GASP – An Animation System for Computational Geometry

David Dobkin and Ayellet Tal
Department of Computer Science
Princeton University
Princeton, NJ 08544
dpd@cs.princeton.edu ayt@cs.princeton.edu

Abstract

This paper describes a system, GASP, that enables the creation and viewing of animations of geometric algorithms. GASP facilitates the task of both the programmer and the viewer. Animations can be based upon either an implementation of an algorithm or upon ASCII data. System defaults make it possible for the creator of the animation to be isolated from decisions of a graphical nature. The viewer is provided with a comfortable user interface. That interface enhances the exploration of an algorithm's functionality and assists in debugging geometric code. We provide several examples through which we demonstrate the value of GASP to the computational geometry community.

This paper does not fit the usual "algorithm-theorem-proof" format typical of a computational geometry paper. Instead, it describes a system intended to help researchers implement and experiment with geometric algorithms. Systems such as the Computational Geometry Workbench, XYZ GeoBench, and LEDA are immensely valuable tools providing the programmer with library routines and user interfaces. Our work follows in their footsteps. However, it adds an orthogonal dimension to these efforts: GASP's primary focus is the animation and visualization of algorithms. Thus, it need not make any assumption about how the geometric code is produced, whether it be stand-alone or library-linked. While providing interactive visualization support, GASP can also be used as a visual debugging facility. The system has already been used by several researchers to produce animations (and videos) of highly complex algorithms (e.g., optimal segment intersection, polyhedral intersections, max-discrepancy, polyhedral separators). We are hopeful that this tool will prove as useful to the computational geometry community —albeit in a different manner— as Workbench, GeoBench, or LEDA.

1 Introduction

The past decade has seen a remarkable growth in the availability, speed, and ease of use of computers. The effects on the computational geometry community have been significant. Faster machines and better software have made LaTeX [12], and Postscript [1] the *lingua franca* of communication and publication. Higher speed networks coupled with wider connectivity have made electronic mail the medium of choice for research communication. However, despite the visual nature of computational geometry, we have been less able to take advantage of similar speed gains in graphics hardware and software. The community has not embraced the available tools and until recently it was rare to have visualization as part of a computational geometry paper. In large part, this was due to the difficulty in using graphics systems combined with the difficulty of implementing geometric algorithms. This combination made it a major effort to animate even the simplest geometric algorithm. However, things are beginning to change. The past two computational geometry conferences have had video proceedings and last year's conference had a paper [20] that was accompanied by an animation [19].

In this paper we describe a visualization system that we have built to help this change. Our system is designed to isolate the user from any concern about how graphics is done. Just as LaTeX isolates a user from having to create their own macros in TeX , our system allows a user to create interesting graphics on an Iris workstation without any knowledge of GL, Inventor [18] or other underlying graphics libraries or tools. Our system is an interface to the graphics library and through style files the user controls the graphical and artistic aspects of the animation. The user needs only to write a short C program to link her program or data to our system to create an animation. The animation can be based upon an implementation of the algorithm or merely upon an ASCII data file. The latter case allows the user to produce an instructive animation without implementing the general form of the algorithm. This process is no more difficult than what would be required to create figures for a paper.

Even simple animations are a powerful adjunct to technical papers. Further improvements can enhance this power. Allowing the viewer to control the speed of an animation improves the understanding. In the ideal animation, the user can also choose various inputs and interact with the animation. Our system allows this. While using our system, the "student" can specify the input and set the speed for the individual steps of the algorithm animation. In addition, the student can rewind and view each step multiple times. The additional effort to make an animation interactive is trivial.

The ability to interact with the animation is useful not only for students who can get insight into geometry and understand the intuition behind the algorithm. It is also very valuable for debugging when trying to ensure the correctness of the program. There is an inherent difficulty in

checking a geometric object (e.g. listing vertices, edges and faces of a polyhedron) in a debugger, a difficulty that can be eliminated once it becomes possible to view the object. Also, degenerate cases can be discovered more easily once seen on the screen. In addition, the ability to "rewind" the animation gives the programmer a way to find the stage where the program started behaving not as anticipated without rerunning the program.

Historically, algorithm animation has been a lively topic of study since graphics devices were able to provide usable displays of running algorithms. They allow students to explore the dynamic behavior of a program and improve the understanding of an algorithm. There has been a lot of success in this area. The best-known systems are Balsa [5], Balsa-II [2], [3], Tango [21], [22] and Zeus [4]. They are designed for fundamental algorithms taught in a university-level data structures or algorithms course. There have also been special purpose animation systems developed for subareas of algorithm analysis. Of particular interest to us are the systems that have been developed for geometric algorithms and related data structures. Among these systems, LEDA [13], the Computational Geometry Workbench [11], and the XYZ GeoBench [16] systems have become the most successful. Their chief goal is to facilitate the implementation of geometric algorithms by providing a library of routines along with a user interface. Our system, *GASP (Geometric Animation System, Princeton)*, supplements such systems by providing a rich set of visualization and animation tools, which can also be used for visual debugging. The main contribution of our work ultimately is an executable that can be retrieved by *ftp* enabling others to produce videos similar to those that we have produced in the past and the collection accompanying this paper. We describe the results of decisions we made to simplify the animator's task. Our "theorems" are given as examples of short programs that have done all the work of making the animations. Our "proofs" consist of the animations on the video tape. We believe that this paper is a complete piece of work describing a tool of use to the community and hope that you will evaluate it as such.

The remainder of the paper consists of examples of how our system can be used. In the next section we work through a case study telling how we created the video that was part of last year's video show [10]. The third section describes various applications of the system to ease the making of videos such as those that appeared in previous year's video reviews. We give short snippets of code and only brief parts of the video to show how things were done and can be done in a more general form. In Section 4 we discuss the use of GASP as a central part of teaching geometry courses. We conclude in Section 5.

2 Case Study

As we were developing GASP, we used the polyhedral hierarchy construction [7, 8] as an example to guide some of our thinking. We had previously implemented the computation of hierarchies as part of a prior system [9]. The goal, when developing GASP, was to create an animation that highlighted this process. As we made the animation, we also wanted to include an example of the hierarchy in action and so we implemented the algorithm of [8] for detecting plane-polyhedral intersection. As this implementation progressed, we started on the animation. As a result, the animation was an aid in debugging the implementation.

The goal of the animation is to explain how the hierarchy is constructed and then how it is used. For the first of these we want to be able to explain a single pluck and then show how the hierarchy progress from level to level. We refer the reader to the accompanying video.

First, we want to show a single pluck. The animation begins by rotating the polyhedron to identify it to the user. Next we highlight a vertex and lift its cone of faces by moving them away from the polyhedron. Then, we add the new triangulation to the hole created. Finally, we remove the triangulation and reattach the cone. This is done in GASP by the following piece of C code which is up to the creator of the animation to write.

```
explain_pluck(poly_vert_no, poly_vertices,
        poly_face_no, poly_faces, poly_names,
        vert_no, vertices, face_no, faces)
{
  /* create the polyhedron */
  Begin_atomic("poly");
  Create_polyhedron("P0", poly_vert_no,
       poly_face_no, poly_vertices, poly_faces);
  Rotate_world();
  End_atomic();

  /* remove vertices and cones */
  Begin_atomic("pluck");
  Split_polyhedron(poly_names, ''P0'',
       vert_no, vertices);
  End_atomic();

  /* add new faces */
  Begin_atomic("add_faces");
  Add_faces(poly_names[0], face_no, faces);
  End_atomic();

  Undo(2); /* undo plucking */
}
```

Each of the operations described above is a single GASP primitive. `Create_polyhedron` fades in the given polyhedron. `Rotate_world` makes the scene spin. `Split_polyhedron` highlights the vertex and splits the polyhedron as described above. `Add_faces` fades in the new faces. Finally, the `Undo` operation removes the triangulation and brings the cone back to the polyhedron. (Notice that the operations are grouped into logical phases, called *atomic units*. Atomic units allow the programmer to isolate phases of the algorithm to be animated concurrently by enclosing them within a `Begin_atomic` and `End_atomic` phrase. In our case, the polyhedron is being created and the scene is spinning as one unit.)

Notice that the code does not include the graphics: Coloring, fading, traveling, speed etc. are not mentioned in the code. In the related *style file*, these operations are controlled. The defaults are set for the speed of rotation, the speed of translate, the duration of time each operation takes etc.

After explaining a single pluck, the next step is to show the pluck of an independent set of vertices. This is no more difficult than a single pluck and is achieved by the following code.

```
animate_one_level_hierarchy(atomic1_name,
     atomic2_name, atomic3_name, poly_name,
     vert_no, vertices, face_no, faces,
     new_polys_names)
{
  Begin_atomic(atomic1_name);
  Split_polyhedron(new_polys_names, poly_name,
                   vert_no, vertices);
  End_atomic();

  Begin_atomic(atomic2_name);
  Add_faces( new_polys_names[0], face_no, faces);
  Finish_early(0.5);
  for (i = 1; i <= vert_no; i++){
    Remove_object(new_polys_names[i]);
  }
  End_atomic();

  Begin_atomic(atomic3_name);
  Rotate_world();
  End_atomic();
}
```

Here again we use the style file to choose speeds at which cones move out, faces fade in, the scene spins etc. We also use the style file to choose a next color that contrasts the new faces with those that are preserved. This allow the user to experiment with the animation without modifying and recompiling the code.

It is a matter of creating a `for loop` to be able to repeat the hierarchy construction down as many levels as desired.

Now, having created the first two segments of the animation, we turn to the task of walking through the hierarchy, showing the intersection detection process. We begin by highlighting a closest vertex and then passing a plane of support through this vertex. Both of these are primitive operations supported by GASP. For the hierarchy to grow between levels, it is helpful to have the rotation stop at an appropriate place to allow consideration of the growth. GASP allows the user to do the rotation by hand and have it recorded so that the animation can then play it back. This replaces the tedious task of iteratively specifying rotations by hand within a C program until the right rate is found.

Next, there is the issue of gluing together the parts of the animation. Initially it was necessary to watch the entire animation when debugging the final section. This was required since earlier stages of the animation set state variables that are needed by later stages. We have changed this in GASP, building an interface which is modeled after VCR controls. Using these controls, the programmer can fast forward over initial fragments to get to the section of interest. It is further possible to single step through the section under consideration, an incredibly valuable tool when debugging.

The last step is to move the animation from the screen to videotape. A problem here is that the colors that are satisfying on the screen are not acceptable on tape (it's a difference between RGB colors and NTSC colors). Anybody who has tried to find the right colors knows how tedious this job can be. Again, GASP addresses this issue by allowing the user to adjust one line in the style file to note that the video (rather than the screen) color palette should be used.

3 Animating Algorithms with GASP

With GASP, the complexity of designing and implementing the animation is minimal. The interface we provide allows the programmer to write brief snippets of C code to define the structure of an animation. The system can be used in one of two ways to make accompanying videos for talks and classes. First, users can implement their ideas and use GASP to create a video of their animation. Second, they can create (in any way they wish) ASCII data of sample runs of their algorithm which can be captured either as a video or a collection of individual images.

Sometimes the user wishes to influence the look of the animation and not only its structure. The user can accommodate his personal taste by editing an ASCII *style file* which controls viewing aspects of the animation. Each operation can generate several possible visualizations, called *styles*; one of which is the default. For example, the creation of a polyhedron can be visualized by fading into the scene, by growing from a point or by moving into the scene. The programmer can choose the specific style by editing the style file. Parameters that can be determined by modifying the style file include the speed of the animation, the size of the vertices, the line width, the transparency value of the objects, the fonts and sizes of titles and text, the style of the operations, the color for the objects and many others. Even when the user changes the style file, the user needs no specific knowledge of computer graphics. The animation is generated automatically by the system. But a different animation will be generated if the style file is modified. We will not discuss style files any further in this paper. We refer the reader to Appendix B for an example of the style file which we used for the example given in the previous section.

In this section we show the flexibility of GASP by describing animations created for various algorithms. We demonstrate how easy it will be for authors to take their data, plug it into GASP and get a video which can go along with any geometry conference paper. Excerpts from the animations appear in the accompanying video.

Displaying ASCII Data: In the previous section we described an animation for a three dimensional algorithm that was first implemented and then animated. Sometimes, the user has already produced the data by other means (e.g. Mathematica). Usually, Xfig or similar tools are used to statically display the data. GASP offers an alternative to this. The data can be displayed dynamically by writing very short but powerful code. As an example, we chose the paper "Objects that cannot be Taken Apart with Two Hands" [20] whose authors generated the data using Mathematica. Although the paper was beautifully visualized in [19], we wanted to check how easy it would be to duplicate the animation using GASP. We asked the authors for the ASCII data and re-animated the first and the last parts of

the video which appear in the accompanying video. It took us far less than a day to do this.

The first part of the animation shows a configuration of six tetrahedra that cannot be taken apart by translation with two hands. The last part of the animation displays a configuration of thirty objects that cannot be taken apart by applying an isometry to any proper subset. Each part of the animation begins by fading each object, in turn, into the scene. The colors of the sticks vary. After all the sticks appear in the scene, the scene rotates so that the configuration of the sticks can be examined.

The animation is produced by the following brief C function. No graphical aspects appear in the code. GASP handles this by making heuristic guesses for the way the animation appears. GASP decides that fading is the appropriate way to make the sticks appear. GASP determines the color of each object. GASP decides the speed of each operation and how many frames it should take. Any decision made by GASP can be changed by the user in the style file for the animation.

In the code below, except for **get_polyhedron**, the other functions belong to GASP. The function **get_polyhedron** reads the ASCII data for each object from a file. **Create_polyhedron** is responsible for fading in a single stick. **Rotate_world** causes the scene to spin.

```
hands(int stick_no)
{
  float (*points)[3];
  long *indices;
  int nmax, fmax, i;
  char *atomic_name, *stick_name;

  for (i = 0; i < stick_no; i++){
    /* stick i */
    get_polyhedron(&points, &indices, &nmax,
        &fmax, &atomic_name, &stick_name);

    Begin_atomic(atomic_name);
    Create_polyhedron(stick_name, nmax,
        fmax, points, indices);
    End_atomic();
  }

  Begin_atomic("Rotate");
  Rotate_world();
  End_atomic();
}
```

Using 3D Displays to Improve on the Visualization of 2D Objects: Three-dimensional animations can exploit many types of continuous change. For example, spinning can help in viewing the scene from various directions. Transparency can be used to show what exists behind the viewing plane. Creating pleasing animations in 2D is more of a challenge. Yet, many algorithms are two-dimensional. To produce prettier animations in two dimensions, GASP visualizes two-dimensional geometry in three dimensions. This allows smooth motion and gives two-dimensional objects "depth". (e.g. a line is presented as a cylinder.)

We give here two examples of two dimensional animations. The first illustrates a motion planning algorithm by [14] and was previously animated in [15]. The second, which is based on [6], finds line segment intersections and was previously visualized in [23]. Both can be viewed in the accompanying video.

Motion Planning: In this algorithm, the object to be moved is a disc and the obstacles are simple disjoint polygons. We made up similar data and duplicated the first part of the original video. The user needs only define the structure of the animation. That is, first show the obstacles, then show the initial and final positions of the disc, display the disc and finally move the disc along the path from the initial position to the target. To do it, the programmer must write the short C function which appears in Appendix A. GASP determines how the animation actually appears. In this case, the animation begins with red obstacles fading together into the scene. Then, two green spheres fade in, representing the initial position and the target. At that point a blue sphere (the disc) fades into the initial position and moves on a given path until it gets to its final position. This animation took only a few hours to create. A simple task!

Line Segment Intersections: This example illustrates a long multi-phase 2D animation. It follows the structure of the animation that appeared in [23], but the presentation is completely different. The animation shows a line segment intersection algorithm in action and illustrates its most important features. The first phase of the animation presents the initial line segments and the visibility map that needs to be built. The second phase demonstrates that the visibility map is being constructed by operating in a sweepline fashion, scanning the segments from left to right, and maintaining the vertical visibility map of the region swept along the way. Finally, a third pass through the algorithm is made, demonstrating that the cross section along the sweepline is maintained in a lazy fashion, meaning that the nodes of the tree representing the cross section might correspond to segments stranded past the sweepline. The animation shows how the tree and the corresponding cross sections on the visibility map change.

Since the structure of the animation has not changed, we used the functions written for [23]. We changed the graphics in these functions to include calls to GASP, and as a result the code became considerably shorter. The animation (from which we show only short excerpts), however, has become much more appealing. Objects, previously presented as 2D lines, points and trees, now appear as cylinders, spheres and trees in 3D respectively. In the first pass, red line segments fade into the scene. While they fade out, a green visibility map fades in on top of the initial segments. This gives the user a chance to watch the correlation between the segments and the map. Yellow points, representing the "interesting" events of the algorithm, then blink. At that point, the scene is cleared (except for the initial segments) and the second pass through the algorithm begins. The viewer can watch as the sweep-line advances by rolling to its new position. The animation also demonstrates how the map is built - new segments to be added to the map fade in in blue, and then change their color to green to become a part of the already-built visibility map. The third pass through the algorithm adds more information about the process of constructing the map by animating the change in the red-black trees which are maintained by the algorithm. In addition, the animation presents the "walks"

on the map.

There are only eleven GASP's calls necessary for the creation of this animation and they are: `Begin_atomic`, `End_atomic`, `Create_line`, `Create_point`, `Remove_object`, `Scale_world`, `Rotate_world`, `Create_Sweepline`, `Modify_Sweepline`, `Create_tree`, `Add_node_to_tree`.
Note that the last four calls represent the support GASP gives for the creation and manipulation of special objects such as sweep lines and trees.

A final point to be made is that since the algorithm is actually running when the animation is executed, the student can not only control the execution of the animation but can also choose the input and view how the animation is changing.

Animating Non-Geometric Algorithms: Though GASP was originally meant to facilitate animations that involve three dimensional geometric computation, we found that the interface we provide actually facilitates the animation of any algorithm that involves the display of three dimensional geometries, among them many of the algorithms in [17]. To show the added power of the system, we chose to animate heapsort.

In the movie, which is displayed in the accompanying video, each element is represented as a cylinder whose height is proportional to its key value. The elements first appear in an array and then it is demonstrated how the array can be looked upon as a tree. From this point, the animation shows two views of the heap - one as an array and the other as a tree displayed in three dimensions. The next step of the animation is to build a heap out of the tree in a bottom up fashion. Whenever two elements switch positions, they switch in both views. After the heap is built, the first and the last element switch and the heap is rearranged. At the end, when the array is sorted, the colors of the elements are "sorted" as well.

4 GASP in the Classroom

The previous section was oriented towards using GASP as a research tool. In this section we describe the use of GASP as an educational tool.

A research videotape can be shown in a computational geometry course as a means of introducing topics and explaining algorithms. This gives students overviews of nontrivial algorithms. A drawback of videotapes is that the animator chooses both the input to be used and the speed at which the animation should run. This is unfortunate since ideally the viewer would like to be able to choose varying inputs and interact with the animation in a way that fits the individual's level of understanding. GASP's interactive environment, illustrated in Figure 1, is designed to be simple and effective and meet the student's needs.

Students, viewing an animation, want to explore the algorithm at their own pace. A student might want to stop the animation at various points of its execution. Sometimes it is desirable to fast-forward through the easy parts and single-step through the hard ones to facilitate understanding. The student may want to "rewind" the algorithm in order to observe the confusing parts of the algorithm multiple times. GASP's environment allows this. The *Control Panel* (Figure 1), which appears upon entering GASP, lets the user direct the execution of the algorithm. To make it as intuitive as possible, the panel uses the familiar VCR metaphor. The panel allows running the algorithm at varying speeds: fast(>>), slow(>) or unit by unit (> |). The analogous <, << and | < push buttons run the algorithm "backwards". The viewer can *PAUSE* at any time to suspend the execution of the algorithm or can *EJECT* the movie.

It is often important for the student to interact with the scene itself in order to observe the objects "behind", to view the object of interest from a different angle or to check the relations of objects. With GASP, the student can rotate, translate or scale the scene to achieve this. The camera can also be reset to a "home" position, or be repositioned. These are done with thumbwheels and push buttons which decorate each of the windows in which the animation runs, called the *Algorithm Windows* (Figure 1).

An important capability in the process of experimenting with an algorithm is the ability to run the algorithm on an input of your choice. When the algorithm is implemented, this becomes an easy task, as was the case for the *Line segment intersection* algorithm described in the previous section. In this example, the user need only define the input in an ASCII file (or ask for random input to be generated) and the required animation runs. We have supporting programs that generate the data in GASP's format, given several standard formats.

In addition, the user can get *information* by pressing the push buttons in the Algorithm Window. It is possible to list the objects currently appearing in the scene, print description of a chosen object (e.g. list of vertices and faces of a polyhedron), list the current transformation of the selected object or the global transformation and create a Postscript file of the screen.

A *Text Window* (Figure 1), supported by GASP, adds the ability to accompany the animation running on the screen with verbal explanations. Text can elucidate the events and direct the student's attention to specific details. Every atomic unit is associated with a piece of text which explains the events occurring during this unit. When the current atomic unit changes, the text in the window changes accordingly. The next version will support voice as well.

Finally, many times students implement algorithms as part of their term projects. Not only does GASP enable them to produce animations of their implementations, but also they can debug their programs more easily. The visual nature of GASP is a first step in easing the task of debugging geometry. GASP does more than this. Single stepping, fast forwarding and especially rewinding are extremely useful in the debugging process. Rewinding is important since typically a bug is discovered only after it has occurred.

5 Conclusion

This paper has described a system for automatically generating animations in three dimensions. The main benefit of the system is that no knowledge of graphics is required. To do this, we distinguish between *what* is being animated and

how it is animated. The code includes only manipulations of objects and modifications of data structures. The programmer need not specify *how* each operation is visualized. GASP makes heuristic guesses for the way the animation appears in order to create a visually pleasing one. It is possible to change the look of the animation by editing a style file. We have shown several animations of geometric algorithms. None of the animations took long to produce. We also discussed GASP's environment which allows the user to control the execution of the algorithm in an easy and enjoyable way. We believe that enough animations can be created easily to improve the way geometric talks and courses are given.

We envision a situation where a variant of a Postscript file can include enclosures that have animations in the same fashion that current Postscript files seamlessly merge text and figures. As an example of our vision, we consider two models. The first is the existing environment on the Macintosh computer. It is possible for a user of Word to include QuickTime animations in a Word document. The reader of the document is presented with an icon in the document. Clicking on the icon causes the animation to play. Most mailers currently being released on UNIX systems give the user the ability to attach enclosures. The widespread availability of X (on all boxes ranging from thousand dollar terminals to special purpose workstations) and its support for animation suggests that a similar situation will exist in the UNIX world very soon. It is not taxing for a workstation having 32 Megabytes of RAM to play 20-30 second clips of low resolution (e.g. 320x240) slow speed (e.g. 15fps) animation in real time. This makes a powerful adjunct to a technical paper. There still remains the problem of enabling users to create the animations to attach to their documents. GASP, or a similar tool, can aid here.

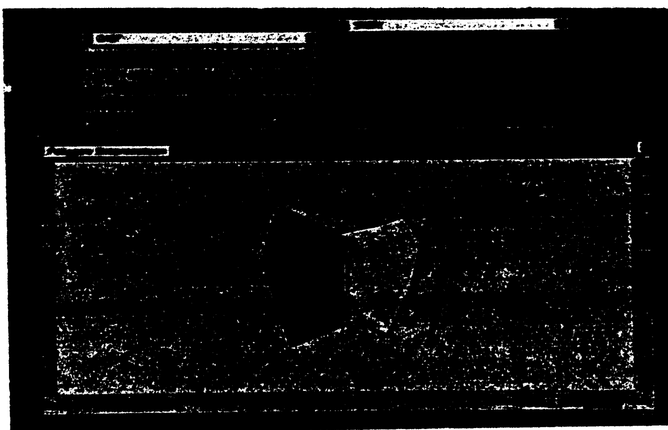

Figure 1 - GASP's environment

REFERENCES

[1] Adobe-Systems-Incorporated. *PostScript language - Reference Manual.* Addison Wesley Publishing Company, Inc.

[2] M.H. Brown. *Algorithm Animation.* MIT Press, 1988.

[3] M.H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, May 1988.

[4] M.H. Brown. Zeus: A system for algorithm animation and multi-view editing. *Computer Graphics*, 18(3):177–186, May 1992.

[5] M.H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(1):28–39, Jan 1985.

[6] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, 1992.

[7] D. Dobkin and D. Kirkpatrick. Fast detection of polyhedral intersections. *Journal of Algorithms*, 6:381–392, 1985.

[8] D. Dobkin and D. Kirkpatrick. Determining the separation of preprocessed polyhedra – a unified approach. *ICALP*, pages 400–413, 1990.

[9] D. Dobkin, S. North, and N. Thurston. A viewer for mathematical structures and surfaces in 3D. In *1990 Symposium on Interactive 3D Graphics*, pages 141–142, March 1990.

[10] D. Dobkin and A. Tal. Building and using polyhedral hierarchies (video). In *The Ninth Annual ACM Symposium on Computational Geometry*, page 394, May 1993.

[11] P. Epstein, J. Kavanagh, A. Knight, J. May, T. Nguyen, and J.-R. Sack. A workbench for computational geometry. *Algorithmica*, 11(4):404–428, April 1994.

[12] Leslie Lamport. *A Document Preparation System LaTeX User's Guide and Reference Manual.* Addison Wesley, 1986.

[13] K. Mehlhorn and S. Näher. LEDA, a library of efficient data types and algorithms. Report A 04/89, Fachber. Inform., Univ. Saarlandes, Saarbr cken, West Germany, 1989.

[14] H. Rohnert. Moving a disc between polygons. *Algorithmica*, 6:182–191, 1991.

[15] S. Schirra. Moving a disc between polygons (video). In *The Ninth Annual ACM Symposium on Computational Geometry*, pages 395–396, May 1993.

[16] P. Schorn. *Robust Algorithms in a Program Library for Geometric Computation.* PhD thesis, Informatik-dissertationen eth zurich, 1992.

[17] R. Sedgewick. *Algorithms.* Addison Wesley, second edition, 1989.

[18] SiliconGraphics. *Iris Inventor Programming Guide.* 1992.

[19] J. Snoeyink. Objects that cannot be taken apart with two hands (video). In *The Ninth Annual ACM Symposium on Computational Geometry*, page 405, May 1993.

[20] J. Snoeyink and J. Stolfi. Objects that cannot be taken apart with two hands. In *The Ninth Annual ACM Symposium on Computational Geometry*, pages 247–256, May 1993.

[21] J. Stasko. The path-transition paradigm: a practical methodology for adding animation to program interface. *Journal of Visual Languages and Computing*, pages 213–236, 1990.

[22] J. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, September 1990.

[23] A. Tal, B. Chazelle, and D. Dobkin. The New–Jersey line–segment saw massacre (video). In *The Eighth Annual ACM Symposium on Computational Geometry*. ACM, 1992.

A Motion Planning Code

```
motion_planning(int N, float x1, float y1,
      float z1, float x2, float y2, float z2,
      float radius))
{
  int j;
  int points_no;
  float (*points)[3];
  char *obstacle_name;
  long *indices;

  /* get and display the obstacles*/
  Begin_atomic("POLYGON");
  for (j = 0; j < N; j++){
    get_obstacle(&obstacle_name,
       &points_no, &points, &indices);

    Create_polyhedron(obstacle_name,
       points_no, points_no+1, points, indices);
  }
  End_atomic();

  /* display the initial and final positions */
  Begin_atomic("goal");
  Create_sphere("start", x1, y1, z1, radius);
  Create_sphere("end", x2, y2, z2, radius);
  End_atomic();

  /* create the disc */
  Begin_atomic("ball");
  Create_sphere("sphr", x1, y1, z1, radius);
  End_atomic();

  /* the disc moves along a path */
  Begin_atomic("move");
  LinearPath_obj("sphr");
  End_atomic();
}
```

B Style File

The following style file was used for the example discussed in the Case Study Section. The style file determines the following aspects of the animation. The background color is light gray. The colors to be chosen by GASP are colors which fit the creation of a video (rather than the screen). Each atomic unit spans 30 frames, that is the operations within an atomic unit are divided into 30 increments of change. If the scene needs to be scaled, the objects will become 0.82 of their original size. Rotation of the world is done 20 degrees around the Y axis. The atomic unit `pluck` is executed over 100 frames, instead of over 30. The colors of the faces to be added in the atomic unit `add_faces` are light green.

```
begin_global_style
  background = 0.9 0.9 0.9;
  color = VIDEO;
  frames = 30;
  scale_world = 0.82 0.82 0.82;
  rotation_world = Y 20.0;
end_global_style
begin_unit_style pluck
  frames = 100;
end_unit_style
begin_unit_style add_faces
  color = 0.3125 0.5078125 0.3125;
end_unit_style
```