Visualization of Geometric Algorithms

Ayellet Tal, David Dobkin

Abstract— **This paper investigates the visualization of geometric algorithms. We discuss how limiting the domain makes it possible to create a system that enables others to use it easily. Knowledge about the domain can be very helpful in building a system which automates large parts of the user's task. A system can be designed to isolate the user from any concern about how graphics is done. The application need only specify "what" happens and need not be concerned with "how" to make it happen on the screen. We develop a conceptual model and a framework for experimenting with it. We also present a system, GASP, which implements this model. GASP allows quick generation of three-dimensional geometric algorithm visualizations, even for highly complex algorithms. It also provides a visual debugging facility for geometric computing. We show the utility of GASP by presenting a variety of examples.**

I. Introduction

The visualization of mathematical concepts goes back to the early days of graphics hardware [21], [2], and continues to the present [18], [16], [15], [19]. These videos use graphics and motion to explain geometric ideas in three dimensions and higher. They have been widely accepted as the necessary companions to the traditional medium of journal publication [32], [33]. Similar gains in exposition are found in the algorithm animation work that has become popular in recent years [1], [8], [5], [6], [27], [7], [24], [23], [22]. The limiting force has been the difficulty of generating the graphics for such animations.

The main principle guiding our work is that algorithm designers want to visualize their algorithms but are limited by current tools. In particular, visualizations would be less rare if the effort to create them was little. In the past, visualizations have been produced by developing sophisticated software for a particular situation but there has been little movement towards more widely usable systems.

By limiting our domain we are able to create such a system that enables others to use it easily. We have chosen the domain of computational geometry to build a system that greatly facilitates the visualization of algorithms regardless of their complexity. The visual nature of geometry makes it one of the areas of computer science that can benefit greatly from visualization. Even the simple task of imagining in the mind a three-dimensional geometric construction can be hard. In many cases the dynamics of the algorithm must be understood to grasp the algorithm, and even a simple animation can assist the geometer.

We describe in this paper our system, *GASP* (Geometric Animation System, Princeton). We present the conceptual model that underlies the development and implementation of our system, and we demonstrate its utility in a series of snapshots taken from a videotape [30].

Three major objectives set GASP apart from other animation systems (e.g., Balsa [8], Balsa-II [5], [6], Tango [27], and Zeus [7]).

- GASP allows the very quick creation of three dimensional algorithm visualizations. A typical animation can be produced in a matter of days or even hours. In particular, GASP allows the fast prototyping of algorithm animations.
- Even highly complex geometric algorithms can be animated with ease. This is an important point, because it is our view that complicated algorithms are those that gain the most from visualization. To create an animation, it is sufficient to write a few dozen lines of code.
- Providing a visual debugging facility for geometric computing is one of the major goals of the GASP project. Geometric algorithms can be very complex and hard to implement. Typical geometric code is often heavily pointer-based and thus standard debuggers are notoriously inadequate for it. In addition, running geometric code is plagued by problems of ro-

Ayellet Tal is with the Department of Applied Mathematics and Computer Science at the Weizmann Institute of Science, Rehovot, Israel. E-mail: ayellet@wisdom.weizmann.ac.il. This work was done at Princeton University.

David Dobkin is with the Department of Computer Science at Princeton University. E-mail: dpd@cs.princeton.edu.

bustness and degeneracies.

There are many ways in which the system can be used. First, it can be used simply as an illustration tool for geometric constructions. Second, stand-alone videotapes to accompany talks and classes can be created by GASP. Third, GASP can ease the task of debugging. Fourth, GASP can significantly enhance the study of algorithms by allowing students to interact and experiment with the animations. Fifth, GASP enables users to create animations to attach to their documents.

Computational geometers describe configurations of geometric objects either through ASCII text as generated by symbolic computing tools (e.g., Mathematica [34]) or through hand drawn figures created with a graphics editor. Our system offers an alternative to this by allowing the geometer to feed ASCII data into a simple program and get a three-dimensional dynamic (as well as static) visualization of objects.

Often, the dynamics of the algorithm must be understood. Animations can assist the geometer and be a powerful adjunct to a technical paper. With GASP, generating an animation requires no knowledge of computer graphics. The interaction with the system is tuned to the user's area of expertise, i.e., geometry.

Until recently, most researchers have been reluctant to implement, let alone visualize, their algorithms. In large part, this has been due to the difficulty in using graphics systems added to the difficulty of implementing geometric algorithms. This combination made it a major effort to animate even the simplest geometric algorithm. Our system can ease some of the unique hardships of coding and debugging geometric algorithms. The inherent difficulty in checking a geometric object (e.g., listing vertices, edges, and faces of a polyhedron) in a debugger can be eliminated once it becomes possible to view the object. In practice, a simple feature such as being able to visualize a geometric object right before a bug causes the program to crash is an invaluable debugging tool.

Visualization can have a great impact in education. Watching and interacting with an algorithm can enhance the understanding, give insight into geometry, and explain the intuition behind the algorithm. The environment in which the animation runs is designed to be simple and effective. The viewer is able to observe, interact, and experiment with the animation.

An important consideration in the design of GASP is the support of enclosures of animations in online documents. GASP movies can be converted into MPEG movies which can be included in Mosaic documents. The reader of such a document can click on the icon and see the animation. For the viewer it takes no more work to view an animation and the effect is better.

In the next section we describe the conceptual model upon which GASP was built. In Sections III and IV we present the specification of the system. We focus on the ways the system meets the needs of both the geometer and the viewer. In Section V we describe, through examples, how our system has been used in various scenarios. Section VI discusses some implementation issues. We summarize and mention open problems in Section VII. This paper is an enhanced version of [31].

II. Conceptual Model

Previous algorithm animation systems (e.g., [5], [6], [7], [27]) have dealt with the general case and thus have attempted to solve many problems at once. They have not made any assumptions about the type of objects and the kind of operations that make up the building blocks of the animations. As a result, no knowledge could be used in the creation of the animation. For example, suppose a user wants to animate a *sorting algorithm*. First, the user needs to decide how the elements should look - rectangles, cubes or maybe cylinders, and generate them. Then, the user has to design and implement the animation of the operations that make up the algorithm, in this case, the *compare* and *swap* operations. There are many possible ways to do it.

An animation system for a restricted domain can be vastly superior to a general-case system. Knowledge about the entities and the operations in this domain can be very helpful in building an animation system which produces animations significantly more easily. Appropriate ways to visualize the entities and to animate the operations can be embedded in the system. Thus, large parts of the user's task can be automated. In this case, a system can be designed to isolate the user from any concern about how graphics is done.

One of the major departures of our work from previous work is the elimination of the animator. We define a conceptual model which allows us to do this. The main principle behind our model is that programmers should be freed from having to design and implement the visual aspects of the animation, and can concentrate solely on the contents of the animation. This is important not only because the job of implementing an animation is time-consuming, but also because it involves graphics design, an area the user is not usually familiar with.

The ability to automate the process of generating animations is very useful for most users. However, some might find it too restrictive and would like to be able to change it. We therefore define a hierarchy of users. While previous systems identified two types of clients, *end-users* and *client-programmers*, we identify three distinct user types for any such system, *end-users*, *naive-programmers*, and *advanced-programmers*.

1. As before, end-users want to experiment with an algorithm to understand its functioning. End-users should be able to run the application (i.e., see the animation) as an interactive experience. That is, it should be possible to play the animation at slow or fast speed, to run it backwards, to pause and alter the objects being considered, and to run the animation on an input of the user's choosing among other things.
2. Naive application programmers want a system which makes generic animation of algorithms as easy as possible. The naive-programmer is not concerned with the presentation aspects of the animation and can choose to be isolated from any decisions of a graphical nature. Typically, the naive programmer needs the animation for one of three purposes. The animation aids in the debugging process, it helps for exploring research ideas, or it serves as a prototype animation and will be refined later on.
3. Advanced programmers want, in addition, to be able to easily modify and extend various visualization aspects of the animation. A major concern when generating animations automatically is that the outcome might be different from what is most useful to the user and might not fit the programmer's taste. It is therefore necessary to provide a way to modify the animation. The advanced programmer should be able to change the style of the animation without having to implement additional code.

To understand these levels, one can draw an analogy with document processing systems. The end-user need not know how the animation was produced. By analogy, the reader of a document does not care about how it was created. The second user is the application writer. The application writer is analogous to the text preparer who typically uses the default style settings of a system. The creator of the document is concerned more with the text the paper includes and less with the visual aspects such as the selection of margins, spacings, and fonts. Similarly, the application writer is concerned with the contents of the animation rather than with its visual aspects. Finally, there are times when the creator of the animation does want to change the viewing aspects (e.g., colors) of the animation. By analogy, there are times when the writer of a document would like to change fonts and margins. Systems such as LaTeX [17] provide this flexibility. The user can change many defaults by creating personal style files of LaTeX. To do it, the user needs additional knowledge. This is the third type of user, the advanced programmer.

We define a similar interface for animating algorithms. The interface we propose in response to these needs consists of library calls for the naive-programmer and external ASCII *style files* for the advanced programmer. The idea of using style files in not new in computer graphics (e.g., see [3]). Its use in animation systems, however, is novel. The naive-programmer writes short snippets of C code to define the structure of the animation. The animation system knows how to generate an appropriate animation from this C code. The advanced programmer can edit an ASCII style file to control the visual aspects of the animation. The animation is still generated automatically by the animation system. But, a different animation will be created, if the style file is modified. Editing the style file allows experimentation with various animations for a given algorithm.

Thus, any animation has four components:

- The algorithm animation system.

- The algorithm implementation.
- Hooks to the animation system within the algorithm implementation.
- Style files.

The programmer need never be concerned with the algorithm animation system. The algorithm implementation is something that would have to be done anyways. Creating the hooks is the main task of the animator. The use of style files is optional.

We believe that the model we suggest is general enough and can be applied to other constrained domains. Only the types of objects and supported operations should be replaced. The structure of the system and the user interfaces should not be changed.

III. GASP's Language

GASP is an algorithm animation system for the domain of computational geometry, which implements the conceptual model presented in the previous section.

Recall that there are two types of programmers: naive-programmers and advanced-programmers. Naive-programmer are concerned only with the contents of the animation, whereas advanced-programmers care also about the visual aspects of the animation. Naive-programmers need to write brief snippets of C code to define the structure of the animation. The code includes only manipulations of objects and modifications of data structures. This code contains calls to GASP's library. Style files are used by advanced-programmers to change from default aspects of the animation to other options. In this section we introduce the programmer interface that GASP provides.

A. Naive-Programmer Interface

GASP's library is a set of building blocks that enable us to write animations with minimal effort. All we need to do is write short snippets of C code, and GASP makes sure that they are powerful enough to generate an animation. To do this, we follow two principles: First, the programmer does not need to have any knowledge of computer graphics. Second, we distinguish between *what* is being animated and *how* it is animated. The application specifies what happens *(The What)* and need not be concerned with how to make it happen on the screen *(The How)*. For example, the creation of a polyhedron is different from the way it is made to appear through the animation. It can be created by fading into the scene, by traveling into its location, etc. The code includes only *the what*, but not any of the visualization issues, such as the way each operation animates, the look of the objects, or the colors.

The scenes of interest to us are built out of geometric objects and displays of data structures. Typical geometric objects are lines, points, polygons, spheres, cylinders and polyhedra. Typical data structures include lists and trees of various forms. The operations applied to these objects depend upon their types. A standard animation in the domain of computational geometry is built out of these building blocks.

The parameter data required by GASP is part of the algorithm being animated. To make its use easy, GASP requires only very simple data types: integers, floats, chars, arrays of integers or floats, and strings. We avoid using more complex data structures (e.g., a more complex data structure which represents a polyhedron) in order to keep the start-up time minimal.

GASP's library contains four classes of operations - operations on objects, atomic units, motion, and undo.

A.1 Operations on Objects

GASP's objects include three-dimensional geometric objects, two dimensional geometric objects, combinatorial objects, views, text, and titles. Objects can be created, removed, and modified. They can also be copied, grouped, and ungrouped.

We use the `Create_XXX` function to create an object of the type `XXX`. Each Create function has different parameters, which are suitable to the object being created. For example, to create a line GASP expects the two end-points of the line as parameters whereas to create a polyhedron GASP needs the number of vertices of the polyhedron, the number of faces, the specification of the vertices, and that of the faces. Each `Create` function has its own default way to animate. A polyhedron fades into the scene, a point blinks in order to attract attention, and a tree is created level after level starting from the root.

We use `Remove_object` to remove an object from

the scene. An object is removed in the reverse fashion to the way it is being created. For example, a polyhedron fades out from the scene, a point blinks, and a tree is removed level after level, starting from the leaves and working its way to the root.

Each object is related to one or more modification functions which are appropriate for this object. For example, we can add faces to a polyhedron, but naturally there is no equivalent operation for atomic objects such as spheres.

`Copy_object` creates an exact copy of the object. This function is very useful when displaying an algorithm in multiple views. We can create multiple copies of the objects, and manipulate them in distinct ways in the various views.

The `Group` function creates an object which contains an ordered list of child objects. Grouping allows us to isolate effects (e.g., motion) to a specific set of objects. The reverse function, `Ungroup`, is also available.

Three-dimensional geometric objects: Typical objects embedded in three-space include spheres, cylinders, cubes, cones, planes, lines, points, line-sets, point-set, sweep-lines, polyhedra, and meshes. A mesh represents a three-dimensional shape formed by constructing faces from given vertices.

Meshes and polyhedra are unique objects, being non-atomic. Six special functions on meshes are supported by GASP: `Split_mesh` removes vertices from a mesh, together with their related cones of faces. This operation is animated by first creating new meshes - a cone for every removed vertex. The new cones travel away from the initial mesh, creating black holes in it. Each cone travels in the direction of the vector which is the difference between the vertex (which created the cone) and the center of the split mesh. `Attach_mesh_to_mesh` attaches a few meshes to each other. This operation is visualized in the reversed way to the split operation. The meshes travel towards a chosen mesh, until they meet. `Bind_mesh_to_mesh` is similar to the attach operation, with one difference. At the end of the binding process, we get a single object. `Add_faces` adds new faces to a mesh and is displayed, by default, by fading in. `Remove_faces` is the opposite operation. `Add_vertices` adds new vertices to a given mesh.

Two-dimensional geometric objects: Typical objects embedded in two-dimensions include circles, rectangles, elliptic arcs, lines, points, line-sets, point-set, sweep-lines, polygons, and splined-polygons.

Here again, GASP supports a few types of displays for each object, one of them is the default. For example, a polygon can be filled or not, two-dimensional objects can be highlighted by using related three-dimensional objects (e.g., cylinders can highlight edges and spheres can highlight vertices), etc.

Combinatorial objects: Combinatorial objects include lists and trees of various types (binary or not, red-black trees etc.).

Unlike geometric objects, combinatorial objects do not have an evident visual representation. A tree can either be presented in two dimensions or in three. GASP can layout a tree in both ways, but will usually prefer the novel three-dimensional style in which the nodes which belong to the same level of the tree reside on a single cycle; the radius of the cycles increase as the level of the tree increases. The creation of a tree is visualized, by default, by fading in the nodes, level after level, starting at the root. Similarly, lists can be displayed in two dimensions (e.g., using rectangles) or in three dimensions (e.g., using cubes).

In addition to the creation and deletion of trees and lists, GASP supports the addition of nodes, and the removal of nodes and subtrees.

Views: A view is more than a window used for rendering. Built on top of Inventor's Examiner-viewer [28], a view contains a camera and a light model. It also contains buttons and thumbwheels that allows use of the mouse to modify the camera placement in the scene.

Text and titles: Text objects and title objects define text strings to be rendered to the screen. We can annotate our graphics with text. Titles ease the creation of videotapes.

By default, we use two-dimensional text and titles, though three-dimensional is supported as well. Text appears on the screen as one unit, while

titles show up line by line. The default fonts and font sizes vary.

A.2 Atomic Units

Every logical phase of an algorithm can involve several operations, which should be animated concurrently. We can isolate phases of the algorithm by grouping primitives into logical phases, called atomic units. We use the `Begin_atomic - End_atomic` phrase to enclose the operations which belong to the same logical phase, and GASP executes their animation as a single unit.

For example, if adding a new face to a polyhedron, creating a new plane, and rotating a third object constitute one logical unit, these operations are animated as one unit. GASP would concurrently fade in the new faces of the polyhedron, fade in the plane, and rotate the cylinder. The code that generates this animation is:

```
Begin_atomic("Example");
Add_faces("Poly", face_no, faces);
Create_plane("Plane", point1, point2,
             point3, point4);
Rotate_object("ThirdObj");
End_atomic();
```

Some properties of atomic units: Like any other object in the system, atomic units are named. Using names (rather than IDs) not only makes the interaction between the programmer and the system more natural, but also allows the end-user to follow the unfolding of the algorithm by listing the names of algorithm's atomic units (assuming that appropriate names have been used). Atomic units can be nested. To nest atomic units within each other, we use the `Start_late` or the `Finish_early` functions. `Start_late` declares when, within the nesting atomic unit, GASP should start animating the current unit. `Finish_early` declares when, within the nesting unit, GASP should terminate the animation of the current atomic unit. Finally, each atomic unit can be accompanied with text and voice which elucidate the events happening during this unit. Since an atomic unit represents a logical phase of the algorithm, this is the appropriate unit to attach explanations to.

A.3 Motion

Smooth motion is a major component of any animation. Motion can be applied to either a single object (or a set of objects, after grouping them), or to the camera. When the camera moves, the whole scene changes.

GASP supports five types of motion. We use the `Rotate_obj` or the `Rotate_world` primitives in order to rotate an object or the camera respectively in terms of an axis and an angle. We use `Scale_world` or `Scale_obj` to scale an object in x, y, z factors. We use `Translate_world` or `Translate_obj` to move an object in x, y, z. We use `LinearPath_world` or `LinearPath_obj` to float an object on a linear path. We use `Path_world` or `Path_obj` to float an object on a Bézier curve. For the last two operations, we need only specify the positions through which the object moves and GASP calculates the exact path through which an object floats.

All the motion primitives are visualized smoothly. For example, a rotation is done gradually, until the desired angle is achieved.

A.4 Undo

We use the undo operation to play the animation backwards. The undo operation takes as a parameter the number of atomic units to be reversed. GASP knows how to reverse visually each primitive within an atomic unit.

B. Advanced-Programmer Interface

Each operation supported by GASP generates a piece of animation which demonstrates the specific operation in a suitable way. If a programmer wants freedom to accommodate personal taste, the parameters of the animation can be modified by editing a *"Style File"*. The animation is still generated automatically by the system but a different animation will be generated if the style file is modified. The style file affects the animation, not the implementation.

Large number of parameters can be changed in the style file. Those parameters can be set either globally, for the whole animation, or for each atomic unit separately. We describe some of these parameters here.

B.1 Visualizing Primitives

Each primitive supported by GASP can be animated in several ways, one of which is the default that GASP chooses. However, a parameter can be set in the style file to change from a default visualization to an optional one.

For instance, objects can be created in various ways: by fading in, by scaling up to their full size, by traveling into the scene, by blinking, by growing - adding one feature after the other (e.g., a tree grows level after level, a mesh grows by adding the faces one at a time), or by appearing at once at the scene. A reasonable subset of the above visualizations is allowed for each of the objects. We can choose our favorite option by editing one line in the style file.

B.2 Visualizing Objects

Objects can be rendered in various fashions. For example, numerous ways exist to present meshes. A mesh can be flat, smooth, or wire-framed. The edges of a mesh can be displayed or not. The same is true for its vertices. A mesh can be opaque or transparent to some degree. Different normals defined for the faces of the mesh influence the colors of the faces. We can modify all those parameters.

Special attention is given to the issue of colors. "Color is the most sophisticated and complex of the visible language components" [20]. GASP chooses colors for the objects and for the features it creates. GASP maintains palettes of preselected colors, and picks colors which are appropriate for the device they are presented on (i.e., screen or video). This is especially important for inexperienced users.

Colors are assigned to objects (or other features such as faces of a polyhedron) on the basis of their creation time. That is, every logical phase of the algorithm is associated with an unused color, and the objects created during that phase get this color. This scheme allows us to group related elements and to make it clear to the observer how the algorithm progresses from phase to phase. Those colors can be changed in the style file.

B.3 Visualizing Motion

The parameters for the motion operations can be also altered in the style file. We can change the axis of the rotation and its angle, the amount of translation or scale, the number of key-frames of a path, etc.

B.4 Miscellaneous

There are many other important parameters for any animation. For instance, we are able to specify in the style file whether the animation is running on the screen or on the video. This is so, because colors look very different on both devices. If we want colors that look good on a video, we must use less saturated colors. GASP knows how to generate appropriate set of colors.

As another example, we can add one line to the style file which tells GASP to stop after every frame and execute a given script file. We found this option to be very useful. We could record a movie frame by frame by writing a suitable script file. We could generate MPEG movies from GASP movies, by providing yet another script file that converted each frame.

B.5 Style File Example

The following is part of the style file for an animation which will be discussed in a later 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 green.

```
begin_global_style
  background = light_gray;
  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 = green;
```

```
end_unit_style
```

Note that the syntax of the style file is eminently simple.

IV. GASP's Environment

The interactive environment is a primary part of the GASP system. It allows researchers, programmers, and students to explore the behavior of their geometric algorithms. It is designed to be simple and effective, and to allow the viewer to observe, interact, and experiment with the animation.

The GASP environment, illustrated in Fig. 1, consists of a *Control Panel* through which the student controls the execution of the animation, several windows where the algorithm runs, called the *Algorithm Windows*, along with a *Text Window* which explains the algorithm.

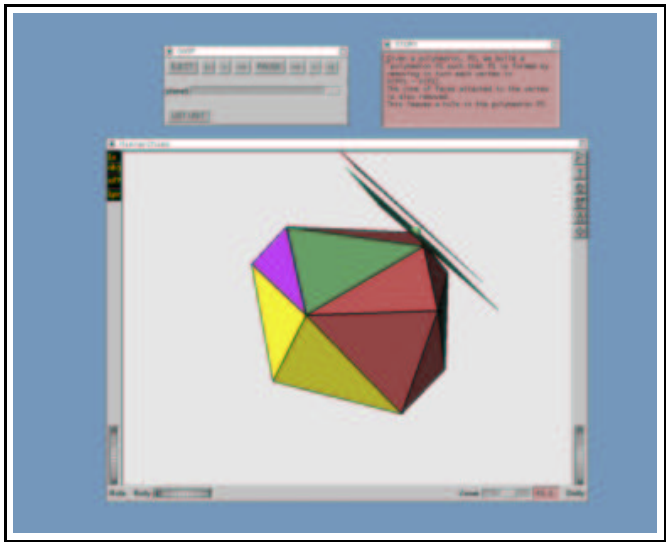

Fig. 1. GASP's Environment

A. The Control Panel

The control panel, at the upper left of Fig. 1, lets us explore the animation at our own pace. It uses the VCR metaphor, to make the interaction intuitive, familiar, and easy.

We might want to stop the animation at various points of its execution. Sometimes we would like to fast-forward through the easy parts and single-step through the hard ones to facilitate our understanding. We may want to "rewind" the algorithm in order to observe the confusing parts of the algorithm multiple times. We may need to *PAUSE* at any time to suspend the execution of the algorithm or to *EJECT* the movie. GASP's environment allows us to do all these.

B. The Algorithm Window

We observe the algorithm in the algorithm windows (at the bottom of Fig. 1). Algorithm windows use Inventor's *Examiner-Viewer* [28] and thus are decorated with thumbwheels and push buttons.

Thumbwheels let us rotate and scale the scene. We use the left thumbwheel for a screen X rotation. We use the bottom thumbwheel for a screen Y rotation. We use the right thumbwheel for dolly (in and out of screen). We use the zoom slider on the bottom to change the camera height (orthographics) or the height-angle (perspective).

The push buttons at the right-hand side of the algorithm window do the following operations. We click the *help* button to display a help card for the viewer. We push the *home* button to reset the camera to a "home" position. We push the *set home* button to set a new home position. We click the *view all button* to reposition the camera so that all objects become visible. The *seek* button makes the camera animate to the center of the selected object.

The left-hand side push buttons give us information about the algorithm and the animation. The *ls* button lists the objects currently appearing on the screen. The *obj* button prints a description of a chosen object. For example, when a polyhedron is picked, its vertices and faces are printed out. The *lu* button lists the atomic units. The *xf* button prints the current transformation of either a selected object or the global transformation. The *lpr* button creates a snapshot file of the screen. Using the *lpr* option, we can create pictures to annotate our papers.

C. The Text Window

We can read about the algorithm and the animation in the text window (at the upper right of Fig. 1). The text window lets the client-programmer accompany the animation running on the screen with verbal explanations. Text can elucidate the events and direct the viewer'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. Voice is also supported by GASP. The viewer can listen to the explanations that appear in the text window.

V. GASP in Action

In this section we describe different scenarios for which we produced animations to accompany geometric papers. Excerpts from the animations are given in a videotape [30]. For each case we present the problem of study, the goal in creating the animation and the animation itself.

A. Building and Using Polyhedral Hierarchies

This algorithm, which is based on [11], [12], builds an advanced data structure for a polyhedron and uses it for intersecting a polyhedron and a plane. The main component of the algorithm is a preprocessing method for convex polyhedra in 3D which creates a linear-size data structure for the polyhedron called its *Hierarchical Representation*. Using hierarchical representations, polyhedra can be searched (i.e., tested for intersection with planes) and merged (i.e., tested for pairwise intersection) in logarithmic time. The basic geometric primitive used in constructing the hierarchical representation is called the *Pluck*: Given a polyhedron, P_0, we build a polyhedron, P_1, by removing vertices in $V(P_0) - V(P_1)$. The cones of faces attached to the vertices are also removed. This leaves holes in the polyhedron P_0. These holes are retriangulated in a convex fashion. Repetition of *plucking* on the polyhedron P_1 creates a new polyhedron, P_2. The sequence $P_0\ P_1\ P_2\ ...\ P_n$ forms the hierarchical representation.

There were two goals for creating the animation ([13]). First, we wanted to create a video that explains the data structure and the algorithm for educational reasons. Second, since the algorithm for detecting plane-polyhedral intersection had not been implemented before, we wanted the animation as an aid in debugging the implementation.

The animation explains how the hierarchy is constructed and then how it is used. For the first of these we explain a single pluck and then show how the hierarchy progresses from level to level.

First, we show a single pluck. The animation begins by rotating the polyhedron to identify it to the user (Fig. 2). Next we highlight a vertex and lift its cone of faces by moving them away from the polyhedron (Fig. 3). Then, we add the new triangulation to the hole created (Fig. 4). Finally, we remove the triangulation and reattach the cone, to explain that plucking is reversible.

This is done in our system by the following piece of C code, which is up to the creator of the animation to write.

```
explain_pluck(int poly_vert_no,
        float (*poly_vertices)[3],
        int poly_face_no,
        long *poly_faces,
        char *poly_names[],
        int vert_no, int *vertices,
        int face_no, long *faces)
{
  /* create and rotate 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 plucking */
  Undo(2);
}
```

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. `Undo` removes the triangulation and brings the cone back to the polyhedron.

Notice that the code does not include the graph-

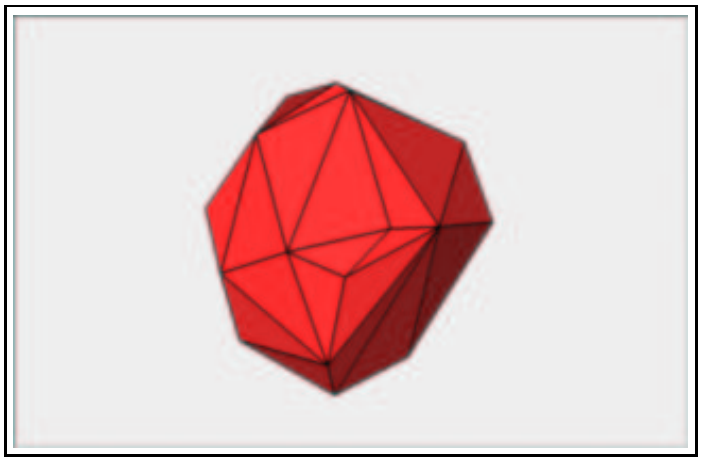

Fig. 2. The Polyhedron

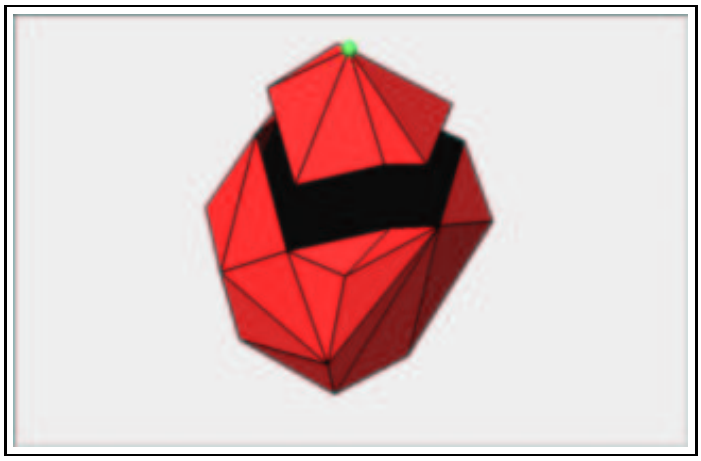

Fig. 3. Removing the Cone of Faces

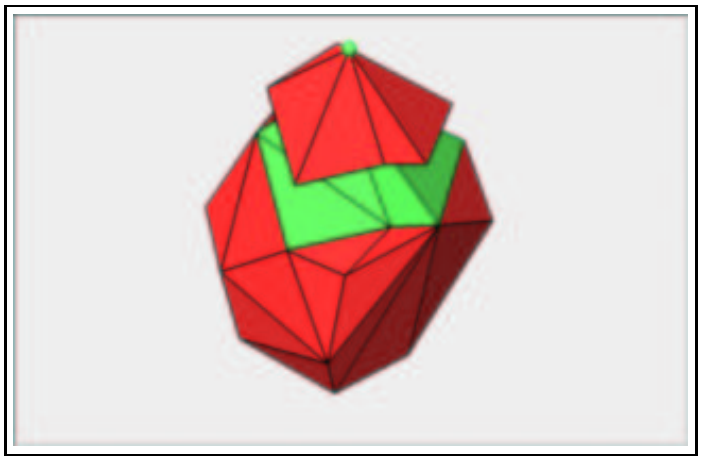

Fig. 4. Retriangulating the Polyhedron

ics. Coloring, fading, traveling, speed, etc. are not mentioned in the code. In the related style file these operations are controlled. This allows the user to experiment with the animation without modifying and recompiling the code.

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(
     char *atomic1_name,
     char *atomic2_name,
     char *atomic3_name,
     char *poly_name,
     int vert_no, int *vertices,
     int face_no, long *faces,
     char *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.

We found GASP to be very helpful in implementing the algorithm for detecting plane-polyhedron intersections. Bugs we were not aware of showed up in the animation (e.g., we got non-convex polyhedra as part of the hierarchical representation). We also found GASP's environment to be very useful. When debugging the algorithm,

it is necessary to watch earlier stages of the animation (the construction process) which set state variables that are needed by later stages. The control panel of GASP allows us to fast-forward over these initial fragments to get to the section of interest. Single-stepping through the section under consideration and rewinding are also highly valuable tools.

B. *Objects that Cannot be Taken Apart with Two Hands*

This animation is based on [26]. This paper shows a configuration of six tetrahedra that cannot be taken apart by translation with two hands (Fig. 5). Then, it presents a configuration of thirty objects that cannot be taken apart by applying an isometry to any proper subset (Fig. 6). The ASCII data of the configurations was produced by using Mathematica.

The purpose of the animation is to illustrate the use of GASP as an illustration tool for geometric configurations. It took us far less than a day to generate that animation. The increased understanding from a moving animation is significant.

The animation has two parts. Each one of them shows one of the configurations described above. Each part begins by fading each object which belong to the configuration, in turn, into the scene. The colors of the objects vary. After all the objects appear in the scene, the scene rotates so that the configuration as a whole can be examined.

The animation is produced by the following brief C function. 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 object. `Rotate_world` causes the scene to spin.

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

  for (i = 0; i < object_no; i++){
    /* object i */
    get_polyhedron(&points, &indices,
          &nmax, &fmax, &atomic_name,
          &object_name);

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

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

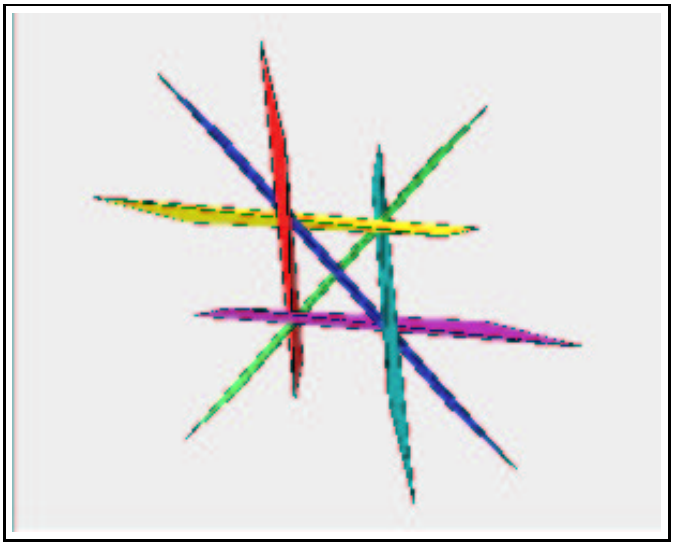

Fig. 5. Objects that Cannot be Taken Apart with Two Hands Using Translation

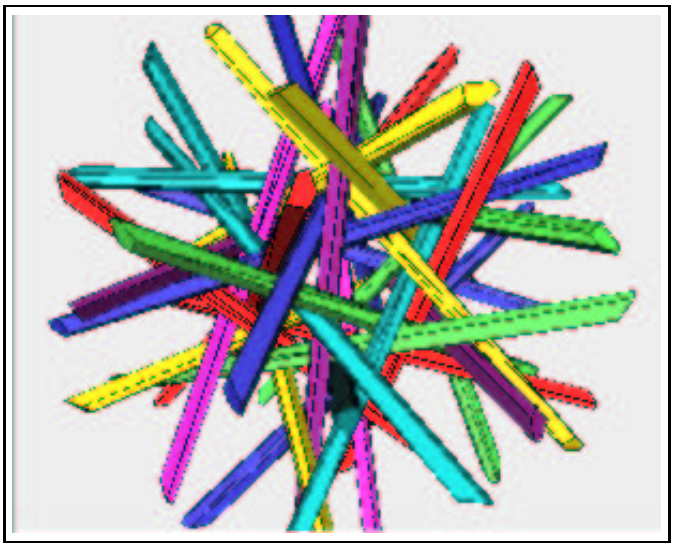

Fig. 6. Objects that Cannot be Taken Apart with Two Hands Using Isometries

C. Line Segment Intersections

This example, which is based on [9], is a short clip from an animation ([29]) which shows a line segment intersection algorithm in action and illustrates its most important features. The goal is to use the animation as an aid in explaining a highly complex algorithm. The viewer of the animation can not only control the execution of the animation but can also choose the input by editing an ASCII file containing the initial line segments. This example also illustrates the use of GASP in creating two-dimensional animations. In a matter of days we generated the animation.

The animation runs in three phases. The first phase presents the initial line segments and the visibility map that needs to be built (Fig. 7). 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 visibility map of the region swept along the way (Fig. 8). 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 (Fig. 9).

In the first pass of the animation, red line segments fade into the scene. While they fade out, a green visibility map fades in on top of them, to illustrate 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 and the second pass through the algorithm begins.

During the second pass the viewer can watch as the sweep-line advances by rolling to its new position (the gray line in Fig. 8). The animation also demonstrates how the map is built - new subsegments fade in in blue, and then change their color to green to become a part of the already-built visibility map.

The third pass adds more information about the process of constructing the map by showing how the the red-black tree which is maintained by the algorithm changes. The animation also presents the "walks" on the map (marked in yellow in Fig. 9).

There are only eleven GASP's calls necessary for

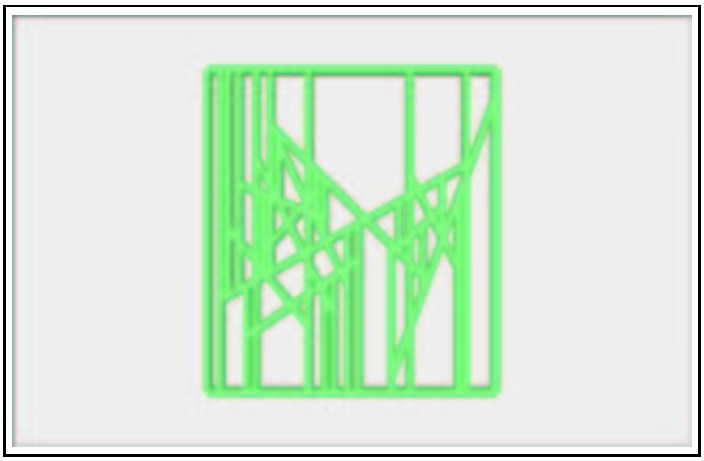

Fig. 7. The Visibility Map

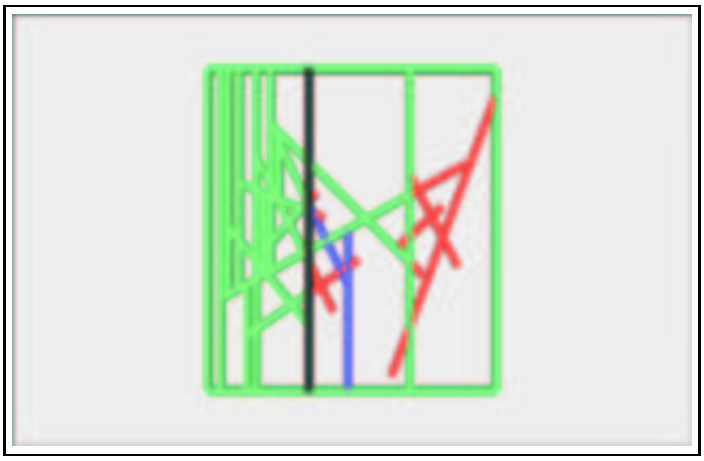

Fig. 8. Building the Visibility Map

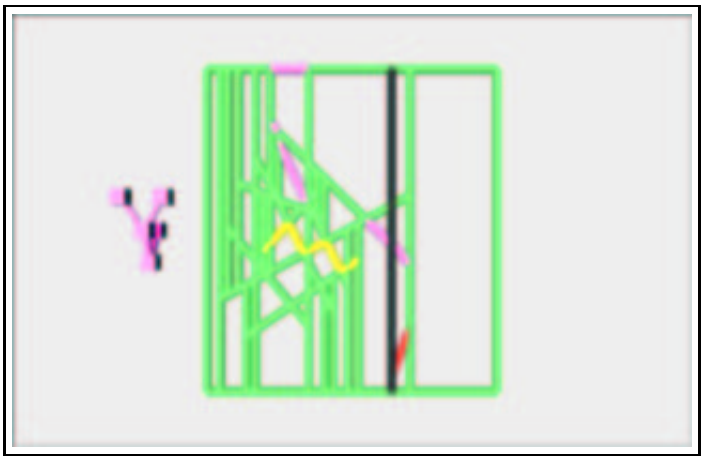

Fig. 9. Maintaining the Cross Section

the creation of this animation and they are: `Begin_atomic`, `End_atomic`, `Rotate_world`, `Scale_world`, `Create_point`, `Create_line`, `Create_Sweepline`, `Modify_Sweepline`, `Create_tree`, `Add_node_to_tree`, `Remove_object`.

D. Heapsort

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 geometry, among them many of the algorithms in [25]. To show the added power of the system, we chose to animate heapsort.

Heapsort is an efficient sorting algorithm that is defined from the basic operations on heaps. The idea is to build a heap containing the elements to be sorted and then remove them all in order.

In the animation, 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 (Fig. 10). The next step of the animation is to build a heap out of the tree in a bottom up fashion (Fig. 11). 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 (Fig. 12).

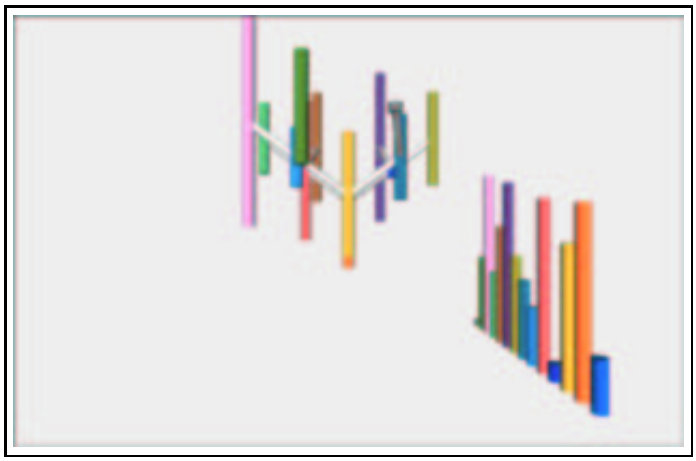

Fig. 10. Two Views of the Array

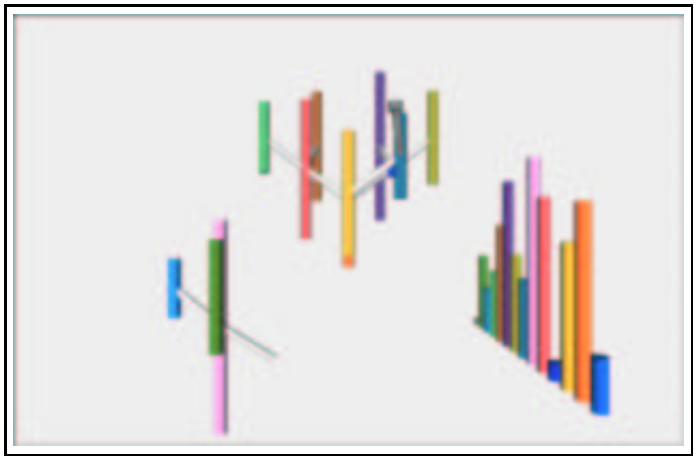

Fig. 11. Building the Heap

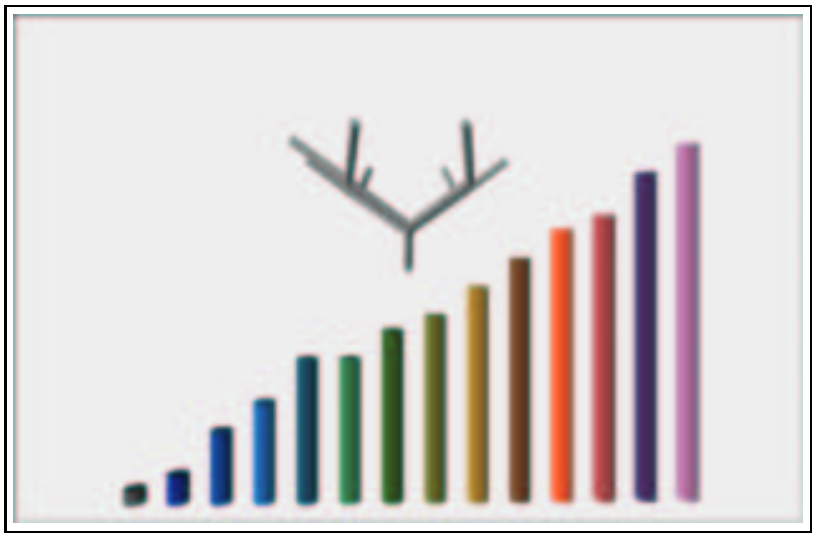

Fig. 12. The Sorted Array

VI. Implementation

GASP is written in C and runs under UNIX on a Silicon Graphics Iris. It is built on top of Inventor [28] and Motif/Xt [14].

GASP consists of two processes which communicate with each other through messages, as shown in Fig. 13. Process 1 includes the collection of procedures which make up the programmer interface. Process 2 is responsible for executing the animation and handling the viewer's input.

The application's code initiates calls to procedures which belong to Process 1. Process 1 prepares one or more messages containing the type

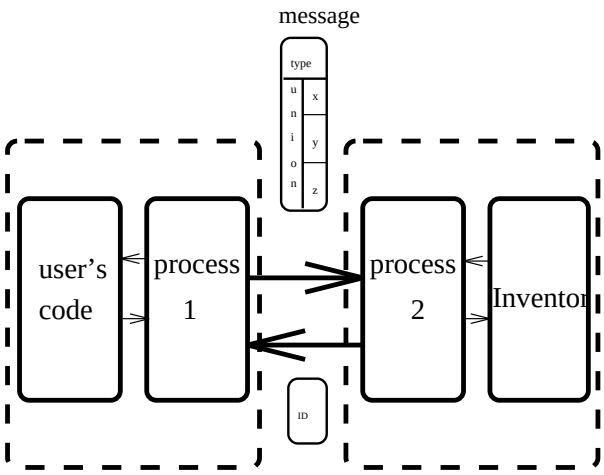

Fig. 13. GASP's Architecture

of the operation required and the relevant information for that operation and sends it to Process 2. Upon receiving the message, Process 2 updates its internal data structure or executes the animation, and sends an acknowledgement to Process 1. The acknowledgement includes internal IDs of the objects (if necessary). Process 1, which is waiting for that message, updates the hash table of objects and returns to the application's code.

This hand-shaking approach has a few advantages. First, it enables the user to visualize the scene at the time when the calls to the system's functions occur and thus facilitates debugging. Since rendering is done within an event mainloop, it is otherwise difficult to return to the application after each call. Second, compilation becomes very quick since the 'heavy' code is in the process the application does not link to. Finally, the user's code cannot corrupt GASP's code and vice versa. This is an important point, because one of the major goals of GASP is to ease debugging. During debugging, it is always a problem to figure out whose bug is it - the application's or the system's.

Process 2, which is responsible for the graphics, works in an event mainloop. We use Inventor's Timer-Sensor to update the graphics. This sensor goes off at regular intervals. Every time it goes off, Process 2 checks which direction the animation is running. If it is running forwards, it checks whether there is still work to do updating the animation (if yes, it does it) or it is at the point when further instructions from Process 1 are needed. In the latter case, it checks to see whether there is a message sent by Process 1. It keeps accepting messages, updating its internal data structure, and confirming the acceptance of messages until it gets an END_ATOMIC message. At that point, Process 2 starts executing all the commands specified for the atomic unit. It informs the first process upon termination. If the animation is running backwards, it updates the animation according to the phase it is in.

VII. Conclusions

GASP has been built as an animation system for computational geometry. Geometric algorithms can be highly complex, hard to implement and debug, and difficult to grasp. The visual nature of geometry makes animations extremely helpful. Researchers can use the system as an aid in exploring new ideas; programmers can use it as a debugging tool; students can enhance their understanding of the studied algorithm and get some intuition into the way it operates.

GASP is a demonstration of a concept. Picking a small domain makes it possible to create an animation system that enables others to use it easily. In a well-defined domain, we can use knowledge about the kinds of objects and operations that need to be visualized. In this case, it becomes practical to hide the graphics system from the user and to automate the creation of the animation. All the user needs to specify is the logical operations that need to be visualized (i.e., the *what*), but not how to do it (i.e., the *how*).

We also recognize that any algorithm animation system has various types of users with differing needs. The naive programmer would like to produce a "quick-and-dirty" animation to check out ideas or for debugging purposes. The naive programmer need not have any knowledge of computer graphics. The code includes only manipulations of objects and modifications of data structures. The algorithm animation system makes heuristic guesses for the way the animation should appear. The advanced programmer would like to have a say in the way the animation looks. The advanced programmer experiments with the animation by editing an ASCII style file, without ever modifying or compiling the code. The end-user would like to experiment with a finished animation. An algorithm animation system should serve these varying levels of user-types by providing dis-

tinct interfaces. GASP supports these levels.

Limiting the domain and providing multiple suitable interfaces make it possible to create an algorithm animation system that allows users to quickly create animations. With GASP, a typical animation can be generated in a very short time. This is true even for highly complex geometric algorithms. This is important because complex algorithms are those that benefit the most from being visualized.

We have shown several animations of geometric algorithms. The system is now at the stage where other people are starting to use it. In fact, three [4], [10], [30] out of the eight segments of animations which appeared in the Third Annual Video Review of Computational Geometry were created by GASP. Two of them were created by the geometers who made movies describing their newly discovered algorithms. They took less than a week to produce. We consider it to be a very short time for a first use of a system. The system is now available for ftp.

In the future, GASP can be expanded to support four-dimensional space. This can be an invaluable tool for research and education. We would like to experiment with GASP in an actual classroom. We believe that animations can be used as a central part of teaching computational geometry, both for demonstrating algorithms, and for accompanying programming assignments. Finally, many intriguing possibilities exist in making an electronic book out of GASP. A user will then be able to sit on the network, capture an animation, and experiment with the algorithm.

We believe that reducing the effort involved in creating animations will increase their proliferation. We hope that GASP is a first step in the creation of animation systems for constrained domains. Visualization can apply to many focused enough domains such as topology, databases, and networks.

Acknowledgements

We would like to thank Bernard Chazelle for numerous discussions and great advice.

This work was supported in part by the National Science Foundation under Grant Number CCR93-01254, by The Geometry Center, University of Minnesota, an STC funded by NSF, DOE, and Minnesota Technology, Inc., and by DIMACS, an STC funded by NSF.

References

[1] R.M. Baecker. Sorting out sorting (video). In *SIGGRAPH Video Review 7*, 1981.

[2] T. Banchoff and C. Strauss. *Complex Function Graphs, Dupin Cylinders, Gauss Map, and Veronese Surface*. Computer Geometry Films. Brown University, 1977.

[3] R. Beach and M. Stone. Graphical style: Towards high quality illustration. In *Computer Graphics (Proc. SIGGRAPH '83)*, pages 127–135, July 1983.

[4] H. Brönnimann. Almost optimal polyhedral separators (video). In *Third Annual Video Review of Computational Geometry*, June 1994.

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

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

[7] M.H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *1991 IEEE Workshop on Visual Languages*, pages 10–17, October 1991.

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

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

[10] D. Dobkin and D. Gunopulos. Computing the rectangle discrepancy (video). In *Third Annual Video Review of Computational Geometry*, June 1994.

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

[12] D. Dobkin and D. Kirkpatrick. Determining the separation of preprocessed polyhedra – a unified approach. *ICALP*, LNCS 443, pages 400–413, 1990.

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

[14] Open Software Foundation. *OSF/Motif - Programmer's Reference*. Prentice Hall, Inc., 1991.

[15] C. Gunn. Discrete groups and visualization of three-dimensional manifolds. In *Computer Graphics (Proc. SIGGRAPH '93)*, pages 255–262, August 1993.

[16] C. Gunn and D. Maxell. *Not Knot (video)*. Jones and Bartlett, 1991.

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

[18] D. Lerner and D. Asimov. The sudanese mobius band (video). In *SIGGRAPH Video Review*, 1984.

[19] S. Levy, D. Maxwell, and T. Munzner. Outside in (video). In *SIGGRAPH Video Review*, 1994.

[20] A. Marcus. *Graphics Design for Electronic Documents and User Interfaces*. ACM Press.

[21] N. Max. *Turning a Sphere Inside Out (video)*. International Film Bureau, 1977.

[22] M.A. Najork and M.H. Brown. A library for visualizing combinatorial structures. In *Proc. '94 Visualization*, pages 164–171, October 1994.

[23] B.A. Price, R.M. Baecker, and I.S. Small. A principles taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1993.

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

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

[26] J. Snoeyink and J. Stolfi. Objects that cannot be taken apart with two hands. In *The Ninth Annual ACM Sym-*

posium on Computational Geometry, pages 247–256, May 1993.

[27] J. Stasko. TANGO: A framework and system for algorithm animation. *Computer,* 23(9):27–39, September 1990.

[28] P.S. Strauss and R. Carey. An object-oriented 3D graphics toolkit. In *Computer Graphics (Proc. SIGGRAPH '92)*, pages 341–349, July 1992.

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

[30] A. Tal and D. Dobkin. GASP – a system to facilitate animating geometric algorithms (video). In *Third Annual Video Review of Computational Geometry*, June 1994.

[31] A. Tal and D. Dobkin. GASP – a system for visualizing geometric algorithms. In *Proc. '94 Visualization*, pages 149–155, October 1994.

[32] J.E. Taylor. *Computing Optimal Geometries.* Selected Lectures in Mathematics, American Mathematical Society, 1991.

[33] J.E. Taylor. *Computational Crystal Growers Workshop.* Selected Lectures in Mathematics, American Mathematical Society, 1992.

[34] S. Wolfram. *Mathematica - A System for Doing Mathematics by Computer.* Addison-Wesley Publishing Company, 1988.