Eurographics/SIGGRAPH Symposium on Computer Animation (2003)
D. Breen, M. Lin (Editors)

On Creating Animated Presentations

Douglas E. Zongker[1] David H. Salesin[1,2]

[1] University of Washington, Seattle, Washington, USA
[2] Microsoft Research, Redmond, Washington, USA

Abstract

Computers are used to display visuals for millions of live presentations each day, and yet only the tiniest fraction of these make any real use of the powerful graphics hardware available on virtually all of today's machines. In this paper, we describe our efforts toward harnessing this power to create better types of presentations: presentations that include meaningful animation as well as at least a limited degree of interactivity. Our approach has been iterative, alternating between creating animated talks using available tools, then improving the tools to better support the kinds of talk we wanted to make. Through this cyclic design process, we have identified a set of common authoring paradigms that we believe a system for building animated presentations should support. We describe these paradigms and present the latest version of our script-based system for creating animated presentations, called SLITHY. *We show several examples of actual animated talks that were created and given with versions of* SLITHY*, including one talk presented at SIGGRAPH 2000 and four talks presented at SIGGRAPH 2002. Finally, we describe a set of design principles that we have found useful for making* good *use of animation in presentation.*

Categories and Subject Descriptors (according to ACM CCS): I.3.4 [Computer Graphics]: Graphics Utilities – application packages I.3.6 [Computer Graphics]: Methodology and Techniques – languages

1. Introduction

By Microsoft estimates, at least thirty million PowerPoint presentations are made every day.[16] Even if this estimate is off by an order of magnitude, the implication is clear: presentation software is a technology that is having an impact on people's lives.

Modern-day presentation software – of which PowerPoint, in representing 95% of the presentation-software market, is the most prominent example by far – is still rooted firmly in the past. Although the software has evolved in many ways, PowerPoint presentations are still essentially static in nature, just as they were when the software, originally designed to create overhead transparencies, was first released in 1987. Even in the latest, animation-enhanced PowerPoint XP, what limited animation capabilities there are exist almost entirely to provide "canned" embellishments to the static layout of the slide—a snazzy entry or exit for a given text or graphical element, or a way of momentarily highlighting a particular element.

As researchers and educators, we give a lot of talks, and we sit through even more. Our own lives would be improved if we could give – and receive – better talks. This paper explores how computers might be used to help us communicate more effectively. In particular, we examine how computers could be used to create meaningful animation, as well as some degree of interactivity, to improve live presentations.

Our approach to this problem has been iterative: we began by trying to make talks that incorporated animation and interactivity using existing software tools. This 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 have coalesced our observations about strategies for authoring animated presentations into a set of three authoring principles, which we discuss in Section 2.

Our current system is called SLITHY. It is an animation tool designed specifically for creating and giving presentations. In designing SLITHY, we ideally wanted to accommodate as wide a range of users as possible. However, try as we

might, we were unable to imagine any single graphical user interface – the type of interface, perhaps, that the highest number of users would find intuitive – that could encompass the staggering variety of animations that we could envision authors wanting to create. Ultimately, we chose to emphasize power over ease of use. SLITHY is therefore a script-based programming system, analogous to TEX for text processing, and as such is better suited for use by more technically-inclined users. Despite these limitations, SLITHY has been used to give a number of presentations (including four at SIGGRAPH 2002 by users other than the authors of the system). 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, but still significant community (including, but not limited to, the technical contributors to SIGGRAPH and other computer graphics conferences) is a worthwhile step. The design and implementation of SLITHY are covered in Section 3.

It is an open question among cognitive psychologists as to whether or not animation improves learning. A number of studies[3, 15, 20] have found a positive effect, but other researchers criticize these results on methodological grounds.[13, 21] The central issue seems to be determining how to make two presentations, one animated and one not, that are exactly equivalent, "except for the animation." Despite the lack of conclusive psychological research, people *are* using animation, even if it is only the simple effects available in PowerPoint. In our experience, audiences seem to appreciate a richer style of animation even more. As we made more and more of these animated talks, we were also interested in learning how *best* to apply animation in presenting material. If animation is going to be used, we can at least try to make it as useful as possible. We have 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 animation[10] do not necessarily work so well for presentations. In Section 4 we detail our observations on principles for good *presentation* animation.

Finally, Section 5 shows examples of some presentations created with our system, Section 6 compares SLITHY, the system we built, to other existing systems, and Section 7 presents some conclusions and directions for future work.

2. Authoring principles

Our first set of principles is concerned with techniques for authoring animation. Since presentation animation commonly has a different purpose and visual style than character animation, we expect that authors will demand a different set of tools for creating the animation. Here we discuss three general authoring techniques that we have found to be useful.

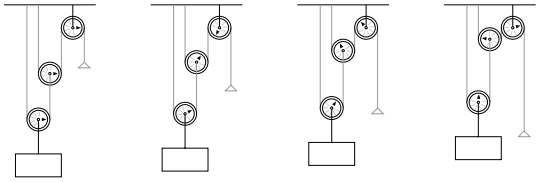


Figure 1 *Four instances of a pulley diagram, with the handle in different positions. Parameterization lets us animate the diagram by manipulating a single abstract "amount of pull" parameter, rather than managing all the individual graphical elements individually.*

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 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.*[14] This simplifies the task of animation considerably. Consider the pulley diagram of Figure 1. 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 just once the mapping between model parameters and the underlying geometry; 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 repeatedly throughout the talk, from the animated transitions to the layout of text on slides. We desire support for creating all these elements through parameterizable macros, partly to avoid repetitious work by the author and partly 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.

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 spe-

cial about "animations" is that their input parameter set happens to consist of a single scalar value. The advantage of this approach is that an animation object does not have to contain everything visible on the screen at once. Instead, we can 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.

Build slides hierarchically. 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.

3. Slithy

Our presentation system, SLITHY, is implemented as a set of libraries and a runtime system for the popular programming language Python. SLITHY users therefore have access to a complete, general-purpose programming language for use in creating their animations. A presentation in SLITHY can be thought of as a collection of *drawing objects*. There are three major types of drawing object available in SLITHY:

- *Parameterized diagrams* can require an arbitrary set of parameters as input, and they produce their graphical output imperatively by executing a procedure that makes calls to the SLITHY drawing library. The user creates a parameterized diagram by writing a Python function; this Python function is executed every time the diagram needs to be redrawn. These functions can contain arbitrary Python code; they are not limited to the primitives available in our graphics library. They can also invoke other parameterized diagrams or animation objects.
- *Animation objects* require exactly one scalar parameter, which we will typically think of as representing time. The object provides a mapping from the time parameter to a set of other drawing objects to invoke, along with values for their parameters. This kind of object is constructed by writing an *animation script* in Python. The script is executed just once to produce the animation object. Each command in the script edits the mapping that the object represents; the finished object is returned at the completion of the script. The SLITHY runtime system can then "play" an animation object by repeatedly invoking it, passing in the current time as the value of its parameter.

  A single animation object can control the parameters of multiple other drawing objects. In addition to user-created parameterized diagrams, the system also has a number of built-in objects to display things like background fills, text boxes, still images, and bulleted lists. These objects are essentially very simple prefabricated parameterized diagrams created to implement commonly used slide elements.
- *Interactive objects* are similar to animation objects in that they represent a mapping from a single scalar time parameter to a set of other drawing objects and their parameters. The difference is that while animation objects are created by a single script, executed just once when the presentation is loaded, interactive objects can be edited while they are being played. The author writes an *interactive controller* that contains handlers for input events such as keystrokes and mouse movements. The handlers can then modify the animation being shown. With interactive controllers, the presenter can effectively generate a new animation object during the presentation.

Every drawing object takes a set of input parameter values and produces graphics on its own notionally infinite canvas. A *camera rectangle* specifies what region of that canvas must be visible in the object's viewport. An object's viewport may be placed on the canvas of another object. In this way, drawing objects can contain each other in a hierarchy. Each object is responsible for providing parameters to the objects it contains. The top object of the hierarchy is always an animation object, whose viewport is the entire SLITHY window, and whose single time parameter is driven by the computer's real-time clock.

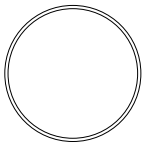
The remainder of this section will discuss the three classes of drawing objects and their implementations in more detail.

3.1. Parameterized diagrams

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

We will illustrate some of the features of our parameterized diagram system by building a simple example—constructing an analog clock face. We begin with this six-line function:

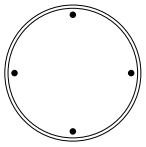
```
def clock_face():
    set_camera( Rect(-12, -15, 12, 12) )
    clear( white )
    thickness( 0.25 )
    circle( 10, 0, 0 )
    circle( 10.5, 0, 0 )
```



The Python keyword `def` is used to introduce a new function, named `clock_face` in this example. The other five lines are calls to functions in the drawing library. First we specify what rectangular portion of the infinite canvas must be visible in the diagram's viewport—in this case, the area from $(-12,-15)$ to $(12,12)$. Then we clear the diagram's canvas to white, set the line thickness, and draw a pair of concentric circles around the origin. Note that diagrams are written in a straightforward imperative style of programming. This function is called every time the diagram is to be drawn (typically, once per frame of animation). There is no object state to track from one invocation to the next; the appearance is completely specified by the sequence of drawing library routines used in the current call to the function.

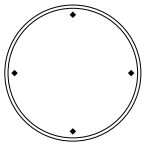
Continuing the example, suppose we want to place marker dots at the 12, 3, 6, and 9 o'clock positions. We could do this with four calls to the `dot` drawing function (which produces a filled circle), but instead we'll encapsulate the marker-drawing code in a new helper function and call that instead. To the function started above, we add:

```
    marker( 9, 0 )
    marker( 0, 9 )
    marker( -9, 0 )
    marker( 0, -9 )

def marker( x, y ):
    dot( 0.5, x, y )
```



This version of the code creates a function `marker` that takes two parameters, `x` and `y`, and draws a marker at the indicated position. For simply drawing a dot this might be overkill, but suppose we then wanted to change every marker from a dot to a diamond. We could do this quite easily by changing the body of the `marker` function.

```
diamond = Path()
diamond.moveto(0.5,0).lineto(0,0.5)
diamond.lineto(-0.5,0).lineto(0,-0.5)
diamond.closepath()

def marker( x, y ):
    push()
    translate( x, y )
    fill( diamond )
    pop()
```



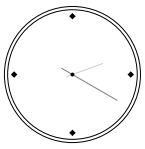
Unlike `dot`, there is no built-in function for drawing diamonds. Here we build one using a path object, another feature provided in our drawing library. With path objects, users can describe arbitrary paths constructed of line and Bézier curve segments, similar to the path construction operators in PostScript. The path object can then be instanced in the diagram using the `stroke` and/or `fill` functions. Here we define a path called `diamond`, which can be drawn within the `marker` function with a single call to `fill`. To position the diamond correctly, we use the `push` and `pop` library functions, which save and restore the current graphics state (including transform matrix, drawing color, etc.), and the `translate` function, which changes the origin of the drawing coordinate system.

Now we will draw a clock with hands. Since we are creating a parameterized diagram rather than a static picture, of course, the time shown on the clock should come as a parameter to the function. We'll begin by declaring a new function `clock` and use the already defined function `clock_face` to draw the dial:

```
def clock( minutes=(SCALAR, 0, 1440),
           label=(STRING, 'San Diego') ):
    clock_face()
```

The `clock` function takes two parameters. One, `minutes`, represents the number of minutes past midnight to display on the clock, while the `label` parameter is a text string to be shown beneath the clock face. Scalar-valued parameters like `minutes` are specified along with their allowable range of values. We can then make use of these parameter values in drawing:

```
    minute_angle = minutes * 6.0
    hour_angle = minutes / 2.0

    push()
    rotate( -minute_angle )
    color( gray50 )
    line( 0, -2, 0, 8 )
    pop()

    color( black )
    text( 0, -13, label, font = labelfont,
          size = 3, anchor = 'c' )
```



The first two lines use the value of the `minutes` parameter to compute the appropriate angles for the minute and hour hands. This computation is done with the ordinary Python arithmetic operators, and the results are assigned to a new local variable. Note that the Python language is dynamically typed, and does not require variable declarations. These features, among others, make Python well suited for beginning programmers.

Since the drawing is constructed as a Python function, all of the features of the Python language are available: operators for computation, control structures such as `if` and `while`, and a rich set of data types including lists and dic-

tionaries. It is not necessary to use all of these abilities for simple diagrams like this example, but they can be helpful for making more complex figures.

The next group of lines actually draw the minute hand by rotating the coordinate system through the appropriate angle and drawing a gray line. A similar block of code (not shown here) is used to add the hour hand as well. Finally, the string-valued `label` parameter is drawn beneath the clock face.

Parameterized diagrams can be tested by loading them into SLITHY's test harness, which lets the user interactively manipulate the diagram's parameters via on-screen widgets and see the results. Users can also click in the diagram and see those points back-projected into the diagram's coordinate system; this aids in placing objects on the canvas. Figure 3(a) shows the clock diagram in the test window—the `minutes` and `label` parameters are mapped to by a slider and a text box, respectively.

3.2. Animation scripts

Our system applies the character animation technique of building models and animating them via high-level controls. Parameterized diagrams provide a way to express one half of this scheme—mapping from the control parameters onto the output drawing. Animation objects provide a convenient way to specify the other mapping—from a single time value onto a set of values for the model control parameter. In contrast to the procedural nature of parameterized diagrams, where the user code is executed every time the diagram is drawn, once per frame, an animation script is executed only once, during the initialization of the presentation. The script builds an animation object that the SLITHY system can then sample to draw the animation. An example script is shown in Figure 2, with the resulting animation illustrated in Figure 5.

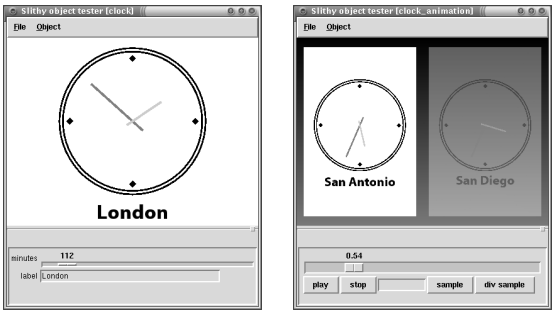
Every parameter controlled by an animation object is represented by a data structure called a *timeline.* The timeline partitions the entire range of possible time values (from negative infinity to positive infinity) into a set of nonoverlapping domains. For each domain, the timeline contains either a constant value for the parameter, or a function that can be called to produce the parameter's value within that domain.

When an animation object is created, a trivial timeline is created for each parameter under the animation's control. This trivial timeline is just a single domain covering all of time, containing the parameter's default value. Subsequent commands within the animation script then edit these timelines to produce the desired animation. Using the `linear` command on a parameter, for instance, will overwrite part of a parameter's timeline with a pair of new domains: one expressing linear interpolation to a new value and one containing the new value for all the following time. This and other timeline editing commands are illustrated in Figure 4.

While the script is executing, the system maintains a

```
def clock_animation():
  bg = Fill( style='horz', color=black, color2=darkgray )
  left = Drawable( get_camera().left(0.5).inset(0.05),
                   clock )
  right = Drawable( get_camera().right(0.5).inset(0.05),
                    clock, _alpha=0.0 )
  start_animation( bg, left, right )

  set( left.label, 'San Antonio' )
  set( left.minutes, 195+120 )
  set( right.minutes, 195 )

  parallel()
  smooth( 3.0, bg.color2, lightgray )
  linear( 3.0, left.minutes, 300+120 )
  linear( 3.0, right.minutes, 300 )
  fade_in( 1.5, right )

  serial()
  wait( 1.5 )
  fade_out( 1.5, left )
  end()
  end()

  return end_animation()
```

Figure 2 *A script for creating a four-second animation containing three drawing objects: a gradient background fill and two `Drawables`, which are containers for other drawing objects (usually parameterized diagrams). In this case both drawables contain instances of the clock example diagram from Section 3.1. The body of the script (between the `start_animation` and `end_animation` calls) consists of commands to manipulate the parameter timelines of the animation's graphical elements.*

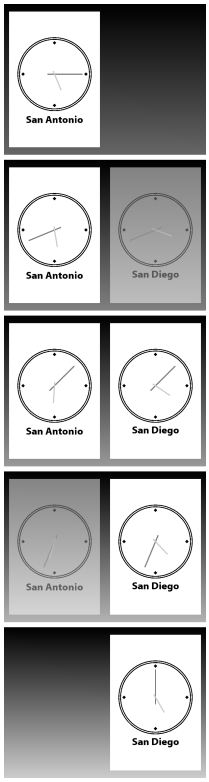


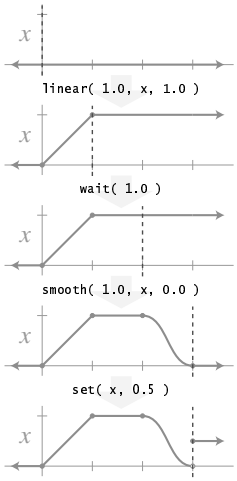
(a) (b)

Figure 3 *Two screenshots of the* SLITHY *object tester. Part (a) shows the clock parameterized diagram; the controls correspond to the diagram's parameters. In part (b), the tester is showing the animation of Figure 2; the controls are a time slider with "play" and "stop" buttons.*

value called the *time cursor,* which specifies where edits will take place. At the start of a script the system is in *serial mode*, which means that every edit command with a duration (such as `linear`, which produces linear interpolation, and `smooth`, which produces smooth interpolation) advances the cursor by that duration. Animation commands will then happen in sequence, one beginning when the previous one ended. The whole script has a single time cursor, so even edits applied to different parameters will not overlap.

To make simultaneous changes to different parameters, the script can use *parallel mode*. In parallel mode the cur-





▲ ***Figure 4*** *The effect of a series of animation script commands on the timeline of a single parameter x. The dotted line represents the position of the time cursor.*

◀ ***Figure 5*** *Frames from the animation of Figure 2.*

sor is *not* advanced after each edit command, so edits begin at the same time. When the script exits parallel mode, the cursor is advanced to the end of the longest of the parallel components. Uses of parallel mode and serial mode can be nested within one another to produce complex overlapping effects.

Other edit commands include `set`, a zero-duration edit that produces an instantaneous change in a parameter's value, and `wait`, which moves the time cursor without changing any of the timelines. Figure 4 illustrates the effect of a series of these commands on a simple timeline. The `get` function can be used to obtain a parameter's value at any point in time.

In addition to the timelines of parameter values for drawing objects contained in the animation, there is also a specialized timeline called the *working-set* timeline that determines which of those objects are drawn and their stacking order at any point in time. The details of this timeline are hidden from the user; it is instead manipulated via the functions `enter` and `exit`, which add and remove objects from the animation, and `lift` and `lower`, which change the stacking order. The `pause` command marks a point where SLITHY will stop and wait for the presenter to press the spacebar before continuing.

Loading an animation object into the test harness as in Figure 3(b) allows the user to interactively scrub to arbitrary points in time as well as playing it back in the normal fashion.

3.3. Interactive controllers

The implementation of an interactive controller is very similar to that of an animation script. Instead of a single function that creates all of the animation, though, a controller is implemented as a class. An instance of this class is created when the interactive object is first shown on the screen. Just like an animation object, an interactive object contains a set of child drawing objects and timelines for controlling their parameters. Unlike an animation object, though, interactive objects can also have various methods that are called to edit the timelines while the animation is being played in response to user input events such as keypresses and mouse movements.

Every time one of these event-handling methods is called, the time cursor is positioned at the current playback time so that edits made by the method will appear immediately. All of the commands available within animation scripts can also be used in these animation-editing methods of interactive controllers: parameter timelines can be modified with `linear`, `smooth`, `set`, etc.; the time cursor can be controlled with `parallel`, `serial`, `wait`, etc.; and the `enter`, `exit`, `lift`, and `lower` functions can modify the set of child drawing objects.

3.4. GUI tools for authoring

Creating presentations by programming may be fine for some technical users, but we hope to eventually make high-quality animations available to a wide variety of presenters. Interactive interfaces for authoring arbitrary animation tend to be quite complex. We believe that by writing tools for specific, small domains, we can limit the range of animations enough to make interactive specification feasible, while still producing useful, content-rich animations. 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 opportunites 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. Figure 8 shows screenshots of these two applications, as well as the animated output of the slideshow tool.

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 typesetting options, people simply chose them *all,* even within a single document. The message was not "look at my content," but "look at what my software can do." Today, too many presentations use animation with similar results. Animation can enhance the content, or it can be visually distracting. By summarizing the results of our experience in making animated presentations as a set of general principles, we hope to encourage the former, 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.

Make all movement meaningful. 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.

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 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.

Avoid instantaneous changes. We suggest making smooth transitions – even something as simple as a cross-fade – the standard way of getting information on and off the screen. Sudden cuts between states of a diagram create uncertainty and tension, causing the audience to focus on the screen so that nothing important is missed. 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.

Good and Bederson[8] 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.

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.*[6] summarize a number of studies supporting a positive correlation between memory for location and memory for content in both text and electronic documents.

Smoothly expand and compress detail. A closely related principle is that of using animation to expand and compress detail. In the previous principle 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 the screen 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. With static slides the screen jumps between scales, which typically requires explanation by the presenter and effort by the audience to make the mental links between the different views.

With animation, this kind of navigation becomes 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 highlighting and explanation from the presenter. 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. 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.

Do one thing at a time. 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, 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. 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.[11, 12]

Distinguish dynamics from transitions. Our final animation principle also deals with reducing the potential for confusion by the audience. We 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 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 the distinction between dynamics and transitions is clear. It is very easy to create animation that can be misinterpreted. As an example, one of our users was using a prototype of our 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 a clever animated transition between the two illustrations. 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.

5. Experience and examples

Figure 7(a) shows some still frames from a SLITHY presentation on image matting. In this sequence the viewer zooms in on one region of interest in a picture. Animation is then used to show how the plot on the left is derived from the image pixels. Further animations (not shown here) then illustrate the operation of the algorithm in the abstract space. This is a good use of animation to illustrate content, rather than just catching the viewer's eye.

Our next example, Figure 7(b), begins with a high-level overview of the system described, introducing the desired input and output as iconic images. As other elements are introduced, the two images shrink and spread apart to make room, preserving the continuity of the sequence. At the end, all the elements slide off save one, which is used to demonstrate a number of challenges with the approach.

The example presentation of Figure 7(c) uses a block diagram of the overall system as a navigation aid for the talk. One by one the blocks are highlighted, with the camera zooming in on each to start the in-depth explanation of that block. This gives a strong impression that the detail slides are located "within" the corresponding block. This structure is used recursively. The figure shows part of the sequence contained within the "Momentum constraints" block. The various types of momentum constraints are presented as a series of animated subfigures; the camera zooms in more to focus on each one. At the end, the camera pulls back to show all of the subfigures in their end state, then pulls back again to return to the original block diagram. This sequence was produced by one of our volunteer users before SLITHY supported arbitrary nesting of objects. Seeing it motivated us to add deep hierarchical assembly to the system so that the detail animations could actually appear within the blocks, rather than camera movement just giving that impression.

Figure 7(d) is a good example of expanding and compressing detail. This sequence starts by animating the construction of a single Bézier curve. The camera then pulls back to reveal that the curve is one of many. After those curves are constructed, forming the boundary of a solid area, the camera pulls back again to show that the curves define a character, before superimposing a grid and illustrating the rasterization process. Without animation, it would be more

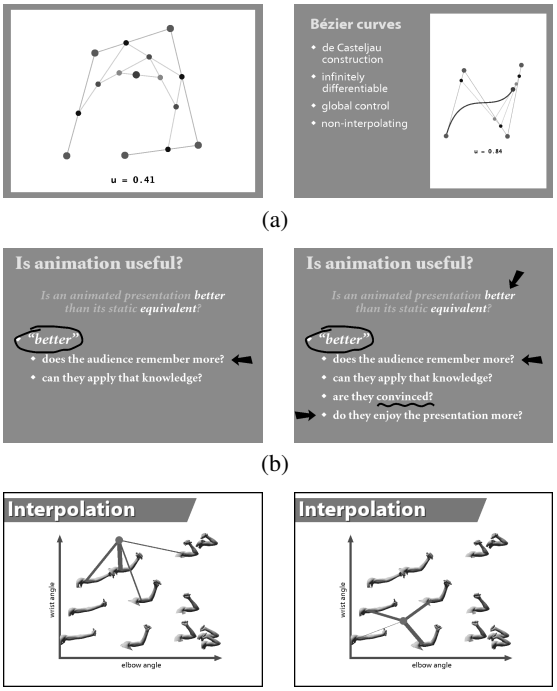


Figure 6 *Three examples of interactive objects. Example (a) lets the presenter interactively place control points to illustrate the de Casteljau algorithm for drawing Bézier curves. Interactive objects can coexist with other* SLITHY *elements—example (b) shows an interactive annotation drawing tool running on top of a text slide. Example (c) comes from the presentation of Figure 7(b). It shows the operation of a k-nearest-neighbor algorithm; lines are drawn between the mouse cursor position ond the four nearest neighbors, with line thickness used to indicate a weight.*

difficult to make clear the relationship between the diagrams at three different scales. Animation obviates the need for any verbal explanation at all.

Figure 6(a) is a simple interactive object that displays a pie chart and allows the user to highlight any wedge by clicking on it. The clicked wedge moves outward with a smooth, animated motion. Figure 6(b) is a more elaborate interactive diagram, from the same talk as Figure 7(b). It illustrates a *k*-nearest-neighbor-based interpolation algorithm; by moving the mouse cursor around in the diagram the sample point is linked to its four nearest neighbors by drawing green lines, with line thickness used to indicate the weight of each neighbor.

6. Comparisons to other systems

Although the SLITHY system itself is just one of the contributions we hope to have made in this paper (the others being, primarily, the principles underlying both the design of any system for creating animated presentations and the design of the animations themselves), it is nevertheless instructive to try to compare SLITHY, as a system, to other related commercial and research systems. Although an exhaustive comparison would take more space than we can reasonably afford, we will at least look at a few of the most salient systems, which we divide into two major classes: systems designed for creating *presentations*, and systems designed for creating *animation*.

6.1. Presentation software

PowerPoint. PowerPoint makes designing static slides very simple, through an intuitive WYSIWYG graphical interface. PowerPoint also features a palette of animation effects that can be applied to slide elements. However, it is extremely difficult to create meaningful animation (the kinds of animation we have termed "dynamics" in Section 4) using PowerPoint's fixed library of effects. Even something as simple as the animated pulley diagram of Figure 1 would be next to impossible to create. For complex animations users must resort to rendering a video file in some other application and playing it from within PowerPoint. Moreover, PowerPoint provides templates for slide layout, but these templates are not parameterized. The user must select a template for each slide and fill in content manually; there's no way to, for instance, say "take this template and create ten slides using these ten image files."

In contrast to PowerPoint's design as a word processor for slides, SLITHY has been built from the beginning with animation in mind. In part to make it fully flexible, and in part to avoid the interface complexities of GUI animation systems such as Maya or Flash, we have chosen a scripting model, where authors write programs that *describe* their animations, rather than creating them directly through graphical interaction. Although certain operations like positioning elements on the screen are more difficult to do with our approach, we have tried to reduce these difficulties with tools that allow users to rapidly test and refine their code, and interactively query canvas coordinates. We have also developed some prototype GUI-based tools that author SLITHY code for certain narrowly defined domains (as described in Section 3.4).

CounterPoint. CounterPoint,[8] is a presentation system implemented using the Jazz toolkit,[1] which itself is a descendent of the Pad "zoomable sketchpad" system.[17] CounterPoint allows PowerPoint slides to be interactively scaled and positioned in arbitrary locations on a large canvas, and paths representing the order of the presentation to be drawn through the slides. Animated zooms and pans across this canvas are then used to transition from one slide to the next. The animation in CounterPoint is focused exclusively on using animated navigation between slides to convey the *structure* of the presentation. In our work, we want to support the animation of *content* as well. SLITHY can be used to create both kinds of animation.

6.2. Creating animation

As we iteratively refined our own system, we were influenced by previous script-based animation systems going back to ANIMA II[9] and DIAL.[7] We use a general-purpose programming language as in ASAS.[19] Modeling and animation are integrated in one language as in CHARLI.[4] We will address a few of the most closely related systems in more detail.

Menv. The Menv system[18] is arguably one of the most successful efforts at using a script-based system for producing animation. A descendant of the system is still in use today by Pixar for producing animated feature films. Models are created in Menv using a specialized language, with primitives for creating 3D geometry and performing common graphics operations. 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. All of these issues are as relevant for the creation of abstract 2D figures in SLITHY as they are for the creation of realistic 3D characters, yet this style of authoring is not commonly seen in 2D tools.

Algorithm animation. One area in which animation systems have been designed for presentation use is in animating algorithms. The Zeus system[2] is typical in that it works by taking an implementation of an algorithm and instrumenting it so that the events that happen in the course of execution are reflected in the graphical display. It is not clear how this style of generating animation would be extended to things that are not algorithms, though.

Alice. Like SLITHY, the Alice project of Conway *et al.*[5] created a graphics programming environment based on the Python programming language. Alice, however, was specifically targeted at users with no graphics or programming experience. It had no modeling component; the animations were created by applying various transforms to premade 3D objects. The emphasis was on creating interactive worlds rather than scripting stories. Alice animation scripts were attached to events such as mouse or keyboard inputs, or collisions between 3D models. Executing an Alice script immediately fed a set of commands for updating the world to a central renderer. This allowed scripts to be activated in parallel, allowing users to create a world full of objects with interesting behaviors. This model makes it difficult to access an animation at arbitrary points in time: there was no representation of the animation apart from the script itself, which had to be run from the beginning. In SLITHY, executing an animation script results in an intermediate animation object, which can be sampled and manipulated arbitrarily. This kind of flexibility is especially important during the authoring process.

Flash. One of the most widely used 2D animation systems today is Flash, from Macromedia. Flash was designed for use on the web. It is a 2D, vector-based keyframe animation system.

The major limitation of Flash (and of a similar, competing product from Adobe called LiveMotion) is that there is no obvious way to create models with complex controls without drawing the graphical elements using hand-written code, just as in SLITHY. The interactively-created drawing primitives and graphical timeline allow only simple transformations to be specified. Primitives can be grouped together, but only simple transformations such as applying affine transforms and modifying opacity can be applied to the group. Without the ability to express nontrivial mappings from the abstract parameters of a group to the parameters of its members, it is impossible to encapsulate interesting behavior and expose that to the animator as a high-level control. (While an animation can be composed of many clips layered together, the only controls offered by each clip are position on the screen, opacity, and which frame is being shown. A clip could be used as a model, but only if that model required just a single scalar parameter, which would be mapped onto the frame number.)

Moreover, we believe there is an advantage in using a script-based interface for expressing the kind of simple, frequently repeated animations used in presentations. Using Flash's scripting language to create intricate complex animations is possible, but awkward. Flash's built-in graphical timeline can only be edited interactively using the mouse. The scripting language can not be used to describe animations using the timeline. To specify animation procedurally, one can write a callback function that is called once per frame and updates parameters of the graphical elements manually based on the frame number. However, manually positioning primitive objects as a function of time is tiresome and error-prone. In contrast, SLITHY is built around the concept of making parameterized models, just as in 3D character animation systems. This parameterization is extended to every part of the system. Even animation scripts can themselves be parameterized, letting users create not just animations but animation-generators, making it easier to automate complex or frequently repeated tasks.

7. Future work and conclusion

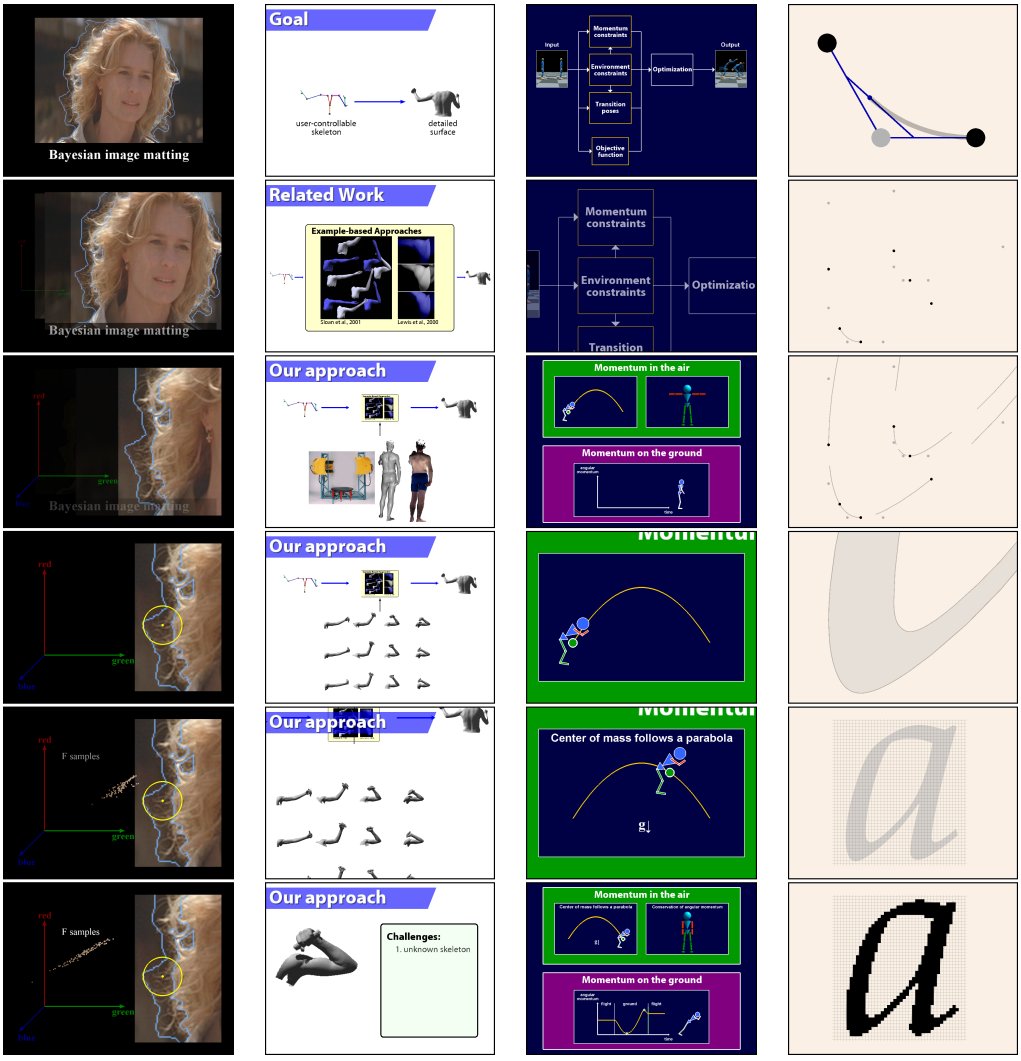
There is still a great deal to be done. Most importantly, we don't yet know how to make an animated presentation tool that is both very general and easy to use. We believe the

prototype tools described in Section 3.4 have promise, but it will take a great deal of work and testing to determine if this is really the way to make animation available to the masses.

Every day computers are being used to tell stories and present ideas in boardrooms and classrooms around the globe. There is a great opportunity here for computer graphics to significantly improve this widely-used medium. When we started working on this problem – trying to design an easy-to-use system to support all kinds of arbitrary animation – it was not at all obvious to us even what kinds of animation would work well for presentations, let alone how to design a system to create them. We feel that the system and principles presented here, while by no means the final word, do at least provide some provocative and useful first steps toward allowing us to create and experience more informative and exciting presentations.

References

1. Benjamin B. Bederson, Jon Meyer, and Lance Good. Jazz: An extensible zoomable user interface graphics toolkit in java. In *Proceedings of User Interface and Software Technology (UIST 2000)*, pages 171–180, 2000.
2. Marc H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *IEEE Workshop on Visual Languages*, pages 4–9, October 1991.
3. Lih-Juan ChanLin. Animation to teach students of different knowledge levels. *Journal of Instructional Psychology*, 25:166–175, 1998.
4. 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.
5. Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. Alice: Lessons learned from building a 3D system for novices. In *Proceedings of the CHI 2000 conference on Human factors in computing systems*, pages 486–493, 2000.
6. 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.
7. S. Feiner, D. Salesin, and T. Banchoff. Dial: A diagramatic animation language. *IEEE Computer Graphics & Applications*, 2:43–54, September 1982.
8. Lance Good and Benjamin B. Bederson. Zoomable user interfaces as a medium for slide show presentations. *Information Visualization*, 1(1):35–49, March 2002.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. O. Park and S. S. Gittelman. Selective use of animation and feedback in computer-based instruction. *Educational Technology Research & Development*, 40(4):27–38, 1992.
16. Ian Parker. Absolute PowerPoint: Can a software package edit our thoughts? *The New Yorker*, 2001.
17. Ken Perlin and David Fox. Pad: An alternative approach to the computer interface. In *Proceedings of SIGGRAPH 93*, 1993.
18. 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.
19. Craig W. Reynolds. Computer animation with scripts and actors. In *Proc. SIGGRAPH 82*, pages 289–296, July 1982.
20. 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.
21. Barbara Tversky, Julie Bauer Morrison, and Mireille Betrancount. Animation: Can it facilitate? *International Journal of Human Computer Studies*, 57(4):247–262, October 2002.



(a) (b) (c) (d)

Figure 7 *In sequence (a) camera zooming is used to focus on one region, then a plot is constructed by animating pixels from the input image. In (b), animation is used to maintain continuity as a simple overview is expanded to show more information. Presentation (c) uses zooming in on parts of diagrams to reflect the hierarchical structure of the talk. Sequence (d) shows the use of smoothly animated zooming to join together the actions at multiple scales.*

(a) (b)

(c)

Figure 8 *Part (a) shows three screenshots from an interactive application for generating animated slideshows. The resulting* SLITHY *animation appears in part (c). Part (b) shows a similar prototype utility for generating animated line chart sequences.*