Obliq-3D: A High-Level, Fast-Turnaround 3D Animation System

Marc A. Najork and Marc H. Brown

***Abstract*—This paper describes Obliq-3D, a high-level, fast-turnaround system for building 3D animations. Obliq-3D consists of an interpreted language that is embedded into a 3D animation library. This library is based on a few simple, yet powerful constructs that allow programmers to describe three-dimensional scenes and animations of such scenes. By virtue of its interpretive nature, Obliq-3D provides a fast-turnaround environment. The combination of simplicity and fast turnaround allows programmers to construct nontrivial animations quickly and easily.**

The paper is divided into three major parts. The first part introduces the basic concepts of Obliq-3D, using a series of graduated examples. The second part shows how the system can be used to implement Cone Trees. The third part develops a complete animation of Dijkstra's shortest-path algorithm.

***Index Terms*—3D graphics; 3D animation; information visualization; algorithm animation; interpreted language; embedded language; scripting language.**

I. Introduction

Designing an enlightening visualization of abstract information is a tricky proposition, requiring both artistic and communication skills. Most successful visualizations undergo dozens of design iterations. Animations of dynamic information are even harder.

Over the past few years, we and our colleagues have been exploring the use of 3D graphics to show the internal operations of computer programs [7]. Fig. 1 shows snapshots of some of our 3D animations. Our earlier experience in developing 2D algorithm animations convinced us that a high-level animation library, coupled with an interpreted language, is instrumental in developing high-quality animations [3], [4]. A high-level animation library allows users to focus on *what* they want to animate, without having to spend too much time on the *how*. An interpreted language significantly shortens the time needed for each iteration in the design cycle because time-consuming recompilations in such an environment are unnecessary.

This paper describes the 3D animation system we developed. It consists of an animation library and an embedded interpreted language [18], [19]. The animation library, called ANIM3D, is implemented in Modula–3 [9]. It uses the X Window System as the underlying window system, and MPEX (Digital's extension of PEX) as the underlying graphics library. Support for other window systems (e.g., Microsoft Windows NT) and graphics libraries (e.g., OpenGL) is nearly complete. ANIM3D is designed to interconnect easily with an embedded language. We chose Obliq [8], an object-oriented, lexically scoped, untyped, interpreted language. The Obliq interpreter is designed to be both easy to extend and easy to embed into Modula–3 programs. The resulting embedded language is called Obliq-3D. It provides all of the features of standard Obliq, plus extra modules that support the construction of 3D animations.

Before we developed Obliq-3D, we created animations by calling MPEX directly from Modula–3. By using Obliq-3D, the size of a typical 3D animation (Dijkstra's shortest-path algorithm, described in Section IV) decreased from about 2,000 lines of Modula–3 to 70 lines of Obliq, and the part of the design cycle time devoted to running a new visualization on a DECstation 5000/200 decreased from over seven minutes to about 10 seconds.

The remainder of this paper is structured as follows. Section II explains the main concepts of Obliq-3D, through a series of graduated examples. Sections III and IV contain two case studies. In Section III, we implement an interactive visualization of Cone Trees. We also build a general-purpose function to perform "shadow casting." This function takes an arbitrary graphical object and projects shadows of the object against walls. In Section IV, we develop an animation of Dijkstra's algorithm for finding the shortest path in a graph. This animation also serves as an introduction to our methodology for animating algorithms. Section V presents an overview of the implementation of the system. Section VI compares Obliq-3D with other general-purpose animation systems and with other algorithm animation systems, Section VII describes the current status of the system and future directions, and Section VIII offers some concluding remarks.

II. Obliq-3D by Graduated Examples

Obliq-3D is founded on three basic concepts: *graphical objects* for constructing scenes, time-variant *properties* for animating various aspects of a scene, and *callbacks* for providing interactive behavior.

Graphical objects include geometric shapes (spheres, cones, cylinders, and the like), light sources, cameras, *groups* for composing complex graphical objects out of simpler ones, and *roots* for displaying graphical objects on the screen. The graphical object (GO) class hierarchy is as follows:

The authors are with Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301 USA; e-mail: najork@src.dec.com, mhb@src.dec.com.

IEEECS Log Number V95008.

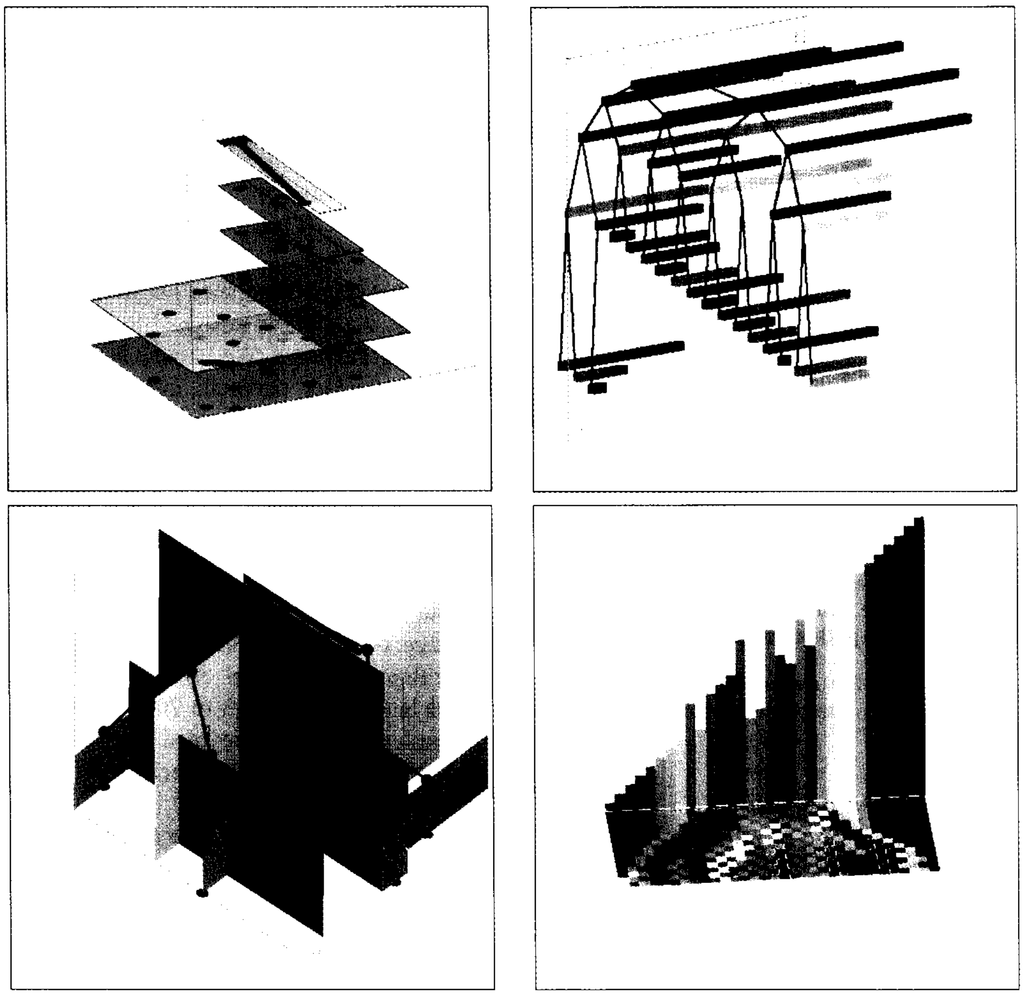
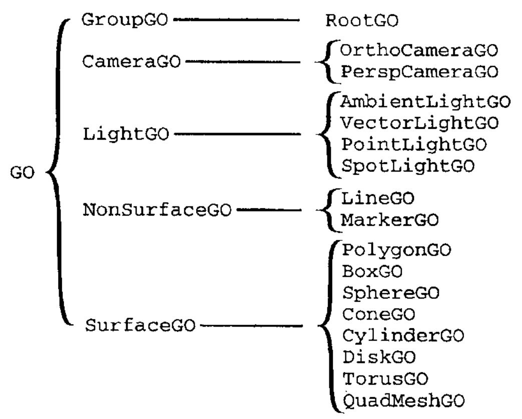


Fig. 1. These snapshots are examples of the type of visualizations for which Obliq-3D is very well-suited. Each visualization requires from 50 to 200 lines of code to produce. The top-left snapshot shows a divide-and-conquer algorithm for finding the closest pair of a set of points in the plane. The third dimension is used here to show the recursive structure of the algorithm. The top-right snapshot shows a view of Heapsort. Each element of the array is displayed as a stick whose length and color are proportional to the element's value. With clever placement, the tree structure of the heap is visible from the front and the array implementation of the tree is revealed from the side. The bottom-left snapshot shows a k-d tree, for $k = 2$. When viewed from the top, the walls reveal how the plane has been partitioned by the tree; when viewed from the front or side, we see the tree. The bottom-right snapshot shows a view of Shakersort. The vertical sticks show the current values of elements in the array, and the plane of "paint chips" underneath provides a history of the execution. The sticks stamp their color onto the chips plane, which is pulled forward as the execution progresses.



Properties describe attributes of graphical objects, such as their color, size, location, or orientation. A property consists of a *name* that determines what attribute is affected and a *value* that determines how it is affected. Property values are not simply scalar values, but rather functions that take a time and return a value. Thus, property values form the basis for animation.

Graphical objects are *reactive*, that is, they can respond to user events. Events are handled by *callbacks*. Associated with each graphical object is a callback stack for each event type (e.g., mouse clicks, keystrokes, mouse movement). The programmer can define or redefine the reactive behavior of a graphical object by pushing a new callback onto the appropriate stack. The previous behavior of the graphical object can easily be reestablished by popping the stack.

The remainder of this section presents more details about graphical objects and properties. We omit a discussion of callbacks, since our model is fairly common in modern interactive graphics toolkits.

A. Graphical Objects

The following program installs a window on the screen, and displays a white sphere:

```
let r = RootGO_NewStd();
r.add(SphereGO_New([0,0,0],0.5));
```
(*Program 1*)

The first line creates a root object, which is a special kind of graphical object, and assigns it to the variable r. Associated with each root object is a window on the screen, and creating the root object creates this window and installs it on the screen.

The expression SphereGO_New([0,0,0],0.5) creates a sphere object, which is another kind of graphical object. This particular sphere object represents a sphere whose center lies at the origin of the coordinate system, and whose radius is 0.5 unit.

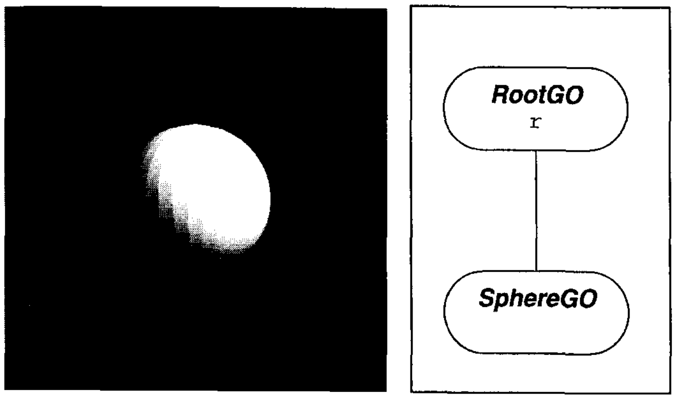


Fig. 2. The scene and the corresponding scene graph created by Program 1. For the sake of simplicity, we omit the camera and the light sources created by the call to RootGO_NewStd() from all scene graphs. We also omit the arrowheads from all arcs in the scene graphs; arcs always implicitly point downwards.

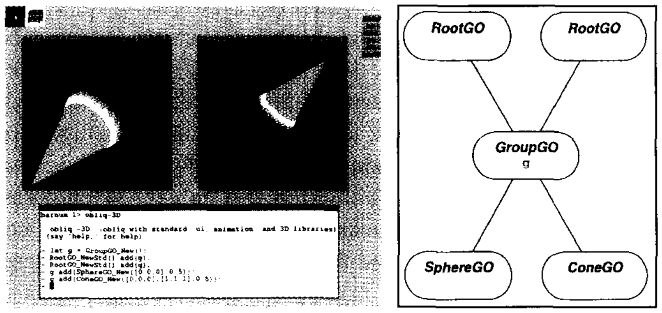
Finally, r.add(...) adds the sphere object to the root object and thereby causes it to be displayed in the window associated with the root object. See Fig. 2 (left). The user can manipulate the scene interactively, that is, he can move, rotate, and scale the objects displayed in a window as a group through mouse gestures. This interactive behavior is not innate to root objects, but rather added to the root object by the function RootGO_NewStd. This function also creates (in addition to the root itself) a camera through which the scene is viewed and several light sources.

The add method, which makes an object part of a larger object, is understood by all objects of type GroupGO. In particular, it is understood by objects of type RootGO, which is the type of root objects and a subtype of GroupGO. This ability to group objects together into larger objects allows us to build up hierarchies of graphical objects. These hierarchies must form a directed acyclic graph, called the *scene graph*. Fig. 2 (right) shows the scene graph corresponding to Program 1.

Because the hierarchy is a directed acyclic graph and not simply a tree, a node can have multiple parents. In particular, a scene graph can have several roots, making it possible to view the same scene in several windows, each with its own camera. The following program creates two windows that display a scene consisting of a sphere and a cone:

```
let g = GroupGO_New();
RootGO_NewStd().add(g);
RootGO_NewStd().add(g);
g.add(SphereGO_New([0,0,0],0.5));
g.add(ConeGO_New([0,0,0],[1,1,1],0.5));
```
(*Program 2*)

The first line creates a group object and assigns it to the variable g. The next two lines create two root objects and add g to each. Creating the two roots also creates two windows on the screen. Because g is a child of both root objects, every object added to g will also be a descendant of both root objects and will consequently appear in both windows. Finally, the last two lines create the sphere and the cone, and add each to group g. The base of the cone is centered at the origin and is of radius 0.5, and the tip lies at the point (1, 1, 1). Fig. 3 shows the two windows and the scene graph.



```
barnum 1> obliq-3D

 obliq -3D  (obliq with standard  ui, animation  and 3D libraries)
 (say 'help;' for help)

- let g = GroupGO_New();
- RootGO_NewStd().add(g);
- RootGO_NewStd().add(g);
- g.add(SphereGO_New([0,0,0],0.5));
- g.add(ConeGO_New([0,0,0],[1,1,1],0.5));
- 
```

Fig. 3. The two windows and the corresponding scene graph created by Program 2. The two windows display the same scene; however, the user has rotated and scaled the scenes differently. This interactive behavior is provided automatically by the root object returned by RootGO_NewStd.

Note that this program does not contain any explicit statements to draw the scene. Obliq-3D is based on a *damage-repair* model: Modifying the scene graph damages the scene, that is, the scene shown on the screen is no longer consistent with the scene graph. An animation server thread detects such damage and repairs it by redrawing the scene. (Obliq-3D also provides a primitive that declares a set of changes to be atomic.) This model is powerful, because it is conceptually simple and minimizes the amount of Obliq-3D code a user has to write.

B. Properties

The graphical objects in a scene graph describe what types of objects are contained in the scene. But there are other attributes to an object besides its shape: its color, location, size, etc. These aspects of an object are described by *properties*.

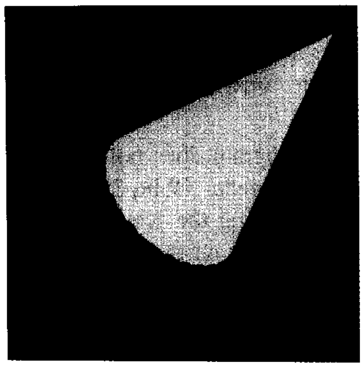
A property consists of two parts: a *name* and a *value*. The name describes what aspect of a graphical object is affected by the property, and the value controls the appearance of that aspect of the object. Many properties can be attached to a graphical object. These properties form a mapping—a partial function from names to values. Properties affect not only the graphical object they are attached to, but also all those objects that are descendants of the object and that do not "override" any attached properties.

For example, the following program displays a red sphere and a yellow cone:

```
let r = RootGO_NewStd();
let s = SphereGO_New([0,0,0],0.5);
r.add(s);
let c = ConeGO_New([0,0,0],[1,1,1],0.5);
r.add(c);
SurfaceGO_SetColor(r,"red");
SurfaceGO_SetColor(c,"yellow");                 (Program 3)
```

The first five lines create a root object that contains a sphere and a cone. Executing these lines causes a white sphere and a white cone to appear. The statement `SurfaceGO_SetColor(r,"red")` attaches the color property "red" to the root object. This causes the root and all objects contained in it, i.e., the sphere and the cone, to be redrawn in red. The statement `SurfaceGO_SetColor(c,"yellow")` attaches the color property "yellow" to the cone, thereby overriding the color that the cone had inherited from the root. The cone turns yellow, while the rest of the root (i.e., the sphere) remains red. Fig. 4 shows the final scene and the corresponding scene graph.

In this program, the name of the property that was attached is `SurfaceGO_Color`, and the values are the colors "red" and "yellow." The procedure `SurfaceGO_SetColor` takes as arguments a graphical object `g` and an expression describing a color and attaches a property with the name `SurfaceGO_Color` and the appropriate value to the graphical object `g`.



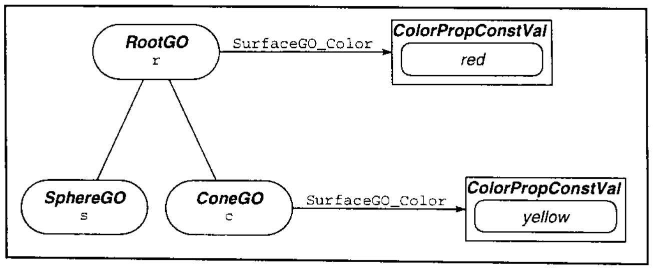


Fig. 4. The final scene and the corresponding scene graph created by Program 3.

Property inheritance works as follows: Let `r` be a root object and let `o` be a leaf object that is a descendant of `r`. The animation server displays an instance of `o` in the window associated with `r` for each path from `r` to `o`, using the properties attached to the graphical objects along that path. Thus, if there are several paths to a particular object, the object will be displayed several times using different sets of properties. The "shadow casting" function `ShadowWrapper` of Section III.C makes use of this feature.

To a first approximation, the sequence in which graphical objects are added to groups, and properties are attached to graphical objects, is immaterial. The appearance of the final scene does not depend on the sequence; however, the appearance of intermediate versions does, should the changes be so complex that the batch of changes is not completed before the scene is repainted. For example, the following program creates exactly the same final scene as Program 3.

```
let r = RootGO_NewStd();
let s = SphereGO_New([0,0,0],0.5);
let c = ConeGO_New([0,0,0],[1,1,1],0.5);
SurfaceGO_SetColor(c,"yellow");
SurfaceGO_SetColor(r,"red");
r.add(c);
r.add(s);                                  (Program 4)
```

As mentioned, to prevent the display of intermediate versions, Obliq-3D provides a primitive that declares a set of changes to be atomic.

Color property values are one type of property value. Other types include Boolean property values, real property values, point property values, and so on.

Properties control not only surface attributes of an object, such as its color and transparency, but also spatial attributes, such as its location and size. For example, the center of a sphere is controlled by a point property named `SphereGO_Center`, and the radius is controlled by a real property named `SphereGO_Radius`. The expression `SphereGO_New(c,r)` creates a sphere object and attaches two properties to it: a property with name `SphereGO_Center` and value `c`, and a property with name `SphereGO_Radius` and value `r`.

Although any property can be attached to any object, an object's appearance does not necessarily depend on all of its properties. For example, it is perfectly legal to attach a `SphereGO_Center` property to a cone; this will not affect the appearance of the cone.

C. Property Values and Behaviors

Property values are time-variant. For example, we could create a real property value that changes from 4.3 to 1.7 over 10 seconds, a point property value whose x component depends on the current value of some real property value, a color property value that alternates between blue and red every two seconds, and of course, a property value that doesn't change over time. Property values are implemented as Obliq objects; they have a method called `value` that takes a time and returns the value at that time.

In Programs 3 and 4 we used `SurfaceGO_SetColor` to attach a color property value. We passed the textual name of a color (i.e., `"red"`) as an argument, and `SurfaceGO_SetColor` used this to create a color property value. These property values were constant; the color didn't change over time.

We can also create property values directly. For example, the expression `ColorProp_NewConst("green")` creates a new constant color property value, whose value is green. One reason why we would like to create property values directly is to be able to attach the same property value to several graphical objects. The following program displays two spheres. The radius of both spheres is determined by the shared property value `rad`:

```
let r = RootGO_NewStd();
let rad = RealProp_NewConst(0.5);
r.add(SphereGO_New([-1,0,0],rad));
r.add(SphereGO_New([1,0,0],rad));        (Program 5)
```

Fig. 5 shows the scene displayed by the program, together with the corresponding scene graph. Changing `rad` affects both spheres.

Note that the second argument to the procedure `SphereGO_New` is overloaded: It can take either a real number (as in Program 1) or a real property value (as above). (The first argument is overloaded as well; as we shall see, Program 7 takes advantage of this feature.) This form of overloading is common in Obliq-3D and leads to shorter and easier-to-read programs.

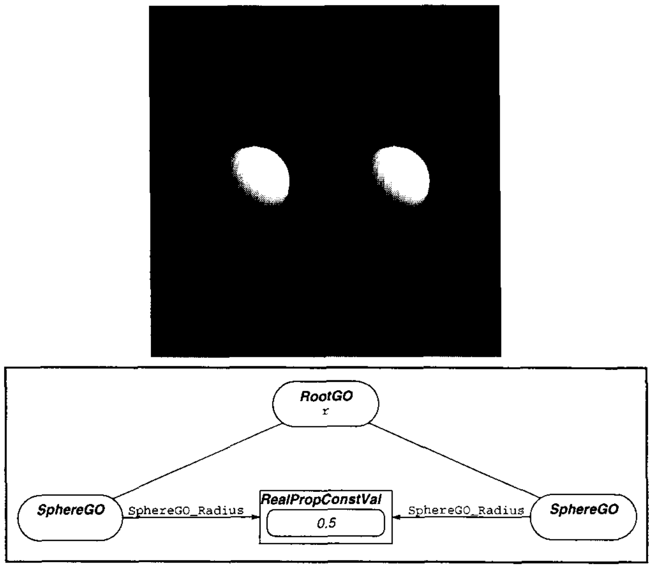


Fig. 5. The scene and the scene graph created by Program 5.

The ability to attach the same property value to several graphical objects provides us with an alternative to "property inheritance." In the example above, instead of attaching `rad` to each sphere, we could have attached it to their common ancestor `r` instead. If no other radius property were attached to either sphere, then the radius of both spheres would have been inherited from `r`. However, there are many cases in which the natural structure of the scene graph prevents us from using property inheritance. For example, in a scene describing a car, we might want to put two tires and an axle into the same group, but we might also want the four tires to have the same color and the two axles to have a different color. Having the tires and the axle in the same group prevents us from using property inheritance to describe their color; in such a case, shared property values are a convenient solution.

Conceptually there are four kinds of property values: constant, asynchronous, synchronous, and dependent. A constant property value does not change over time. An asynchronous property value changes perpetually, independent of other property values. A synchronous property value changes from one value to another when signaled. And finally, a dependent property value changes whenever some other property value has changed. If we were to realize these four kinds as subtypes of a property value, `PropVal`, these subtypes would have different method suites. For instance, a constant property value would have a "set" method to change its value, whereas a synchronous property would have an "interpolate" method.

We cannot realize the four kinds of property values as subtypes because we would like to allow a user to dynamically change the kind of property value. However, in Obliq (like most programming languages), it is not possible to change the method suite (i.e., the type) of an object on the fly.

To achieve this effect, we introduce a level of indirection. We introduce a new object type, `PropBeh`, that has four subtypes corresponding to the four ways that property values can behave. Each property value has a private data field that contains its behavior. If `v` is a property value, the method call `v.getBeh()` returns the behavior of `v`. Similarly, if `b` is a behavior, the method call `v.setBeh(b)` sets the behavior of `v` to `b`.

Thus, property values serve merely as "clerks" for behaviors: They forward value-inquiring messages from the client program or the animation server to the behavior. Replacing one behavior with another one is transparent to the client program and to the animation server, because the property value acts as an intermediary.

We now consider each type of behavior in more detail.

C.1. Constant Behaviors

Constant behaviors are behaviors that do not change over time. For example, the statement

```
let rad = RealProp_NewConst(r);
```

that we encountered in Program 5 creates a real property value whose behavior is constant. All constant behaviors understand the `set` message. For example, adding the statement

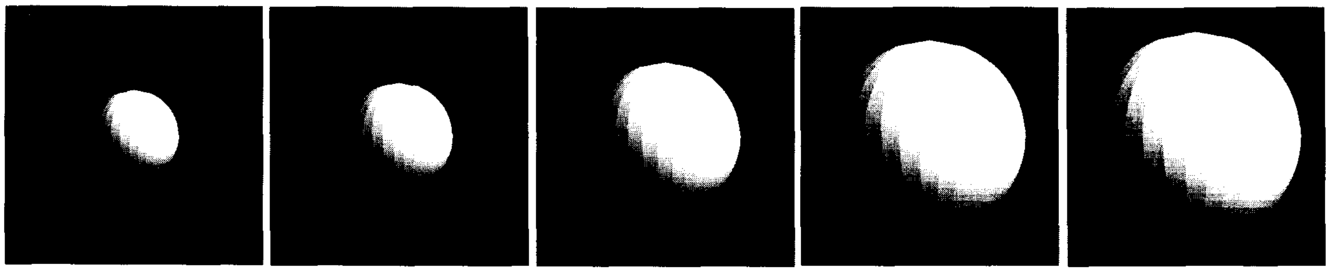


Fig. 6. Snapshots of the animation created by Program 6.

```
rad.getBeh().set(0.3);
```

to the program changes the radius of both spheres to 0.3, since the radius property of both spheres was `rad`.

C.2. Asynchronous Behaviors

Asynchronous behaviors are behaviors that change over time, and they change throughout their entire lifetimes. They do not depend on other property values, nor do they synchronize their changes with them. For example, the following program displays a sphere whose radius oscillates between 0.3 and 0.7 unit:

```
let root = RootGO_NewStd();
let rad = RealProp_NewAsync(meth(self,time)
            (0.2 * math_sin(time)) + 0.5 end);
root.add(SphereGO_New([0,0,0],rad));
```

(*Program 6*)

The second line creates a new real property value `rad` with an asynchronous behavior. This behavior is defined by a method that takes two arguments, the behavior itself and the current time, and returns a real number. In this case, the value of `rad` at time t is defined to be $0.2 \cdot \mathit{sin}(t) + 0.5$, a value between 0.3 and 0.7. This method will be periodically called by the animation server whenever the behavior is reachable from one of the root objects.

The object `rad` is attached as the radius property to a sphere object, which causes the sphere to pulse. Fig. 6 shows a "filmstrip" of successive snapshots of the scene generated by this program.

C.3. Synchronous Behaviors

A *synchronous behavior* represents a value that does not change continuously, but rather upon being signaled and for a specified finite period of time. Each synchronous behavior is connected to a synchronization object, called an *animation handle*. We say that the animation handle *controls* the behavior. Many synchronous behaviors can be controlled by the same handle.

Conceptually, a synchronous behavior consists of a *current value* and a priority queue of requests, called the *request queue*. Requests are objects that change the current value of the behavior. For each type of behavior, there is a set of predefined requests that perform linear interpolations between the current value of the behavior and a new one; in addition (as we shall see in Section II.D), the client program can define arbitrary new requests.

Associated with each request is a *start time* and a *duration*. A request with start time s and duration d begins to affect the value of the behavior s seconds after the controlling animation handle is signaled and ceases to do so d seconds later.

The following program demonstrates the use of synchronous behaviors. It creates a red sphere at the origin and then gradually changes its color to green. This change takes four seconds. One second after the color starts to change, the sphere smoothly moves from the origin to the point (0,1,0). The movement takes three seconds.

```
let ah = AnimHandle_New();
let p = PointProp_NewSync(ah,[0,0,0]);
let c = ColorProp_NewSync(ah,"red");
let s = SphereGO_New(p,0.5);
SurfaceGO_SetColor(s,c);
let r = RootGO_NewStd();
r.add(s);
p.getBeh().linMoveTo([0,1,0],1,3);
c.getBeh().rgbLinChangeTo("green",0,4);
ah.animate();
```

(*Program 7*)

The first line creates a new animation handle `ah`. The next two lines create a point property value `p` whose initial value is (0, 0, 0) and a color property value `c` whose initial value is "red." Both `p` and `c` are synchronous property values (that is, property values containing a synchronous behavior), and both are controlled by the animation handle `ah`. The next two lines create a sphere object whose center depends on the current value of `p` and whose color depends on the current value of `c`. So, initially the sphere is red and centered at the origin. The following two lines create a root object, to which the sphere object is added. In the next line, we send the message `linMoveTo` to the behavior of `p`, asking it to change from its present value to (0, 1, 0). This change will start one second after the animation handle is signaled and will take three seconds to complete. Similarly, we send a message `rgbLinChangeTo` to the behavior of `c`, asking it to change from its present value (that is, red) to green. The change will start immediately when the animation handle is signaled and will take four seconds to complete. Fig. 7 shows the state of the scene graph after this message.

Up to this point, no changes have actually taken place. Sending the message `animate` to the animation handle `ah` causes the requests to be processed. The `ah.animate()` call returns only after all requests have been processed, that is, it will run for four seconds.

Many requests can be added to the request queue of a synchronous behavior. However, the time intervals of these requests must not overlap. The requests are processed in increasing order of their start times.

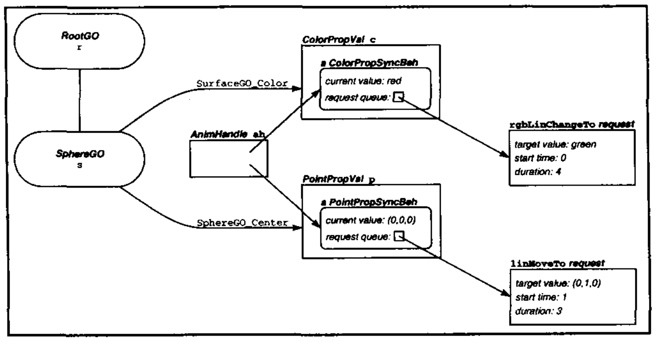


Fig. 7. The scene graph created by Program 7 before the call to `ah.animate`.

C.4. Dependent Behaviors

A *dependent behavior* is a behavior whose value depends on other property values. Dependent behaviors allow us to specify functional dependencies between property values. Such dependencies are often called *one-way constraints*. A

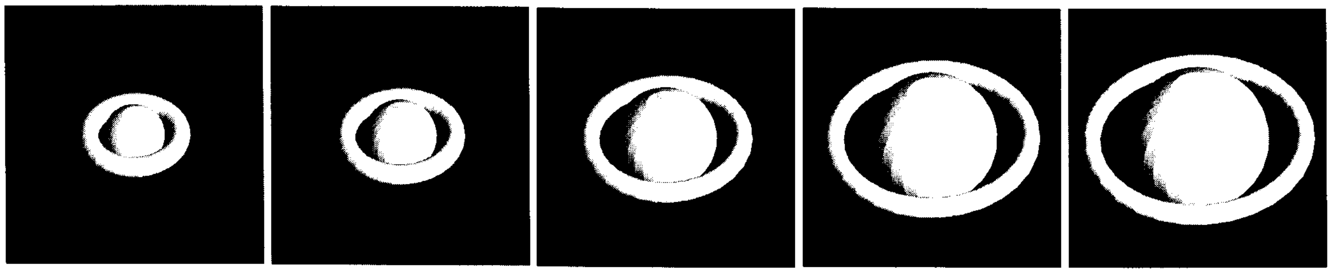


Fig. 8. Snapshots of the animation created by Program 8.

checked run-time error occurs if the dependency graph induced by the dependent behaviors includes any cycles.

A dependent behavior is defined just like an asynchronous behavior, through a method that is passed to its creation function. This method takes the behavior itself and the current time as arguments and returns the value for the behavior at that time. The fundamental difference is that the body of the method may access other property values.

The following program creates a sphere and a torus. As in Program 6, the radius of the sphere is defined by an asynchronous property value, which causes the sphere to pulsate. The radius of the torus is dependent on the radius of the sphere; it will always be 1.5 times as large.

```
let root = RootGO_NewStd();
let rad1 = RealProp_NewAsync(
             meth(self,time)
              (0.2 * math_sin(time)) + 0.5
             end);
root.add (SphereGO_New ([0,0,0],rad1));
let rad2 = RealProp_NewDep (
             meth(self,time)
              1.5 * rad1.value(time)
             end);
root.add (TorusGO_New([0,0,0], [0,1,0],
                       rad2, 0.1));
```

(*Program 8*)

The first three statements of this program are identical to Program 6. The fourth statement defines a new dependent property value `rad2`, whose value at a given time is 1.5 times the value of `rad1` at that time. The last statement creates a torus, whose center is (0, 0, 0) and whose normal vector is (0, 1, 0). The major radius of the torus is defined to be `rad1`, and the minor radius has a constant value of 0.1. Fig. 8 shows a "filmstrip" of successive scenes created by this program.

D. Requests

Section II.C.3 showed how synchronous behaviors can be used to change a property value from its current value to a new one. In order to make the transition appear smooth, continuous property values (e.g., reals, points, colors, and transformations) must be interpolated between the initial and final values. Each synchronous behavior provides one or more default interpolation methods. An interpolation method creates a *request* object for performing the interpolation and adds the request to the behavior's request queue.

Obliq-3D allows the programmer to specify arbitrary interpolations by creating customized request objects. Such a custom request object `r` can then be added to the request queue of a synchronous behavior `b` by calling `b.addRequest(r)`. As stated before, the time intervals of requests in the request queue must not overlap.

For instance, a point request object `r` is created by calling the function `PointProp_NewRequest`(*s, d, m*). This function takes the desired start time *s* of the interpolation (relative to the time at which the controlling animation handle is signaled), its desired duration *d*, and a method *m*, which will be used to interpolate between the initial point value and the desired new one. The interpolation method *m* takes three arguments: the request itself, a point that is the value to start interpolating from, and the time that has elapsed since the controlling animation handle was signaled. It returns the value the point should have at this time. Later on, the animation server will call this method repeatedly to perform the actual interpolation.

Let's assume that we want to move a sphere along a straight path. However, we don't want to move at constant speed as we did in Program 7 by calling `linMoveTo`. Rather, we'd like the movement to be in "slow-in/slow-out" fashion [10], where the sphere gradually picks up speed and gradually slows down again. The function `SlowInSlowOut` implements this type of animation. It takes an endpoint, a start time, and a duration and returns a point request object:

```
let SlowInSlowOut =
 proc (endpoint,s,d)
   let m =
     meth (self,startpoint,reltime)
       if d is 0.0 then
         endpoint;
       else
         let f = (reltime - s) / d;
         let r = 0.5 + 0.5 *
               math_sin ((f - 0.5) * math_pi);
         Point3_Plus(startpoint,
               Point3_TimesScalar(
                  Point3_Minus(endpoint,startpoint),
                  r));
       end
     end;
   PointProp_NewRequest(s,d,m)
 end;
```

`SlowInSlowOut` first defines the interpolation method, `m`. This method takes three arguments: `self`, `startpoint`, the initial value of the point, and `reltime`, the time that has elapsed since the animation handle was signaled.

If the request has a duration of 0, calling `m` immediately returns the endpoint. Otherwise, it computes a value `f`, which indicates what fraction of the duration of the animation has already passed. It then computes a real value `r`, which depends on `f`, that indicates what fraction of the path the point has

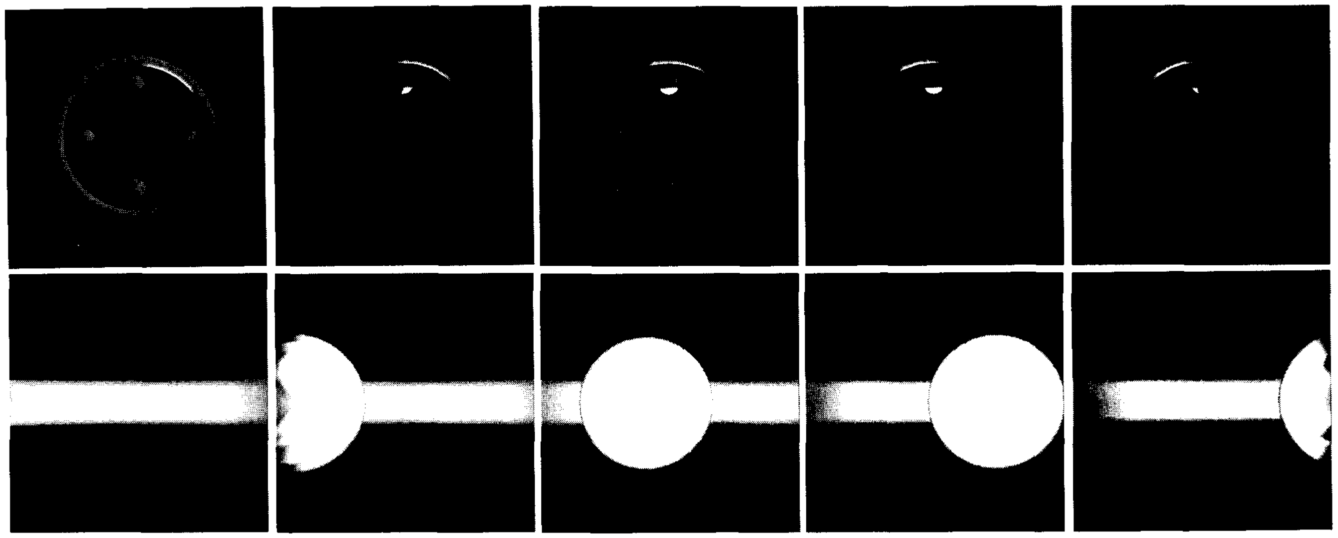


Fig. 9. Snapshots of the animation created by Program 10.

traversed. If we were to implement constant-speed motion, r would be equal to f. To achieve a slow-in/slow-out effect, we instead choose $r = \frac{1}{2} + \frac{1}{2} sin(\pi(f - \frac{1}{2}))$. Finally, m returns a point that lies between startpoint and endpoint, depending on the value of r.

Now that m is defined, SlowInSlowOut creates a new request object with start time s, duration d, and value method m and returns it. The values endpoint, s, and d are captured in the closure of the method m. This is due to the fact that Obliq is a higher-order language, which treats methods (just like procedures) as first-class values.

The following program uses the slow-in/slow-out effect. It creates two spheres and then moves them up. The left sphere moves at a constant speed, whereas the right sphere uses slow-in/slow-out:

```
let root = RootGO_NewStd();
let ah = AnimHandle_New();

let left = PointProp_NewSync(ah,[-1,-1,0]);
root.add(SphereGO_New(left,0.3));
left.getBeh().linMoveTo([-1,1,0],0.0,3.0);

let right = PointProp_NewSync(ah,[1,-1,0]);
root.add(SphereGO_New(right,0.3));
right.getBeh().addRequest(
            SlowInSlowOut([1,1,0],0.0,3.0));

ah.animate();
```
(Program 9)

E. Light and Camera Objects

In Section II.A, we mentioned that the function RootGO_NewStd() creates not only a root object, but also two light sources and a camera. Obliq-3D treats cameras and light sources as ordinary graphical objects. They can be added to groups, properties can be attached to them, and their appearance is controlled by properties. Cameras and lights "inherit" the current values of properties that are attached along the path between them and the root, just like every other graphical object does.

In particular, both cameras and light sources, like all graphical objects, are affected by the GO_Transform property. This property implements time-variant transformation matrices. We exploit this fact in the following program, which positions a camera inside of a rotating beacon. The beacon is a group containing a camera, a light source, and a cone. Changing the transformation property attached to this group rotates all three objects contained in it.

```
let cam = PerspCameraGO_New([0,0,0],[0.5,0,0],
                            [0,0,1],0.75);
let beacon = GroupGO_New();
beacon.add(ConeGO_New([0.5,0,0],[0,0,0],0.2));
beacon.add(SpotLightGO_New("white",
                            [0,0,0],[0.5,0,0],
                            1,0.38,0.5,0.5));
beacon.add(cam);

let rot = TransformProp_NewAsync(
            meth (self,time)
              Matrix4_RotateZ(Matrix4_Id,time)
            end);
GO_SetTransform(beacon, rot);

let g = GroupGO_New();
SurfaceGO_SetShading(g,"Gouraud");
g.add(SphereGO_New([0,1,0],0.2));
g.add(SphereGO_New([1,0,0],0.2));
g.add(SphereGO_New([0,-1,0],0.2));
g.add(SphereGO_New([-1,0,0],0.2));
g.add(TorusGO_New([0,0,0],[0,0,1],1.5,0.1));
g.add(beacon);

let root1 = RootGO_NewStd();
root1.add(g);

let root2 = RootGO_NewStd();
root2.add(g);
root2.changeCamera(cam);
```
(Program 10)

The first five lines create a group object beacon and add a cone, a spot light source, and a perspective camera to it. The camera is located at the tip of the cone, and its viewing direction is parallel to the central axis of the cone. Similarly, the spot light source is located at the tip of the cone, and the cone of light emitted by it roughly overlaps with the cone object.

The next four lines create an asynchronous transformation property value, which continuously rotates around the *z* axis, and attach it to beacon. This causes beacon and all objects contained in it (i.e., the cone, the spot light, and the camera) to rotate around the *z* axis.

Next, we create a group object g and attach a property to it that causes all objects in g to be Gouraud-shaded. We add four spheres, which are arranged in diamond formation, a torus, and the beacon object to g.

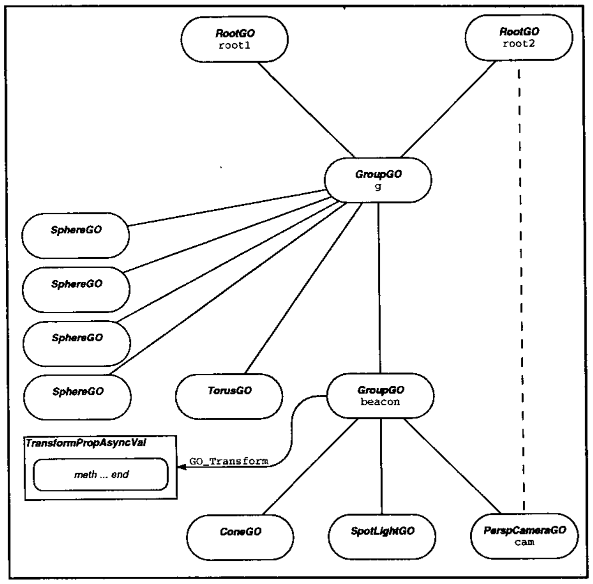


Fig. 10. The scene graph created by Program 10.

Finally, we create two standard root objects, root1 and root2, and add g to each of them. We also change the camera used by root2 to be not the default camera, but rather cam, the camera that is part of the rotating beacon.

Fig. 9 shows a series of snapshots of the animation in progress. The upper row of pictures shows root1, the scene as observed by the stationary camera; the lower row shows root2, the scene as observed by the rotating camera. The corresponding scene graph is shown in Fig. 10.

III. Case Study: Interactive Cone Trees With Shadows

This section shows how one can use Obliq-3D for creating an interactive display of Cone Trees. A *Cone Tree* is a three-dimensional representation of a tree structure [23], where the root of a tree (represented by a cube, a sphere, or some other appropriate object) is located at the tip of a transparent cone, and the children of the root node are arranged around the base of the cone. Each child can be the root node of a subtree, which is represented in a recursive fashion by a cone whose tip is located at the object representing the child.

In this section, we first give the code to create a static Cone Tree display. We then add interactivity: The user can select a node in the tree, and that node and its ancestors rotate to the front of the display. Then, we show how to improve the quality of the picture by casting shadows of the tree onto adjacent walls. The code for shadow-casting is not specific to Cone Trees; it can be used for shadow-casting of any object. Finally, we further improve the image by adding more perspective cues, such as grid lines on the surrounding walls and heightened viewing perspective.

A. Static Cone Trees

The function ConeTree takes the Obliq representation of an arbitrary tree and returns a graphical object that represents the Cone Tree visualization of the tree. Each node of the tree contains a numeric value and an arbitrary number of children. A tree *t* with root value *v* and subtrees $t_1, \ldots, t_n$ is represented by an Obliq array $[v, a_1, \ldots, a_n]$, where each a_i is the representation of t_i. For simplicity, we do not deal with empty trees.

```
let rec ConeTree =
 proc (tree)
   let H = 0.6; (* height of the cone *)
   let R = 0.3; (* radius of the cone *)
   let r = GroupGO_New();
   let ball = SphereGO_New([0,H,0],0.1);
   let hue = real_float(tree[0]) * 0.01;
   SurfaceGO_SetColor(ball,color_hsv(
                                 hue,1.0,1.0));
   r.add(ball);
   if #(tree) > 1 then
     let cone = ConeGO_New([0,0,0],[0,H,0],R);
     r.add(cone);
     SurfaceGO_SetTransmissionCoeff(cone,0.8);
     SurfaceGO_SetColor(cone,"gray");
   end;
   for i = 1 to #(tree)-1 do
     let angle = 2.0 * math_pi * real_float(i-1) /
                   real_float(#(tree) - 1);
     let ctr = [math_sin(angle)*R, 0,
                math_cos(angle)*R];
     let edge = LineGO_New(ctr,[0,H,0]);
     r.add(edge);
     LineGO_SetColor(edge,"gray");
     let child = ConeTree(tree[i]);
     r.add(child);
     let M = Matrix4_Id;
     let M = Matrix4_Scale(M,0.5,0.5,0.5);
     let M = Matrix4_Translate(M,0,-H/2.0,R);
     let M = Matrix4_RotateY(M,angle);
     GO_SetTransform(child,M);
   end;
  r;
 end;
```

The function creates a group object r and a sphere object ball and adds ball to r. The sphere represents the root node of the tree, and it is colored to indicate the value of the node. If the root node has any subtrees (that is, if the array tree has more than one element), we add a transparent gray cone to r, whose tip coincides with the center of the sphere. Finally, we iterate over the subtrees of the root node. For each subtree we add a gray line to r. The line leads from the tip of the cone to a point on its base. The points on the base are evenly spaced. We then call ConeTree recursively on the subtree, and add the returned Cone Tree representation to r. We also attach a transformation property to the subtree's Cone Tree, which scales it down by a factor of 0.5 and translates it such that its tip coincides with the endpoint of the gray line.

The following program illustrates the use of ConeTree, and Fig. 11 (upper-left) shows the scene created by the program

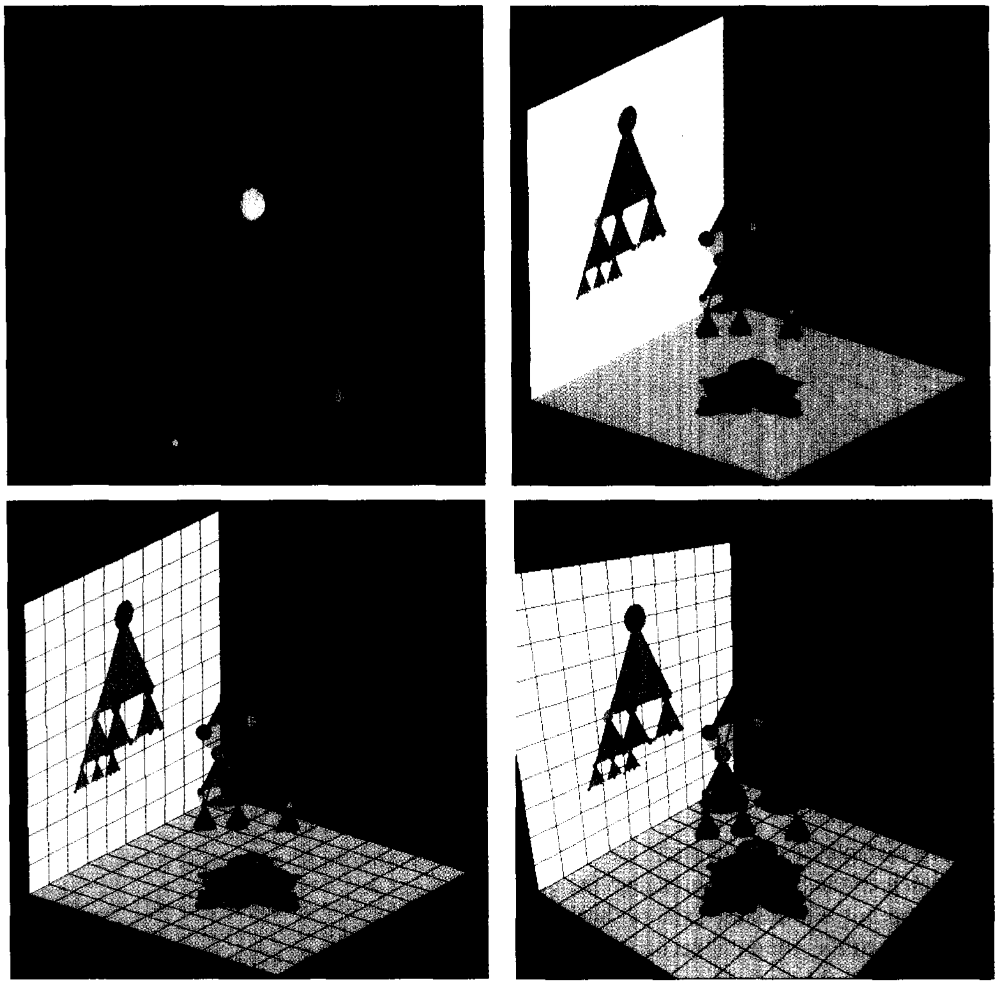


Fig. 11. These snapshots show the successive refinements of the Cone Tree visualization. The upper-left image shows the Cone Tree by itself. In the upper-right, shadows have been added. In the lower-left, a grid has been superimposed on the walls upon which the shadows have been cast. And finally, in the lower-right, the camera has been modified to increase the perspective impression.

```
let t = [56,[43, [32, [62], [18], [7]]
                 [85, [40], [22]],
                 [59],
                 [27]],
            [63, [13, [95], [48], [1], [99]],
                 [81],
                 [37]],
            [91],
            [17, [67], [24], [83]]];
    let root = RootGO_NewStd();
    root.add(ConeTree(t));                 (Program 11)
```

B. Interactive Animation

The cone trees generated by the `ConeTree` function are static. We'll now add animation, similar to that performed by the Xerox PARC "Information Visualizer" project, to focus on a particular node in the tree:

> When a node is selected with the mouse, the Cone Tree rotates so that the selected node and each node in the path from the selected node up to the top are brought to the front and highlighted. The rotations of each substructure are done in parallel, following the shortest rotational path, and are animated so the user sees the transformation at a rate the perceptual system can track. Typically, the entire transformation is done in about one second. [23]

To keep things simple, we omit the code for selecting a tree node with the mouse. Instead, the user triggers an animation by calling `Focus`(*t*, *v*), where *t* is the cone tree returned by the `ConeTree` function, and *v* is the value of the node that he wants to focus on. We will assume that *v* is unique. This call triggers the following animation: First, the tree node corresponding to *v* and all its ancestors in the cone tree start to blink, at a rate of five pulses per second. One second later, the selected node and all its ancestors are brought to the front. The rotations of all substructures are done in parallel, following the shortest rotational path. The rotations take two seconds. Finally, one second after the rotations have completed, the selected node and its ancestors stop blinking.

We first create an animation handle `ah`. This will be used to synchronize the rotations of the subtrees and the blinking of the nodes:

```
let ah = AnimHandle_New();
```

Next, we replace the `ConeTree` procedure with the following version:

```
let rec ConeTree =
 proc (tree)
  let H = 0.6; (* height of the cone *)
  let R = 0.3; (* radius of the cone *)
  let r = GroupGO_New().extend(
           {val => tree[0],
            kids => [], angle => 0.0,
            lumin => RealProp_NewSync(ah,0.5)});
  GO_SetTransform (r,
        TransformProp_NewSync (ah, Matrix4_Id));
  let ball = SphereGO_New([0,H,0],0.1);
  let hue = real_float(tree[0]) * 0.01;
  SurfaceGO_SetColor(ball,color_hsv(hue,1.0,1.0));
  r.add(ball);
  if #(tree) > 1 then
    let cone = ConeGO_New([0,0,0],[0,H,0],R);
    r.add(cone);
    SurfaceGO_SetTransmissionCoeff(cone,0.8);
    SurfaceGO_SetColor(cone,"gray");
  end;
  for i = 1 to #(tree)-1 do
    let angle = 2.0 * math_pi * real_float(i-1) /
                  real_float(#(tree) - 1);
    let ctr = [math_sin(angle)* R, 0,
           math_cos(angle)*R];
    let edge = LineGO_New(ctr,[0,H,0]);
    r.add(edge);
    LineGO_SetColor(edge,"gray");
    let sub = GroupGO_New();
    r.add(sub);
    let M = Matrix4_Id;
    let M = Matrix4_Scale(M,0.5,0.5,0.5);
    let M = Matrix4_Translate(M,0,-H/2.0,R);
    let M = Matrix4_RotateY(M,angle);
    GO_SetTransform(sub,M);
    let child = ConeTree(tree[i]);
    sub.add(child);
    r.kids := r.kids @ [child];
  end;
  r;
 end;
```

The group object `r` is extended with four fields: `val` stores the numeric value of the node; `kids` is an array containing the group objects representing the children; `angle` stores the cumulative rotation of the node; and `lumin` is a synchronous real property value that controls the brightness of the node and is used to make the node blink. Next we attach a synchronous transformation property to `r` that controls its animated rotation. Both the real and the transformation property values are controlled by the animation handle `ah`. As before, we create a sphere `ball` that represents the node in the tree. We attach `r.lumin` to the ball as the ambient reflection coefficient (or, loosely speaking, the luminosity of the sphere). As before, we color the sphere according to its value and create a transparent gray cone if the node has any children.

Then, we iterate over all the children of the tree node. As before, for each such child (or subtree), we draw a gray line from the tip of the cone to a point on its base and we call `ConeTree` recursively to visualize each subtree. In the original version, we attached a constant transformation property to the child node that was used to scale it to the right size and move it to the proper position. This is not possible in the modified version, since the recursive call to `ConeTree` also attaches a synchronous transformation property to the child. So, we create a new group object `sub`, place it between `r` and the child, and attach the constant transformation property to it instead. Finally, we add the child object to `r.kids`, the array of children.

The method `BlinkMeth` is used to create a request object, which will be used by the synchronous property that controls the brightness of a node:

```
let BlinkMeth =
 meth(self,startval,reltime)
  if reltime is self.dur() then
    0.5
  else
    if (real_round(reltime * 5.0) % 2) is 0
                 then 0.3 else 0.7 end
  end
 end;
```

`BlinkMeth` will be invoked by the animation server when the request object is activated. It returns a value that alternates between 0.3 and 0.7, changing five times a second. After the request is completed, the value is reset to its original value of 0.5.

The procedure `FocusRec` is used to bring a node to the front in an animated fashion. It recursively walks the tree structure looking for the desired node. Once it finds the desired node, it bottoms out of the recursion, scheduling the animations that blink the current node and rotate it to the front. It returns true if the requested node was found in the tree, and false otherwise. This implementation uses depth-first search to find the correct node.

```
let rec FocusRec =
 proc(node,val)
  if node.val is val then
    node.lumin.getBeh().addRequest(
         RealProp_NewRequest(0,4,BlinkMeth));
    true
  else
    var i = 0;
    var found = false;
    loop
      if (i >= #(node.kids)) or
                        found then exit end;
      found := FocusRec(node.kids[i],val);
      if found then
        let angle = 2.0 * math_pi * real_float(i) /
                  real_float(#(node.kids));
        var delta = angle - node.angle;
        node.angle := angle;
        if delta > math_pi then
             delta := delta - 2.0 * math_pi end;
        if delta < -math_pi then
             delta := delta + 2.0 * math_pi end;
        node.getProp(GO_Transform).getBeh().
             rotateY(-delta,1,2);
        node.lumin.getBeh().addRequest
            (RealProp_NewRequest(0,4,BlinkMeth));
      end;
      i := i + 1;
    end;
    found
  end;
 end;
```

The last procedure, `Focus`, is the client's interface. Calling `Focus(`*t*, *v*`)` calls `FocusRec` to schedule the appropriate animations and then signals the animation handle `ah` to actually perform them:

```
let Focus =
 proc(tree,val)
  FocusRec(tree,val);
  ah.animate();
 end;
```

C. Casting Shadows

Shadows provide powerful cues for disambiguating the shapes and locations of three-dimensional objects [32]. There are several techniques for implementing shadows. A particularly simple technique for generating the shadow cast by an object is to "flatten" the object (or a copy of it) so that it appears to be planar and superimpose the flattened object onto a polygon that acts as the surface onto which the shadow falls [1], [30], [16]. Objects can be flattened by applying a non-uniform scaling operation to them.

The function ShadowWrapper takes an arbitrary graphical object and returns a group that contains the original object together with three walls and shadows of the original object cast against the walls:

```
let ShadowWrapper =
 proc(go)
   let top = GroupGO_New();
   top.add(go);
   top.add(PolygonGO_New([[-1,-1,-1],[-1,-1, 1],
                          [-1, 1, 1],[-1, 1,-1]]));
   top.add(PolygonGO_New([[-1,-1,-1],[-1,-1, 1],
                          [1,-1, 1],[1,-1,-1]]));
   top.add(PolygonGO_New([[-1,-1,-1],[-1, 1,-1],
                          [1, 1,-1],[1,-1,-1]]));
   let CastShadow = proc (a,b,c,x,y,z)
     let g = GroupGO_New();
     top.add(g);
     g.add(go);
     GO_SetTransform(g,
        Matrix4_Translate(
          Matrix4_Scale(Matrix4_Id,a,b,c),
          x,y,z));
   end;
   CastShadow(0.0001, 1, 1, -0.995, 0, 0);
   CastShadow(1, 0.0001, 1, 0, -0.995, 0);
   CastShadow(1, 1, 0.0001, 0, 0, -0.995);
   top;
 end;
```

The function ShadowWrapper takes a graphical object go as its argument. The object (and its descendants) must lie inside the cube whose corner points are (1, 1, 1) and (−1, −1, −1) in go's local coordinate system. When invoked, ShadowWrapper creates a new group object top and adds go to it. It also creates three square-shaped polygons, which form three adjacent sides of a cube, and adds them to the group object. These polygons provide the surfaces onto which the shadow of the object go will be projected. Next, it adds the three "shadows" of go to the group object top. This is achieved by calling the nested function CastShadow. A call to this function creates a new group object g, whose parent is top and whose only child is go. Attached to g is a transformation property which scales all objects in the group and then translates the scaled objects. Choosing a scaling factor close to 0 in one dimension, say x, causes the objects in go to have almost no extent in the x dimension. Translating the flattened objects in the x dimension such that they (almost) lie on the square-shaped polygon orthogonal to the x axis creates the impression of a shadow of go, cast onto this polygon.

It is possible to construct a general-purpose shadow-casting function with so little code because the scene hierarchy can be a directed acyclic graph, as opposed to a simple tree. We can insert the object *g* that is to be shadow-enhanced into several places in the scene graph without having to copy it. Moreover, if *g* or one of its descendants changes in any way, the shadows automatically remain consistent.

The following program demonstrates the use of the MakeShadow function. It creates a group object that contains a red cube and a red torus. It then creates a root object and adds the shadow-enhanced group to it.

```
let g = GroupGO_New();
g.add (BoxGO_New([-0.1,0.3,-0.1],[0.1,0.5,0.1]));
g.add (TorusGO_New([0,0,0],[0,1,0],0.5,0.1));
SurfaceGO_SetColor(g,"red");
RootGO_NewStd().add(ShadowWrapper(g));
```

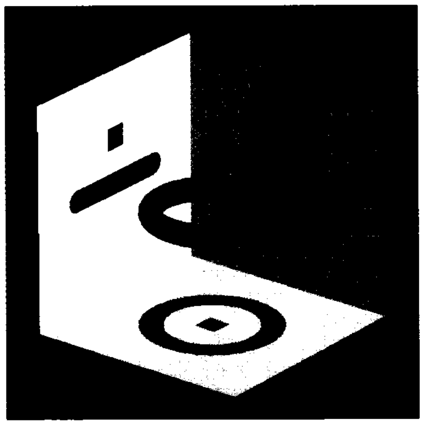


Fig. 12. A scene that uses the shadow-function.

Fig. 12 shows the result of executing this program.

We can add shadows to the Cone Tree visualization by replacing the last line of Program 11 by

```
root.add(ShadowWrapper(ConeTree(t)));
```

Fig. 11 (upper-right) shows the resulting scene.

D. Perspective Cues

We can further ease the viewer's perception of the spatial structure of the scene by drawing grid lines on the walls onto which the shadows are cast.

The function NewWall creates a wall object with purple grid lines and returns it. It utilizes the fact that Obliq-3D allows the edges of surfaces (objects composed of polygons) to be shown in a different color than the surface itself.

```
let NewWall =
 proc (fun)
   var pts = array2_new(11,11,ok);
   for i = 0 to 10 do
     for j = 0 to 10 do
       let p = fun(i,j);
       pts[i][j] :=
           [(0.2 • real_float(p[0])) - 1.0,
            (0.2 * real_float(p[1])) - 1.0,
            (0.2 * real_float(p[2])) - 1.0];
     end;
   end;
   let wall = QuadMeshGO_New(pts);
   SurfaceGO_SetEdgeVisibility(wall,true);
   SurfaceGO_SetEdgeColor(wall,"purple");
   wall
 end;
```

We want NewWall to be able to return walls that are aligned

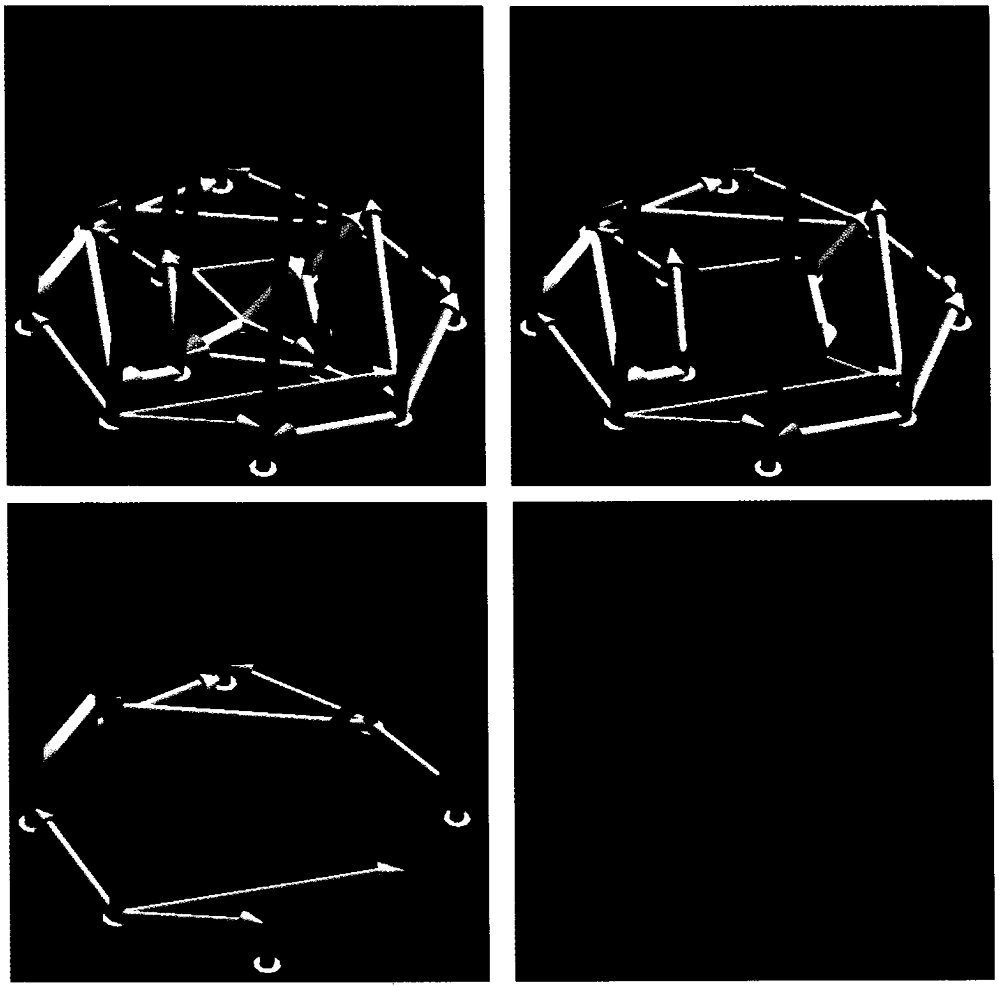


Fig. 13. These snapshots are from the animation of Dijkstra's shortest-path algorithm described in Section IV. The top-left snapshot shows the data just before entering the main loop. The top-right snapshot shows the algorithm about one-third complete. In the bottom-left snapshot, the algorithm is about 2/3 complete, and the snapshot at the bottom-right shows the algorithm upon completion.

to either the *xy*, the *xz*, or the *yz* plane. We achieve this by passing a lower-order function `fun` to it. This function takes two arguments and returns an array of length 3, representing a 3D point. `NewWall` creates an 11 × 11 array of points and uses `fun` to map the index of each cell to a point in 3-space. Based on this grid of points, it then creates a quadrilateral mesh, makes its surface edges visible, colors them purple, and returns the mesh.

We now modify `ShadowWrapper` to use `NewWall` by replacing the lines

```
top.add(PolygonGO_New([[-1,-1,-1],[-1,-1, 1],
                        [-1, 1, 1],[-1, 1,-1]]));
top.add(PolygonGO_New([[-1,-1,-1],[-1,-1, 1],
                        [1,-1, 1],[1,-1,-1]]));
top.add(PolygonGO_New([[-1,-1,-1],[-1, 1,-1],
                        [1, 1,-1],[1,-1,-1]]));
```

by

```
top.add (NewWall(proc(i,j) [i,j,0] end));
top.add (NewWall(proc(i,j) [i,0,j] end));
top.add (NewWall(proc(i,j) [0,i,j] end));
```

Fig. 11 (lower-left) shows the resulting scene.

Finally, we can further improve the viewer's perception of the scene by using a wide-angle camera that is located close to the Cone Tree. This is done by adding the line

```
root.changeCamera(PerspCameraGO_New([0,0,6],
                          [0,0,0],[0,1,0],0.7));
```

to the end of the program. This line replaces the camera that came with the standard root object by a camera that is closer to the Cone Tree and has a wider field-of-vision. Fig. 11 (lower-right) shows the resulting scene. The complete program for generating this scene (building a cone tree with animation, shadows, grid lines, and a wide-angle camera) is only 117 lines of code.

IV. CASE STUDY: SHORTEST-PATH ALGORITHM ANIMATION

This section contains a case study of using Obliq-3D with the Zeus algorithm animation system [2] to develop an animation of Dijkstra's shortest-path algorithm. We first describe the algorithm, then sketch the desired visualization of the algorithm, and finally present the actual implementation of the animation.

A. The Algorithm

The single-source shortest-path problem can be stated as follows: Given a directed graph $G = (V, E)$ with non-negative weighted edges and a designated vertex s, called the *source*, find the shortest path from s to all other vertices. The length of a path is defined to be the sum of the weights of the edges along the path. The following algorithm, due to Dijkstra [13], solves this problem:

```
for all $v \in V$ do $D(v) := \infty$ endfor
$D(s) := 0$; $S := \varnothing$
while $V \setminus S \neq \varnothing$ do
  let $u \in V \setminus S$ such that $D(u)$ is minimal
  $S := S \cup \{u\}$
  for all neighbors $v$ of $u$ do
    $D(v) := \min\{D(v), D(u) + W(u, v)\}$
  endfor
endwhile
```

In this pseudo-code, $W(u, v)$ is the weight of the edge from u to v, S is the set of vertices that have been explored thus far, and $D(v)$ is an upper bound on the distance from s to v. Moreover, for all $v \in S$, $D(v)$ is the length of the shortest path from s to v. The notation $V \setminus S$ denotes those elements in V that are not also in S.

B. The Desired Visualization

An interesting 3D animation of this algorithm is shown in Fig. 13. The vertices of the graph are displayed as white disks in the xy plane. Above each vertex v is a green column representing $D(v)$, the best distance from s to v known so far. Initially, the columns above each vertex other than s are infinitely high. An edge from u to v with weight $W(u, v)$ is shown by a white arrow that starts at the column over u at height 0 and ends at the column over v at height $W(u, v)$.

Whenever a vertex u is selected to be added to S, the color of the corresponding disk changes from white to red. The addition $D(u) + W(u, v)$ is animated by highlighting the arrow corresponding to the edge (u, v) and lifting the arrow such that its tail starts at the top of u's column (i.e., raising it by $D(u)$). If $D(u) + W(u, v)$ is smaller than $D(v)$, the top of the arrow will still touch the green column over $D(v)$. In this case, we shrink the column over v to height $D(u) + W(u, v)$ to reflect the assignment of a new value to $D(v)$, and color the arrow red, to indicate that it has become part of the shortest-path tree. Otherwise, the head of the arrow is higher than the green column, and the arrow simply disappears.

Upon completion, the 3D view shows a set of red arrows that form the shortest-path tree and a set of green columns which represent the best distance $D(v)$ from s to v.

C. Implementation of the Animation

In the Zeus framework, strategically important points of an algorithm are annotated with procedure calls that generate "interesting events." These events are reported to the Zeus event manager, which in turn forwards them to all interested views. Each view responds to interesting events by drawing appropriate images. The advantages of this methodology are described elsewhere [6].

C.1. The Interesting Events

The interesting events for Dijkstra's shortest-path algorithm (and many other shortest-path algorithms) are as follows:

`addVertex`(u, x, y, d) adds a vertex u (where u is an integer identifying the vertex) to the graph. The vertex is shown at position (x, y) in the xy plane. In addition, $D(u)$ is initialized to d.

`selectVertex`(u) indicates that u was added to S.

`addEdge`(u, v, w) adds an edge from u to v with weight w to the graph.

`raiseEdge`(u, v, d) visualizes the addition $D(u) + W(u, v)$ by raising the edge (u, v) by d (where the caller passes $D(u)$ for d).

`lowerDist`(u, d) indicates that $D(u)$ gets lowered to d.

`promoteEdge`(u, v) indicates that the edge (u, v) is part of the shortest-path tree.

`demoteEdge`(u, v) indicates that the edge (u, v) is not part of the shortest-path tree.

In addition, we need another event for house keeping purposes:

- `start`(m) is called at the very beginning of an algorithm's execution; it initializes the view to hold up to m vertices and up to m^2 edges.

C.2. Annotating the Algorithm

Here is an annotated version of the algorithm we showed in Section IV.A.

```
views.start($|V|$)
for all $v \in V$ do $D(v) := \infty$ endfor
$D(s) := 0$; $S := \varnothing$
for all $v \in V$ do views.addVertex($v$, $v_x$, $v_y$, $D(v)$) endfor
for all $(u, v) \in E$ do views.addEdge($u$, $v$, $W(u, v)$) endfor
while $V \setminus S \neq \varnothing$ do
  let $u \in V \setminus S$ such that $D(u)$ is minimal
  $S := S \cup \{u\}$
  views.selectVertex($u$)
  for all neighbors $v$ of $u$ do
    views.raiseEdge($u$, $v$, $D(u)$)
    if $D(v) < D(u) + W(u, v)$ then
      views.demoteEdge($u$, $v$)
    else
      $D(v) := D(u) + W(u, v)$
      views.promoteEdge($u$, $v$)
      views.lowerDist($v$, $D(v)$)
    endif
  endfor
endwhile
```

In this pseudo-code, `views` is the dispatcher provided by Zeus. The dispatcher notifies all views selected by the user of events.

C.3. The View

A view is an Obliq object that has a method corresponding to each interesting event and a number of data fields. In this view, the data fields are as follows: a `RootGO` object that contains all graphical objects of the scene, together with a camera and light sources; arrays of graphical objects holding the disks (vertices), columns (distances), arrows (graph edges), and shortest-path tree edges; and a synchronous animation handle for triggering animations. This leads us to a skeletal view:

```
let view = {
 scene  => RootGO_NewStd(),
 ah     => AnimHandle_New(),
 verts  => ok, (* initialized by start method *)
 dists  => ok, (* initialized by start method *)
 parent => ok, (* initialized by start method *)
 edges  => ok, (* initialized by start method *)
 start        => meth(self,m)... end,
 addVertex    => meth(self,u,x,y,d)... end,
 selectVertex => meth(self,u)... end,
 addEdge      => meth(self,u,v,w)... end,
 raiseEdge    => meth(self,u,v,z)... end,
 lowerDist    => meth(self,u,z)... end,
 promoteEdge  => meth(self,u,v)... end,
 demoteEdge   => meth(self,u,v)... end
};
```

The Zeus system has a control panel that allows the user to select an algorithm and attach any number of views to it. Whenever the user creates a new 3D view, a new Obliq interpreter is started that reads the view definition. The algorithm and all views run in the same process, but in different threads (thread creation is very lightweight). The expression above creates a new object `view` and initializes `view.scene` to be a `RootGO` and `view.ah` to be an animation handle.

The remainder of this section fleshes out the eight methods of `view`, which correspond to the eight interesting events:

- The `start` method is responsible for initializing `view.verts`, `view.dists`, and `view.parent` to be arrays of size *m* and `view.edges` to be an *m* × *m* array. The elements of the newly created arrays are initialized to the dummy value `ok`. Here is the code:

```
start => meth(self, m)
 self.verts := array_new(m, ok);
 self.dists := array_new(m, ok);
 self.parent := array_new(m, ok);
 self.edges := array2_new(m, m, ok);
end
```

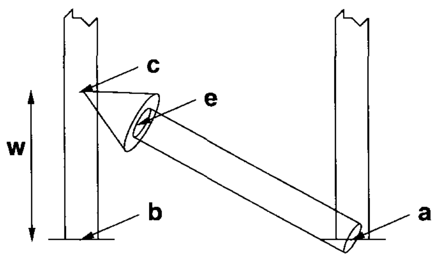
- The `addVertex` method adds a new vertex to the view. Vertices are represented by white disks that lie in the *xy* plane. Above each vertex, we also show a green column of height *d*, provided that *d* is greater than 0. The location of the cylinder's base is constant, while its top is controlled by a synchronous point property value.

```
addVertex => meth(self,u,x,y,d)
 self.verts[u] :=
           DiskGO_New([x,y,0],[0,0,1], 0.2);
 self.scene.add(self.verts[u]);
 if d > 0.0 then
  let top =
        PointProp_NewSync(self.ah,[x,y,d]);
  self.dists[u] :=
        CylinderGO_New([x,y,0], top, 0.1);
  SurfaceGO_SetColor(self.dists[u],"green");
  self.scene.add(self.dists[u]);
 end;
end
```

- The `selectVertex` method indicates that a vertex *u* has been added to the set *S* by coloring *u*'s disk red:

```
selectVertex => meth(self,u)
 SurfaceGO_SetColor(self.verts[u],"red");
end
```

- The `addEdge` method adds an edge (represented by an arrow) from vertex *u* to vertex *v*. The arrow starts at the disk representing *u* and ends at the column over *v* at height *w*. An arrow is composed of a cone, a cylinder, and two disks; its geometry is computed based on the "Center" property of the disks representing the vertices to which it is attached. The following diagram illustrates this relationship:



And here is the code for the method:

```
addEdge => meth(self,u,v,w)
 let a =
     DiskGO_GetCenter(self.verts[u]).get();
 let b =
     DiskGO_GetCenter(self.verts[v]).get();
 let c = Point3_Plus(b,[0,0,w]);
 let d = Point3_Minus(c,a);
 let e =
     Point3_Minus(c,Point3_ScaleToLen(d,0.4));
 let grp = GroupGO_New();
 GO_SetTransform(grp,TransformProp_NewSync(
                 self.ah,Matrix4_Id));
 grp.add(DiskGO_New(a,d,0.1));
 grp.add(CylinderGO_New(a,e,0.1));
 grp.add(DiskGO_New(e,d,0.2));
 grp.add(ConeGO_New(e,c,0.2));
 self.edges[u][v] := grp;
 self.scene.add(grp);
end
```

- The `raiseEdge` method highlights the edge from *u* to *v* by coloring it yellow and then lifting it up by *z*. The arrow is moved by sending a `translate` request to its *transformation property*. The translation is controlled by the animation handle `self.ah` and takes two seconds to complete. Calling `self.ah.animate()` causes all animation requests controlled by `self.ah` to be processed.

```
raiseEdge => meth(self,u,v,z)
 SurfaceGO_SetColor(
                self.edges[u][v],"yellow");
 let pv = GO_GetTransform(self.edges[u][v]);
 pv.getBeh().translate(0,0,z,0,2);
 self.ah.animate();
end
```

- The method `lowerDist` indicates that the cost *D*(*u*) of vertex *u* has decreased, by shrinking the green cylinder representing *D*(*u*). This is done by sending a `linMoveTo` ("move over a linear path to") request to the "Point2" property of the cylinder. Again, the animation takes two seconds.

```
lowerDist => meth(self,u,z)
 let pv = CylinderGO_GetPoint2(self.dists[u]);
 let p = pv.get();
 pv.getBeh().linMoveTo([p[0], p[1], z], 0, 2);
 self.ah.animate();
end
```

- The method `promoteEdge` indicates that (*u*, *v*), the edge that is currently highlighted, has become part of the shortest-path tree. This is indicated by coloring the edge red. If there already was a red edge leading to *v*, it is removed from the view.

```
promoteEdge => meth(self,u,v)
 SurfaceGO_SetColor(self.edges[u][v],"red");
 if self.parent[v] isnot ok then
   self.demoteEdge(self.parent[v],v);
 end;
 self.parent[v] := u;
end
```

- Finally, the method `demoteEdge` removes the edge (*u*, *v*) from the view:

```
demoteEdge => meth(self,u,v)
  self.scene.remove(self.edges[u][v]);
end
```

This completes our example. The complete view is about 70 lines of Obliq-3D code, compared to the roughly 2,000 lines of Modula–3 code of the PEXlib-based version that generated the animations presented in [7]. This measure is honest; we did not add any functionality (such as a new class `ArrowGO`) to the base library in order to optimize this example. Furthermore, turnaround time during the design of this view was limited only by the design process per se (and our typing speed), whereas compiling a single Modula–3 file and relinking with the Zeus system takes several minutes.

C.4. The Actual Algorithm

Here we present a version of the algorithm, implemented in Obliq, that drives the view from Section IV.C.3 directly, without using the Zeus event dispatcher:

```
let alg = proc (V,E)
 var D = array_new(#(V), 10.0);
  D[0] := 0.0;
  var S = array_gen(#(V), (proc(i) i end));
  view.start (#(V));
  for i = 0 to #(V) - 1 do
    view.addVertex(i,V[i].x,V[i].y,D[i]);
  end;
  foreach e in E do
    view.addEdge(e.f,e.t,e.w);
  end;

  loop
    var best = {i => -1,d => 1000.0};
    foreach i in S do
      if i isnot ok then
        if D[i] < best.d then
          best := {i => i, d => D[i]};
        end;
      end;
    end;
    let u = best.i;
    if u is -1 then exit end;
    S[u] := ok;
    view.selectVertex(u);
    foreach e in E do
      if e.f is u then
        let v = e.t;
        view.raiseEdge(u,e.t,D[u]);
        if D[v] < D[u] + e.w then
          view.demoteEdge(u,v);
        else
          D[v] := D[u] + e.w;
          view.promoteEdge(u,v);
          view.lowerDist(v, D[u] + e.w);
        end;
      end;
    end;
  end;
end;
```

To start the animation, we define the data structures holding the vertices and edges, and then invoke the algorithm:

```
let V = [
          {x => 0.0, y => 0.0},
          {x => -1.0, y => -1.0},
          ...
        ];
let E = [
          {f => 0, t => 1, w => 0.3},
          {f => 0, t => 2, w => 0.5},
          ...
        ];
alg(V,E);
```

V. Implementation

Obliq-3D consists of two major layers: an animation library called ANIM3D that provides a Modula–3 3D programming interface and a customized Obliq interpreter that allows the client to call ANIM3D procedures from within Obliq. Both ANIM3D and the Obliq interpreter are themselves implemented in Modula–3.

Every 3D-specific Obliq object has a Modula–3 counterpart. Calling object creation functions such as `SphereGO_New` not only creates an Obliq object (a `SphereGO`), but also a corresponding Modula–3 object (a `SphereGO.T`), and links the two objects together.

Adding a graphical object *o* to a group *g* does not actually establish a connection between the Obliq objects representing *o* and *g*, but rather between their Modula–3 counterparts. So, the scene graph is represented internally by a Modula–3 data structure.

The Obliq-3D runtime contains a dedicated thread, called the *animation server thread*, that periodically locks the scene graph, determines if there have been any changes (or "damages") since the last rendering, and redraws those parts of the scene that have changed.

Damage is caused by operations that change the structure of the scene graph and by property values whose values differ from those of the prior rendering.

Detecting changes to the scene graph is trivial, because changes are always triggered by client operations. Examples of such operations are adding a graphical object to a group, attaching a property to a graphical object, and changing the behavior of a property value. The operation that inflicts the damage sets a flag.

Detecting changes in property values is harder. In particular, the behavior of asynchronous and dependent property values is defined by client-supplied Obliq methods, which take a time as an argument and return a result. It is not possible to decide whether the result of a method at a given time differs from the result at an earlier time without actually evaluating the method. This means that at the onset of a rendering cycle, the animation server thread must traverse the portions of a scene graph that are reachable from root objects and call the Obliq interpreter to evaluate the asynchronous and dependent property values encountered along the way. Fortunately, the cost for these interpreter invocations is insignificant when compared with the cost of rendering (at least on our hardware and for the animations we have built). Also, evaluation results are cached, so the Obliq interpreter evaluates a property value at most once per rendering cycle.

Once the animation server detects that the content of the window corresponding to a particular root object has changed, it redraws the window. We improve rendering speed by caching primitive objects, such as spheres. In PEX, we use "structures" for caching; in OpenGL, we use "display lists." Structures and display lists are expensive in terms of memory, so caching every graphical object in the scene graph would be problematic. This problem is amplified when using a high-level interpreted language like Obliq, where the user normally does not perform explicit memory management. For this reason, we do not cache all graphical objects in the scene graph; instead, we cache "prototypical objects." For instance, we cache the unit sphere at various resolutions, where resolution refers to the number of triangles per sphere. We then use the cached unit-spheres to render arbitrary spheres by applying appropriate transformation matrices.

VI. Related Work

There are two bodies of work related to Obliq-3D: general-purpose 3D animation libraries and algorithm animation systems used for developing 3D views.

The most closely related general-purpose animation library is Silicon Graphics' Inventor [28], [29], an object-oriented graphics library with a C++ application programmers' interface. Obliq-3D adopts some of Inventor's key ideas: Scenes are modeled as directed acyclic graphs of graphical objects, and geometric primitives, cameras, and light sources are treated uniformly. But there are also some important differences. In Inventor, both shapes and properties can be added as nodes to the scene graph. Therefore, the order in which nodes are inserted into the graph affects the appearance of the scene: Inserting first a color node and then a sphere node will produce a different image than that obtained by adding them in reverse. Another key difference is that Inventor properties are not inherently time-variant. Animation is achieved through *engines*, which change the state of property nodes and shape nodes. Engines are very similar to Obliq-3D's asynchronous and dependent behaviors; however, it requires a fair amount of work (involving a chain of various kinds of engines) to achieve the effect of Obliq-3D's synchronous behaviors. Finally, Inventor lacks an interpreted language, a feature that we have found crucial to rapid prototyping. The lack of an interpreter becomes a more acute problem when developing visualizations, a task that typically requires many design iterations.

TBAG [15] is a recent tool kit developed at SunSoft for building interactive 3D animations. It too has many similarities with Obliq-3D: Both systems distinguish between values that define geometry and values that define attributes (such as color); both allow the user to impose a hierarchical structure on the geometry of the scene; both systems treat light sources like any other geometric primitive; and both systems have the notion of time-varying values. TBAG allows for a more general class of constraints than Obliq-3D, and it provides support for building distributed, collaborative applications. Like Inventor, TBAG does not have an embedded interpreted language. Also, as of this writing, TBAG is not available.

The earliest 3D animation system we know of to provide an interpreted language is the Actor/Scriptor Animation System (ASAS) developed by Reynolds at the Architecture Machine Group at MIT [22]. The interpreted language used in ASAS is based on the Actor paradigm and uses Lisp as the underlying substrate.

The Brown (University) Animation Generation System (BAGS) [27] is another influential 3D animation system that features an interpreted language, called SCEFO, to describe the structure and animation behavior of a scene. Although quite powerful, SCEFO is only an animation language, not a full-fledged programming language. It provides assignment and iteration constructs, a set of arithmetic functions, and procedural abstraction, but no conditionals. However, BAGS allows the user to write C code stubs that can be compiled into the system and can then be called from SCEFO.

BAGS was succeeded by the Unified Graphics Architecture (UGA) [33], [17], which uses an interpreted language, called FLESH, that is based on the prototype-and-delegation paradigm. Scenes are modeled as collections of objects. Attached to each object is a list of "change operators" that define some aspect of the object's appearance, such as its color, transformation, and even its shape. Moreover, the change operators are functions of time; they are the basis for animation in UGA.

Alice [11], [12], developed at the University of Virginia, is another rapid prototyping system for creating 3D animations. It uses Python as its embedded interpreted language. There are many similarities between Alice and Obliq-3D. Both systems use an object-oriented interpreted language to allow for rapid prototyping. Both separate the application from the rendering. (Alice uses a process to perform the rendering; Obliq-3D uses a separate animation server thread.) Both systems allow for hierarchical grouping of graphical objects. However, the two systems differ in their basic animation model: Alice treats a scene as a hierarchy of graphical objects, each of which has a list of *action routines* that are called each frame of the simulation and that are responsible for changing the internal states of the objects. New animation effects are created by defining new action routines and adding them to the graphical object. Obliq-3D, on the other hand, uses *time-variant properties* to specify the appearance of objects. In other words, Alice performs frame-based animation, whereas Obliq-3D has an explicit notion of time.

We now describe three systems designed for producing 3D algorithm animations: Pavane, Polka3D, and GASP. These systems tend to provide less powerful primitives that the more general 3D rendering systems described so far.

In Pavane [24], the computational model is based on tuple-spaces and mappings between them. Entering tuples into the "animation tuple space" has the side-effect of updating the view. A small collection of 3D primitives are available (points, lines, polygons, circles, and spheres), and the only animation primitives are to change the positions of the primitives.

Polka3D [25], like Zeus, follows the BALSA model for animating algorithms. Algorithms communicate with views using "interesting events," and views draw on the screen in response to the events. The graphics library is similar to Obliq-3D, but oriented primarily to algorithm animations. Unlike our use of Obliq-3D within Zeus, views are not interpreted, so turnaround time is not instantaneous.

The GASP [31] system is tuned for developing animations of computational geometry algorithms involving three (and two) dimensions. Because a primary goal of the system is to isolate the user from details of how the graphics is done, a user is limited to choosing from among a collection of animation effects supplied by the system. The viewing choices are typically stored in a separate "style" file that is read by the system at runtime; thus, GASP provides rapid turnaround. However, it does not provide the flexibility to develop views with arbitrary animation effects.

VII. Current Status and Future Directions

We are currently developing an OpenGL back-end for the system. The home page for Obliq-3D,

```
http://www.research.digital.com/SRC/3D-animate
```

contains current information about the system, including the availability of the code.

Thus far, Obliq-3D has only been used within our research lab. In particular, the authors have animated a handful of algorithms (see Figs. 1 and 13), and another colleague has developed animations of some advanced mathematical concepts (see Fig. 14) [20], [21].

We have just begun to work on a distributed version of the system. In particular, we would like to be able to view the same scene on different machines in a heterogeneous environment, where the topology of the sharing varies dynamically. Fortunately, Obliq provides us with a rich infrastructure for distributed computation.

VIII. Conclusions

The first part of this paper described Obliq-3D, a high-level, fast-turnaround animation system. The second part showed some nontrivial uses of the system, for implementing animated Cone Trees and general-purpose shadow casting. The third part presented a case study showing how to use Obliq-3D within the Zeus algorithm animation system for producing a 3D visualization of a graph-traversal algorithm. The final parts of the paper described the implementation of Obliq-3D, its relationship to other systems, and the current status of the system.

Obliq-3D is based on three concepts: scenes are described by directed acyclic graphs of graphical objects, time-variant property values are the basis for creating animations, and callbacks are the mechanism by which reactive behavior is specified. These concepts provide a simple yet powerful framework for building animations. The system provides fast turnaround by incorporating an interpretive language that allows the user to modify the code of a program even as it runs.

Our experience with Obliq-3D convinces us that the combination of a high-level language and a fast-turnaround environment are instrumental for developing effective animations.

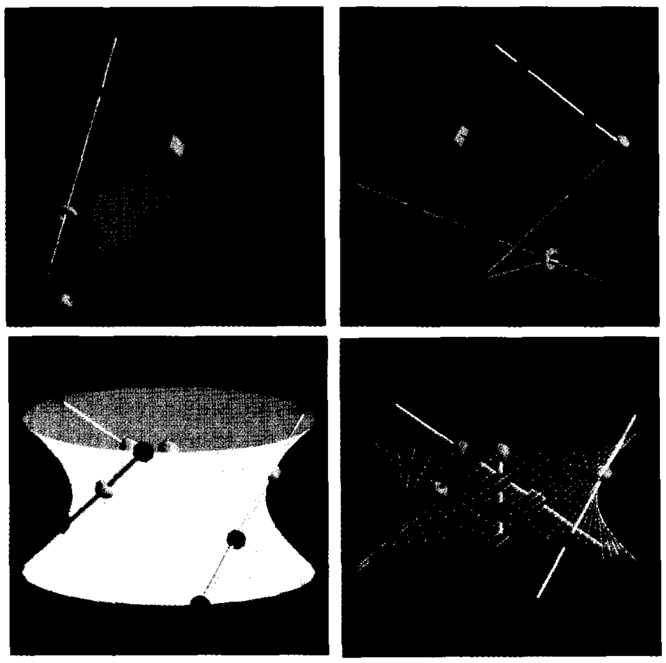


Fig. 14. These scenes show instances of the *budget configurations* $B_{2,1,1}$ and $B_{1,1,1,1}$. The Budget Rules, recently discovered by Ramshaw and Saxe, determine a family of intriguing configurations, some familiar and some novel. They used Obliq-3D to illustrate the novel budget configurations that live in 3-space.

Acknowledgments

Lyle Ramshaw was a gracious and patient alpha-tester of the system. Allan Heydon improved the clarity of this paper significantly. We appreciate their help.

References

[1] J.F. Blinn, "Jim Blinn's corner: Me and my (fake) Shadow," *IEEE Computer Graphics and Applications*, vol. 8, no. 1, pp. 82-86, Jan. 1988.

[2] M.H. Brown, "Zeus: A system for algorithm animation and multi-view editing," *1991 IEEE Workshop on Visual Languages*, pp. 4-9, Oct. 1991.

[3] M.H. Brown, "The 1992 SRC algorithm animation festival," *1993 IEEE Symp. Visual Languages*, pp. 116-123, Aug. 1993.

[4] M.H. Brown, "The 1993 SRC algorithm animation festival," Research Report 126, Digital Equipment Corp., Systems Res. Ctr., Palo Alto, Calif., 1994.

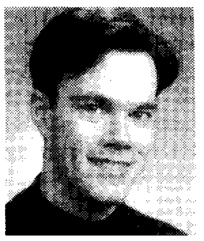
[5] M.H. Brown and J. Hershberger, "Color and sound in algorithm animation," *Computer*, vol. 25, no. 12, pp. 52-63, Dec. 1992.

[6] M.H. Brown and R. Sedgewick, "A system for algorithm animation," *Computer Graphics*, vol. 18, no. 3, pp. 177-186, July 1984.

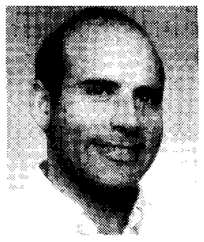
[7] M.H. Brown and M.A. Najork, "Algorithm animation using 3D interactive graphics," *ACM Symp. User Interface Software and Technology*, pp. 93–100, Nov. 1993. Also appeared as Research Report 110a, Digital Equipment Corp., Systems Res. Ctr., Palo Alto, Calif., Sept. 1993. There is an accompanying video tape, SRC Research Report 110b.

[8] L. Cardelli, "A language with distributed scope," 22nd Ann. *ACM Symp. Principles of Programming Languages*, pp. 286-297, Jan. 1995.

[9] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, W. Kalsow, and G. Nelson, "Modula-3 language definition," *Sigplan Notices*, vol. 27, no. 8, pp. 15-42, Aug. 1992.
[10] B.-W. Chang and D. Ungar, "Animation: From cartoons to the user interface," *ACM Symp. User Interface Software and Technology*, pp. 45-55, Nov. 1993.
[11] M. Conway, R. Pausch, R. Gossweiler, and T. Burnette, "Alice: A rapid prototyping system for building virtual environments," *Conf. Companion, CHI '94*, pp. 295-296, Apr. 1994.
[12] R. DeLine, "Alice: A rapid prototyping system for three-dimensional interactive graphical environments," Univ. of Virginia, Dept. of Computer Science, May 1993.
[13] E.W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269-271, 1959.
[14] R.A. Duisberg, "Animated graphical interfaces using temporal constraints," *ACM CHI '86 Conf. Human Factors in Computing*, pp. 131-136, Apr. 1986.
[15] C. Elliott, G. Schechter, R. Yeung, and S. Abi-Ezzi, "TBAG: A high level framework for interactive, animated 3D graphics applications," Computer Graphics Proc., *Ann. Conf. Series (SIGGRAPH '94 Proc.)*, pp. 421-434, July 1994.
[16] K.P. Herndon, R.C. Zeleznik, D.C. Robbins, D.B. Conner, S.S. Snibbe, and A. van Dam, "Interactive shadows," *ACM Symp. User Interface Software and Technology*, pp. 1-6, Nov. 1992.
[17] P.M. Hubbard, M.M. Wloka, and R.C. Zeleznik, "UGA: A unified graphics architecture," Technical Report CS-91-30, Brown Univ., Dept. of Computer Science, Providence, R.I., June 1991.
[18] M.A. Najork and M.H. Brown, "A library for visualizing combinatorial structures," *IEEE Visualization '94*, pp. 164-171, Oct. 1994. Also appeared as Research Report 128a, Digital Equipment Corp., Systems Res. Ctr., Palo Alto, Calif., Sept. 1994. There is an accompanying video tape, SRC Research Report 128b.
[19] M.A. Najork, "Obliq-3D tutorial and reference manual," Research Report 129, Digital Equipment Corp., Systems Res. Ctr., Palo Alto, Calif., Dec. 1994.
[20] L. Ramshaw and J.B. Saxe, "From quadrangular sets to the budget matroids," Research Report 134a, Digital Equipment Corp., Systems Res. Ctr., Palo Alto, Calif. (forthcoming).
[21] L. Ramshaw, "Introducing the budget configurations," Research Report 134b, Digital Equipment Corp., Systems Res. Ctr., Palo Alto, Calif. (forthcoming).
[22] C.W. Reynolds, "Computer animation with scripts and actors," *ACM Computer Graphics (SIGGRAPH '82)*, vol. 16, no. 3, pp. 289-296, July 1982.
[23] G.G. Robertson, J.D. Mackinlay, and S.K. Card, "Cone trees: Animated 3D visualizations of hierarchical information," *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems*, pp. 189-194, Apr. 1991.
[24] G.-C. Roman, K.C. Cox, C.D. Wilcox, and J.Y. Plun, "Pavane: A system for declarative visualization of concurrent computations," *J. Visual Languages and Computing*, vol. 3, no. 2, pp. 161-193, June 1992.
[25] J.T. Stasko and J.F. Wehrli, "Three-dimensional computation visualization," *1993 IEEE Symp. Visual Languages*, pp. 100-107, Aug. 1993.
[26] J.T. Stasko, "TANGO: A framework and system for algorithm animation," *Computer*, vol. 23, no. 9, pp. 27-39, Sept. 1990.
[27] P.S. Strauss, "BAGS: The Brown animation generation system," Technical Report No. CS-88-22, Brown Univ., Dept. of Computer Science, Providence, R.I., May 1988.
[28] P.S. Strauss, "IRIS Inventor, a 3D graphics toolkit," *ACM SIGPLAN Notices* (*OOPSLA '93 Proc.*), vol. 28, no. 10, pp. 192-200, Oct. 1993.
[29] P.S. Strauss and R. Carey, "An object-oriented 3D graphics toolkit," *ACM Computer Graphics* (*SIGGRAPH '92*), vol. 26, no. 2, pp. 341-349, July 1992.
[30] D.J. Sturman, D. Zeltzer, and S. Pieper, "Hands-on interaction with virtual environments," *ACM Symp. User Interface Software and Technology*, pp. 19-24, 1989.
[31] A. Tal and D. Dobkin, "GASP—a system for visualizing geometric algorithms," *IEEE Visualization '94*, pp. 149-155, Oct. 1994.
[32] L.R. Wanger, "The effect of shadow quality on the perception of spatial relationships in computer generated imagery," Symp. Interactive Three-Dimensional Graphics, pp. 39-42, Mar. 1992.
[33] R.C. Zeleznik, D.B. Conner, M.M. Wloka, D.G. Aliaga, N.T. Huang, P.M. Hubbard, B. Knep, H. Kaufman, J.G. Hughes, and A. van Dam, "An object-oriented framework for the integration of interactive animation techniques," *Computer Graphics* (*SIGGRAPH '91 Proc.*), vol. 25, no. 4, pp. 105-112, July 1991.



Marc Najork is a member of the research staff at Digital Equipment Corporation's Systems Research Center. His current research focuses on 3D animation, information visualization, algorithm animation, user interfaces, and visual programming. Dr. Najork received his PhD in Computer Science from the University of Illinois at Urbana-Champaign in 1994, where he developed Cube, a three-dimensional visual programming language.



Marc H. Brown has been a member of the research staff at Digital Equipment Corporation's Systems Research Center since receiving his PhD in Computer Science from Brown University in 1987, where he worked with Andries van Dam and Robert Sedgewick on the "Electronic Classroom" project. Dr. Brown was primarily responsible for the BALSA system, the courseware environment used in the classroom for interactive animation of computer programs. This led to his dissertation, *Algorithm Animation*, which was selected as a 1987 ACM Distinguished Dissertation. His current research focuses on (parallel) algorithm animation and auralization, data visualization, user interface toolkits and techniques, Web browsing, and computer science education.