# Glyphic Codeworks the 9/30/94 Release

Glyphic Technology

Saturday, June 18, 2005

Welcome to the 9/30/94 release of Glyphic Codeworks. This release is provided for evaluation and experimentation with the product.

This release incorporates a few changes since the last release, 9/20:

- **Open & Close Events Fixed**
  Codeworks™ now handles Open and Close Apple Events. When you double click on a document to launch the application, it now correctly opens the document. If you simply double click the application, you now get a new document.

- **File Save Bug Fixed**
  Saving documents to volumes other than the system volume failed after the first time without warning. This has been fixed.

- **Drawing**
  Drawing primitives, input handling and drawable views are now in the system. You can now create your own custom views that are drawn with scripts. These views can interact with the user via mouse handling scripts as well. See the new manual section accompanying this release.

In the Documents folder you will find two new documents:

- **Drawing Examples** is full of examples of using each of the new drawing primitives. It also contains object for exploring how to handle mouse interaction and coordinate scaling.

- **Mancalla Game** is an implementation of the pebble counting game from Africa. You play against a computer opponent. (The computer plays very badly... there is a rumor that someone wrote an alpha/beta tree system in Glyphic Script, but it's not ready for prime time!) Note: this document was created before the current drawing system, an so the board drawing & pointer scripts are in its view object. (See the comments in the Drawable Views section of the new manual addendum.)

As always, we are interested in hearing about problems and successes with the system and its documentation. Please tell us about how you've used the system. Feel free to call or e-mail us.

Welcome to Glyphic Codeworks.

—Glyphic Technology

http://www.glyphic.com/

# Canvas

Canvas is the object that you send messages to in order to draw. You can only send these messages in response to the draw or pointer-req messages sent to a drawable view.

The canvas represents the drawable surface of your drawable view. It imposes a coordinate system upon the display. Initially, the point 0,0 is the upper left hand corner and, the coordinates increase down and to the right. The scaling is one unit to one pixel on the screen. You can change the coordinate system to something more convenient, see set-unit-size below.

*[ This drawing system is not totally well-defined at the moment. For example, under Macintosh, lines draw down and to the right, whereas on Windows they are centered. Furthermore, there are some items that will most likely change in the future. See the other bracketed comments. ]*

## Point Arguments

Many messages to canvas take one or more points as arguments. You can specify these points in a number of ways, which ever is most convenient to your program. Here we use the example of draw-rectangle which takes two points (the upper left and lower right corners). Each of these is equivalent:

```
canvas draw-rectangle   10, 20, 80, 90.
          four coordinates

p1 := new point of 10, 20.
p2 := new point of 80, 90.
canvas draw-rectangle   p1, p2.
          two points

a := new array size 4.
a @ 1 := 10.
a @ 2 := 20.
a @ 3 := 80.
a @ 4 := 90.
canvas draw-rectangle   a.
          an array (or group) of coordinates
```

```
g := new group.
g add (new point of 10, 20).
g add (new point of 80, 90).
canvas draw-rectangle   g.
          a group (or array) or points
```

In the descriptions below, generally the coordinate form will be shown, although any form will work. If the message takes other arguments, then the points always come last.

## Basic Drawing

Most of these operations come in 'draw-' and 'fill-' variants. When drawing, the edge of the figure is stroked with a line using the current settings of line-thickness and paint. When filling, the figure is solidly filled in with the paint. Filling does not also draw the line on the edge.

```
canvas draw-line  x1, y1, x2, y2
   draws a line between the two points
```

Draws a line between the two points. Drawing a line from point a to point b is the same as drawing it in the other order.

```
canvas draw-rectangle   x1, y1, x2, y2
   draws a rectangle bounded by the two points
canvas fill-rectangle   x1, y1, x2, y2
   fills a rectangle bounded by the two points
canvas draw-ellipse   x1, y1, x2, y2
   draws an ellipse bounded by the two points
canvas fill-ellipse   x1, y1, x2, y2
   fills an ellipse bounded by the two points
```

These messages draw or fill the indicated graphic figure. Ellipses are defined to fit within the given rectangle.

```
canvas draw-window        draws the display edge
canvas fill-window        fills the display
```

These messages draw or fill entire view, including any borders that set-unit-size may have created.
*[ Window is a poor choice of name for these messages. ]*

» Fill-window is an easy way to clear the view to a color other than white.

```
canvas draw-polygon   g   draws a polygon
canvas fill-polygon   g   fills a polygon
```

These messages draw or fill a polygon. The polygon is usually specified with a group or array of points or coordinates. If there are four or fewer points, the points may be specified directly as arguments. The polygon is always closed, even if the last point doesn't equal the first.

```
canvas draw-bezier   p1, p2, p3, p4 segments  n
   draws a bezier curve from p1 to p4
```

A bezier curve is specified with two endpoints, p1 & p4, and two control points, p2 & p3. You can think of the curve starting out from p1, heading for p2, then curves to end up at p4 as if it was coming from p3. The curve is approximated by line segments. If you specify n, that number of segments is used: less is faster, more is more accurate. If you leave n out, a suitable value will be chosen.

## Drawing State

These two properties of canvas are used by the basic and text drawing messages.

```
canvas.paint        the current paint value
canvas.paint := v   set the current paint value
```

Paint is the gray-scale value to draw with. It is a number between 0 (black) and 100 (white). *[ These numbers are probably backwards from what they should be. ]*

```
canvas.line-thickness           the current line width
canvas.line-thickness := v      set the current line width
```

This is the width of lines. The value is interpreted according to the current coordinate grid.

## Text Drawing

```
canvas draw-text  t, x, y   draw the string t
```

Draws the string t at the indicated point. The point is the left end of the baseline that the text sits on. The text is drawn in the current text size, font, style, as set by the set-font message and in the current color.

```
canvas set-font  size  s
   set just the font size
canvas set-font  size  s group  i italic  bold
   set all of the font characteristics
```

The size s is interpreted in the current units, initially (if you haven't called set-unit-size) it is in pixels. Group, if specified, sets the font as follows:

```
   canvas set-font  group  1  ⇒ a Serif font
   canvas set-font  group  2  ⇒ a Sans-Serif font
   canvas set-font  group  3  ⇒ a Headline font
```

You can specify bold and/or italic versions of the font. You can specify just the size, if you want the other font settings to remain the same.

```
canvas measure-text   t      the width of the string if drawn
```

Returns the width of the string t if it were drawn. The text is measured using the current text size, font, style, as set by the set-font message. The width is returned in the current units. Note that when measuring italic text, the top part of the text might lean further right than the result of the message indicates.

## Units

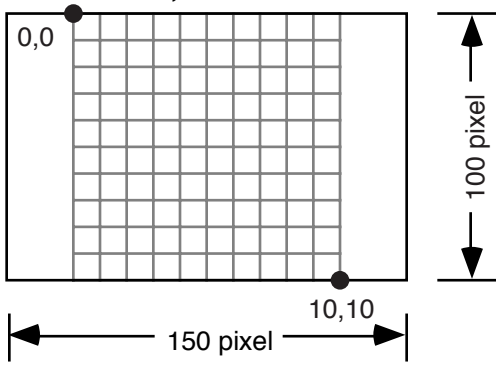
```
canvas set-unit-size
   reset the units to pixels
canvas set-unit-size   x, y
   scale the display to x, y units
canvas set-unit-size   x, y integer-scale
   scale the display, keeping integer ratios
```

These messages change the coordinate system used for drawing. The first message form (no arguments) resets the coordinate system so that the point 0,0 is at the upper left of the view, and each unit is equal to one pixel, increasing down and to the right.

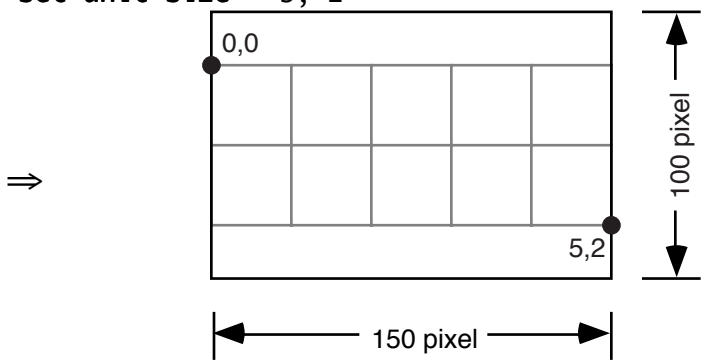
The second and third message forms will create a coordinate system that has x units horizontally and y units vertically. The units will be scaled so that this x by y grid takes up as much of the view as possible. Note that unless the aspect ration of x to y is the same as the view's, there will be border space around the grid. For example, if the view is 100 x 150 pixels:

```
canvas set-unit-size   10, 10
```

⇒



```
canvas set-unit-size   5, 2
```




The scaling affects all future values passed in messages to canvas. In particular, the line-thickness and size argument to set-font are interpreted in these new units. However, the current settings are not affected. For example, assuming the view is 150 by 100 pixels as above:

```
canvas set-unit-size.      reset to pixel coordinates
canvas.line-thickness := 2.   draw 2 pixel wide lines
canvas set-unit-size   10, 10.      set new units
canvas draw-line  0, 5, 10, 5.
   draws a horizontal line, 2 pixels wide.
x := canvas.line-thickness.   ⇒ 0.2
   returns scaled line-thickness
```

The integer-scale flag causes the ratio between pixels and units to be an integer or the inverse of an integer. This has the advantage that drawing maps to the screen pixels cleanly without round-off artifacts. However, to achieve the integer ratio and still fit the coordinate grid on the screen, the grid may be considerably reduced.

```
canvas pixels-per-unit    return scale factor
```

This message returns the number of pixels per unit of the coordinate grid. If the coordinate grid is the default pixel grid, this is 1. If you have changed the coordinate system with set-unit-size, then this is the scaling factor canvas is using to draw with. In the two examples above, the results of this message would be:

```
canvas pixels-per-unit     ⇒ 10
canvas pixels-per-unit     ⇒ 30
```

```
canvas get-bounds          return the view bounds
```

The get-bounds message returns to you a rectangle that this the full boundary of the view. If the coordinate grid is the default pixel grid, then this is the size of the view in pixels. If you have changed the coordinate system with set-unit-size, then this is the bounding rectangle in those units, which may be bigger than what you asked for. In the two examples above, the results of this message would be:

```
canvas get-bounds     ⇒ (-2.5, 0) :: (12.5, 10)
canvas get-bounds     ⇒ (0, -0.667) :: (5, 2.667)
```

## Pointer

```
canvas.pointer             the location of the pointer
canvas.pointer-active      true if the pointer is pressed
```

These messages return information about the pointing device (a mouse, trackball, or pen). The location returned is relative to the current units.

# Drawable Views

You can create drawable views for an object. A drawable view is not composed of other user interface components, but is instead drawn on the screen by a script you write. In addition, you can specify what happens when the user clicks in the view.

To create a drawable for a class when you create the class: set the View Style to "drawable view". If you what to create an new view for an object, follow the procedure "Creating an alternate view" under the section Alternate Views in the Procedures manual. When you choose the form type from a pop-up list, choose "drawable view".

## Drawing the view

When the system needs to redisplay a drawable view, it sends the message 'draw' to your object. In your object's draw script, you can send messages to the canvas object to draw the graphics on the screen. For example:

```
for 5 to 1 by -1 do [ $ index i.
   $ j.
   j := i * 10.
   canvas.paint := 100 - 2 * j + 20.
   canvas fill-rectangle
      60 - j, 60 - j, 100 + j, 60 + j.
].
```

When just before your draw script is called, the view is cleared to white. In addition, the canvas is reset to a normal state. It is as if the following script ran just before you draw script:

```
canvas set-unit-size.       reset coordinates to pixels
canvas.paint := 100.        draw in white
canvas fill-window  .       clear the view to white
canvas.paint := 0.             draw in black
canvas.line-thickness := 1.    draw one pixel wide lines
canvas set-font  size 12 group 1.
   reset font to 12pt, serif font
```

*[ This scheme of drawing can only handle one drawable view per class. Actually, the system first sends the draw message to the view object. Drawable views by default simply pass the draw message onto the class. If you want to have a second drawable view for a class that uses a different draw script, you must override draw in the drawable view object and have it send a different message, not draw, onto the class. Clearly we need to handle this better.*

*In the title-bar menu for a drawable view, in author mode, you will see an entry for Edit Draw Script. This edits the draw script of the view, not the class. Therefore, in general, you should not use it. ]*

## Handling Input

Drawable views can also handle input. When the user activates the pointer (clicks the mouse, taps the pen, etc.) in the view, then the object is sent the message pointer-req. This message passes a number of arguments:

```
o pointer-req  loc p clicks c drag d
```

Loc is the location of the pointer. Clicks indicates the number of clicks. Drag is true if pointer is being dragged. Your object gets sent a pointer-req message for each part of a user operation, with clicks and drag set appropriately. For example, if the user double-clicks, your object will be sent these three messages:

```
o pointer-req  loc p clicks 0. the pointer went down
o pointer-req  loc p clicks 1. the pointer went up
o pointer-req  loc p clicks 2. the pointer clicked again
```

If the user presses the button and then drags, your object will be sent these two messages:

```
o pointer-req  loc p clicks 0. the pointer went down
o pointer-req  loc p clicks 0 drag .
   the pointer started to move while down
```

In response to these messages, you should interpret the users action. You can draw if you want by sending messages to the canvas object. The canvas will be set up, just as it is for drawing, except that the view will not be cleared to white. Using the canvas messages pointer and pointer-active you can track what the user is doing with the pointing device.

The script for pointer-req can return the value true to indicate that no further processing of this user event is needed. For example, if you handle a click request, you could return true to indicate that the system shouldn't look for double or triple clicks.

Here is a sample script that records which part of the view the user clicked in:

```
$ clicks  c, drag .
$ pt.
if c == 0 then  [ return  ].  we aren't interested in down

canvas set-unit-size   3,3.   divide the canvas in thirds
pt := canvas.pointer.         find where the pointer is

   set a property of this object, part,
   to a descriptive string
if pt.y < 1 then  [ part := "top" ].
if (pt.y between  1 and  2) then  [ part := "middle" ].
if pt.y > 2 then  [ part := "bottom" ].

return  true.     don't look for double clicks, etc.
```

Notes:

- This script doesn't use the loc parameter that is passed with pointer-req. This is because that point is in the default coordinate system and this script wants it in the 3x3 coordinate system.

- As written, this script only does something when the user clicks. It is active when the user completes the click by letting up on the pointer. The script works this way because it checks for clicks == 0, meaning the pointer went down, and simply returning, allowing further processing by the system. If you want to change this script to operate on the down, simply remove the first if statement.

*[ Like the draw script, this scheme of handling input can only handle one drawable view per class. Actually, the system first sends the pointer-req message to the view object. Drawable views by default simply pass the pointer-req message onto the class. If you want to have a second drawable view for a class that uses a different pointer-req script, you must override pointer-req in the drawable view object and have it send a different message,*

*not pointer-req, onto the class. Note that pointer-req as sent to the view is sent parameters x and y, not loc. Look for implementors of pointer-req for examples. Clearly we need to handle this better. ]*