Real Clock Time Animation Support for Developing Software Visualizations

J. T. Stasko
D. S. McCrickard

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280 U.S.A.
E-mail: {mccricks,stasko}@cc.gatech.edu

Abstract

Developers building software visualizations must use a graphics library and user interface toolkit as an underlying support platform. Often, these support environments are large, difficult to learn, low-level, and lacking primitives for capabilities such as animation. We have developed a graphics support environment called Polka-RC for building software visualizations. Polka-RC is a second generation system that leverages the continuous animation primitives of the mature system Polka, and adds the capability of specifying real clock time-based animation activations and durations. The new Polka-RC animation model also provides a flexible multiprocess program-to-visualization mapping. In this article we describe the Polka-RC methodology, list advantages of the approach, and describe how the methodology influences the design of software visualizations and algorithm animations.

Keywords: software visualization, algorithm animation, computer graphics, graphics libraries and toolkits

ACM Classifications: I.3.4 Graphics utilities, graphics packages; I.3.8 Computer Graphics applications; I.6.8 Simulation and Modeling, Types of simulation, animation.

1 Introduction

Learning how algorithms work, learning how to program, and learning how to debug programs are still challenging activities. Many instructional techniques and software tools have been developed over the past 30 years to aid these activities. This article examines the area of *software visualization*, the use of computer graphics, visualization, and animation to help illustrate how algorithms and programs work (Stasko & Patterson 1992, Price, Baecker & Small 1993). By providing concrete graphical depictions of the normally intangible abstract workings of programs, software visualizers seek to facilitate program understanding and comprehension.

Software visualization systems have been used for a number of different purposes, ranging from instructional aids for teaching algorithms (so called *algorithm animations* (Brown 1988*b*, Brown 1988*a*, Stasko 1990)) to software engineering tools to assist program development and debugging (Reiss 1985, Shimomura & Isoda 1991, Kimelman, Rosenburg & Roth 1994). In all software visualization systems, the visualization or animation depicted must be built using an underlying graphics support environment. This graphics support can be a powerful, but low-level toolkit such as Xlib for X Windows, or it can be an extremely high level interactive visual environment such as the Lens system (Mukherjea & Stasko 1994).

A trade-off exists in these two alternatives. Low-level toolkits are very powerful and can provide rich visualizations, yet they are usually quite large, difficult to learn, and building visualizations with them can be time-consuming. High-level environments, typically utilized in scenarios where end-users build their own visualizations, are smaller and easy to learn and use, but they typically provide only a restricted set of visualization and animation primitives. Consequently, software visualizers have developed their own software visualization support toolkits that are tuned to provide the types of graphics and animation one usually encounters in software visualization and algorithm animation systems.

Over time these toolkits have evolved to be quite sophisticated graphics systems. The early Balsa environments provided black-and-white images in multiple views (Brown & Sedgewick 1985); Animus provided temporal constraints in a Smalltalk based environment (Duisberg 1986); Tango added color and smooth animation primitives (Stasko 1990); Zeus added sound in a general object-oriented framework (Brown & Hershberger 1992); Polka-3D examined the use of 3D for software visualization (Stasko & Wehrli 1993); and recent systems have focused on high-level, powerful toolkit primitives (DeTreville 1993).

One area yet to be explored, however, is the capability of real clock time animations, that is, animations whose initiation and duration are specified in physical times of milliseconds, seconds, and so on. These types

of animation environments have long existed in 3D computer graphics where an exact frames-per-second rate is used, and recently they have surfaced in user interface development toolkits (Hudson & Stasko 1993).

We have built a software visualization support toolkit called Polka-RC (Polka Real Clock) that provides such primitives. Polka-RC is an evolution of the mature software visualization toolkit called Polka (Stasko & Kraemer 1993) that provides frame, as opposed to time, based animations. Readers familiar with the original Polka will be able to gain more from this article, though we try to explain key points about both so that the ideas are accessible to all. In the next section we provide a brief review of Polka and its components. After that, we describe Polka-RC and the primitives it provides, and we give some examples of its use. Finally, we discuss the trade-offs involved in this new methodology. Our initial experiences with Polka-RC have uncovered advantages and some challenges in a real clock-time based approach.

2 Polka

The original Polka software visualization toolkit was designed to support end-user developed software visualizations. In particular, its focus was on expressive algorithm animation-style views involving concurrent animations, thus facilitating illustrations of parallel programs. Polka is implemented in C++ on top of the X Window System.

Polka provides a small number of abstract data types, each with a rich set of operations. Animations are built within **Views** that can provide different, unique depictions of the program or algorithm being visualized and that inhabit one window on the display. Within a View, programmers manipulate three different data types:

- **Location** - A positional marker that denotes a particular position in the View coordinate system.
- **AnimObject** - A graphical object primitive such as a line, circle, rectangle, text, etc.
- **Action** - A motion or change primitive such as movement along a path or a change in color.

All animation in Polka is frame-based; that is, each Action such as a movement or a resize has a duration in frames. Programmers can specify any number of frames in an Action, but once that is done, the number does not change. For example, a movement between two locations may take 10 frames that correspond to ten equally spaced steps between the two locations. In addition to allowing the programmer to specify this frame duration, Polka includes Action operations to add or subtract frames from a default Action, thus slowing down or speeding up animations, respectively. Polka's run-time user interface also includes a speed control bar that adjusts the frame-to-frame display rate. That is, it allows changing the speed of the animation as a whole (all Actions are affected in the same way).

More specifically, consider the example below.

```
Circle *circ;
Loc *loc, *center;
Action *act;
int len;

circ = new Circle(this,1, 0.2,0.3, 0.1,
                  "red", 1.0);
circ->Originate(time);

center = circ->Where(PART_C);
loc = new Loc(0.6, 0.5);

act = new Action("MOVE", center, loc, 20);
len = circ->Program(time, act);
time = Animate(time, len);
```

This code creates a red circle centered at location (0.2, 0.3) with radius 0.1. It then proceeds to schedule and move the circle to position (0.6, 0.5) along a straight path of 20 steps or animation frames. The variable `time` is a member of the View class provided by Polka and it is used to maintain the current frame count. The routine `Program` binds an Action to an AnimObject and schedules it to occur at the specified frame number. This call returns the Action's length in frames. The final call, `Animate`, is critical here to actually generate the animation. It takes a beginning frame count and a duration as arguments, and generates that many new animation frames. It is the only way that animation occurs in Polka. Figure 1 illustrates the set of frames from this simple animation in Polka and it exhibits the system's user interface. The Polka-RC animations will have the same appearance and interface.

The Polka Action primitive `Interpolate` can be used to adjust the number of steps or offsets in a path. The call below makes a new movement action that covers the same x, y coordinates as as the one above, but with 30 steps instead of 20.

```
Action *b = act->Interpolate(1.5);
```

Although this frame-based methodology has proven to be powerful and useful, it can be problematic as well. If the code for an animation developed on a particular machine is transferred to a different machine with a faster CPU and X server, then the same number of frames will translate to a much faster animation, probably more so than the animation designer had intended. A similar problem occurs when a machine is heavily loaded with other processes. The animation then may simply be too slow. To a certain degree, the inclusion of a speed control bar in Polka addresses this relative timing problem. Nonetheless, the inability to specify precise times and durations can be frustrating.

Accordingly, we wondered what it would be like to be able to specify that a movement take 2.5 seconds

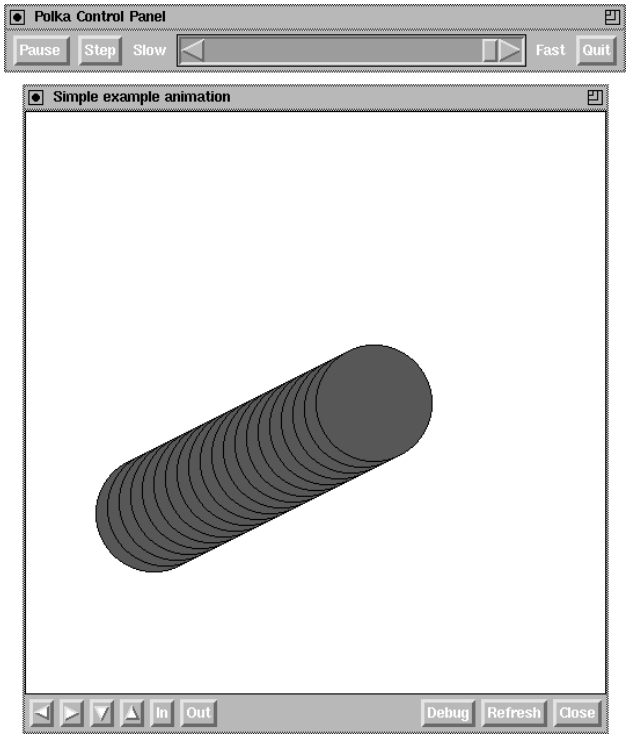

Figure 1: Superimposed animation frames from the example Polka code (circle outline added to show motion).

rather than 20 frames, for example. In particular, we would want the Action to take this duration regardless of what type of workstation it is running on and how busy the machine is. Also, we might want the motion to start slowly, speed up, then slow again near its termination, one of the classic animation techniques for presenting movement (Lassiter 1987). In addition, we want a model in which the animation occurs concurrently with the execution of the program it is representing, without the need for an explicit `Animate` call. Finally, we wondered how this type of toolkit would influence the design of software visualizations and animations. These inquiries led us to develop Polka-RC.

3 Polka-RC

The following sections describe the Polka-RC environment, highlighting the differences between it and the original Polka. We begin by describing the new real time animation primitives introduced in Polka-RC, which are partly modeled after those introduced in (Hudson & Stasko 1993). Next, we describe the two process, asynchronous communication model that the environment supports, and we give a small example of what code in Polka-RC looks like. Finally, we discuss some of the implementation details and issues raised by this new system.

3.1 Animation Primitives

Like Polka, Polka-RC contains a primitive called a View that is an abstraction for a graphical perspective or depiction of a program. Just as static data can have multiple views (such as a pie graph and a line graph), Polka-RC allows users to create several Views of a dynamic program. Each View is made up of one or more Scenes (user-defined member functions of their View subclass) that encapsulate different graphical behaviors to occur in the View. At run-time, events from the driver program cause the graphical updates in Scenes that make animation happen in each View.

Polka-RC provides four basic classes of objects which are created and manipulated in a View to implement an animation: AnimObject, Location, Trajectory, and Action. The Location and AnimObjects are largely unchanged from the original Polka. The AnimObject is simply a graphical object such as a line, circle, rectangle, text, etc., and is the only data type with a visual manifestation evident in a View. A Location is a marker position within the animation coordinate system and is useful for positioning AnimObjects and as the endpoints of movement actions. The Trajectory is a new data type, however, and the role of the Action has been expanded. These two data types are described in more detail below.

A **Traj**(ectory) is a path or curve along which an update to an AnimObject occurs. For example, an AnimObject can move along a Trajectory or use a Trajectory to define a change in fill pattern. Each Trajectory is defined by three elements: displacement, motion type, and pace.

The *displacement* specifies the beginning to ending distance of a Trajectory and can be specified in two ways, by designating the absolute (x, y) distance that the Trajectory covers or by designating two Locations that represent the "start" and the "end" of the Trajectory.

The *motion type* describes the type of curve or path form taken by the Trajectory. We have found three simple types to be useful: straight, clockwise, or counterclockwise. A straight Trajectory simply follows the straight line shortest path between its starting and ending points. The clockwise and counterclockwise Trajectories follow an arc of at most 180 degrees. Imagine that the points designating the distance to be traversed are points on a clock dial and the clock hand sweeps out the trajectory. Since two points could define a number of different arcs, Polka-RC enforces the additional condition that one of the points is either at nine or at three on the dial. The clock hand sweeps out the arc defined by these points in either a clockwise or counterclockwise manner depending on the motion type. These three motion types are provided to simplify the work of the animation designer. To create more complex arcs and curves, it is necessary to combine a number of simpler

arcs and lines using composition operations provided by Polka-RC.

The *pace* refers to the rate at which an AnimObject is modified along the Trajectory curve. In Polka, the pace was always uniform (each discrete step of a path corresponded to an animation frame). However, it can be useful to traverse a curve at different speeds depending on the current position on the curve. For example, it has been shown that slowly building up the speed of an object as it begins a motion (called "slow out") helps draw attention to the object (Lassiter 1987). Similarly, slowing down the speed of an object as it is about to stop (called "slow in") indicates to a viewer that the object is about to stop. A pace function is a way to specify the rate at which a curve is traversed. Mathematically, a pace function accepts a percent of time that has passed as a parameter and returns the percent of the total displacement on the curve that should be covered in that time. Polka-RC provides uniform and slow-in/slow-out pace functions as primitives, and it also allows designers to create and use their own pace functions.

An **Action** is an update to an AnimObject that changes its position or appearance, such as a movement or a color change. Two types of actions exist: discrete and continuous.

Discrete Actions happen at one specific point in time and do not change the size or position of an AnimObject. Polka-RC provides five discrete Actions:

- COLOR - change the AnimObject's color to some new color (any X Window color name or RGB-valued color).
- VIS - toggle the visibility of the AnimObject. If an AnimObject is invisible, other Actions can still be performed on it, but their results will not be seen until the AnimObject is made visible.
- RAISE - raise the AnimObject to the topmost viewing plane. Thus, if several AnimObjects overlap, RAISE can be used to raise an AnimObject to the top.
- LOWER - lower the AnimObject to the bottommost logical viewing plane (similar to the raise Action).
- ALTER - change the contents of a text string (valid only for the Text AnimObject type).

In addition to the Action type, an animation designer also specifies the AnimObject on which the Action will be performed and the time at which the Action will occur in order to create an animation. The methods for specifying the time at which a discrete Actions occurs are the same as the methods for specifying the start time for continuous Actions and are discussed later.

Here are some example calls that create discrete Actions.

```
Action vis("VIS", rect1,
           START_AT, Sec(15));              (1)

Action rai("RAISE", rect1,
           START_AFTER_START_OF, &vis, 0); (2)
```

The first Action changes the visibility of an AnimObject at time 15 (15 seconds after the animation commences). The second raises the rectangle to the topmost view plane at the same time as the visibility Action.

Unlike the discrete Actions, continuous Actions happen over a period of time and follow some path (represented by a Trajectory). The continuous Action types provided by Polka-RC are:

- MOVE - move the AnimObject along the Trajectory.
- RESIZE - resize the AnimObject. The different types of AnimObjects have different methods in which they are resized. Resizing a line changes the length of the line. Resizing a rectangle corresponds to dragging the upper right corner of the rectangle along the given Trajectory. For circles, the radius changes according to the x component of the Trajectory. On an ellipse, both the x and y size components can change, and so on.
- GRAB - grab and drag part of an AnimObject. For polylines, polygons, and splines, the Action moves only one specified vertex and leaves all others fixed (unlike RESIZE, which modifies all vertices following the specified one).
- FILL - change the fill style for the AnimObject. For objects such as rectangles, circles, ellipses and polygons, the X-coordinate of the Trajectory used as a parameter is interpreted as a modifier of the fill percentage (0-100) of the AnimObject.

In addition to the Action type, the AnimObject modified, and the Trajectory that the Action will cover, an animation designer must specify the start time and duration over which a continuous Action will occur.

Polka-RC provides several methods for specifying the starting time for each Action. The time simply can be given in seconds or milliseconds (assumed relative to the start of the animation); for example, the start time might be 15 seconds into the animation. Alternatively, it is possible to use the `Now()` function provided by Polka-RC to acquire the current time. Finally, the designer can specify the starting time with respect to another Action. In Example 2 above, the designer requested that an Action start at the same time as another Action. In Example 4 below, Action `a2` will start a half second after `a1` ends.

Polka-RC also provides two methods for specifying the ending time for continuous Actions. One simple method is to provide the duration of the Action in seconds or milliseconds (see Example 3 below). The second

method is to specify the velocity of the Action. Velocity is defined in terms of the percentage of the computer display that is covered in one second. In Example 4, the designer specifies the velocity to be 50, meaning it will take one second to move across 50 percent of the display. Velocity is useful when a number of AnimObjects' movements should occur at the same rate.

In the examples below, the first Action resizes a rectangle AnimObject so that it becomes wider (increase in x) and shorter (decrease in y) over a period of 1.5 seconds. The Action also uses a slow in/slow out pace. The second Action moves the rectangle from loc1 to loc2 in a straight line at velocity 50. Note that the first Action is specified to start at the current time and the second starts a half second after the first completes.

```
Action a1("RESIZE", rect1,
      Traj(CLOCKWISE,0.2,-0.1,slowinout),
      START_AT, Now(),
      DURATION, Sec(1.5));                (3)

Action a2("MOVE", rect1,
      Traj(STRAIGHT,loc1,loc2,uniform),
      START_AFTER_END_OF, &a1, Sec(0.5),
      VELOCITY,50)                        (4)
```

Once Actions such as these have been defined, they must be added to the list of "active" Actions manipulated by Polka-RC using the `Schedule` call.

```
Schedule(&a1);
```

When an Action is scheduled, it is added to the working set of Actions in the animation. Whenever a specified commencement time of an Action occurs, Polka-RC initiates the action.

The critical difference to the original Polka here is that no `Animate` routine is necessary to generate animations. Polka-RC constantly monitors Actions in the "background" and keeps updating them as quickly as it can. That is, animation is always occurring, or rather, has the potential to occur. It is not based on a particular call.

When Polka-RC processes an Action, it examines the beginning time and duration of the Action with respect to the current time. It then calculates where and how to draw the object, taking into account the Trajectory and pace function if they exist. If an Action is encountered that should have completed in the past, the AnimObject is simply updated to its resultant configuration. Thus, Polka-RC never falls significantly behind in an animation. If many operations are occurring and the animation server process is slow, it simply appears that objects change more discretely as opposed to a gradual smooth transition.

3.2 Organizing an Animation

In the original Polka, the program being visualized and its animation design code are usually written in separate files for modularity, then compiled into a single executable. Consequently, the transmission of events from program to animation is synchronous. That is, an event is passed to the animation, the corresponding animation is scheduled and executed, then control is transferred back to the driver program which then proceeds with dispatching the next event. In Polka-RC, the program and animation are written, compiled, and executed separately, as two different processes. The program communicates with the animation by sending it messages though a socket (see Figure 2).

The program being visualized (we call this the "driver") can be written in any language, but to communicate with the animation code, it must be able to connect to a socket. A socket is simply a generalization of a UNIX pipe in which neither process is an ancestor of the other. Thus, the animation does not need to know the name of the program that will connect to it, and the program does not need to know the animation name. The animation finds a free socket and advertises its id number. Polka-RC provides a library of socket functions used for connecting to a socket and sending messages through the socket. Any existing program easily can be annotated with socket connection calls and message passing calls in order to communicate with an animation program. Later, as the program actually executes, it dispatches a series of events across the socket to the animation. These events characterize the operations that are occurring in the program.

The animation code consists of three elements: a main routine, a Controller routine, and a set of animation service functions. The main routine contains the startup function calls for the animation. The controller maps the run-time events sent by the driver program to the appropriate animation functions. These functions are used to create and manipulate the graphical objects in the animation views that represent operations in the driver program.

Several reasons for separating the program and its animation exist. First, the program developer and animation developer might be different people. Since Polka-RC allows the units to be written and compiled separately, the developers can work independently.

Second, the separation of program and animation allows any algorithm to connect to any compatible animation. For example, several different animations of sorting algorithms may exist. A particular sorting algorithm does not have to be physically linked with each animation in order to be executed with it. Rather, it simply connects to the socket of the animation and dispatches events.

Finally, and perhaps most importantly, the animation component must now have its own independent control and event processing loop because of the real clock time aspect. In the original Polka, if the driver program failed to provide execution time for the animation, perhaps because it was waiting for input from the user or because it was performing many complex calculations, then the animation would be blocked. In Polka-

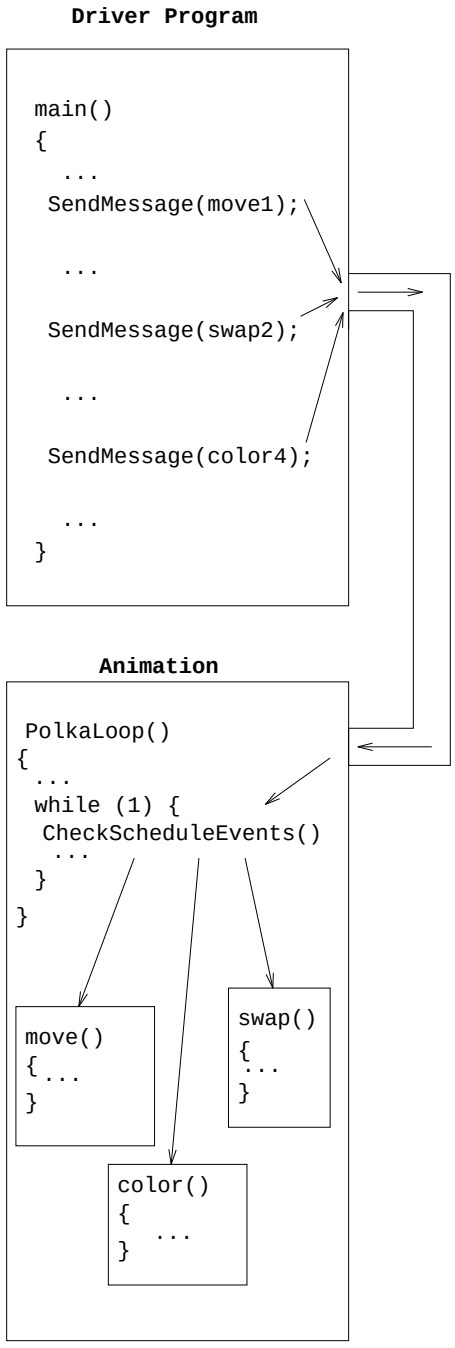

Figure 2: Model of program to animation communication in the Polka-RC framework.

RC the animation component cannot block—it updates graphical objects appropriately at regular intervals and processes events generated by the algorithm during the remaining time. Since the animation does not have the responsibility of executing the algorithm, Polka-RC animates complex algorithms more quickly and smoothly than the original Polka.

3.3 An Example

This section provides a brief example of what it is like to build a software visualization with Polka-RC. The two subsections below describe the two programs that must be created: one for the animation and one for the program which is being visualized and is driving the animation. We begin by describing the animation program and then follow this with the code for a driver program.

3.3.1 Animation Component

In Polka-RC as in Polka, an object of a class called Animator is the intermediary between the program being visualized and its animation. The Animator initially finds and advertises a free socket, then waits for the driver program to connect to the socket and to send messages. Each Polka-RC animation must have one Animator object (actually a derivation is always used) and one or more View objects. The Animator should include the declarations for the View(s) and the virtual function Controller. The View should include the animation functions available for the View (these are called Scenes) as well as any data structures which may be used.

The messages (events) sent by the program request these Scenes to take place. A Scene is simply the visual manifestation of an update in the program. For example, an event in a sorting program may correspond to a swap of two data values. This swap event can be mapped to one or more Scenes that will graphically represent the swap in the animation View(s); for example, a Scene may switch the positions of two rectangles that represent the swapped elements in the sorting program.

In the following example, two Views are created for a sorting algorithm animation. `BlocksView` may represent the values being sorted as a row of rectangles. A second View, `SticksView`, may represent distances between swapped items with lines. Parts of the declarations have been omitted for brevity.

```
class MyAnimator : public Animator {
   public:
      int Controller();
      BlocksView bv;
      SticksView sv;
      MyAnimator() {
         RegisterView(&bv);
         RegisterView(&sv);
         ...
```

```
    };
};


class BlocksView : public View {
   public:
      int Init(int);
      int Exchange(int,int);

   private:
      Rectangle *elts[100];
      double wid;
};

class SticksView : public View {
   ...
}
```

The main procedure of the animation program, part of which is shown below, has three basic tasks. First, it finds and advertises the socket to which the driver program will connect via the Animator member function `SocketInit`. Second, it creates the View windows where AnimObjects will be seen. Third, it enters a loop which waits for the driver program to connect and send messages over the socket. The Animator member function `PolkaLoop` performs the third task.

```
MyAnimator anim;

main()
{
  anim.SocketInit();
  anim.bv.Create("Polka Blocks View");
  anim.sv.Create("Polka Sticks View");
  ...
  anim.PolkaLoop();
}
```

The animation program also must contain a Controller routine that provides a mapping from the event names and parameters passed from the driver program to the member function names and parameters of the animation Views. When an event arrives over the socket from the driver program, its name (id) is stored in the Animator class variable, AlgoEvtName, and its parameters are stored in the member arrays, AnimInts, AnimDoubles, and AnimStrings, according to their types.

The example controller below maps the *INIT* and *SWAP* messages from the sorting algorithm to animation functions for the different Views. Note that a single event can map to any number of View Scenes.

```
int
MyAnimator::Controller()
{
   if (!strcmp(AlgoEvtName,"INIT")) {
      bv.Init(AnimInts[0]);
      sv.Init();
   }
   else if (!strcmp(AlgoEvtName,"SWAP")) {
      bv.Exchange(AnimInts[0], AnimInts[1]);
      sv.DrawLine(AnimInts[0] - AnimInts[1]);
   }
   else ...
   ...
}
```

The View functions (Scenes) contain the code that performs the animation. AnimObjects are created and Actions are performed on them. For example, a rectangle can be created with the following command.

```
int i;

...
Rectangle *elt[i] =
      new Rectangle(this, 1, 0.3, 0.3,
                    wid, i*0.4,
                    "orange", 0.625);
elt[i]->Originate( Now() );
```

The first parameter to the Rectangle creation is the View in which the rectangle will appear. The second is the visibility: 1 means the rectangle will be visible when it is first shown. The next two parameters are the initial location of the rectangle. The next two parameters are the width and height of the rectangle. The next parameter is the color and the final parameter is the fill percent. Creating an AnimObject allocates its internal data structures, but the member function `Originate` is necessary to specify the time at which the AnimObject should first appear in the View.

Location objects are often used in Trajectories to specify the "from" and "to" positions. Locations are easily specified; simply provide the x and y coordinates. For Trajectories, the curve type, displacement and pace function must be given. The displacement can be either an (x, y) displacement or "from" and "to" locations. Below are examples of Location and Trajectory definitions.

```
Loc *to = new Loc(0.05, 0.1);
Loc *from = new Loc(0.1, 0.15);

Traj *t1 = new Traj(CLOCKWISE,from,to,uniform);
Traj *t2 = new Traj(STRAIGHT,0.1,0.5,slowinout);
```

Finally, we will provide examples of Actions that will exchange the position of two rectangles and change the color of one of them.

```
Action mov1("MOVE", elt[i],
        Traj(CLOCKWISE,from,to,uniform),
        START_AT, ASAP(), VELOCITY, 50);
Schedule(&mov1);

Action mov2("MOVE", elt[i+1],
        Traj(CLOCKWISE,to,from,uniform),
        START_AFTER_START_OF, &mov1, Sec(0),
```

```
        VELOCITY, 50);
Schedule(&mov2);

Action col("COLOR=red", elt[i],
        START_AT, ASAP());
Schedule(&col);
```

The first Action is a clockwise movement of a rectangle. It is given a start time of `ASAP`, a value we have yet to discuss but that will be fully described in Section 4, and a velocity to cover half of the screen per second. The second Action is a movement of another rectangle. It starts at the same time as (0 seconds after the start of) the previous Action. Since the velocity and distances are the same, the two rectangles will stop at the same time.

The final Action is a discrete Action. It changes the color of one of the rectangles to red ASAP, essentially after the rectangles have finished moving. Note that no velocity or duration is given with this Action since it is discrete. Recall that all of the Actions must be scheduled in order to occur in the View.

3.3.2 Algorithm Component

In order to interact with the animation program, the driver program first must connect to the socket that has been acquired by the animation program. Therefore, the socket number must be passed to the driver. The easiest way to connect to a socket is to use the `SocketConnect` function provided by Polka-RC. If the socket number was passed as the first argument via the command line, this call would look like

```
SocketConnect(atoi(argv[1]));
```

Once the driver program has connected to the socket, messages can be passed to the animation program. Polka-RC provides a function `SendMessage` which is used to send messages across the socket.

There are two basic types of messages that the driver sends to the animation: Register messages and Send messages. Register messages are used to describe the events that the driver may dispatch. Each event can have a list of parameters whose types must be indicated in the Register message. Send messages are actual event transmissions from the driver program. When a Send message is received, the Animator reads the message to determine the event name and parameters, and it calls the controller function. Additionally, Polka-RC uses a special End message that tells the animation program that the algorithm is done sending messages and is closing the socket.

Every event that the program designer wishes to instantiate first must be registered with a Register message. The Register message contains the name of the event and the parameter type list. The parameter type list contains characters representing the type of parameters that will be sent much like a `printf` statement: d for integer, f for float, and s for string. A sorting algorithm might define a SWAP event with two integer parameters as follows.

```
SendMessage("Register SWAP dd");
```

Whenever the program designer wants to animate an event that occurred in the driver program, a Send message is sent. The message should include the event name followed by the list of parameters. Because `SendMessage` accepts only one string parameter, it is sometimes necessary to use a Unix `sprintf` command to compress all of the data into one string. An example is given here of the SWAP event registered previously with integer variable parameters for the indices of the two elements being exchanged.

```
char msg[80];

sprintf(msg,"Send SWAP %d %d", i, i+1);
SendMessage(msg);
```

A `SendMessage` can include an explicit time (perhaps calculated in the driver) to communicate to the animation, or it can simply defer timing to the animation. If animations are coded to occur `Now()`, then the real-time delay between `SendMessage` calls also determines the delay between actions in the animation.

3.4 Implementation

In the original Polka implementation, each View maintains a list of AnimObjects as its primary data structure. Essentially, this list serves as a display list just as in traditional computer graphics toolkits. Each AnimObject maintains a list of Actions that will be performed on it. When an `Animate` call occurs, a designated number of new frames are generated using a two phase algorithm at each frame. In the first phase, the Action list for all AnimObjects with pending Actions is traversed, and all necessary changes to the AnimObjects' data structures are made. In the second phase, the new frame is generated by drawing each AnimObject onto an offscreen pixmap, then copying the pixmap to the drawing window. An XFlush call is made to flush the request buffer. This implementation has a few drawbacks, especially when generating clock-based animations. Most notably, the dependence on program initiated `Animate` calls and the buffering of graphics redraw requests by X presents problems for a real-time approach.

Unlike Polka, each View in Polka-RC maintains a list of Actions as its primary data structure. The AnimObject list is only used in the drawing phase, and AnimObjects no longer must reference Actions. Rather, each Action stores a pointer to its AnimObject so that the Action/AnimObject relationship is maintained. When an Action is created, the starting time, duration, and trajectory are determined. In certain cases calculation

is necessary since some parameters are not defined absolutely; for example, the starting time can be defined with START_AFTER_START_OF, or a velocity can be specified instead of a duration. Instead of maintaining complex relationships such as pointers to other Actions, all of these calculations are done at the time of Action creation. Every Action also stores a value indicating the percentage of the Action completed so far (initially 0).

After an Action is created, it is scheduled in a View. This involves inserting the Action into the View's list of Actions, which is being sorted by the starting time of the Action. To find the correct position, the Action list is traversed from the latest to the earliest scheduled Action until the correct place for the new Action is found. The reason for a backwards traversal is in the belief that Actions are scheduled in the order they occur (for the most part). The correct insertion point is found more rapidly by traversing the list backwards. This is important because any type of calculation or operation like this in the animation process robs time from the image redisplay loop. Minimizing computation thus enhances the smoothness of the animation.

Rather than requiring regular `Animate` calls to generate frames, Polka-RC uses a timeout-based approach. Usually, Polka-RC loops between checking for X events like button presses and checking the pipe for animation events from the driver program. An X Toolkit timeout signals the need to update the display. An important consideration is the interval of time between timeouts. If the interval is too small, then frames are generated more rapidly than needed and the load on the machine will be unnecessarily high. If the interval is too large, changes in AnimObject appearances will be noticeable to the observer. Through trial and error, we determined a good interval to be 25 milliseconds on a Sun SPARCstation 2.

Even though timeouts are set at regular intervals, they do not always occur when the animation program is at an appropriate place. Timeouts can only be processed when polling the X event queue. For instance, a timeout can logically occur while a function of the animation is executing. In this case, the timeout is held pending until the next query of the X event queue. Thus, we must always determine the actual time using the Unix call `ftime` rather than using the regular data value of the timeout. The retrieved time is sent to each of the Views, which then update their displays. Since it takes time to update and redraw the Views, they will be slightly out-of-date by the time the new AnimObject configuration is shown. Efforts could be made to estimate the delay and adjust accordingly, but delay spikes (times when the system is busy and delay is high) can cause the delay to be overestimated (Hudson & Stasko 1993). When the next timeout occurs, it may be earlier than the last estimated update time, which would result in the AnimObjects moving backwards! Since the out-of-date time is very small, we decided it was acceptable.

As in the original Polka, there are two phases in updating a View. First, the AnimObjects must be updated. In Polka-RC, the update function traverses the View's Action list and sends update messages to each Action until it reaches an Action with beginning time after the current time. Since the list is ordered by starting time, all Actions later in the list must have starting times in the future as well, so the list does not need to be traversed any further.

Using the current time, each Action will compute the appearance and position for its AnimObject. For discrete Actions like COLOR, if the scheduled change time has passed, the change is straightforward and immediate. For continuous actions like MOVE, the AnimObject must be updated according to the Action's Trajectory. To effect this change, the Action determines the new appearance and the old appearance, calculates their difference, and instructs the AnimObject to change its appearance according to this delta. The first step in this process is to calculate the percentage of time that has elapsed since the start of the Action. This value is passed to the pacing function, which returns a modified percentage that reflects the desired pacing value. Next, the old and new appearances are calculated using the motion type, displacement, and percentage of elapsed time (the newly calculated percentage for the new, and the old stored percentage for the old). The difference is calculated and the AnimObjects are updated if necessary.

In the next phase of the View update, the new animation frame is generated in the same way as in Polka with one important exception. The XFlush call flushes the request buffer, but the frames may be buffered before they are displayed. The delay caused by this buffering disturbs the synchronization that Polka-RC tries to establish. Instead of XFlush, Polka-RC uses the XSync call to insure that the graphics requests are processed and the new frame is displayed immediately.

4 Discussion

In building software visualizations with Polka-RC and its real-time, asynchronous animation model, we discovered a number of interesting advantages as well as challenges in this approach. First, we'll discuss the advantages of this model over the more conventional approaches of prior systems.

The foremost advantage of Polka-RC is the exactness and clarity in scheduling the start and duration of animations. This is particularly true in a model such as Polka's that emphasizes smooth, continuous animations. For example, when two rectangles exchange positions in a sorting algorithm animation, we can now specify that this should take 2.5 seconds. No guesswork about the graphics speed and frame rate of the machine being used is necessary. Basically, the capability of real clock time specifications provides the animation

designer with a much more detailed and sophisticated set of capabilities and tools.

A second advantage of Polka-RC's model is more subtle, yet just as useful. As many developers worked with the original Polka library, they requested that more user interface capabilities be added to it. For instance, designers requested that the viewer be able to use the mouse to select View coordinates which then subsequently get utilized in the animation. Similarly, designers requested that selection callbacks be associated with AnimObjects so that a mouse click on an object would invoke a particular piece of code. Both these capabilities were subsequently added to Polka.

The difficulty in such user interface operations in the original Polka stemmed from their interaction with the animation cycle (frame generation). When polling the user for input, the Polka animation cycle cannot operate simply because the system blocks on input. Even when Polka animation code (but not the `Animate` call) is executing, no animation can occur.

Other examples of this problem also exist. Whenever a View was panned or zoomed, whether by the end-user or programmatically, a fundamental question arose of whether new animation frames were generated. In fact, Polka includes a somewhat inelegant programmatic View panning operation that synchronizes properly with the normal animation cycle.

Generally, these problems characterize the mismatch between a frame generated animation approach and one that also attempts to provide rudimentary user interface capabilities. The new Polka-RC model simply makes these problems go away.

In addition to exhibiting the benefits mentioned above, building animations with Polka-RC also uncovered some challenges or "unexpected issues." A first question we faced was whether to continue to support a relative speed control bar, and if so, how should it work? We introduced real clock time animations so that an animation designer would be able to specify precise animations independent of the hardware and software environment of the animation. However, viewers of an animation may wish to view particular sequences of the animation more slowly or quickly. After all, not every viewer has the same rate of understanding, and not every viewer will be interested in the same aspects of the animation. We felt that it was necessary to include the relative speed bar for the benefit of the animation viewer.

Following the decision to include a speed bar, the next question was how to implement it. The solution we used was to treat the speed bar like the accelerator in an automobile. When the value of the speed bar is increased, AnimObjects appear to move faster, and when the value is decreased, AnimObjects move more slowly. We accomplished this by maintaining two variables, the real clock time and the simulated Polka-RC time which starts at 0 when the animation begins. The real clock time is the value returned by the Unix system call, and the Polka-RC time is the time at which the animation actions are based. When the new time is generated with a system call, the difference between the new and old times are noted. This difference is adjusted according to the speed bar value and added to the Polka-RC time. Thus, if one second passes in real time and the speed bar is set to double the speed, two seconds would be added to the Polka-RC time, effectively doubling the speed of the animation.

A second challenge arose in the design and implementation of animation code specifying the start of Actions under varying circumstances. Consider the following example. Suppose a designer seeks to visualize a computer program that reads input and carries out an operation. The animation response to this operation should be to create a circle whose radius is scaled to the input value and then moved smoothly across the view.

In the synchronous, frame-based model of the original Polka, the Action for this operation would be

```
Action a("MOVE", fromloc, toloc, 20);
len = circ->Program(curtime, &a);
curtime = Animate(curtime, len);
```

This simply creates and schedules a movement between the positions `fromloc` and `toloc` of 20 frames that commences at frame number `curtime`, which is kept to be the current frame counter. It then animates for 20 frames and updates `curtime` appropriately.

We wrote the "corresponding" animation code in Polka-RC and it looked like

```
Action a("MOVE", circ,
      Traj(STRAIGHT, fromloc,toloc, uniform),
      START_AT, Now(), DURATION, Sec(3.0));
Schedule(&a);
```

This schedules a movement Action to commence "now" and take 3 seconds. The code works fine when the driver program fetching input values is run interactively and animations occur immediately in response to events from the driver program. Whenever the user enters a value, the appropriate event is passed through to the animation program and the movement is scheduled and generated immediately. Even if the user waits 30 seconds before entering a new value, this code works properly.

Now suppose the same driver program reads input from a file as opposed to fetching interactive replies from a terminal. Clearly, the program can read hundreds of values and call the animation program with these events in only a second or two. If we keep the same code using `Now()`, we will see a multitude of values (circles) all moving together. This is not what the designer wanted. The designer was actually hoping for a synchronous model where each new movement animation follows the conclusion of the prior one. So, to achieve this behavior, they may rewrite the code in the following way:

```
a = new Action("MOVE", circ,
        Traj(STRAIGHT, fromloc,toloc, uniform),
        START_AFTER_END_OF, b, Sec(0.0),
        DURATION, Sec(3.0));
Schedule(a);
b = a;
```

In this code, we save each old Action and make the new one start right after the old one finishes. Of course, we must be careful about the very first Action, but basically this code addresses the problem above.

Now return to our example where the program is run interactively. Let's say that the user enters one value, waits for 10-15 seconds, and enters another value. In this case, the code above that uses the START_AFTER_END_OF technique will be problematic. Making a new Action start immediately after the end of the prior Action effectively schedules the new Action in the past. In Polka-RC, the system would encounter this Action and immediately place the AnimObject at its resultant target position because both the scheduled beginning and ending times of the Action are in the past. So, we see that neither of these coding approaches works in all cases. What is needed is an Action that commences at either the current time or at a time after all other pending scheduled Actions have completed, whichever is later. This approach would solve both problem scenarios discussed earlier.

In Polka-RC we include the special value `ASAP()` to provide such functionality. When an Action with a start time of ASAP is supplied, Polka-RC internally checks the pending Actions and resolves this time as needed, either to be the current time or the earliest time after all pending Action have completed. We have found many such uses of the ASAP functionality, and it has turned out to be a commonly utilized operation. The resultant code for this final solution is shown below.

```
Action a("MOVE", circ,
      Traj(STRAIGHT, fromloc,toloc, uniform),
      START_AT, ASAP(), DURATION, Sec(3.0));
Schedule(&a);
```

5 Conclusion

We have introduced a real time based animation model for developing symbolic visualizations and simulations. This model has been implemented in a toolkit called Polka-RC which is particularly well-suited for building software visualizations and algorithm animations. Polka-RC relieves software visualizers from using low level graphics libraries, and it allows them to specify precise animation commencements and durations. We have discussed a number of the advantages of this real clock approach and a few challenging issues raised by our implementation of the system. We hope that this model and our experiences will be useful for the developers of evolving future frameworks for software visualization.

Acknowledgments

Partial support for this project was provided by Mitsubishi Research Labs in the form of a graduate research assistantship.

References

Brown, M. H. (1988*a*), 'Exploring algorithms using Balsa-II', *Computer* **21**(5), 14–36.

Brown, M. H. (1988*b*), Perspectives on algorithm animation, *in* 'Proceedings of the ACM SIGCHI '88 Conference on Human Factors in Computing Systems', Washington D.C., pp. 33–38.

Brown, M. H. & Hershberger, J. (1992), 'Color and sound in algorithm animation', *Computer* **25**(12), 52–63.

Brown, M. H. & Sedgewick, R. (1985), 'Techniques for algorithm animation', *IEEE Software* **2**(1), 28–39.

DeTreville, J. (1993), The GraphVBT interface for programming algorithm animations, *in* 'Proceedings of the 1993 IEEE Symposium on Visual Languages', Bergen, Norway, pp. 26–31.

Duisberg, R. A. (1986), Animated graphical interfaces using temporal constraints, *in* 'Proceedings of the ACM SIGCHI '86 Conference on Human Factors in Computing Systems', Boston, MA, pp. 131–136.

Hudson, S. E. & Stasko, J. T. (1993), Animation support in a user interface toolkit: Flexible, robust and reusable abstractions, *in* 'Proceedings of the 1993 ACM Symposium on User Interface Software and Technology', Atlanta, GA, pp. 57–67.

Kimelman, D., Rosenburg, B. & Roth, T. (1994), Strata-Various: Multi-layer visualization of dynamics in software system behavior, *in* 'Proceedings of the IEEE Visualization '94 Conference', Washington, D.C., pp. 172–178.

Lassiter, J. (1987), Principles of traditional animation applied to 3d concurrent animation, *in* 'Proceedings of SIGGRAPH '87', pp. 35–44.

Mukherjea, S. & Stasko, J. T. (1994), 'Toward visual debugging: Integrating algorithm animation capabilities within a source level debugger', *ACM Transactions on Computer-Human Interaction* **1**(3), 215–244.

Price, B. A., Baecker, R. M. & Small, I. S. (1993), 'A principled taxonomy of software visualization', *Journal of Visual Languages and Computing* **4**(3), 211–266.

Reiss, S. P. (1985), 'Pecan: Program development systems that support multiple views', *IEEE Transactions on Software Engineering* **SE-11**(3), 276–285.

Shimomura, T. & Isoda, S. (1991), 'Linked-list visualization for debugging', *IEEE Software* **8**(3), 44–51.

Stasko, J. T. (1990), 'TANGO: A framework and system for algorithm animation', *Computer* **23**(9), 27–39.

Stasko, J. T. & Kraemer, E. (1993), 'A methodology for building application-specific visualizations of parallel programs', *Journal of Parallel and Distributed Computing* **18**(2), 258–264.

Stasko, J. T. & Patterson, C. (1992), Understanding and characterizing software visualization systems, *in* 'Proceedings of the 1992 IEEE Workshop on Visual Languages', Seattle, WA, pp. 3–10.

Stasko, J. T. & Wehrli, J. F. (1993), Three-dimensional computation visualization, *in* 'Proceedings of the 1993 IEEE Symposium on Visual Languages', Bergen, Norway, pp. 100–107.

Biographical Note

John T. Stasko is an Associate Professor in the Graphics, Visualization and Usability Center and the College of Computing at the Georgia Institute of Technology. His research interests include software visualization, human-computer interaction, programming environments, and parallel programming.

D. Scott McCrickard is a PhD candidate in the Graphics, Visualization and Usability Center and the College of Computing at the Georgia Institute of Technology. His interests include information visualization and user interfaces.