Creating Animation for Presentations

Douglas Zongker

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

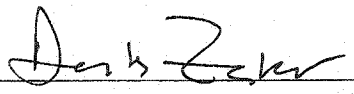
University of Washington

2003

Program Authorized to Offer Degree: Computer Science & Engineering

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature 

Date 9 June 2003

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Douglas Zongker

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Chair of Supervisory Committee:

David H. Salesin

Reading Committee:

Maneesh Agrawala

Brian Curless

David H. Salesin

Date: 9 June 2003

University of Washington

Abstract

Creating Animation for Presentations

by Douglas Zongker

Chair of Supervisory Committee:

Professor David H. Salesin
Computer Science & Engineering

In recent years the use of computer-generated slides to accompany live presentation has become increasingly common. There is a potential for using computer graphics to increase the effectiveness of this type of presentation. The hardware for generating and projecting complex scenes and animation is in place, yet few efforts have been made in creating software to fully utilize these capabilities.

This dissertation presents a system, called SLITHY, for giving live presentations accompanied by rich animated slides. This system has been developed and refined through several iterations of creating animated talks, then trying to simplify the authoring process by improving the system itself. We show some examples of presentations we've created with the system, as well as some from outside users who have used our system to present their own work.

As we repeated the design cycle, we were also learning what types of animation were best suited for use in presentation. In this work we summarize our accumulated experience as a set of presentation animation principles. It is certainly possible to use animation poorly, but we believe that having a set of broadly applicable guidelines can assist presentation authors in applying these new tools in ways that enhance rather than detract from their subject matter.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I've worked on a number of projects during my time at the University of Washington. Dissatisfaction with the tools we had to present these other assorted pieces of work was what led to the creation of SLITHY. I feel privileged to have worked with so many fantastic collaborators on these other projects, including Michael Wong (floral ornament); Yung-Yu Chuang, Brian Curless, Rick Szeliski, Joel Hindorff, and Dawn Werner (environment matting); Geraldine Wade (font hinting); and George Hart (blending polyhedra). In addition, I must thank some early collaborators on SLITHY itself: John Hughes, Tomer Moscovich, and Andy van Dam. Maneesh Agrawala, Michael Cohen, and Marc Levoy provided valuable insights on turning our prototype system into a bona fide research project.

Of course, the person I've worked with most closely at UW has been my incredible advisor, David Salesin. He has been a great teacher not only in the technical aspects of computer graphics, but also in the broader skills of doing research, especially writing and presenting.

The computer graphics lab at UW has been a great group to be a part of. I'm especially grateful to Brett Allen, Steve Capell, Yung-Yu Chuang, and Karen Liu for risking embarrassment in front of hundreds of SIGGRAPH attendees by using an untested research presentation system for real conference talks. Their experiences and feedback have been invaluable in improving the system (not to mention producing several of the figures in Chapters 4 and 5).

Though I never managed to coauthor a paper with either of them, my friends and fellow students

Craig Kaplan and A.J. Bernheim contributed to both this work and my graduate career in ways I can't begin to enumerate.

It's hard to imagine a more fun, exciting, and inspiring place in which to get a PhD than the UW CSE department. I owe a great debt to all the faculty, staff, and students who've made this happen over the years. Any list of names I give would probably be incomplete, so I won't try. Thanks to you all.

Lastly I want to thank my family – my parents Earl and Chris and my sister Deirdre – for the love, encouragement, and support that they've given me for the past 27 years. I'd never have done any of it without them.

Chapter 1

INTRODUCTION

As researchers and educators, we give a lot of presentations, and we are part of the audience for even more. Our own lives would be improved if we could give – and receive – better talks. There are many ways in which one can imagine giving any particular talk, but being in the field of computer graphics, we naturally think of improving the *visuals* that almost invariably accompany a modern presentation.

Despite the fact that communication tools, including word processing, email, the Web, etc., have been some of the most successful and widely adopted applications for computers, little work has been done on making effective use of graphics technology for live presentation. Today's presentation software – of which Microsoft's PowerPoint, in representing the vast majority of the presentation-software market, is the most prominent example by far – is still rooted firmly in the past. Presentations today are being delivered directly from the computer, thanks to the increasingly ubiquitous data projector, but the software is still geared toward producing 35mm slides and overhead transparencies. Even the term "slides" reflects this way of thinking: one page after another of static information. Slowly, animation features are being added to the software, but even these are mostly limited to simple effects used to draw attention to an otherwise lifeless slide.

The hardware in use today is powerful enough to produce complex animated graphics to explain and enlighten, rather than just compete with the speaker for the audience's attention. In this work we describe SLITHY, a system for producing presentations with real, content-rich animations. Our approach to this problem has been iterative: we began by trying to make talks that incorporated animation and interactivity using existing software tools. These attempts led to a wish list of effects

we wanted to achieve and ways we wished the authoring worked. We began implementing and using our own system, alternately creating talks and improving the system itself. We will recount the history of this effort to motivate some of the design decisions made in SLITHY's development.

As we made more and more of these animated talks, we were also learning how best to apply the power of animation in presenting material. We tried to understand why some uses of animation seemed to make information clearer, while others appeared to be simply gratuitous and distracting. For example (and to our own surprise), we found that many of the principles of classical character animation [32] do not work so well for presentations. In this work we will summarize our observations as a set of principles for presentation animation. We believe these principles are broadly applicable across a wide range of presentation subjects, and can provide valuable guidance in using animation effectively.

While our own experiences with giving animated presentations have been very positive, it would be nice to be able to show that presentations with animation are better than equivalent static presentations in some objective way. This is a difficult question: to answer it would require precise definitions of what is meant by *"better"* and *"equivalent."* There is a great deal of research in the educational psychology field on this subject, where *better* is determined by testing the learners' recall and/or problem-solving abilities on the subject matter. Even if this narrow definition of *better* is accepted, the conclusions of these studies – while tending to fall in favor of animation – are still in dispute due to disagreements by researchers over what really constitutes an *equivalent* presentation.

By some estimates, at least thirty million PowerPoint presentations are made every day [46]. Presentation technology is having an impact on the way millions of people communicate. This is not a small problem, and we do not claim to have an ideal solution. Our work in this area has led us to three conclusions: first, that animation can often make for clearer, more engaging presentations. Second, that presentation animation is sufficiently different from character animation that their effectiveness is governed by a different set of principles. Lastly, we believe that a scripting interface is well-suited for the demands of presentation animation. Although we recognize that this style of authoring is not for everyone, we feel that the problem of creating better presentations is

important enough and hard enough that even a solution that serves only the needs of a more limited community is a worthwhile step. While none of these statements can really be quantitatively proved or disproved, it is the intent of this thesis to argue in support of all three.

Overview

The next chapter will recount some of the history of SLITHY, from our first pre-SLITHY animated presentations to the present system, in order to explain and justify some of the design decisions we made. Chapter 3 is a tutorial-style introduction to the system as it stands today. We show how SLITHY can be used to construct parameterized diagrams, animations, and interactive objects. In Chapter 4 we describe and demonstrate our set of principles for making good use of animation in a presentation. Chapter 5 contains samples of more complete examples of animated presentations, including some put together by other users for presenting their own work. Related work is addressed in Chapter 6: we look at psychological research on the effectiveness of animated instruction, as well as comparing SLITHY to existing software systems for animation and presentation. In the final chapter we present conclusions and offer some possibilities for future work.

Chapter 2

EVOLUTION OF AN ANIMATED PRESENTATION SYSTEM

The design of SLITHY [1] was motivated by our own experiences authoring and giving presentations. In this chapter we recount these experiences and use them to justify the various design choices made in building SLITHY.

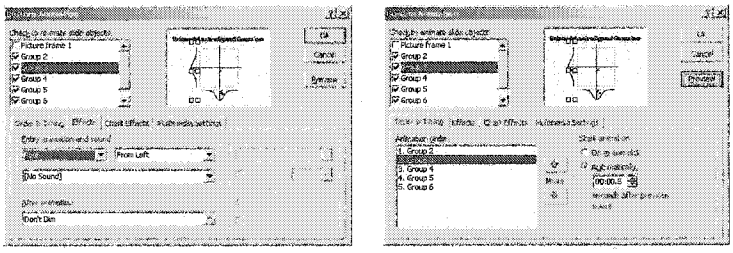
2.1 PowerPoint

The canonical software package for creating presentations is PowerPoint [37], now part of the Office suite from Microsoft. This product predates the use of electronic projectors; it began not as a live presentation tool but as a kind of specialized word processor. It was originally designed for creating static documents to be output to overhead transparencies or 35mm slides. As projector technology advanced, it became more common to use its "slide show" features to run the presentation directly from the computer. Today this method is the most common mode of showing PowerPoint presentations in business and academia.

PowerPoint's animation features started as a way to make the transitions between slides more eye-catching. There was a fixed palette of effects from which authors could select transitions. Effects could also be applied to individual elements (text, images, drawn shapes) on a slide. Each effect could be set to start either a fixed time after the previous effect, or on a mouse click or keypress. Each effect caused an element to enter or exit the slide in some animated way.

Many PowerPoint users developed a collection of tricks for using these simple entry/exit effects to simulate motion. To "move" an element from one place on the slide to another, for instance, it was common to slide on a rectangle matching the background to cover up the element, then add a

[1] Pronounced "sly–thee." The word is a compound of "slimy" and "lithe," taken from the poem *Jabberwocky*. The author, Lewis Carroll, interprets the word as "smooth and active." [10]



(a) (b)

Figure 2.1 *Screenshots showing PowerPoint 2000's animation interface. The "Effects" tab (part (a)) is used to select which animated effect is used for a graphical object. The "Order & Timing" tab (part (b)) sets the order in which objects make their animated entrances.*

copy of the element to the slide, sliding it from offscreen to the new position. It didn't really look like the element was moving between locations on the slide, but it at least gave the impression of movement.[2] The only other way of incorporating animation into a PowerPoint presentation was to create it in some other program, save it as an MPEG or AVI movie, and drop it into the presentation. We had created a number of presentations using one or both of these types of animation, including presentations for papers at SIGGRAPH 99 and SIGGRAPH 2000 [65, 14].

Neither method of incorporating animation was entirely satisfactory. PowerPoint's animated effects were too limited—most of the available effects were too flashy to be useful, and if the desired animation couldn't be constructed out of the small set of useful effects there was nothing the user could do. Animations could not modify any of the shapes or text on the screen, only make the whole element appear or disappear. In addition, the user interface consisted of a single cramped dialog box (see Figure 2.1) that made it difficult to manage complex slides with many animated elements.[3]

To create video files for use within PowerPoint, we would typically write small, single-use OpenGL-based programs to draw and save each frame. These programs were written in a mixture of C and Tcl—C to do the actual OpenGL drawing, Tcl to provide a simple user interface that let us interactively manipulate a few parameters (most commonly the position of the camera, so that the

---

[2]The latest version of PowerPoint [38], which was released when we were well in to developing SLITHY, has added the ability to move elements on the slide. Figure 6.1 illustrates this new "motion paths" feature.

[3]Newer versions of PowerPoint, which we discuss in the chapter on related work, have improved somewhat both the variety and usability of animation, but this was the state of the system in 2000 when our work began.

animation could be centered in the frame visually). These programs would render a series of still images, saving each frame to disk. We would then use a video encoding package to assemble the frames into a video clip.

While this approach gave us the power to display arbitrary animations in our presentations, it was very cumbersome—editing the animation required changing the program, dumping out a new set of frames, assembling them into a compressed video, and replacing the video in the presentation document. Obtaining good quality video was also a challenge. Since video codecs were designed with live action, using them to compress line-drawing-style animated illustrations tended to show artifacts or require enormous file sizes. Full-screen high-resolution video wasn't feasible with the technology of the time, so the animated section of the slide tended to be constrained to a small section of the screen. A typical use of video was to show an animated transition from a diagram on one slide to a different diagram on the next, but aligning the video with PowerPoint-drawn elements was difficult and the effect was never very seamless. Overall, the time- and resource-intensive nature of video restricted us to using this type of animation sparingly.

PowerPoint's built-in animation was integrated with the rest of the presentation, but it was limited in its capabilities. Video gave us much more power in what we could show, but separated authoring the animation from authoring the rest of the presentation and made integration difficult. Eventually we would seek a solution that incorporated the best aspects of both.

2.2 The hinting talk

Initially, we were not pursuing the idea of improving presentation tools as a research project. We were simply trying to give good presentations. The lack of animation features was sometimes irritating, but we made do with what we had. The first truly animated talk we produced was really something of an accident.

A paper of ours on hinting for digital typography had been accepted to SIGGRAPH 2000.[4] The

[4] *Hinting* is a procedure whereby outline fonts are augmented with extra information that improves rasterization at small sizes. For more information, see Zongker *et al.* [64].

SIGGRAPH conference proceedings allowed authors to include four minutes of video in addition to the printed paper, and since our paper was about typography, we didn't need the video to illustrate our results. We decided to use our allotted four minutes to make a short tutorial-style introduction to the problem of hinting, in order to motivate our work.

To produce this video we used the same approach we had used in creating videos for use in PowerPoint—a small program, written in C and Tcl, that drew frames using OpenGL and saved them to disk. While we were working on this video, we had to give a short presentation about current work to some visitors to the department. We didn't yet have any slides prepared for the hinting work, but we did have this short introductory video. It wasn't complete – it was missing a voiceover, for one thing – but we decided to use it anyway. We dumped the animation onto videotape, plugged a camcorder into the projector, and gave the presentation, using the pause button on the camcorder to start and stop the video as we talked over it.

This ad hoc presentation worked extremely well. Even though our topic was not inherently dynamic – typography is, after all, largely concerned with things printed on paper – we found that animation was a more natural way to express concepts than static images would have been. Immediately after this short talk, we began thinking about how to do our entire SIGGRAPH talk in the same style.

The first option we considered was videotape, but videotape has a number of shortcomings. The picture quality is fairly low, especially for things like line drawings, and the picture is degraded further when the tape is paused. Second, using the pause control to move forward is awkward—the presenter has to press buttons to both start and stop each piece of animation, making it difficult to focus on speaking. The pause control is also not very accurate; it is hard to stop the tape at precisely the right point. Lastly, the authoring process is complicated by the fact that the medium is linear, so making any change requires re-recording the entire tape (at least, from that point onward). While these problems had not prevented us from giving a short, informal talk using videotape, we felt that for a full, formal presentation in front of a large audience, videotape would be too cumbersome.

The alternative was to use software to render the talk live. Because the diagrams we were using

were simple, the program we had to generate frames was almost fast enough to run in real time. Our experience giving the talk from videotape convinced us that these simple graphics were enough, and were perhaps even more clear than a more fancily rendered representation would have been.

We optimized the program for speed, and added the necessary features to use it as a live presentation tool (full-screen display, keyboard navigation, etc.) We also extended it to produce not just the four minutes of video originally planned, but an entire twenty-minute presentation. The result, a program called `hinttalk`, was successfully used to deliver a talk at the SIGGRAPH conference.

Of course, creating several minutes of animation took a great deal of effort. Very little of the resulting code is reusable, most of it deals specifically with drawing figures related to hinting. The system's overall architecture, though, is worth discussing as it shaped the structure of SLITHY.

The hinting talk is divided into 17 sections. For each section there is a single C function that does all the drawing for that section. Here's a piece of the first section's drawing function:

```
int Redraw1Cmd( . . . )
{
    double xpos, ypos, scale, pix, alpha_mult;

    . . .

    xpos = double_option( interp, "xpos" );
    ypos = double_option( interp, "ypos" );
    scale = double_option( interp, "scale" );

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();

    draw_background( interp );

    glTranslatef( window_width/2.0-xpos*scale,
                  window_height/2.0-ypos*scale,
                  0.0 );
    glScalef( scale, scale, 1.0 );

    glPushMatrix();
    glScalef( 0.18, 0.18, 1.0 );
    glTranslatef( 100, 150, 0 );

    alpha_mult = double_option( interp, "fill0" );
    if ( alpha_mult > 0.0 )
    {
```

```
    color_option( interp, "interior0" );
    draw_outline( Book_a, M_SOLID );
}

. . .

pix = double_option( interp, "pixels0" );
if ( pix > 0.0 )
{
    glPushMatrix();
    glScalef( 2048.0/90.0, 2048.0/90.0, 1.0 );
    glColor3d( 0.0, 0.0, 0.0 );
    draw_pixlist( Book_a_big, pix, S_BOX );
    glPopMatrix();
}

. . .
```

(Note that some setup code present in every function has been elided with "...".)

There are a few things to note about this function. First, the drawing is controlled by a number of "options." (Note the calls to `double_option()` and `color_option()` to access the value of these options.) The values of these options are set in the Tcl code that invokes this drawing function. By exposing as many controls as possible to the Tcl code via this option mechanism, we made it possible to change the appearance of the figure without recompiling the C function. Frequently these options are used to specify graphical parameters like color and opacity of drawn elements and position of the virtual camera, but other options are more abstractly related to how the scene is drawn (such as the `pixels0` option above, which controls whether a section of code is executed or not. Option values aren't always used verbatim in drawing—sometimes they are inputs to computations (as in the call to `glTranslatef()`.)

Another thing to note is that not all of the low-level drawing commands are in this one function. The code makes extensive use of user-defined helper functions (such as `draw_pixlist()`) for portions of the drawing that are used repeatedly. These helper functions are shared between the different section drawing functions.

Not all the data for drawing comes from the options or from the code itself. Objects like `Book_a` – which contains a representation of the outline of one character from a font – are loaded from

files when the presentation begins. This means that the presentation can be edited in some ways without changing the drawing function. To change the character drawn by this function, we need only change the underlying data files referenced by the drawing function.

The flexibility that these capabilities provide proved very useful in authoring and editing the presentation, though we didn't fully recognize this at the time. Our first attempt at a general tool for building animated presentations was missing some of these features, but eventually we sought to incorporate them into SLITHY.

For the hinting talk, the C drawing functions were driven by animation scripts written in Tcl. Here's a small piece of the script used to drive the previous function:

```
proc script1 {} {
    set g_options(pixels0) 0.0
    set g_options(fill0) 0.0

    set g_options(xpos) -856.560
    set g_options(ypos) -433.025
    set g_options(scale) 15.000

    . . .

    linear_transition 15 {partial0_one 0.0 1.0} {decasteljau 0.0 1.0}
    after 500

    # draw first bezier segment
    linear_transition 90 {umax0_one 0.0 1.0}
    after 100

    # pause
    singleframe

    # draw segment without decastaljau construction
    linear_transition 20 {decasteljau 1.0 0.0}
    set g_options(partial0) 1.0

    . . .
```

Some details have been removed for clarity, but the essential parts are shown. The script starts by setting initial values for all of the options. (Note that the drawing function actually has more options than are referenced in the short excerpt given above.) The duration of animation commands such as `linear_transition` are specified in terms of frames, assuming 30 frames per second

(remember that this application was originally created for offline rendering of animation frames to be assembled into video). The first command linearly interpolates the parameters `partial0_one` and `decasteljau` from 0.0 to 1.0 over 15 frames. Option values can be set instantaneously with the `set` command. The `after` command is used to insert a pause (specified in milliseconds) and the `singleframe` command makes the program pause, waiting for the user to press the spacebar before continuing.

This script is actually being executed as the animation is playing. For instance, when the `linear_transition` function is called, the program goes off and draws the specified number of frames on the screen before returning control to the script and proceeding to the next command. We'll call this property *live execution*. Live execution of the script complicated the development of animation scripts for a number of reasons.

First, errors in the script could cause the program to exit halfway through running an animation. Because Tcl is an interpreted language, even simple syntax errors were not discovered until the interpreter tried to execute that line of code. Having to run all the way through an animation to find each error of this kind was slow and often frustrating.

Second, even when an animation script was error-free, there was no way to back up while playing an animation. To do so would require backing up the state of the partially executed animation script—a difficult if not impossible proposition. The only recourse was to start a script over from the beginning, which meant backing up to the start of the section.

Specifying overlapping actions was also awkward with this live execution model. The special case where the actions both start and end simultaneously and use the same interpolation style was straightforward, as illustrated by the first call to `linear_transition` above. It changes the values of both `partial0_one` and `decasteljau` in parallel. If the second action was to start midway through the first, however, the code became substantially trickier:

```
set plan [plan_linear_transition 20 {outline1 0.0 1.0}]
set plan [execute_plan $plan 5]
set plan [merge_plans $plan [plan_linear_transition 20 {grid1 0.0 1.0}]]
execute_plan $plan
```

Translated into English, this code works like this:

1. Create a "plan" for the first action (changing the `outline1` option from 0.0 to 1.0 over 20 frames). Nothing is drawn, a plan is simply a list of option-value pairs for each upcoming frame.

2. Execute the first 5 frames of the plan.

3. Create a plan for the second action (changing the `grid1` parameter), and merge it with the remainder of the first plan.

4. Execute the merged plan.

This "plan" mechanism was the first way we separated *specifying* an animation from *running* it. With the live execution model, a single animation command does both. The ability to specify animations before running them, however, often makes it easier to describe a complex set of overlapping actions. Animation events do not have to be specified in the order that they actually occur. An author can describe a complex series of actions to affect one part of the picture, then "back up" in time to explain a second set of actions that run in parallel with the first. With live execution each frame must be completely specified (and displayed) before the next.

2.3 A general system

After seeing the warm reception our animated hinting talk received, we began to think about how to make it easier to produce this kind of presentation. Our system, the main subject of this dissertation, is called SLITHY. It has been developed through an iterative process: using it to build and give presentations, then taking that experience and using it to improve the system itself. The hinting talk described above may be thought of as the beginning of this iterative process. While the refinement of SLITHY has been a slow and gradual process, in this chapter we will focus on three major "milestones" of the system: SLITHY-0, when we first made a serious attempt to make a presentation with

a general system; SLITHY-1, the first version given to other users to try out; and SLITHY-2, the system in its current state.

In creating SLITHY-0, we focused on addressing four signficant problems that we encountered in developing the hinting video software into a live presentation:

The first problem was that frame-based timing was hard to work with. Having each animation command's duration be expressed in terms of an integral number of frames made it hard to make small tweaks to the timing. Speeding up an animation by 10%, for instance, couldn't be done by simply applying a global modification to the whole animation; instead we had to go through the animation script and change every frame count. In our new system, all durations would be expressed with floating-point values. These values nominally represented seconds, but the animation could be scaled continuously by simply changing the scaling factor between animation time and wall-clock time.

The second problem was the use of the live execution model discussed previously, where the user's animation script is being executed as the animation plays. The hinting talk mixed some commands which used live execution with others that used the "plan" mechanism. In SLITHY-0, we got rid of the live execution model entirely. The entire animation script is effectively creating a single plan that represents the whole animation. This planning model is a significant advantage. Parallelism and overlapping actions can be expressed quite naturally. When planning is used exclusively, rather than being mixed with live execution, much of the extra syntax required in the previous example can be hidden from the user. The next sample script will illustrate this "light" planning syntax.

Another advantage of a plan-based animation system is that the user's script is executed completely before the presentation begins, so errors can be caught more quickly. Since the playback system is simply evaluating a plan, rather than running arbitrary user code, it can effect meaningful changes to the planned animation. It can speed up or slow down the playback, run it backwards, skip around arbitrarily, or play the animation multiple times, without requiring the script author to explicitly support any of these capabilities. The script is executed just once to build an inter-

nal representation of the animation, then this representation is handed to the runtime system to be displayed.

Our third problem was that our graphics primitives were too primitive. The hinting talk was based on OpenGL, which is a optimized for drawing a lot of triangles very fast but doesn't have a very rich imaging model—poor support for line drawing and curves, no support for text, etc. We wanted a system where we could build diagrams that were as clean and professional-looking as those produced for print in tools like Illustrator [1]. We looked at using alternate graphics libraries such as Java2D or SVG (Scalable Vector Graphics), but rejected these because of poor performance. Ultimately we went with a compromise and implemented our own set of higher-level primitives atop OpenGL. The resulting library is not quite as complete as many of the third-party libraries we considered, but is sufficiently high-performance to be used in animation on today's machines.

The fourth problem had to do with our choice of implementation langugage. This was an important consideration since we had chosen a scripting model for authoring, so creating a presentation in the system meant writing code in some language. Neither C nor Tcl, which we'd used together for the hinting talk, was entirely satisfactory for this purpose. Writing diagrams in C meant that with each change the program had to be recompiled, which slowed down the development cycle. This extra step was especially galling when the changes were minor, such as repeatedly adjusting the coordinates of an element to position it on the screen. We also wanted the system to be as accessible as possible to people without programming experience, and C does not meet that requirement.

At the same time, doing everything in Tcl was unappealing as well. The original designer of Tcl admits [42] that the language was designed as a "glue" language, for easily hooking together larger pieces of C code. It was not originally intended for writing large applications natively. While recent releases of the language have attempted to improve Tcl's shortcomings in that regard, the need for consistency and backward compatibility have kept the language somewhat hobbled. Nearly everything in the language – conditionals, loops, and even arithmetic – is done with a function call, and the result is a system that is flexible and extensible but somewhat slow, and frustrating for many users of traditional programming languages.

We briefly considered creating our own customized language for building animations, but ultimately decided to use an existing programming language, Python [25], instead. Choosing a general-purpose language over creating our own language has the advantage that a complete set of programming constructs is available within the scripts for free. Script authors can use variables, create complex data structures, perform computation, do I/O, etc., without our having to create a parser, compiler, or interpreter sufficiently powerful to support all those features. Also, having the scripts and the system itself implemented in the same language means that as we gain more experience creating these presentations, constructs which we found ourselves using over and over can be easily migrated into the base system itself for all to use.

Of course, there are disadvantages to using an existing general purpose language. The primary one is that we are locked into the syntax of whatever language we choose. A major reason for choosing Python was the relative lightness of its syntax—variables are dynamically typed so no declarations are needed, and the language is relatively free from the myriad braces and semicolons required by other alternatives. It is widely considered to be a good language for teaching introductory programming, and we wanted the system to be accessible to presentation authors who are not already programmers but are willing to learn, at least a little.

However, Python syntax is not perfect for our purpose – in particular, it has strict rules about indentation, while we would prefer a more "free-form" treatment of whitespace – but on the whole we felt Python was the best choice. Syntax was not the only reason for choosing Python. Another major factor in our decision was that it is free, widely available on both Unix and Windows platforms, and is easily extended with C. This extension facility allowed us to write the high-performance OpenGL drawing code in C and drive it from the users' Python scripts. The Python language also contains a variety of container types (lists, dictionaries, etc.) that simplify construction of complex data structures.

Believing we had come up with solutions to all the problems we encountered while creating the hinting presentation, we started to build our new, more general system. Sadly, though, one important notion from the hinting talk was lost: the idea of building graphical models with high-level param-

eters and driving those parameters from an animation timeline. In hindsight, we now recognize this parameterization as a valuable way of structuring the animation process for the author as well as for the computer, but at the time, the division of drawing into "model" and "animation" seemed like an artifact of the division between C and Tcl code. The use of "options" as intermediaries between the two we saw as an attempt to minimize the amount of data crossing the boundary between the two languages. When we moved to a single implementation language, the reasoning went, why not do the drawing and the animation together? This proved to be a mistake, but an instructive one.

In designing this monolithic system SLITHY-0 we were also strongly influenced by PowerPoint. PowerPoint let us populate a slide with elements like text, images, and simple drawing shapes, but its animation features didn't give us enough control over those elements. Our system would be different. It would have the same types of primitives as PowerPoint: text boxes and static images, as well as shapes like lines, rectangles, and ovals. The difference would be that every parameter of every element, down to the coordinates of individual vertices, would be exposed to the animation interface, allowing the user to script how its value should vary over time.

We chose scripting over a graphical interface for creating the animations. We had a number of reasons for this choice. As computer scientists, we were accustomed to using tools like LaTeX and GNU Emacs that are fundamentally based on programming languages. Language-based tools tend to have a steep learning curve, in exchange for far more power and flexibility than a typical GUI provides. With scripts it is easier to express things that are done repeatedly as resuable macros. For character animation, which has little exact repetition, this capability might not be so critical, but we felt it would be important for our domain. We expected the animations used for presentations to be simpler and more repetitive than would be found in a character-driven story. (Imagine how many times a "fade in bullet point" animation might be used in a single presentation.) Finally, we wanted to test our ideas quickly, and implementing a GUI on top of the animation system would have slowed development considerably. We believed it would be relatively easy to come back and layer a GUI on top of an existing script-based system, while doing the opposite – extending a GUI with a scripting facility – would be more difficult.

Here is how a very simple script in this system might have looked:

```
def slide1():
    # create two objects, a line and a text box
    tx = Text( text = 'Slide One', color = yellow, size = 20,
               x = 30.0, y = 80.0 )
    ln = Line( x1 = 10.0, y1 = 10.0, x2 = 40.0, y2 = 40.0,
               color = blue, thickness = 2.0 )

    # define an animation with these objects on-screen
    start_animation( [tx, ln] )

    linear( 2.0, ln.color, red )

    set( tx.text, 'New Text' )

    parallel()
    smooth( 1.0, ln.x2, 80.0 )
    smooth( 2.0, ln.y2, 60.0 )
    end()

    fade_out( 0.5, tx, ln )

    return end_animation()
```

This script is somewhat hypothetical, since we found this approach had serious shortcomings before the initial implementation was even complete, and the system underwent substantial changes. Before discussing those faults, though, let's look at the features of the system that worked well.

In contrast to the hinting talk, where the animation was produced onscreen as the function executed, in this system each animation function created and returned an *animation object,* which was then fed into the playback system to produce graphics on the screen. First, the script created some *elements* (a text box and a line, in this example) that would appear in the animation, assigning default values to each of their parameters. Then it would use them to define an animation—everything between the `start_animation()` and `end_animation()` calls defines a single animation object. An animation object is similar to a "plan" from the hinting talk: it is a declarative representation of the entire animation that can be saved and played back later.

Note that because the *entire* script is constructing a single animation object and we are no longer trying to mix the two scripting styles – live execution and plan construction – the mechanics of

constructing the objects can be largely hidden from the user. While the hinting talk required the author to explicitly call procedures like `merge_plans` and `execute_plan`, in SLITHY-0 all this was hidden away. The line "`linear( 2.0, ln.color, red )`" is superficially very similar to a call to `linear_transition` in the hinting talk. It gives the parameter[5] to operate on, the final value, and transition style and duration. The fact that it is just constructing an internal representation of the animation rather than actually going out and drawing frames on the screen is not apparent to the user.

What the script author does perceive, though, is that the new system does have much greater flexibility with respect to time. Consider the two calls to `smooth()` in the above example script. These two commands are supposed to execute simultaneously. The code needed to get that functionality – overlapping two actions of different duration – in the hinting talk was qualitatively different than the code to execute the same two actions in sequence. Roughly translated into the language of the hinting talk, the two alternatives would resemble this:

```
# execute in sequence
smooth_transition 30 {ln_x2 80.0}
smooth_transition 60 {ln_y2 60.0}

# execute in parallel
execute_plan [merge_plans [plan_smooth_transition 30 {ln_x2 80.0}]
                          [plan_smooth_transition 60 {ln_y2 60.0}]
```

In contrast, in SLITHY the behavior of `smooth()` is changed simply by bracketing the calls in calls to `parallel()` and `end()`. In this parallel mode, time is backed up after each command is issued so that other commands in the script can start at the same time. The details of the data structures behind this mechanism will be discussed in the next chapter.

SLITHY-0 represented times and durations with floating-point values, rather than integer-valued frame numbers as in the hinting talk and many earlier script-based animation systems. This allows Slithy animation objects to be sped up, slowed down, or played at continuously varying rates of speed without temporal aliasing artifacts. This choice complicates the implementation somewhat –

[5] "Option," in hinting talk terminology.

the internal structures must be represented functionally, rather than as a simple list of a parameter values for each frame – but with Python's treatment of functions as first-class objects this is straightforward to implement. In fact, for lengthy transitions, this functional representation can actually be more compact than an explicit list of values.

2.3.1 Drawbacks of SLITHY-0

While the basic animation framework of Slithy was reasonably easy to use, it turned out to be tiresome and difficult to create complex animated figures. Slithy had little support for grouping and transforming primitives together, or expressing relationships between primitives. As a concrete example, consider the pulley depicted in Figure 2.4 on page 37. Graphically it is fairly complex, made of a considerable number of lines and circles. Using our system to animate the pulley's behavior as the handle is pulled down would require a separate animation command to move each coordinate of each graphical shape. (SLITHY-0 did include commands for applying affine transforms to one or more elements in addition to being able to animate the parameter values themselves, but even so, animating the pulley would still be a tedious proposition.) As soon as we moved beyond creating the simplest diagrams this shortcoming became evident.

What was needed was a way to encode relationships between the individual lines and circles and other graphical elements. In the case of the pulley figure, in fact, there is really only one variable in the system—the distance the handle is pulled. Everything else – the position of the load, the rotations of the wheels, the lengths of the various sections of rope – is a consequence of this single abstract parameter. We wanted a way to wrap up these dependencies once, so a smaller set of more meaningful parameters could be exposed to the animation script.

One approach we considered was using some sort of constraint system to tie together the parameters of individual elements. One could express constraints like "this line segment's endpoint must be coincident with that other line segment's endpoint," "this line segment has a fixed length l," and so on. Manipulating one graphical parameter would then invoke the constraint solver to propagate changes across the system.

We chose not to use the constraint approach for a number of reasons. One was a feeling that while it seemed natural for the mechanical device illustrations we were working with early in the research, it might not be so appropriate for more abstract diagrams. We were concerned about the performance of the constraint solvers, given that our system had to run in real time. Finally we were worried that systems of constraints might be too unintuitive for novice users. It is not always clear how to translate an idea of how a system should behave into a correct set of constraints on its parts.

We opted to return to the model used in the hinting talk, where the author writes a function that takes a set of parameter values as inputs and procedurally draws a picture. In SLITHY these objects are called *parameterized diagrams.* The parameterized diagram encodes how the diagram's appearance should change in response to various control values. This behavior can be defined once and then driven from an animation script.

There is an obvious analogy here with character rigging in commercial 3D animation packages such as Maya [2]. Animations aren't produced by moving each vertex of a character independently. The first step is to create an *articulated model,* allowing the character's geometry to be manipulated by high-level controls. Then the animator typically uses keyframes and spline interpolation to set the values of those controls over time. While this two-step process is the accepted workflow for 3D animation packages, we have not seen 2D packages that work this way. (A more thorough comparison to existing work can be found in Chapter 6.)

The simple animation commands of SLITHY-0 – like `linear()` and `smooth()` – while cumbersome for manipulating graphics at the level of individual vertices, turn out to be well-suited for manipulating the abstract parameters needed for this kind of animation. Figure 2.2 shows an example. The figure shows the animation timelines used for the opening segment of the hinting talk, recreated in SLITHY-2. A single diagram with eleven parameters underlies this whole piece of animation. Even though the figure is quite complex, once it is abstracted into a parameterized diagram with an appropriate set of high-level controls, the manipulations of those controls required to produce interesting animation is fairly simple.

The programming approach, while offering near-total control over how the graphics appear and

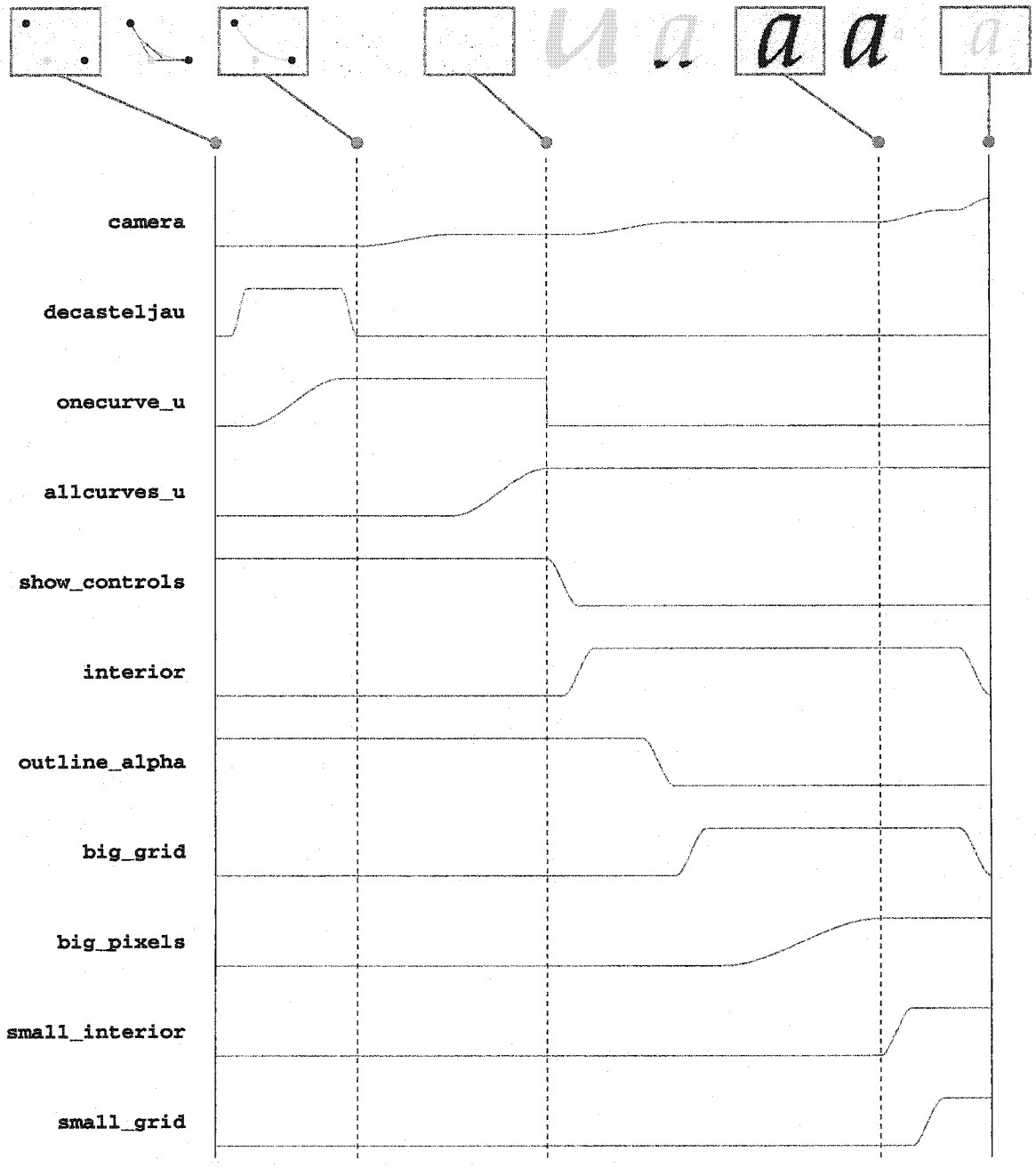


Figure 2.2 *Here we show the animation curves that control diagram parameters for part of the hinting animation. The resulting frames are shown along the top. The dashed lines indicate where the presentation pauses and waits for the user to press spacebar to continue. Notice that all parameter changes are done with a single style of interpolation from one value to the next. Because the ways in which parameters change are so simple in this type of animation, it is less important to provide an elaborate interface for precisely keyframing the parameters.*

move, is admittedly not suitable for many nontechnical users. It *is* computer programming, with both the power and complexity that entails. We have hidden the details of OpenGL behind a library that provides higher-level drawing primitives, but authors are still writing a Python function, and so must understand to some degree the syntax and semantics of the language. In Section 2.4.1 we describe some attempts to provide a graphical interface for creating certain classes of diagrams and animations, but for now the primary method of creating material in SLITHY is writing Python code using a standard text editor.

2.4 SLITHY-1 *in production*

SLITHY-1 was the first version of the system to incorporate the idea of parameterized diagrams. Using it, we were able to put together our first nontrivial presentations. In SLITHY-1, simple graphical elements like "`Line`" and "`Rectangle`" were replaced with a single "`Diagram`" element type. A `Diagram` element represents one instance of a parameterized diagram on the slide. When a `Diagram` element is bound to a particular parameterized diagram function, it takes on the parameters of the parameterized diagram. An example of this scheme looked something like this:

```
def pulley( pull = (SCALAR, 0, 1) ):
    . . .   # procedural drawing commands

def example_animation():
    tx = Text( text = 'Slide One', color = yellow, size = 20,
               x = 30.0, y = 80.0 )
    d = Diagram( ..., draw = pulley )

    start_animation( [tx, d] )

    smooth( 1.0, tx.size, 30.0 )
    linear( 2.0, d.pull, 1.0 )

    return end_animation()
```

Here the `pulley` function is a parameterized diagram; it takes a single parameter `pull` and uses our drawing library to draw a figure. The other function, `example_animation`, constructs and returns an animation object that uses this parameterized diagram. This is done by creating a `Diagram` element whose `draw` argument is the desired parameterized diagram function. (The other

arguments, which have been elided with "..." for brevity, relate to positioning the element on the screen.) While defining the animation, the parameters of the diagram function become parameters of the diagram element, and can be manipulated by the same commands (`linear()`, `smooth()`, etc.) as parameters of built-in elements like `Text`.

In this example, the script uses `linear` to manipulate the `d.pull` parameter. Note that since the pulley's intrinsic behavior in response to the `pull` parameter (rotating the wheel, lifting the load, etc.) has been encapsulated once in the parameterized diagram, producing a more elaborate pulley movement (such as raising and lowering it repeatedly) is now just a matter of manipulating the single parameter `pull` in the appropriate way.

Multiple instances of a parameterized diagram can be on the screen simultaneously, simply by creating multiple Diagram elements that use the same underlying draw function:

```
def example_animation_2():
    d1 = Diagram( ..., draw = pulley )
    d2 = Diagram( ..., draw = pulley )

    start_animation( [d1, d2] )

    parallel()
    linear( 2.0, d1.pull, 1.0 )
    smooth( 2.0, d2.pull, 0.5 )
    end()

    return end_animation()
```

A parameterized diagram function is stateless; all information about the state of the diagram must be encoded in the values of the parameters. This animation has two instances of the `example_diagram` diagram, which can be manipulated completely independently of each other.

2.4.1 Graphical interfaces for authoring

At this point in the research we began experimenting with more accessible interfaces for creating presentations. We thought that scripting was a reasonable choice for creating animation objects, since the typical patterns of parameter change are so simple (as illustrated in Figure 2.2). Although these scripts are Python code, they typically make little if any use of more programming-oriented

features such as variables and control structures. Most animation scripts could be thought of as a simple list of commands rather than a program. We believe providing a GUI for this part of authoring would have been a straightforward exercise in software development, and was uninteresting from a research perspective.

A much more challenging problem was providing a GUI for authoring parameterized diagrams. Here we frequently take full advantage of the programming environment to create diagrams that change in complex, interesting ways in response to input parameters. While this degree of control is often useful, writing calls to individual library drawing functions is not the most intuitive way to create graphics. Given the dual-mode nature of the diagram creation task (one is both *writing a function* and *drawing a picture*), we thought it would be nice to have a dual-mode interface—one where the user could see both a graphical representation of the diagram and the underlying code. Edits to the code would be immediately reflected in the picture, and drawing objects interactively in the picture would cause the code to be changed appropriately.

We put off the question of describing how the diagram would change based on the parameter values, and first focused on a dual-mode system for creating static pictures. We created an extension mode for Emacs [24] to allow the text editor to communicate (via a socket connection) to a running instance of SLITHY-1's interactive diagram test window. One direction of editing worked well: an author could now make a change to the diagram code and see it on-screen with a single keystroke. The system would also catch syntax and runtime errors in the diagram function and send them back to Emacs, causing the editor to position the cursor at the point where the error was encountered. By streamlining the edit-test loop, the process of iteratively refining a diagram became faster and more convenient.

The other editing direction proved to be far more difficult. Our first tool let the user draw simple shapes and appended the code to generate them onto the function. This let the user create static pictures acceptably, but complicated the process of making them change according to the diagram's parameter values. A user who had interactively drawn a pulley diagram, for instance, would find that the generated code was simply a structureless list of line- and circle-drawing commands. To make

them move correctly in response to a parameter value, the author would first need to understand which drawing commands corresponded to what part of the figure, and to understand the coordinate system in which things were drawn so as to work out appropriate motions. This knowledge comes naturally when the drawing commands are written by hand, but interactively drawing a figure hides all that from the user.

We attempted to improve upon this situation by making the editor insert a newly drawn object into the function at the point where the cursor was, rather than always placing it at the end of the function. Since the user could control how the resulting code was structured, the thinking went, he or she would be able to group the commands logically to make parameterizing them easier.

This change added a new requirement, though: since the diagram library allows coordinate system transformations, we needed to know what transforms had been applied at the insertion point in order to compute the drawn shape's coordinates. For instance, consider this snippet of parameterized diagram code:

```
    # P1
translate( 3, 4 )
rotate( 45 )
    # P2
scale( 2, 0.5 )
    # P3
```

If the user interactively draws a circle on the diagram, the coordinates of the newly inserted code will differ depending on whether the cursor is at point `P1` or point `P2`. Moreover, if the cursor is at point `P3`, the shape will not even be a circle (in the drawing coordinate system) due to the nonuniform scale that has been applied. The system might need to generate completely different code to match what the user has drawn with the mouse depending on where it is to be inserted in the function.

The situation gets worse when the effects of conditionals, loops, and helper functions are considered. Consider the diagram function shown below. What code should the tool insert when the user draws a circle on-screen with the cursor at point `P4`?

```
def test_diagram( a = (INTEGER, 0, 5) ):
    while a > 0:
```

```
rectangle( 0, 0, 1, 3 )
  # P4
rotate( 20 )
a -= 1
```

Any code inserted at P4 could be executed zero, one, or multiple times depending on the value of the parameter a. Without an understanding of what the user *means* by the existing code, no tool will be able to generate the "right" code to add an interactively drawn object.

Trying to salvage something, we tried placing more and more restrictions onto where the interactive editor could be used—only in the top-level diagram function, only outside a loop, etc., etc. Eventually we pared the system down to a tool that let the user query coordinates on the screen—click a point with the mouse, see its coordinates projected back into the diagram coordinate system at the point of the cursor.[6] While this functionality is sometimes handy (and is incorporated in modified form into the current SLITHY diagram tester), it is far from the general diagram-creation tool we had hoped to build. We believe the degree of semantic understanding necessary to generate useful code is beyond the reach of current analysis methods.

We have had more success in building interactive tools for creating diagrams and animations for very *limited* domains such as "line charts" or "image slideshows," but even these tools do not have take the dual-mode approach to authoring. There is no need for the tool to analyze any human-written code; they automatically generate an entire diagram based on interactive input, and the generated code is not intended for human modification. These tools are discussed more thoroughly in Section 3.4.

Another aspect of authoring for which we attempted to provide a useful graphical interface was in laying out elements within animations. We expected that users would want the ability to create PowerPoint-style slides within SLITHY: text accompanied by figures, the difference being that the figures could be animations instead of static images. To make creating this type of animation easier, we created an interactive tool that let users graphically arrange elements on the screen. An animation script would then load the resulting "layout templates" and add the slide-specific content. A single

[6]To be precise, into the coordinate system at the *last* time execution passed the cursor.

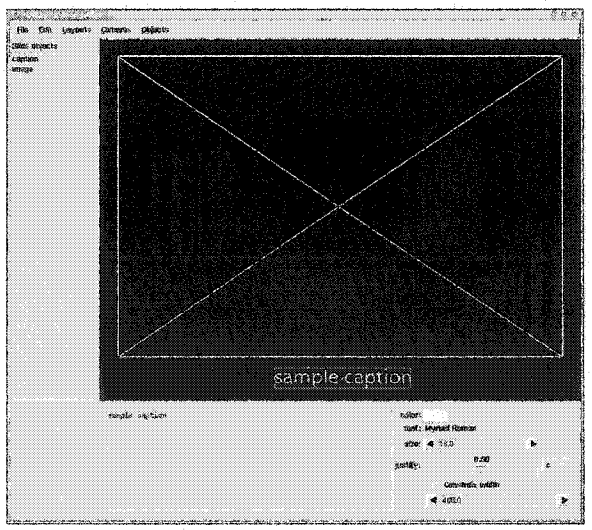


Figure 2.3 *The layout GUI included with* SLITHY-1. *Animation elements could be created and positioned interactively. The saved collection would be instanced from within an animtion script.*

template could be used repeatedly throughout a presentation, so all the slides would have a consistent look. The layout template could be edited using the tool, and all the animation scripts that used it would automatically pick up the changes.

The tool also provided a master template, modelled after the "slide master" feature in PowerPoint. Elements on the master template would be added to every layout template, allowing content that appeared throughout a presentation (such as a background fill or a running footer) to be specified in just one place instead of in every animation script.

2.4.2 Interactive diagrams

Since parameterized diagrams allow authors to hide much of the graphical complexity of a diagram and expose only a relatively small set of domain-specific parameters, it is natural to imagine giving the presenter the ability to manipulate those parameters live, during the presentation. A live interaction with a diagram could have even more impact during a presentation than a pre-scripted animation.

To enable this, we implemented *interactive controllers,* which could be used instead of an animation script to drive a parameterized diagram. Both animation scripts and interactive controllers

could produce a set of time-varying values for a diagram's parameters, but while the values produced by the animation object depend only on the time, an interactive controller can produce parameter values based on keyboard and mouse events (as well as time).

From a user perspective, our implementation was similar to that of Alice [16]: the controller had a number of hooks that are called when various user events happen, and the author could attach small animation scripts to these hooks. In SLITHY-1, the same set of commands available within an animation script were available in interactive controllers. Here's a very simple example of a controller for the `pulley` parameterized diagram used above:

```
class ExampleController(Controller):
    def key( self, k ):
        if k == 'a':
            set( self.d.pull, 0.0 )
            smooth( 2.0, self.d.pull, 1.0 )

ic = ExampleController( pulley )
```

Now the `ic` object is an interactive controller wrapped around the `pulley` function. This controller responds to only one event: when the 'a' key is pressed, the `pull` parameter of the diagram is instantaneously set to 0.0 and then raised smoothly to 1.0 over the next two seconds.

SLITHY-1 supported only a very limited type of interactive object. An interactive controller was wrapped around a single parameterized diagram, translating user input events into animated changes of that diagram's parameters. Interactive objects were inserted into the presentation at the level of the presentation script—that is, an interactive object could be shown at the top level *instead* of an animation object. The single diagram controlled by the interactive object could be shown on top of a static layout template, as described above. In SLITHY-2, many of these restrictions were removed, as described below.

2.4.3 The presentation script

In order to tie together all the animation objects created for use in a particular presentation, we introduced a new layer of script, called the *presentation script.* The purpose of this script is to

specify the order in which animations should be shown, where the system should pause and wait for the presenter to press spacebar, and to bookmark certain points in the presentation for quick random access. In SLITHY-1 it also controlled when interactive controller objects and video objects were shown instead of animation objects. Here's a sample presentation script, which simply plays a sequence of animation objects:

```
import demoanim      # import the animation objects,
                     # stored in a separate file

bookmark( 'start' )
play( demoanim.maintitle )

bookmark( 'caveats' )
play( demoanim.caveats )
play( demoanim.purpose )

bookmark( 'images')
play( demoanim.images )

bookmark( 'diagrams' )
play( demoanim.simple_diagram )
play( demoanim.invisible_diagram )
play( demoanim.image_alpha )

run_presentation()
```

The `play()` function is used to play back an animation object (or a list of such objects). The `bookmark()` function assigns a name to the current point in the presentation, allowing the presenter to jump directly to that point while presenting by selecting the name from a menu.

These presentation script commands work in a manner similar to animation script commands: they aren't executed immediately, but instead construct an object describing the presentation which can be played back later. The `run_presentation` command, always the last thing in a presentation script, is where the display of a presentation actually starts. This allows the user to interactively navigate back and forth or jump to arbitrary points during the presentation, instead of displaying things in the exact order they are mentioned in the script.

2.4.4 *User experiences*

The pieces described so for comprise SLITHY-1, the first version of SLITHY that was given out to other users. It included parameterized diagrams driven by either animation objects or interactive controllers, presentation scripts for describing the order in which to display the animated slides, and a graphical layout tool for arranging elements within an animation. We gave this system to four computer science graduate students who used it to make presentations for technical papers at the SIGGRAPH 2002 conference. We were not attempting a formal user study; we wanted mainly to get feedback on the system from an outside perspective. (Also, we had colleagues who wanted to use animation more extensively in their presentations.)

There were a number of practical concerns we had to address for our users. The first one was that all of our potential users needed support for playing video files within the presentations—not as a way of showing animated figures, but for showing results that were in video format. We added video support to the Windows version of SLITHY by using Microsoft's DirectShow library. A second consideration was ensuring that the presentations would run on the A/V equipment available at the conference. SLITHY requires substantially more system resources (especially in terms of graphics hardware) than the average PowerPoint presentation, and we would not be able to test the system before arriving at the conference, just a few days before the first of the talks. We were fortunate in that SIGGRAPH is a conference dedicated to computer graphics, and consequently makes high-powered computing and graphics hardware available to its presenters. Excerpts from these animated presentation are shown in Chapter 5.

We met with the four users as a group after their presentations had been delivered at the conference. Their reaction to SLITHY was generally positive, though they felt building presentations with the system was too time-consuming to be used for anything but major talks. For talks where mostly static slides with a few animated features would suffice, users felt they'd be more likely to make do without the animation than to use SLITHY to create the talk. This was not unexpected—after all, we strongly believe that creating good instructive animation is *intrinsically* more difficult than a typical bulleted-list type of presentation, regardless of the authoring tool used. The effort of creating

animation may be justified only for especially important presentations, or for material that is going to be used multiple times. Making SLITHY material as reusable as possible was a major goal for SLITHY-2.

A common complaint noted by the users was that positioning elements, within both animations and parameterized diagrams, was one of their most time-consuming tasks. We had already recognized this shortcoming of the system, and hoped that the graphical layout tool would be helpful, at least in positioning elements in animation scripts. Unfortunately this tool proved insufficient, since the layouts it produced were static. Users who wanted to animate the layout of the slide itself were forced to hand-edit the automatically-generated code to augment the layouts with motion. This process was difficult and made it impossible to load the layouts back into the editor.

Since the results of our attempts at making GUI-based authoring tools had been discouraging, we changed approaches and tried to improve the way authors positioned elements manually via scripting. Originally each element type had its own set of parameters for controlling its position. For instance, a static image element had two parameters defining a reference point, two more defining a width and a height, and a fifth giving the image's position relative to the point, while a diagram element took four parameters that defined an axis-aligned rectangle. This variety made it difficult to figure out what combinations of parameters would lead to a desired position on the screen, and what animation commands would lead to a desired motion.

In creating SLITHY-2 we redesigned all the animation elements to have a common positioning mechanism. Each element's position is now given by a single `Rect` object (a data type defined by SLITHY). This object specifies a rectangle (not necessarily axis-aligned) on the canvas. Users can create these rectangles from raw coordinate values, but a more natural way of using them is to start with a rectangle encompassing the entire screen (or animation window) and use the object's built-in methods to subdivide it. For instance, an element might now be created like this:

```
tx = Text( get_camera().left( 0.5 ).top( 0.15 ), ... )
```

This would create a text element that occupied the top 15% of the left half of the animation's window. Since elements in SLITHY-2 use a common positioning mechanism, this rectangle could serve as

the position for any element type. An element's position can either be a static rectangle, as shown here, or a function that returns a rectangle and whose arguments are animatable parameters. A more in-depth discussion of this mechanism appears in Section 3.2.3.

To address the difficulties with positioning graphics within parameterized diagrams, we integrated the point-querying tool into the main object tester application. Previously it had been a separate application, the remainder of our attempt at a dual-mode authoring tool. It was somewhat bothersome to start up this application and load a diagram into it just to query a point, and some of our users had not realized this separate application even existed. Previously the tool worked only with parameterized diagrams; the new integrated tool worked with animation objects and interactive objects as well. Moreover, we extended it so that it works with multiple levels of objects—if one is viewing an animation that contains a parameterized diagram, the querying mode can return positions in the coordinate system of either the animation object or the embedded parameterized diagram. The ability to query coordinate systems across different levels of the hierarchy makes it easier to align things on screen across object boundaries.

The most significant change we made after hearing from our initial group of users, however, was to loosen the relationship between parameterized diagrams and animation objects. In SLITHY-1, as in the hinting talk, there was a strict one-level hierarchy: the presentation displayed a single animation object, which could contain zero or more parameterized diagrams. This structure reflected that of existing animation systems. These systems compose a scene with character models; the animation is the final output. In presentations, however, the relationship between the "models" (diagrams) and animations is much more flexible.

Consider a situation where the presenter wants to show two animated diagrams side-by-side. In the original SLITHY, the animation script would have taken this form:

```
left = Diagram( ... )
right = Diagram( ... )

start_animation( [left, right] )
parallel()
serial()
... animation commands for left diagram ...
```

```
end()
serial()
... animation commands for right diagram ...
end()
end()
end_animation()
```

This script creates a single animation object containing two parameterized diagrams, and uses SLITHY's animation commands to drive them in parallel. It is fairly straightforward, but what if the presenter later wants to show only the left animation by itself? The animation object above is of no help—it contains two diagrams being animated in parallel. The author must create a different animation script with a single instance of the diagram, and duplicate the animation commands needed to drive the diagram in the new script. Now this animation is defined in two places in the source, which complicates editing and maintaining the presentation.

It would be easier to define this presentation if animation objects could somehow *include* one another. If this were possible, we could first build the left and right animations independently, defining each with its own script. (These scripts would essentially be the sequences of animation commands elided in the previous code segment.) Suppose that `leftanim` and `rightanim` are animation objects that have been defined in this way. The animation script that shows both in parallel can simply include these objects and play them back:

```
left = Anim( ..., anim = leftanim )
right = Anim( ..., anim = rightanim )

start_animation( [left, right] )
parallel()
left.play()
right.play()
end()
end_animation()
```

The component animations can be defined in only one place and reused more flexibly.

Our users encountered other cases where this capability would have made code reuse easier. Consider a standard slide layout with a title, some body text, and an animated figure. The user has constructed an animation that does the following:

1. The title slides onto the screen and the figure fades in.

2. The figure animates.

3. Some body text appears.

4. The figure animation continues.

5. The title slides onto the screen and the figure and body text fade out.

This sequence of events would be unremarkable even with PowerPoint. It is straightforward to construct something like this with SLITHY, but in the resulting code the sequence of commands for manipulating each piece would be interwoven. To reuse the title-sliding effect, for instance, would require extracting those portions of the script and copying them to a new script. This kind of "reuse" is frequently more time-consuming than just recreating the effect from scratch each time, as some of our users noted.

Our solution was to unify the concepts of "model" (parameterized diagram) and "animation." Instead of treating an animation as a sequence of frames to be displayed, SLITHY-2 treats an animation object much like a parameterized diagram, one with only a single scalar-valued parameter called "t" (for "time"). Now both diagrams and animations are simply objects that can be invoked to produce a picture based on a set of input parameters. We use the term *drawing objects* to refer to this type of object collectively. "Parameterized diagrams" and "animation objects" might be thought of as subclasses of this "drawing object" type, though the implementation is not structured this way. We have not encountered this unified treatment of animations and models in existing animation systems.

Under the new scheme, SLITHY-2 animations could include other animations just as easily as parameterized diagrams.

```
d = Drawable( ..., some_diagram )
a = Drawable( ..., some_animation )
```

```
start_animation( d, a )

smooth( 2.0, d.foo, 1.0 )    # manipulate some_diagram's parameters
linear( 2.5, d.bar, 6.5 )

linear( 4.0, a.t, 4.0 )      # manipulate some_animation's time parameter

end_animation()
```

The `Drawable` element replaced the `Diagram` element, to reflect the fact that it could contain either kind of drawing object. In this example `some_diagram` is a user-defined parameterized diagram function, while `some_animation` is an animation object. (There is also an `Anim` element, which is essentially a wrapper around `Drawable` that provides some convenience methods specifically for displaying animations.)

We also extended the drawing library available within parameterized diagrams to allow them to include either type of drawing object. In SLITHY-1 it was possible for one diagram to include another simply by calling it, since they are simply Python functions. In revising the system we added the `embed_object()` function, which takes a rectangle, a drawing object, and a set of parameter values for that drawing object, and causes the object to draw itself on the parameterized diagram's canvas in that location. Diagrams can use this new mechanism to include other parameterized diagrams or animation objects. In the latter case the point in time to be displayed is provided by the enclosing diagram.

Creating this unified model for diagrams and animations was the most extensive revision to SLITHY, but numerous smaller changes were made as well. The C implementations of the built-in animation elements (such as `Text`, `BulletedList`, `Image`, etc.) were replaced with Python implementations that made use of the parameterized diagram graphics library. This change cut the amount of C code in the system by roughly one-third.

SLITHY-1 had three separate Python libraries, one each for parameterized diagrams, animations, and interactive objects. They were mutually incompatible, so a user could only import one of them into any single input file. This restriction forced the three different types of object to be segregated into three separate files. Using multiple files was expected for large presentation projects,

but our four users noted frustration at having to organize their presentations according to SLITHY's restrictions (e.g., all the parameterized diagrams in one file, all the animations in another) instead of their own more logical schemes (e.g., all the material for a particular section of the talk in its own file). We combined the three libraries into one, which had the additional positive effect of eliminating more redundant code.

SLITHY-1 also had three separate test applications, one for interactively testing type of object. SLITHY-2 combined these into a single test harness, which can display drawing objects in any combination and provides a useful point-querying mechanism (as noted above) for all objects. One thing we learned from giving the system out to other users is that seemingly small things like this (having a single tester application rather than three) matter a lot. User interface matters, even for non-graphical systems. Removing as many of SLITHY-1's arbitrary-seeming restrictions as we could made the system much less of a chore to use, even though it is still based on programming.

The last significant change to SLITHY-2 was to reimplement interactive objects to be more like animations. In SLITHY-1, an interactive controller functioned as a wrapper around a single parameterized diagram, mapping input events like mouse clicks and keypresses into changes in the diagram's parameters. In SLITHY-2, an interactive controller looks much more like an animation script—it can contain zero or more animation elements and control the parameters of all of them. Elements can be created dynamically and added to or removed from the interactive animation as it is running. Parameterized diagrams are included through use of the `Drawable` element, just as in regular animation scripts.

2.5 Authoring principles

We believe that our current approach works well for creating animation that is effective and highly reusable. We can summarize our experiences in three authoring principles, with emphasis on comparing our approach to that taken by most character animation tools:

Use parameterization. The first principle is the use of parameterization at all levels of the system. The use of parameterized models is common in 3D character animation tools. Since it is

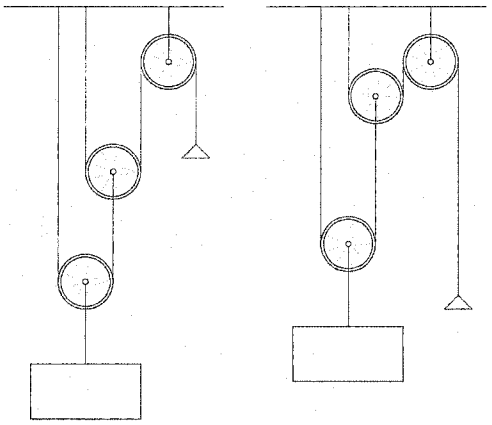


Figure 2.4 *Two instances of a pulley diagram, with the handle in different positions. Parameterization lets us animate the diagram by manipulating a single abstract "amount of pull" parameter, rather than managing all the individual graphical elements individually.*

impractical to create 3D animation by keyframing individual pieces of geometry, a layer of indirection is added. *Models* are created that encapsulate the details of geometry and expose high-level logical parameters to the animator.

This idea is just as useful in 2D as it is in 3D, though it is not so commonly seen in 2D animation tools. When we create a figure for use on an animated slide, we want to create not just a picture but also a set of behaviors that restrict how the parts of the diagram move and change, similiar to the work of Ngo *et al.* [41]. Encapsulating some of the diagram's behavior this way simplifies the task of animation considerably. Consider the pulley diagram of Figure 2.4. It is much easier to create and edit an animation by changing an abstract "pull" parameter than by moving the rectangle, lengthening and shortening the lines, rotating the triangle, and so on. We express the mapping between model parameters and the underlying geometry just once; then we can (potentially) use that model again and again in multiple animations. Of course, just as in character animation, the model and the animation cannot be designed in isolation from each other. If a character needs to smile in one scene, the model had better have a smile-control. If a slide diagram needs to animate in a certain way, the diagram creator needs to make sure it exposes the appropriate controls.

Combining graphical primitives into models is not the only application for parameterization within a presentation authoring system. Many elements of a presentation are typically used repeat-

edly throughout the talk, from the animated transitions to the layout of text on slides. We desire support for creating all these elements through parameterizable functions, to avoid repetitious work by the author and to encourage the use of a unified visual style throughout a presentation—including the ability to make changes to the style without editing each individual slide. Since SLITHY presentations are created by writing code in a general-purpose programming language, users can potentially write functions to generate any SLITHY object—parameterized diagrams, animations, or interactive objects.

Treat animations as models. The second principle we have observed is the usefulness of treating animations themselves as parameterized models that happen to have a single parameter: time. By this way of thinking, both animations and models are objects that map a set of input parameters onto a set of output graphical primitives. The only thing special about "animations" is that their input parameter set happens to consist of a single scalar value.

It would be possible in SLITHY to create an animation using the same mechanism as parameterized diagrams—by writing a Python function that takes a single parameter (time) and draws the state of the animation for the time value it is passed. This method would work but is not very practical for nontrivial animations. SLITHY animation scripts let the user create the animation by specifying a set of persistent animation elements, then describing how those elements change from one moment to the next. The resulting animation object behaves like a diagram with a single time parameter, but is constructed in an easier and more natural way.

Build slides hierarchically.

An animation object should not have to contain everything visible on the screen at once. Instead, we would like construct animations in smaller logical units and combine them to make slides, just as we would combine static graphics and text in standard presentation tools.

The result of combining animations together is, of course, a new composite animation. This suggests our final authoring principle, that of supporting deep hierarchical assembly. We want the ability to nest these characters and models within each other to any degree of depth. This ability is not typically necessary in a traditional character animation setting. There, the modeled characters

are placed into a scene, their controls manipulated via keyframing, and frames rendered out. In presentations, though, the models and animations can be much more abstract, and it often makes sense for them to be included in one another. For example, imagine a slide (an animation) that features a block diagram of a system. The diagram would be created as a parameterized model. Each block of the diagram might contain a thumbnail animation to suggest to the audience the task performed inside that block. The small animations would each contain their own models as well. While very deep nesting is not necessary – a few levels is all that is probably useful in practice – it is clearly useful to support more than just one level of models-in-animations.

In SLITHY, each object type has a mechanism for including other objects. Parameterized diagrams can use the `embed_object()` function to imperatively draw other objects, just like drawing a simple rectangle or circle. Animations and interactive objects can make use of the `Drawable` element, which is a container for any drawing object. The system supports nesting objects almost arbitrarily deep (in practice, the maximum depth is limited by the underlying OpenGL representation, but is typically in the hundreds). The next chapter will cover these mechanisms and the rest of the SLITHY system in greater detail.

Chapter 3

THE SLITHY ANIMATION SYSTEM

In this chapter, we will describe the final design and implementation of SLITHY in detail. (This is the system we referred to as SLITHY-2 in the previous section—from now on we will refer to this system simply as SLITHY.)

The fundamental building block in SLITHY is the *drawing object.* A drawing object has a fixed set of parameters; it uses the values of those parameters to produce a picture. The method used to produce the picture is dependent on the particular type of drawing object and will be discussed further below. First, though, we will introduce some terminology common to all viewing objects.

Drawing objects draw on a notionally infinite plane called the *canvas* (see Figure 3.1). The drawing might include simple shapes like lines and circles, text, or bitmap images. The drawing object also defines a *camera,* given as a rectangle, not necessarily axis-aligned, that determines what part of the canvas will be visible. The visible part is mapped into the object's *viewport,* which could be the SLITHY application window (or the whole screen in full-screen mode), or could be a rectangle on some other drawing object's canvas. If the viewport and the camera have differing aspect ratios, some parts of the canvas lying outside the rectangle will be visible. The camera rectangle will always be centered in the viewport, axis-aligned, and as large as possible.

There are three major types of drawing object available in Slithy:

- A *parameterized diagram* is a function, written by the user, that is executed on each redraw. It produces output by calling the drawing functions in our library of graphics primitives. All of the Python language is available within the diagram function; the author may use variables, arithmetic, control structures, and imported modules. The parameterized diagram may also invoke other drawing objects, passing in the parameters they need to draw themselves.

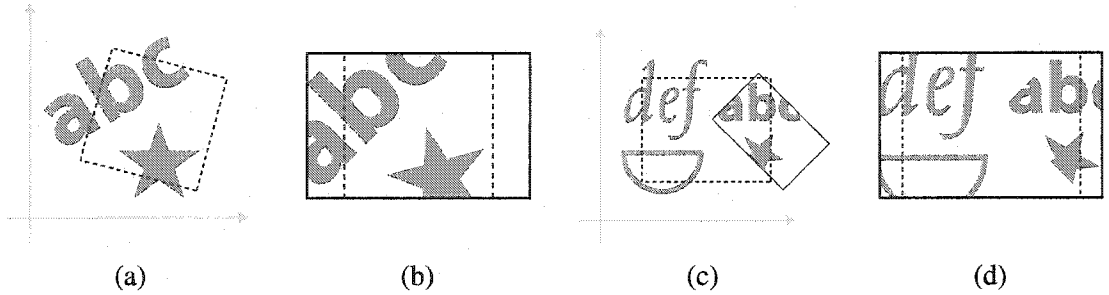


Figure 3.1 *A simple drawing object canvas (part (a)) contains some text and a star shape. The camera rectangle, shown in red, is specified in canvas coordinates. When the drawing object is mapped into a viewport (part (b)) with a different aspect ratio, additional parts of the canvas outside the camera may be visible as well. The viewport could be the entire* SLITHY *application window, or it could be a rectangle on the canvas of a different drawing object (as in part (c)). This second object has its own camera, which is mapped into a viewport in the same way (part (d)). Objects may be nested to any depth.*

- An *animation object* is a drawing object that takes a single real-valued parameter *t*, which we will usually think of as representing time. These objects are also constructed by writing Python code, but in a different way. Instead of writing a function that is executed for each redraw, as for a parameterized diagram, the animation object author writes a function (an *animation script*) that is executed just once to create the animation object. The animation object is a complete description of the *entire* animation. The SLITHY runtime system can then take this animation object and sample it to display the animation at arbitrary points in time.

- *Interactive objects* are very similar to animation objects in that they represent a mapping from a single scalar time parameter to a drawing. Unlike animation objects, though, which are completely specified before the presentation begins to play, interactive objects can change as they are playing, in response to input events such as mouse motion and keypresses. With interactive objects, the presenter is effectively generating a new animation object live, on the fly, during the presentation.

The remainder of this section will discuss the three classes of drawing objects and their implementations in more detail.

3.1 Parameterized diagrams

Parameterized diagrams are the most straightforward kind of drawing object. A parameterized diagram is simply a Python function that imperatively executes drawing commands when called. SLITHY provides a graphics library that has a variety of primitives beyond the lines and triangles provided by OpenGL.

In order for SLITHY to use a Python function as a parameterized diagram, it must know the number, names, and types of arguments expected by the function. This information is encoded using Python's default argument syntax. Like many other languages, Python functions can provide default values for their own arguments, to be used when the caller omits arguments:

```
def some_function( a = 5, b = 'hello' ):
    print a, b

some_function(3, 21)        # prints "3 21"
some_function(3)            # prints "3 hello"
some_function()             # prints "5 hello"
```

SLITHY co-opts this syntax as a convenient way to describe a diagram's parameters. Each parameter for a diagram is an argument to the function. The "default value" is not an actual value for the parameter, but is instead a tuple that describes the parameter's type and other necessary information. Here is a simple example:

```
def my_diagram( x = (SCALAR, 0.0, 1.0, 0.5),
                y = (STRING, 'hello') ):
    . . .
```

When asked to use this function as a parameterized diagram, SLITHY will access Python's internal representation of the function to find these "default value" tuples, interpreting them as a description of the parameters. In this case, the parameter "x" is defined to be a real-valued number between zero and one, with a default value (an actual default value this time) of 0.5. Similarly, "y" is defined as a string-valued parameter with "`hello`" as its default value. Note that if we were to call this function with no arguments, then parameter `x` would receive the value "`(SCALAR, 0.0, 1.0, 0.5)`",

a Python tuple containing four items. This would almost certainly cause errors within the function body, which is expecting `x` to be a real-valued number. Whenever SLITHY invokes a parameterized diagram, it will explicitly pass values for all parameters to avoid this situation.

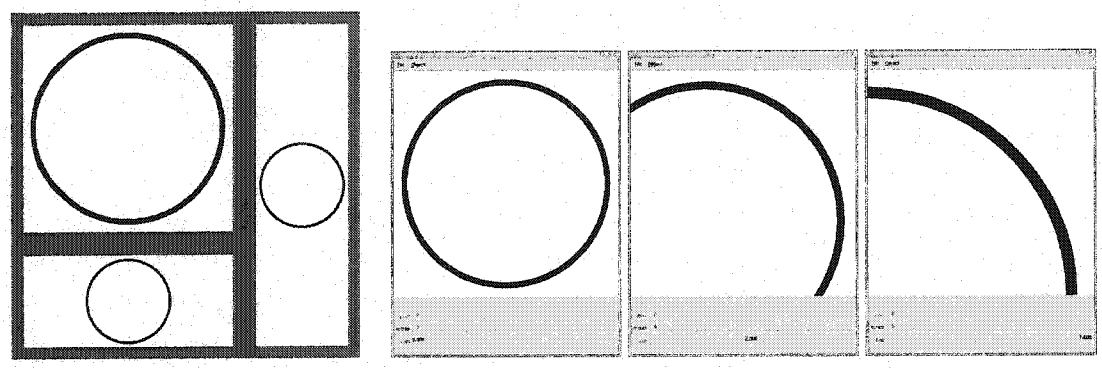
The first item of each parameter's description tuple is its type. The values `SCALAR` and `STRING` are constants defined in the SLITHY library. Table A.1 contains a complete listing of the available types. The type information is used by the tester application to provide appropriate UI controls for interactively manipulating the diagram (in this case, a slider for `x` and a text entry box for `y`). The information is also used within animation scripts, since scripts reference parameters by name. Also, some animation commands can only operate on parameters of certain types—a linear interpolation, for instance, can be performed on a scalar-valued parameter but not on a string-valued parameter.

3.1.1 Specifying the camera

The first thing that most parameterized diagrams must do is specify the camera. The camera is implemented by manipulating the OpenGL model-view matrix, so it must be set before any drawing is done. The camera is given as a rectangle (a `Rect` object, defined by SLITHY and documented in Section A.2). Here we'll begin building a simple diagram to display a clock:

```
def clock( hour = (INTEGER, 0, 24, 0),
           minute = (INTEGER, 0, 60, 0) ):
    set_camera( Rect( -10, -10, 10, 10 ) )
    clear( white )
    thickness( 0.5 )
    circle( 9, 0, 0 )
```

In this example the call to `set_camera()` sets the camera to the square $[-10, 10] \times [-10, 10]$. This region of the canvas – a square area slightly larger that the circle drawn by the last line – is guaranteed to be seen in the viewport. If the viewport happens to be square, then the camera rectangle will fill the viewport exactly. If the viewport is wider or taller than the camera, then more of the canvas will be visible, in equal amounts either above and below the camera rectangle, or on the left and right sides. Figure 3.2(a) shows an animation object that includes three instances of this diagram in viewports of various sizes. (The remaining commands of the function clear the canvas



(a) (b)

Figure 3.2 *Showing how the camera rectangle affects a diagram's appearance. In part (a), we see a diagram with a fixed camera rectangle as it appears in viewports of various aspect ratios. Part (b) shows a diagram whose camera rectangle is computed based upon the* `cam` *input parameter. The images of part (b) are screenshots of the* SLITHY *test application, which maps the parameters of the diagram onto the sliders at the bottom of the window to allow interactive manipulation.*

to white and draw a black circle of radius 9 centered on the origin. They will be introduced in the following sections; they are included here so that something is visible in the figure.)

Like everything else in a parameterized diagram, the camera may be computed based on the parameter values. The next example adds a parameter `cam` which is used to interpolate the camera between two values:

```
def clock( hour = (INTEGER, 0, 23, 0),
           minute = (INTEGER, 0, 60, 0),
           cam = (SCALAR, 0, 1, 0) ):
    set_camera( Rect(-10,-10,10,10).interp( Rect(0,0,10,10), cam ) )
    clear( white )
    thickness( 0.5 )
    circle( 9, 0, 0 )
```

Figure 3.2(b) shows the result as we interactively manipulate the `cam` parameter using the object tester. In each image the circle is drawn in the same place on the diagram's canvas; manipulating the camera produces the change in view.

3.1.2 *Drawing primitives*

To draw on the screen, the Slithy graphics library contains a set of functions for common primitives: `line()` draws a polyline, `circle()` and `dot()` draw stroked and filled circles, respectively, `frame()` and `rectangle()` draw stroked and filled rectangles, respectively, and `polygon()` draws an arbitrary filled polygon. All of SLITHY's "stroke" primitives are actually drawn with strips of triangles rather than with OpenGL lines. This method of line-drawing reduces performance somewhat, but means that stroke thickness is specified in canvas coordinate space, which is easier to work with than screen space and scales correctly when the diagram is drawn at different sizes. Stroke thickness is set using the `thickness()` function.

All of the drawing functions draw in the current drawing color, which is set with a call to `color()`. The `color()` function accepts graylevel or RGB tuples, with an optional alpha component. (Transparency is handled using a simple painter's algorithm.) The `color()` function can also accept a *color object* as its argument. SLITHY predefines a number of color objects with names like `red` and `blue`, and the user can define new color objects as well. Defining new objects is useful for creating a consistent style throughout the presentation—for instance, the author might define a color object called "`caption_color`" and use it throughout the presentation. If it is used consistently, then to change all the captions in the presentation one only needs to change the definition of `caption_color`. Color objects also have methods for translating RGB to and from other color spaces and interpolating between colors.

The `clear()` function clears the diagram viewport to a given solid color. The current drawing color is not used; instead, the color is specified as an argument to `clear()` (using any of the forms legal for the `color()` function). It is not required to clear the canvas before drawing on it; by not clearing the canvas the objects drawn by a parameterized diagram will appear on top of whatever object includes the diagram. (Essentially this means that the background of the diagram is transparent.)

3.1.3 Graphics state

The current color and stroke thickness are two components of the *graphics state*. Another component is the current user space transform, used to map the coordinates given to primitive drawing functions to positions on the canvas. Initially this transform is the identity, but various affine transformations can be multiplied into it using the `translate()`, `rotate()`, `scale()`, and `shear()` functions.

The functions `push()` and `pop()` are used to save and restore the current graphics state, similar to the `gsave` and `grestore` operators in PostScript. The `push()` function pushes a copy of the current state onto a stack, while `pop()` discards the current state and replaces it with the state popped off the stack. State changes can thus be confined to a particular section of code by bracketing it in a `push()`/`pop()` pair.

We'll use the rectangle drawing primitive and coordinate system transforms to add hour markers to the clock:

```
push()
for i in range(12):
    push()
    translate( 7.5, 0 )
    scale( 0.5, 0.125 )
    rotate( 45 )
    rectangle( -1, -1, 1, 1 )
    pop()

    rotate( 30 )
pop()
```

Within the loop, a series of transforms concatenated together serve to turn the square drawn by `rectangle()` into a flattened diamond and position it at the edge of the clock face. The effect of individual transforms is illustrated in Figure 3.3(a). The diamond is drawn twelve times, turning the coordinate system after each one is drawn to position it correctly near the circumference of the face. The resulting picture is shown in Figure 3.3(b).

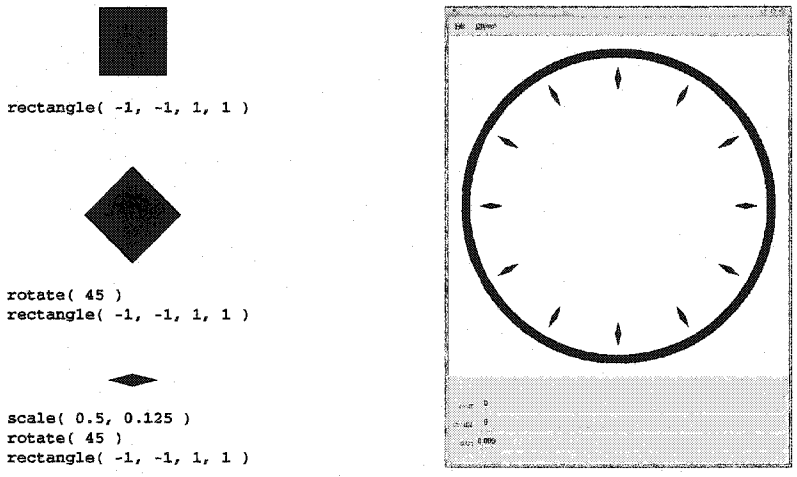


Figure 3.3 *Adding diamond-shaped hour markers to the clock face. Part (a) shows how coordinate system transforms are used to turn a square into a flattened diamond shape. Part (b) shows the clock face after a loop places 12 markers around its circumference.*

3.1.4 Path objects

For more complex shapes, SLITHY's *path objects* can be used. A path object encapsulates a path that is built via a sequence of method calls (e.g., `moveto()`, `lineto()`, `curveto()`, etc.) that work similarly to their PostScript namesakes. Path segments can be straight lines, or quadratic or cubic Bézier curves. Paths can contain multiple subpaths, each of which can be either open or closed. One difference that users accustomed to PostScript may encounter is that changes to the user space transform do not affect a path's *definition*. The transform is applied only when the path is drawn on the screen.

Constructing a path object does not draw anything on the screen. To draw a path, the object is passed to the `stroke()` or `fill()` functions, which draw the path stroked or filled in the current color (and line thickness, in the case of stroking). Paths can also be drawn with arrowheads at the ends, since arrows are a common graphic element in presentation diagrams.

Rendering a path is a relatively expensive operation. Curve segments must be expanded to polylines (SLITHY uses an adaptive subdivision algorithm to do this). Filling a path requires triangulating it in order to handle convexities and holes. Stroking it is even more expensive since Slithy

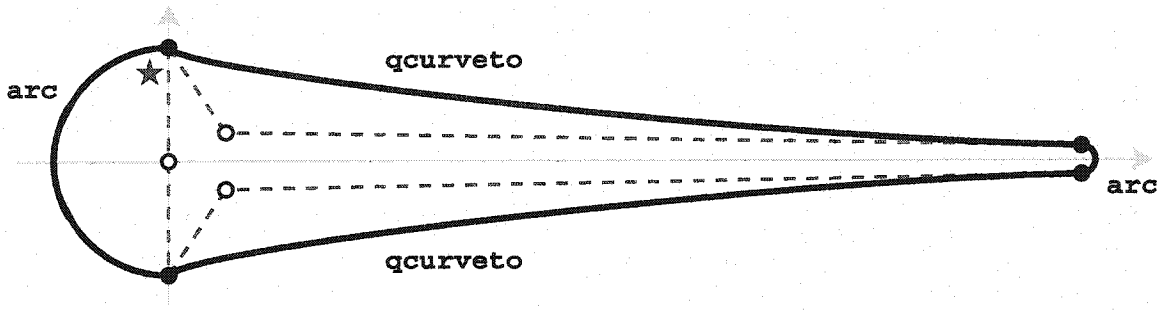


Figure 3.4 *Schematic diagram of the path object used to draw the hands of the clock example. The open circles and dotted lines mark reference points used in the shape's construction, such as the center of circular arcs and the off-curve control points for Bézier curves.*

must construct a triangle strip for each segment and then join the strips together with an appropriate miter or bezel. For efficiency, Slithy caches the result of triangulating a path object in an OpenGL display list, so that the computation can be avoided when an object is drawn repeatedly.

Path objects are created by calling the `Path()` constructor. Line and curve segments are appended to a path object by calling methods on it; the available methods are detailed in Section A.5.2. For the clock example, we'll use a path object to define a curved, tapered object for the hands of the clock:

```
hand = Path()
hand.moveto(0, 1).arc(0, 0, 90, 270, 1)
hand.qcurveto(0.5, -0.25, 8, -0.125)
hand.arc(8, 0, 270, 90, 0.125)
hand.qcurveto(0.5, 0.25, 0, 1)
hand.closepath()
```

This shape is illustrated in Figure 3.4. The path is constructed starting at the vertex marked with a star and proceeds counterclockwise. Path object methods use a common technique of returning a reference to the object itself, so that multiple method calls can be chained together. The second line of the above sample shows this usage, where `moveto()` and `arc()` are both called in a single statement. All of the method calls could have been chained together in this fashion, but here they have been shown as individual statements for clarity.

An important point to note is that this `hand` object should be constructed *outside* the param-

eterized diagram function—the above block of code should go *before* the definition of `clock()`, rather than within it. The primary purpose of path objects is to allow SLITHY to cache complex shapes that are drawn repeatedly; to define it within the diagram function would result in the path being constructed, drawn, and discarded each time the diagram is redrawn. Since the clock hand shape does not change between redraws, we can incur the cost of constructing and triangulating it just once, no matter how many times the diagram function is called.

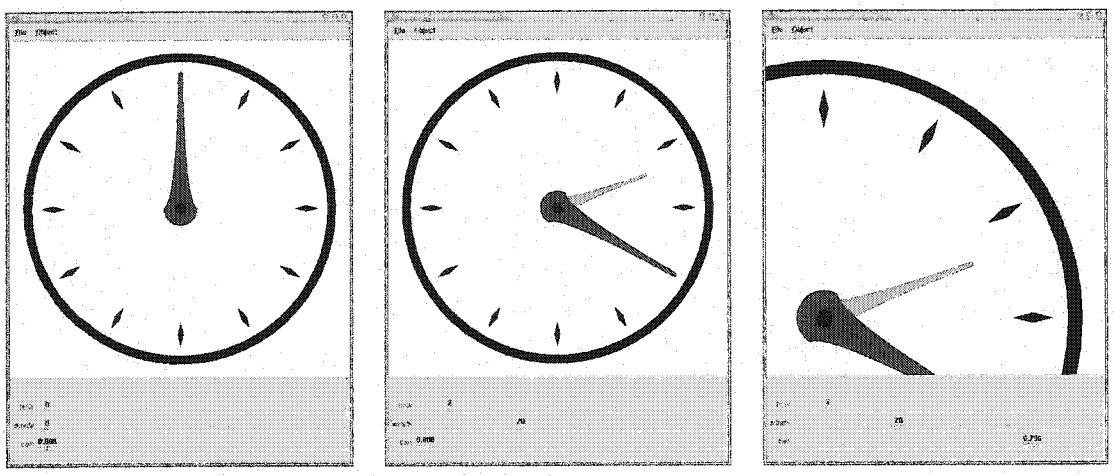
Within the diagram code, we use the `fill()` and `stroke()` functions to draw a path object on the canvas. These draw the path object in the current color, using the current user space transformation. Here we will draw one instance of the `hand` path in red to form the minute hand:

```
rotate( 90 )

push()
color( red )
rotate( -minute * 6 )
fill( hand )
pop()
```

This section of code is the first time we have made use of the `minute` parameter defined at the top of the function. The first call to `rotate()` turns the coordinate system so that the hand path object, which would point to the right (as in Figure 3.4), points straight up instead. We then multiply the value of the minute parameter by six to transform it into the correct rotation angle in degrees, relative to the straight-up position. (The negative sign is necessary because the `rotate()` function rotates counterclockwise; we want our clock hand to move in the clockwise direction.) We then invoke `fill()` to draw the hand in the correctly rotated coordinate system.

The next part of the code will draw a second instance of the path object in green as the hour hand. We want the hour hand to be underneath the minute hand, so we must draw it *before* the minute hand. This block will be inserted between the "`rotate( 90 )`" and the block that draws the minute hand.

```
push()
color( green )
rotate( -(hour * 30 + minute / 2.0) )
```



(a) (b) (c)

Figure 3.5 *The clock example diagram after the hands have been added. Part (a) shows the diagram in its default state. In part (b) the parameter sliders have been used to make the clock display 2:20. Changing the camera rectangle with the* `cam` *parameter still affects the whole diagram, as shown in part (c).*

```
scale( 0.7 )
fill( hand )
pop()
```

Here we must use both the `hour` and `minute` parameters to calculate the correct angle for the hour hand. We also use `scale()` to shrink down the hand slightly. Figure 3.7 shows the resulting diagram.

3.1.5 Text and images

The library can render high-quality text from PostScript Type 1 or TrueType fonts using the open-source Freetype library. When a font is loaded, SLITHY creates a texture map containing characters from the font. To render text into a diagram, the system draws one quadrilateral for each character, setting the texture coordinates appropriately to extract each character shape. This process is illustrated in Figure 3.6. The library's `text` function supports a variety of positioning and justification options, simple word-wrapping, and using multiple fonts and colors within a single text string.

Bitmap images are also drawn using texture maps. Storing images as textures allows them to be

(a) (b) (c)

Figure 3.6 SLITHY *renders text by drawing a quadrilateral for each character (part (a)). A font is represented by rasterizing each character into a texture map (part (b)), packing the characters together tightly to conserve texture memory. Applying the appropriate part of the texture map to each quadrilateral results in rendered text (part (c)).*

scaled, rotated, and made semitransparent, just like the primitive drawing shapes. It also means that image data is stored on the graphics card rather than in system memory, increasing the performance of the system.

Scripts use the `load_image` function to load an image from a file (most common image file formats are supported, thanks to use of the Python Imaging Library). This function returns an *image object,* which can be thought of as a handle to the image in memory. Within a parameterized diagram, the `image` function is used to draw an image object to the screen, with controls for scaling and positioning.

The return value of both the `text` and `image` functions is a Python dictionary object, which contains information about the bounding box of the text or image drawn. This information is often useful in aligning other drawing elements. This feature was added in response to requests from our first set of users.

We can use the `text()` function to add a label beneath the clock. We will add the label text as a parameter of the diagram, and adjust the camera to allow a little bit of extra room beneath the clock. Here's how the updated clock diagram begins, with the changes marked in bold type:

```
def clock( hour = (INTEGER, 0, 23, 0),
           minute = (INTEGER, 0, 60, 0),
```

```
        cam = (SCALAR, 0, 1, 0),
        label = (STRING, '') ):
    set_camera( Rect(-10,-12,10,10).interp( Rect(0,0,10,10), cam ) )
```

Then, at the end of the function we can call `text()` to draw the string:

```
    text( 0, -10.5, label, font = thefont, size = 2 )
```

The function takes a pair of coordinates for positioning the text and the string of text to be drawn. The font is specified by passing a *font object,* which is a token return by SLITHY when it loads a font file. Complete documentation on the `text()` function and font objects can be found in Appendix A.

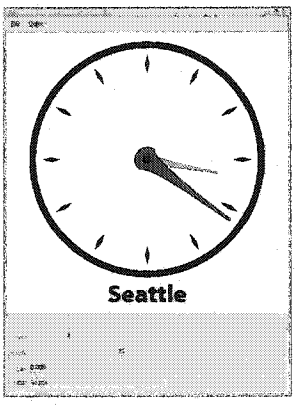
Figure 3.7 shows a complete SLITHY input file defining the final clock diagram.

3.2 Animation scripts

Earlier we claimed it is desirable to treat animations as simply a special case of parameterized diagrams, where the diagram has just a single scalar-valued parameter representing time. We can imagine explicitly building animations as parameterized diagram functions. Consider creating a four-second animation based on the clock: first the clock advances from 2:00 to 2:45 in two seconds, then pauses for one second, then reverses back to 2:30 in the final second. We could write a parameterized diagram to implement this animation by computing the value for each of the clock diagram's parameters based on a single time parameter:

```
def clock_animation( t = (SCALAR, 0, 4, 0) ):
    label = 'Seattle'
    cam = 0
    hour = 2

    if t < 2:
        minute = int(t / 2.0 * 45)          # interval [0.0, 2.0)
    elif t < 3:
        minute = 45                         # interval [2.0, 3.0)
    else:
        minute = int(45 - 15 * (t-3.0))     # interval [3.0, 4.0]

    clock( hour, minute, cam, label )
```

```
from slithy.library import *

# define the hand shape
hand = Path().moveto(0,1).arc(0,0,90,270,1)
hand.qcurveto(0.5,-0.25,8,-0.125).arc(8,0,270,90,0.125)
hand.qcurveto(0.5,0.25,0,1).closepath()

thefont = load_font( 'wmb.pfb', 60 )      # 'wmr.pfb' is a PostScript
                                          # Type 1 font file

def clock( hour = (INTEGER, 0, 23, 0),
           minute = (INTEGER, 0, 60, 0),
           cam = (SCALAR, 0, 1, 0),
           label = (STRING, '') ):
    set_camera( Rect(-10,-12,10,10).interp( Rect(0,0,10,10), cam ) )
    clear( white )
    thickness( 0.5 )
    circle( 9, 0, 0 )

    push()                          # draw the diamond-shaped markers
    for i in range(12):
        push()
        translate( 7.5, 0 )
        scale( 0.5, 0.125 )
        rotate( 45 )
        rectangle( -1, -1, 1, 1 )
        pop()

        rotate( 30 )
    pop()

    push()
    rotate( 90 )

    push()                          # draw the hour hand
    color( green )
    rotate( -(hour * 30 + minute / 2.0) )
    scale( 0.7 )
    fill( hand )
    pop()

    push()                          # draw the minute hand
    color( red )
    rotate( -minute * 6 )
    fill( hand )
    pop()
    pop()

    color( black )
    dot( 0.3 )

    text( 0, -10.5, label, font = thefont, size = 2 )

test_objects( clock )
```



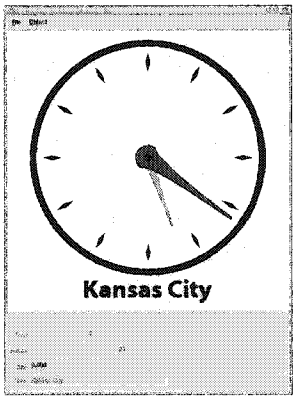


Figure 3.7 *The complete* SLITHY *script for defining a simple clock parameterized diagram, constructed in Section 3.1. Two screenshots of the tester window with various parameter settings are shown on the right.*

Three of the clock diagram's parameters (`label`, `cam`, and `hour`) are constant throughout the animation, so determining their values is simple. The minute parameter, however, goes through three distinct phases during the animation, so we must first determine into which interval the current value of `t` falls, then compute `minute` appropriately for each interval.

While creating an animation in this way is difficult, editing it is even worse. To lengthen the animation by expanding the initial two-second part of the animation to two and a half seconds causes changes to cascade down through the other sections:

```
    if t < 2.5:
```

```
    minute = int(t / 2.5 * 45)          # interval [0.0, 2.5)
elif t < 3.5:
    minute = 45                         # interval [2.5, 3.5)
else:
    minute = int(45 - 15 * (t-3.5))     # interval [3.5, 4.5]
```

Here all the changed values have been marked in bold. If we wanted to introduce effects like smooth rather than linear interpolation between values, the code would get even more complicated. Clearly using the parameterized diagram mechanism directly is not the right approach to making animations that behave like models.

SLITHY's solution is to use a different kind of script, called an *animation script*. Instead of being called once per redraw to draw a single frame, an animation script is called only once to produce a description of the entire animation. In effect, the output of the animation script is a data structure equivalent to the `clock_animation()` function listed above. It does not contain any actual drawing functionality, but it substitutes for the code that computes a diagram's parameter values based on the input time parameter `t`.

Here is how our simple clock animation example might be written as an animation script:[1]

```
set( label, 'Seattle' )     # set initial values
set( cam, 0 )
set( hour, 2 )
set( minute, 0 )

linear( 2, minute, 45 )     # over 2 sec, run "minute" up to 45
wait( 1 )                   # do nothing for 1 sec
linear( 1, minute, 30 )     # over 1 sec, run "minute" down to 30
```

Now, even without the comments, the code more clearly reflects the design of the animation. Executing this piece of code produces a data structure that encapsulates a function for each parameter:

$$\texttt{label}(t) = \texttt{'Seattle'}$$

$$\texttt{cam}(t) = 0$$

[1]For simplicity, this code is presented in a somewhat abstracted version of SLITHY's syntax. More realistic examples will appear later in this section.

$$\texttt{hour}(t) = 2$$

$$\texttt{minute}(t) = \begin{cases} \text{linear}(0,0,2,45,t) & \text{for } t \in [0,2) \\ 45 & \text{for } t \in [2,3) \\ \text{linear}(3,45,4,30,t) & \text{for } t \in [3,4] \end{cases}$$

where

$$\text{linear}(x_1, y_1, x_2, y_2, t) = y_1 + (y_2 - y_1)(t - x_1)/(x_2 - x_1)$$

This set of functions can be evaluated at any time $t \in [0,4]$ to produce the appropriate parameter values for the `clock` parameterized diagram. These functions are clearly reminiscent of the parameterized diagram-style code we originally used to create this animation: the if–elif–else structure, for instance, is reflected in the block structure of the `minute` function. The difference is that since we are using an animation script to describe the mapping from t to parameter values at a somewhat higher level than direct computation, editing the animation becomes easier. To make the same change we did before, extending the initial segment of animation by half a second, now requires just one fairly intuitive change:

```
linear( 2.5, minute, 45 )      # over 2.5 sec, run "minute" up to 45
```

Executing the modified script will propagate the effects of the change automatically, so the correct functions and intervals are generated.

3.2.1 Animation scripts in SLITHY

In the simplified example above, we showed an animation script controlling the parameters of the clock parameterized diagram. In practice, it is often useful to have multiple objects under the control of a single animation. In SLITHY these objects are called *animation elements* (or just *elements*). All of an element's parameters may be controlled via the animation script. Each element has a position on the animation's canvas (these positions may be animated as well). Just like a parameterized diagram, the animation has a camera rectangle that defines what portions of the canvas are visible.

The simplest element is the `Drawable` element, which acts as a container for another drawing object (such as a parameterized diagram or animation object). The `Drawable` element takes on the parameters of whatever object it is used to contain. Here we create an element to contain the clock diagram:

```
d = Drawable( get_camera().left(0.5).inset(0.05),
              clock,
              label = 'Seattle', hour = 2 )
```

The first argument is the position of the element on the canvas, which is computed in this case by taking the default camera rectangle (returned by `get_camera()`) and subdividing it to obtain the desired location for the diagram. The second argument is the drawing object itself, which for this example is the `clock` diagram. The rest of the arguments are optional, and provide default values for the parameters of the contained object.

SLITHY provides other element types for drawing certain commonly needed slide elements. The `Text` element draws a single string of text, the `Image` element draws a single static bitmap image, the `BulletedList` element provides a simple PowerPoint-style text box with hierarchical indentation and bullet points, and so on. Each of these can be thought of as a `Drawable` element and a very simple parameterized diagram function rolled into one—each has a position on the canvas and a set of animatable parameters for controlling what they draw. The rendering of these specialized elements is, in fact, done using the very same graphics library that parameterized diagrams use. For our example we will create an instance of the `Fill` element, which draws a simple colored rectangle:

```
bg = Fill( style = 'horz', color = black, color2 = blue )
```

Because this element is commonly used to fill the whole screen (thus the name `Fill`), the position argument is optional. If omitted, as it is here, it fills the entire viewport. The other arguments are default values for the element's parameters, just as with the `Drawable` element. Here we have specified a horizontal gradient fill, going from black at the top to blue at the bottom.

Once we have created the required elements, we can define an animation object.

Figure 3.8 *The sole frame of a trivial zero-length animation object.*

```
start_animation( bg, d )
animobj = end_animation()
```

The `start_animation()` function starts a new animation object. It takes zero or more elements as arguments—these are the elements that initially appear in the animation. The order is significant. Elements are rendered in the order in which they appear in the list, so later elements will be drawn atop earlier ones. Both the set of elements and their ordering can be changed within the animation; the arguments to `start_animation()` only specify the initial state.

The `end_animation()` function finishes up an animation definition and returns the resulting animation object. Since no commands appeared between the start and end, this example defines an animation of length zero. This trivial animation is shown in Figure 3.8. It shows both the black-blue gradient fill in the background and an instance of the clock diagram on the left side. The figure also shows that some of the clock diagram's built-in parameter defaults have been overridden in creating the element: the hour hand is at 2, and the label reads "Seattle."

To make something actually happen in the animation, we insert some animation commands between `start_animation()` and `end_animation()`:

```
start_animation( bg, d )
```

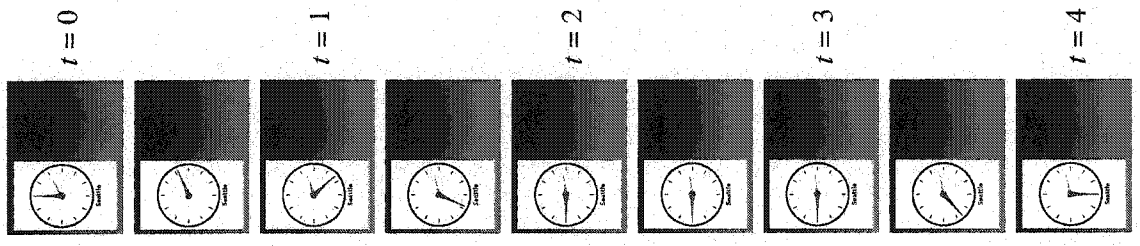


Figure 3.9 *Frames from a simple clock animation (rotated sideways).*

```
linear( 2, d.minute, 45 )
wait( 1 )
linear( 1, d.minute, 30 )
animobj = end_animation()
```

The notation "*element.parameter*" is used to refer to a particular parameter, since a given parameter name may be used by multiple elements. Setting the default values when creating elements takes the place of the initial `set()` commands used in the first abstract version of this animation script. This example uses SLITHY's actual syntax.

Frames from this simple animation are shown in Figure 3.9. As expected, the minute hand advances by 45 minutes, then stops, then reverses back to the :30 position. Notice also that the hour hand is moving as well, since its position is also tied to the "minute" parameter. This is the primary advantage of using parameterized diagrams. Even though two graphical elements are moving in this animation, we have already expressed their position in terms of the abstract parameter `minute`. Now to make an animation, we need only worry about how to make the abstract parameters change in the way we want, and the graphics will take care of themselves.

Within an animation object, each parameter of each element is represented by a data structure called a *timeline*. A timeline represents the function mapping the input time t to a value for that parameter. A timeline is a partition of the interval $(-\infty, \infty)$ into a set of nonoverlapping domains. For each domain the timeline has either a constant value for that parameter, or a function that will produce the value based on the value of t.

At the start of the animation each timeline is initialized to a single domain, extending to infinity in both directions, with a constant value equal to that parameter's default. Animation commands

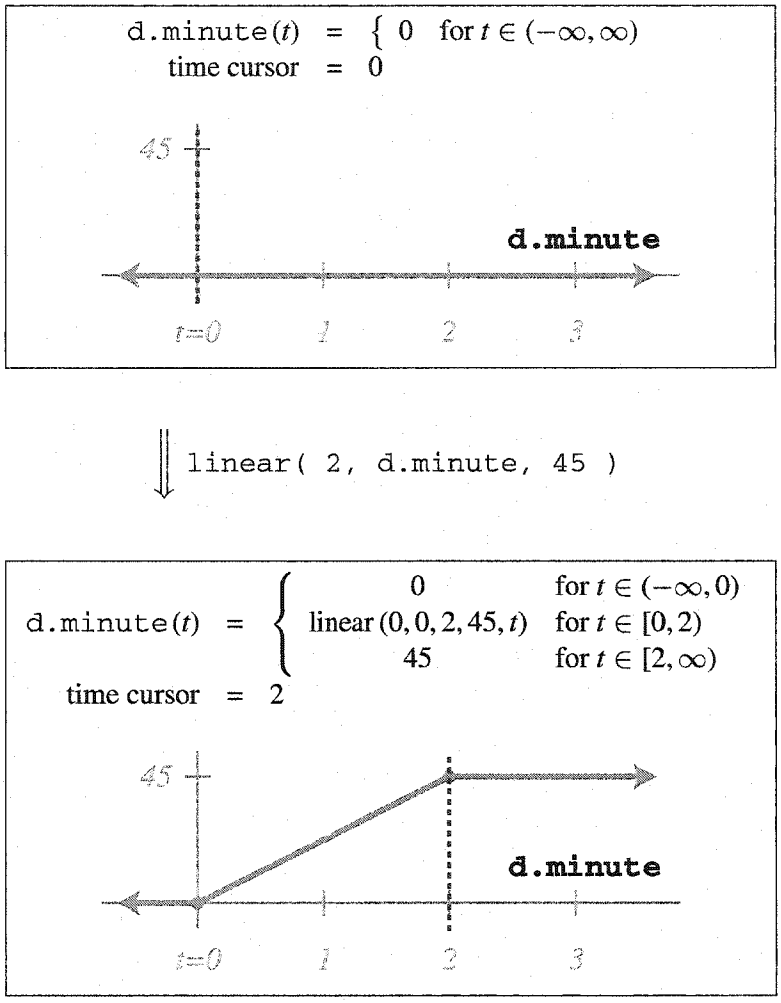


Figure 3.10 *Illustrating how the* `linear` *command operates by overwriting a portion of a parameter's timeline starting at the time cursor.*

such as `linear()` then edit the timeline of the parameter they operate on, overwriting portions of the timeline with new domains. Figure 3.10 illustrates the effect of the `linear()` function on a timeline.

This type of edit typically takes place at the position of the *time cursor*. There is only a single time cursor in an animation script; it is not unique to each parameter or each element. Therefore a series of commands will appear to happen in sequence, even if they modify different parameters:

```
linear( 2, d.minute, 45 )
linear( 2, d.hour, 10 )
```

In this example, we assume the time cursor begins at $t = 0$. The first command changes the `minute` over a two-second duration, which leaves the time cursor at $t = 2$. The second command then modifies the `hour` parameter, also over a two-second duration, starting at $t = 2$ (and afterwards, advancing the cursor to $t = 4$). The net result is a four-second animation in which the minute hand moves first, followed by the hour hand.

3.2.2 Expressing parallelism

Many animations require multiple parameters to be changing simultaneously. For example, we might wish to have an animation with two instances of the clock diagram, and show the clocks running together. SLITHY implements parallelism in animation scripts with three functions: `parallel()`, `serial()`, and `end()`.

At the start of an animation script the system is in *serial mode*, and the time cursor behaves as described above. After each command that has a duration (such as `linear()`), the cursor is advanced by the duration of the command. Calling the `parallel()` function switches the system into *parallel* mode, which changes the behavior of the cursor. In parallel mode the cursor is *not* advanced after each command, so all commands will begin at the same time. Instead, the system remembers the durations of each command within the parallel block, and when it is ended (by calling `end()`) the cursor is advanced once, by the duration of the longest of the parallel events.

The net effect is that a script like this:

```
parallel()
linear( 2, d.minute, 45 )
linear( 3, bg.color2, green )
end()
```

causes the parameters `d.minute` and `bg.color2` (one of the colors of the background fill) to change together. At the end of these four lines the cursor has been advanced by three seconds, the

length of the longest single animation command.

We can think of a `parallel()/end()` block as creating a "composite" animation event: a collection of component animation commands that are executed together, but moving the time cursor just once, as if a single command were given whose duration was equal to the maximum of the individual commands. A `serial()/end()` block creates a different kind of composite: it causes the component commands to be executed in sequence, moving the time cursor just like a single command whose duration was the *sum* of the individual commands. These composite commands may be nested within one another to any level, and mixed with the primitive commands like `linear`.

3.2.3 Moving elements on the canvas

An element's position on the canvas may be animated just like its intrinsic parameters, by specifying the element's position not as a static rectangle, but as a *pseudoelement*. A pseudoelement is like a regular element in that it has parameters that are controlled by a timeline, but instead of producing a picture as an element does, a pseudoelement produces a rectangle object. This rectangle is then used as the position for an ordinary element.

As an example, here is an animation script where the clock element moves during the animation:

```
# create a pseudoelement that interpolates between two rectangles
dv = viewport.interp( get_camera().left(0.5).inset(0.05),
                      get_camera().right(0.5).inset(0.05) )

d = Drawable( dv,   # the pseudoelement provides the diagram's position
              clock,
              label = 'Seattle', hour = 2 )
bg = Fill( style = 'horz', color = black, color2 = blue )

start_animation( bg, d )
linear( 2, d.minute, 45 )
smooth( 2, dv.x, 1 )        # the pseudoelement's 'x' parameter
                            # controls the interpolation
animobj = end_animation()
```

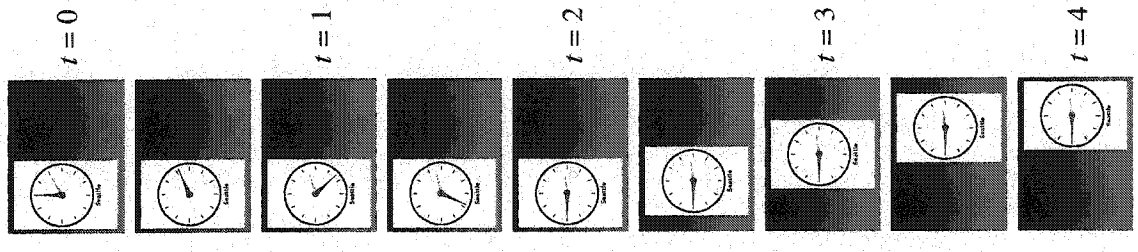


Figure 3.11 *The result of using a pseudoelement to animate an element's position.*

This animation is shown in Figure 3.11. The effect of changing the `x` parameter of the `dv` pseudoelement is to change the `d` element's position on the canvas. Also, this example is our first use of the `smooth()` function, which transitions a parameter to a new value over a duration, just like `linear()`, but using a slow-in–slow-out interpolation instead of linear interpolation.

Figure 3.12 shows a complete SLITHY animation script, along with frames of the resulting animation. It uses a common authoring convention of defining each animation in its own function ("`def clock_animation():`") which returns the animation object. At the end, the line "`clock_animation = clock_animation()`" executes the function, then binds the name `clock_animation` to the return value (the animation object), effectively discarding the animation script itself once the execution is complete.

3.2.4 Changing the element set

The `start_animation()` function takes as arguments the initial set of elements to appear in the animation and their stacking order, but as mentioned above, neither the set nor the ordering is fixed for the whole animation. The `enter()` and `exit()` functions add and remove elements from the set of active elements, while `lift()` and `lower()` change their relative stacking order. These are all zero-duration commands, like `set()`, that take place at the position of the time cursor. The sequence of objects over time is represented using the same timeline mechanism as for element parameters; the four functions mentioned here are simply specialized animation commands for manipulating this working set timeline. If `a`, `b`, `c`, and `d` are elements, then the animation script

```
from slithy.library import *

. . .  # definition of the "clock" parameterized diagram here

def clock_animation():
    c = get_camera()
    d1v = viewport.interp( c.restrict_aspect(2.0/3.0).inset(0.05),
                           c.left(0.5).inset(0.05) )
    d1 = Drawable( d1v,
                   clock,
                   label = 'Seattle', hour = 2 )
    d2 = Drawable( get_camera().right(0.5).inset(0.05),
                   clock,
                   label = 'Kansas City', hour = 4, _alpha = 0 )
    bg = Fill( style = 'horz', color = black, color2 = blue )

    start_animation( bg, d1, d2 )

    parallel()
    linear( 5, d1.minute, 60 )
    linear( 5, d2.minute, 60 )
    linear( 5, bg.color2, green )

    serial()
    wait( 1 )
    smooth( 2, d1v.x, 1 )
    fade_in( 2, d2 )
    end()
    end()

    return end_animation()
clock_animation = clock_animation()

test_objects( clock_animation )
```

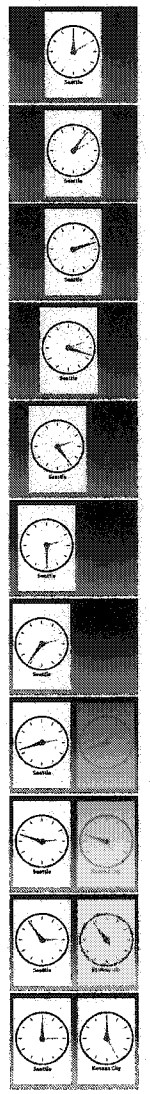


Figure 3.12 *A complete* SLITHY *script for defining a simple clock animation (except for the definition of the clock diagram, which can be found in Figure 3.7).*

```
start_animation( a, b )
  . . .
enter( c )
  . . .
exit( a )
lower( b )
  . . .
enter( d )
lift( b )
  . . .
end_animation()
```

---

where each "..." represents one second of animation, results in this working set timeline:

$$\texttt{working\_set}(t) = \begin{cases} [\texttt{a},\texttt{b}] & \text{for } t \in (-\infty, 1) \\ [\texttt{a},\texttt{b},\texttt{c}] & \text{for } t \in [1, 2) \\ [\texttt{c},\texttt{b}] & \text{for } t \in [2, 3) \\ [\texttt{c},\texttt{d},\texttt{b}] & \text{for } t \in [3, \infty) \end{cases}$$

To render a frame of the animation, SLITHY first evaluates the working set function at the desired time t to determine which elements to render and in what order.

3.2.5 Partitioning animation objects

Frequently, presentation authors will want to break a long animation up into sections, inserting stops at particular points where the animation will pause, allowing the speaker to catch up or highlight details that could be missed. SLITHY makes it simple to split an animation up into pieces in this manner, by using the `pause()` command.

Here we show a simple seven-second animation that makes use of `pause()`:

```
start_animation()
set( x, 0.0 )
linear( 3.0, x, 1.0 )       # over 3 seconds, raise parameter x to 1
pause()
smooth( 4.0, x, 0.0 )       # over 4 seconds, lower it back to 0
end_animation()
```

In this example, the `end_animation()` function will return a list of two animation objects, one containing the animation from the beginning up to the `pause()`, and one containing the animation from the pause to the end.

Scripts may contain multiple pauses. For a script with k pauses, the `end_animation()` will return a list of $k+1$ animation objects—one containing the portion of the animation from the start to the first pause, one from each pause to the next, and one from the final pause to the end. In this way authors can create a series of animation objects which run seamlessly end-to-end without having to manually match up the parameter values in separate animation scripts.

Internally, SLITHY creates just one set of timelines for the entire animation sequence, as if the pauses were not present. This object is called the *base* animation object. The `pause()` command creates *virtual* animation objects that refer to this base object with offsets in time. In the above example, the script creates the base animation object A, which contains a timeline for the parameter `x`:

$$x_A(t) = \begin{cases} 0 & \text{for } t \in (-\infty, 0) \\ \text{linear}(0,0,3,1,t) & \text{for } t \in [0,3) \\ \text{smooth}(3,1,7,0,t) & \text{for } t \in [3,7) \\ 0 & \text{for } t \in [7,\infty) \end{cases}$$

The call to `end_animation()`, though, returns not A but two virtual objects derived from it, A_1 and A_2:

$$x_{A_1}(t) = \begin{cases} x_A(0) & \text{for } t \in (-\infty, 0) \\ x_A(t) & \text{for } t \in [0,3) \\ x_A(3) & \text{for } t \in [3,\infty) \end{cases}$$

$$x_{A_2}(t) = \begin{cases} x_A(3) & \text{for } t \in (-\infty, 0) \\ x_A(t+3) & \text{for } t \in [0,4) \\ x_A(7) & \text{for } t \in [4,\infty) \end{cases}$$

While this looks somewhat complicated written in function form, the implementation is quite simple. Each animation object has a `render()` method that accepts a time t and draws the frame. Normal animation objects render themselves by looping through the elements of the animation, determining the parameters for each by accessing the timelines, and drawing each element. Virtual animation objects contain neither a list of elements nor a set of timelines. They perform their rendering by simply calling the render method of the base animation object, offsetting and clamping the t value as appropriate.

3.2.6 Extensibility

One possible disadvantage of using SLITHY-style animation scripts compared to implementing the t-to-parameter mapping "by hand" as at the start of this section (page 52) is that the manual method might allow a wider variety of functions from t to a parameter value. As with parameterized diagrams, the manual method would allow arbitrary code to compute the mappings. With animation scripts the author is limited to whatever transition functions (such as `linear()` and `smooth()`) are built in to the system. In practice we have not found this to be a serious limitation, but we recognize that other users may have needs that go beyond these two simple interpolators.

Consequently, we have worked to make SLITHY's animation commands as extensible as possible. The `linear()` and `smooth()` "functions" are not really functions at all, but objects of the class `Transition`. Each class essentially encapsulates a function from the interval [0, 1] to [0, 1]; SLITHY's transition infrastructure handles all the scaling necessary to turn these into the correct interpolation. By writing new subclasses of `Transition`, users could add new interpolation methods to SLITHY, which would function just like `linear()` and `smooth()`. This mechanism is made possible by a feature of Python which allows objects with a "`__call__`" method to act as functions, so the following two lines are equivalent:

```
x(...)
x.__call__(...)
```

A similar technique is used in the building of undulators. Undulators are similar to transitions except that instead of changing a parameter to a new value and stopping, an undulator produces a repeating periodic pattern in the parameter's value. The built-in animation command `wave()` that varies a parameter sinusoidally is not a true function, but an instance of a subclass of the `Undulation` class. By writing new subclasses authors can extend SLITHY's command set.

3.3 Interactive objects

SLITHY-1 allowed a very limited form of interactivity: instead of displaying an animation object on the screen, the system could substitute an *interactive controller* that allowed a single parameterized

diagram to be manipulated interactively. The manipulation was done by user code that mapped input events (like keystrokes and mouse actions) to changes in diagram parameters.

This mechanism was enhanced and integrated more closely into the rest of the SLITHY-2 system. Now interactive objects are essentially animation objects whose timelines are constructed while they are being played. The interactive object can control an arbitrary set of animation elements, rather than a single parameterized diagram. The set of elements can be modified with the `enter()` and `exit()` functions, just as in an animation. In addition, the interactive object itself is no longer inserted into the presentation script in place of an animation object. Instead, interactive objects are added to animations with a special `Interactive` element. This allows interactive objects to coexist on the slide along with other pre-scripted parts of the animation.

In this section we will develop a simple interactive object that manipulates the clock diagram. Interactive objects are implemented as Python classes, which the SLITHY system will instantiate at run time.

```
class InteractiveClock(Controller):
    def create_objects( self ):
        self.d = Drawable( get_camera(), clock,
                           hour = 2, label = 'Seattle' )
        return self.d
```

The first line declares the class `InteractiveClock`, which inherits from the base class `Controller`. `Controller` is part of the SLITHY library; all interactive object classes are derived from this class.

Within the class we define methods (like `create_objects()`) that encode the class's behavior. All methods take as their first argument a reference to the object whose method is being invoked. It is analogous to the `this` pointer in C++, but in Python this object is traditionally called `self`. All references to the object's instance variable must be done via explicit references to `self`.

The `create_objects()` method is called by SLITHY when a new instance of the class is created; its purpose is to create the set of elements controlled by the interactive object. It corresponds to the part of an animation script above the call to `start_animation()`, where elements like `Drawable` and `Text` and `Fill` are created. In this example we create just a single `Drawable`

element that displays the clock. The `create_objects()` method returns the objects that should initially appear on the canvas. Note that this method also stores a reference to the element in an instance variable (`self.d`) so that the element can be manipulated inside other methods.

The remaining methods use animation commands to edit the parameter timelines. When SLITHY receives an input event (such as a keystroke), it positions the interactive object's time cursor at the current playback time and invokes the method. Here we will add a `key()` method to the `InteractiveClock` class:

```
    def key( self, k, x, y, m ):
        if k == 'a':
            m = get( self.d.minute )
            smooth( 1.0, self.d.minute, m+60 )
            set( self.d.minute, m )
            set( self.d.hour, get(self.d.hour)+1 )
```

This method receives five arguments: the standard `self` reference, which key was pressed, the x and y coordinates of the mouse cursor when the key was pressed, and a list of the modifier keys ("shift" and/or "control") which were down at the time of the keypress. In this example, when the 'a' key is pressed, the time shown on the clock is advanced by one hour in a one-second animation.

Interactive objects can respond to mouse as well as keyboard events. The drawing library has a hit detection mechanism to make it easier to detect mouse clicks on specific objects. This mechanism uses the OpenGL depth buffer to store an object ID that can be queried later. Object IDs are nonnegative integers; the exact range available depends on the underlying OpenGL implementation but typically values in the range 0–16383 are available. The graphics library has a function to set the "current drawing ID," in much the same way that the current drawing color is set. To use this mechanism we will make a small change to the `clock()` diagram function, to draw the minute hand using object ID #1:

```
def clock( . . . ):

    . . .

    push()                          # draw the minute hand
    color( red )
```

```
id( 1 )                          # draw the hand in ID #1
rotate( -minute * 6 )
fill( hand )
pop()

. . .
```

Now whenever the clock is drawn, the pixels covered by the minute hand will get ID #1 in the invisible object ID buffer. We can query the buffer from within the interactive object's methods:

```
def mousedown( self, x, y, m ):
    i, = query_id( x, y )
    if i == 1:
        smooth( 0.5, self.d.minute, 5, rel=1 )
```

Here the `mousedown()` method, which is called whenever the mouse button is pushed, is used to advance the clock by five minutes whenever the user clicks on the minute hand. This determination is made by querying the ID buffer at the location of the mouse click. Multiple locations can be queried in a single call to `query_id()`; the return value is a tuple of object IDs. A call that queries two locations might look like this:

```
id_a, id_b = query_id( xa, ya, xb, yb )
```

(A quirk of Python syntax requires the presence of a comma when the tuple returned is a singleton, as in the `mousedown` example above.)

Some examples of interactive objects that use this enhanced functionality are shown in Section 5.2. Appendix B shows the implementation of a more elaborate interactive object.

3.4 Interactive tools for authoring

Section 2.4.1 documents our various attempts to create interactive tools for authoring parameterized diagrams and animation scripts. Our approach there was to seek what we called "power assist" tools—graphical tools that would aid the author in writing SLITHY code, and would be applicable for a wide variety of subjects. As we noted, this effort was largely unsuccessful. We encountered

two major problems in writing tools to automatically edit user-written code: understanding the input (without putting onerous restrictions on it), and producing output in a form that meshed with the author's conception of the code's structure.

We can imagine a second type of graphical authoring tool that bypasses both of these problems. These tools have graphical interaction as their *only* input mechanism. They output SLITHY code, but the code is not intended for subsequent modification by the user (except through the tool). PowerPoint can be thought of as a tool like this for a more restricted domain: the space of animations where otherwise static elements fly onto the screen using one of a predefined set of motions. SLITHY explores the other end of the design spectrum: a much wider variety of possible animations, at the expense of losing WYSIWYG graphical input and editing.

There may be a kind of "sweet spot" that lies between these two extremes. By choosing a small domain, we can limit the range of animations enough to make interactive specification feasible, while still producing useful, content-rich animations. For instance, consider the example of Figure 5.3 on page 90. We can imagine a bar-chart tool that lets us put in data and animate the graph in a limited number of meaningful ways. The output would be a SLITHY animation, ready to be integrated into a presentation.

In the ideal case, we can imagine assembling a library of these small tools that cover a wide range of presentation topics. One tool might be used for producing ordinary bulleted-list slides, another for producing animated data plots, a third for showing still images. (Even with still images there are opportunities for useful animation: zooming in for closeups, labeling and captioning, etc.) Hand-authoring of SLITHY code would be limited to the subjects so specialized that no tool covers them—which, for some presentations, could be an empty collection.

While this grand vision remains for the moment just that – a vision – we have produced simple prototype implementations of tools that work in this manner. The first is a tool for creating still image slideshows, inspired by the work of documentary filmmaker Ken Burns. Our tool allows the user to load in images and to interactively specify zooms and pans over them and animated transitions between them. The output is a complete SLITHY animation. An example of using this

Figure 3.13 *The left column shows screenshots of the prototype image slideshow tool. The user can load images, optionally select a subregion of the image to show (as in the meddle row) and specify the type and duration of the transition. The right side shows frames from the resulting* SLITHY *animation.*

tool is shown in Figure 3.13.

A second prototype tool produces simple line chart animations. The user can use the controls

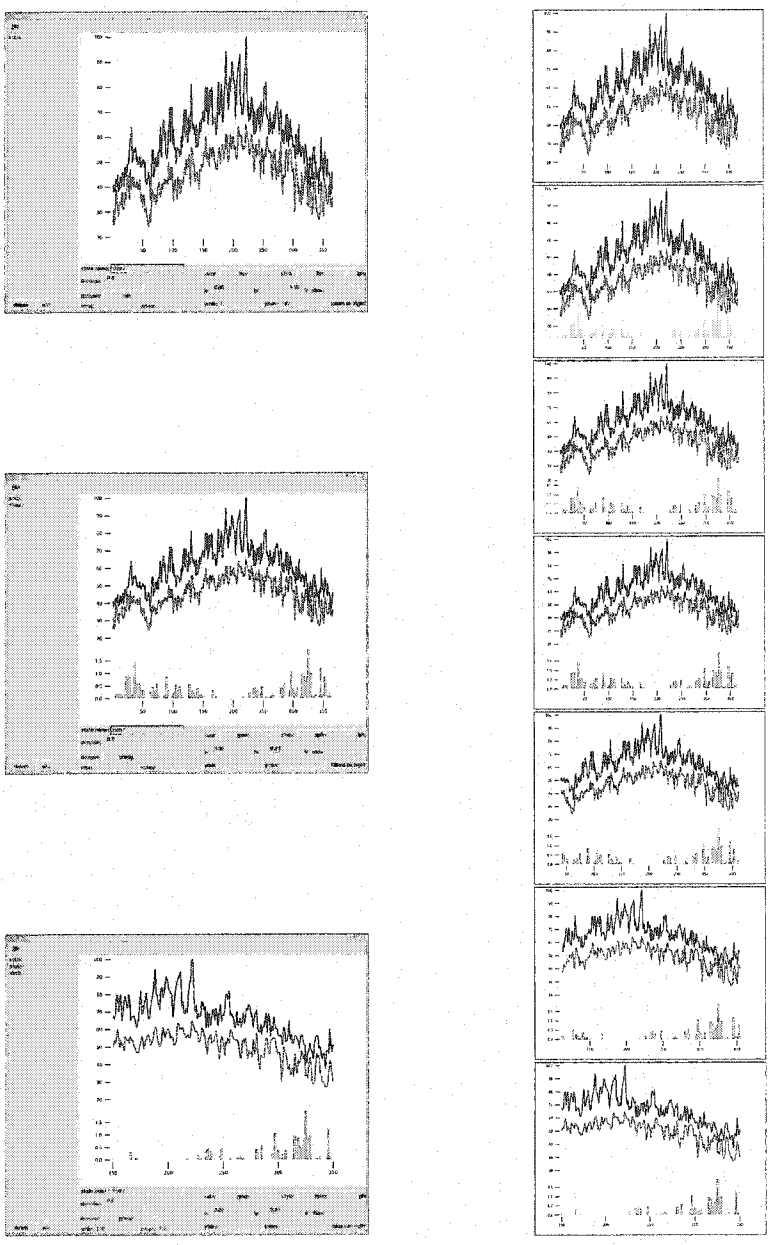


Figure 3.14 *The left column shows screenshots of the prototype GUI line-chart tool. Here the user has interactively arranged the data in three different ways. The right side shows frames from the resulting* SLITHY *animation, which interpolates smoothly between successive states.*

at the bottom of the window to specify the appearance of the chart. Successive appearance states are saved in the list on the left side, and the tool generates SLITHY code to animatedly transition between the states. Screenshots of the tool and its output are shown in Figure 3.14.

Chapter 4

ANIMATION PRINCIPLES

When desktop publishing and laser printers started to become more common, displacing the typewriter, the immediate result was not better-looking documents. Confronted by dozens of type-setting options, people simply chose them all, even within a single document. The message was not "look at my content," but "look what I can do with my software!" Today, too many presentations use animation with similar results. Animation can enhance the content, or it can be visually distracting. Over the course of this project we have accumulated a fair amount of experience in creating animated presentations. By summarizing the results of our experience in making animated presentations as a set of general principles, we hope to encourage good animation, leading to more engaging and informative presentations. It is important to remember that these are not meant as *rules,* but more as a set of defaults. Like most rules, the principles here should at times be judiciously broken.

Use motion economically. When we first started adding animation to presentations, we naturally tried to apply traditional animation principles such as *squash and stretch* and *exaggeration*, with generally poor results. These principles are intended to turn a drawing (or a rendered model) into a *character* in the mind of the viewer. While this liveliness is desirable in animation made to entertain, it is distracting when the goal is to inform. The audience is drawn away from the speaker and becomes focused on the animation itself, wondering what interesting thing is going to happen on the screen next. We had better results when motion was as economical as possible. Figure 4.1 contrasts a character animation-style motion (column (a)) with a simpler, economical motion (column (b)).

Other classical animation principles such as *anticipation* and *staging* are employed to draw the audience's attention to the right part of the screen at the right time. In presentations, though, it is usually better to do this in a way that maintains a distinction between the attention-getting animation and the action the audience needs to see. If something interesting is about to happen in a particular

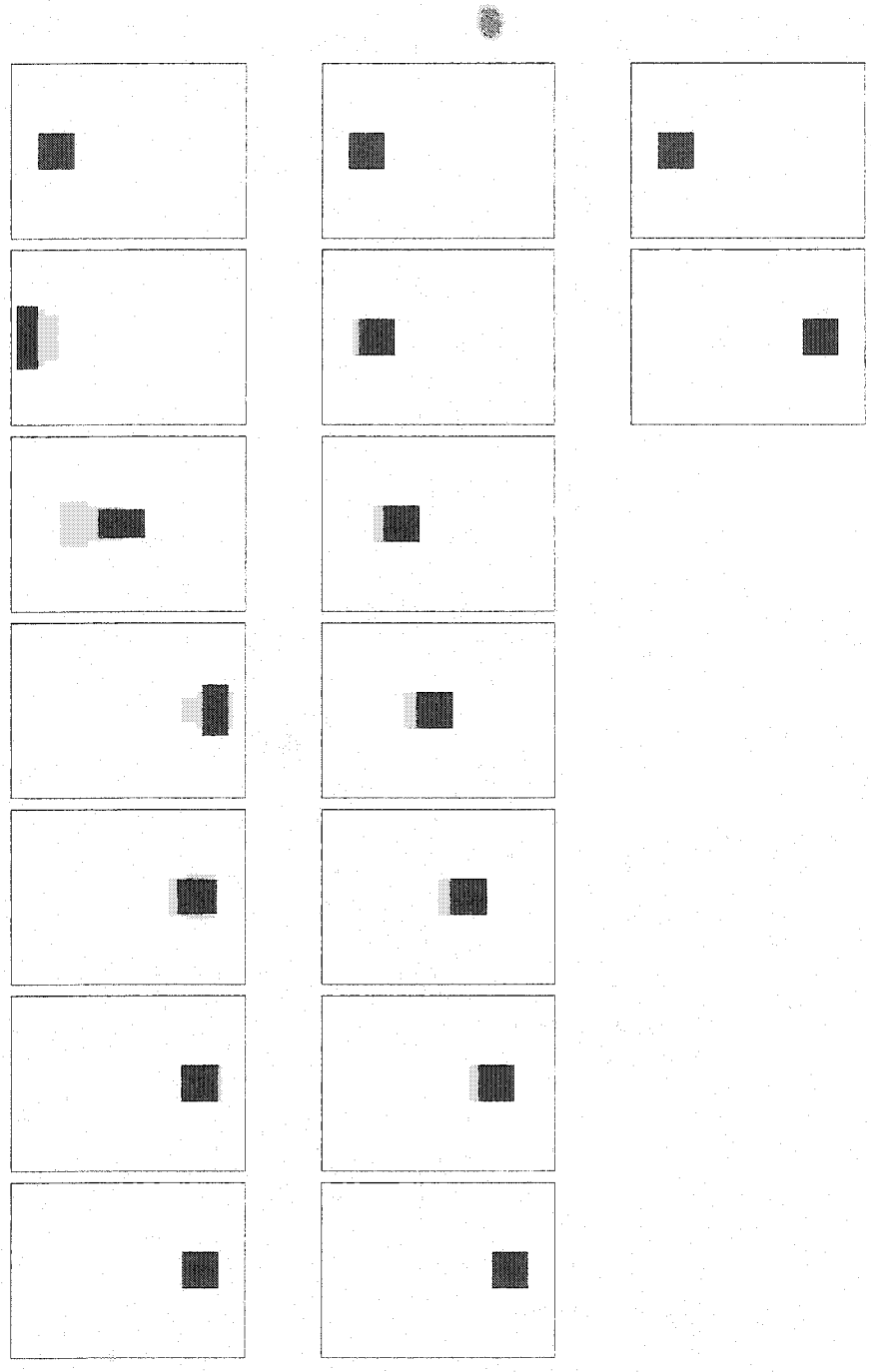


Figure 4.1 *Contrasting three ways of moving a square from one side of the screen to the other: (a) using traditional animation principles of anticipation and squash and stretch, (b) using an economical slow-in-slow-out motion, and (c) instantaneously, with no animation. (The ghosted images are not shown by* SLITHY, *but are added here to suggest the motion.)*

section of a figure, that section should be highlighted by a color change, a superimposed arrow, or even the speaker manually pointing at it with the cursor—anything that can't be confused with the interesting action itself. There is no need to add an artificial buildup to the start of the action in order to draw attention to it.

While we do caution against using overexaggerated motions, we also suggest avoiding instantaneous change. We have come to believe that smooth transitions – even something as simple as a crossfade – should be the standard way of getting information on and off the screen. When jump cuts are used on a small scale, as in adding a single bullet point or subtly modifying a diagram, it becomes easy for the viewers to miss the change. This principle is known in film editing as well [40]. In presentations, it causes the audience to focus intently on the screen, to make sure that nothing important is missed. Figure 4.1 contrasts an instantaneous change with a simple smooth motion. Even very brief transitions are better than sudden cuts at creating a feeling of continuity, which lets the focus move easily from the screen to the speaker and back as needed.

Reinforce structure with transitions. An advantage of using subtle transitions is that it increases the impact of the more showy effects when they *are* used. A presentation in which every single bullet point tap-dances its way onto the screen is a presentation where the audience quickly learns to ignore the tap-dancing. Used carefully, transitions can reinforce the structure of the presentation. A section can be visually tied together with simple transitions. Using a more dramatic effect to move to a new section will then create a visual break, subtly punctuating the visual half of the talk as the speaker punctuates the verbal half. Figure 4.2 shows a simple application of this principle.

Good and Bederson [27] call this effect the "sense of semantic distance." In their system static PowerPoint slides are arranged on a large canvas at various scales; the transitions from one slide to the next are then pans and zooms of the camera across this canvas. The natural way of laying out slides in clusters by topic then leads to small transitions between related slides and longer, sweeping motions between more distant sections. Our recommendation can be thought of as a generalization of this effect, where the concept of a "bigger" movement is extended to more than simple Euclidean distance.

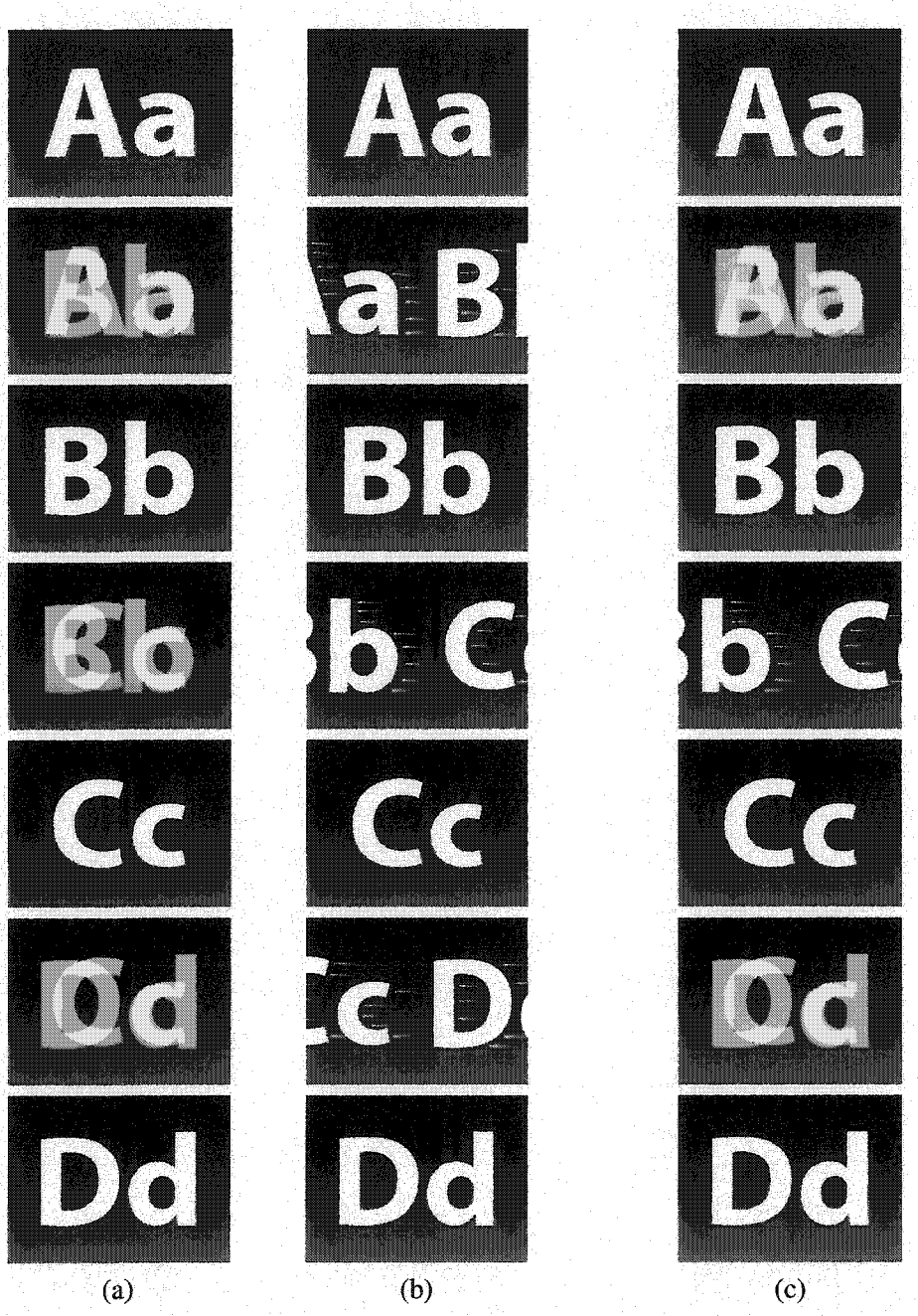


Figure 4.2 *Used uniformly, both a crossfade transition (column (a)) and a sliding transition (column (b)) are simply noise, adding little to the content of the talk. Different transitions can be used meaningfully, though, by choosing each transition to reflect the structure of the talk. In column (c), the use of a different transition creates a visual break, dividing the four slides into two distinct groups—the Aa/Bb group, and the Cc/Dd group. (Lines have been manually drawn on some of the still images to suggest motion.)*

Create a large virtual canvas. Often when creating a presentation it seems like there is not enough room on the slide to include everything the author thinks is important. Animated panning and zooming can be used to naturally increase the effective real estate of the screen. A figure that slides off one side of the screen remains more "visible" in the mind's eye of the audience than one that simply blinks out of existence. This effect is supported by psychological research: Dillon *et al.* [17] summarize a number of studies supporting a positive correlation between memory for location and memory for content in both text and electronic documents.

As a concrete example of this, imagine a situation in which the presenter is comparing items. Comparison is a very common thing to do in presentations—to show "before" vs. "after," or "standard method" vs. "our new method," etc. A typical sequence would go something like this: show each item by itself, in order to give detailed description. Then, go to a comparison slide that shows all the items side-by-side, to highlight the differences among them. A set of slides that follow this pattern is shown in column (b) of Figure 4.3. It takes mental effort on the part of the audience and spoken explanation by the presenter to make the connection between the item on the first slide (and its corresponding narrative description) to what is shown on the third slide. By simply replacing the instantaneous flips with a pan over to show the second item, then a zoom out to show both, we very naturally give a strong impression of comparing two things side-by-side. This kind of sequence is shown in Figure 4.3(a). This sequence also illustrates the previous principle: by replacing the abrupt jump cuts between slides with smooth motion, we reinforce the idea that this part of the presentation is one interrelated section, rather than four disconnected slides.

Smoothly expand and compress detail. A closely related principle is that of using animation to expand and compress detail. Earlier we suggested using camera pans and zooms to give the impression of the screen as a window onto a very large space. It is also effective to use it as a kind of magnifying glass for examining figures at a variety of scales. In this way the presentation can easily fill the screen with the active portion of a diagram, shutting out the parts not relevant to what the speaker is saying. Figure 4.4 contrasts two versions of an animation: one where scale changes are used to fill the screen with the region of highest interest, and one where a single intermediate

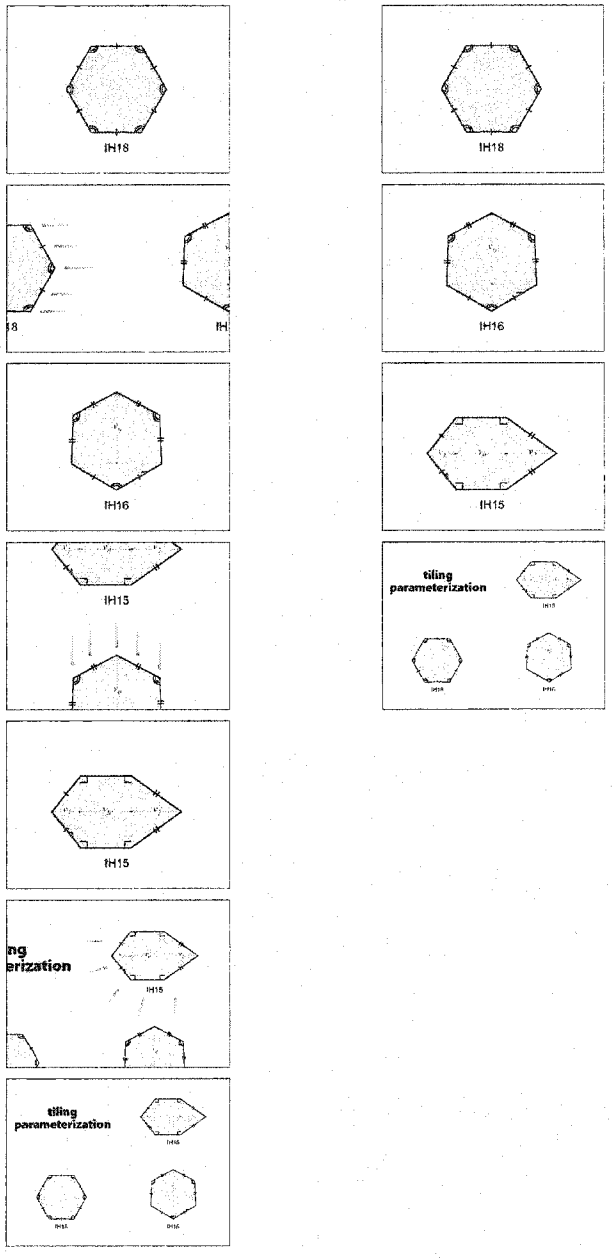


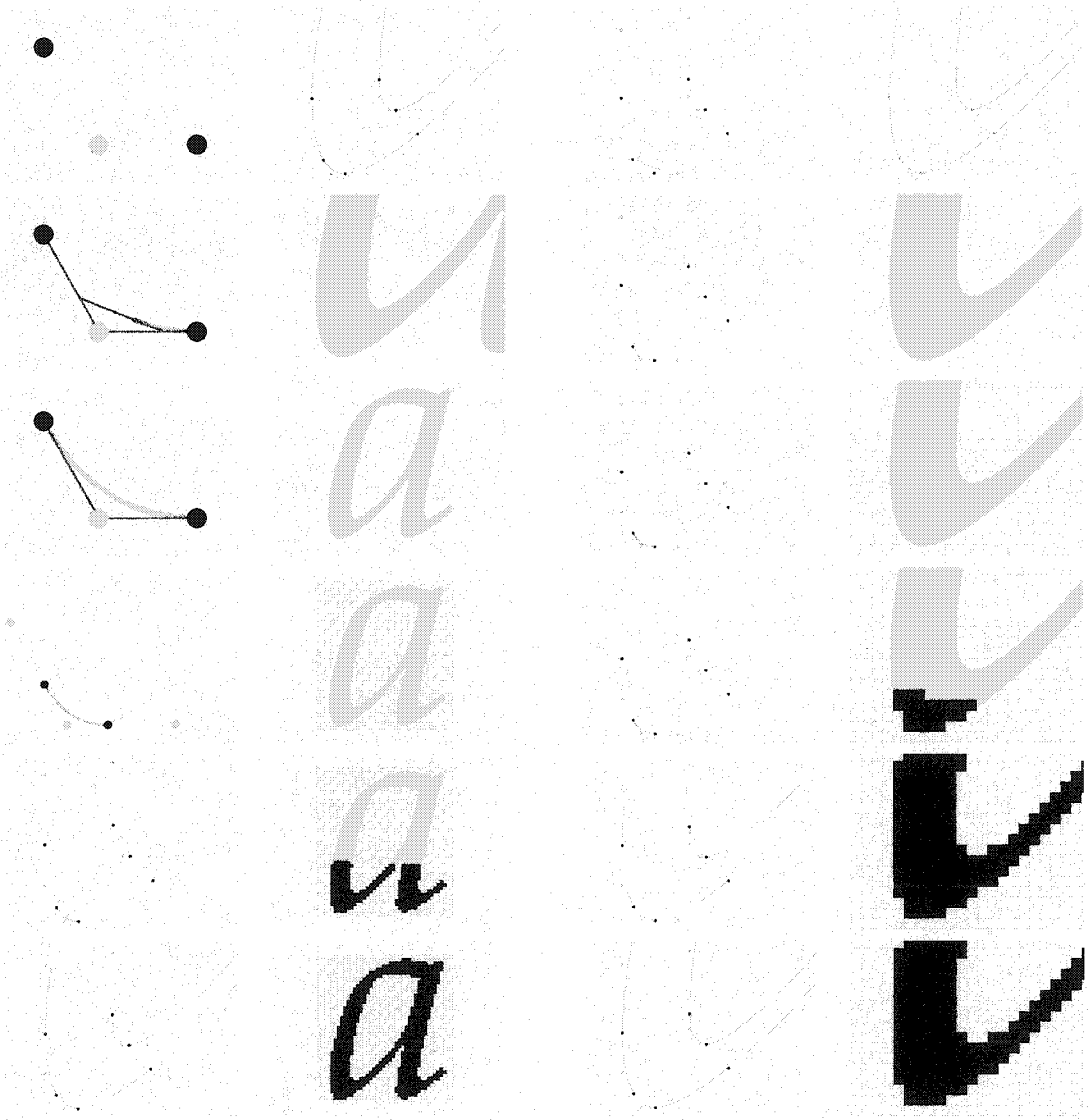
Figure 4.3 *In column (a), motion is used to suggest the three colored shapes lie on a virtual canvas larger than the slide. (Lines have been manually drawn on these still images to suggest motion.) When the camera pulls back to show all three, the arrangement of the objects in space makes it immediately clear which one is which. In column (b), where the screen simply blinks from one picture to the next, viewers must refer to the labels to establish the correspondence.*

scale is used throughout.

Another alternative to animated zooming is to jump instantly between the different scales. Figure 4.5 illustrates this alternative. It typically requires explanation by the presenter and effort by the audience to make the mental links between the different views. While it may not be too taxing to follow one scale jump, covering a complex figure may require examination of several locations at multiple levels of detail, and it is easy to lose track of the relationship between what is being shown on the screen and the figure as a whole.

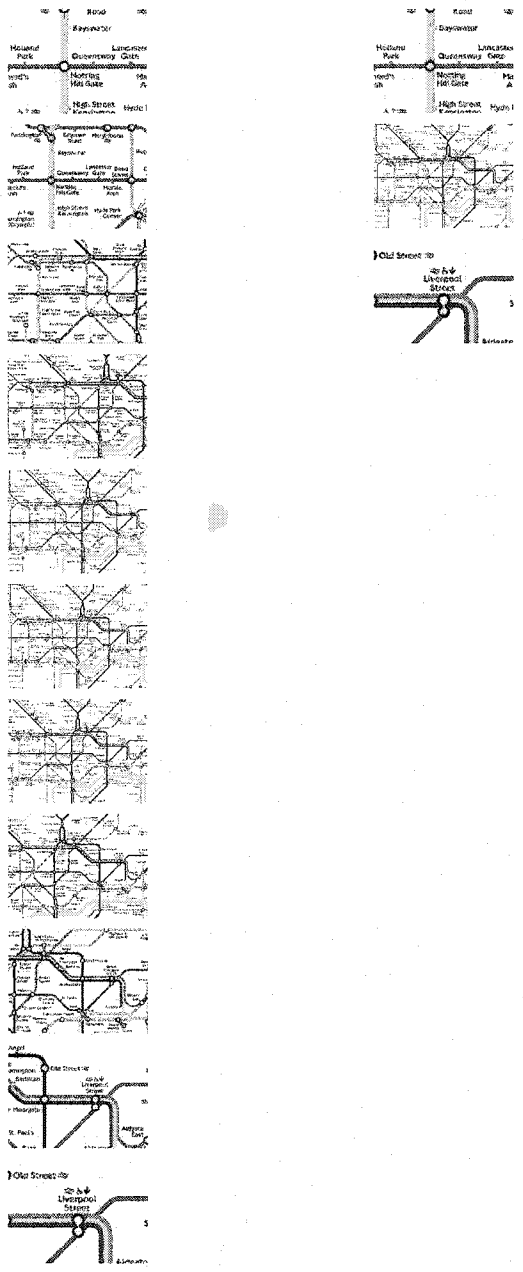
Animation makes this kind of navigation much easier to follow. Linking the different views with smooth, continuous camera motion takes advantage of the viewers' natural spatial abilities, with less need for artificial visual mechanisms like superimposed highlight rectangles and verbal explanation from the presenter. The audience does not need to be told that the display is about to focus in on one section because they can *see it happen.* Zooming in to emphasize detail can be done much more often because there is less overhead involved in maintaining context.

Manage complexity through overlays. Panning and zooming allow attention to be focused on one spatial region of a figure, keeping unnecessary detail off the screen while providing context. Instead of breaking a diagram into pieces spatially, one can imagine instead slicing along an axis of "complexity," separating detail into layers that become visible only as required. This idea of accumulating complexity through layering does not require computer graphics to implement—overhead transparencies work very well for this kind of sequence, where different layers are printed on separate sheets and are stacked up on the projector. The use of electronic projection and sequences of perfectly aligned slides made through cutting and pasting is arguably *less* effective than the older technology. Now, instead of seeing the presenter physically add a transparency to the stack and watching the screen as the new sheet is aligned, we often see new elements blink up onto the screen instantly, indistinguishable from what was there before. This is one area where PowerPoint's animated transitions could be used for great effect but too often are not. A simple animated transition such as a quick fade-in or a small sliding motion can provide a subtle and effective cue for differentiating the layers of information.



(a) (b)

Figure 4.4 *Using animation to expand and compress detail. The figure has three levels of detail: a closeup on the three points defining one curve segment, and intermediate level showing curves connecting end-to-end, and an overall view showing the whole character as it is filled with pixels. In part (a), animated zooming is used to smoothly connect the three levels, so each can be shown full-screen. In part (b), the animated content is the same; only the zoom is missing. Now the screen is too cramped for the final part of the sequence, while it is hard to see the detail of the initial part.*



(a) (b)

Figure 4.5 *Comparing animated zooming to the use of multiple scales without animation. Part (a) shows an animated transition from one point on a map to another. The animation helps orient the viewer in the space and gives a sense of distance. Part (b) shows the same map but with instantaneous changes between the different scales. It requires much more effort on the part of the viewer to determine how the different views are linked.*

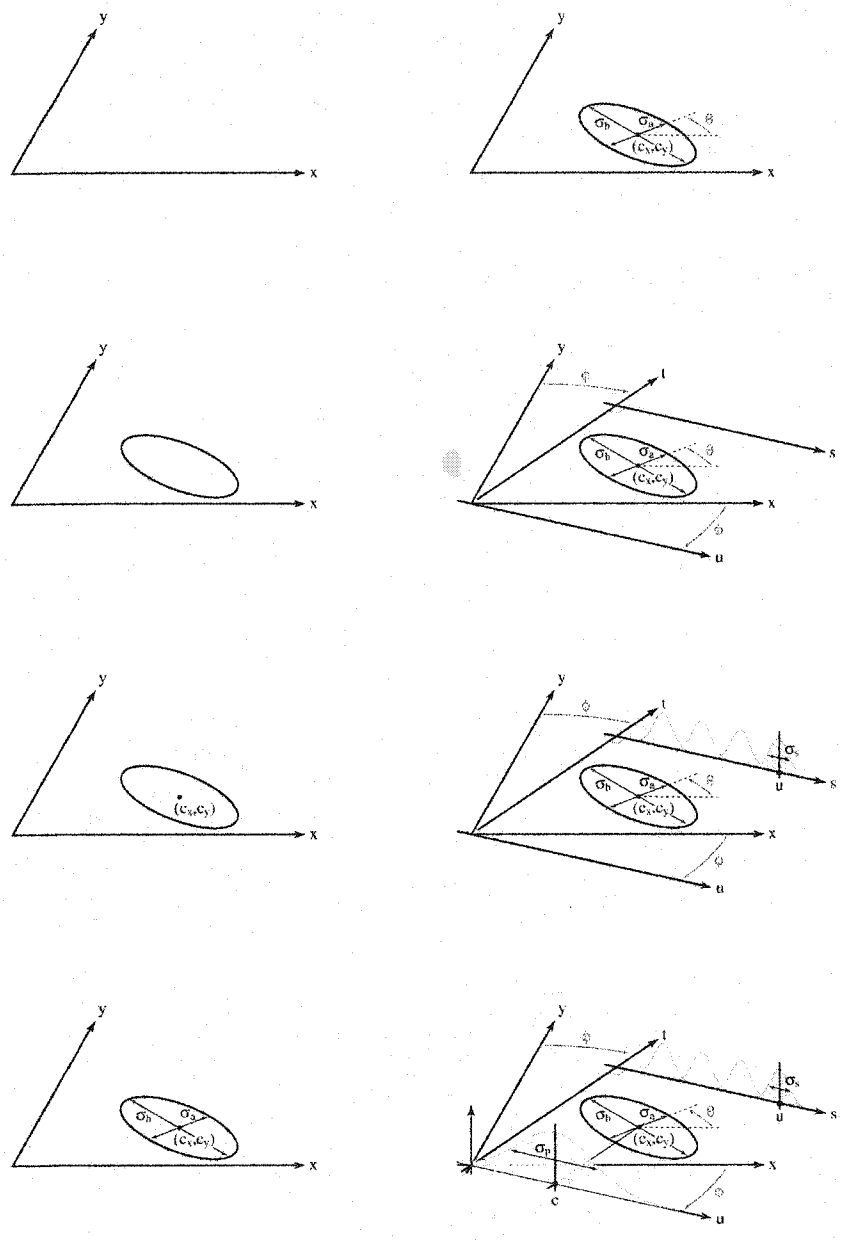


Figure 4.6 *Building up a complex diagram gradually by means of successive overlays. This technique predates the use of computers for projection, but can be implemented effectively in* SLITHY *with motion and fading.*

Do one thing at a time. The last two principles are useful for focusing attention on a single, important part of a figure. In general, doing one thing at a time is a useful principle for making animation that explains rather than distracts. Animations where many things are changing at once give an overall impression of the change, but make it difficult to concentrate on any single part. We have had the best results when complex diagrams are animated relatively slowly and with frequent pauses, as in Figure 4.7(b), so that the animations track the speaker's words. The technological advances in slide creation and projection have made it increasingly common for the presenter's words to take a back seat to the elaborate visuals.[1] The extensive use of animation threatens to make this effect worse. We believe it is important to treat any visuals – animated or otherwise – as an accompaniment to the *talk,* rather than the other way around. The presenter can only talk about one thing at a time; the animation on the screen should match.

Reinforce animation with narration. The idea of using animation simultaneously with narration is a useful one. In our own presentations we have noticed a frequent impulse to try and make two points at once—to have the animation showing one thing on the screen while we talk about something else. Even though the two topics are usually closely related, it is very difficult to follow both threads, and usually the result is that neither point gets made very effectively. When used simultaneously, animation and narration should reinforce each other. The speaker should describe what is happening on the screen as it happens. To make a point that isn't illustrated, a pause in the on-screen motion will naturally shift attention back to the presenter. The effectiveness of narration in concert with animation has been demonstrated in a series of studies by Mayer and Anderson [35, 36].

Distinguish dynamics from transitions. Our final animation principle also deals with reducing the potential for confusing the audience. We can divide presentation animations into two major classes: dynamics and transitions. *Dynamics* refers to perhaps the most natural use of animation: depicting change over time in a real-world process. This change could be physical, such as a moving

---

[1]The worst incarnation of this style is the all-too-common presentation where the speaker puts up one text-filled slide after another and simply reads them aloud.

(a) (b)

Figure 4.7 *In this sequence, a series of dynamic animations, fades, and camera motions elaborate a simple diagram into a filled character. In version (a), many of the individual actions are overlapped as in traditional character animation. The effect is smooth and continuous, but difficult to follow. We recommend separating distinct actions, as in version (b), so that each piece can be seen on its own, and natural pause points are created where speaker can catch up with the animation.*

illustration of a mechanical system, or abstract, such as data flowing through a computer algorithm. The essential notion is that the animation is used to show some kind of change in the material being presented. *Transitions* is the term we use to capture all the other uses of animation—using it to highlight, to draw attention, to move the talk from one topic to the next. Here, the animation serves to help guide the audience through the presentation itself.

We have found it important to make sure that there is a clear distinction between dynamic animations and transition animations. It is very easy to create animation that can be misinterpreted. For example, one of our SLITHY-1 users was using the system to prepare a talk on a technique for simulating the motion of nonrigid bodies. He wanted to contrast between two different states of his system and had created an animated transition between the two illustrations. Because the subject of the talk was a system for simulating motion, viewers were often confused by the transition, thinking that the motion they were seeing represented the output of the simulation. Fortunately, this problem was identified before the final presentation: replacing the confusing motion transition with a simple crossfade resolved the ambiguity, making it clear that the sequence was showing two static states rather than an actual motion. A recreation of the sequence and its alternatives appears in Figure 4.8.

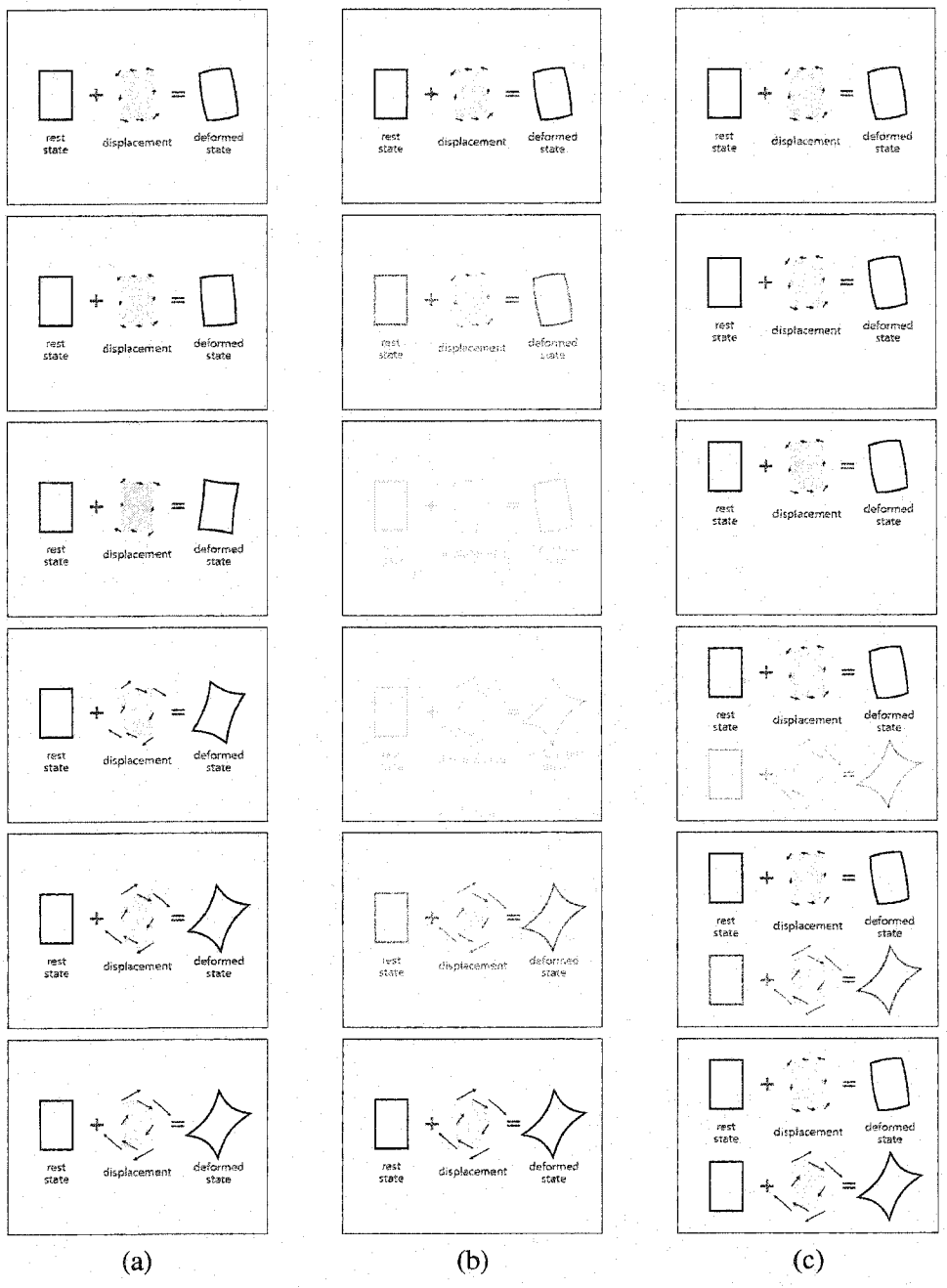


Figure 4.8 *Each column shows a transition from one state of the diagram to another. Because the topic of this presentation was the dynamics of nonrigid bodies, it was easy to mistake the transition of column (a) for output of the system, rather than a simple transition between two static diagrams. In column (b), the motion has been replaced with a simple fade, eliminating the ambiguity. Another option would be to slide the diagram up to make room for a second copy, allowing the comparison to be done directly.*

Chapter 5

EXAMPLES

In this chapter we will show some animated sequences created using SLITHY. While paper is not the ideal medium for showing the output of SLITHY– these were, after all, created to be shown in motion, accompanied by a live speaker – we hope that these examples will illustrate some of the possibilities of using meaningful animation effectively in a presentation setting.

Our first example is Figure 5.1, a visual proof of the Pythagorean Theorem, similar to that used in the study of Thompson and Riding [58]. Squares have been constructed on each of the three sides of a right triangle, and we want to show that the green square on the hypotenuse is equal in area to the sum of the other two squares. First, an altitude of the triangle is constructed, splitting the green square into two rectangles. Then three area-preserving transformations: a shear, a rotation, and another shear are used to transform the yellow rectangle into the blue square. A second shear–rotation–shear sequence lines up the purple rectangle with the red square.

This animation is based on a single diagram, shown in Figure 5.2. This diagram has nine parameters. The `a` and `b` parameters control the lengths of the legs of the triangle (and thus the size of the squares). The opacity of the altitude line and the length of its extension across the hypotenuse square are controlled by `linealpha` and `lineextend`, respectively. The opacities of the yellow and purple shapes are jointly controlled by the `subdivide` parameter; note that in Figure 5.1 the animator has left these shapes at least partially transparent throughout the animation so that the original squares can be seen underneath. The `textlabel` parameter controls the opacity of the labels on the squares. The `cam` parameter moves the diagram's camera; it is fixed in the animation of Figure 5.1, though other sequences that use this diagram make use of it. The last two parameters, `slidea` and `slideb`, have the most complex implementations. They control the transformations of the yellow and purple shapes, respectively. Each ranges from zero to three. The first shear takes

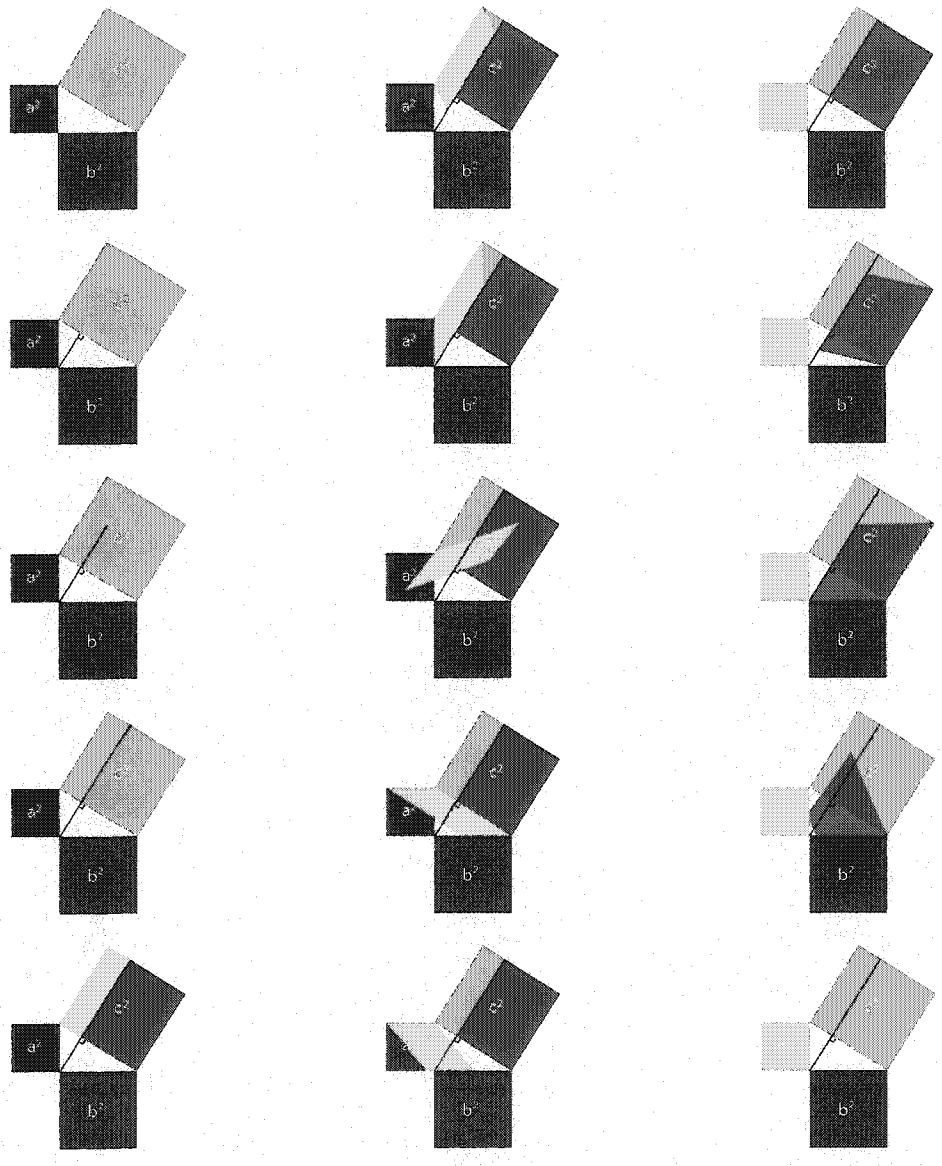
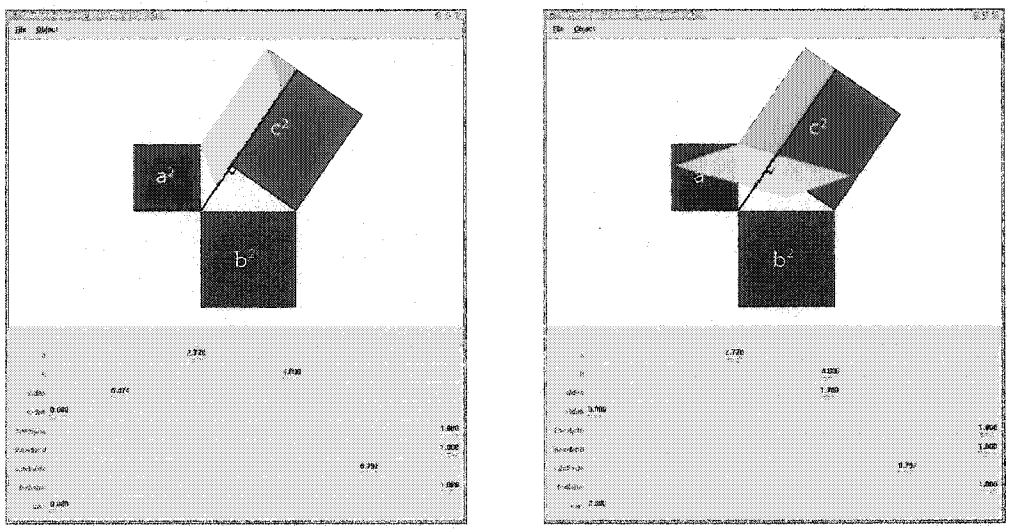


Figure 5.1 *A* SLITHY *sequence that illustrates a proof of the Pythagorean Theorem based on shears and rotations.*



(a) (b)

Figure 5.2 *The diagram used to create the Pythagorean Theorem animation of Figure 5.1, shown in the test window. Parts (a) and (b) illustrate two different settings of the* `slidea` *parameter.*

place over the interval [0, 1], the rotation over the interval [1, 2], and the second shear over the interval [2, 3]. Figure 5.2 shows this diagram inside SLITHY's object tester; the two screenshots differ only in their setting of the `slidea` parameter.

The last two parameters are good examples of how parameterized diagrams are used to encapsulate complexity: even though this three-step transformation is graphically complex, it can be expressed as a function of a single parameter. The diagram object allows us to express that function once and then animate at a higher level—to change the timing or order of the animation we only have to change how we manipulate the `slidea` and `slideb` parameters, which abstractly represent all the low-level graphical manipulations.

Figure 5.3 shows an example of using animation in a business context. The sequence begins with a chart of quarterly revenue, introduced one year at a time. Next the data slides out to the left, leaving only the last three bars, which are then broken out by company division. Some of the divisions fade to gray, leaving two particular ones highlighted. Next the graph transforms from showing revenue in dollars to percentage of the total: the y-axis labels fade out and are replaced while the bars themselves stretch out to the full height of the graph. Labels are then introduced,

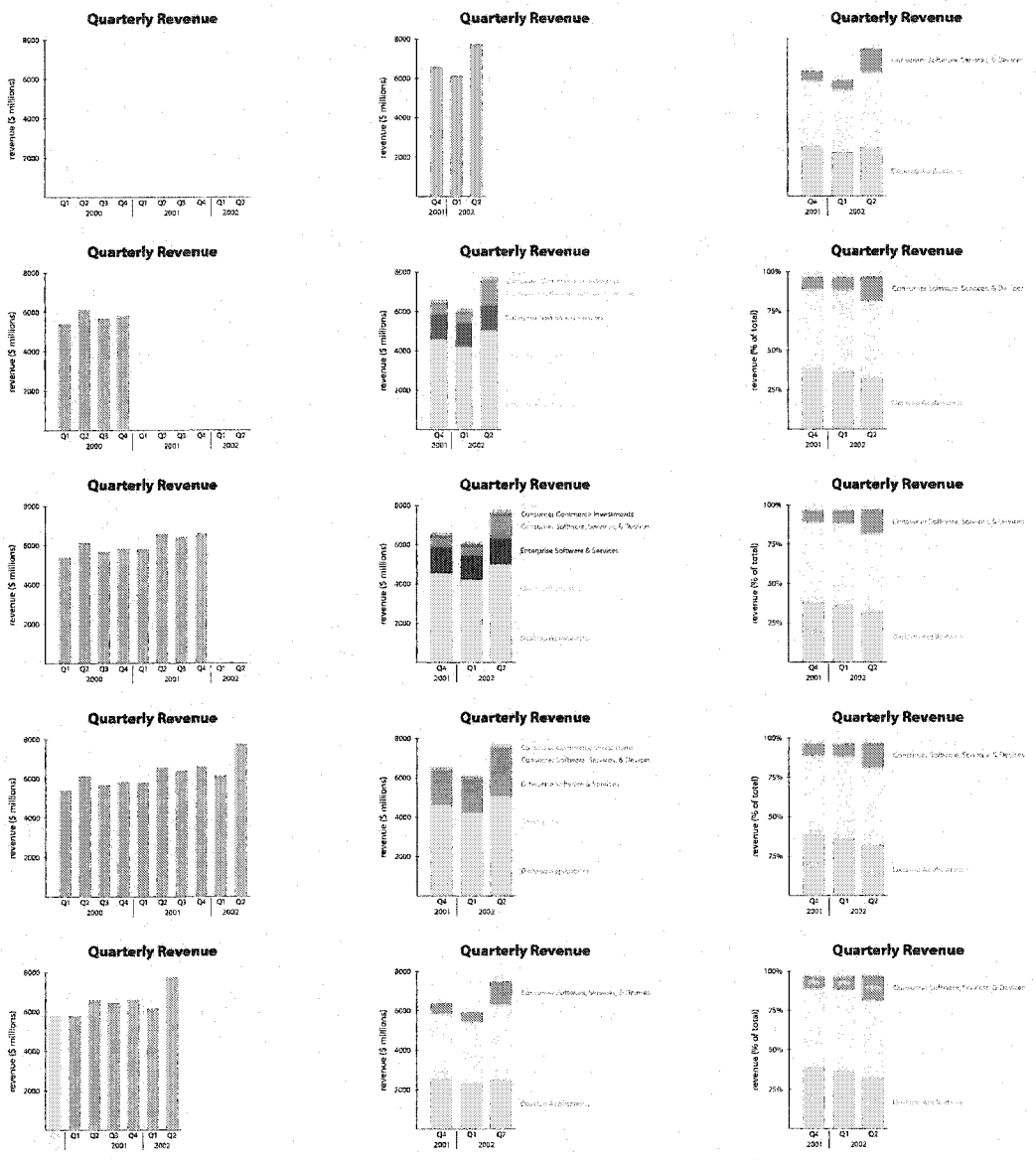


Figure 5.3 A SLITHY *sequence that uses an animated graph to illustrate a set of financial data.*

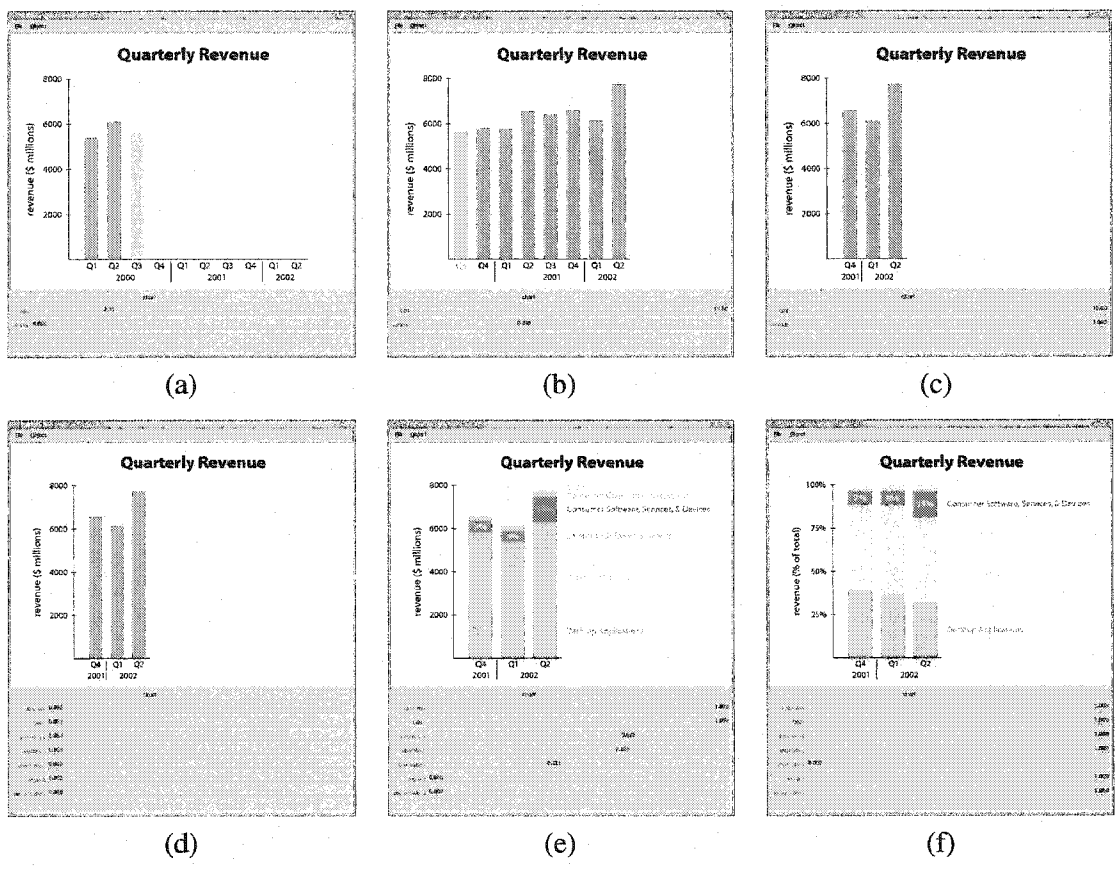


Figure 5.4 *Each row shows one of the parameterized diagrams used in the creation of the animation of Figure 5.3. The diagram of parts (a)–(c) has just two controls: one for the number of bars to draw and one to slide out all but the last three bars. The second diagram has a different set of parameters, which draw only the last three bars but in a wider variety of styles: colored to show a breakdown by division, as percentage of the total, with labels, etc.*

underscoring how the green division is slowly decreasing in percentage while the blue division has jumped up sharply.

In this example, financial data is being used to tell a story, just as characters are used to tell a story in animation made for entertainment. Without animation, this sequence would be more like a flipbook of graphs, forcing the speaker to explain verbally how each graph is connected to the next. Animation shows these connections naturally, freeing the presenter to focus on explaining the content rather than the visualization.

The implementation of this talk also illustrates one of the advantages of a programming-based authoring mechanism mentioned by the authors of Menv—*precision* [50]. This animation actually makes use of two different parameterized diagrams, as shown in Figure 5.4. One is used to draw the first half of the sequence, up to the point where there are just three blue bars (Figure 5.3, top center). At that point the second parameterized diagram is substituted. It only knows how to draw the last three bars, but has controls for breaking them down by division, showing them as percentages, adding labels, etc. Since we are creating these pictures with code rather than interaction, it is a simple matter to make the diagrams line up exactly so that the transition is seamless. Note how parts (c) and (d) of Figure 5.4 show identical pictures, even though they are drawn by different diagram functions with different parameter sets.

5.1 Real-world presentations

The next set of examples are taken from the presentations created by the four users we recruited to try out SLITHY. These were made using the first implementation of SLITHY, but should be easily portable to the latest version. They illustrate a number of our animation principles as well as some of the situations that led to improvements in the design of SLITHY.

Figure 5.5 has three examples of animated diagrams. Part (a), from the work of Liu and Popović [33], shows the cumulative effect of rotations on a 3D object (implemented in SLITHY with an OpenGL diagram). Part (b) shows how a deformed object is represented as a rest state plus a displacement. As the rest state is varied, the displacements change to compensate. This diagram

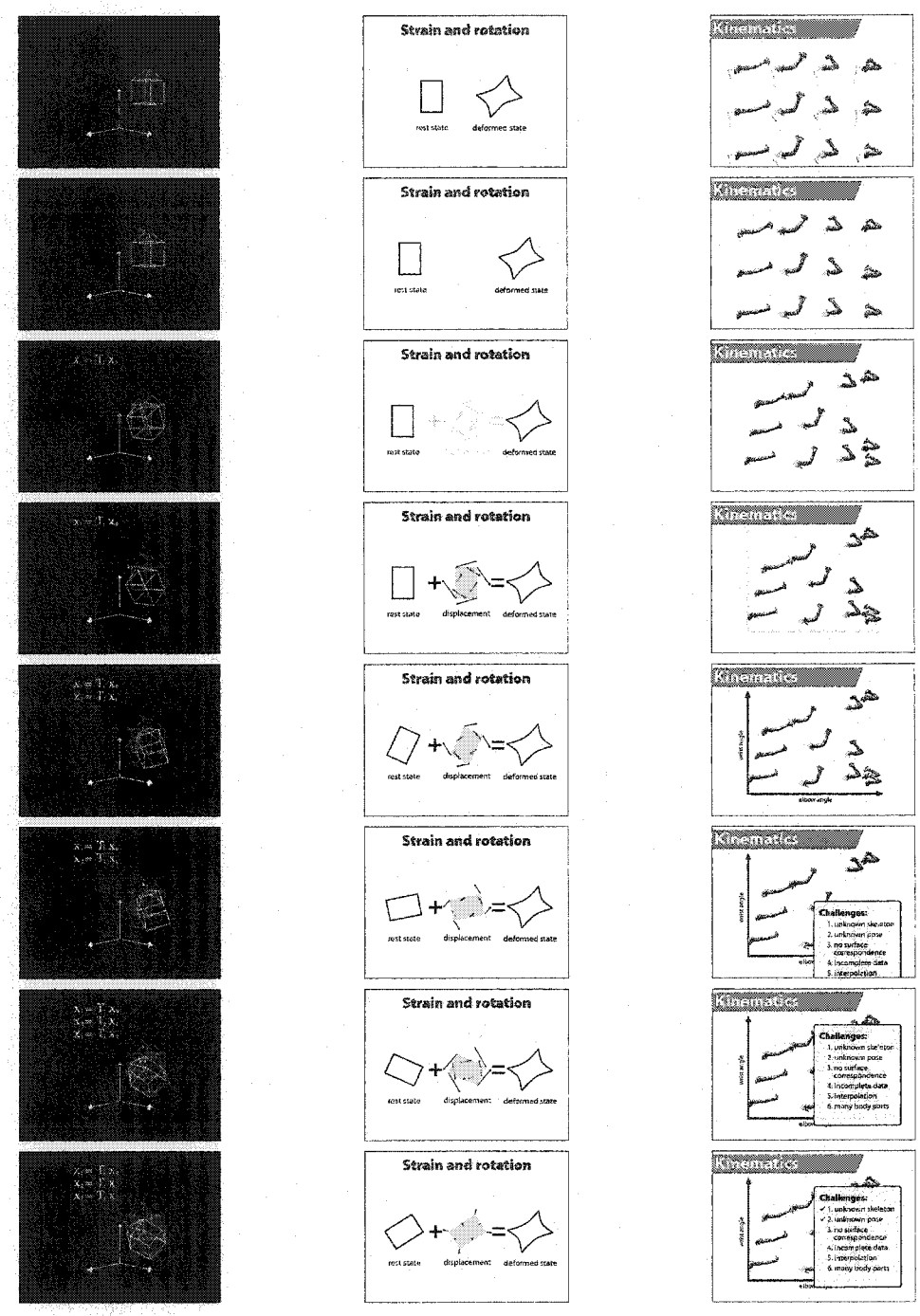


Figure 5.5 *Three examples of animation being used to show actual movement in a physical or virtual space.*

is a 2D representation of a technique that is actually carried out in 3D (in the work of Capell *et al.* [8]). The final example, column (c), is a more abstract example that shows how Allen *et al.* [3] take twelve sets of input data (range scans of a human arm) that have been fit to a model (the colored skeleton overlaid in the first image) and place them in a space whose axes are the model parameters. Again, this diagram shows just two of a larger number of dimensions. Animation is used to slide each example from the arbitrary grid to its correct position in the map, then to bring in a list of challenges and check off those that are addressed by this representation.

These are examples of the "dynamics" type of animation, where the motion is used to illustrate movement of objects in either physical or virtual spaces. Another is illustrated in Figure 5.6, which shows how a template surface (the blue-and-white checkered surface, whose control points are the yellow spheres) is adjusted to fit a set of range scan data (shown superimposed, at the top of the second column). The camera is then rotated for a better view, and a cross-section through the arm is shown.[1] These types of animation are perhaps the most obvious applications of animation, at least for topics where they can be usefully applied. Next we will look at some examples of animation being used more abstractly.

The sequence in Figure 5.7 is used to introduce the goal of the work, cover some related work, and then move into the newly proposed method. It begins with a simple slide indicating the desired input and output of the system. The two icons then shrink and slide apart to make room for some illustrations of previous attempts to solve the problem. The pink box groups together one kind of approach to the problem, while the yellow box contains a different set of techniques. One issue with the yellow set is highlighted via inset closeups that emerge from the picture. The presentation then moves from talking about related work to talking about the new method, indicated by the "Related Work" title bar sliding out to be replaced by the "Our approach" title. The figure shrinks down to make room, and then images of the scanning apparatus and collected data are shown. These are then replaced by a grid of twelve images representing a set of input data obtained by the scanner. Finally,

[1]This animation, like all 3D examples in SLITHY, is created by bypassing SLITHY's Python drawing library and implementing a parameterized diagram directly in C and OpenGL.

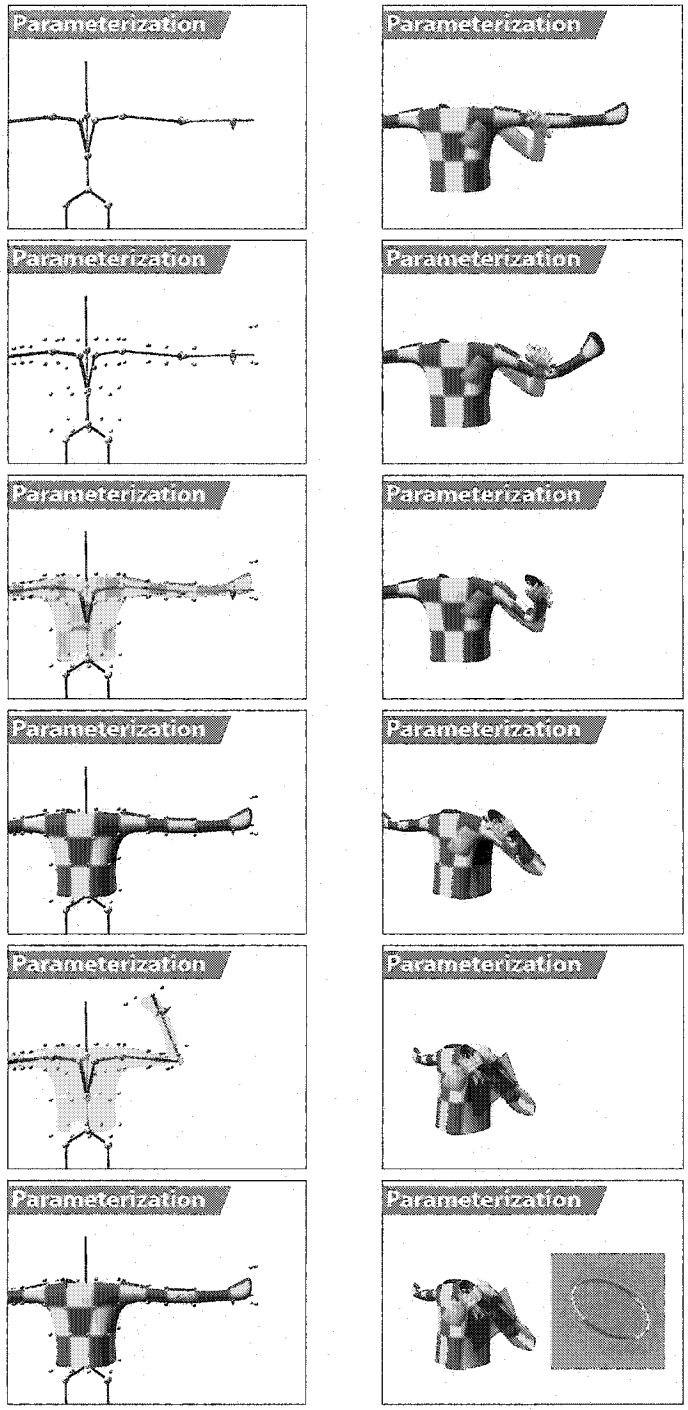


Figure 5.6 *Frames from an animation that uses a 3D parameterized diagram implemented in C and OpenGL.*

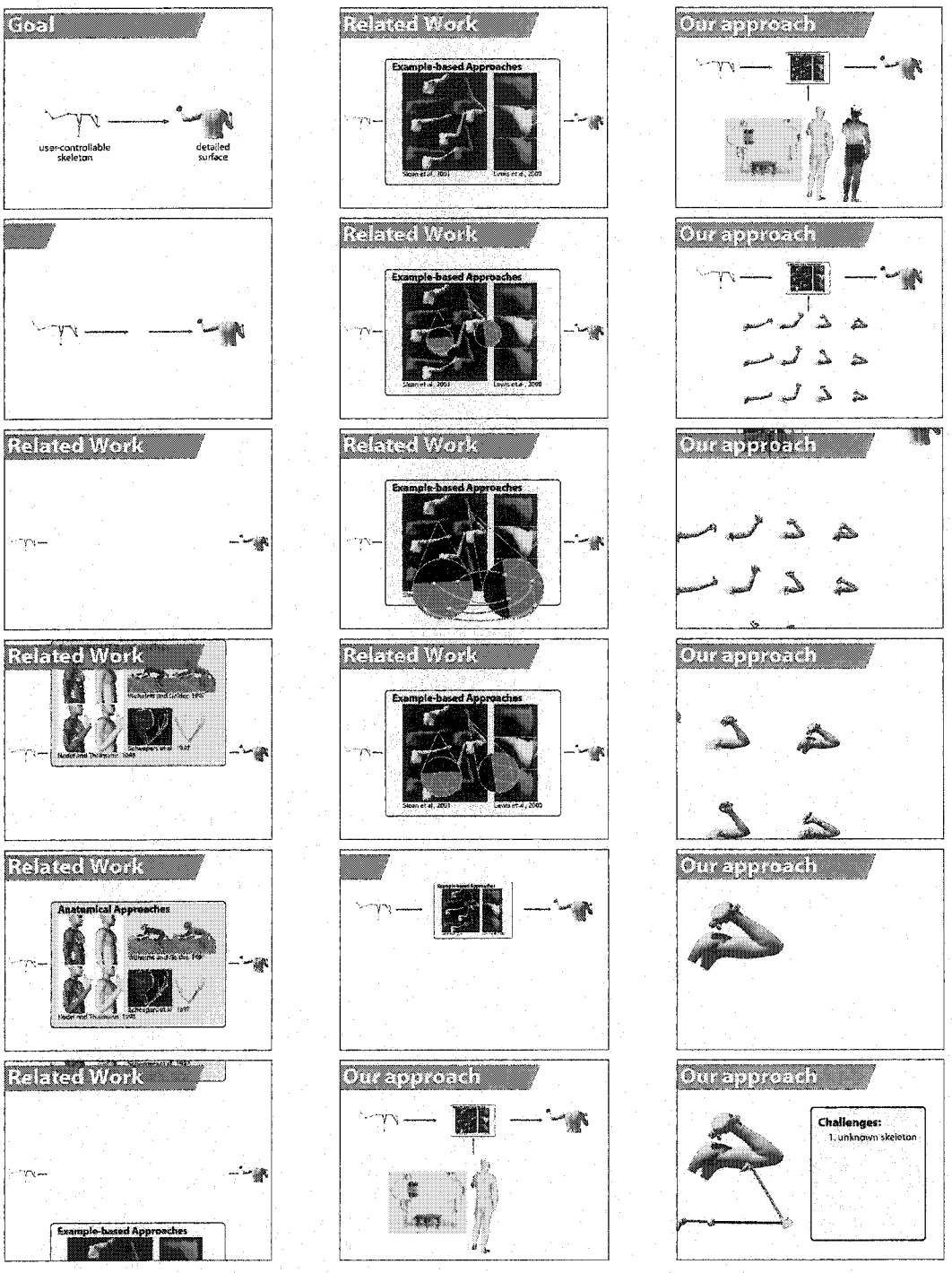


Figure 5.7 A SLITHY *sequence that uses motion to show the overall goal of the system is related to previous work in the area as well as the new approach. Images representing different parts of the system slide around on the screen, maintaining continuity while making room for new information.*

the screen focuses in on one of these examples and begins to enumerate a list of challenges to be addressed in processing the input.

This sequence makes excellent use of animation, even though movement is not part of the topic it addresses. The two images representing the overall system goal remain onscreen throughout the related work section and into the "our approach" section, providing continuity and underscoring how the other pieces fit into that goal. It would be hard to get this effect with a series of still slides: While the two images could be placed on each slide, every time they change position or scale might require explanation. With animation the audience can immediately see how the two large images on the initial slide are connected to the small icons of the other sections. The fact that they have smoothly slid around and shrunk has not changed their identity, whereas an image that instantaneously blinks from one spot to another might not be recognized as representing the same "thing."

The next two sequences, Figures 5.8 and 5.9, were created to present the video matting work of Chuang *et al.* [13]. The first sequence illustrates the image compositing algorithm, which takes three input images: a background image, foreground image, and an alpha mask, and combines them to form a single output image. The animation starts by showing the three inputs. Then the camera pulls back to make room, and the foreground and alpha images are combined to form an intermediate image. Next a copy of the alpha mask slides over and is inverted, and the inverted mask is combined with the background image. The two intermediate images are then combined to form the final composite. Finally, the intermediate constructions disappear and the remaining images slide together to summarize the overall process. This is a subtle but nice example of using animation to expand and compress detail. It would be hard to come up with a layout that worked well for showing all seven images of the construction, but also looked good when the intermediates were removed to show a summary. Animation is used here to solve the problem of changing the layout of the slide and making the changes transparent to the viewer—the audience doesn't need to look at the labels to establish correspondences, because each element has moved smoothly from one position to the next.

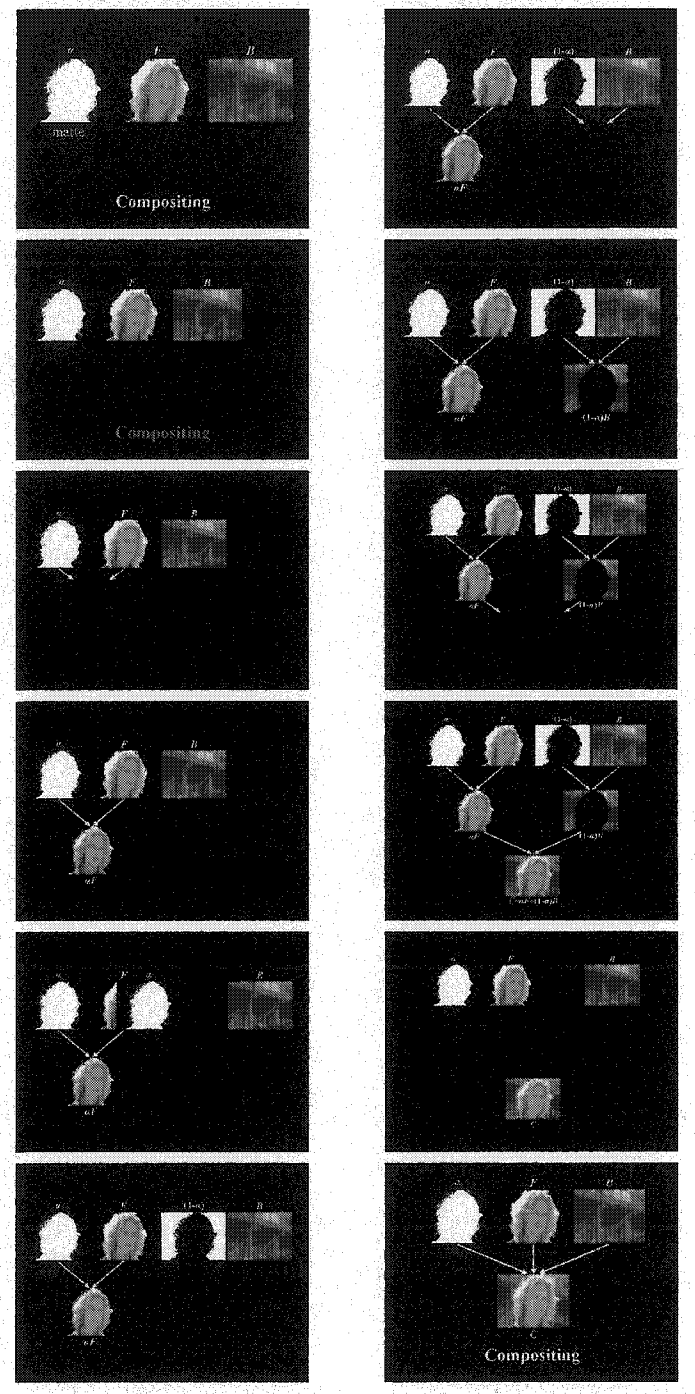


Figure 5.8 *This sequence starts with three images and illustrates the steps by which they are combined to form a composite. Animation makes it possible to slide the images around as needed to lay out each stage of the construction attractively, without having the pieces jump around between slides.*

The second example from Chuang *et al.*, Figure 5.9, uses larger scale changes to focus on very small details. This sequence starts with an illustration of the overall matting problem, which is to reverse the compositing operation and extract the three source images from the composite. (Note the similarity of the opening frame to the final frame of Figure 5.8. Though not shown here, animation is also used to move continuously from that figure to this one.) In this case the algorithm requires some supplemental information, called a trimap and represented by the blue contours.

The contours slide over to line up with the image as the camera focuses in on it. First the algorithm's overall strategy is shown, by highlighting the pixels in yellow in the order in which they are calculated. The camera then zooms in even more to focus on one small part of the image and illustrate how the algorithm operates on a single target pixel. An RGB cube appears next to the image, and pixels from the target pixel's neighborhood fly out to populate the RGB space. These clouds of points are replaced by statistical distributions, drawn as ellipses, that are then used to estimate the alpha mask at the target pixel. Having demonstrated the complete algorithm on one pixel, the camera pulls back to show the result on the whole image.

Figure 5.10 is similar to the example of Figure 5.7 in that images are used to represent different parts of the system in a diagram that shows their relationship. The sequence begins with the desired input and output of the system (the images look the same in this figure because the "images" are actually video clips showing different animation styles, and the two videos have the same ending frame). The images representing input and output shrink down and slide apart to make room for the block diagram that will develop between them. As each block is introduced, a callout image emerges to illustrate the need for that part of the system.

This block diagram lends structure to the whole presentation. Figure 5.11 shows one portion of the talk. The camera zooms in to the "momentum constraints" block of the system diagram while fading to a new slide, as though all the slides of this section are contained within that block. The first diagram illustrates breaking the motion up into flight stages and ground stages. A pair of boxes, one for each of these classes, then slide in. Each box contains smaller diagrams illustrating a particular type of constraint. These boxes are not still images, but small animations. The camera zooms

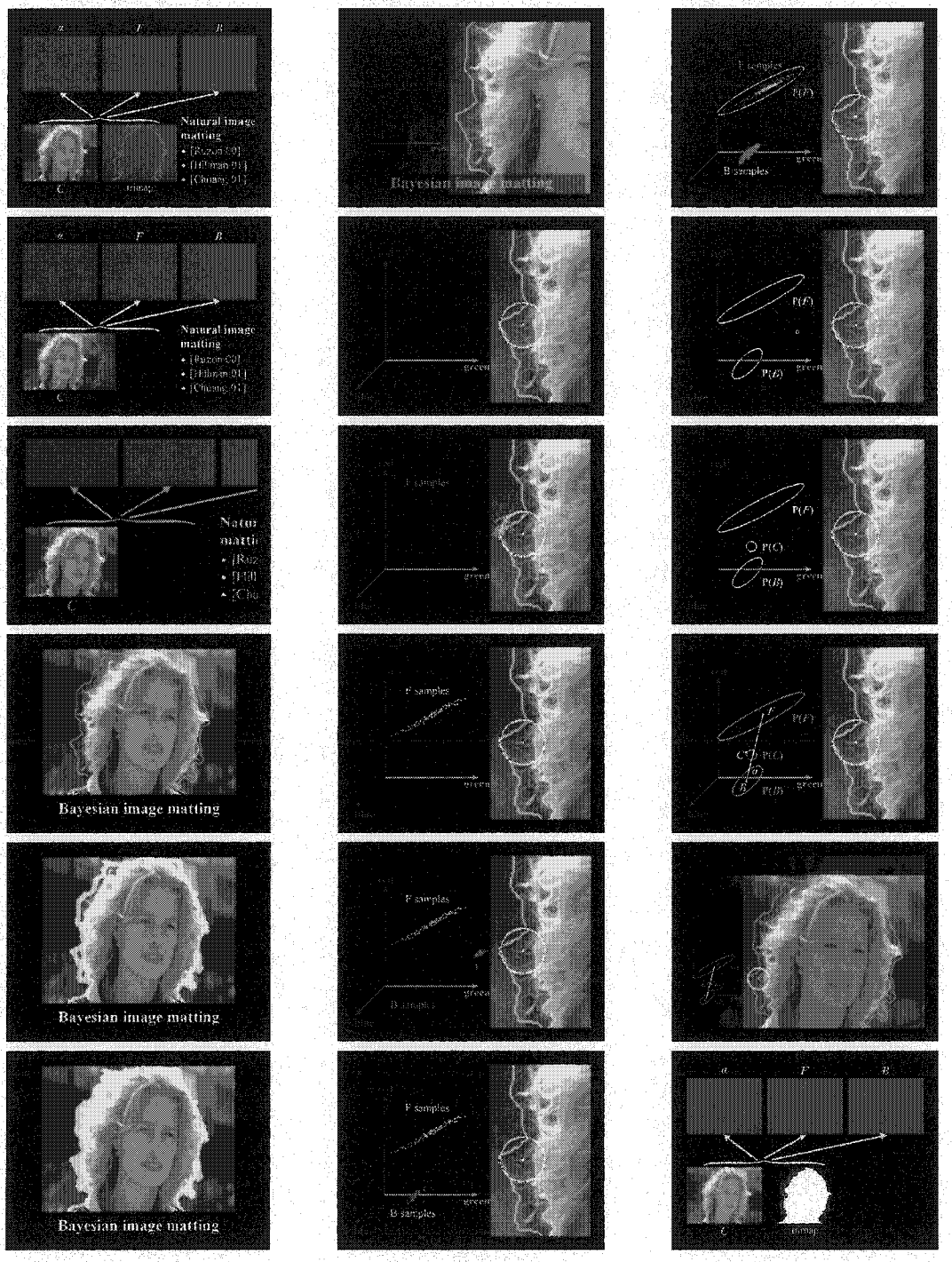


Figure 5.9 *Animation illustrating the Bayesian matting algorithm of Chuang et al. [13]. This animation uses animated zooming to show the technique at both micro and macro scales.*

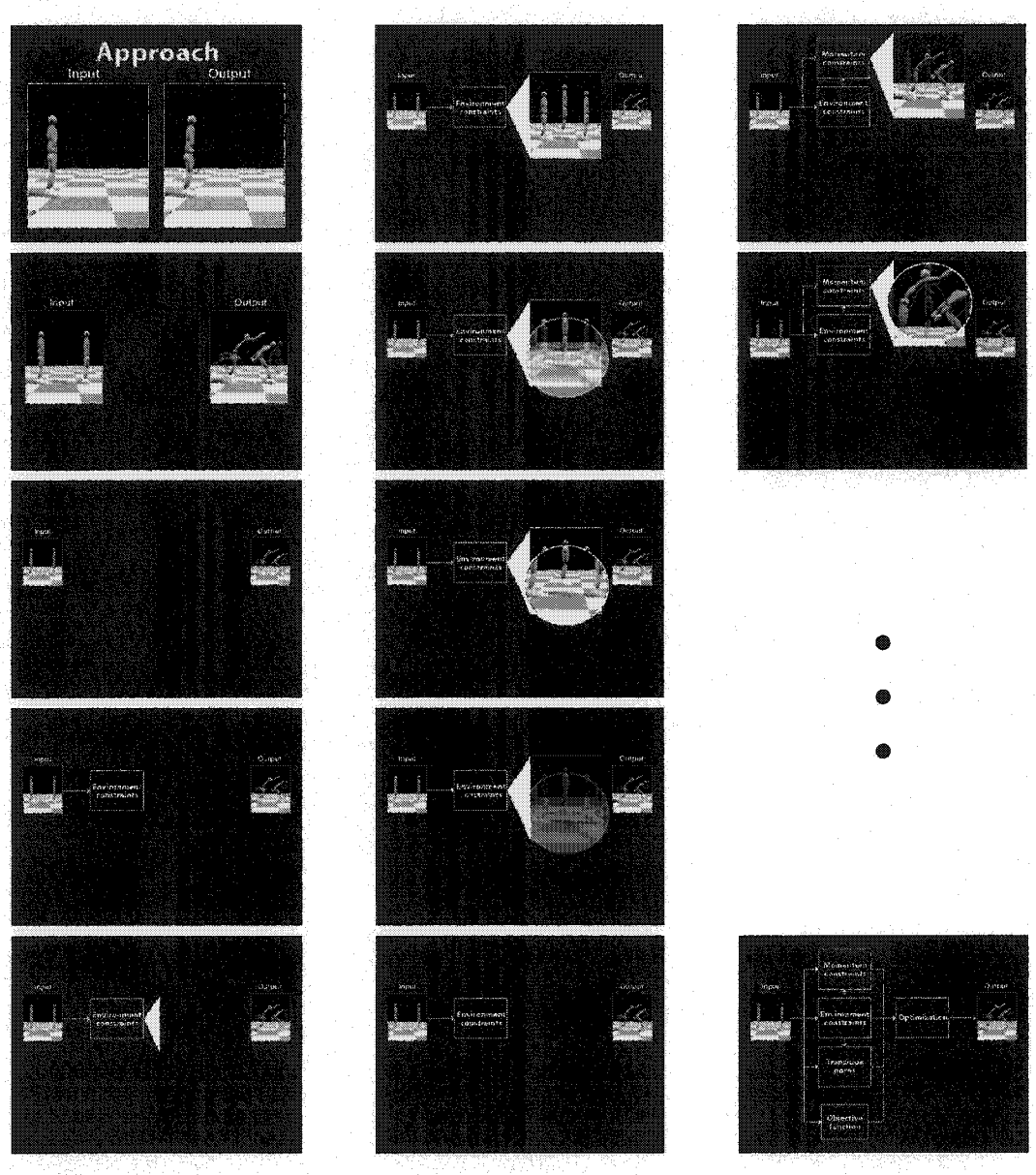


Figure 5.10 *This sequence constructs a block diagram of a system, beginning with the input and desired output. Small callout images and animations emerge from each block to highlight specific issues with that part of the system.*

further in to each of these animated vignettes, where motion is used to illustrate the constraint. The camera moves from box to box, showing each illustration. When it pulls out again, each of the small illustrations is now in its end state. This part of the talk concludes by pulling back out to reveal the overall block diagram. The other blocks are explained with similar sequences. This presentation uses the transitions between slides to reflect the hierarchical organization of the talk, which addresses the various issues in a depth-first manner.

This example illustrates the need for deep hierarchical assembly of slides. It would be nice to actually show the detail slides inside the diagram blocks, rather than just suggesting the containment by a zoom and crossfade. This presentation was authored with SLITHY-1, which did not allow this kind of deep assembly. Seeing examples like this one spurred us to allow a richer relationship between diagrams and animations in subsequent versions.

5.2 Using interactivity

Despite the severe restrictions imposed on interactive objects by the first version of SLITHY (see Section 2.4.2), one of our users was able to integrate a nice use of interaction into his presentation. Figure 5.12 shows four screenshots of the interactive portion of this presentation. The chart shows several example input data sets laid out in parameter space. The algorithm described in the presentation uses a weighted k-nearest neighbor algorithm to generate data for new points in this space. The interactive tool lets the presenter move the sample point around with the mouse, connecting the point to its four nearest neighbors. The weight of each neighbor is reflected in the thickness of the connecting line.

This kind of simple "live demo" can be much more engaging to the audience than an entirely prescripted sequence. In smaller-scale situations, it also allows the presenter to respond to hypotheticals and questions from the audience.

The improvements to interactive objects described in Section 3.3 make it easier to implement more elaborate interactive demos. Figure 5.13 shows an interactive demonstration of the de Casteljau algorithm for drawing Bézier curves. The user can click in the background to add control points,

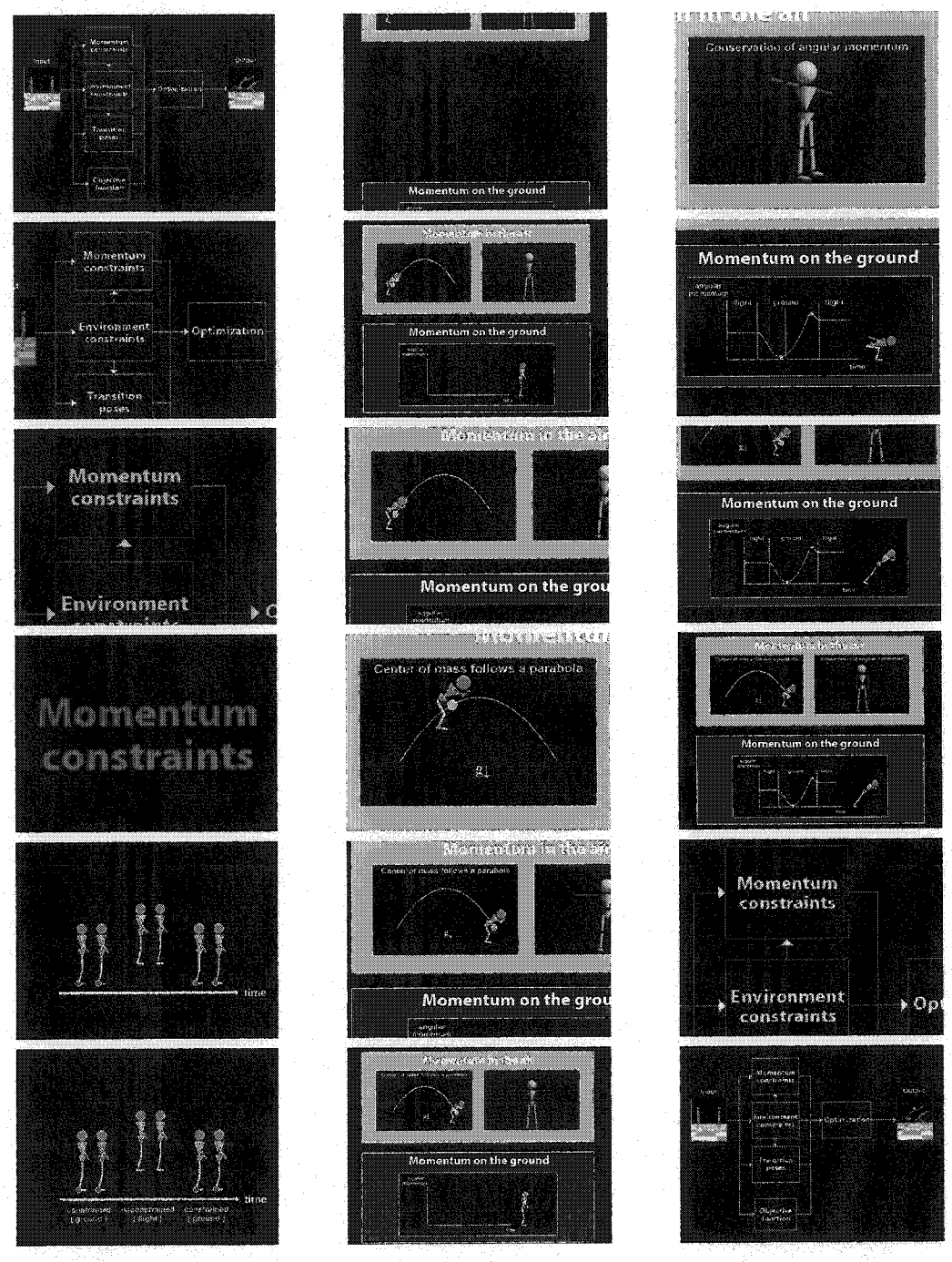


Figure 5.11 *This sequence uses the block diagram to structure the remainder of the talk. Zooming is used to give the impression that each set of detail slides are located within the corresponding block. This containment in used recursively; with some of the detail slides the camera zooms in further to focus on individual animated images.*

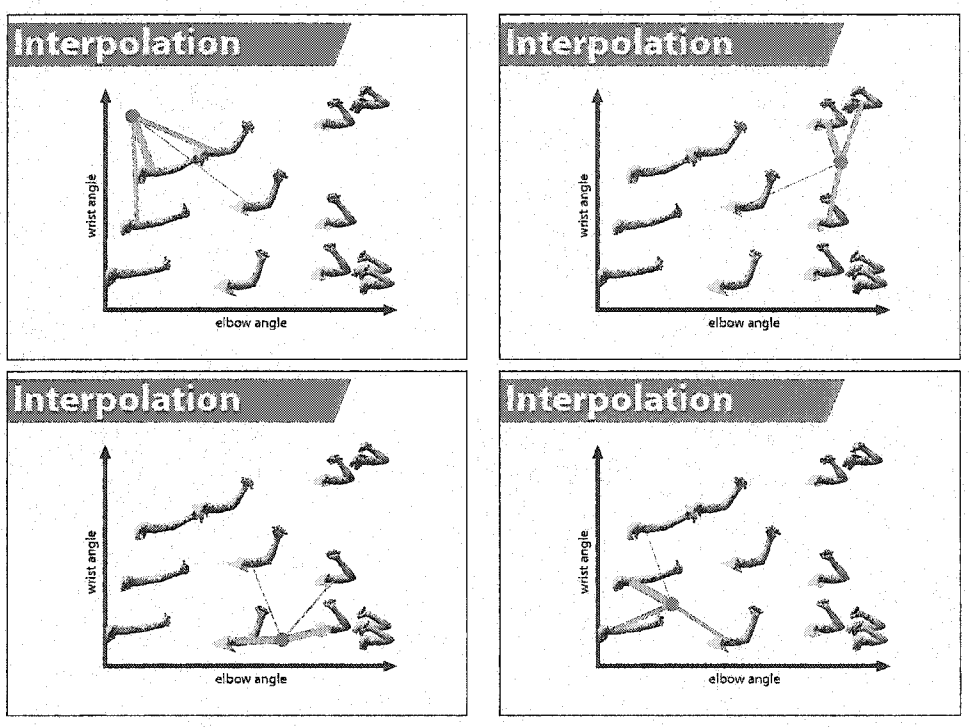


Figure 5.12 *This "slide" lets the presenter move the sample point using the mouse, allowing interactive exploration of the diagram.*

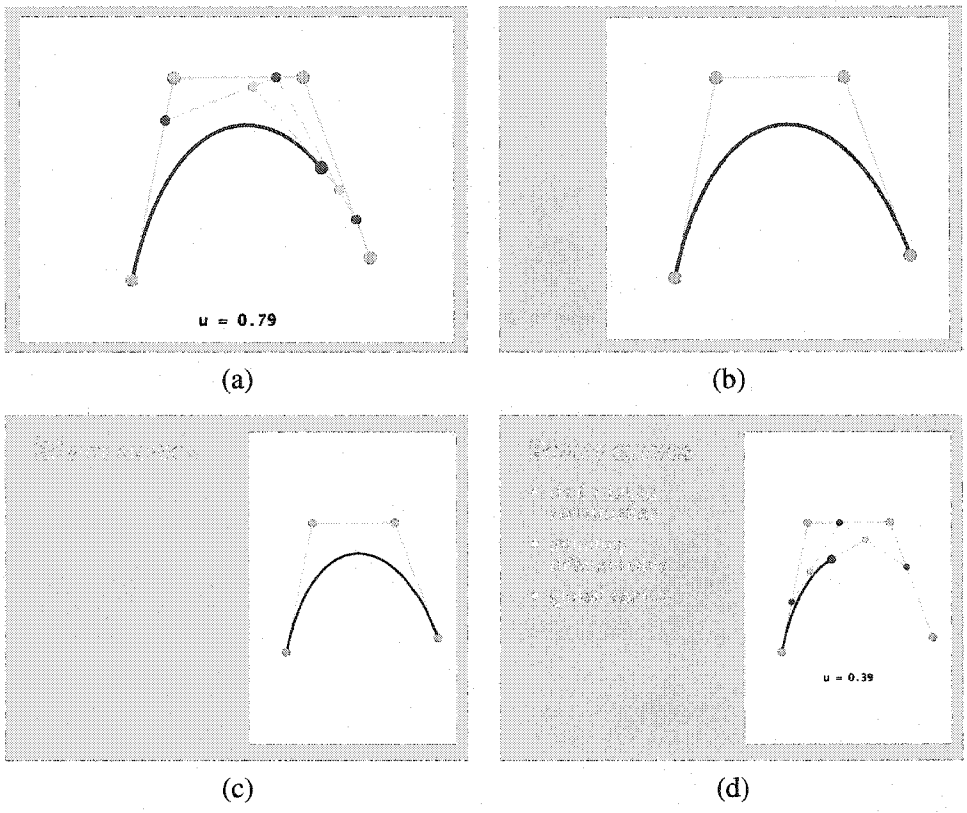
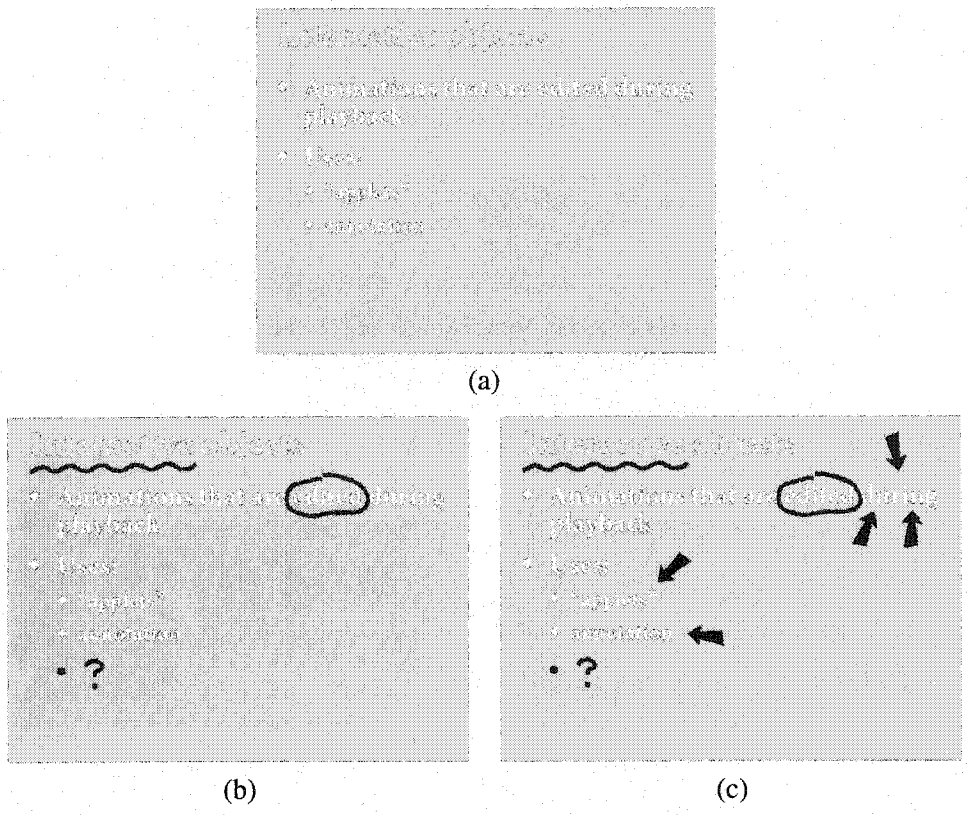


Figure 5.13 *A complex interactive object that lets the user draw Bézier curves and animate their geometric construction. Green control points can be added and moved around with the mouse, while the dark blue points can be dragged around to modify the value of u. The interactive object coexists on the slide with other* SLITHY *elements such as text and bulleted lists.*

shown in green. Keypresses are used to advance through steps of the construction (using animated transitions). The green control points and blue construction points can be dragged around with the mouse, with the rest of the diagram adjusting in real time. The later frames show how the interactive object itself is part of a larger SLITHY animation: it shrinks and slides over to make room on the screen for the bulleted list. The movement on the screen and the text bullets are not part of the interactive object itself; they are elements of the animation object that contains the interactive object.

Another use of interactive objects made possible by their integration with other animation objects is the ability to extend SLITHY's interface to some degree. Figure 5.14 shows two interactive objects that allow on-screen annotation of running animations. In each case the interactive object



(a)

(b)

(c)

Figure 5.14 *Interactive objects are being used to allow the presenter to make annotations on top of a running* SLITHY *animation. Part (a) shows an ordinary* SLITHY *slide before any annotations have been added. In part (b) the mouse is used to sketch lines on the slide; at the conclusion of each stroke the actual path recorded is replaced with a smooth curve. Part (c) shows how the same interactive objects also lets the presenter draw arrows gesturally by clicking where the arrow is to point and moving the mouse to indicate the direction.*

is laid on top of an existing animation object, by creating a new animation object that contains two elements: the original animation object, and the interactive element. No change to the original animation script is necessary, because the system can manipulate the object that results from executing the script.

Chapter 6

RELATED WORK

In this chapter we will discuss three areas of previous work that are relevant to our research. First, we will look at some of the efforts by cognitive and educational psychology researchers to quantify the effectiveness of animation. The question of whether animation is superior to static graphics is still unsettled, though many studies seem to lean at least slightly in favor of animation.

Sections 6.2 and 6.3 compare SLITHY to other software systems: first those created for *making animation*, then those created for *giving presentations*. Our system appears to be the first specifically intended for doing both in a general way.

6.1 Effectiveness of animation

Just as a static graphic uses position to represent relationships between elements in physical spaces (e.g., maps) or abstract space (e.g., an organizational chart), it is widely believed that animation can use time as a natural representation for a variety of processes, procedures, and algorithms. The effectiveness of animation in instruction has been studied by a number of researchers in the fields of psychology and education. In this section we will summarize some of those results.

Reiber [54] summarizes twelve prior studies of animation's effectiveness in learning to come up with three broad recommendations regarding the instructional use of animation. The first is to use animation only when it fits the subject matter well. Animation is most effectively used to show motion and trajectory, so the observed usefulness of animation depends on how fundamental these kinds of concepts are to the learning task. He criticizes one study [9] that found no differences between *text*, *text plus still graphics*, and *text plus animated graphics* for using subject matter not well-suited for visual support. Another study [55] used a subject that most students found "exceedingly demanding." Measurements of the time spent viewing each lesson frame suggested that the

students may have been devoting the time intended for watching the animation for other tasks, such as reading text. A follow-up study that simplified the lesson material and allowed the students to proceed at their own pace (by breaking the lesson frames down into "chunks" and letting students add chunks one at a time by pressing a key) found that the animation group *did* perform better than both the static graphic and no graphic groups.

Rieber's second recommendation is concerned with the fact that novices may not know what the relevant cues and details are when viewing an animation, and this may detract from the animation's effectiveness. The use of "chunking" – breaking down an instructional sequence into individually-displayed components – as a major contributing factor in finding a positive effect for animation is supported by both the above-mentioned study and a second, separate study by Rieber [52]. We informally came to the same conclusion in our own experiences with animation, and this is reflected in some of the animation principles discussed in Chapter 4.

Rieber's final recommendation notes that the potential of animation for computer-based instruction may be greatest in the area of interactive lessons, where students have some control over the animated dynamics. A number of studies have demonstrated effective learning by using animated "applets," though it is hard to isolate the effect of the animation from that of the interactivity, which is known to contribute to learning on its own (for an example, see Ferguson and Hegarty [23]). Our current work has not explored this line of research. While our system allows the creation of interactive parts of the presentation, it is the presenter who is controlling the interaction, not the audience. Our primary goal in adding the option of interactivity is to allow the author to add some flexibility and variety to the presentation, not to give each audience member a personally varied experience. This might be an interesting application of our system, however, and will be mentioned again in the section on future work. For now we are focusing on the more common, passive style of presentation.

A study by ChanLin [11] tested this type of passive presentation. The experiment was performed with 135 college undergraduates learning about how genetic information is used to encode protein structure. The participants were classified as either "novice" or "experienced" learners based on their major; those with majors in some area of science were considered to be experienced learners,

while those in the social sciences were considered novice learners for this subject area. The participants were randomly assigned into one of three treatment groups: non-graphics, still graphics, or animated graphics. Each treatment included the same textual information, but the graphical ones were augmented with either static or animated graphics. After viewing the lesson, learners were given a pencil-and-paper test that had questions about both descriptive facts (defined as "describing an object, a definition, a rule, or specific factual information") and procedural facts (which "[refer] to a series of executable actions, or [contain] to-be-remembered steps presented in a specific time sequence.")

For descriptive knowledge, both types of graphics treatments performed better than the non-graphical treatment for novice learners. For the experienced learners, though, only animation was significantly better than the text-only treatment. ChanLin hypothesized that for these students who already had a scientific background, the use of animation served as a mnemonic device to help them remember the new factual information.

For procedural knowledge, the results were somewhat different. Again there was no statistically significant difference between the treatment groups for novice learners. For experienced learners, though, only static graphics were better than the text-only treatment. Here the author observes that the pacing of the presentation may play a role: both the animated version and the text-only version presented the information sequentially, with the timing controlled by the system, not the student. The still-graphics version presented a procedure as a single graphic, allowing students to process each step at their own pace.

It is worth noting that almost all of our experience with animated presentations has been in giving technical talks to a well-informed audience, who may be classified as "experienced learners." This audience bias may account for at least part of our positive experience with animated presentations; it would be interesting to see if SLITHY presentations were as effective with novices (e.g., in a classroom situation).

Park and Gittelman [44] compared students who viewed an animated tutorial on electronic circuits to those who viewed a tutorial using static graphics. While the animated tutorial took no more

time than the static one (students could advance both tutorials at their own pace; the difference in time taken was not statistically significant), those who learned from animation needed significantly fewer trials to troubleshoot faulty circuits on a subsequent test.

Thompson and Riding [58] found that showing an animated proof of Pythagoras's Theorem (similar to Figure 5.1) to a group of middle- and high-school students was more effective than static diagrams presented either in series on a computer screen or as small multiples on paper. While the difference was statistically significant, it was small. The authors speculate that this was due to the highly controlled nature of the experiment, which was fairly dissimilar to a normal classroom situation. In this experiment, the teacher gave no explanations during the presentations; the students sat and watched in silence. No questions from the students were allowed.

The tight controls imposed on a formal study of this nature can be both a curse and a blessing. On one hand, it allows the experimenters to reasonably determine if it is the animation itself which brought about the improvement, and not some combination of other factors. On the other hand, it says very little about the efficacy of animation in a real-world situation.

Tversky and colleagues [62, 39] also raise objections to some studies of this type from a psychologist's perspective. They question the conclusions of all of the above-mentioned studies on the basis that the static and animated graphics did not present exactly the same information. In Rieber's study evaluating graphics on Newton's laws of motion [55], for instance, both types of graphics were used to accompany text. The animation, however, illustrated some aspects of the motion that were not discernible from the static graphics. Students viewing the static graphics would have had to extract this information from the text.

The claimed difference between the static and animated graphics of the Thompson and Riding study [58] is even more subtle. They compared a smoothly animated graphic to a short series of discrete diagrams shown flipbook-style. Tversky *et al.* criticize this comparison on the basis that the animation showed the individual "micro-steps" leading from one state of the diagram to the next, while students viewing the static graphic condition had to *infer* the motion for themselves.

From our perspective, the fact that animation allows us to show more dynamic information (and

more information overall) does not make the comparison to static graphics an invalid one—it is in fact that difference that is being tested. We are forced to wonder how Tversky *et al.* would go about constructing static and animated graphics that were *exactly* equivalent, "except for the animation." A study of this nature might be interesting from a cognitive research perspective, but would seem to be of little use in guiding the creation of presentations for actual use. For this and other reasons, we have elected not to attempt rigorous, formal user studies to see if presentations made with our system are "better" than those made with PowerPoint or other available systems. There are so many factors that go into making a presentation effective – from the enthusiasm and skill of the speaker to the knowledge and interest levels of the audience – that such a comparison seems doomed from the start.

Even without the requirement that both types of presentation contain exactly the same information, it is difficult to test the hypothesis that presentations with animated visuals are more effective than presentations with static visuals. For one thing, "effective" can be interpreted in many different ways. All of the studies we have seen concentrate on some educational aspect, with performance measured via a post-presentation test of recall or problem-solving related to the content. Not all presentations are made in order to educate, though. The presenter may be trying to convince the audience of a point, or perhaps just entertain them, or some combination of the three.

Many confounding factors complicate the design of a formal study—the presenter's experience with both systems, the audience's interest in and knowledge of the subject, and so on. One interesting effect that we have observed with animated presentations is that audience members typically respond well to the presentation due to the novelty of using animation. One of our early users noted that people coming up with questions after his talk were often asking about the presentation tool itself rather than about his work. It may be difficult to make a fair comparison between our animated presentations and traditional formats until audiences are as accustomed to seeing animation as they are to seeing standard PowerPoint material.

In the meantime, the use of animation, static graphics, and text are not mutually exclusive. Our goal in building SLITHY is to provide a good animation tool; we leave it up to the judgment of the

presenter as to whether or not to use it in any given situation.

Incidentally, in addition to the criticisms noted above, there is no shortage of studies that found no benefit to animation over static graphics (e.g., [48, 49, 53]), though many of these have also been criticized on methodological grounds and on the appropriateness of animation for the subject matter used. However, the fact remains that people *do* use animation, despite the lack of conclusive psychological research in support of it. The animated effects available in presentation software today, crude though they may be, are frequently seen in practice. (Gagné [26] points out that attention-gaining is an important precursor to instruction, which may serve as some justification for using the flying-text type of animation.)

Since many presentation authors seem to believe that animation is an effective tool, we may as well enable creating the most useful animation that we can. Here the psychological literature is somewhat more useful in providing specific recommendations. Even Tversky *et al.* [62], while faulting the published studies for what they see as poor methodology, offer principles guiding the use of animation in instruction. Park and Hopkins [45] suggest five specific instructional conditions where animation can be effectively applied:

1. demonstrating procedural actions,

2. simulating system behaviors,

3. explicitly representing invisible movements or phenomena,

4. illustrating structural, functional, and procedural relationships among objects and events, and

5. focusing the learners' attention on important concepts.

Current commercial presentation tools are geared exclusively towards the last of these situations, using animation only to draw the viewer's eye. Research efforts in the presentation domain, such as Pad [47] and Counterpoint [27], address the fourth situation, by using animated navigation to

illustrate the relationship between slides of static content. In this work we have sought to enable the creation of animations that cover all five of these possible scenarios.

A series of experiments by Mayer and Anderson [35, 36] tested animation used in concert with narration. Their findings are consistent with predictions based on the dual-coding theory of Paivio [43], which says that learners can build separate mental representations based on visual and verbal input, and can form referential connections between the two mental models. In their largest experiment, Mayer and Anderson divided a set of college undergraduates into seven groups:

- One control group received no instruction.
- Two groups received either three repetitions of a narrated explanation of a mechanical system (the "NNN" group) or three repetitions of an animation of that system (the "AAA" group).
- Four groups received three repetitions each of both the narration and the animation, interspersed in various ways ("AAANNN", "NNNAAA", "ANANAN", "NANANA").
- A final group received three repetitions of the animation and the narration delivered simultaneously, in synch with each other (the "concurrent" group).

The results showed that all the groups receiving instruction did equally well on a subsequent test of simple recall. The concurrent group, though, did significantly better than the others on a test of problem-solving, such as diagnosing a fault in the system. The authors hypothesize that seeing and hearing both modes of instruction together helps build mental links between the separate models, which later aids in creative reasoning about the material. These results suggest that narration is an essential accompaniment to animation, which we have also found to be the case using SLITHY.

One possible objection to our system is that the lack of a WYSIWYG graphical interface for creating diagrams limits the ability of authors in creating elaborate illustrations. We have not found this to be a problem in practice. Many guides on giving presentations (such as [63] and [30]) and creating scientific illustrations (such as [5]) encourage authors to make their visual displays as simple as possible, to focus the audience's attention on the most essential aspects. This philosophy

is also reflected in the well-known works of Edward Tufte [59, 60, 61], who calls it "maximizing the data-ink ratio."

The use of simple graphics in presentations is also supported by educational research. Dwyer [18] compared using three different anatomical illustrations along with oral instruction: a realistic photograph, a detailed, shaded drawing, and a simple line drawing. (A fourth treatment group received only the oral instruction, with no visualization.) The results showed that the photograph was the *least* effective visualization, while the line drawing was the most effective. This conclusion was also supported by a number of follow-up studies [19, 20, 21], though [20] found that while different visualizations produced differences in immediate retention, these effects did not show up on tests given later, after some time had passed.

6.2 Creating animation

In this section we look at prior systems for creating animation using scripting. Most of them predate the widespread availability of high-performance graphics hardware and graphical interfaces on the desktop. Such systems are designed for batch rendering, not real-time display. The scripting style of interface seems to have been largely abandoned by the computer graphics community in favor of interactive, mouse-based editing. Scripts are associated with a more mechanical, less expressive style of animation. While that style may be bad for character animation, it is often exactly what is needed for making presentations.

SLITHY has borrowed or reinvented many ideas from these prior scripted systems, adapting them for real-time use. One apparently unique aspect of SLITHY is our time cursor model for expressing overlapping actions, and the use of the `parallel()` and `serial()` functions to alter how other animation commands move the time cursor. We have not found a system that uses a similar mechanism in the literature.

We will now take a closer look at each related system, going in chronological order.

Anima II

Amima II [28] was an early system that used a scripting language to animate 3D objects. The language consisted of directives such as

```
CHANGE POSITION name TO, x, y, z, FROM frame 1, TO frame 100
```

The input script was read in by a parser and stored as a sequence of "command blocks." To generate each frame, the system would go through and determine which command blocks were active, based on the current frame number. Each active command block was evaluated to execute its change, and the resulting scene was rendered. By specifying overlapping ranges of frame numbers, parameters could be animated in parallel.

One limitation of this evaluation scheme is that frames must be evaluated in order from the beginning of the sequence. Note that the `CHANGE` command above does not specify the starting point of the position change, only the end point. Change command blocks would examine their parameter's value the first time the block became active (in this case, at frame 1) and use that value to compute a per-frame delta vector that was stored in the command block itself. This vector was used to incrementally compute the value of the parameter for subsequent frames. This method of computation did not provide random access to frames of the animation. This was not considered a drawback, since rendering on the hardware of the day took over one second per frame anyway.

ASAS

The Actor/Scriptor Animation System of Reynolds [51] was one of the first scripting-based modeling and animation systems to be based on a general purpose language, in this case, Lisp. It has a number of similarities to SLITHY.

In ASAS, models were defined using primitive functions (or "operators") such as `vector` and `polygon`. These functions created and returned geometric objects. Other operators like `move` and `rotate`, when applied to an object, returned a transformed copy of the object. Because the system was built as an extension of Lisp, users could use all the regular features of the language

to build parameterized objects. The following code defines a new operator `rotated-triangle` that takes an angle parameter and returns a triangle:

```
(defop rotated-triangle
       (param: angle)

       (rotate angle y-axis (polygon blue
                                     (vector 1 0 0)
                                     (vector 0 1 0)
                                     (vector 0 0 1))))
```

SLITHY's parameterized diagrams are very much reminiscent of this design—the system itself provides a set of graphical primitives from which the user can build functions to draw arbitrary things. One difference is that ASAS primitives *returned* their graphical objects; the object returned by the top-level call is what was drawn on the screen. This style of programming is common for functional languages like Lisp. SLITHY is based on Python, an imperative language, so SLITHY primitives are implemented as commands that have drawing side effects.

Animation was also created through Lisp code. Animators would write a function called an *actor,* which would be executed once per frame and would add objects to the scene for that frame by calling model operators like those described above. Multiple actors could be active at once. A top-level Lisp program, called a *script,* controlled the overall animation by starting and stopping actors at the appropriate times. As each actor was called, the ASAS system would save and restore its local variables so that, to the actor, it looked as if it were executing by itself. There was also a message-passing interface allowing actors to communicate with each other, so behavioral simulations such as flocking could be implemented.

Like ANIMA II, ASAS was intended for offline rendering. The state of the animation was distributed across multiple modules (the actors, in this case), and each actor encoded only how to go from one frame to the next, so the animation had to always be generated starting at the first frame. While the modeling systems of ASAS and SLITHY are quite similar (modulo the choice of implementation language), we believe that random access to the animation is important for the presentation domain, as is having time defined continuously, rather than as discrete frames.

Dial

The Dial ("diagrammatic animation language") system of Feiner *et al.* [22] took a novel approach that combined scripting and graphical definitions of animation. In Dial, an animation script defined *events* using an ordinary command language, but the timing was specified through an ASCII art-style diagram. Here is a very simple Dial script:

```
% throw moverel "ball" 0.0 0.5

throw   | |- |--   |---    |----     |-----
```

The first line defines an event called "throw" which will move the ball model. (The object "ball" is defined elsewhere and loaded into the animation system; Dial itself had no modeling component.) The definition specifies what is to happen (a relative movement) and how far the movement should be, but says nothing about timing. That is specified in the second line. On this *execution line* each frame is represented by one character position. A single "|" character indicates that the event should be executed in that frame. An event can be stretched out in time by following it with "-" characters, so "|---" indicates that the event is to happen over four frames. In this example, the ball will be moved six times, moving more slowly each time.

Parallelism is expressed through multiple execution lines. Here is a more complex example:

```
% throw moverel "ball" 0.0 10.0
% swing rotate  "bat"  0.0 1.0 90.0
% hit   moverel "ball" 0.0 -50.0

throw    |--------
swing          |--
hit               |---------------
```

Here the three execution lines will run in parallel; the "throw" will start first, then end at the same time as the "swing" event, and both will be followed by the "hit" event.

Hanrahan and Sturman

Hanrahan and Sturman [29] describe a language for creating parameterized 3D models that can be manipulated interactively.[1] Here is the definition of a cylinder with a spherical endcap:

```
{% tube
  parameter scalar a = 2, r = 1
  {% center
    rot -90 z
    scale r, 2*a, r
    cylinder
  }
  {% cap
    move -a, 0, 0
    scale r
    sphere
  }
}
```

This example defines a model with two parameters, a and r. These parameters are used in computing the transformations that are applied to the `cylinder` and `sphere` primitives. The example shows the use of arithmetic expressions, but this is not a general-purpose programming language—there are no loops or conditionals. Because of this, the system can statically determine which parts of the model are affected by which parameters, allowing for fast incremental update when parameters change. This incremental update property is important to allow for real-time modification of the parameters, and would not be possible if more complex control flow constructs were available.

In SLITHY the entire model function must be executed from the beginning on each redraw. The faster machines available today allow us to trade some efficiency for the additional power of performing arbitrary computation within parameterized diagram functions. We still rely on the author to write code that can be executed reasonably efficiently, but a lot more computation can be done per-frame on current hardware than was possible in 1985.

The Hanrahan and Sturman language does not do animation, but they produce animation using a simple keyframe mechanism. The models are manipulated by binding their parameters to physical

[1] They do not give their system a name.

input devices like dials and sliders, then adjusting them interactively. Once a desired combination of parameters is found it is saved as a keyframe, and the keyframes are then interpolated to animate the model.

BAGS/SCEFO

SCEFO, the animation component of the Brown Animation Generation System (BAGS) described by Strauss [57], is another example of a special-purpose animation language. Here is the script for a simple animation in SCEFO:

```
read ("cube.off") cube;

change (cube)  translate <0, {0, 0, 0}>
                         <10, {3, 2, 1}>,

               set_color <0, RED>,
                         <5, GREEN>,
                         <10, BLUE>;
```

The first line loads a polygonal object from a file `cube.off` and gives it the name `cube`. The `change` statement is the main animation-producing command in SCEFO. It takes any number of sub-actions, called *change-ops*. In this example `translate` and `set_color` are the change-ops. Each change-op takes a list of keyframe times and values. When the script is evaluated, the appropriate value for each change-op at that point in time is used, applying linear interpolation between the keyframes as necessary. Thus, this system can render frames independently, as a frame does not rely on the state left behind by rendering previous frames.

While SCEFO does allow the renderer to display arbitrary points in time (in principle, at least—the overall system was built for batch rendering, so it's not clear that this property was ever needed or used), the language itself does not allow general computation. Its "variable" and "function call" mechanisms work much like a macro pre-processor; these are expanded at compile time into straight-line code.

CHARLI

The CHARLI system of Chmilar and Wyvill [12] takes an innovative approach in which modeling and animation are specified together in a single script. The animated value of a parameter is specified in the model itself. A simple example is this animated triangle:

```
def triangle
    polygon( 0,0,0, 1,0,0, 0.5,{0.6667 at 0 sec linear 2 at 1 sec},0 );
end;
```

Here one of the numeric parameters in the call to `polygon` is replaced by a curly-brace expression that specifies a value that varies over time. These model-animation objects can use recursion to produce very intricate and complex animations. While this approach to animation is probably best suited for algorithmic or simulation-based animation, the idea of models with their own built-in animations could be useful in the presentation domain. A pendulum diagram, for instance, might have a parameter for controlling the pendulum's length, but the swinging behavior would be built in to the diagram itself, with no need for the animation driving the diagram to manage moving a "swing" parameter back and forth. This capability is currently lacking in SLITHY, but presents an interesting avenue for possible future work.

Menv

The Menv system [50] is one of the most successful efforts at using a script-based system for producing animation. A descendant of the system is still used today by Pixar for producing animated feature films. Models are created in Menv using a modeling language called ML. ML is a specialized language with primitives for creating 3D geometry and performing common graphics operations, but it also has many of the features found in general-purpose programming languages: variables, expressions, control flow constructs, procedures, and so on.

Menv's authors point out three major advantages that language-based modeling systems have over interactive ones: *replication, parameterization,* and *precision.* While both types of systems allow replicating a model through instancing, a language-based system has the additional power

to allow calculation of how many times to replicate and how to transform the various instances. A procedural specification of models also allows for complex parameterization, so that multiple instances can vary in nontrivial ways. The third advantage, precision, derives from the fact that the model's subparts can be positioned through calculation, eliminating the problems in alignment that can come from graphical placement, especially as the model is animated. Menv was built for doing 3D character animation, but all of these issues are as relevant for the creation of abstract 2D figures as they are for realistic 3D characters.

ML models are animated by tagging certain model parameters as *articulated variables*, or *avars*, which exposes them to the animation system so that their values may be varied over time. Menv uses an interactive keyframe-based system for animating these parameter values, though they note that procedural animation techniques could also be used.

Alice

The goal of the Alice project of Conway *et al.* [16, 15] was to create a 3D graphics programming environment specifically targeted at users with no graphics or programming experience. They concentrated heavily on user testing and careful attention to the terminology used in the API (e.g., using the term "move" instead of "translate"). It has no modeling component; Alice animations are created by manipulating 3D primitives created in other systems. Like SLITHY, Alice animations are created by writing scripts in Python, but Alice's animation model is closer to that of ASAS [51] than SLITHY. Here is a sample Alice script, which makes a bunny model jump into the air:

```
doinorder(
    bunny.resize( toptobottom, 0.5, likerubber ),      # scrunch down
    dotogether(
        bunny.resize( toptobottom, 2, likerubber ),    # leap
        bunny.move( up, 1 )
    ),
    bunny.move( down, 1 ),                             # fall
    bunny.resize( toptobottom, 0.5, likerubber ),      # bounce
    bunny.resize( toptobottom, 2, likerubber )
)
```

Each animation command (such as "`bunny.move`") creates an object containing information about

the model to be affected, the animated transform to apply, the extent and style of that transform, and the duration. (All animation commands in Alice have a default duration of one second, which is why no durations explicitly appear in the example above.) These objects are added to a central queue. Once per frame a scheduler goes through this queue and calls each object's update method to make its changes to the scene. This process is somewhat similar to ASAS's calling all the active actors once per frame. When a particular animation is finished, the corresponding object is removed from the queue.

The `doinorder()` and `dotogether()` functions create composite animations. Consider this simplified bunny example:

```
doinorder(
    bunny.move( ... ),
    bunny.resize( ... )
)
```

When this script is run, the `bunny.move()` and `bunny.resize()` calls are executed first, and their objects are inserted into the queue. Normally this would make both actions run simultaneously, which is not the desired effect. However, each animation function returns a reference to the queued object it created. These references are what `doinorder()` receives as its arguments. It immediately pauses these objects, which causes the scheduler to ignore them. Since they are created and paused in the same frame, no change is visible on the screen. The `doinorder()` function creates its own queued animation object, whose update method unpauses the paused objects at the correct times to make them play in sequence. Like other animation functions, `doinorder()` returns a reference to its own animation object. This object can itself be paused and unpaused, so these composite animations can be nested arbitrarily.

While `doinorder()` and `dotogether()` serve a similar purpose as the `serial()` and `parallel()` commands of SLITHY, their implementation is very different. The Alice commands take animation objects as arguments and build a new composite animation object. The SLITHY commands take no arguments, but instead change the state of the system so that other animation commands behave differently with respect to time. This model makes it possible to use control

structures like loops, conditionals, and function calls within an animation script. Alice scripts can't use control structures because an Alice script is always a single Python expression.

Alice is used to create interactive 3D worlds, with these animation scripts attached to events such as user input or collision between models. For instance, the first example script above might be attached to a mouse event so that the bunny jumps whenever the mouse button is clicked. The animation mechanism is well suited for running many scripts "simultaneously"—an animation script is actually completely evaluated instantaneously (between frames), leaving a set of animation objects in the central queue to play out the effects of the script. This queue naturally merges the effects of several scripts, so many animations can be affecting the scene simultaneously.

Like ASAS, though, this model does not easily provide random access to the animation. To answer the question "where is the bunny at time t?" there are two alternatives. One is to examine the script source itself. Since Alice scripts can contain arbitrary Python code, to answer this question is equivalent to solving the halting problem. The alternative is to look at the collection of animation objects created by executing the script. We would need to look at all the objects that affect the bunny's position, but that is not sufficient. We would also need to examine all the objects that could pause or unpause those that move the bunny, and all the objects that could pause or unpause *those* objects, and so on. Each of these objects has some update method attached to it, so we would need to understand what each update method was going to do. Essentially we would be running the animation from the beginning each time we wanted to know its state at some particular time.

Algorithm animation

A number of systems for producing animated representations of computer algorithms have been proposed [6, 7, 56]. These work by providing a library that allows instrumenting an implementation of the algorithm with graphical output events. Here is an example of an insertion sort animation: (from [6, Figure 5.2])

```
for i := 2 to N do
    begin
    v := a[i]; j := i;
```

```
while a[j-1] > v do
    begin
    a[j] := a[j-1]; j := j-1; OutputEvent.Swap(j, j+1)
    end;
a[j] := v;
end
```

Note that this insertion sort animation actually *is* an implementation of insertion sort. Within the call to `OutputEvent.Swap` some graphical representation of that action takes place, such as two bars swapping positions on the screen. These systems provide elaborate systems for displaying representations of key events like this, but at their core the thing that is generating the "plot" of the animation is an implementation of the algorithm. It is not clear how this approach could be applied in other fields of instruction.

While the techniques used to build these systems may not be generally applicable, a number of studies have been done to gauge the effectiveness of algorithm animation in teaching. Hundhausen *et al.* [31] surveys 24 such studies, finding that about half of them find that algorithm visualization significantly improves learning. (Of the rest, most find the difference to be statistically insignificant. Only one study found algorithmic visualization to produce worse results than a text based system.) These results lead us to be hopeful about the utility of using animation more generally, though it should be noted that many of these studies were using animated visualizations in non-lecture settings.

Flash

The most widespread commercial system for producing the kind of 2D animations we see with SLITHY is Flash, from Macromedia [34]. Flash was designed for use on the web; a freely available player and low bandwidth requirements have made it popular on a wide variety of web sites. It is mainly a 2D, vector-based keyframe animation system. Originally it had no programmability at all, but in later versions added "ActionScript," a customized version of the JavaScript language.

Flash could certainly be used to build animated presentations. A more interesting question is how well Flash supports the style of authoring we've found to be useful: building diagrams with

arbitrary parameters, then scripting how the values of those parameters change in order to create an animation.

Flash's graphical interface has two major components: the *stage* and the *timeline*. The stage represents the screen; the author can use interactive tools to draw graphical objects like lines, curves, and text. The user can then move to a different frame in the timeline and edit the graphics to define a new keyframe; Flash will automatically generate the in-between frames. The resulting object is called a *clip*. A clip may be played by itself or layered together with other clips; a typical use of ActionScript is to fire off certain clips in response to a mouse click or key press.

The interactively-drawn graphical primitives cannot be combined to form parameterized models in any meaningful way. The primitives can be grouped, but only simple changes like affine transforms and opacity and color changes can be applied to the group as a whole. To express complex relationships between the parts, we must turn to ActionScript.

While ActionScript can modify the graphics primitives used in a clip, it can only modify the simplest things: affine transform, opacity, color. It is not possible, for instance, to change one control point of an interactively specified Bézier curve from an ActionScript function, only apply a transform to the whole shape. Even the things that can be modified must be specified by writing code, just as in SLITHY—the interactive drawing tools are of no use here.

If we want to modify the drawing in nontrivial ways, we can't use the primitive objects. Instead, we can use ActionScript's imperative drawing commands (such as `lineTo`) that draw immediately on the screen. Now the ActionScript function can draw arbitrary things on the stage given a set of parameter values, but all the drawing is done by writing code. In effect, we have simply recreated SLITHY's parameterized diagrams using JavaScript instead of Python.

Now let us turn to animating the diagrams that we have built. In SLITHY this is done by writing an animation script. In Flash, the timeline UI provided by the system can't be used—it only allows the parameters of the built-in graphical primitives to be keyframed. It can't be used to drive an arbitrary ActionScript function. We could write code to compute parameter values "by hand" based on the frame number. This would be similar to the straw man `clock_animation` example at the top

of Section 3.2, on page 52. ActionScript has no facilities for describing animation—the normal use of the language is to implement per-frame or per-event callbacks that take some immediate action: drawing something in the current frame, or starting playback of one of the premade clips stored in the Flash document. An alternative would be to write our own animation system in ActionScript, or use Flash's "custom components" facility to create a new timeline UI. (The custom components mechanism allows a user interface for a new Flash component to be built in Flash itself.)

Essentially, in order to use SLITHY's authoring model in Flash, we could need to first reimplement most of SLITHY, substituting JavaScript and ActionScript's immediate drawing library for Python and OpenGL. The natural way of using Flash is by keyframing graphical primitives directly, missing all the benefits of encapsulating graphics in parameterized models.

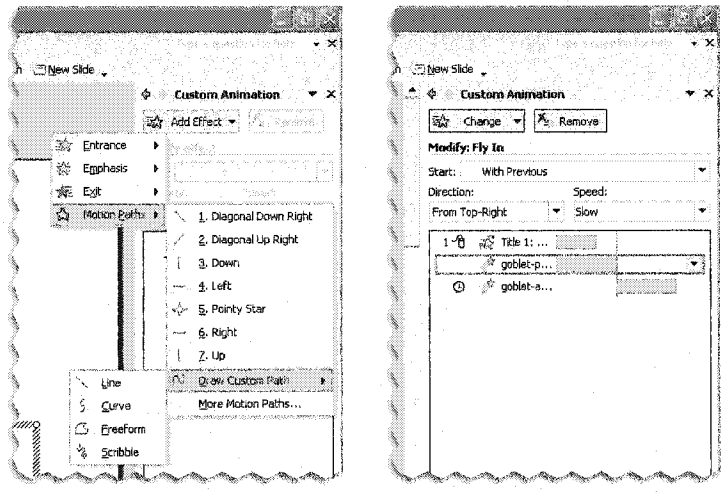
6.3 Presentation software

Finally, we will look at a few other systems that are designed or have been used for giving presentations.

PowerPoint

PowerPoint, the presentation graphics part of the Office suite from Microsoft, is by far the most commonly used piece of software for giving presentations today. It makes designing static slides very simple, through an intuitive WYSIWYG graphical interface. PowerPoint is essentially a word processor targeted at making slides. Presentation authors can interactively create and position text boxes, images, and simple graphical shapes such as lines and rectangles. It lets users edit text directly on the slide, and can perform some basic editing operations on imported images, such as cropping and brightness adjustment.

Its animation features, however, are somewhat limited in scope. Figure 6.1 shows two screenshots of PowerPoint's animation interface. PowerPoint doesn't really have animation, so much as it has "animated effects." Slide elements (or groups of elements) can slide on and off the screen in a variety of eye-catching ways. These features, with the possible exception of "motion paths,"



(a) (b)

Figure 6.1 *The animation interface of PowerPoint XP. Part (a) shows how the available animated effects are divided into four categories: entrance, emphasis, exit, and motion paths. Part (b) shows PowerPoint's "advanced timeline" view. Animations can have one of five durations, and can be set to start on a mouse click, in parallel with the previous animation in the list, or after the previous animation.*

were clearly not intended to be used to create meaningful animated diagrams, and consequently attempting to do so is difficult.

PowerPoint is, of course, a commercial product, and ease of use for the nontechnical user was certainly a primary consideration in its design. Consequently the product has animation features that cover what most business users probably want: a way to add a little motion and visual excitement to an otherwise text-centered presentation, with an interface that makes using these effects relatively straightforward. For SLITHY we made the opposite choice: trading ease-of-use for a more general purpose animation system that lets us draw and animate in a much wider variety of ways.

While our script-based authoring system has its disadvantages – most notably the awkwardness of positioning elements by typing coordinates – total programmability has advantages even outside of producing animation. Authoring a presentation often presents many repetitive tasks; the ability to easily use functions to automate tasks is useful. PowerPoint has a facility for recording and playing back macros, but these don't offer any parameterizability. To implement a more complex macro that requires arguments or user input, PowerPoint's COM interface must be used from within Visual

Basic or some other language. A significant leap in sophistication is required to go from creating simple presentations to writing macros that help automate the creation process. A user, even an experienced one, who needs to create a series of nearly identical slides in PowerPoint will often use cut-and-paste and hand editing rather than switch to Visual Basic and create a macro. SLITHY's use of one model for everything means that macros and customization are more closely tied to the basic authoring mechanism.

Pad

The Pad system of Perlin and Fox [47] was introduced not as a presentation tool but as a new metaphor for interface design. In Pad, information is arranged spatially on an infinite 2D plane. A *portal* is a viewport onto this plane; the computer screen itself is treated as a portal. Portals can roam around the plane to show any piece of information. Portals can also zoom in to magnify portions of the plane, and objects on the plane can draw themselves in different ways depending on the level of magnification. Zooming in on one day of a calendar display might reveal a list of events scheduled for that day, for instance.

While neither the original paper [47] nor the paper on its followup system Pad++ [4] specifically mention presentations, they have been used for that application. The "slides" are laid out in the Pad space, and an animated portal is used to navigate through the material. This mechanism allows for creation of some of the animations we've found to be effective: creating a large virtual canvas, expanding and compressing detail, and reinforcing the presentation structure through transitions. This last animation style particularly was explored further in the CounterPoint system.

CounterPoint

The CounterPoint system of Good and Bederson [27] is a descendent of Pad created expressly for giving presentations. It is integrated with PowerPoint, allowing the PowerPoint slides to be scaled and positioned in arbitrary locations on a large virtual plane. A path representing the order of the presentation is drawn through the slides, and animated zooms and pans are used to transition from

one slide to the next. The authors of CounterPoint argue that using this zoomable user interface paradigm in the presentation domain exploits the spatial memories of the audience members in order to assist in following the material. This argument is supported by psychological research that suggests humans encode spatial information distinctly from verbal information, so a presentation tool that takes advantage of both encoding mechanisms can lead to a better understanding and appeal to a wider variety of learning styles.

CounterPoint's animation is focused exclusively on navigation, and conveying the *structure* of the presentation. The slides themselves are ordinary PowerPoint slides; only the transitions between them are animated. In our work, we want to support the animation of *content* as well. SLITHY can be used to create both kinds of animation.

Chapter 7

CONCLUSIONS AND FUTURE WORK

In this work we've created a system for giving presentations that use animation to add *meaning* rather than novelty. SLITHY occupies a different point in the design space than applications like PowerPoint: we've given up a graphical interface and some ease of use in order to have much more variety and power in our animations. We've built on others' work in scripting systems for animation and in designing graphics libraries in order to create a useful, modern system. Our model of animation combines the advantages in precision and encapsulation of a script-based system with the testing and viewing features required for a practical production system.

Our representation of animations as time-based models allows a more flexible relationship between models and animations than is typically seen in systems designed for character animation. This unified design has proven useful in creating the kind of abstract figures and diagrams seen in the presentation domain.

Our tool has allowed us to experiment with the medium of animation in presentations. We've gained an understanding of how to use these capabilities well, and summarized our recommendations into a set of guiding principles that we believe are applicable in a wide range of domains.

This work also presents a number of interesting opportunities for the future.

Improved authoring tools. The largest obstacle to making this kind of presentation widely available is the ease of authoring. The vast majority of potential users are unable or unwilling to write code in order to create a presentation. We believe the prototype tools described in Section 3.4 have promise, but it will take a great deal of work and testing to determine if they can be made to be both easy to use and cover a sufficient part of the animation design space.

It may also be possible to make the job of developing SLITHY presentations easier for those who do use the programming interface. Our attempts to build power-assist tools for writing param-

eterized diagrams were unsuccessful, due to the arbitrary nature of user-written code that is used in diagrams. Animation scripts, however, tend to be much simpler. While things like loops, conditionals, and function calls *can* be used (and are sometimes quite useful), many animation scripts – perhaps most – are simple straight-line code. Figure 3.12 is an example of a straightforward animation script. We can imagine a timeline-style graphical interface for creating this kind of simple animation. Such an interface may ease the learning curve for new users of the system, but it's not clear that this solution makes authoring and maintaining a presentation easier in the long term.

Increased scope for interactive objects. In the current system interactions are simply animations that are edited on-line. It turns out that we can use these objects not just for putting little interactive widgets in the presentation, as we'd originally imagined, but for doing meta-level tasks like drawing annotations on the slide and controlling the playback of animations. We can imagine extending this power further, by making hooks into SLITHY's runtime system available within interactive event handlers. An interactive object could make the system jump to a completely different part of the presentation, for instance. A SLITHY presentation would not be just a fixed sequence of visuals, but a small animated hypertext system. Such a system could have applications beyond simply giving presentations—with enough structure embedded in the presentation itself, perhaps it could be handed out to students for self-study or review.

Restricted manipulation of diagrams. Another possible enhancement that could make it easier to use SLITHY as a system for creating self-study interactive applets would be if parameterized diagrams could place more restrictions on the parameter values they receive. Currently there are no such restrictions, apart from enforcing the types of the parameter values. Many diagrams, though, only produce sensible pictures with certain combinations of parameter values.

Figure 7.1 illustrates this idea with the Pythagorean Theorem diagram of Chapter 5. In this diagram, the `subdivide` parameter controls the visibility of the yellow and purple shapes, while the `slidea` and `slideb` parameters control the deformation of the two shapes. The diagram is intended to be used in a sequence where the shapes become visible, they deform one at a time to their end state, then they fade away. However, the diagram function itself imposes no restriction on

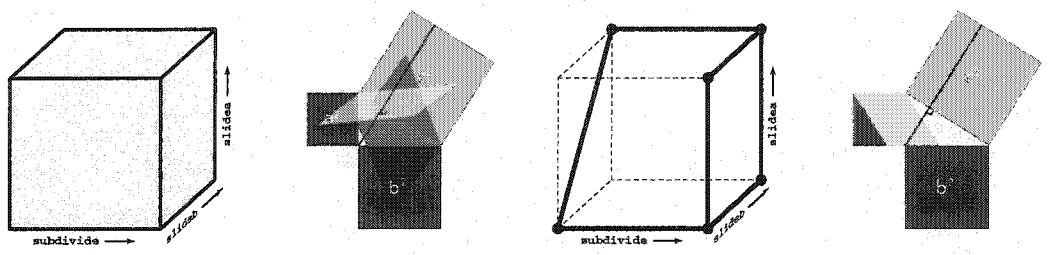


Figure 7.1 *Part (a) represents the parameter space for three of the parameters of Figure 5.2.* SLITHY *allows any combination of parameters within this cube, which can result in nonsensical diagrams (part (b)). Restricting the parameter values to a structured subspace – the thick lines of part (c) – ensures that the diagram is always in a meaningful state (part (d)).*

following this sequence. An arbitrary combination of parameter values may be passed in, resulting in a nonsensical picture like that of Figure 7.1(b). It is left up to the author of the animation script to ensure that the diagram doesn't pass through these states.

A representation like simplicial complexes [41] could be used to enable the diagram to specify which parts of its parameter space are allowed. Allowing diagrams to encode how they are intended to be used – which combinations of parameter settings make sense – could also make it easier to provide and use libraries of prepackaged diagrams.

Self-animating diagrams. Chmilar and Wyvill [12] mention the possibility of a modeling system that allows models to have intrinsic animations built in. Right now in SLITHY all time-based behavior must be driven from an animation object; parameterized diagrams keep no state of their own. It is possible to obtain the value of the computer's real-time clock from with a diagram, but we would like to have a richer set of animation primitives than manually calculating values based on a the time. This could also aid in the creation of premade diagrams libraries, by encapsulating as much behavior as possible inside the diagram itself.

Hard-copy output. For some presentation situations – most notably classroom use – the ability to produce a paper handout is important. Automatically producing a still representation of a SLITHY animation is an interesting direction for research that we have not explored. This task amounts to constructing a storyboard from the finished animation—the reverse of the usual task. Even some-

thing as simple as choosing a sequence of individual frames to convey the idea of the animation is difficult. (All the examples of Chapter 5 were done by hand.) A more complicated approach might involve using "ghosted" figures or superimposing motion lines on the images to indicate the motion within a single still image.

Appendix A

SLITHY REFERENCE GUIDE

This appendix contains reference information on all the functionality included in the SLITHY library. It is not a tutorial; the intent is to give a complete and in-depth description of the system. Each function or object method introduced is marked with a triangle, like this:

▶ **`function_name(`** *required argument*, [*optional argument* = `default value`],
`named_argument` = *named_argument*,
[**`optional_named_argument`** = `default value`] **)**

Library functions may have up to four different types of argument:

Required arguments. The caller *must* pass a value for this argument.

Optional arguments. The caller *may* pass a value for this argument. If a value is not given, the indicated default value is used.

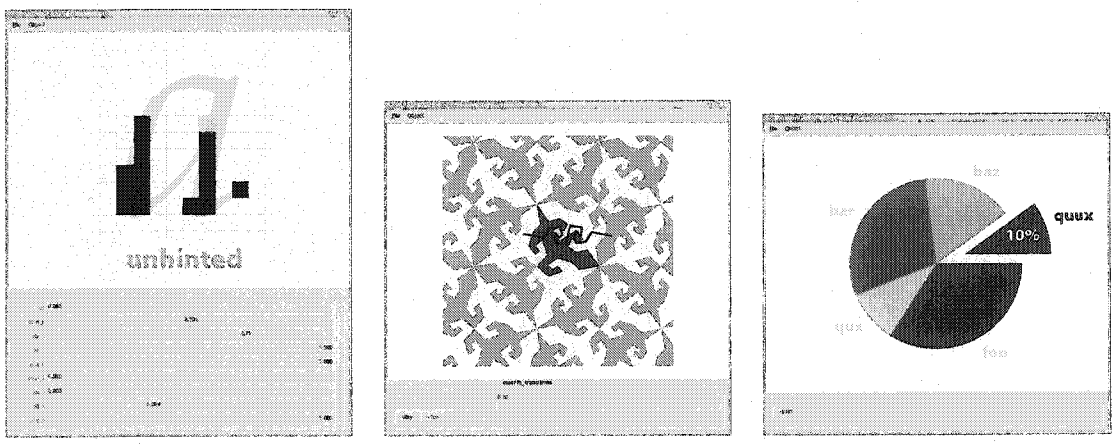
Named arguments. The caller *must* pass a value for this argument, but the value must be preceded by the argument name and an equals sign — e.g., "`foo( argname = value )`".

Optional named arguments. As the name implies, these arguments are optional but if they are used, they must be preceded with the argument name.

The order of arguments matters only for those which are not named, but all named arguments must come after any unnamed arguments. The square brackets that denote optional arguments are not part of the syntax.

A.1 Preliminaries

Parameterized diagrams, animation objects, and interactive controllers are all created by writing code in the Python language. Most of the SLITHY system is a library (or *module,* in Python termi-



(a) (b) (c)

Figure A.1 *The* SLITHY *object tester showing different kinds of drawing object. Part (a) shows a parameterized diagram; the tester provides widgets for interactively setting all the diagram's parameters. Part (b) shows an animation object in the tester; here the controls are "play" and "stop" buttons as well as a slider for scrubbing time. Part (c) shows an interactive controller; the user can use the mouse and keyboard to interact with the object.*

nology); SLITHY scripts begin by importing the contents of the library:

```
from slithy.library import *
```

The materials created for a presentation may be spread out across multiple files. The `import` statement may be used to access objects defined in another file. For instance, if the file "`file1.py`" contains the definition of the object `obj`, then another file might say

```
import file1
```

and then refer to the object as "`file1.obj`". The Python documentation [25] contains more information on using import to access objects across source files.

A.1.1 Testing objects

While developing a SLITHY presentation, it is often useful to interactively test out the objects being authored. Scripts may contain a call to the `test_objects()` function to bring up an interactive test harness. Usually this call will come at the end of the script, after all the object definitions.

▶ **test_objects(** [*object*$_1$], [*object*$_2$], [...], [**screen_size** = (800,600)], [**clear_color** = white])

The object tester can be used for parameterized diagrams, animation objects (or lists of animation objects), and interactive controllers. Figure A.1 shows screenshots of the tester with each of these types of object. `test_objects()` can only be called once in a script, but many objects may be passed to it and the user can interactively choose which to show. The optional arguments are used to set the initial window size and background color.

A.2 Rectangles

SLITHY uses oriented rectangles for a number of different purposes, including positioning elements in animations and describing cameras within parameterized diagrams. Rectangles are represented using the `Rect` data type. This section describes functions for creating and manipulating these objects.

The `Rect` type is a subclass of the Python `tuple` type. All rectangles are represented as 5-tuples (o_x, o_y, w, θ, a), where (o_x, o_y) is the lower-left corner of the rectangle, w is its width, θ the angle of its baseline with respect to the x-axis, and a its aspect ratio (width divided by height). This representation is illustrated in Figure A.2.

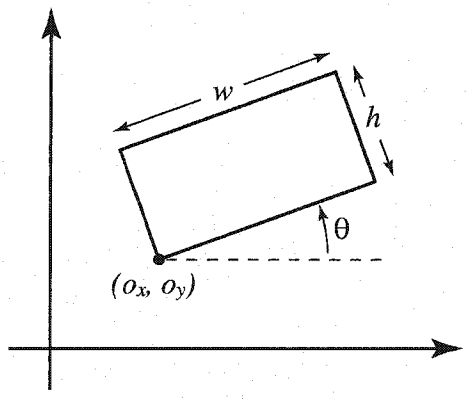


Figure A.2 *The five parameters defining a rectangle in* SLITHY. *Note that the rectangle type stores the aspect ratio w/h instead of the height itself.*

Rectangle objects are created by calling the `Rect()` function, which has a few different forms:

- **`Rect(`** *ox, oy, width, theta, aspect* **`)`**

This form constructs a rectangle object from the five-parameter representation used internally (see Figure A.2).

- **`Rect(`** *x1, y1, x2, y2* **`)`**

This form constructs an axis-aligned rectangle given the coordinates of two diagonally opposite points.

- **`Rect(`** *x, y,* **`width`** `=` *width,* **`height`** `=` *height,* `[` **`anchor`** `=` `'c'` `]` **`)`**

This form constructs an axis-aligned rectangle of the given height and width. By default it is centered on the point (x, y), but this can be changed with the optional anchor parameter. An anchor value of `nw`, `ne`, `sw`, or `se` will place the appropriate corner of the rectangle at (x, y), while specifying `n`, `s`, `e`, or `w` centers the corresponding side on (x, y).

`Rect` objects also have a number of methods that can be called to create new rectangles. These are illustrated in Figure A.3. Note that calling one of these methods on a rectangle does not modify the original rectangle, but instead returns a newly constructed `Rect` object. `Rect` objects themselves are immutable; once created their values can never be changed.

- **`r.top(`** *f* **`)`**
- **`r.bottom(`** *f* **`)`**
- **`r.left(`** *f* **`)`**
- **`r.right(`** *f* **`)`**

Each of these methods creates a rectangle by taking a fraction f of the rectangle r along the specified side, for $f \in (0, 1]$.

- **`r.move_up(`** *f,* `[` **`abs`** `=` `0` `]` **`)`**
- **`r.move_down(`** *f,* `[` **`abs`** `=` `0` `]` **`)`**
- **`r.move_left(`** *f,* `[` **`abs`** `=` `0` `]` **`)`**
- **`r.move_right(`** *f,* `[` **`abs`** `=` `0` `]` **`)`**

These methods creates a rectangle by sliding the rectangle r by f of its extent in the indicated direction. If the optional argument `abs` is true, then f is interpreted as a distance rather than a fraction of the original rectangle's size.

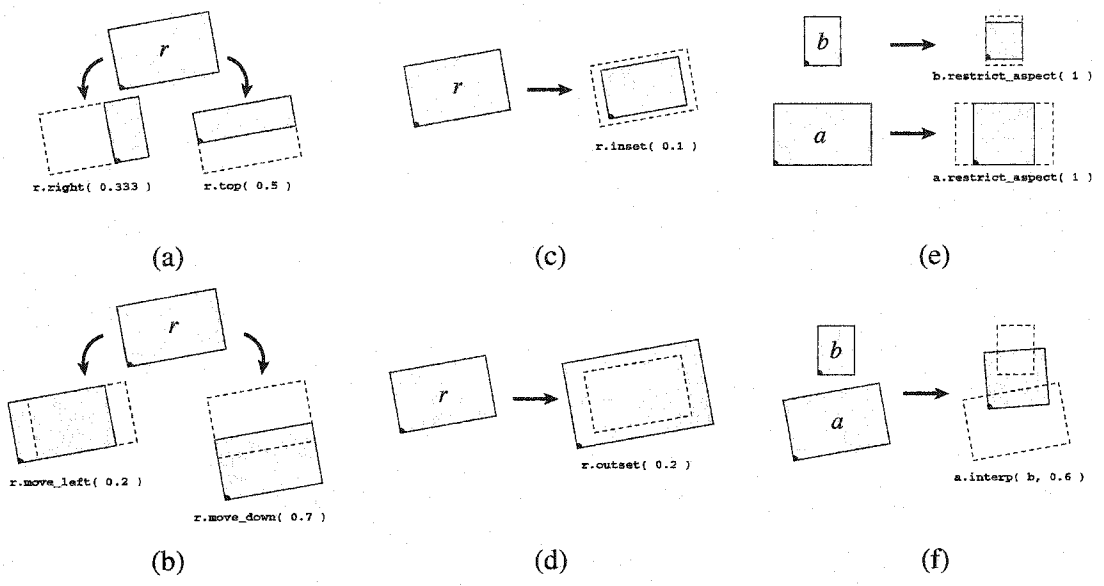


Figure A.3 *Methods that can be applied to* `Rect` *objects to generate new rectangles.*

▶ **`r.inset(`** *f*, **`[abs = 0])`**
▶ **`r.outset(`** *f*, **`[abs = 0])`**

These methods generate a new rectangle by moving all four sides of the rectangle *r* in (or out) by the given fractional amount *f*. (For `inset()`, *f* should not exceed 0.5—trying to create a degenerate rectangle will result in an error message.) The optional parameter `abs`, if true, will cause *f* to be interpreted as an absolute distance instead of a fraction.

▶ **`r.restrict_aspect(`** *a* **`)`**

This method generates a new rectangle that has the same center and orientation as the rectangle *r*, has an aspect ratio of *a* (width divided by height), and is as large as possible. The effect is to slice an equal amount off the top and bottom sides or the left and right sides of the original rectangle to achieve the required aspect ratio.

▶ **`r.interp(`** *other*, *f* **`)`**

This method interpolates *f* of the way from the rectangle *r* towards the rectangle *other*. When *f* is zero, the original rectangle *r* is returned. When *f* is one, the rectangle *other* is returned, and intermediate values of *f* produce a smooth interpolation between the two.

▶ **`r.width()`**

▶ **r.height( )**
▶ **r.aspect( )**

These convenience methods do not return new rectangles, but rather just return the width, height, or aspect ratio of the rectangle *r*.

A.3 External resources

There are two types of external resources that SLITHY presentations can make use of: fonts and images. These are called external resources because they are not stored in the SLITHY script itself but are loaded at runtime from files on disk. This section details how to load these items.

A.3.1 Using fonts

▶ **load_font(** *filename*, *pixels* **)**

This function loads a TrueType (`.ttf`) or Type 1 (`.pfb`) font, returning a *font object* that represents that font. This object is used anywhere SLITHY requires a font: the `text()` function within parameterized diagrams, the `font` parameter of `Text` elements in animation scripts, etc.

Slithy draws text by rendering one instance of each character into an OpenGL texture map, then drawing textured polygons on the screen. The *pixels* argument is used to specify the height of characters in the texture map. This value should be approximately the height of the characters as they will be seen on the screen. Too small a value will result in poor-looking text; too large will consume a great deal of texture memory and slow down rendering (and may also look bad). 50 is a good default value for most presentation-sized text.

It is acceptable to load the same font file multiple times with different values for *pixels* if a font is needed at widely-varying sizes. The call will return a different font object; make sure to use the right one depending on what size text is needed.

The *filename* may be an absolute pathname, or a pathname relative to the current directory. To make it easier to relocate presentations, it is recommended that the `search_font()` function described below be used instead of `load_font()`.

▶ **fontpath**

▶ **search_font(** *filename, pixels* **)**

`search_font()` functions just like `load_font()`, except that it searches for *filename* in each of the directories in the list `fontpath` before looking in the current directory. `fontpath` is a Python list that is initially empty; it can be manipulated with any of the standard list operations, such as"`fontpath.append(` *pathname* `)`".

▶ **add_extra_characters(** *unicodestring* **)**

Normally, when SLITHY loads a font, it looks for only characters in the Unicode range U+0000 to U+00FF (the "Basic Latin" and "Latin-1 Supplement" pages). This is sufficient to cover English and many other Western European languages. Characters that were not loaded will be silently dropped whenever SLITHY renders a string of text. To request characters not in the default range, the `add_extra_characters()` can be called with a string of additional characters to search for in the font file. For example,

```
add_extra_characters( u'\u0107' )
```

will cause SLITHY to look for character U+0107 ("LATIN SMALL LETTER C WITH ACUTE") in every subsequently loaded font file. The call to `add_extra_characters()` must come *before* any calls to `load_font()` or `search_font()` to have effect. Note also that there is no guarantee that a requested character will be available; most fonts contain only a subset of the characters defined by Unicode, and a particular font might lack a requested character.

A.3.2 Using bitmap images

▶ **load_image(** *filename* **)**
▶ **search_image(** *filename* **)**
▶ **imagepath**

These two functions for loading bitmap images work analogously to their font-loading counterparts, with `search_font()` using the list `imagepath` in place of `fontpath`. The return value is an *image object* that can be used anywhere an image is required. SLITHY, through the Python Imaging Library, can load most common image formats, including JPEG, PNG, Targa, BMP, and

GIF. If the image file includes an alpha channel, SLITHY will make use of it when drawing the image on the screen.

A.4 Colors

SLITHY uses *color objects* to represent colors in parameterized diagrams and animation scripts. A number of commonly used colors are predefined in the library:

red **orange** **yellow** **green**

blue **purple** **black** **white**

In addition to these standard objects, new color objects can be created with the `Color()` constructor:

- **`Color(`** *gray*`,` [*alpha* = `1.0`] **`)`**
- **`Color(`** *red*`,` *green*`,` *blue*`,` [*alpha* = `1.0`] **`)`**
- **`Color(`** *color object*`,` [*alpha* = `1.0`] **`)`**

These constructors create `Color` objects from individual component values (which range from 0.0 to 1.0), or from pre-existing color objects. When the last form is used, if the *alpha* argument is given then its value is used instead of the alpha value of the input color object.

- **`hsv(`** *hue*`,` *saturation*`,` *value*`,` [*alpha* = `1.0`] **`)`**

This function constructs a `Color` object using the HSV color space. All three components (hue, saturation, and value) range from zero to one.

- **`c1.interp(`** *c2*`,` *frac* **`)`**

`Color` objects have an `interp` method that interpolates between two colors. For instance,

```
red.interp( blue, 0.25 )
```

returns a new color object that is 25% of the way from red to blue (in the RGB color space).

A.5 Parameterized diagrams

Parameterized diagrams are written as ordinary Python functions, using the `def` keyword. The diagram's parameters are expressed as the function's arguments, and the default for each argument

Table A.1 *Parameter types available within parameterized diagrams. The "Tester?" column indicates if a parameter can be manipulated interactively in the object tester. The "Interpolate?" column indicates if* SLITHY *can interpolate between values of this type—a requirement for using commands like* `linear()` *and* `smooth()` *to animate the parameter. Non-interpolatable values can only be changed within animations by the* `set()` *function.*

Type	Format	Tester?	Interpolate?
real value	(**SCALAR,** *min*, *max*, [*default* = *min*])	Y	Y
integer	(**INTEGER,** *min*, *max*, [*default* = *min*])	Y	Y
color	(**COLOR,** [*default* = `black`])	Y	Y
string	(**STRING,** [*default* = `''`])	Y	N
Boolean	(**BOOLEAN,** [*default* = 0])	Y	N
object	(**OBJECT,** [*default* = `None`])	N	N

is used to express its type as well as its default value. Here is an example:

```
def sample( name = (STRING, 'hello'),
            number = (SCALAR, 0, 10, 3.5),
            yesno = (BOOLEAN, 1) ):
    . . .
```

This code starts defining a parameterized diagram called "sample" that has three parameters: a string value called "name," a real-valued number called "number", and a Boolean value called "yesno." These parameters have default values of "hello," 3.5, and *true,* respectively. The other two values given for the "number" parameter are used to set the endpoints of the controlling slider when this object is loaded into the object tester. Note that these min and max values are used *only* in the object tester; an animation script that contains this diagram may provide values for "number" that lie outside this range.

Parameter names can be any legal Python identifier that does not begin with an underscore. Names beginning with underscores may conflict with SLITHY's internal workings and produce undefined behavior.

Table A.1 summarizes the available parameter types. The `COLOR` type defines a parameter whose value is a color object as defined in Section A.4. The `OBJECT` type can be used to pass an arbitrary Python object to a diagram.

A.5.1 Graphics state

The following commands do not draw anything, but are used to manipulate and query the state of the drawing library.

▶ **`set_camera(`** *rectangle* **`)`**

SLITHY's drawing commands affect an infinite virtual canvas. This function is used to specify what portion of that canvas is mapped onto the viewport (which may be the whole screen, or a smaller region if the diagram is included in an animation). The argument is a `Rect` object, as described in Section A.2. This "camera rectangle" is centered in the viewport and made as large as possible. If the aspect ratio of the camera rectangle does not match that of the viewport, then some of the canvas outside the camera rectangle will also be visible (as strips along the top and bottom, or left and right, of the viewport.)

Typically this function will be called just once at the top of a parameterized diagram function, but the camera can be changed in the middle of a function as well.

The default camera rectangle is centered on the origin, has a height of 2 units, and has the same aspect ratio as the viewport.

▶ **`camera( )`**
▶ **`visible( )`**

`camera()` returns the current camera rectangle. `visible()` returns a rectangle that fills the viewport exactly. This rectangle will always contain the camera rectangle, but will be larger if the camera and viewport aspect ratios do not match.

`set_camera()` can be thought of as controlling a mapping from the virtual canvas "world coordinates" into the viewport on the screen. A second transform matrix, analagous to the model-view matrix in OpenGL or the CTM in PostScript, is used to map from the "user coordinates" given in primitive drawing functions to world coordinates. The effects of all these functions is illustrated in Figure A.4.

▶ **`translate(`** t_x**`,`** t_y **`)`**

This function translates the user coordinate system by (t_x, t_y).

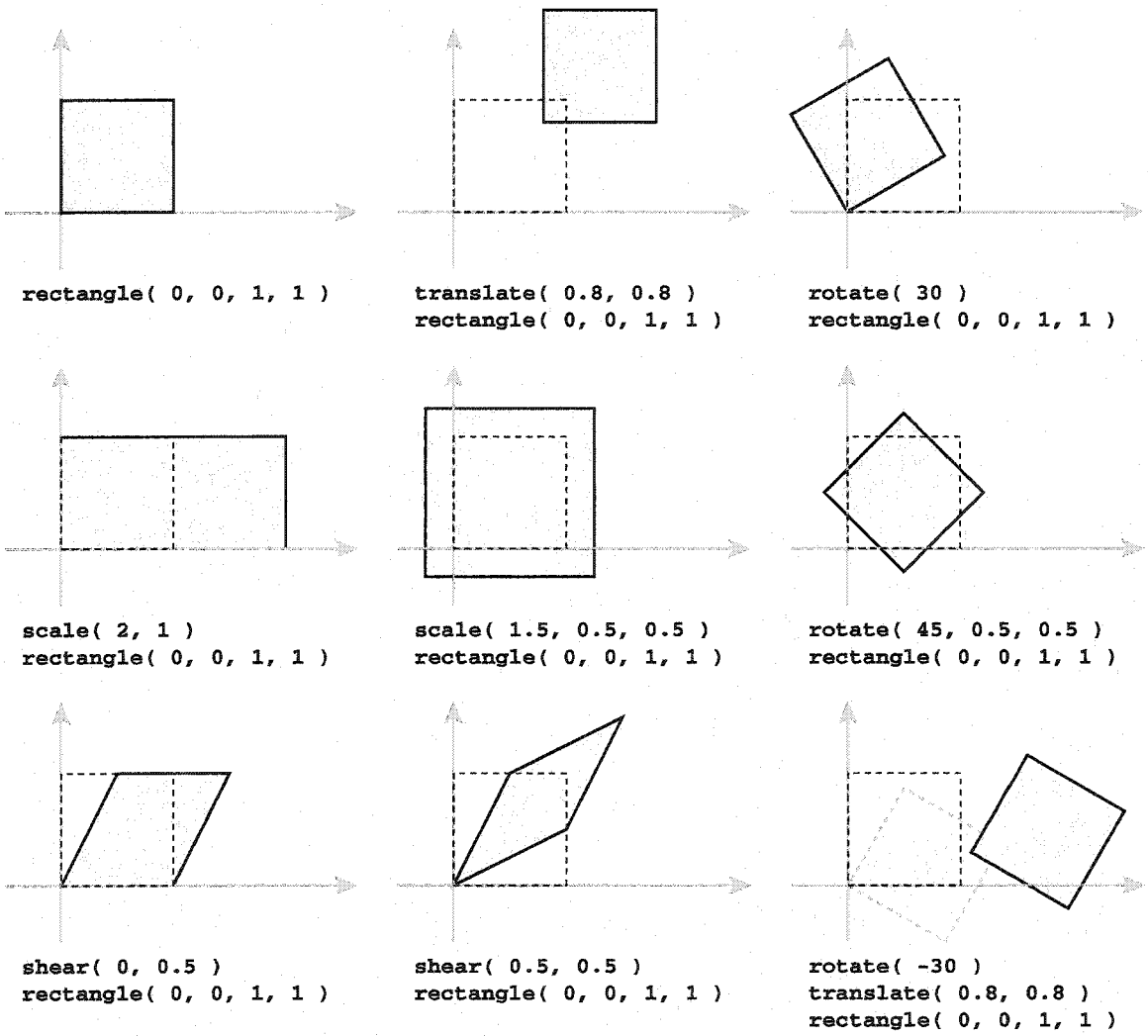


Figure A.4 *Illustrations of various coordinate system transforms applied to a square.*

▶ **rotate(** *deg*, [$c_x = 0.0$], [$c_y = 0.0$] **)**

This function rotates the user coordinate system by *deg* degrees counterclockwise. The center of the rotation is the point (c_x, c_y); if these are omitted then rotation is about the origin.

▶ **scale(** s **)**
▶ **scale(** s_x, s_y **)**
▶ **scale(** s, c_x, c_y **)**
▶ **scale(** s_x, s_y, c_x, c_y **)**

This function scales the user coordinate system. The forms with an s argument do uniform scaling, while those with both s_x and s_y allow different scale factors in the x and y directions. If c_x and c_y are specified, then scaling is relative to the point (c_x, c_y), otherwise the scaling is relative to the origin.

▶ **shear(** h_x, h_y **)**

This function shears the user coordinate system by the given amounts. Shearing is always relative to the origin.

▶ **push( )**
▶ **pop( )**

Like other graphics systems, SLITHY maintains a stack of graphics states to allow for easy modification and restoration of state. The SLITHY graphics state includes the current user transform matrix, as well as the current drawing color and line thickness. The `push()` function saves a copy of the current state on the stack. Calling `pop()` pops a state off the stack and uses it to replace the current state, which is discarded. These functions are analagous to the `gsave` and `grestore` operators in PostScript.

▶ **color(** *color_object* **)**
▶ **color(** *color_object*, *alpha* **)**
▶ **color(...)**

The `color()` function sets the current drawing color, which is initially black. The color can be specified in a number of different ways: as a color object, a color object multiplied by an additional alpha, or using any of the ways to construct color objects as described in Section A.4. Here are some examples:

```
color( red )                # red
color( blue, 0.5 )          # blue, alpha = 0.5
color( 0 )                  # black
color( 0.2, 0.5 )           # dark grey, alpha = 0.5
color( 0, 1, 0 )            # green
color( 1, 1, 0, 0.3 )       # yellow, alpha = 0.3
```

▶ **thickness(** *t* **)**

This function sets the current line thickness to *t*. Line thickness is drawn in user space, so it is affected by the current user transform matrix. Applying a scale to the coordinate system (including a nonuniform scale) will affect the appearance of lines on the screen.

▶ **id(** *id* **)**

This function sets the current drawing object ID. When SLITHY draws a shape, in addition to writing pixels of the current color into the framebuffer, it writes "pixels" of the current ID into an invisible ID buffer, which can be later queried to determine what was drawn at a particular point.

IDs are nonnegative integers. The range available depends on the number of bits in the depth buffer; a 16-bit depth buffer will allow object IDs ranging from 0 to 16383. If the `id()` function is called with a negative argument, writing into the ID buffer is disabled, so subsequent drawing commands do not change it. Initially the buffer is initialized to all zeroes.

An example of using the object ID buffer appears in Section A.7.1 on page 180.

A.5.2 Drawing functions

SLITHY provides a number of simple primitives for drawing on the diagram canvas. These are illustrated in Figure A.5. All the functions (except `image()`) use the current drawing color.

▶ **line(** x_1**,** y_1**,** x_2**,** y_2**,** `[...]` **)**
▶ **polygon(** x_1**,** y_1**,** x_2**,** y_2**,** `[...]` **)**

`line()` and `polygon()` take a series of coordinates and draw a polyline and a filled polygon, respectively. `line()` makes use of the current line thickness; the drawn stroke is centered on the ideal mathematical line.

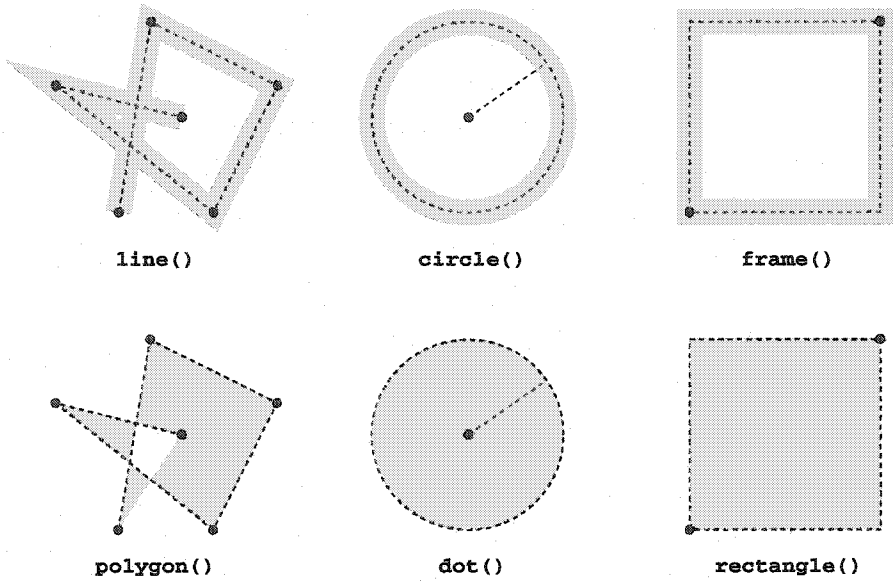


Figure A.5 Primitive drawing shapes available in SLITHY. *The red dots indicate the coordinates passed to each function; the gray areas are what is actually drawn.*

▶ **circle(** *r*, [*x*=0.0], [*y*=0.0] **)**
▶ **dot(** *r*, [*x*=0.0], [*y*=0.0] **)**

`circle()` and `dot()` draw outlined and filled circles, respectively. Outline circles are drawn using the current line thickness and the stroke is centered on the ideal circular path. The *x* and *y* parameters specify the center of the circle.

▶ **frame(** x_1, y_1, x_2, y_2 **)**
▶ **frame(** *rect* **)**
▶ **rectangle(** x_1, y_1, x_2, y_2 **)**
▶ **rectangle(** *rect* **)**

Rectangles can be drawn from either the coordinates of diagonally-opposite corners, or from a `Rect` object as described in Section A.2. `rectangle()` draws a solid filled rectangle, while `frame()` draws an outline rectangle using the current line thickness.

▶ **text(** *x*, *y*, *text*, *font*, [**size** = 1.0], [**justify** = 0.0],
[**anchor** = 'c'], [**wrap** = -1.0], [**nodraw** = 0] **)**

The `text()` function is used to draw text on the screen. The first four arguments are required:

- *x* and *y* specify a position on the canvas. By default the bounding box of the text string is centered on this point, but this behavior can be changed with the `anchor` parameter.

- *font* is a font object, as returned by the `load_font()` and `search_font()` functions (see Section A.3.1).

- *text* is the text to be drawn. In its simplest form it can just be a string (or Unicode string). The more complex form is a list containing:

  - strings,
  - font objects, to change the font,
  - color objects, to change the color,
  - `RESETFONT`, to return to the original font,
  - `RESETCOLOR`, to return to the original color,
  - `RESET`, which combines `RESETFONT` and `RESETCOLOR`.

  We will call this kind of list a *text list* — every text-drawing facility in SLITHY can take this kind of list instead of a simple string.

Here is a simple example:

```
text( 0, 0, 'Hello, world!', romanfont )
```

Assuming that `romanfont` is a valid font object, this code will draw the string "Hello, world!" in the current drawing color, centered at the origin. Here is a more complex example, which uses a text list instead of a single string:

```
text( 0, 0,
      [ 'This is ', red, italicfont, 'not', RESET, ' a test' ],
      romanfont )
```

Assuming that `italicfont` is also a valid font object, this code will draw the phrase "This is not a test" centered at the origin. The word "not" will be drawn in red (and a different font), while the rest of the string will be drawn in the current drawing color.

`text()` can also take a number of optional arguments:

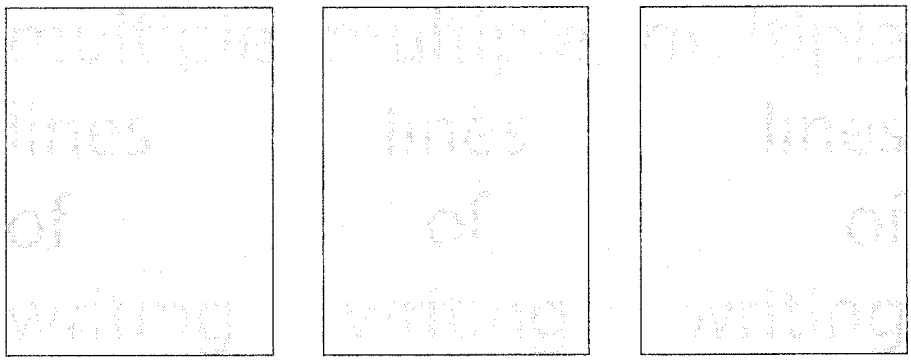


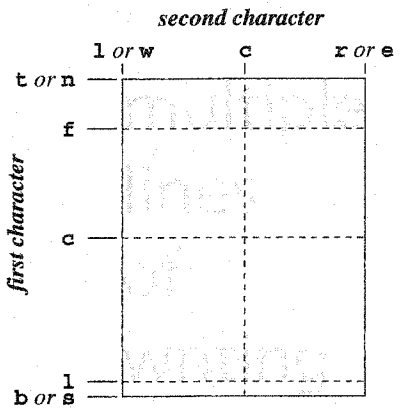
Figure A.6 *The* justify *parameter controls the horizontal positioning of multiline text within the bounding box.*

- *size* specifies the em-height of the text in user-space units.

- *justify* controls the justification when the text has multiple lines. A value of 0.0 means left justification, 0.5 centers each line horizontally within the bounding box, and 1.0 pushes each line over to the right. The effect of this value is illustrated in Figure A.6. Intermediate values interpolate between these three positions. This parameter has no effect on the positioning of the bounding box relative to the reference point; that is controlled by the *anchor* parameter described below. *justify* only controls the positioning of different lines of text within the bounding box.

- *wrap* controls word-wrapping. If this parameter is less than zero, no wrapping is done—line breaks occur only where explicit newline characters appear in the text string. A positive value causes line breaks to be inserted at spaces in the string so that no line is longer that *wrap* units long. (However, if an individual word is wider than this parameter's value, it will not be broken.)

- *anchor* specifies how the text is positioned relative to the reference point (x, y). A value of `c` means that the text bounding box will be centered on the point. Values of `n`, `s`, `e`, and `w` place



first character	
t, n	top of bounding box
f	baseline of first line
c	vertical center of bounding box
l	baseline of last line
b, s	bottom of bounding box

second character	
l, w	left edge of bounding box
c	horizontal center of bounding box
r, e	right edge of bounding box

Figure A.7 *The two-character form of the anchor parameter to* `text()` *can select one of 15 possible placements of the text relative to the reference point.*

the point at the midpoint of the north, south, east, or west sides, respectively. (`t`, `b`, `r`, and `l` are synonymous with `n`, `s`, `e`, and `w`, for those who like to think in terms of top-bottom-left-right.) A two-character value can be used to select other positionings; see Figure A.7 for details.

- The return value of the `text()` function is a six-element Python dictionary containing the information about the text's bounding box. The `'left'` and `'right'` keys in this dictionary index the *x*-coordinates of the bounding box's left and right sides. Similarly `'top'` and `'bottom'` store the *y*-extents of the bounding box, and `'width'` and `'height'` store its size. This information can be used to position other drawings relative to the text.

  The *nodraw* parameter, if set to true, suppresses all drawing, so that computing and returning this dictionary is the *only* effect of calling `text()`.

▶ **`image(`** *x, y, imageobj,* **`[width = None], [height = None], [anchor = 'c'], [alpha = 1.0] )`**

The `image()` function draws bitmap images on the canvas. The *imageobj* parameter is an image object representing the image to be drawn; these objects are returned by the `load_image()` and `search_image()` functions described in Section A.3.2. *x* and *y* position the image on the

canvas. By default the image is centered on this reference point, but this behavior can be changed with the optional *anchor* parameter.

The *anchor* parameter works just as it does for the `text()` function (see Figure A.7), omitting those anchor positions that refer to text baselines. Alternatively, the `image()` *anchor* parameter may be given as a 2-tuple of numbers to allow for continuously variable positioning of the image. An *anchor* value of "`(0.0, 0.0)`" is equivalent to `'sw'`, while "`(1.0, 1.0)`" is equivalent to `'ne'`.

The *width* and *height* parameters control the size of the image drawn. If both are omitted then the image is drawn one unit wide, with the height scaled to preserve its aspect ratio. If exactly one of these parameters is given, then the other is scaled to match. By specifying both a nonuniform scaling of the image can be obtained. Passing in the value `None` for either is equivalent to omitting it entirely.

Like the `text()` function, `image()` returns a six-element dictionary containing information about the drawn image's extents and size.

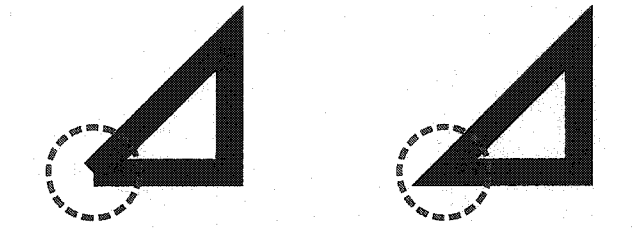
Path objects

For drawing more complex or frequently-repeated shapes, the `Path` object can be used. A `Path` object stores a path made up of straight and curve segments. The methods for constructing these paths are similar to the drawing operators of PostScript. A path object can then be instanced in the diagram in a filled or outline style. A path may consist of multiple subpaths, each of which can be open or closed.

▶ **`Path( )`**

An empty path object is created by calling the `Path()` constructor, which takes no arguments. The geometry of the path is created by calling methods of the path object — each method appends a segment to the path. Path objects contain a *current point,* which is where the next segment will begin. The current point is initially undefined.

▶ **`p.moveto(`** *x, y* **`)`**



```
p = Path.moveto(0,0)
        .lineto(1,0)
        .lineto(1,1)
        .lineto(0,0)
widestroke( p, 0.2 )
```

```
p = Path.moveto(0,0)
        .lineto(1,0)
        .lineto(1,1)
        .closepath()
widestroke( p, 0.2 )
```

Figure A.8 *The right image shows a path that has been closed with the* `closepath()` *method. The first segment is joined to the last. The left image illustrates a path where a line segment back to the origin has been added, but the path is left open. No join is drawn.*

▶ **`p.rmoveto(`** *dx*, *dy* **`)`**

`moveto()` begins a new subpath by moving the current point to the location (x, y). The `rmoveto()` method is identical except that the arguments are interpreted as relative to the current point at the time of the call.

▶ **`p.lineto(`** *x*, *y* **`)`**
▶ **`p.rlineto(`** *dx*, *dy* **`)`**

Both of these methods append a line segment from the current point to the a new location, and move the current point to that location. `lineto()`'s arguments are absolute coordinates, while `rlineto()`'s are taken as offsets from the current point.

▶ **`p.closepath( )`**

This method closes the current subpath by appending a line segment from the current point back to the start of the subpath (that is, the destination of the most recent `moveto()` or `rmoveto()`). The current point becomes undefined. Closing a path is different from simply doing a `lineto()` back to the starting point in that closed paths are drawn with the first and last segments joined together, while open paths (those not ended with a call to `closepath()`) are not. Figure A.8 illustrates this distinction.

▶ **`p.curveto(`** x_1, y_1, x_2, y_2, x_3, y_3 **`)`**
▶ **`p.rcurveto(`** dx_1, dy_1, dx_2, dy_2, dx_3, dy_3 **`)`**

`curveto()` draws a cubic Bézier segment from the current point to (x_3, y_3), with (x_1, y_1) and (x_2, y_2) as the two off-curve control points. `rcurveto()` is the relative form of the command, with all arguments taken as offsets from the initial current point. The endpoint of the curve becomes the new current point.

▶ **p.qcurveto(** x_1, y_1, x_2, y_2 **)**
▶ **p.rqcurveto(** dx_1, dy_1, dx_2, dy_2 **)**

These methods are analagous to `curveto()` and `rcurveto()`, except that they draw a quadratic rather than a cubic Bézier segment. They require only a single off-curve point.

▶ **p.arc(** c_x, c_y, *startangle*, *endangle*, r **)**
▶ **p.arcn(** c_x, c_y, *startangle*, *endangle*, r **)**

These methods are used to draw circular arcs, centered at (c_x, c_y) with radius r. `arc()` produces an arc that runs counterclockwise from *startangle* to *endangle,* while `arcn()` produces a clockwise arc. Both angles are specified in degrees.

If the current point is undefined when `arc()` or `arcn()` is called, an implicit `moveto()` is done to set the current point to the start point of the are. If the current point is defined, an implicit `lineto()` the start of the arc is added instead. (Any zero-length line segments produced will be culled.) The end point of the arc always becomes the new current point.

All of these methods return a reference to the path object, so that method calls may be easily chained together. For example, these two blocks of code construct the same path, a unit square:

```
p1 = Path()
p1.moveto(0,0)
p1.lineto(1,0)
p1.lineto(1,1)
p1.lineto(0,1)
p1.closepath()

p2 = Path().moveto(0,0).lineto(1,0).lineto(1,1).lineto(0,1).closepath()
```

Figure A.9 illustrates a path with all the different types of segment, constructed using the chaining technique.

▶ **fill(** *path* **)**

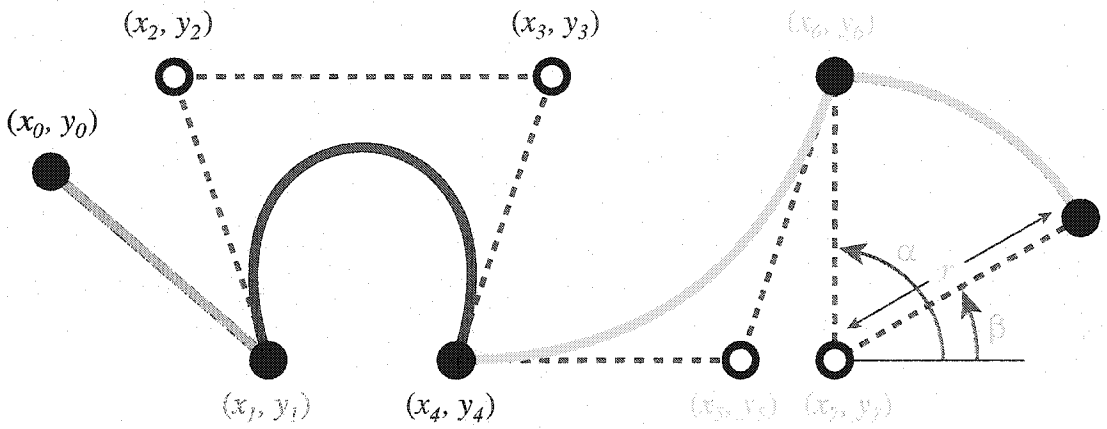


Figure A.9 *Construction of a path object containing line, cubic and quadratic Bézier, and circular arc segments.*

▶ **widestroke(** *path*, *width* **)**

These functions are used to draw a path object on the screen. `fill()` draws the path as a filled shape, implicitly closing every open subpath. `widestroke()` draws a stroke of the specified width along the path. Both functions use the current drawing color.

Since drawing a path can be computationally expensive, SLITHY caches the OpenGL commands needed to fill or stroke a particular path object in a display list. To take best advantage of this, path objects that are used repeatedly should be constructed *outside* any diagram function, like this:

```
thepath = Path().moveto(...).lineto(...).curveto(...)   # etc.

def my_diagram( param1 = (SCALAR,0,1),
                param2 = (STRING) ):
    . . .
    fill( thepath )     # draw the path here
    . . .
```

Using path objects this way means that SLITHY incurs the expense of converting the path to a set of triangles just once — the first time the path is drawn. Code structured this way:

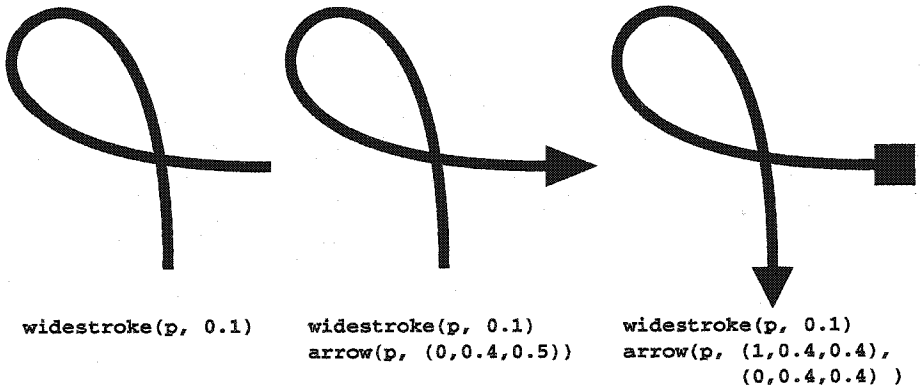


Figure A.10 *The* `arrow()` *function is used to draw arrowheads of various styles at the ends of a path drawn with* `widestroke()`.

```
def my_diagram( param1 = (SCALAR,0,1),
                param2 = (STRING) ):
    . . .
    thepath = Path().moveto(...).lineto(...).curveto(...)    # etc.
    fill( thepath )     # draw the path here
    . . .
```

means that the path object is constructed, converted to triangles, drawn, and discarded every time the diagram is redrawn. Of course, if the path's shape depends on the diagram's parameters, then there is no alternative to constructing a new path on each redraw. For paths that are static, though, it is best to take advantage of the caching behavior wherever possible.

▶ **`arrow(`** *path*, *endarrow*, [*startarrow* = `None`] **`)`**

The `arrow()` function does not draw the path itself, but draws arrowheads on either or both ends of each open subpath of a path object. In combination with `widestroke()` this can be used to draw a path as an arrow.

The *endarrow* and *startarrow* parameters are each either the value `None`, or a 3-tuple (*style*, *width*, *length*). *style* can be either 0 to draw a triangle or 1 to draw a rectangle. The arrowheads are always drawn in the current drawing color. Some examples of arrowheads are shown in Figure A.10.

▶ **`embed_object(`** *viewport*, *object*, *parameter dictionary*, [**`clip`** = `1`],
[**`_alpha`** = `1.0`] **`)`**

With this function, parameterized diagrams can incorporate the output of other drawing objects (animation objects and other parameterized diagrams). The *viewport* argument, which must be a `Rect` object, specifies where on the diagram canvas the object will be drawn. The second argument is the object itself. The third argument is a dictionary containing parameter values to be passed to the drawn object. For animation objects, this dictionary will have a single key `'t'` whose value is a number. For paramaterized diagrams, the number, name, and types of the parameters will depend on the diagram itself.

The optional argument *clip*, if false, lets the included object draw outside the boundary of the viewport. By default the object's drawing is clipped to the viewport rectangle. The other optional argument *_alpha* is can be used to control the overall opacity of the embedded object by applying a scaling factor to the alpha value of everything it draws.

A.5.3 Miscellaneous functions

There are a few more functions included for use within parameterized diagrams that don't fall into any of the above sections.

▶ **`clip(`** *rect* **`)`**
▶ **`unclip( )`**

`clip()` restricts subsequent drawing to occur only within the given rectangle (specified as a `Rect` object). Multiple calls to clip stack, so that drawing only happens within the intersection of all the clip rectangles. Calling `unclip()` cancels the effect of the most recent call to `clip()`.

▶ **`mark(`** *name* **`)`**

The `mark()` function takes a single string argument and stores the current user coordinate system and camera away in an internal dictionary under that name. The object tester can then access that dictionary and use it to project screen coordinates back into the user coordinate space. This mechanism provides a convenient way to position drawings within a diagram, even after a series of coordinate system transformations: insert a call to `mark()` into the desired place in the diagram code, and load the diagram into the tester. Clicking in the test window will then display the

coordinates of the mouse position, translated back into the coordinate system in effect at the time of the `mark()` call.

`mark()` also returns a token representing the current coordinate system that can be used with the `unproject()` function described below.

- **`project(`** *x, y* **`)`**
- **`unproject(`** *x, y* **`)`**
- **`unproject(`** *x, y, token* **`)`**

`project()` takes a point in the current user space and returns its coordinates in screen space (i.e., in pixels). `unproject()` does the reverse, transforming pixel coordinates into user space. The two-argument form returns coordinates in the *current* user space, and consequently can be used only within a parameterized diagram function (where the notion of user space is well-defined). `unproject()` is also available in a three-argument form, where the third argument is a token returned by the `mark()` function, and returns coordinates in user space at the point of the call to `mark()`. This form can be used anywhere within a presentation.

These functions are useful mostly in writing interactive objects, and will be illustrated in more detail further below, in Section A.7.1.

A.6 Animation objects

Animations in SLITHY function very much like parameterized diagrams that happen to have just a single real-valued parameter called *t*. It would be possible to create an animation by writing a diagram function as described in the previous section. Such a function would have to start with the input value of *t* and compute from scratch everything that should be displayed for that point in time. This way of constructing an animation would be extremely tedious and error-prone. SLITHY provides another way of putting together an animation, by creating *animation objects*.

Animation objects function just like parameterized diagrams, but instead of the author writing the code that is executed on each redraw, the author writes code that *describes* the desired animation by creating a set of elements that are used in the animation and specifying how they change over time. This piece of "description code" is called an *animation script*. Executing an animation script

produces an animation *object,* which the system can then play back.

The animation object contains a timeline for each parameter of each element that tells what that parameter's value should be at each point in time. An animation script therefore consists of two major sections: first creating and setting up the elements, then describing the parameter timelines. Here is a small sample animation script:

```
def sample_animation():

    # create elements
    bg = Fill( color = black )
    tx = Text( get_camera(), text = 'hello, world!',
               font = fonts['roman'], size = 40, color = white,
               justify = 0.5, vjustify = 0.5 )

    start_animation( bg, tx )

    # script parameter changes
    smooth( 3.0, bg.color, red )
    smooth( 3.0, tx.size, 60 )

    return end_animation()
sample_animation = sample_animation()
```

We'll start by giving a high-level overview of what happens in this piece of code, with details of the individual library functions appearing later.

The first line starts defining a new Python function. It is often useful, though not required, to write each animation script in a presentation as its own function. To do so helps structure the presentation code and frequently aids in debugging, so we will do so throughout this document. This function takes no arguments, though it could if we wanted to parameterize this animation script, making it return different animation objects depending on the parameters passed to it.

The first block within the function creates two elements: a `Fill` element, which fills the viewport with a solid color, and a `Text` element, which draws a string of text. The first argument to `Text()` positions it on the screen. The remaining arguments are default values for the element parameters. (`Fill` elements do not take a positioning argument, since they always fill the entire animation screen.)

The next line (`start_animation()`) starts defining a new animation. Its arguments are the elements that initially appear in the animation. Note that order is important: in this example, the `tx` element will be drawn on top of `bg`. Both the set of elements and their stacking order can be changed dynamically as the animation proceeds.

Between `start_animation()` and `end_animation()`, library calls are used to edit the parameter timelines to produce the goal animation. In this short example just two animated changes take place: first, the `color` parameter of the `bg` element is smoothly changed to red over a 3-second interval, then the text element's `size` parameter is smoothly increased to 60 over the following three seconds. This section is where the bulk of a typical animation script lies.

The call to `end_animation()` finishes defining the animation and returns the resulting animation *object*. We simply return this object to the caller, so that the return value from a call to `sample_animation()` is an animation object.

The final line of the example is another convention that we have found useful in creating presentations. Before this line is executed, `sample_animation` is an *animation script* — a Python function that when executed will return an animation object. We must call this function in order to obtain the animation object we give to the SLITHY player. Since the source animation script is typically not useful once the animation object is obtained, we can discard it. The last line calls the `sample_animation()` script, then throws the script away and binds the identifier "`sample_animation`" to the animation object returned.

A.6.1 Animation elements

This section describes the constructor functions that create elements for use in animations. Animations are similar to diagrams in that elements are placed on an infinite virtual canvas, and a camera rectangle (the *animation camera*) is used to map a portion of this canvas onto the animation's viewport. Frequently the animation's viewport will be the entire screen, but it may be some subregion if the animation object is used within a composite object.

Most of these element constructors require a *viewport* argument, which specifies where the

element is positioned on the animation's virtual canvas. This position will always be a rectangle on the canvas — a simple `Rect` object (see Section A.2) is the most common value for this argument. More complex methods of specifying this position that allow for animating the element's position are detailed in the next section.

Each type of element has a global default value for each of its parameters, which is given in the tables below. This default may be overridden for a particular instance of the element by giving it in the constructor. For example, both of these calls create valid `Fill` elements:

```
bg1 = Fill()
bg2 = Fill( color = yellow )
```

`bg1` uses all the global defaults for the fill element type. `bg2` overrides the default value of `color`, but uses the global defaults for the remaining parameters. All parameters may be modified as the animation proceeds by using the animation commands of Section A.6.2.

Now we'll describe in detail the element types available.

▶ **`Fill(`** [*parameter defaults*] **`)`**

parameter	type	default
`style`	string	`'solid'`
`color`	color	`black`
`color2`	color	`black`
`_alpha`	scalar	`1.0`

The `Fill` element creates a solid or gradient color fill, and is usually used as a background for other elements. This element has no position on the canvas; it always fills the entire animation viewport and so is unaffected by the animation camera.

The *style* parameter has three legal values: `'solid'` fills the viewport with solid color *color*, `'horz'` creates a horizontal gradient from *color* at the top of the viewport to *color2* at the bottom, and `'vert'`, which creates a gradient from *color* on the left to *color2* on the right. The *_alpha* parameter controls the transparency of the fill.

▶ **`Text(`** *viewport*, [*parameter defaults*] **`)`**

parameter	type	default
`text`	string or list	`''`
`color`	color	`black`
`font`	object	`None`
`size`	scalar	`1.0`
`justify`	scalar	`0.0`
`vjustify`	scalar	`0.0`
`_alpha`	scalar	`1.0`

This element creates a text box on the animation canvas. The text starts at the upper-left corner of the viewport rectangle, and is word-wrapped to the viewport's width. If there is more text than fits in the viewport it may extend out the top or bottom.

The *text* parameter may be a simple string, or a text list of strings, fonts, and colors, as described on page 148. *color* and *font* specify the initial text color and font, respectively. *size* gives the em-height of the text, in animation canvas units. *justify* specifies the horizontal justification of the text within the viewport: 0.0 for left justification, 0.5 for centered text, 1.0 for right-justified text. *vjustify* similarly specifies the vertical justification. The *_alpha* parameter scales the transparency of the entire text element.

▶ **`Image(`** *viewport*, [*parameter defaults*] **`)`**

parameter	type	default
`image`	image object	`None`
`fit`	object	`BOTH`
`anchor`	object	`'c'`
`_alpha`	scalar	`1.0`

The `Image` element places a static image on the canvas. The *image* parameter is used to specify the image, given as an image object (described in Section A.3.2). The *fit* parameter specifies how the image is scaled to fit the viewport. It can take one of four constants:

- `BOTH` ensures that the image fits within the viewport, scaling it uniformly to be as large as possible.

- `WIDTH` scales the image uniformly so that it fills the entire width of the viewport. Depending on the relative aspect ratios of the viewport and the image, the image may extend outside the viewport on the top and bottom sides.

- Similarly, `HEIGHT` fits the image's height to the viewport's height, allowing it to extend out the left and right sides if necessary.

- `STRETCH` stretches the image to fill the viewport exactly, even if this means the image is scaled nonuniformly.

While *fit* determines the image's size, *anchor* determines its placement in the viewport. The default value of `'c'` centers the image. The strings `'n'`, `'s'`, `'e'`, and `'w'` cause the image to be centered against the corresponding side of the viewport (north, south, etc.), while `'nw'`, `'ne'`, `'sw'`, and `'se'` push the image into a corner of the viewport.

Like most other elements, the *_alpha* parameter scales the transparency of the entire element.

▶ **`Drawable(`** *viewport*, [*drawing object* = `None`], [*parameter defaults*] **)**

parameter	type	default
`_alpha`	scalar	`1.0`
...parameters of drawing object ...		

The `Drawable` element is used as a container for other drawing objects—most frequently for parameterized diagrams. It can also be used to contain animation objects, but it is usually more convenient to use the specialized `Anim` element, described next, for this purpose. The drawable element itself has only one native parameter, *_alpha*, for controlling the overall transparency, but the element also takes on the parameters of whatever object it is being used to contain.

For instance, if we wanted to include the parameterized diagram `sample` from page 142 in an animation, the code might look like this:

```
d = Drawable( viewport, sample, name = 'bob', yesno = 0 )
```

The arguments *name* and *yesno* are parameters of the diagram `sample`. (The diagram has a third parameter, *number*, but in this example we've chosen not to override its default value.) Now

we can animate the diagram by using animation commands to manipulate the parameters `d.name`, `d.number`, and `d.yesno`. We can also manipulate `d._alpha` to fade the whole diagram in and out.

A parameterized diagram can be used multiple times within an animation by creating multiple `Drawable` elements that contain it.

▶ **`Anim(`** *viewport*, [*parameter defaults*] **`)`**

parameter	type	default
`anim`	object	`None`
`t`	scalar	`0.0`
`_alpha`	scalar	`1.0`

The `Anim` element is a kind of `Drawable` that has specialized methods for showing animation objects. Its three parameters are *anim*, the animation object to display, *t*, the time point to show, and *_alpha*, the standard transparency scaling factor. The contained animation can be played back by using the standard animation commands (Section A.6.2) to manipulate these three parameters, but the `Anim` element itself has a number of methods to simplify common tasks. These methods can be called in the part of the script that defines the animation timelines (i.e., inside a `start_animation()`/`end_animation()` pair).

▶ **`an.play(`** *animation object or list*, [**`fade`** `= 0`], [**`fade_duration`** `= 0.5`], [**`pause`** `= 1`], [**`duration`** `= None`] **`)`**

This plays back a single animation object or a list of animation objects inside the element. If the *fade* parameter is true, the playback is preceded by a fade-in of the first frame and followed by a fade-out of the last frame. The duration of this fade is given by the *fade_duration* parameter.

If the *pause* parameter is true, then a pause is inserted between the playback of successive animation objects, as well as after the fade-in and before the fade-out (if *fade* is also true). See page 170 for an explanation of pauses.

The final parameter, *duration*, can be used to manipulate the length of the contained animations. If it is `None` then each animation object is played back at its regular speed. Otherwise it should be a positive number; each animation object will be scaled in time to take this long to play.

▶ **`an.fade_in_start(`** *animation object or list*, *duration* **`)`**

▶ **an.fade_out_end(** *animation object or list*, *duration* **)**

These methods perform just the fading-in and -out parts of the `play` method above. The first of these fades in the first frame of the given animation object (the first animation object, if a list is given). The second method fades out the final frame of the object (the last object, if a list). The fade duration must be given for both methods.

▶ **an.show(** *animation object or list*, [*t* = 0.0] **)**

The `show` method causes the `Anim` element to show a still of the specified animation object at the specified point in time.

▶ **an.clear( )**

This method clears the `Anim` element so that it draws nothing.

Here is an example of using these methods with an `Anim` element to show an animation object `sample_anim` with a caption:

```
an = Anim( viewport )
cap = Text( viewport, text = 'Here is the caption',
                      _alpha = 0.0,
                      ... )

start_animation( an, cap )
an.fade_in_start( sample_anim, 0.5 )      # fade in the start frame
linear( 0.5, cap._alpha, 1.0 )            # then, fade in the caption
pause()
an.play( sample_anim )                    # play the animation
pause()

parallel()
an.fade_out_end( sample_anim, 0.5 )       # fade out the last frame ...
linear( 0.5, cap._alpha, 0.0 )            # ... and the caption together.
end()
```

Some of the commands in this example (`parallel()`, `linear()`, etc.) have not yet appeared in this appendix; they are documented in subsequent sections.

▶ **`BulletedList(`** *viewport*, [*parameter defaults*] **`)`**

parameter	type	default
`font`	font object	`None`
`color`	color	`black`
`size`	scalar	`1.0`
`sizescale`	scalar	`0.8`
`indent`	scalar	`1.0`
`leading`	scalar	`0.5`
`bullet`	string or list or `None`	`'-'`
`bulletsep`	scalar	`0.5`
`show`	scalar	`0.0`
`_alpha`	scalar	`1.0`

The `BulletedList` element provides a set of hierarchially-indented text strings with optional bullet markers, of the type commonly seen in presentations. A bulleted list contains a number of text *items*, each of which has an indent *level.* Note that all of the element parameters control the list layout; the content is added by calling the `add_item()` method described below.

As with the `Text` element, the *font* and *color* parameters set the initial font and color for the text items contained in the list.

The *size* argument sets the *base size* of the bulleted list element. This is the text size of items at level 0 (the outermost level). In general, the size of an item at level k is the base size times the value of the *sizescale* parameter, raised to the power k. With *sizescale* set at 0.8, the font size of a level-1 item will be 80% of the *size* parameter, a level-2 item will be $0.8^2 = 64\%$ of the *size* parameter, and so on. A level-k item will be indented $k \times indent \times size$ units, and will have $leading \times sizescale^k \times size$ vertical units of space after it.

If *bullet* is not `None`, a bullet will be drawn before each item. This argument may be a string or a text list, as described on page 148. The bullet will typically be just a single character or very short string—the purpose of allowing a text list is to allow the bullet character to come from a different font or be in a different color than the body text. The *bulletsep* parameter gives the spacing (in multiples of the base size) between the bullet and the body text.

The *show* parameter controls which items are visible. For example, if *show* were 3.4, then the

first three items would be fully visible, and the fourth would be drawn with $\alpha = 0.4$. Items can be made to fade in one-by-one by smoothly changing the *show* parameter. Everything in the element has its transparency scaled by the value of the element's *_alpha* parameter.

▶ **`bl.add_item(`** *level*, *string or list*, [*duration* = `0.0`], [*trans* = `linear`] **)**

Bulleted list elements are empty when they are first created; this method is used to actually add an item to the list. *level* must be a nonnegative integer, and the second argument is a string or a text list, just as in the `Text` element. It is legal to call this method either before the call to `start_animation()` or after it. If it is called in the middle of the animation (that is, after `start_animation()`), the optional *duration* and *trans* parameters can be used to specify a fade-in for the new item, which is done by just manipulating the element's *show* parameter. Usually the default linear transition is fine, but any transition function can be specified via the *trans* argument; see Section A.6.4 for details on transition functions.

▶ **`bl.remove_item(`** [*duration* = `0.0`], [*trans* = `linear`] **)**

`remove_item()` removes the last item from the bulleted list; the *duration* and *trans* arguments function just as those for the `add_item()` method.

▶ **`Interactive(`** *viewport*, [*parameter defaults*] **)**

parameter	type	default
`controller`	object	`None`
`_alpha`	scalar	`1.0`

The `Interactive` element places an interactive controller object in the animation. The argument for the *controller* parameter should be a controller *class*, which the runtime system will instance as necessary. See Section A.7 for details and examples of interactive controllers.

▶ **`Video(`** *viewport*, *filename*, [*parameter defaults*] **)**

parameter	type	default
`fit`	object	`BOTH`

The `Video` element is used to play digital video (MPG, AVI, etc.) from within a SLITHY presentation. It is subject to a number of limitations:

Table A.2 Keystroke commands used to control `Video` *elements.*

key	*effect*
‘ *(backtick)*	toggle play/pause (with **`control`** to play in loop mode
insert	play (with **`control`** to play in loop mode
delete	pause
left-arrow	step back one frame
right-arrow	step forward one frame
home	jump to beginning
backspace	jump to beginning and pause
up-arrow	increase playback speed
down-arrow	decrease playback speed
end	reset playback speed to standard

- Video is only supported on Windows, using the DirectShow library. `Video` elements will be ignored on other platforms.

- The viewport must be static and aligned with the axes of the screen. SLITHY will take the locations of the lower-left and upper-right corners of the viewport, projected into screen space, as the corners of the bounding rectangle for the video. The *fit* parameter governs the placement of the actual video image within this rectangle. The legal values are the same as those for the *fit* parameter of the `Image` element, listed on page 161.

- Video will always appear on top of everything drawn by SLITHY (it's actually drawn in a separate window atop the SLITHY window). The stacking order of overlapping video elements is undefined.

- Video can not be faded in or out.

- Video elements do not appear in the object tester, only when the animation object is shown from a presentation script (see Section A.8).

The keyboard is used to control the playback while a video element is on the screen. The key commands available are listed in Table A.2.

```
def sample_slide():
    c = get_camera()
    x = c.top(0.15).inset(0.05)
    y = c.bottom(0.85).left(0.6).inset(0.05)
    z = c.bottom(0.85).right(0.4).inset(0.05)

    title = Text( x, ... )
    bl = BulletedList( y, ... )
    im = Image( z, ... )

    ...
```

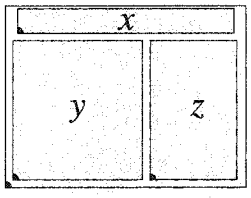


Figure A.11 *Using* `Rect` *object methods to subdivide the area of the slide. The upper right image shows the rectangles x, y, and z in relation to the camera rectangle. The bottom right image is a screenshot of a slide produced with this layout.*

Element viewports

The *viewport* argument to each element constructor (except for `Fill`, which does not take such an argument) is used to specify the element's position on the animation canvas. For elements that do not move, this argument may simply be a `Rect` object as described in Section A.2.

▶ **`get_camera( )`**

A common strategy for laying out slide elements is to first obtain a rectangle representing the whole screen, then use the methods available for `Rect` objects to subdivide it appropriately. The `get_camera()` function, when used while defining an animation (that is, between calls to `start_animation()` and `end_animation()`), returns the current camera rectangle (as a `Rect` object). When called while not defining an animation, it returns the default camera rectangle (the axis-aligned rectangle from $(0, 0)$ to $(400, 300)$).

Figure A.11 shows an example of using this approach to lay out a typical slide arrangement.

SLITHY also allows the viewport of an element to be animated itself. This is done by creating a special *viewport pseudoelement*. A viewport pseudoelement is something like a parameterized diagram in that it takes an arbitrary set of animatable parameters, but instead of producing a drawing, it produces a single `Rect` object that will be the viewport for some animation element.

Currently there is only one kind of viewport pseudoelement implemented.

▶ **viewport.interp(** *rect0*, [*rect1*], [...] **)**

This constructor takes one or more `Rect` objects and returns a viewport object that interpolates between the rectangles. The viewport object has a single parameter called *x*. An example will help to make this clear:

```
tx_v = viewport.interp( vp0, vp1 )
tx = Text( tx_v, text = 'hello, world!', ... )

start_animation( tx )
smooth( 2.0, tx.color, white )
smooth( 2.0, tx_v.x, 1.0 )
end_animation()
```

`tx_v` is a viewport object. It interpolates between rectangles `vp0` and `vp1` (which we assume are `Rect` objects defined elsewhere). `tx_v` is then passed to the `Text` element constructor as its *viewport* argument. Once we begin defining an animation that contains the `tx` element, we can animate the *x* parameter of the viewport object just like the parameters of the text element. Changing the viewport object's *x* parameter will cause the element to move around on the canvas.

A.6.2 Animation commands

Next we will describe the library functions for creating animation objects from a collection of elements.

▶ **start_animation(** [*element*$_1$], [*element*$_2$], [...],
[**camera** = `Rect(0,0,400,300)`] **)**

This function begins defining an animation. The arguments are the initial set of elements to appear in the animation. The order of the arguments is significant—elements later in the list will be drawn on top of earlier elements. Both the stacking order and the set of elements appearing can be modified dynamically as the animation proceeds.

The optional named argument *camera* can be used to specify a starting camera rectangle other than the default. The `start_animation()` function returns a *camera element* that can be used to change the camera rectangle during the animation; this is described in Section A.6.3.

▶ **end_animation( )**

This function finishes up defining an animation and returns a list of the animation object or objects created. (A single `start_animation()/end_animation()` pair may create multiple animation objects through use of the `pause()` function, described next. The return value is always a list, even if just a single animation object is created.

The remaining commands in this section may only appear while defining an animation, that is, between calls to `start_animation()` and `end_animation()`.

▶ `pause( )`

The `pause()` function is used to split an animation sequence into multiple sub-objects, each one continuing where the previous one left off. The most common use of this is to break up a long animated sequence with pauses where SLITHY waits for the presenter to press a key to continue.

If there are k calls to `pause()`, then `end_animation()` will return a list of $k + 1$ animation objects. For instance, this code would produce three animation objects:

```
start_animation()
...
pause()              # first animation object ends here
...
pause()              # second animation object ends here
...
end_animation()      # third animation object ends here
```

A collection of elements in animated in SLITHY by changing their parameter values over time. An animation object has one timeline for every parameter of every element used in the animation. The call to `start_animation()` creates these timelines and initializes each to a constant default value (the value specified in the element constructor, if any, otherwise a built-in default for that parameter).

Each animation command makes an edit to one or more of these timelines, overwriting portions of them with new values. At the end of the script, when `end_animation()` is called, SLITHY bundles up all the parameter timelines into an animation object, which can then be displayed at an arbitrary point in time.

While an animation script is in progress, SLITHY maintains a value called the *time cursor* that marks the point at which edits will take place. Some edits (such as `set()`) have zero duration, and

do not change the position of the time cursor. Others (such as `linear()`) have a duration, and generally cause the time cursor to advance to the end of the edit.

All edit durations are nominally expressed in seconds, though a completed animation object can be sped up or slowed down arbitrarily. Parameters are specified with dot notation, with "`e.p`" denoting the parameter *p* of element *e*.

- **`linear(`** *duration*`,` *parameter*`,` *tovalue*`,` `[` **`relative`** `= 0 ]` **`)`**
- **`linear(`** *duration*`,` *parameter*`,` *fromvalue*`,` *tovalue*`,` `[` **`relative`** `= 0 ]` **`)`**
- **`smooth(`** *duration*`,` *parameter*`,` *tovalue*`,` `[` **`relative`** `= 0 ]` **`)`**
- **`smooth(`** *duration*`,` *parameter*`,` *fromvalue*`,` *tovalue*`,` `[` **`relative`** `= 0 ]` **`)`**

The `linear()` and `smooth()` functions change a parameter to a new value through continuous interpolation over a finite interval. `linear()` uses linear interpolation, while `smooth()` uses a slow-in-slow-out interpolation. Figure A.12 illustrates the effect of these commands (and others) on a sample parameter timeline.

If *fromvalue* is omitted, then the interpolation begins at the parameter's current value at the position of the time cursor. If *fromvalue* is present, the parameter jumps instantaneously to *fromvalue* at the start of the interpolation. Passing a true value for the optional argument *relative* causes *tovalue* (and *fromvalue*, if present) to be treated as relative to the parameter's current value at the start of the transition.

These functions may only be used for parameters that have interpolatable values—real numbers, integers, and colors. Attempts to use `linear()` or `smooth()` to change parameters of other types (such as strings) will raise an exception.

- **`set(`** *parameter*`,` *value* **`)`**

This function instantaneously changes the given parameter's value to the new given value, at the position of the time cursor. It can be applied to parameters of any type (though no type-checking is done on the value). This edit is of zero duration.

- **`wait(`** *duration* **`)`**

This function produces an edit of the specified duration, but does not change any parameter timelines.

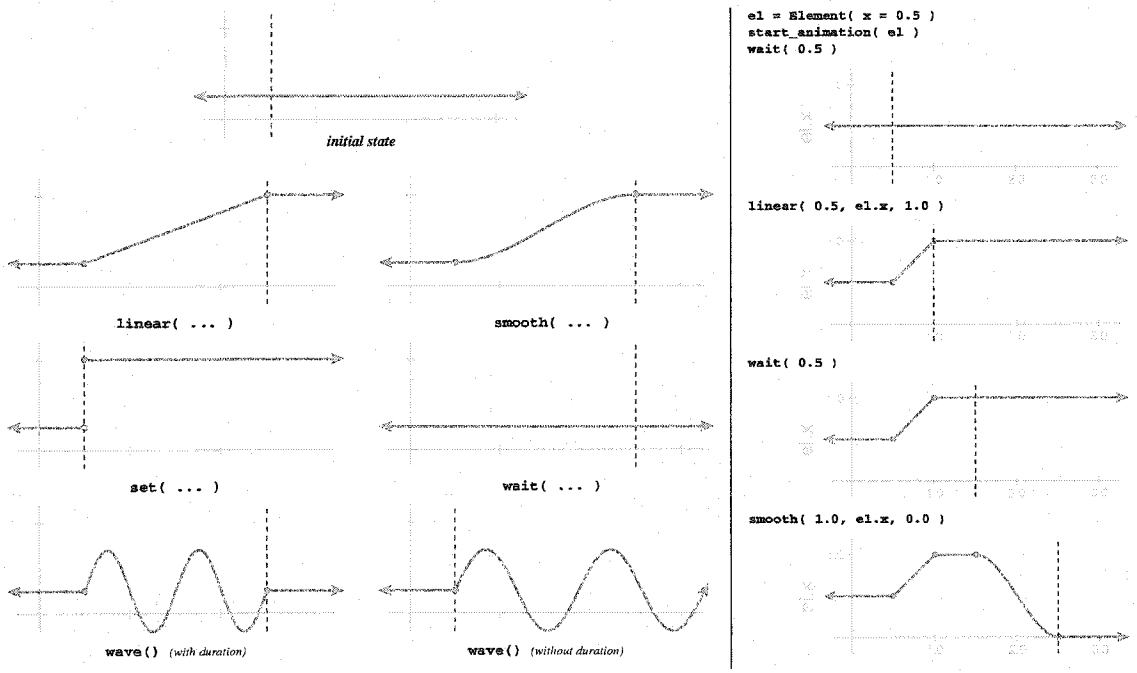


Figure A.12 *Illustration of animation commands applying edits to a parameter timeline. The red dotted line indicates the position of the time cursor. The left side shows the results of six different indiviual commands, each applied to the initial state shown in the top row. The right side shows how multiple commands build up a complex timeline, by showing the state of the parameter timeline after each command in a script.*

▶ **wave(** *parameter*, [**min** = *min*], [**max** = *max*], [**mean** = *mean*],
[**amplitude** = *amplitude*], [**duration** = *duration*],
[**period** = *period*], [**cycles** = *cycles*] **)**

The `wave()` function sinusoidally varies a parameter value (which must be of an interpolatable type). Only certain combinations of the optional arguments listed are allowed. The first restriction is that exactly two of *min*, *max*, *mean*, and *amplitude* must be given—this is sufficient to specify the range of the variation.

The last three arguments are used to specify the waving motion's duration, period (in cycles per second), and/or number of complete cycles respectively. If exactly two of these are given, then the motion has a finite duration. Alternatively, if only the *period* argument is given, the edit is considered to have zero duration. The waving motion is written into the timeline starting at the position of the time cursor and continuing indefinitely. Both applications of `wave()` are illustrated in Figure A.12.

Currently there is no control over the phase of the motion; it always starts at the beginning of a whole cycle.

▶ **parallel( )**
▶ **serial(** [*delay* = `0.0`] **)**
▶ **end( )**

These functions are used to group together the basic editing commands (such as `linear()`, `set()`, etc.) into composite edits. All the edits contained in a `serial()...end()` block are concatenated in sequence, so they happen one after the other. The duration of the composite edit is the sum of the individual durations.

A `parallel()...end()` block is used to overlap edits. All the edits contained in one of these blocks begin simultaneously; the duration of the whole block is the maximum of the components' durations.

These two types of blocks can be nested to produce complex overlap patterns. A few examples are shown in Figure A.13. The optional argument to `serial()` is used to shorten the common idiom of `serial()` immediately followed by `wait()`; the following two blocks are equivalent:

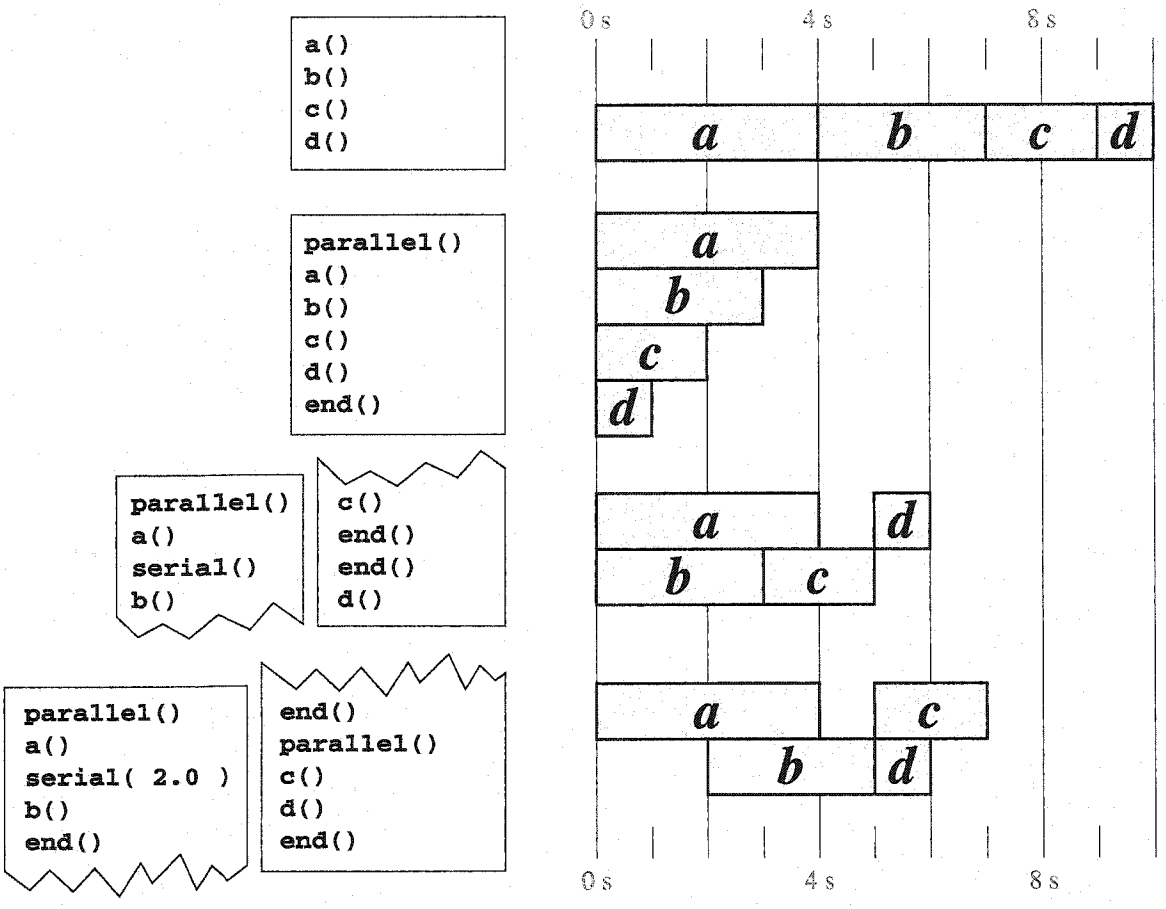


Figure A.13 *Using* `parallel()` *and* `serial()` *to overlap animation functions.* ***a***, ***b***, ***c***, *and* ***d*** *represent animation edits with durations of 4, 3, 2, and 1 seconds, respectively.*

```
serial()                          serial( 5.0 )
wait( 5.0 )
```

Each animation script begins in "serial mode," as if the whole script were contained in a `serial()...end()` block.

▶ **`enter(`** [*element*$_1$]**,** [*element*$_2$]**,** [...] **)**
▶ **`exit(`** [*element*$_1$]**,** [*element*$_2$]**,** [...] **)**

These two functions add elements to and remove elements from the animation, at the position of the time cursor. This is a zero-duration edit. When elements are added, they are placed at the top of the stacking order, in the order they are listed in the call to `enter()`.

If an element is going to be invisible for a significant length of time (by being positioned off-camera or having its alpha set to zero), then it is more efficient to remove it from the animation, with `exit()`, and bring it back with `enter()` when it is needed again.

▶ **`lift(`** [*element*$_1$], [*element*$_2$], [...], [**`above`** = `None`])
▶ **`lower(`** [*element*$_1$], [*element*$_2$], [...], [**`below`** = `None`])

`lift()` raises elements in the stacking order. If an *above* argument is given then it should be another element in the animation; the moving elements are restacked so that they end up just above the *above* element.[1] If *above* is `None`, then the elements are moved to the top of the stacking order. In all cases, the relative stacking order of the moved elements is preserved.

`lower()` functions analagously, moving a set of elements down in the stacking order, either to the bottom of the stack or to just below a given element. Both functions are zero-duration edits.

There are a few more functions available in the library that are purely informational—they do not affect the animation being defined but may be useful in writing scripts.

▶ **`get(`** *parameter*, [t = `None`])

`get()` samples a parameter's timeline and returns the value at a point in time. The time may be specified explicitly via the second argument. A time of `None`, the default if no t argument is supplied, samples at the position of the time cursor.

▶ **`current_time( )`**

This function returns the current position of the time cursor (a real-valued number).

▶ **`defining_animation( )`**

This function returns a true value if and only if it is called while an animation is being defined (that is, between calls to `start_animation()` and `end_animation()`).

A.6.3 The animation camera

Similar to parameterized diagrams, an animation is comprised of elements laid out on a virtual canvas. A camera rectangle is used to determine which part of this canvas appears in the animation's

---

[1]Note that this usage can actually result in some elements being moved down. If the initial stacking order (from top to bottom) is $\{a, b, c, d\}$, then `lift(a,above=d)` results in the order $\{b, c, a, d\}$—that is, element a is moved *down* so that it is *just above* element d.

viewport. By default this is the rectangle from (0, 0) to (400, 300), though this can be changed via the optional *camera* parameter to `start_animation()`.

`start_animation()` also returns a *camera object* that can be used to animate the camera rectangle itself during the animation. This camera object functions very much like an ordinary element with a single parameter called `rect`. This parameter's timeline can be manipulated just like any other element:

```
cam = start_animation( ... )
...
smooth( 3.0, cam.rect, Rect(...) )     # smoothly move camera
...
set( cam.rect, Rect(...) )             # instantly move camera
...
end_animation()
```

A.6.4 Transition and undulation objects

While `linear()` and `smooth()` behave much like functions in animation scripts, they are actually objects of the class `Transition`. Transition objects are used to represent the style of interpolation between two values. `linear()` and `smooth()` are defined in the SLITHY library like this:

```
linear = Transition( style = 'linear' )
smooth = Transition( style = 'smooth' )
```

These are the only two styles currently implemented, but the smooth style in particular has optional arguments `s` and `e` that control the shape of the interpolation. `s` and `e` raise or lower the tangent of the curve at the start and end of the interpolation, respectively. These arguments can be used in creating a new transition object:

```
other = Transition( style = 'smooth', s = -0.5, e = 0.5 )
```

Once `other` is created, it can be used within animation scripts anywhere that `linear()` and `smooth()` can. When called as a function, it takes the same arguments as `linear()` and

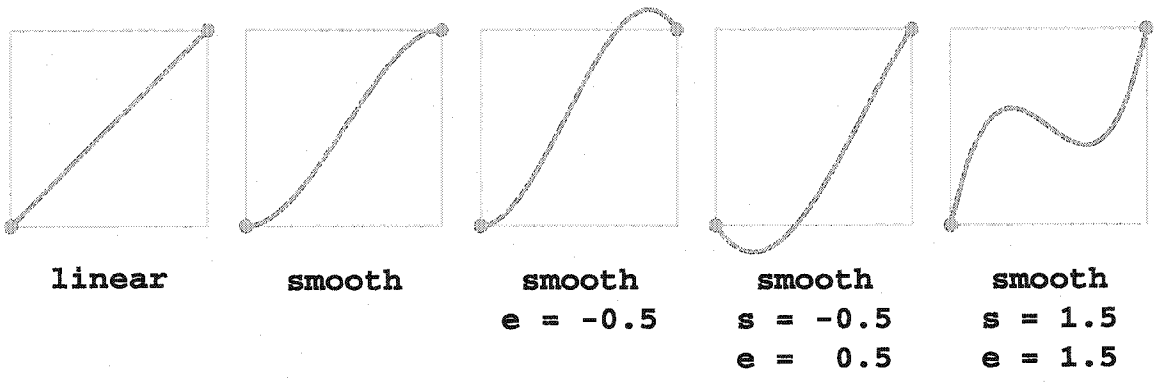


Figure A.14 *Various transition styles available in* SLITHY.

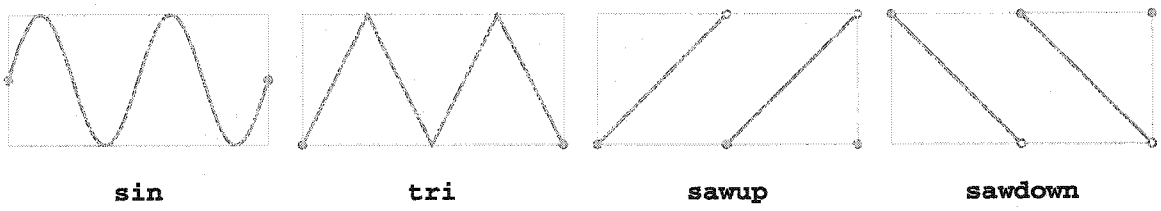


Figure A.15 *Various undulation styles available in* SLITHY. *Two complete cycles of each style are shown.*

`smooth()` (see page 171). Figure A.14 shows examples of various transition styles.[2]

Similarly, `wave()` is not a true function but an example of an *undulation* object. Undulation objects can be created much like transition objects:

```
wave = Undulation( style = 'sin' )          # predefined in Slithy library
sawtooth = Undulation( style = 'sawup' )
```

After executing the second line, the object `sawtooth` can be used just like the predefined object `wave` can (see page 173 for details of its arguments). Where `wave` produces a sinusoidally-varying pattern, though, `sawtooth` will produce a sawtooth pattern. Figure A.15 illustrates the four undulation styles implemented in SLITHY.

[2]The transition mechanism is designed to be extensible; look in `transition.py` for how styles are implemented. To add a new style, create a new subclass of `TransitionStyle`, then add it to the `Transition.styles` dictionary.

A.7 *Interactive objects*

Interactive objects are implemented by deriving a class from the SLITHY class `Controller`. Interactive objects work much like animation scripts, in that they describe a set of animation elements and time-varying values for each element's parameters. The difference is that an animation script is executed as a single unit, while the code for an interactive object is broken up into different methods that are executed at different times by the SLITHY runtime system.

This section describes the methods that interactive object authors may provide in order to implement their object's behavior.

▶ **`create_objects(`** *self* **`)`**

Most interactive objects will have a `create_objects()` method. This method is called whenever a new instance of the class is created in order to create the initial set of elements. All the element types of Section A.6.1 are available. Just as animation scripts typically store element references in local variables, interactive objects must keep references in instance variables so that they can be used from within other methods.

This method should return a tuple of elements that should initially appear on the canvas. (This is the equivalent to passing the elements to `start_animation()` in animation scripts.) If only a single element is to appear, the element itself may be returned rather than a singleton tuple.

Here is the initial section of a simple interactive object, which uses a single diagram element that draws a clock:

```
class SampleInteractive(Controller):
    def create_objects( self ):
        self.d = Drawable( get_camera(), clock )
        return self.d
```

An analogous animation script would look like this:

```
def sample_animation():
    d = Drawable( get_camera(), clock )
    start_animation( d )
```

Just as in animation scripts, the set of elements on the canvas may be changed during the anima-

tion, via the `enter()` and `exit()` functions.

▶ **`start(`** *self* **`)`**

All the remaining methods in this section, including `start()`, are used to edit the animation timelines in response to certain user events. All of the commands available in animation scripts work within these methods as well: changing parameter values, `wait()`, parallel and serial modes, as well as adding, removing, and restacking elements.

If it is defined, `start()` is called once when the object is first displayed; a common use is to make the object fade in rather than appearing abruptly. Adding this functionality to the previous example would look like this:

```
class SampleInteractive(Controller):

    . . .

    def start( self ):
        set( self.d._alpha, 0 )
        fade_in( 0.5, self.d )
```

▶ **`key(`** *self*, *key*, *mouse*$_x$, *mouse*$_y$, *mod* **`)`**

This method, if defined, is called when the user presses a letter or number key. (These are the only keys that are passed to interactive objects; all others are reserved for SLITHY's use.) Every interactive object present on the screen receives every input event, so a single keystroke will invoke the `key()` methods of all visible interactive objects.

- The *key* argument will be the key that was pressed. This will always be a digit or lowercase letter.

- *mouse*$_x$ and *mouse*$_y$ are the coordinates of the mouse pointer, in screen coordinates, at the time of the keypress. The next section describes ways to make use of these values.

- *mod* is a tuple that describes which modifier keys were pressed along with the key. It will contain the strings "`shift`" and/or "`control`" to indicate the presence of those modifiers.

In this example, the `key()` method produces one animated change when the 'B' key is pressed, and a different change when Control-A is pressed:

```
def key( self, key, x, y, mod ):
    if key == 'b':
        linear( 1.0, self.d.hours, 5 )
    elif key == 'a' and 'control' in mod:
        smooth( 0.5, self.d.minutes, 0 )
```

All other keystrokes are ignored.

▶ **mousedown(** *self*, $mouse_x$, $mouse_y$, *mod* **)**
▶ **mousemove(** *self*, $mouse_x$, $mouse_y$, *mod* **)**
▶ **mouseup(** *self*, $mouse_x$, $mouse_y$, *mod* **)**

These methods are used to report mouse events to the interactive object. Mouse events are reported to all interactive objects on the screen, regardless of the location of the mouse pointer. The screen coordinates of the mouse pointer and the list of active modifier keys is reported, just as with the `key()` method. Only the primary mouse button (the left button, in most cases) can be used to manipulate interactive objects; other mouse buttons are reserved for SLITHY's use. The `mousemove()` method is only called when the mouse button is down. There is no way for interactive objects to track the mouse when the button is not pressed.

A.7.1 Using cursor coordinates

There are two main ways of making use of the mouse cursor coordinates within event handler methods. The first is to use SLITHY's object buffer to record object ID's while a diagram is being drawn, then query those coordinates in the event handler. This query is done using the `query_id()` function.

▶ **query_id(** x_1, y_1, [...] **)**

This function takes one or more pairs of screen coordinates and looks up those locations in the object ID buffer. The resulting IDs are returned as a tuple of integers.

Here are a sample diagram and interactive controller that illustrate this idea:

```
def diagram():
    . . .
    id( 1 )
    color( green )
    rectangle( 0, 0, 10, 10 )
    . . .
    id( 2 )
    color( red )
    dot( 2, 0, 0 )
    . . .

class Interactive(Controller):
    def create_objects( self ):
        self.d = Drawable( get_camera(), diagram )
        return self.d

    def mousedown( self, x, y, m ):
        id, = query_id( x, y )
        if id == 1:
            # clicked on the green rectangle
            . . .
        elif id == 2:
            # clicked on the red dot
            . . .
```

Note how the `mousedown()` method uses `query_id()` to determine what object is underneath the location of the mouse. Of course, the parameterized diagram and the interactive object driving it must be designed together so that they agree on the meanings of the different ID values.

The second technique for using mouse coordinates is to use the `mark()` and `unproject()` functions to transform the coordinates from screen space to the canvas space of the diagram, where it is usually easier to use them in calculations. The next example illustrates setting the clock using the mouse:

```
def clock( hour = (INTEGER, 0, 23, 0),
           minute = (INTEGER, 0, 60, 0),
           label = (STRING, ''),
           controller = (OBJECT, None),
           ):

    . . .  # draw the clock here

    if controller:
        controller.coords = mark()
```

```
class InteractiveClock(Controller):
    def create_objects( self ):
        self.d = Drawable( get_camera(), clock, controller = self )
        return self.d

    def mousedown( self, x, y, m ):
        px, py = unproject( x, y, self.coords )

        theta = atan2( py, px ) * 180 / pi
        minutes = (90-theta) / 6
        if minutes < 0: minutes += 60

        smooth( 0.5, self.d.minutes, minutes )
```

Only one change is required to the clock diagram itself: it now has a parameter called `controller`. If an object other than `None` is passed in to this argument, the diagram function will save its coordinate system (obtained using the `mark()` function from page 156) in the object.

The interactive object, when it creates a `Drawable` with an instance of the clock, sets the value of this `controller` parameter to be the *interactive object itself.* Thus, whenever this instance of the clock is drawn, the coordinate system will be stored in the interactive object.

The `mousedown()` method uses `unproject()` to transform the mouse coordinates into the stored diagram coordinate system. This example computes the nearest value for the `minutes` parameter; the net result is that the time on the clock changes so that the minute hand points toward the location of the mouse click. This calculation is relatively simple due to the fact that the center of the clock is at the origin in the diagram coordinate system. Because the calculation is being done in diagram coordinates, it will work no matter where the clock is placed on the slide or how it is oriented.

A.8 Presentation scripts

The commands covered so far allow the creation of parameterized diagrams, animation objects, and interactive objects. All of these different types of objects can be defined within a single Python file if desired. To define a presentation that makes use of these objects, a separate file called a *presentation script* must be created. This script defines the order in which the top-level animation objects are to

be shown and adds some navigation features for use while presenting.

We'll begin with an example. Assume that the files `intro.py` and `related.py` have been created, and each contains animation objects `slide1`, `slide2`, and `slide3`. Here is a simple script that assembles these into a presentation:

```
from slithy.presentation import *

import intro
import related

bookmark( 'Introduction' )
play( intro.slide1 )
play( intro.slide2 )
play( intro.slide3 )
bookmark( 'Related Work' )
play( related.slide1 )
play( related.slide2 )
play( related.slide3 )

run_presentation()
```

The first line initializes this script as a SLITHY presentation script. Note that we do *not* import "`slithy.library`"; that is for scripts defining diagrams, animations, and interactive objects. Next we import the files containing the animation objects we want to show, using the Python import mechanism as described in Section A.1. Once we've imported all the resources needed for the presentation, we can begin laying out the presentation itself.

▶ **`bookmark(`** *name* **`)`**

The `bookmark()` function gives a name to the current point in the presentation. While presenting, right-clicking the SLITHY window will bring up a menu of bookmarks that allow skipping to those points in the talk. In this example we set up bookmarks that allow us to jump to the beginning of each section.

▶ **`play(`** [*anim*$_1$], [*anim*$_2$], [...], [**`pause_between`** = 1],
[**`pause_after`** = 0] **`)`**

This function plays a sequence of one or more animation objects. Each of the *anim* arguments may be a single animation object or a list of animation objects. The *anim* arguments are flattened and concatenated to form the list of objects to play. The optional arguments control whether SLITHY

Table A.3 Keystroke commands used to control SLITHY *during presentations.*

key	*effect*
space	end pause (continue presentation)
<, comma	skip backwards (to previous pause point)
>, period	skip forwards (to next pause point)
ctrl-pgup	jump to presentation start
ctrl-pgdn	jump to presentation end
tab	toggle fullscreen mode
=	save screenshot of SLITHY window
escape	quit SLITHY

inserts a pause between each pair of objects or not, and whether a pause is inserted after the final animation is played. At a pause, the presentation runtime waits for the user to press the spacebar to continue.

▶ **`pause( )`**

This function manually inserts a pause (waiting for the user) at the current point in the presentation.

▶ **`run_presentation( )`**

A call to this function should always be the final line of the script. The previous functions all build up an internal description of the presentation. Inside `run_presentation()` is where this description is used to actually display the presentation.

During the presentation, SLITHY is controlled primarily from the keyboard. Table A.3 summarizes the keystroke commands available during presentations.

Appendix B

A COMPLETE EXAMPLE

In this appendix we will show a complete example of a nontrivial interactive object and animation in SLITHY. We will construct an interactive applet illustrating the de Casteljau algorithm for Bézier curve construction (see Figure 5.13 on page 105). There are two components to this applet: a parameterized diagram to draw the figure, and an interactive controller to allow manipulation of the parameters. The resulting applet starts as a blank canvas, and allows the following interactions:

- Clicking on the blank canvas adds a control point.
- Clicking on the control polygon sets the subdivision fraction u.
- Control points can be dragged with the mouse to arbitrary places on the canvas.
- The outermost construction points can be dragged along the control polygon to change the subdivision fraction.
- Keystrokes can be used to advance through the steps of the de Casteljau construction, turn display of the resulting curve on or off, and reset the diagram.

B.1 Preliminaries

The file begins with some standard SLITHY headers: importing the SLITHY library and defining some color objects that will be used in the drawing. The parameterized diagram will also need a font for printing the current value of u as part of the figure. In this project, we have defined a file called `resources.py` (not shown) that loads all the fonts needed throughout the presentation.

Here, we import that file and select the font object we need. This structure simplifies using fonts consistently throughout a presentation.

```
from slithy.library import *

darkblue = Color( 0, 0, 0.6 )
lightblue = Color( 0.3, 0.6, 0.8 )
darkred = Color( 0.8, 0, 0 )
bgcolor = Color( 0.407, 0.580, 0.709 )

from resources import fonts
thefont = fonts['mono']
```

Next we will define some helper functions useful in illustrating the de Casteljau algorithm:

```
def reduce( p, u ):
    if len(p) < 2:
        return p
    return [(x1*(1-u)+x2*u, y1*(1-u)+y2*u)
            for (x1,y1),(x2,y2) in zip(p[:-1], p[1:])]
```

The `reduce()` function takes a list of n points $(p_1, p_2, \ldots, p_n)$ and a fraction u, and returns a list of $n-1$ interpolated points

$$((1-u)p_1 + up_2, (1-u)p_2 + up_3, \ldots, (1-u)p_{n-1} + up_n).$$

Each point is represented as a 2-tuple (x, y). This computation corresponds to one step of the de Casteljau algorithm.

```
def map_to_line( x, y, (px,py), (qx,qy) ):
    u = ((x-px) * (qx-px) + (y-py) * (qy-py)) /
        ((qx-px) * (qx-px) + (qy-py) * (qy-py))
    if u < 0:
        return 0
    if u > 1:
        return 1
    return u
```

The `map_to_line()` function projects the point (x, y) onto the closest point lying on the line segment $\overline{PQ}$. The projected point is returned as a value $u \in [0, 1]$ representing a fraction of the

distance from P to Q. We will make use of this function when the user clicks on the control polygon to set the subdivision fraction.

The parameterized diagram that draws this figure is going to expect a parameter called `info` that contains the coordinates of the control points. Normally this object will be the interactive controller object, but for testing we will create a "fake" object that has a fixed set of control points. Using this object will allow us to test the parameterized diagram independently of the interactive controller.

To create this test `info` object, we use the common Python idiom of a class with no methods. We create one instance of this class – the rough Python equivalent of a C struct – and place a set of control points for testing. The `curve_dirty` flag signals when the set of control points has changed, so that the curve can be recomputed only when necessary.

```
class Blank:
    pass

test_info = Blank()
test_info.controls = [ (10,10), (10,90), (90,90), (60,40) ]
test_info.curve_dirty = 1
```

The next helper function will actually compute the Bézier curve from the control points. We will use the naïve algorithm of simply sampling the curve at equally-spaced values of u and applying the de Casteljau algorithm, using the `reduce()` function defined above. This method of computing the curve is relatively slow, but it is simple to implement and sufficient for the purposes of this applet.

This function has a single argument, which should be an object possessing `controls` and `curve_dirty` attributes, such as the `test_info` object defined above. The computed curve is stored in the `curve` attribute of the argument object.

```
def compute_curve( info ):
    if len(info.controls) == 0:
        info.curve_dirty = 0
        return None

    # first point is the first control point
    p = [ (0,)+info.controls[0] ]
```

```
    # compute intermediate points
    for i in range(1,100):
        u = i / 100.0
        # start with the control points, reduce until
        # only one point is left
        q = info.controls
        while len(q) > 1:
            q = reduce( q, u )
        p.append( (u,)+q[0] )

    # last point is the last control point
    p.append( (1,)+info.controls[-1] )

    # store the curve in the info object
    info.curve = p
    info.curve_dirty = 0
```

B.2 The parameterized diagram

Now we are ready to begin defining the parameterized diagram itself.

```
def bezier( info = (OBJECT, test_info),
            u = (SCALAR, 0, 1),
            show_const = (BOOLEAN, 0),
            show_curve = (BOOLEAN, 0),
            const_reveal = (SCALAR, 0, 10),
            ):
```

The diagram has five parameters: `info`, as described above, is an object that contains the control points; `u` sets the subdivision fraction to be displayed; `show_const` controls whether or not the construction lines and points are displayed; and `show_curve` controls whether or not the final curve itself is displayed. Figure B.1 shows the appearance of the diagram as the different parts of the figure are drawn.

The fifth parameter, `const_reveal`, controls how much of the construction is shown. For a curve with n control points, we illustrate the de Casteljau algorithm in $2n - 3$ steps: $n - 1$ sets of construction points are drawn, alternating with $n - 2$ sets of construction lines. The final set of construction points is always just a single point, which lies on the output curve. As `const_reveal` increases from zero to $2n - 3$, more and more of the steps will be drawn. This process is

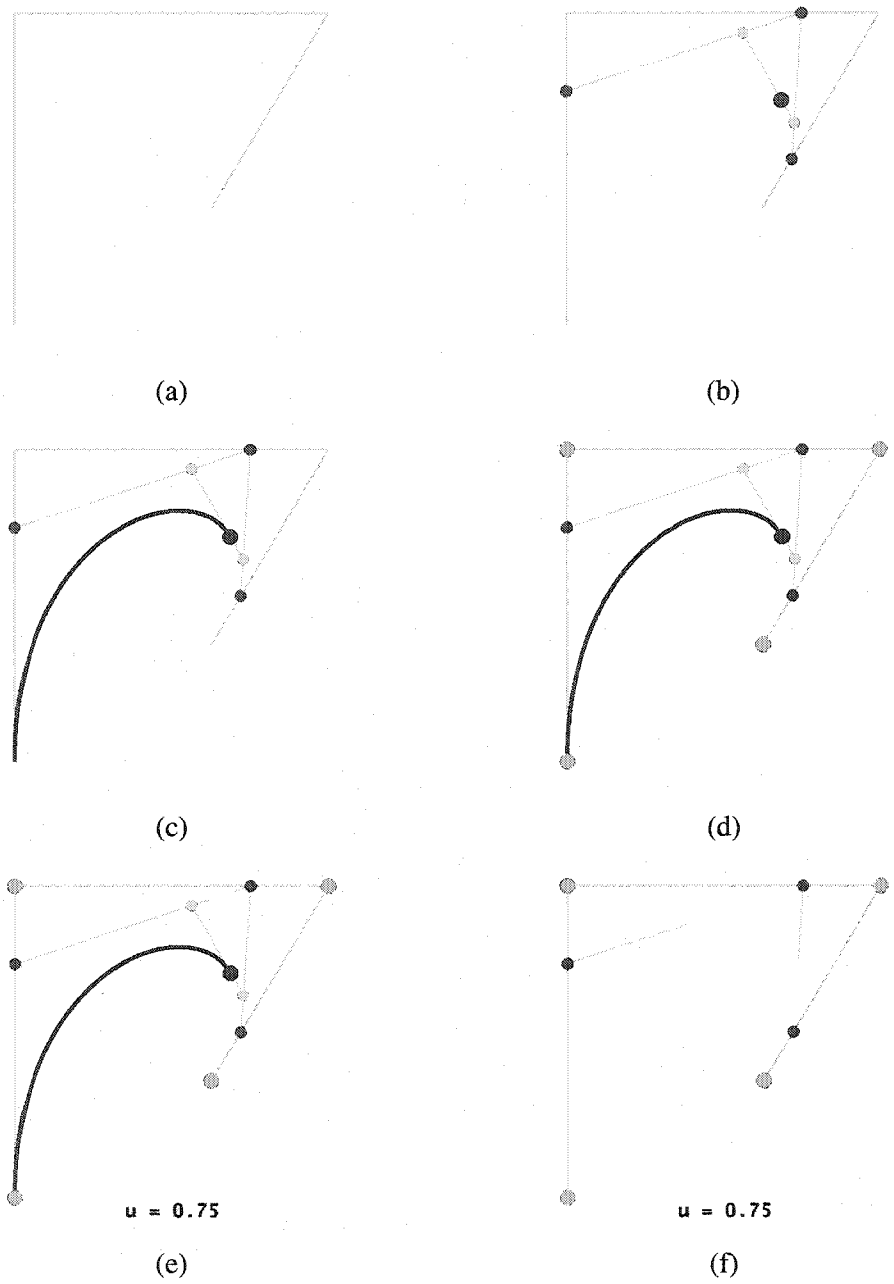


Figure B.1 *Different parts of a diagram for illustrating construction of a Bézier curve. The control polygon (part (a)) is drawn first. Part (b) adds the construction lines and points, and part (c) adds a portion of final curve. The control points are drawn in part (d), and part (e) shows the final figure, with the subdivision fraction u displayed underneath. Part (f) illustrates the effect of the* `const_reveal` *parameter, which causes only some steps of the construction to be drawn. Here only the first set of construction points and half of the first set of lines are shown.*

illustrated for a curve with four control points in Figure B.2.

```
set_camera( Rect(0,0,100,100) )
clear( white )

if show_curve and info.curve_dirty:
    compute_curve( info )
```

The function begins by setting the camera and clearing the canvas. Then, if the curve needs to be displayed and the control points have changed since the last time the curve was recomputed, the curve is recomputed.

```
if len(info.controls) > 1:
    i = 1000
    push()
    for (x1,y1),(x2,y2) in zip(info.controls[:-1],
                               info.controls[1:]):
        # draw a wide, invisible band for clicking on
        id( i )
        thickness( 3 )
        color( invisible )
        line( x1, y1, x2, y2 )

        # then draw the visible line, which is narrower
        thickness( 0.5 )
        color( 0.7 )
        line( x1, y1, x2, y2 )
        i += 1
    pop()
```

The first thing to be drawn is the control polygon – the set of lines connecting successive control points – which is always visible. Each line is assigned a unique ID number. The line from p_i to p_{i+1} is given id $(i + 1000)$. These IDs will be used in the interactive controller to determine which edge of the control polygon was clicked. Because the visible gray line is fairly narrow, clicking on it exactly is difficult. To make it easier to click on the polygon, a wider, invisible line is drawn along each visible line using the same ID number. The user must only hit somewhere in this wider region in order to click "on the line."

The next block of code is the most complex part of the diagram: drawing the construction lines. Here is the top of the loop:

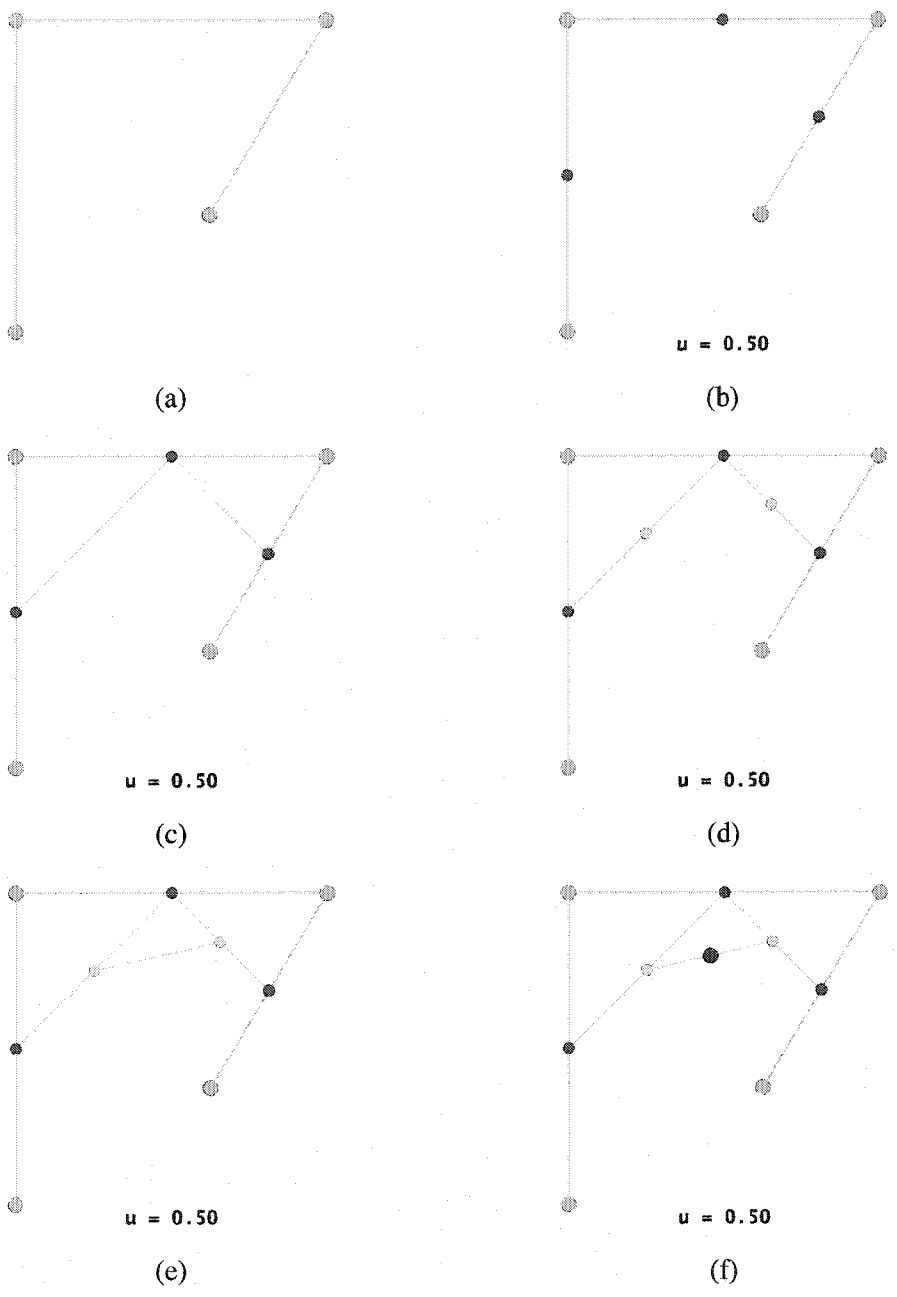


Figure B.2 *For curves with n control points, the applet illustrates the de Casteljau algorithm in* $2n - 3$ *steps. Here the case* $n = 4$ *is shown. Part (a) shows the initial state of the diagram before the subdivision fraction u is specified. Each successive image adds one set of construction points or construction lines in an animated fashion. Each step corresponds to increasing the* `const_reveal` *parameter by one.*

```
if show_const:
    p = info.controls
    levels = len(p)-2
    i = 0
    while len(p) > 1:
        if levels > 1:
            thecolor = darkblue.interp( lightblue,
                                        float(i)/(levels-1) )
        else:
            thecolor = darkblue

        if const_reveal < i*2:
            thecolor = invisible
        elif const_reveal < i*2+1:
            thecolor = Color( thecolor, const_reveal - i*2 )

        if const_reveal < i*2+1:
            linefrac = 0
        elif const_reveal < i*2+2:
            linefrac = const_reveal - (i*2+1)
        else:
            linefrac = 1
```

Each pass through the while loop will draw one set of construction points and one set of construction lines. At the top of the loop we determine the color of the construction points. The first set is drawn in dark blue, and following sets get progressively lighter in color. We also use the `const_reveal` parameter to determine what fraction of the lines should be drawn and how opaque are the points (so that each step of the construction is drawn in gradually as `const_reveal` is increased.

```
        p = reduce( p, u )
        if len(p) > 1:
            # draw the lines
            color( 0.8 )
            thickness( 0.5 )
            for (x1,y1),(x2,y2) in zip( p[:-1], p[1:] ):
                x2 = x2 * linefrac + x1 * (1-linefrac)
                y2 = y2 * linefrac + y1 * (1-linefrac)
                line( x1, y1, x2, y2 )

            # draw the points
            color( thecolor )
            if i == 0:
                # outermost set of points
                j = 3000
```

```
                for x, y in p:
                    id( j )
                    dot( 1.5, x, y )
                    j += 1
                id( -1 )
            else:
                # all other sets of points
                for x, y in p:
                    dot( 1.5, x, y )
        i += 1
```

The bottom half of the loop computes the new set of points using `reduce`, then draws in the points and the lines connecting them. Construction points in the outermost set are drawn with IDs starting at 3000 so that the interactive controller can determine when they are clicked.

```
    if levels*2 <= const_reveal < levels*2+1:
        color( darkred, const_reveal - levels*2 )
        dot( 2, p[0][0], p[0][1] )
    elif levels*2+1 <= const_reveal:
        color( darkred )
        dot( 2, p[0][0], p[0][1] )
```

Outside the while loop, the final point – the point on the curve – is drawn in red. This point's opacity is also controlled by the `const_reveal` parameter.

```
if show_curve and info.curve:
    v = info.curve
    p = Path().moveto( v[0][1], v[0][2] )
    for vu,x,y in v[1:]:
        if vu > u:
            break
        p.lineto( x, y )
    color( darkred )
    widestroke( p, 1 )
```

After the construction elements are drawn, the curve itself is drawn (if the `show_curve` parameter is true) by building a path object for the $[0, u]$ interval of the curve and stroking the path in red.

Next the control points are drawn in. They are given IDs starting with 2000 so that mouse clicks on them can be easily detected.

```
    i = 2000
    push()
    color( green )
    for x, y in info.controls:
        id( i )
        dot( 2, x, y )
        i += 1
    pop()
```

The last thing to be drawn is the string of text that displays the `u` parameter. The visibility of this string is also controlled by the `const_reveal` parameter; it fades in along with the first set of construction points.

The final line of the diagram function stores the current coordinate system in the `info` object, so that the interactive controller can map mouse coordinates from screen space back into the drawing coordinate system of the diagram.

```
    if show_const:
        if const_reveal < 1:
            color( black, const_reveal )
        else:
            color( black )
        text( 50, 5, 'u = %.2f' % (u,), thefont,
              size = 5, anchor = 'fc' )

    info.last_coords = mark()
```

B.3 The interactive controller

Once we have an appropriately parameterized diagram function, constructing an interactive controller to manipulate the parameters is relatively straightforward. The first method is quite simple:

```
class BezierDemo(Controller):
    def create_objects( self ):
        self.last_coords = None
        self.drag = None
        self.controls = []
        self.curve_dirty = 1
        self.steps = 0
```

```
        self.d = Drawable( None, bezier, info = self )
        return self.d
```

The `create_objects()` method creates the animation elements that will be controlled interactively. In this case we create just a single instance of the Bézier-drawing diagram. Note that the value of the `info` parameter is the interactive controller instance `self`, rather than the `test_info` object we used for testing the diagram by itself. We also use this method to initialize some of the instance variables used in the controller's methods.

The next set of methods are used to handle mouse events.

```
    def mousedown( self, x, y, m ):
        what, = query_id( x, y )

        if 2000 <= what < 3000:
            # user presses button over one of the control points
            self.drag = what

        elif 3000 <= what < 4000:
            # user presses button over one of the outermost
            # construction points
            self.drag = what
            what -= 3000
            x, y = unproject( x, y, self.last_coords )
            u = map_to_line( x, y, self.controls[what],
                                   self.controls[what+1] )
            set( self.d.u, u )
```

The `mousedown()` handler is used to detect the start of dragging interactions. (For detecting individual clicks, the `mouseup()` method is preferred, so that the action happens when the user releases the button, rather than when he or she presses it.) The handler uses the `query_id()` function to determine the object ID present at the pixel location of the button press. The object IDs returned are those set using the `id()` function within the parameterized diagram. In this case, an ID between 2000 and 2999 indicates one of the control points—the handler simply sets the `self.drag` variable to remember which control point was selected, so that it can be moved within the `mousemove()` handler.

An ID in the range 3000–3999 indicates one of the outermost construction points. Each of these points can be dragged back and forth along its control polygon edge to set the subdivision fraction

u. This handler first converts the location of the button press into diagram coordinates, then projects that point onto the control polygon edge to determine the *u* value. The animation command `set()` is then used to set the diagram's u parameter to that computed value.

```
def mousemove( self, x, y, m ):
    d = self.drag
    if d is None:
        return

    if 2000 <= d < 3000:
        # move a control point
        self.controls[d-2000] = unproject( x, y, self.last_coords )
        self.curve_dirty = 1

    elif 3000 <= d < 4000:
        # drag a construction point along the control
        # polygon edge
        what = d - 3000
        x, y = unproject( x, y, self.last_coords )
        u = map_to_line( x, y, self.controls[what],
                               self.controls[what+1] )
        set( self.d.u, u )
```

The `mousemove()` handler is called each time the mouse moves with the button pressed down. It looks at the `self.drag` variable, which contains the object ID of the pixel originally clicked on, and uses that to determine how to change the diagram. Object IDs 2000–2999 represent control points—the handler simply changes the coordinates of the dragged control point and tells the diagram to recompute the curve. Dragging of the construction points is slightly more complicated, since each must be constrained to remain on its edge of the control polygon. The handler projects the position of the mouse onto the control polygon edge in order to determine the value for the diagram's u parameter.

```
def mouseup( self, x, y, m ):
    what, = query_id( x, y )
    x, y = unproject( x, y, self.last_coords )

    if what == 0 and self.drag is None:
        # clicked in the background; add a control point
        self.controls.append( (x, y) )
        self.curve_dirty = 1
```

```
    elif 1000 <= what < 2000:
        # clicked a control polygon line
        what -= 1000
        u = map_to_line( x, y, self.controls[what],
                               self.controls[what+1] )
        set( self.d.show_const, 1 )
        set( self.d.u, u )
        smooth( 1.0, self.d.const_reveal, 0, 1 )
        self.steps = 1

    self.drag = None
```

The final mouse-related handler, `mouseup()`, is primarily for producing actions by clicking (as opposed to dragging). There are two click actions in this applet: clicking any point of the background (object ID 0) adds a new control point. Clicking on any control polygon edge (object IDs 1000–1999) sets the subdivision fraction u and shows the first step of the de Casteljau construction. Subsequent steps are added with keystrokes, implemented in the next handler.

```
def key( self, k, x, y, m ):
    if k == 'a':
        self.steps += 1
        smooth( 1.0, self.d.const_reveal, self.steps )
    elif k == 'g':
        if len(self.controls) > 1:
            set( self.d.show_const, 1 )
            set( self.d.show_curve, 1 )
            set( self.d.u, 0.0 )
            smooth( 0.5, self.d.const_reveal,
                    len(self.controls)*2-3 )
            smooth( 5.0, self.d.u, 1.0 )

            smooth( 0.5, self.d.const_reveal, 0 )
            set( self.d.show_const, 0 )
            self.steps = 0
    elif k == 'c':
        set( self.d.show_const, 0 )
        set( self.d.const_reveal, 0 )
    elif k == 'v':
        set( self.d.show_curve, 0 )
    elif k == 'd':
        if len(self.controls) > 0:
            self.controls.pop()
            self.curve_dirty = 1
```

This handler implements actions for five different keystrokes. The 'a' key advances the de Casteljau algorithm one step by incrementing the `const_reveal` parameter by one, using a smooth interpolation. The 'g' key shows the whole process in one animation: all of the constructions are made visible, then the u parameter is smoothly advanced from zero to one, tracing out a visible curve as it goes. When the curve is complete the construction lines disappear, leaving only the curve and the original control polygon. The remaining three keys perform comparatively simple actions: 'c' turns off display of the construction lines, 'v' turns off display of the curve, and 'd' removes the last control point. All of these actions are implemented by using ordinary animation commands to manipulate the parameters of the diagram function.

B.4 A simple animation

To use this interactive object within a presentation, it must be included as an element in an animation. Here we will describe a simple animation script that begins with the interactive object occupying the whole screen, then shrinks it down to make room for a list of information about Bézier curves.

```
def bezier_anim():
    c = get_camera().left(0.5).inset(0.05)
    bg = Fill( color = bgcolor )
    tx = Text( c.top(0.1).move_right(0.1),
               font = fonts['sansb'],
               text = 'Bzier curves',
               color = yellow,
               size = 24,
               _alpha = 0.0 )
    bl = BulletedList( c.bottom( 0.85 ).move_right(0.1),
                       font = fonts['sans'],
                       color = white,
                       size = 18,
                       bullet = [fonts['ding'],'w'] )
```

Besides the interactive object, the animation has three elements: a background fill, a text object for the "slide title" and a bulleted list object. We start by creating these objects and positioning them on the animation canvas.

```
    vi = viewport.interp( get_camera().inset(0.03),
```

```
                        get_camera().right(0.5).inset(0.05) )
i = Interactive( vi, controller = BezierDemo, _alpha = 0.0 )
```

Because the interactive object will change position over the course of the animation, its position is given not as a static rectangle but as the pseudoelement `vi`, which interpolates between two rectangles.

```
start_animation( bg, i )
fade_in( 0.5, i )
pause()
```

The animation begins very simply: initially only the background and the interactive object are on the screen. The interactive object fades in and the animation pauses. While the animation is paused, the presenter can interact with the diagram using the mouse and keyboard.

```
smooth( 1.0, vi.x, 1 )
enter( tx, bl )
fade_in( 0.5, tx )
pause()
```

When the presenter hits the spacebar, the animation of this script continues. First the interactive diagram shrinks down to fill just the right side of the display (note how this is done by changing the `x` parameter of the pseudoelement controlling the interactive object's position). After the object has shrunk down to make room, the title and bulleted list are added to the slide. The title is initially totally transparent; it is made visible with a fade. There is no need to fade in the bulleted list as it is initially empty.

```
bl.add_item( 0, 'de Casteljau construction', 0.5 )
pause()

bl.add_item( 0, 'infinitely differentiable', 0.5 )
pause()

bl.add_item( 0, 'global control', 0.5 )
pause()

bl.add_item( 0, 'non-interpolating', 0.5 )
```

Each time the presenter presses the spacebar, another item is added to the bulleted list using a half-second fade. The interactive object can be manipulated throughout this whole process.

```
    return end_animation()
bezier_anim = bezier_anim()
```

The animation ends after the last bullet point is added. The last line calls the animation script just defined, and assigns the animation object it returns to the variable `bezier_anim`. All three objects – the diagram, the interactive object, and the animation object – can be passed to the `test_objects()` function to be tested interactively:

```
test_objects( bezier, BezierDemo, bezier_anim )
```

Figure B.3 shows each of these objects running inside the tester.

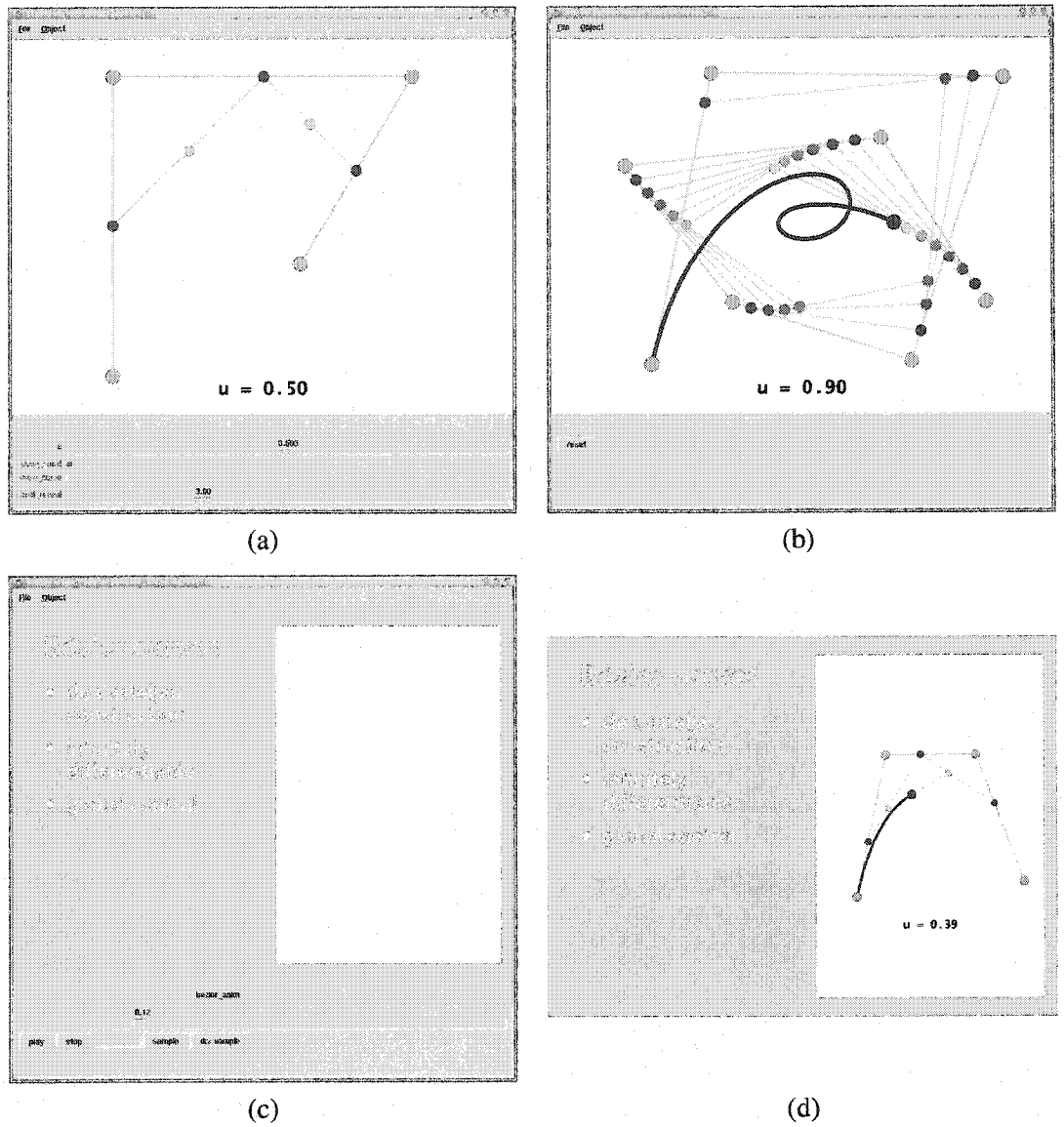


Figure B.3 *The three* SLITHY *objects created for the interactive Bézier applet. Part (a) shows the parameterized diagram. The control point locations are taken from the* `test_info` *object. Part (b) shows the interactive object, with control points specified using the mouse. Part (c) shows an animation that includes the interactive object along with other animation elements like text and a bulleted list. In the tester, interactive objects inside animations are not active, so the interactive object is blank. When shown in a presentation, as in part (d), the object is fully interactive throughout the animation.*

BIBLIOGRAPHY

[1] Adobe. Adobe Illustrator. Computer software.

[2] Alias|Wavefront. Maya 1.0. Computer software.

[3] Brett Allen, Brian Curless, and Zoran Popović. Articulated body deformation from range scan data. *ACM Transactions on Graphics*, 21(3):612–619, July 2002.

[4] Benjamin B. Bederson, James D. Hollan, Ken Perlin, Jonathon Meyer, David Bacon, and George Furnas. Pad++: A zoomable graphical sketchpad for exploring alternative interface physics. *Journal of Visual Languages and Computing*, 7:3–31, 1996.

[5] Mary Helen Briscoe. *Preparing Scientific Illustrations*. Springer-Verlag, New York, 1995.

[6] Marc H. Brown. *Algorithm Animation*. PhD thesis, Brown University, 1987.

[7] Marc H. Brown. Exploring algorithms using BALSA-II. *IEEE Computer*, 21(5):14–36, May 1988.

[8] Steve Capell, Seth Green, Brian Curless, Tom Duchamp, and Zoran Popović. Interactive skeleton-driven dynamic deformations. *ACM Transactions on Graphics*, 21(3):586–593, July 2002.

[9] J. Caraballo. *The Effect of Various Visual Display Modes of Selected Educational Objectives.* PhD thesis, The Pennsylvania State University, 1985.

[10] Lewis Carroll. *The Annotated Alice.* W. W. Norton & Company, 2000.

[11] Lih-Juan ChanLin. Animation to teach students of different knowledge levels. *Journal of Instructional Psychology*, 25:166–175, 1998.

[12] Michael Chmilar and Brian Wyvill. A software architecture for integrated modeling and animation. In R. A. Earnshaw and B. Wyvill, editors, *New Advances in Computer Graphics: Proceedings of CG International '89*, pages 257–276. Springer-Verlag, 1989.

[13] Yung-Yu Chuang, Aseem Agarwala, Brian Curless, David H. Salesin, and Richard Szeliski. Video matting of complex scenes. *ACM Transactions on Graphics*, 21(3):243–248, July 2002.

[14] Yung-Yu Chuang, Douglas E. Zongker, Joel Hindorff, Brian Curless, David H. Salesin, and Richard Szeliski. Environment matting extensions: Towards higher accuracy and real-time capture. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 121–130, July 2000.

[15] Matthew J. Conway. *Alice: Easy-to-Learn 3D Scripting for Novices.* PhD thesis, University of Virginia, 1997.

[16] Matthew J. Conway and Tommy Burnett et al. Alice: 3D interactive graphics programming made easy. *9th Annual ACM Symposium on User-Interface Software and Technology*, November 1996.

[17] Andrew Dillon, Cliff McKnight, and John Richardson. Space — the final chapter, or, why physical representations are not semantic intentions. In C. McKnight, A. Dillon, and J. Richardson, editors, *Hypertext: A Psychological Perspective*, chapter 8. Ellis Horwood, 1993.

[18] Francis M. Dwyer. Adapting visual illustrations for effective learning. *Harvard Educational Review*, 37:250–263, 1967.

[19] Francis M. Dwyer. Effect of visual stimuli on varied learning objectives. *Perceptual and Motor Skills*, 27:1067–1070, 1968.

[20] Francis M. Dwyer. The effect of stimulus variability on immediate and delayed retention. *The Journal of Experimental Education*, 38:30–37, 1969.

[21] Francis M. Dwyer. Visual learning: An analysis by sex and grade. *California Journal of Educational Research*, 22:170–176, 1971.

[22] S. Feiner, D. Salesin, and T. Banchoff. Dial: A diagramatic animation language. *IEEE Computer Graphics & Applications*, 2:43–54, September 1982.

[23] E. L. Ferguson and Mary Hegarty. Learning with real machines of diagrams: Application of knowledge to real-world problems. *Cognition and Instruction*, 13:129–160, 1995.

[24] Free Software Foundation. GNU Emacs. Computer software.

[25] Python Software Foundation. Python 2.2. `http://www.python.org/`.

[26] Robert Mills Gagné. *The Conditions of Learning and Theory of Instruction*. Holt, Rinehart, and Winston, New York, 1985.

[27] Lance Good and Benjamin B. Bederson. Zoomable user interfaces as a medium for slide show presentations. *Information Visualization*, 1(1):35–49, March 2002.

[28] Ronald J. Hackathorn. Anima II: a 3-D color animation system. In *Computer Graphics (Proceedings of SIGGRAPH 77)*, volume 11, pages 54–64, San Jose, California, July 1977.

[29] Pat Hanrahan and David Sturman. Interactive animation of parametric models. *The Visual Computer*, 1(4):260–266, December 1985.

[30] Edward Hodnett. *Effective Presentations: How to Present Facts, Figures, and Ideas Successfully*. Parker Publishing, New York, 1967.

[31] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 2002 (in press).

[32] John Lasseter. Principles of traditional animation applied to 3D computer animation. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, volume 21, pages 35–44, July 1987.

[33] C. Karen Liu and Zoran Popović. Synthesis of complex dynamic character motion from simple animations. *ACM Transactions on Graphics*, 21(3):408–416, July 2002.

[34] Macromedia. Flash MX. Commercial software package.

[35] Richard E. Mayer and Richard B. Anderson. Animations need narration: An experimental test of a dual-coding hypothesis. *Journal of Educational Psychology*, 83(4):484–490, 1991.

[36] Richard E. Mayer and Richard B. Anderson. The instructive animation: Helping students build connections between words and pictures in multimedia learning. *Journal of Educational Psychology*, 84(4):444–452, 1992.

[37] Microsoft. PowerPoint 2000. Computer software.

[38] Microsoft. PowerPoint XP. Computer software.

[39] Julie Bauer Morrison, Barbara Tversky, and Mireille Betrancourt. Animation: Does it facilitate learning? In *Smart Graphics: Papers from the 2000 AAAI Symposium*, pages 53–60, 2000.

[40] Walter Murch. *In the Blink of an Eye: A Perspective on Film Editing*. Silman-James Press, Los Angeles, 1995.

[41] Tom Ngo, Doug Cutrell, Jenny Dana, Bruce Donald, Lorie Loeb, and Shunhui Zhu. Accessible animation and customizable graphics via simplicial configuration modeling. In *Proceedings of SIGGRAPH 2000*, pages 403–410, 2000.

[42] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[43] Allan Paivio. *Mental Representations: A Dual Coding Approach*. Oxford University Press, New York, 1990.

[44] Ok-choon Park and Stuart S. Gittelman. Selective use of animation and feedback in computer-based instruction. *Educational Technology Research & Development*, 40(4):27–38, 1992.

[45] Ok-choon Park and R. Hopkins. Instructional conditions for using visual displays: A review. *Instructional Science*, 21:427–449, 1993.

[46] Ian Parker. Absolute PowerPoint: Can a software package edit our thoughts? *The New Yorker*, 2001.

[47] Ken Perlin and David Fox. Pad: An alternative approach to the computer interface. In *Proceedings of SIGGRAPH 93*, 1993.

[48] H. J. Peters and K. C. Daiker. Graphics and animation as instructional tools: A case study. *Pipeline*, 7:11–13, 1982.

[49] S. K. Reed. Effects of computer graphics on improving estimates to algebra word problems. *JEP*, 77:285–298, 1985.

[50] William T. Reeves, Eben F. Ostby, and Samuel J. Leffler. The Menv modelling and animation environment. *Journal of Visualization and Computer Animation*, 1(1):33–40, August 1990.

[51] Craig W. Reynolds. Computer animation with scripts and actors. In *Proc. SIGGRAPH 82*, pages 289–296, July 1982.

[52] Lloyd P. Rieber. The effects of visual grouping of textual and animated presentations on intentional and incidental learning. Unpublished raw data.

[53] Lloyd P. Rieber. The effects of computer animated elaboration strategies and practice on factual and application learning in an elementary science lesson. *Journal of Educational Computing*, 5:431–444, 1989.

[54] Lloyd P. Rieber. Animation in computer-based instruction. *Educational Technology Research & Development*, 38(1):77–86, 1990.

[55] Lloyd P. Rieber. Using computer animated graphics in science instruction with children. *Journal of Educational Psychology*, 82:135–140, 1990.

[56] John T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.

[57] Paul S. Strauss. *BAGS: The Brown Animation Generation System.* PhD thesis, Brown University, May 1988. TR No. CS-88-22.

[58] S. V. Thompson and R. J. Riding. The effect of animated diagrams on the understanding of a mathematical demonstration in 11- to 14-year-old pupils. *British Journal of Educational Psychology*, 60:93–98, 1990.

[59] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, Connecticut, 1983.

[60] Edward R. Tufte. *Envisioning Information.* Graphics Press, Cheshire, Connecticut, 1990.

[61] Edward R. Tufte. *Visual Explanations: Images and Quantites, Evidence and Narrative.* Graphics Press, Cheshire, Connecticut, 1997.

[62] Barbara Tversky, Julie Bauer Morrison, and Mireille Betrancount. Animation: Can it facilitate? *International Journal of Human Computer Studies*, 57(4):247–262, October 2002.

[63] D. Eric Walters and Gale Climenson Walters. *Scientists Must Speak: Bringing Presentations to Life*. Taylor & Francis, London, 2002.

[64] Douglas E. Zongker, Geraldine Wade, and David H. Salesin. Example-based hinting of truetype fonts. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 411–416, July 2000.

[65] Douglas E. Zongker, Dawn M. Werner, Brian Curless, and David H. Salesin. Environment matting and compositing. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 205–214, August 1999.

VITA

Douglas Earl Zongker was born in 1976 in Olathe, Kansas, where he grew up. He spent four years at Michigan State University in East Lansing, Michigan, obtaining a BS degree in computer science in 1996. He moved to Seattle, Washington, to spend seven years at the University of Washington, receiving an MS degree in computer science in 1998 followed by a PhD in 2003. He will be heading to Mountain View, California, to join the engineering staff at Google.