1 Introduction

The *graphix* package allows the Grace programmer to:

- generate basic graphic shapes,
- add event listeners to those shapes, and
- add sounds, using built-in MP3 files

The implementation uses the createjs JavaScript library (http://www.createjs.com/), which draws on an HTML–5 canvas. Thus, it is available only when running Grace in a web browser.

2 Graphics

To work with the *graphix* package, you need to include the following line at the top of your Grace file:

```
import "graphix" as g
```

The graphics object needs to be created with the following command:

```
def graphics = g.create(width, height)
```

where width and height correspond to the desired graphics window width and height. For example:

```
def width = 300
def height = 300
def graphics = g.create(width, height)
```

3 Shapes

The following shape objects are available in *graphix*: Circle, Rectangle, Rounded-rectangle, PolyStar, Ellipse, Arc, Text, Line, Custom Shape, and Input Box. To draw one of these objects to the screen, make a request on the graphics object:

- addCircle
- addRectangle
- addPolyStar
- addRoundRectangle
- addEllipse
- addArc
- addText
- addLine
- addCustomShape
- addInputBox
- addButton

For instance, to add a circle to the window, you might do the following:

```
import "graphix" as g
def graphics = g.create(300, 300)
def circle = graphics.addCircle
circle.draw
```

Each shape has attributes that are used to create it. These attributes have default values, so you don't need to set each one every time you create a shape. Many of the attributes, such as location, are common to all shapes, while others are specific to a particular shape. For example, a circle has a radius, but a rectangle does not. Attributes can be both observed and set, so if you have created a shape myCircle, you can observe its location by requesting myCircle.location, and you can also change its location by requesting myCircle.location := 100@50.

3.1 Common Attributes

The attributes that are common to all types of shape are as follows.

- **location** (Point): The (x, y) coordinates where the shape will be placed in the graphics window. Coordinates are expressed as Grace **Point** objects, such as 10@50. Keep in mind that the origin is in the upper left corner of the graphics canvas, so 10@50 will be 10 pixels right and 50 pixels down from the top-left corner of the canvas.
- **color** (String): The color of the shape. Basic colors can be set using strings such as "red" and "blue". You can also use strings containing 6-digit hex numbers such as "#CC3300" that represent an HTML 5 hex color. See http://www.w3schools.com/tags/ref_colorpicker.asp for more details on colors.
- **fill** (Boolean): Whether or not you want to fill in the shape when it is drawn on the window.
- **visible** (Boolean): Whether or not you want the shape to be visible; it is true by default. Updating this value does not require for the shape to be re-drawn with the draw method.

Chaining Methods

In order to make your code more compact, *graphix* has a number of "chaining" methods that you can use instead of setting attributes individually. The common ones are:

- at(point) — sets the location
- colored(color) — sets the color
- filled(boolean) — true makes the shape filled, false makes it stroked

The chaining methods modify the receiver, but also return it, so that the first request can then be used as the receiver for the second request, and so on. This allows you to construct the object like this:

```
import "graphix" as g
def graphics = g.create(300, 300)
graphics.addCircle.at(100@200).colored("red").filled(true).draw
```

Other Methods

The following methods are also available on all shape objects:

- **moveBy(x, y)**: This moves the shape relative to its current location. Both positive and negative numbers can be used.

- **drawAt(p:Point)**: Moves the shape to the absolute location p, and draws it (making it visible if it wasn't visible).
- **contains(p:Point)**: This determines whether or not p is inside the receiving shape object. Returns true or false.

3.2 Circle

Create: graphics.addCircle

Attributes:

- **radius**(Number): The length of the circle radius

Chaining Methods:

- **setRadius(Number)**

3.3 Rectangle

Create: graphics.addRectangle

Attributes:

- **width**(Number): Width of the rectangle
- **height**(Number): Height of the rectangle
- **size**(Point): Width and height of the rectangle

Chaining Methods:

- **setWidth(Number)**
- **setHeight(Number)**
- **setSize(Point)**

3.4 Rounded Rectangle

Create: graphics.addRoundRect

Attributes:

- **width**(Number): Width of the rectangle
- **height**(Number): Height of the rectangle
- **radius**(Number): Radius of the rounded corners
- **size**(Point): Width and height of the rectangle

Chaining Methods:

- **setWidth(Number)**
- **setHeight(Number)**
- **setRadius(Number)**

- **setSize(Point)**

3.5 PolyStar

Create: graphics.addPolyStar

Attributes:

- **size**(Number): Length of each side of the star
- **sides**(Number): Number of sides
- **pointSize**(Number): Size of the points
- **angle**(Number): Angle between the points

Chaining Methods:

- **setSize(Number)**
- **setSize(Point)** (sets size equal to (Point.x + Point.y) / 2)
- **setSides(Number)**
- **setPointSize(Number)**
- **setAngle(Number)**

3.6 Ellipse

Create: graphics.addEllipse

Attributes:

- **width**(Number): Width of the ellipse
- **height**(Number): Height of the ellipse
- **size**(Point): Width and height of the ellipse

Chaining Methods:

- **setWidth(Number)**
- **setHeight(Number)**
- **setSize(Point)**

3.7 Arc

Create: graphics.addArc

Attributes:

- **radius**(Number): Radius of the arc
- **startAngle**(Number): Starting angle of the arc
- **endAngle**(Number): Ending angle of the arc

- **anticlockwise**(Boolean): Draw the arc anticlockwise if set to true. This is set to false by default.

Chaining Methods:

- **setRadius(Number)**
- **setStartAngle(Number)**
- **setEndAngle(Number)**
- **setAnticlockwise(Boolean)**

3.8 Text

Create: graphics.addText

Attributes:

- **content**(String): The content of the string
- **font**(String): Size and font of the text (eg. "12px Arial")

Chaining Methods:

- **setContent(String)**
- **setFont(String)**

3.9 Line

Create: graphics.addLine

Attributes:

- **start**(Point): Location of the starting point of the line
- **end**(Point): Location of the ending point of the line

Chaining Methods: Attributes:

- **setStart(Point)**
- **setEnd(Point)**

3.10 Custom Shape

This shape consists of a set of points that you add in order to make a custom shape. Instead of configuring preset attributes, you just add points to the shape.

Create: graphics.addCustomShape Methods:

- **addPoint**(Point): Add this point to shape the object

The **addPoint** method returns the object, so that you can chain together **addPoint** requests. For example:

```
import "graphix" as g
def graphics = g.create(300, 300)
graphics.addCustomShape.colored("red").addPoint(40@40).addPoint(0@40).addPoint(40@0).draw
```

3.11 Buttons

A button is a combination of a text object and a shape object, containing all the common attributes of the other shape objects.

By default, the button will be a light grey rectangle with the text "button." To change the text, use the **setText** method. This will change *only* the text itself. If you want to change other text attributes such as the color, size, and font, create a **text** object with the desired attributes, and then set the text object of the button with the **setTextObj** request.

For convenience, several other attributes and methods are available to set the height, width, and color of the button. These affect the default shape (rectangle) of the button. If a custom shape is specified, these settings are ignored.

To change the default shape of the button, you can create a shape object with the desired attributes, and then install the shape object into the button using the **setShape** request.

Some caveats:

- When the default shape object is used, the graphics library centers the text in the button automatically. However, if you change the default shape object, you should also create a custom text object and use the "location" attribute or "at" method on the text object to set its location relative to the reference point of the shape object. Otherwise, the text may not appear centered on the button as desired.
- If you are assigning a click handler to the button and you are using a custom shape, be sure to set **filled** on the shape to **true**. The button click handler works only on the visible parts of the text and shape. Therefore, it will work much better if the shape is filled. Otherwise, you will need to click on the edges of the shape or the actual text, since the empty space between the edge of the shape and the text will not trigger the handler.

Attributes:

- **buttonText**(String): text string used in the button
- **textObj**(Object): Text object created with addText command detailed above
- **buttonShape**(Object): Shape object created with one of the shape commands detailed above
- **color**(String): Set the color of the shape
- **width**(Number): Set the width of the shape
- **height**(Number): Set the height of the shape
- **size**(Point): Set the width and height of the shape

Chaining Methods:

- **setText**(String)
- **setTextObj**(Object)
- **setShape**(Object)
- **colored**(String)
- **setWidth**(Number)
- **setHeight**(Number)
- **setSize**(Point)

```
import "graphix" as g
def graphics = g.create(200, 200)
def easyButton = graphics.addButton.setText("click me").at(10@10).draw

def text = graphics.addText.colored("red").setContent("click me too").at(−30@(−5))
def circle = graphics.addCircle.setRadius(50).colored("lightblue").filled(true)
def fancyButton = graphics.addButton.setShape(circle).setTextObj(text).at(70@70).draw
```

3.12 Input Box

An input box (created with **addInputBox**) allows you to create an input box that can be used to accept strings from the user. There are a number of attributes that can be adjusted with the input box, but, similar to the shapes objects, all attributes have default values and can be considered optional.

Attributes:

- **value**(String): The text in the input box.
- **width**(Number): Width of the input box
- **height**(Number): Height of the input box
- **size**(Point): Width and height of the input box
- **fontSize**(Number): Size of the font
- **fontFamily**(String): Name of font family (e.g. "Arial")
- **fontColor**(String): Color of the font used in input text
- **backgroundColor**(String): Color used in background of input box
- **borderColor**(String): Color used in border of input box
- **onSubmitDo**(Block): Code block to execute when return key is pressed

Chaining Methods:

- **setWidth**(Number)
- **setHeight**(Number)
- **setSize**(Point)
- **setFontSize**(Number)
- **setFontFamily**(String)
- **setFontColor**(String)
- **setBackgroundColor**(String)
- **setBorderColor**(String)

Other Methods:

- **focus**: This will set the curser (focus) on the input box. This is automatically done when an input box is created, but the focus can be on only one input box at a time, so you will need to either create the input box that should have the focus last, or set the focus explicitly after all input boxes have been created.

- **draw**: Just as with the shape objects, the **draw** method must be requested to draw the input box.

4 Drawing a Shape

To draw a shape on the graphics window, first create it, then configure it, and then draw it. The following code creates the output down in Figure 1.

```
import "graphix" as g
def graphics = g.create(200, 200)
def circle = graphics.addCircle
circle.color := "red"
circle.radius := 20
circle.location := 30@30
circle.fill := true
circle.draw
```

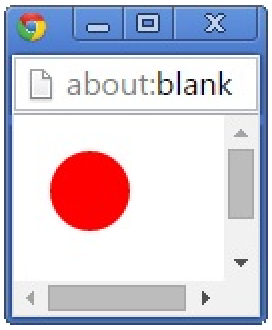

Figure 1: Creating a red circle

5 Adding a Click Handler

Adding a click handler to a shape defines a block of code that will be executed when the shape is clicked. For instance, let's say that we want the red circle to turn blue when it is clicked, and we want to add a message to the user. Then you would add something like this:

```
   import "graphix" as g
 2 def graphics = g.create(200, 200)
   def circle = graphics.addCircle
 4 circle.color := "red"
   circle.onClickDo {
 6     print("clicked circle")
       circle.color := "blue"
 8     circle.draw
   }
10 circle.draw
```

The circle.draw request on line 8 is used to update the circle object after its color has been changed. This is necessary because, although the request circle.color := "blue" updates the state of the circle object, the image on the canvas won't change color until the circle is re-drawn. The request circle.draw makes this happen.

The following requests on shapes can be used to set-up handlers; they all take a block as an argument.

- **onClickDo**
- **onMouseUpDo**
- **onMouseDownDo**
- **onPressMoveDo**
- **onMouseOverDo**

There are also handlers for events on the graphics object itself, rather than on a particular shape:

- **onMouseDownDo**: Triggers when the mouse is pressed on the graphics object
- **onMouseUpDo**: Triggers when the mouse is released on the graphics object
- **onMouseMoveDo**: Triggers when the mouse is moved on the graphics object
- **onMouseExitDo**: Triggers when the mouse is moved from the graphics object

6 Timed Events and Ticker Events

The graphics object, but not the individual shapes, understands requests to set up one-time or repeating timers.

6.1 Timed Events — One-time Timers

- **after(timeDelay) do(block)**: executes **block** once, **timeDelay** milliseconds in the future.
- **clearTimedEvent**: clears any delayed blocks.

Timed events are useful when you want to execute a block of code after a delay of some fixed time. The method after(timeDelay) do (block) can be requested on the graphics object; timeDelay is measured in milliseconds, and is the minimum delay that will be observed; the actual delay may be longer. For example.

```
import "graphix" as g
import "sys" as sys
def graphics = g.create(300, 300)
def circle = graphics.addCircle.at(100@200).colored("red").filled(true)
circle.onClickDo {
    def clickTime = sys.elapsed
    graphics.after 1000 do { print("click delayed for {sys.elapsed - clickTime}") }
}
circle.draw
```

If you have set up multiple timed events and want to remove them all, request the **clearTimedEvent** method on the *graphics* object.

6.2 Ticker Events — Repeating Timers

- **every(interval) do(block)**: executes **block** repeatedly, approximately every **interval** miliseconds.
- **clearTicker**: clears all repeating events

These methods deal with *Tick events*; they are useful if you want to make an animation. The request graphics.every(interval) do(block) establishes a block of code that will execute repeatedly at a rate determined by the interval argument, which specifies the desired interval between frames in millisecond. The request **graphics.clearTicker** will clear all ticker events.

7 Adding sound

The *graphix* module also supports basic sounds. Sounds are preloaded in the browser and cannot be customized at this time. To play a sound, request the play method of the graphics object. For example:

```
import "graphix" as g
def graphics = g.create(200, 200)
def rectangle = graphics.addRectangle.at(125@180).colored("blue").filled(true)
rectangle.onClickDo {
    print("clicked rectangle")
    graphics.play("bicycle_bell")
}
rectangle.draw
```

The following sounds are available: note1, note2, note3, note4, note5, note6, note7, note8, bicycle_bell, snap, whoosh, shutter.