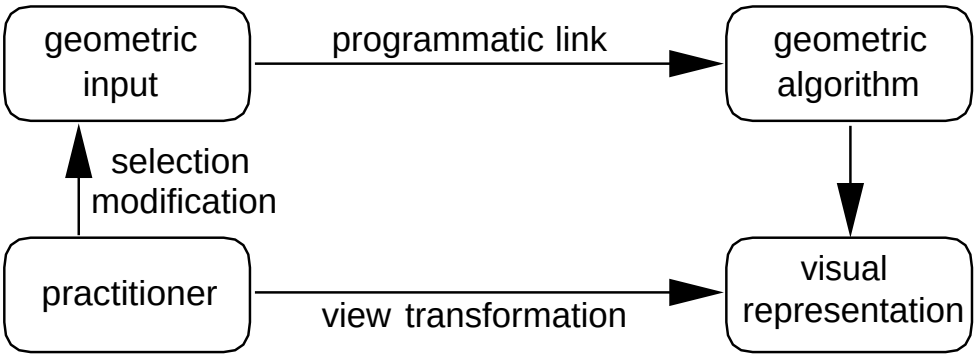

Figure 1: The interaction model. The practitioner modifies the geometric inputs through selection and direct manipulation. The visual representations of both inputs and outputs are displayed and inspected via directly-manipulable viewing transformations.

Specific Recommendations for Practitioners

Interactive operation allows efficient, high-level directed search of a many-dimensional input space, and (with suitable techniques such as alignment and snapping (Bier 1990) allows easy generation of highly degenerate input. The combination of these techniques probably bests stochastic or (non-exhaustive) systematic search of the input space, since it can be interactively directed by the practitioner.

In addition to our recommendation that batch techniques be replaced by interactive methods, we make the following recommendations:

- **Geometric Witnesses.** Geometric computations should be designed to prove their conclusions with visually representable data objects. These objects should be computationally verifiable by some agent other than the original algorithm.
- **Integrated Development and Use.** While under development, The algorithm inputs should be manipulated with a user interface at least as sophisticated as that of the algorithm's intended application.

- **Interaction techniques.** All geometric inputs should be mapped to a directly observable and manipulable representation. Moreover, the *space* embedding the problem instance should be directly manipulable (i.e., via viewing transformations). Interaction should be direct and non-modal to the extent possible. This allows any aspect of the input to be examined, regardless of scale or position.

- **Data Invariants.** Valid properties of data should be checked at input and enforced by all subsequent operations to maintain a correctness invariant.

- **Temporal Coherence.** A rapid succession of interactively generated images allows the visual system to detect errors that are almost impossible to notice in single-frame renderings.

We argue that the construction of a simple, but powerful, interactive environment, and the adoption of these principles regarding algorithm design, visualization, and interaction techniques, will substantially improve the practitioner's intuition about geometric algorithms, and the robustness of the algorithms themselves. Moreover, we argue that the general opinion of interactive environments as "too hard" to construct is in error, and perhaps due more to the enormous amount of bad system software than to reality.

Geometric Algorithm Development is Analogous to Scientific Visualization

Other investigators have developed interactive systems that are used, for example, to visualize scientific data. A strong analogy can be made between scientific visualization and interactive algorithm development. Scientific visualization is a method by which attributes of complex data are transformed into a visual or aural representation. The primary goal of the representation is to make these data primarily accessible to the high-bandwidth sense organs of a human observer, in order that the physical *processes* by which the data are generated can be better understood. Analogously, interactive algorithm development and inspection is a technique in which a space of algorithm *inputs* is navigated and continuously subjected to a geometric computation – the algorithmic process. The goal of the interaction is to understand, verify, and perhaps extend the computation. The computation is the analogue of the physical process.

Motivation From a Real-World Visual Simulation System

Our need for robust geometric algorithms operating on real-world data arose as part of a group development effort of a system for constructing and simulating extremely complex architectural models (described extensively in Teller, 1991; Khorramabadi, 1991; Funkhouser, 1992; Teller, 1992b). The system, based on a techniques for three-dimensional spatial subdivision, visibility computation and database management, achieves interactive display rates for models that, without such techniques, would require several seconds or even minutes to render even on the fastest workstations currently available. The fact that the algorithms are in daily use for actual design, evaluation, modification, and visual simulation of a complex architectural model served as a strong incentive to make the algorithms quite robust.

This paper presents several examples of the use of the proposed methodology during the conception, development, and verification of complex geometric algorithms. Throughout, the examples illustrate the value of a graphical environment in visualizing the operation of an interactive geometric algorithm, and how this visualization affords verification that the algorithm is operating correctly.

Geometric Witnesses

The notion of "geometric witnesses" is central to the methodology that we propose. The term "witness" has long been used among computational geometers to connote a data object that evinces the outcome of a geometric computation. These computations are typically *existence* computations, in that they determine (for a given input) whether or not there exists a geometric

datum satisfying some collection of constraints. Perhaps the simplest example of a geometric witness occurs for the computation of an intersection between two line segments in the plane. If the segments intersect, a witness to their intersection is a point lying on both segments. If the segments do not intersect, a witness to this fact is a line separating the two input segments. In either case, the witness serves as visual "proof" that the answer computed by the algorithm is correct. Moreover, the witness can be computationally verified by an agent other than the intersection routine: the witness point can be compared against the interiors of the input line segments, or the two halfspaces of the witness separating line can be compared against the segment endpoints.

Many of the details of the geometric algorithms developed for the walkthrough system are outside the scope of this paper. However, the algorithms can be characterized by the fact that they require geometric input and produce geometric output. The geometric libraries were developed using an interactive graphics environment that supplies a uniform interface for algorithm inspection. A template graphics program operates solely in terms of a "region of interest" (typically a 3D bounding box), and arranges this area as the graphical center of attention, by default. To be useable in this template, a data object need only be able to report its bounding region. At run time, this region appears centered in the window, scaled to a reasonable window-filling size, with reasonable perspective, and preserved aspect ratio. The environment then enters a standard command loop in which the area of interest can be scaled, rotated, and translated via mouse actions. The command loop also supports selection and modification actions, which depend on the dimensionality and type of the input.

2 THE WALKTHROUGH SYSTEM

The architectural simulation system takes as input a building model consisting of a set of large planar polygonal *occluders*, and a set of bounding volumes of complex *detail objects* (Khorramabadi 1991; Teller 1991; Funkhouser 1992; Teller 1992b). The occluders represent the gross structural detail of the building, whereas the detail objects represent complex entities such as personal items and furnishings, which typically cause little occlusion. The space of the model is then partitioned into convex *cells*, and any non-opaque portions of shared cell boundaries are explicitly represented as convex *portals* (Figures 2 and 3). Each portal stores with it an identifier for the cell to which the portal leads. We call a spatial subdivision with explicitly enumerated portals a *conforming* spatial subdivision.

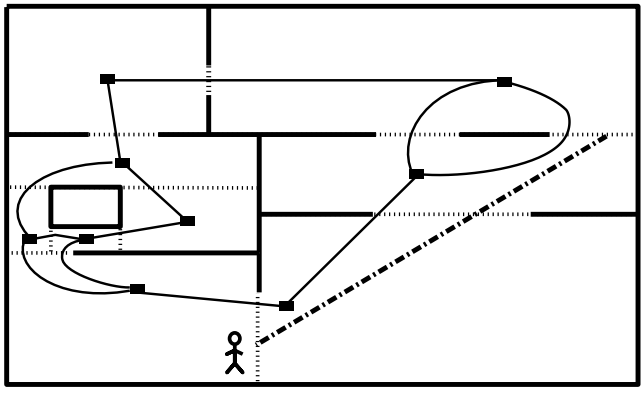

Figure 2: A conforming spatial subdivision in 2D, and its corresponding adjacency graph. Portals are (dashed) line segments. An observer is schematically represented at the lower left, and a sightline (broken) stabs a portal sequence of length three.

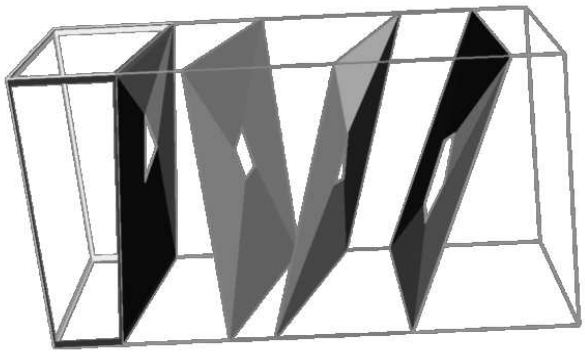

Figure 3: A conforming spatial subdivision with five convex cells in 3D. Occluders are rendered as gray polygons. Four convex polygonal portals (the white central areas in each subdivision plane) are shown.

The cell and portal abstraction for the spatial subdivision is effectively a directed graph whose vertices are cells, and whose edges are portals. A number of useful static (observer-independent)

and dynamic (observer-dependent) visibility computations can be framed as constrained *depth-first searches* (DFS) of this graph. For example, if a given portal is treated as a light source, then the region illuminated by this light source in the remainder of the subdivision constitutes an upper bound on the visibility of an observer situated on or behind the plane of the portal. This "antipenumbral" region (Teller 1992a) is exactly the bundle of lines that *stabs*, or pierces, all portal sequences originating at the light source (Figure 4).

Data Invariants

Note that the construction of a valid spatial subdivision amounts to enforcing a *data invariant* on the input model, in that during construction, all overlapping and interpenetrating polygons are found, as are any gaps or cracks between polygons. Errors found at this stage can be reported and/or corrected; the "correctness" (manifold properties) of the model can therefore be ensured at the beginning of the program, and assumed by all further operations.

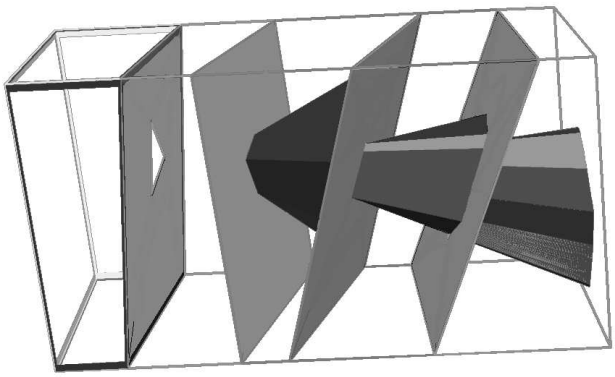

Figure 4: The regions visible to an observer constrained to the leftmost cell.

Since the illuminated region is an upper bound on visibility, only objects whose bounding volumes are spatially incident on this region can possibly be visible to an observer known to be in the source cell. In sufficiently occluded environments, therefore, significant rendering accelerations can be achieved by an offline computation of potentially visible object sets for each cell of the spatial subdivision. During a later interactive phase, the precomputed visibility data is retrieved and subjected to a more discriminating *dynamic culling* operation based on the known position and field of view of the observer. Only the objects surviving this cull need be rendered. These objects typically form a small fraction of the entire model data; we achieved rendering speedups of about one hundred using a model of a seven-story structure with an atrium, terraced balconies, scores of hallways, hundreds of rooms, thousands of textures and detail objects, and nearly a million individual polygons (Khorramabadi 1991; Funkhouser 1992; Teller 1992b).

3 A CASE STUDY

Our development environment consisted of Silicon Graphics workstations with varied computing and graphics capabilities. A shared repository of geometric libraries comprised routines for manipulating: points, vectors, planes, hyperplanes, etc.; linear programs of general dimension; planar convex polygons; convex polytopes of general dimension; spatial subdivisions; stabbing problems; and visibility data structures. Routines were designed and implemented for two-, three-, and higher-dimensional operands as needed. This paper is concerned with the

development of spatial subdivision abstractions and dynamic visibility computations, which are discussed below.

The hardware platform provided a C-language interface, the "SGI GL" or graphics library, for operations such as: specification of perspective and modeling view transformations; simulated point light sources; immediate-mode issuance of drawing commands for such primitive objects as points, wires, and polygons. Rendering is typically double-buffered for smooth animation, although single-buffering and immediate-mode graphics are sometimes useful. The hardware and software also support a graphical selection operation in which objects rendered near the mouse position could be reported to the application.

Dynamic Eye-Based Culling

The dynamic culling operation is an interactive and highly geometric computation. Given a model and spatial subdivision, and an instantaneous observer position and field of view, the dynamic culling operation must compute the set of cells and detail objects potentially visible to the observer. Moreover, the dynamic culling operation must complete in time comparable to interactive frame rates: about one twentieth of a second. In the remainder of this paper, the dynamic culling algorithm is used as an illustrative vehicle for our recommendations about algorithm development.

The precomputation of antipenumbral bundles has only approximate knowledge of the observer position, and can therefore produce only gross upper bounds on the visibility of any particular observer. During the real-time walkthrough phase, however, the simulation system tracks the precise instantaneous position and field of view of the observer. The observer variables can be modeled as an eye-centered view pyramid, defined as the common interior of four halfspaces in three dimensions (Figure 5). In the context of algorithm inspection, the observer variables form the algorithm input; the set of potentially visible objects form the algorithm output.

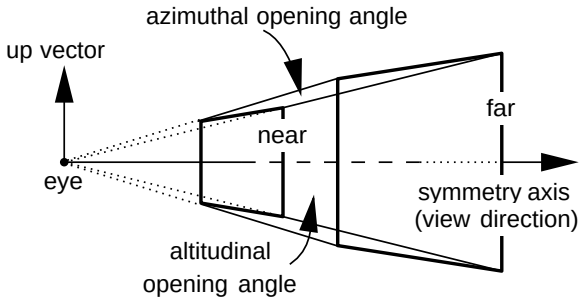

Figure 5: The observer variables in three dimensions.

Linear Programming

The dynamic culling operation employs a constrained depth-first search of the subdivision graph. A list of constraints is initialized to the four halfspaces whose intersection is the view frustum. The source cell is placed on a stack, and a recursive search begins. Each portal leading out of the current cell is considered in turn. Each portal edge encountered contributes a single linear constraint to the sightline specification; the edge and the eye span a plane, whose orientation is chosen so that its positive halfspace contains the interior of the associated portal (Figure 6). The DFS advances recursively only as long as the current portal sequence admits a sightline through the eye.

These constraints are cast as a two-dimensional linear program as follows. If the k^{th} plane constraint has normal $\mathbf{n}_k$, any stabbing line through the eye and the active portal sequence must have a direction vector $\mathbf{v}$ such that

$$\mathbf{n}_k \cdot \mathbf{v} \geq 0, \qquad \text{for all } k.$$

Note that the linear program is two-dimensional since all of the planes contain the eyepoint; therefore, we need only compute a *vector* that has a non-negative dot product with each

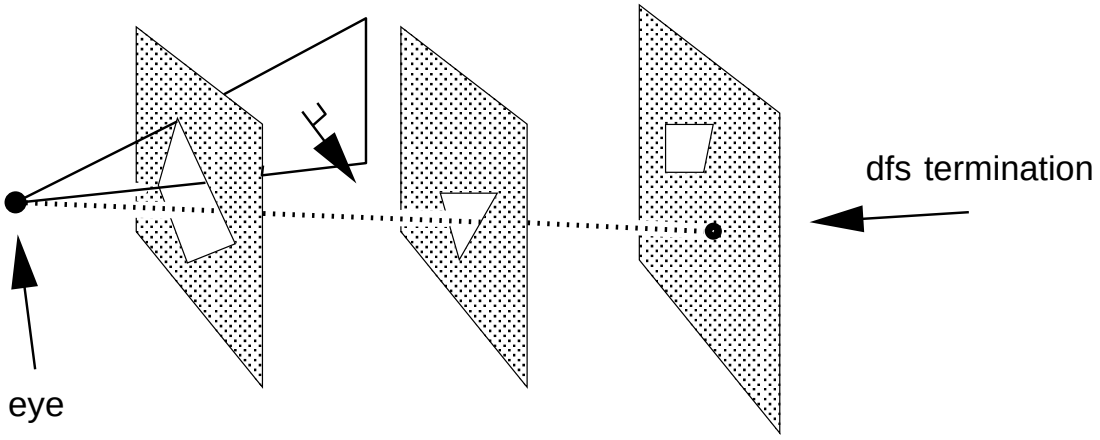

Figure 6: Encountering a portal in the eye-to-cell DFS.

of the plane normals. We examine this collection of three-coefficient linear constraints for a feasible solution in linear time using a linear programming algorithm (Seidel 1991). If the linear program fails to find a stabbing line through the eye, the most recent portal is impassable, the reached cell is not visible through the current portal sequence, and the active branch of the DFS terminates.

Otherwise, the cell is at least partially visible; its contents (the incident detail objects) are then examined for sightlines. Recall that only the axial bounding boxes of objects are retained. Convex objects always have convex silhouettes; in the case of axial boxes, the silhouette edges are easily obtained via a table-lookup of the eye position. Generically, a cube has a hexagonal silhouette; again, each silhouette edge spans a plane with the eye, oriented so as to contain the object centroid in its positive halfspace. The augmented set of linear constraints is examined for a sightline using linear programming, as before.

Geometric Witnesses for Linear Programming

We wish to define geometric witnesses for both the success and failure cases in the DFS; i.e., geometric "proof" data for the instances in which eye-based sightlines do and do not exist. The witness for the success case is straightforward: the linear program computes and returns a vector whose inner product with all of the active constraints is positive (this can be straightforwardly checked by examining this product for each active constraint). The witness in this case has an obvious visual representation: a line segment, originating at the eye, and ending at the plane of the newly encountered portal (Figure 10).

The witness for the failure case is slightly more complex. We must consider the linear programming algorithm (the discussion here follows that of Seidel 1991). The algorithm is given a set h_k of halfspaces, and a linear objective function to minimize. Recursively, the algorithm removes a halfspace H from the set, and computes the optimum with respect to the remaining halfspaces. The removed halfspace is then replaced, and the computed optimum is examined with respect to this halfspace. If the optimum is in the halfspace, we are done. If it is not, then, if there is any feasible solution, there must be a solution on h, the bounding hyperplane of H (this is true by convexity of the feasible region). Thus, the algorithm projects the active constraints onto h and solve the $d-1$-dimensional linear program there. Infeasibility is established when the algorithm projects to a one-dimensional problem instance and the active constraints are found to be infeasible. For our problem, this means that three constraints are necessary: one that is the current subspace, and two that together produce an infeasible region in this space.

Visual Representations and Visual Correctness

Figure 7 depicts the dynamic eye-based culling operation as seen from "outside" of the model,

i.e., above the room containing the observer (backfacing polygons have been removed so that we can see through the ordinarily opaque walls and ceiling). Note the visual representations of the observer, a one-eyed stick-figure, and the instantaneous view frustum, emanating from the center of the observer's head. Portals are depicted as $\times$ shapes. The window portal is successfully stabbed by a sightline through the eye and inside the view frustum. The object bounding boxes shown are those surviving the eye-based visibility cull (the objects' many constituent polygons are not displayed as they are irrelevant to the culling operation).

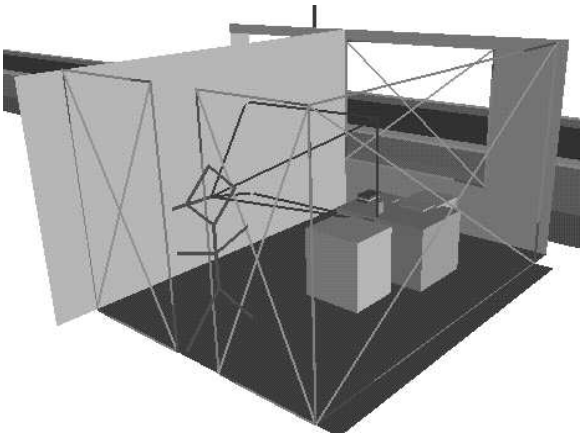

Figure 7: An outside view of the point observer.

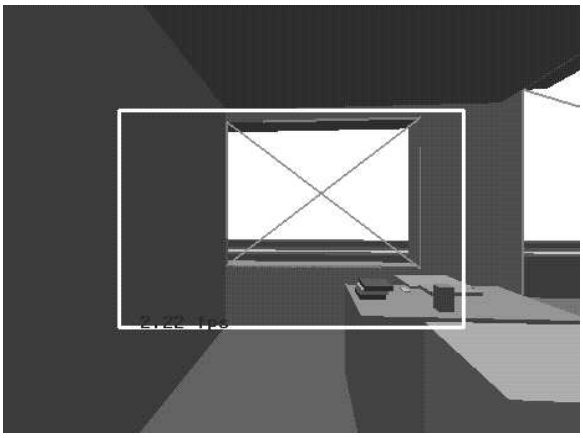

Figure 8: The point observer's view. Note contracted view frustum.

Figure 8 shows the same situation, from the point of view of the observer. The window portal, and the object bounding boxes shown, are clearly incident on the view frustum. However, the view frustum is displayed so that it does *not* fill the display window. This is so that, from the inside view, we can ascertain visually that A) portals and objects outside the view frustum are discarded, as desired, and B) no portals or objects outside the view frustum are traversed. This is difficult to do from the outside view, and would be impossible from the observer's view if the frustum were to fill the display window (as it does in the intended application).

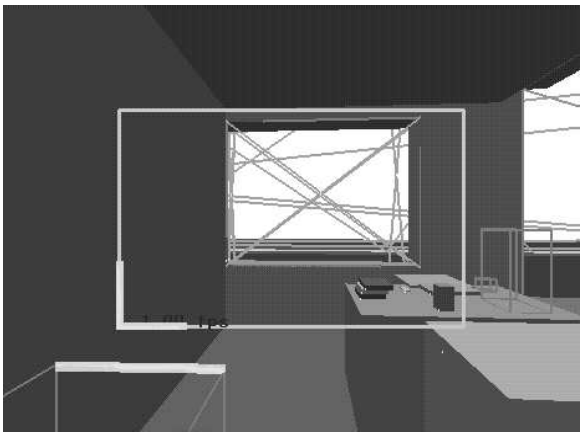

Figure 9: Objects failing the cull are drawn in wireframe.

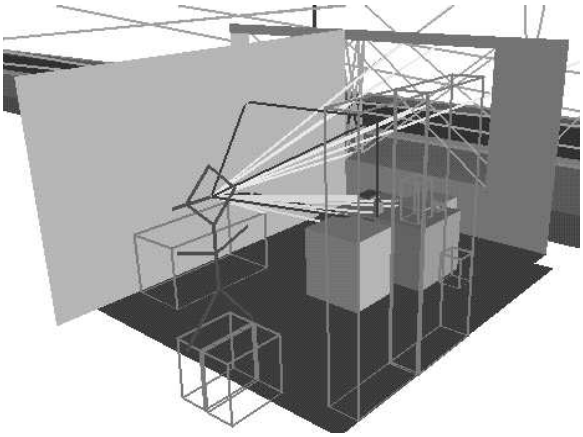

Figure 10: The outside view, with witness object sightlines.

Figure 9 illustrates the utility of the contracted view frustum. Objects surviving the visibility cull are drawn as solid parallelepipeds, whereas objects failing the visibility cull are drawn in wireframe. The witness for the failure case is three edges, where one edge defines an extremal edge of the visible region, and two further edges show that there are no feasible points on that edge. One such witness is shown for the wireframe object at the lower left; the leftmost edge of the view frustum is constraint on which the recursive linear programming algorithm "bottomed out," and the oppositely directed (horizontal) edges clearly exclude any feasible points on this edge. The three witness edges are rendered as thick white line segments.

The correctness of the visibility cull is evident; all wireframe objects are outside of the view frustum. Finally, Figure 10 shows the outside view of the same situation. The wireframe objects are again seen (the witness from the previous figure is visible as a white line segment on the top of the wireframe object at lower left). For each object surviving the cull, a witness *sightline* is drawn from the observer to some point on the object (in Figure 9, each sightline appears as a single pixel, since it necessarily contains the eye).

We distinguish the technique of outside/inside views presented above from traditional "multi-view" systems. In traditional systems, some dataset is traversed with different display routines, and the resulting differing visual representations are simultaneously displayed. The technique above is subtly different, in that the traversal (i.e., culling and rendering) routine that generates the pictures is *exactly identical* in both cases; only the viewing transformations differ. This means that the resulting visual representations *must* agree. This ensures consistency of representations in a somewhat stronger sense than does a traditional system, which must arrange that differing traversal codes produce isomorphic representations of the traversed data.

Temporal Coherence

The above figures represent a single observer position out of the many thousands generated by a typical interaction session with the walkthrough system. When many successive positions on the observer's path are considered, a new and useful phenomenon becomes evident: *temporal coherence*. The human visual system is extremely sensitive to sudden changes in the visual field. In ordinary human experience, objects tend to change position and appearance slowly and smoothly. Therefore, we are well-equipped to detect transient or rarely-occurring errors in geometric algorithms, simply by changing the input smoothly, and watching the output for non-smooth behavior. We found several important errors in exactly this manner, when a user of the system noticed objects within the view frustum "flashing" on and off in the visual field. Moreover, scripting of the user's path allowed the error to be reliably reproduced.

During the constrained depth-first search, each portal edge encountered gives rise to a halfspace spanned by that edge and the eye (cf. Figure 6). These halfspaces can be drawn explicitly, for example as shaded triangles. However, there is a more effective way to visualize their aggregate effect. When the DFS successfully arrives at a cell, a single portal sequence (and thus set of halfspaces) is active. These halfspaces are appended to the list of halfspaces bounding the reached cell, and their common intersection is computed (using a standard three-dimensional convex hull algorithm). The result is a necessarily convex volume that bounds the region of the reached cell potentially visible to the observer through the active portal sequence (Figure 11). Computing and drawing these convex regions in real time gives a powerful visual confirmation that the DFS is operating correctly, and that the correct potentially visible regions are being enumerated by the dynamic culling algorithm.

Robustness

As a piece of evidence that the techniques proposed here actually help to make code more robust, we relate the following experience. In our system, there are several versions of libraries to compute the convex hull of a set of points in three dimensions. Each of the libraries functions differently, but is encapsulated by a code "wrapper" that presents a uniform interface to the calling code. This wrapper code also performs consistency checks on the hull data structure, checking for example that it satisfies Euler's relation ($V - E + F = 2$), that every edge is shared by two faces, that every vertex is used (referenced) by at least three faces, etc. If the computed data structure does not pass all of these tests, an error bit is set in the hull data structure. When the hull is later drawn, its color is overridden and set to bright red to indicate the existence of the error. This visual cue alarms the practitioner, and the problem can be isolated.

After (an estimated) several hundred thousand interactive invocations of the convex hull code, a list processing bug became apparent, in code that had been used by "fifty people around the world" and was said to be "bug-free," i.e., to handle all possible inputs correctly (O'Rourke

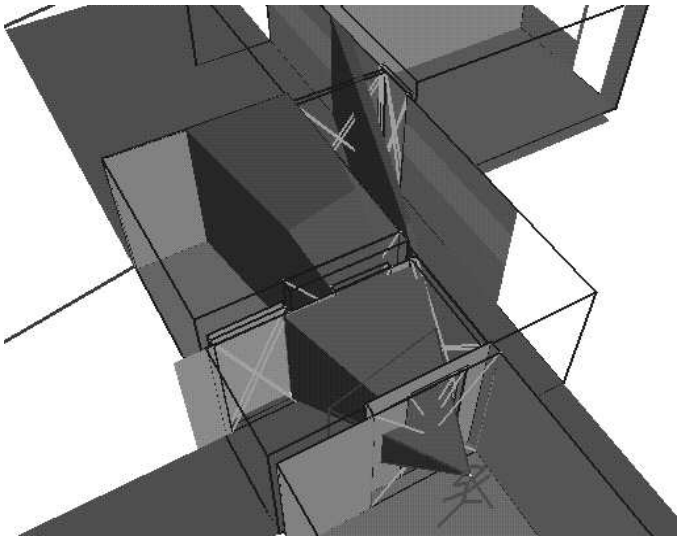

Figure 11: The polyhedral regions potentially visible to the observer.

1992a); this was a reasonable statement to make, because the algorithm operates in integer coordinates. The author of the code explained:

> Found the bug and fixed it. It was.... a basic list-processing bug, which I wonder if now is not elsewhere in the code. The vertex list at a certain point consists of (0,6,5,4,3,2,1), and 0 and 6 are marked for deletion. The loop deletes 0, but since 0 is the head, the head gets changed to 6, and the loop [terminates].... so 6 never gets deleted.

This bug turned out to be very subtle, and had gone undetected by scores of other users of the code. We claim that it was found largely because of the particular interactive fashion in which the code was exercised.

Interaction Techniques

Finally, it is worthwhile to examine the method by which the algorithm developer generates observer positions from the "outside" view. Figures 12 through 14 show three successive observer positions, fields of view, and dynamic cull results. The observer is depicted as a stick-figure and frustum as before. The observer's position in the architectural model is derived from the mouse position by intersecting a line from the eye to the mouse (in model coordinates) with a horizontal plane containing the last-known observer position (the graphics library makes the aggregate viewing transformation constantly available, so the conversion of the line to model coordinates requires a 4×4 matrix inversion). The observer's field of view is derived by computing a weighted average of the mouse-motion vector with the current view-direction, using a small adjustable coefficient (usually about 0.05) for the mouse term. Thus the observer's view direction smoothly and exponentially relaxes to the predominant direction of mouse motion, and the observer can be directed anywhere in the model by the developer. Finally, the interface is slightly modal in that, when an interesting user position is found, it can be "frozen" with a key-press; further mouse motion does not change the observer position, and can be used, for example, to effect view transformations. This freeze technique was used to generate Figures 7 through 10; the inside/outside views were toggled via a key-press, or they can be simultaneously displayed.

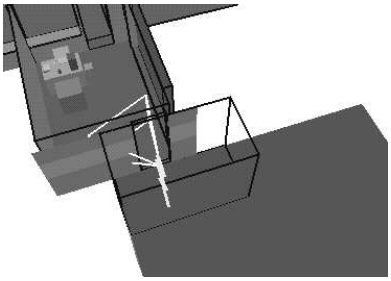

Figure 12: Frame A.

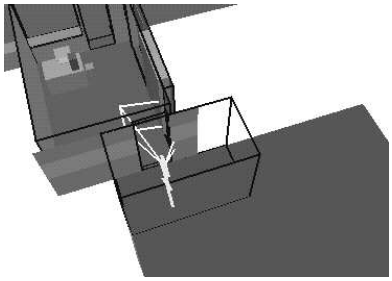

Figure 13: Frame B.

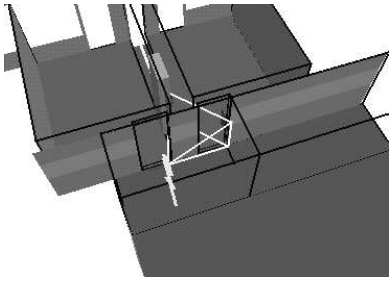

Figure 14: Frame C.

CONCLUSION

Practitioners can develop robust geometric algorithms more effectively by transferring interactive functionality typically found only in applications to graphical environments used for algorithm development and inspection. We discussed some simple but important elements of a successful algorithm visualization framework, and presented some examples of the use of each element. This system was used to develop a robust architectural simulation application that depends on the correct operation of many non-trivial complex geometric algorithms. We maintain that these algorithms were much easier to develop and verify using the proposed interaction techniques than they would have been otherwise, and that the interaction techniques were employed for little incremental effort over that needed to develop the geometric algorithms themselves.

ACKNOWLEDGMENTS

The author is grateful to Carlo Séquin and to Ari Rappoport for their encouragement and helpful comments.

The general-dimensional linear programming code was supplied by Michael Hohmeyer of U.C. Berkeley from a description in (Seidel 1991). The fast three-dimensional convex hull code was adapted from integer code supplied by Joe O'Rourke of Smith College (O'Rourke 1992).

BIBLIOGRAPHY

Bier, E (1990) Snap-dragging in three dimensions. In: *ACM Symposium on Interactive 3D Graphics*, pp 193–204.

Funkhouser, T, Séquin, CH, Teller, SJ (1992) Management of large amounts of data in interactive building walkthroughs. In: *Proc. 1992 Workshop on Interactive 3D Graphics*, pp 11–20.

Khorramabadi, D (1991) A walk through the planned CS building. Technical Report UCB/CSD 91/652, Computer Science Department, U.C. Berkeley.

O'Rourke, J (1992a) Personal Communication, November 1992.

O'Rourke, J (1992b) Computational geometry in C: Chapters 3 & 4 convex hulls. Technical Report TR # 017, Department of Computer Science, Smith College.

Seidel, R (1991) Small-dimensional linear programming and convex hulls made easy. In: *Discrete and Computational Geometry*, pp 423–434.

Teller, SJ (1992a) Computing the antipenumbra cast by an area light source. In: *Computer Graphics (Proc. SIGGRAPH '92)*, 26(2):139–148.

Teller, SJ (1992b) *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, Computer Sciences Department, U.C. Berkeley.

Teller, SJ and Séquin, CH (1991) Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proc. SIGGRAPH '91)*, 25(4):61–69.

Seth J. Teller is currently a postdoctoral researcher in the Institute of Computer Science at the Hebrew University of Jerusalem. His research interests include computer graphics and computational geometry. Teller received his BA in Physics from Wesleyan University in 1985, and his MSc and PhD in Computer Science from the University of California at Berkeley in 1990 and 1992, respectively. He has also been a part-time member of the Research and Development group at Silicon Graphics since 1988.

Address: Institute of Computer Science, Hebrew University of Jerusalem, Givat Ram, Jerusalem, 91904, Israel. Phone: +972-585867; Fax: +972-585439.