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

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.

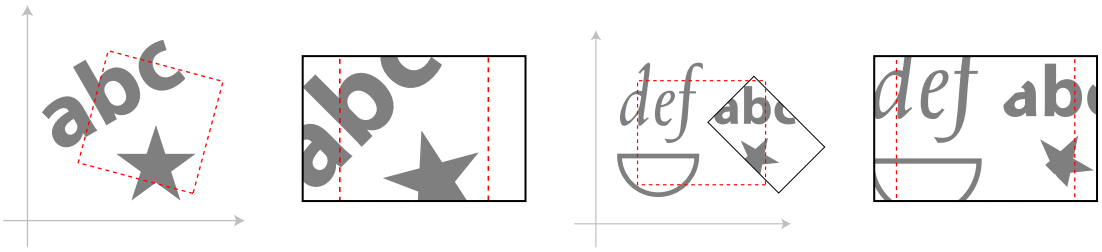


Figure 3.1 *A simple drawing object canvas (part (a)) contains some text and a star shape. The camera rectangle, shown in red, is specified in canvas coordinates. When the drawing object is mapped into a viewport (part (b)) with a different aspect ratio, additional parts of the canvas outside the camera 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.

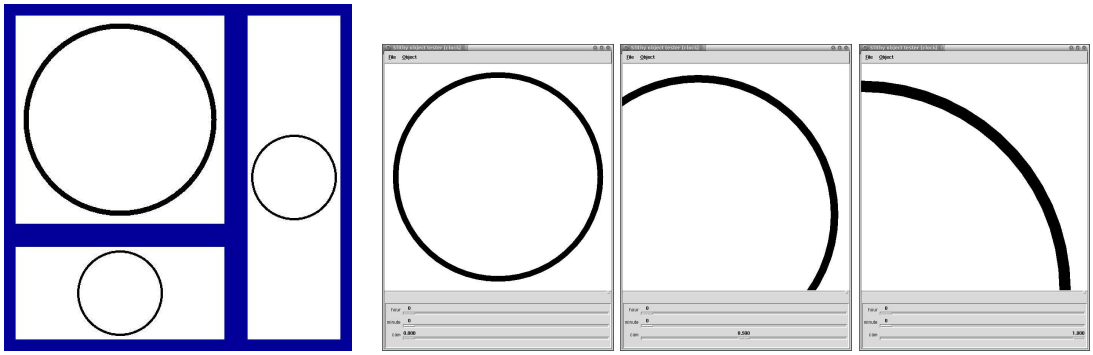
The first item of each parameter's description tuple is its type. The values `SCALAR` and `STRING` are constants defined in the SLITHY library. Table A.1 contains a complete listing of the available types. The type information is used by the tester application to provide appropriate UI controls for interactively 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).

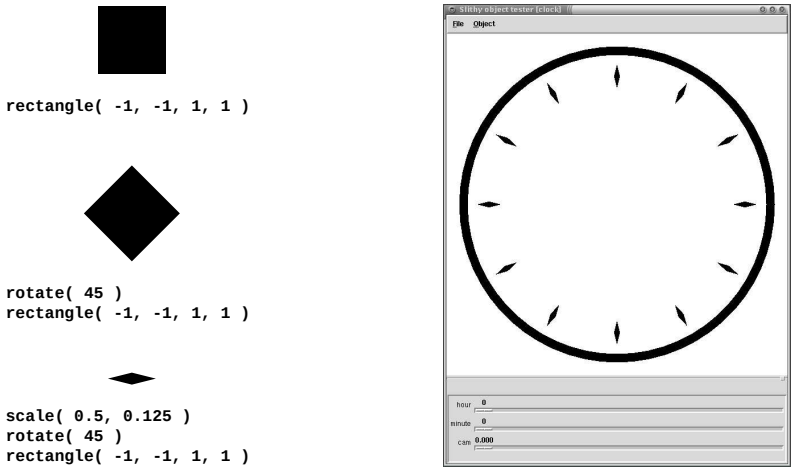


Figure 3.3 *Adding diamond-shaped hour markers to the clock face. Part (a) shows how coordinate system transforms are used to turn a square into a flattened diamond shape. Part (b) shows the clock face after a loop places 12 markers around its circumference.*

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

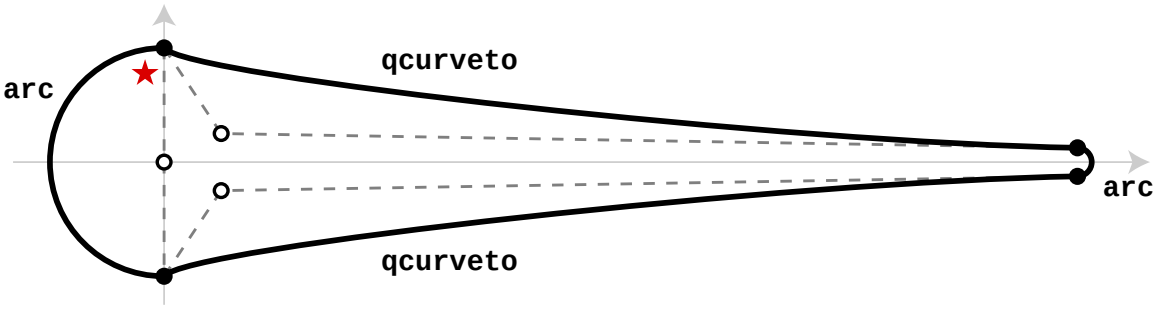


Figure 3.4 *Schematic diagram of the path object used to draw the hands of the clock example. The open circles and dotted lines mark reference points used in the shape's construction, such as the center of circular arcs and the off-curve control points for Bézier curves.*

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.

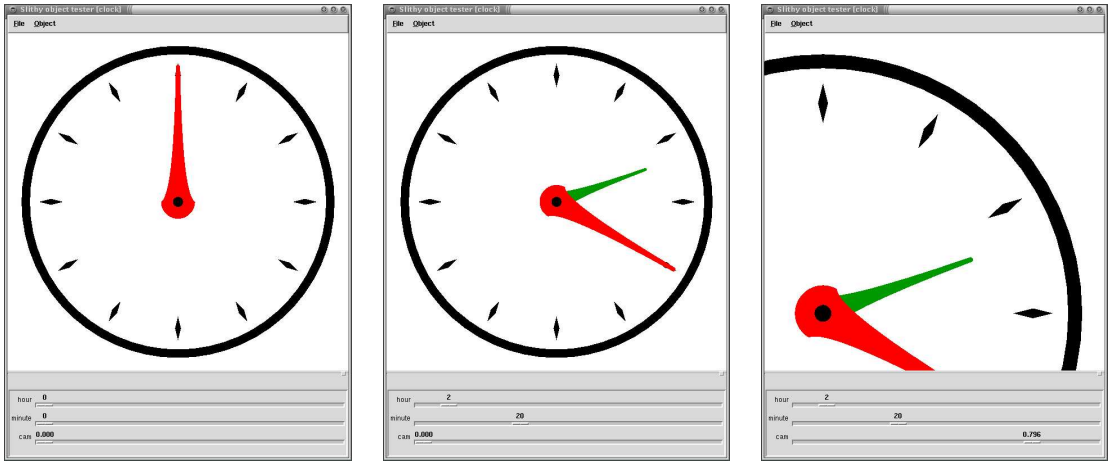
Within the diagram code, we use the `fill()` and `stroke()` functions to draw a path object on the canvas. These draw the path object in the current color, using the current user space transformation. Here we will draw one instance of the `hand` path in red to form the minute hand:

```
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.

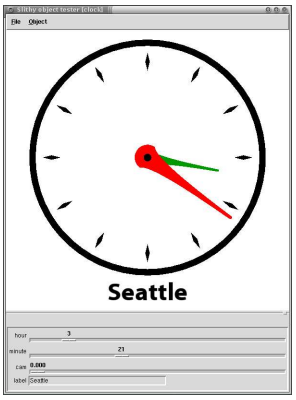
Figure 3.7 shows a complete SLITHY input file defining the final clock diagram.

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 )
```



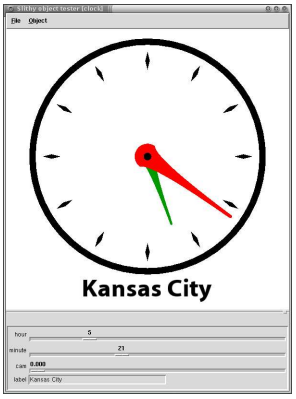


Figure 3.7 *The complete* SLITHY *script for defining a simple clock parameterized diagram, constructed in Section 3.1. Two screenshots of the tester window with various parameter settings are shown on the right.*

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:

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

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

[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.

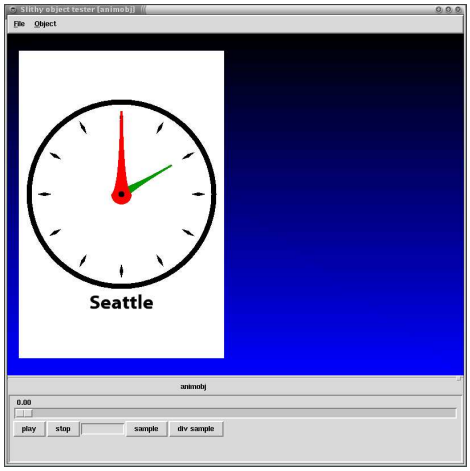


Figure 3.8 *The sole frame of a trivial zero-length animation object.*

```
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 )
```

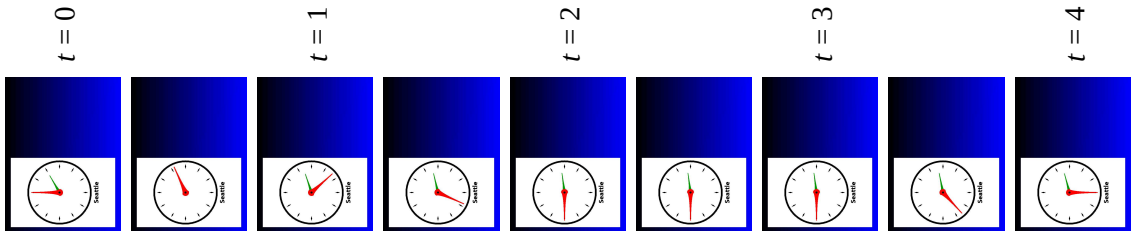


Figure 3.9 *Frames from a simple clock animation (rotated sideways).*

```
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

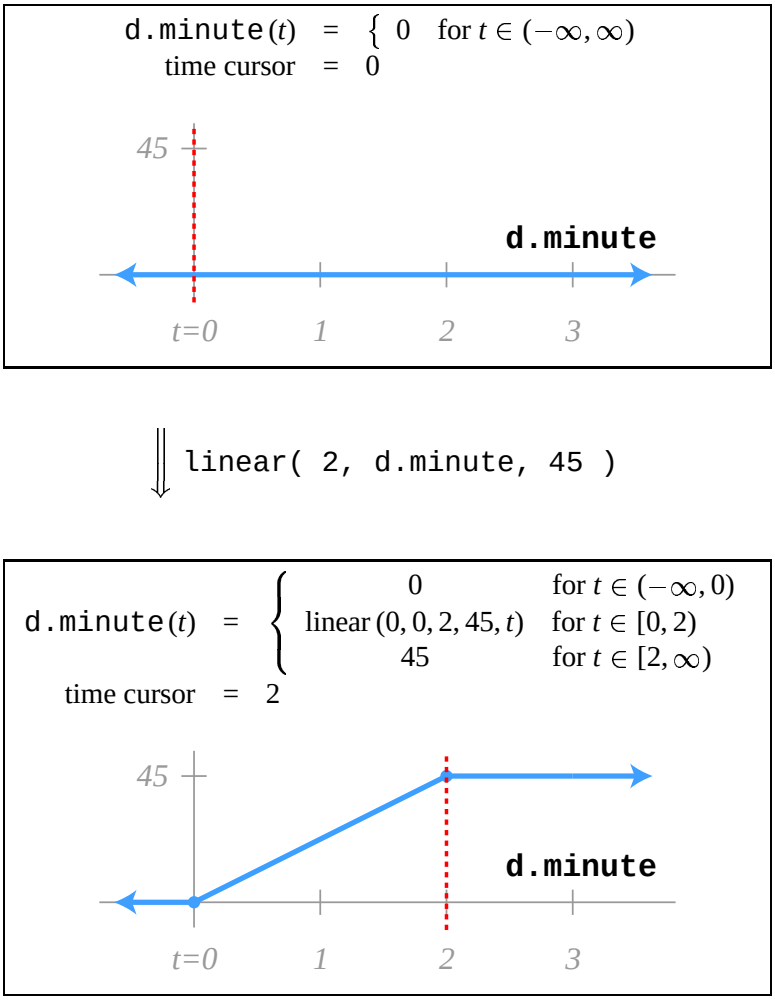


Figure 3.10 *Illustrating how the* `linear` *command operates by overwriting a portion of a parameter's timeline starting at the time cursor.*

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()
```

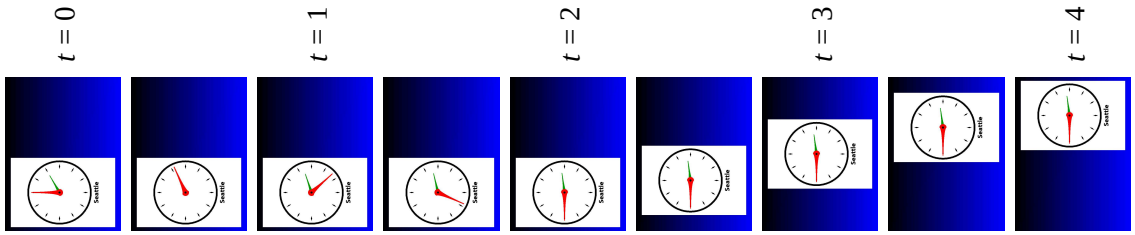


Figure 3.11 *The result of using a pseudoelement to animate an element's position.*

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 )
```

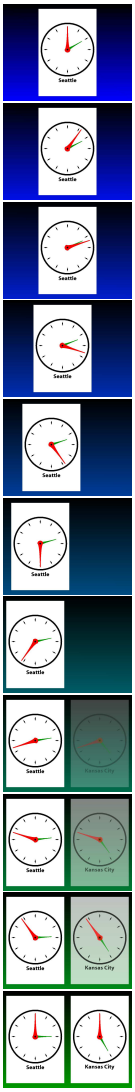


Figure 3.12 *A complete* SLITHY *script for defining a simple clock animation (except for the definition of the clock diagram, which can be found in Figure 3.7).*

```
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

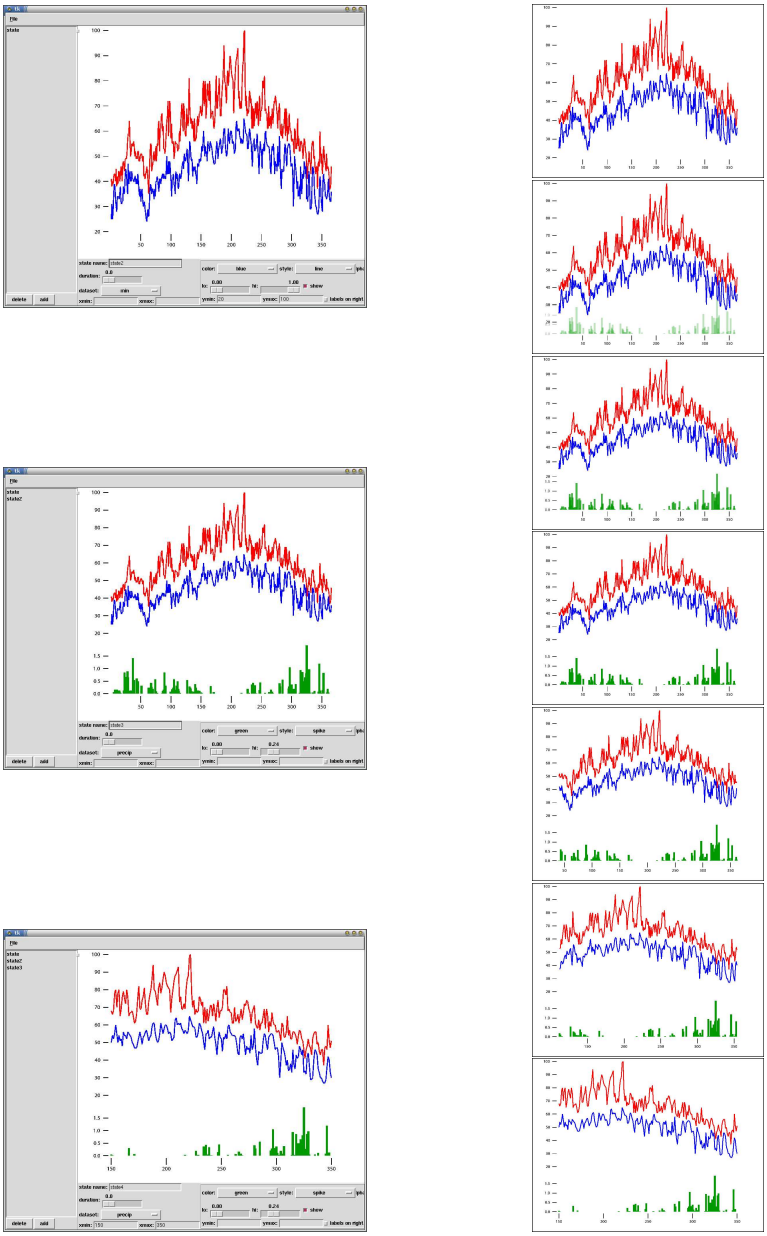


Figure 3.14 *The left column shows screenshots of the prototype GUI line-chart tool. Here the user has interactively arranged the data in three different ways. The right side shows frames from the resulting* SLITHY *animation, which interpolates smoothly between successive states.*

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.

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.

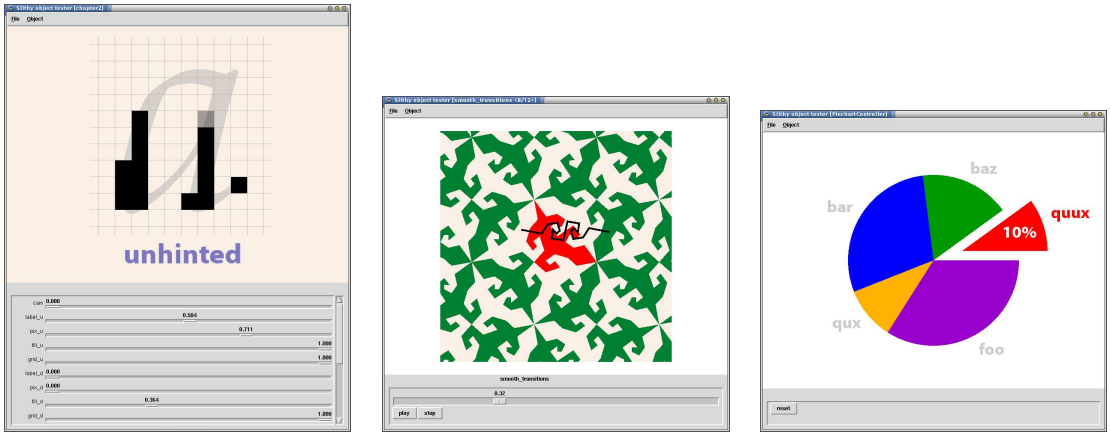
Named arguments. The caller *must* pass a value for this argument, but the value must be preceded by the argument name and an equals sign — e.g., "`foo( argname = value )`".

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.

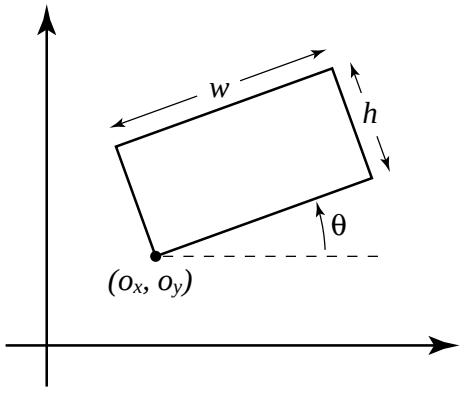


Figure A.2 *The five parameters defining a rectangle in* SLITHY. *Note that the rectangle type stores the aspect ratio w/h instead of the height itself.*

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.

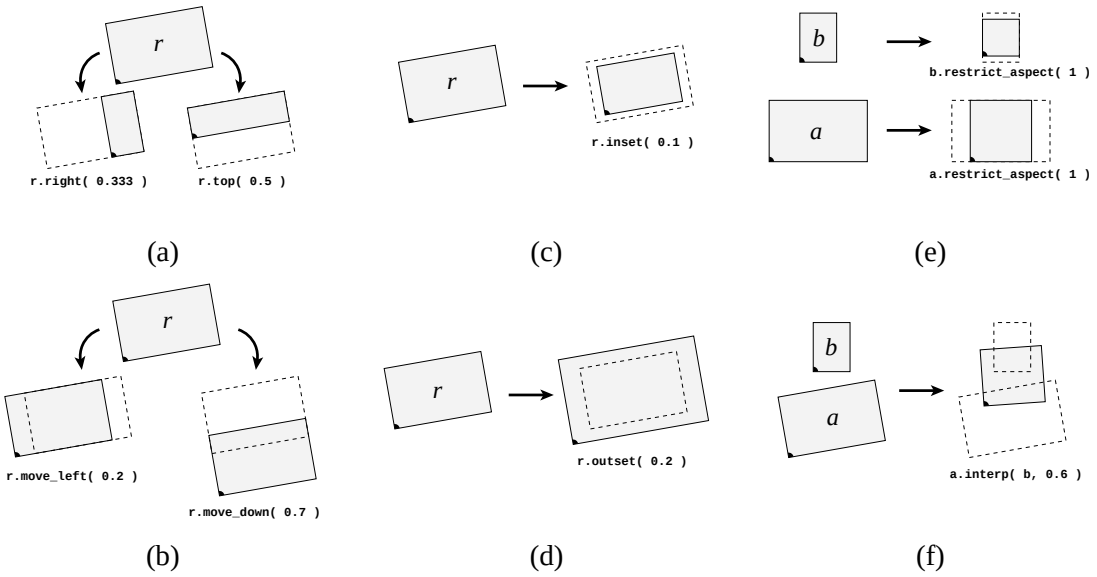


Figure A.3 *Methods that can be applied to* `Rect` *objects to generate new rectangles.*

- **`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(`** *fi lename*`,` *pixels* **`)`**

`search_font()` functions just like `load_font()`, except that it searches for *fi lename* 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(`** *fi lename* **`)`**
▶ **`search_image(`** *fi lename* **`)`**
▶ **`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).

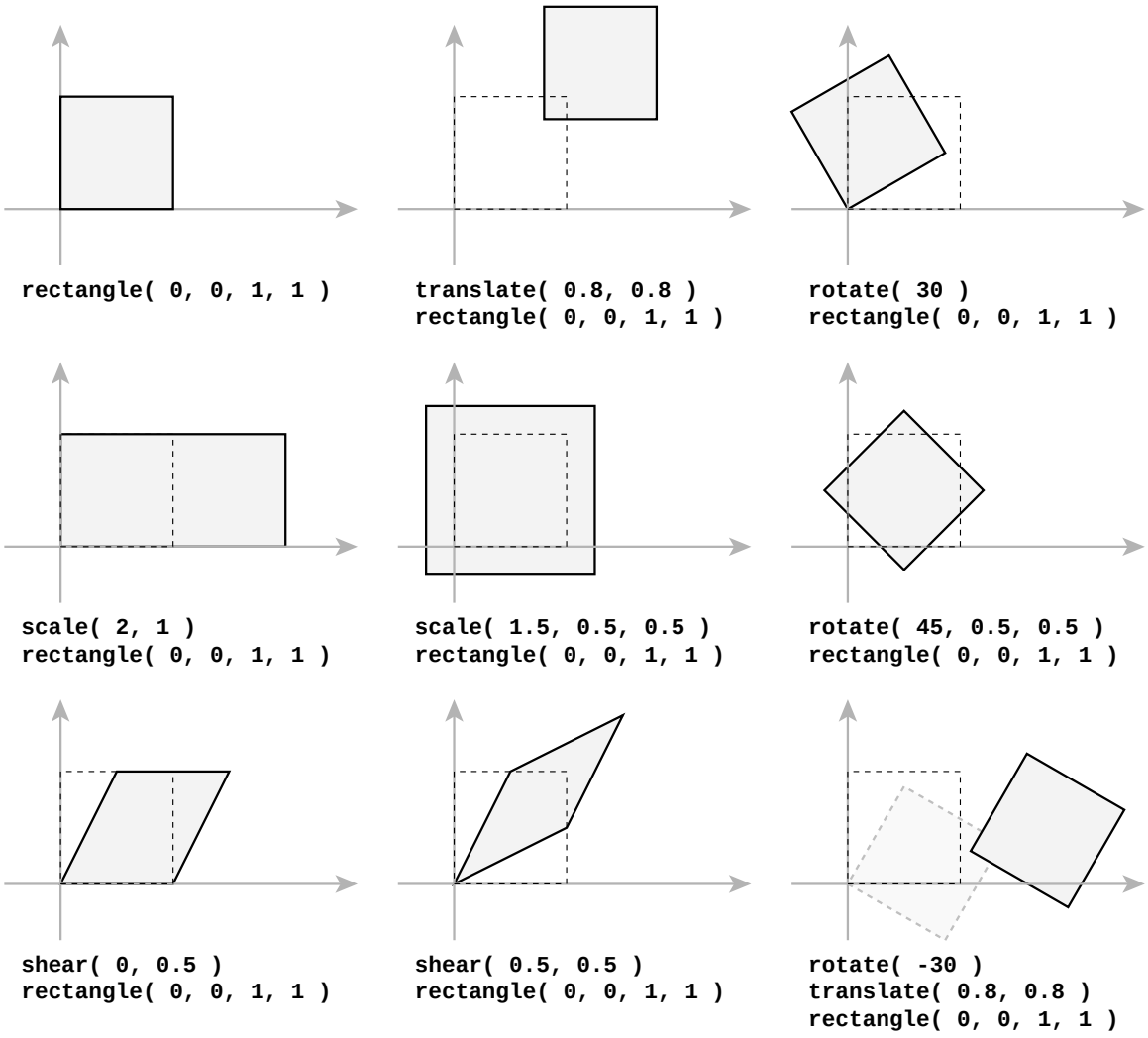


Figure A.4 *Illustrations of various coordinate system transforms applied to a square.*

▶ **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.

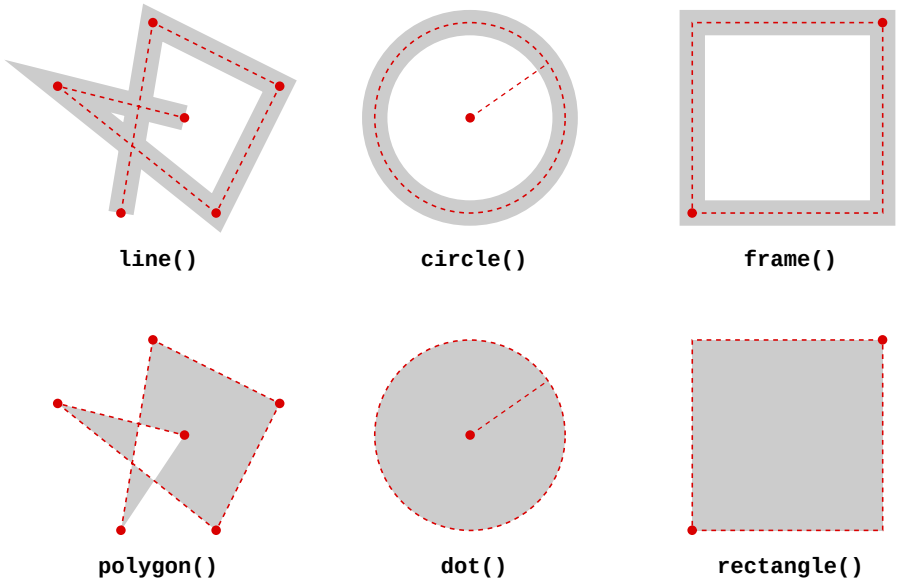


Figure A.5 *Primitive drawing shapes available in* SLITHY. *The red dots indicate the coordinates passed to each function; the gray areas are what is actually drawn.*

- **`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:

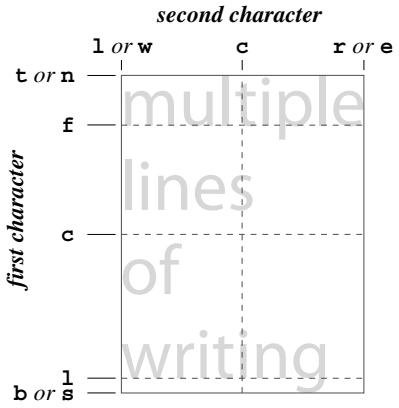
Figure A.6 *The* justify *parameter controls the horizontal positioning of multiline text within the bounding box.*

- *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* **`)`**

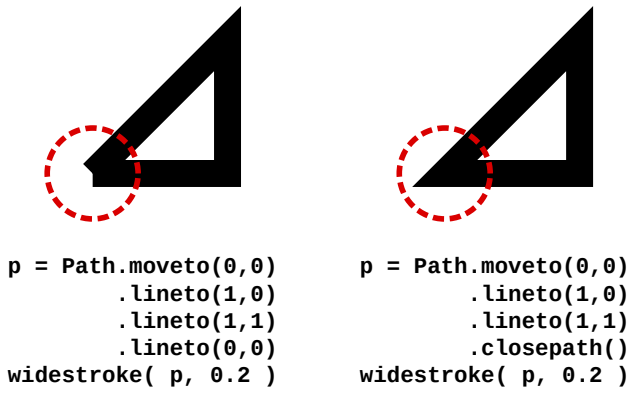


Figure A.8 *The right image shows a path that has been closed with the* `closepath()` *method. The first segment is joined to the last. The left image illustrates a path where a line segment back to the origin has been added, but the path is left open. No join is drawn.*

▶ **`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* **)**

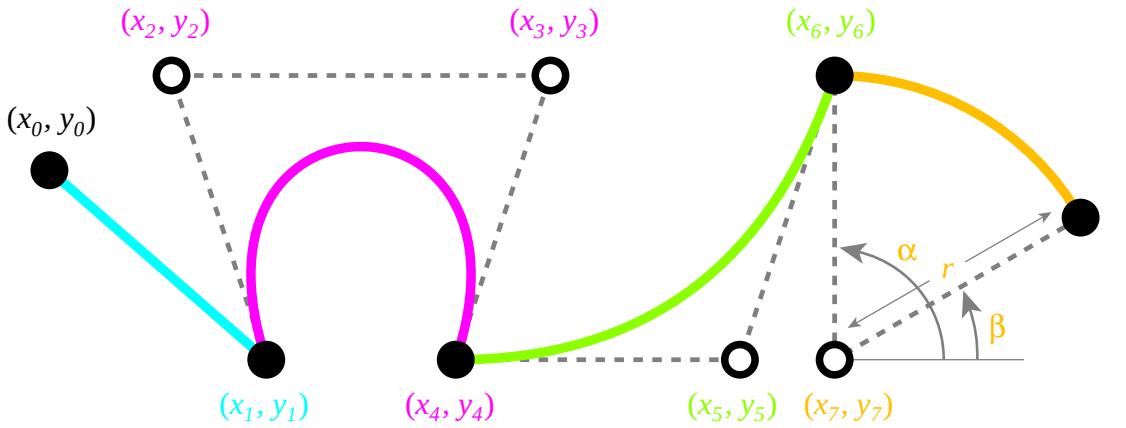


Figure A.9 *Construction of a path object containing line, cubic and quadratic Bézier, and circular arc segments.*

▶ `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:

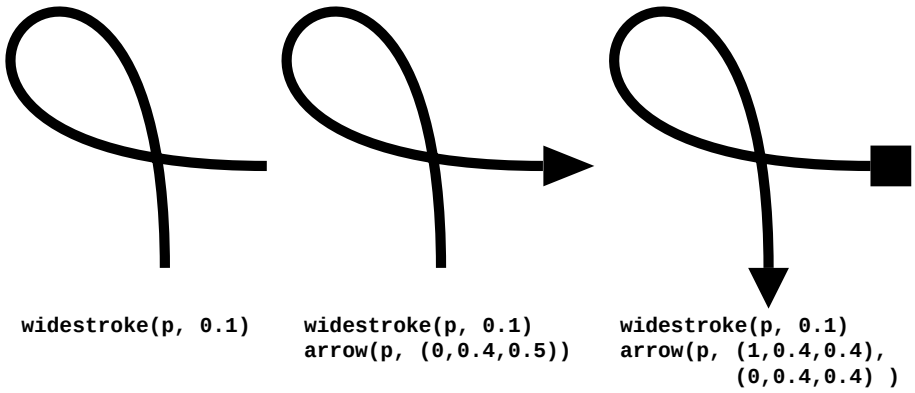


Figure A.10 *The* `arrow()` *function is used to draw arrowheads of various styles at the ends of a path drawn with* `widestroke()`.

```
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, ... )

    ...
```

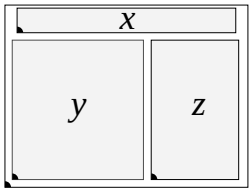


Figure A.11 *Using* `Rect` *object methods to subdivide the area of the slide. The upper right image shows the rectangles x, y, and z in relation to the camera rectangle. The bottom right image is a screenshot of a slide produced with this layout.*

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.

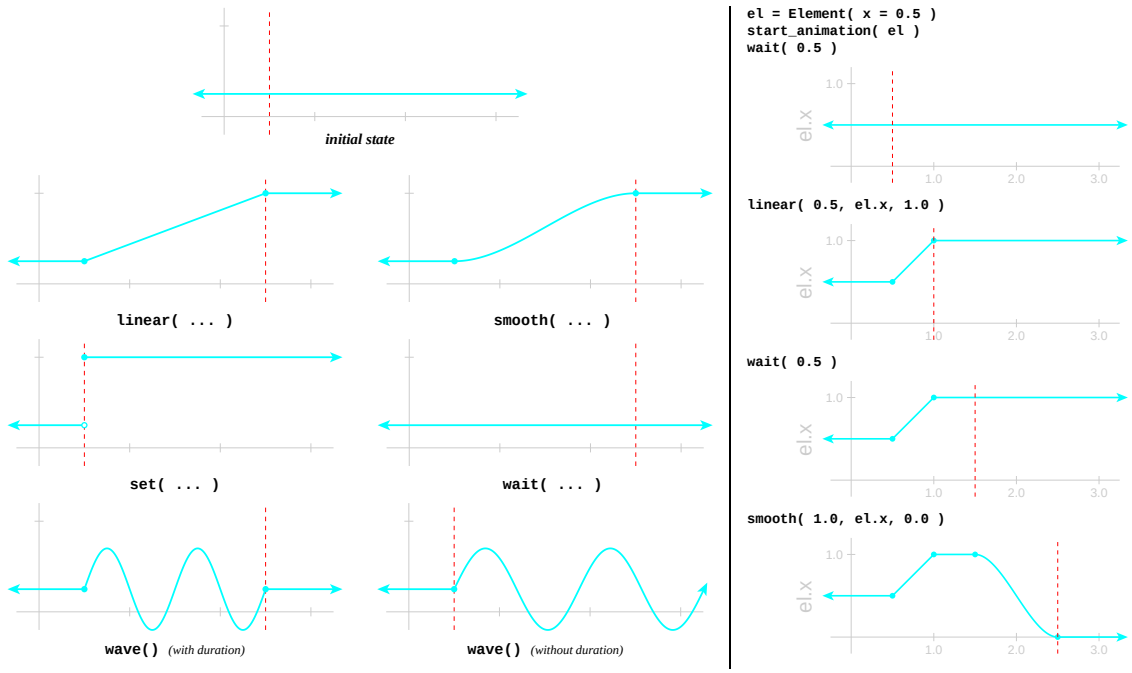


Figure A.12 *Illustration of animation commands applying edits to a parameter timeline. The red dotted line indicates the position of the time cursor. The left side shows the results of six different indiviual commands, each applied to the initial state shown in the top row. The right side shows how 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:

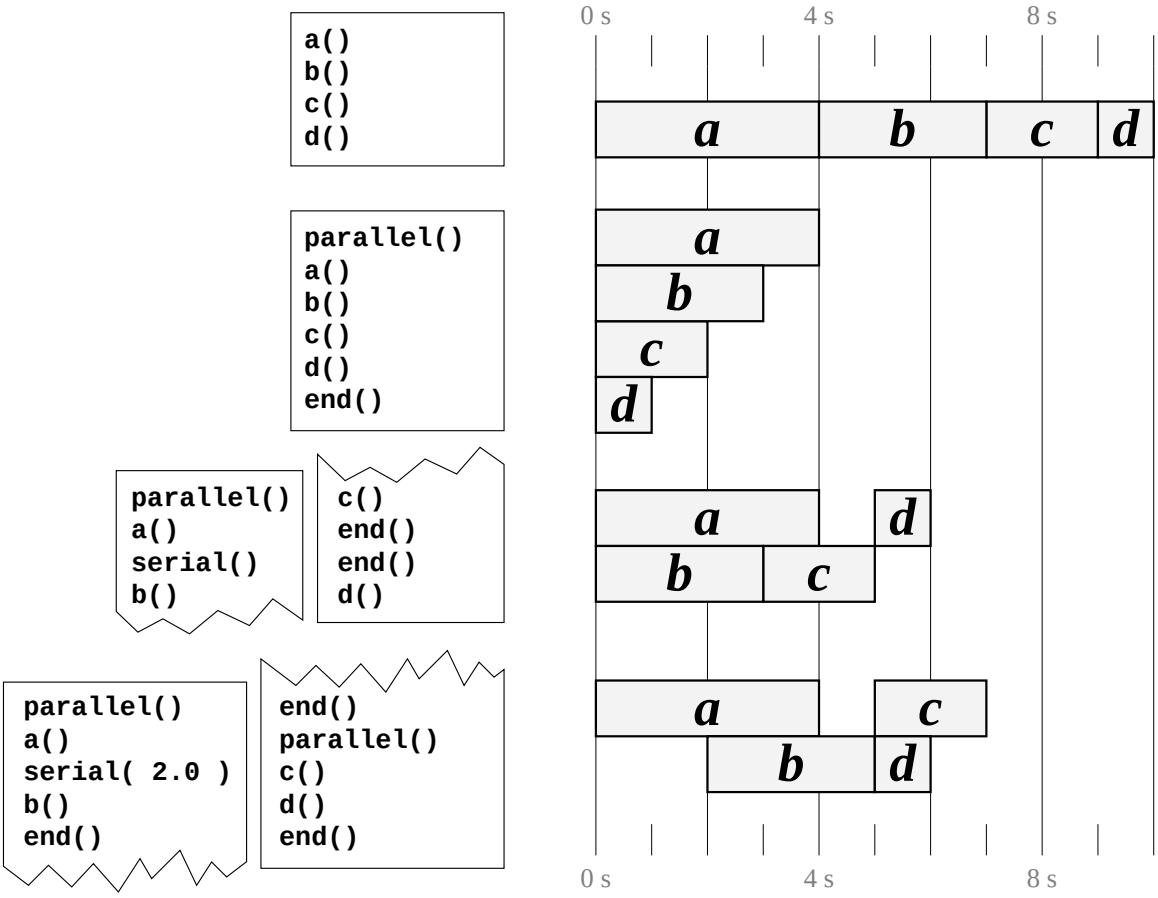


Figure A.13 *Using* `parallel()` *and* `serial()` *to overlap animation functions.* ***a***, ***b***, ***c***, *and* ***d*** *represent animation edits with durations of 4, 3, 2, and 1 seconds, respectively.*

```
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

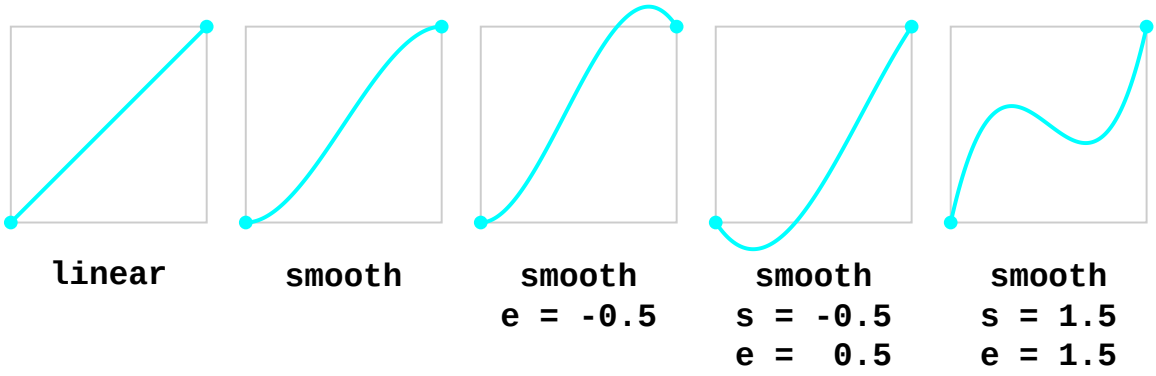


Figure A.14 *Various transition styles available in* SLITHY.

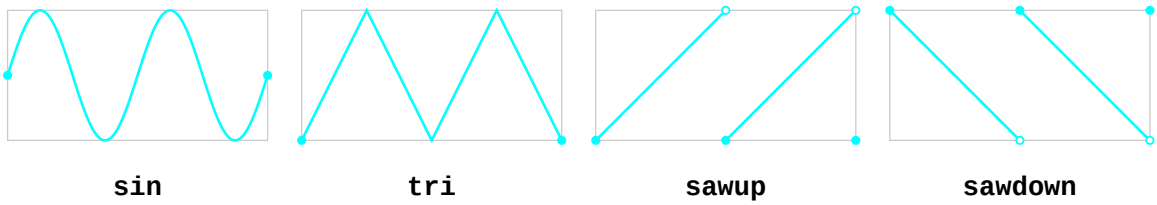


Figure A.15 *Various undulation styles available in* SLITHY. *Two complete cycles of each style are shown.*

`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 `mouse-move()` 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

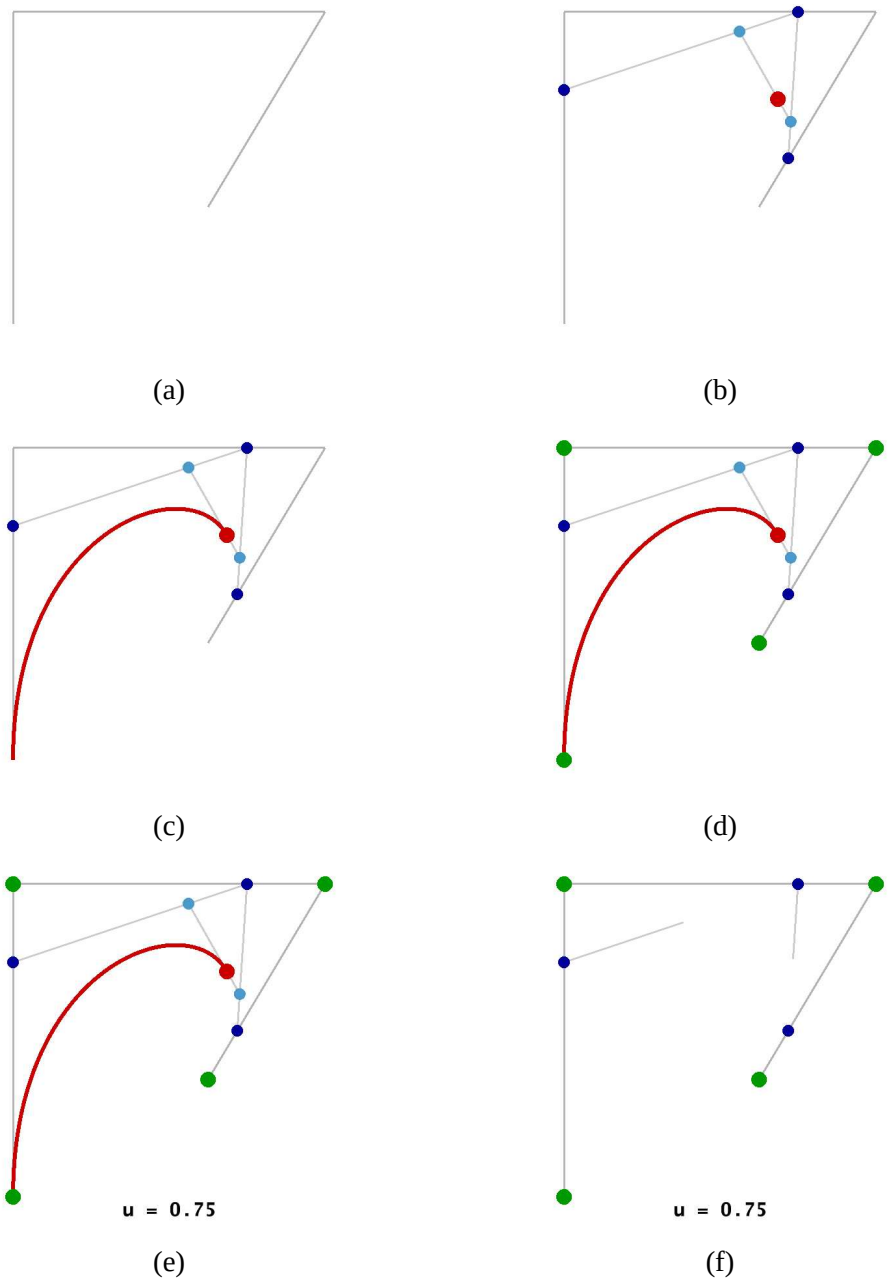


Figure B.1 *Different parts of a diagram for illustrating construction of a Bézier curve. The control polygon (part (a)) is drawn first. Part (b) adds the construction lines and points, and part (c) adds a portion of final curve. The control points are drawn in part (d), and part (e) shows the final 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:

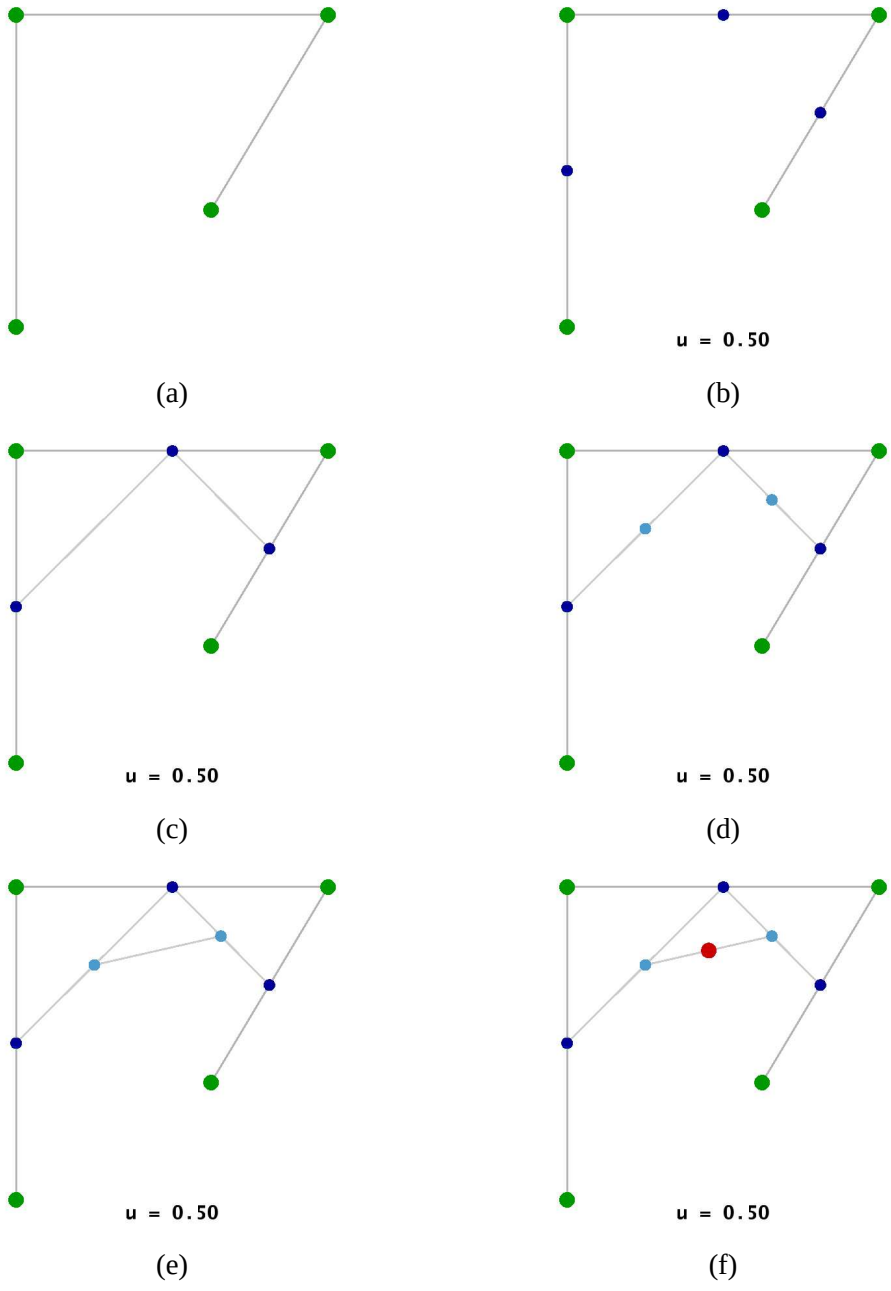


Figure B.2 *For curves with n control points, the applet illustrates the de Casteljau algorithm in* $2n - 3$ *steps. Here the case* $n = 4$ *is shown. Part (a) shows the initial state of the diagram before the subdivision fraction u is specified. Each successive image adds one set of construction points 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 )
```

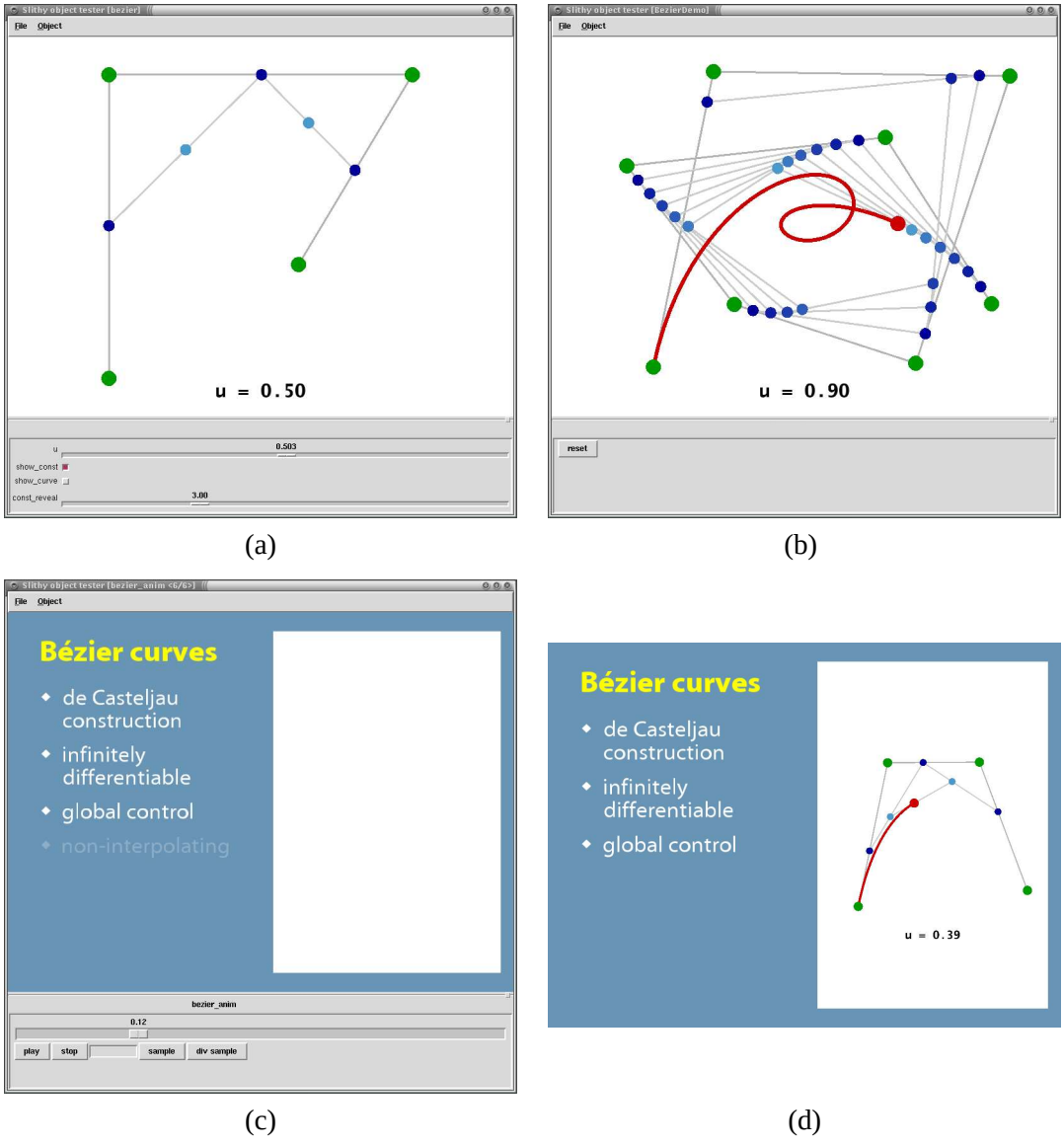
Each time the presenter presses the spacebar, another item is added to the bulleted list using a half-second fade. The interactive object can be manipulated throughout this whole process.

```
    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.



(a) (b)

(c) (d)

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.*