Alex,

Thanks so much for sharing this. I have put a few comments throughout the document, as little yellow "post-it notes". Most are suggestions for renaming, simplification, or completeness.

Overall, it's great that you are discussing this interface. I was a bit surprised, though, that there were so few objects in the interfaces; it read more like a functional language library, where everything had to be a top-level function because they don't have objects and methods. I think that Phix can be simplified a lot if you take advantage of objects.

Grace - Phix

Graphics API for the Grace programming lanuage

Last updated on:

21-01-2014

The document doesn't really describe the graphical model, so I had trouble with drawables, windows and components. How are these used by an application? Do I create a canvass and draw on it? Do I create a canvas, and then create a line, and then place the line on the canvas? I can infer that the answer must be the latter, by scanning the canvas methods on page 10, but if you told me this up-front, I wouldn't have to infer anything! OK, so what can I add to a canvas, that is, what are the Drawable objects? I don't see that anywhere.

Finally, I didn't see any text objects. Most GUIs need to put text objects somewhere!

Alex Sandilands

To evaluate Phix effectively, I would need to see some applications written using it. You probably have these, for example, re-writes of some of Kim's ObjectDraw programs. Show the reader some, compare side by side with the old interface, and explain Phix applications are simpler. It's hard for me to see how well an interface works without using it in some way.

Andrew

Contents

1 Summary **3**
1.1 Component Hierarchy . . . 3

2 Graphics **4**
2.1 Interface . . . 4

3 Window **8**
3.1 Interface . . . 8

4 Canvas **10**
4.1 Interface . . . 10

5 Drawable **14**
5.1 Interface . . . 14

6 Components **19**
6.1 Interface . . . 19

7 Utilities **21**
7.1 GMath . . . 21
7.2 Color . . . 22
7.3 Vector2 . . . 24

1 Summary

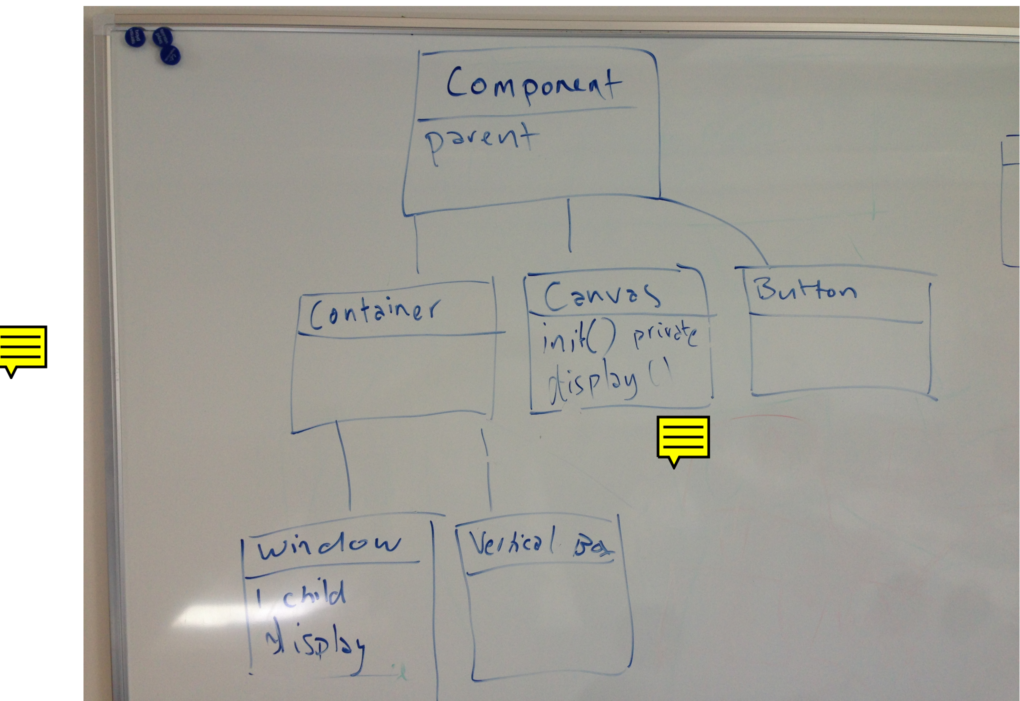
This is a very simple graphics api, designed to be flexible with many different backends. Currently, the GTK+3 backend is nearing completion. Animations are going to be added very soon then it will be finished.

We are also currently working on a Javascript backend, so the graphics library can be run on the web IDE as well. The idea is that if someone wrote a program using the GTK backend library, their code could also work on the web IDE without changing anything.

The library has been designed so the backend can be very easily changed, so if other graphics sources are ported into Grace they can be used with Grace-Phix.

1.1 Component Hierarchy

This is a very basic draft of the hierarchy of how the graphical components work. This was made quite a while ago so it is a little outdated, but the general idea still holds. A nicer diagram will be made soon.



2 Graphics

This module is a list of object constructors for all of the graphical components this library uses. It also has constructors for the drawable shapes, which can be used and combined to create new drawable objects that can be added and drawn on a canvas. It is split up into sections for readability, based on which type of objects the methods are creating.
This is the main module that will be imported into a program when using the graphics library.

2.1 Interface

Listing 1 shows the interface for Grace-Phix graphics.

Listing 1: Grace-Phix graphics interface

```
// ------------------------ //
//                          //
//      WINDOW METHODS      //
//                          //
// ------------------------ //

// Default window, sized 640x480
method createWindow -> Window

// Creates and returns a default window at the position passed in
method createWindowAt(x : Number, y : Number) -> Window

// Creates and returns a window at the specified position with the
// specified size
method createWindowAt(x : Number, y : Number)
        withSize(w : Number, h : Number) -> Window

// Creates and returns a window with the specified size
method createWindowWithSize(w : Number, h : Number) -> Window

// Creates a window with name, position and size specified
method createWindowCalled(t : String)
                at(x : Number, y : Number)
          withSize(w : Number, h : Number) -> Window



// ------------------------ //
//                          //
//      CANVAS METHODS      //
//                          //
// ------------------------ //

// Default canvas sized 640x480
method createCanvas -> Canvas

// Canvas sized wxh
method createCanvasWithSize(w : Number, h : Number) -> Canvas
```

```
// ------------------------ //
//                          //
//    COMPONENT METHODS     //
//                          //
// ------------------------ //


// BOXES

method createVerticalBox -> Container

// A vertical box containing the list of components l
method createVerticalBoxContaining(l : List<Component>) -> Container

method createHorizontalBox -> Container

// A horizontal box containing the list of components l
method createHorizontalBoxContaining(l : List<Component>) -> Container


// BUTTONS

// Default button, no label, no actions
method createButton -> Button

// Button with a label
method createButtonCalled(s : String) -> Button

// Button with a label and an action on clicked
method createButtonCalled(s : String) onClicked(b : Block) -> Button


// ------------------------ //
//                          //
//     DRAWABLE METHODS     //
//                          //
// ------------------------ //

// RECTANGLES


// Rectangle. Has default values: Black, at (25, 25), sized 50x50
method createRectangle -> Rectangle

// Rectangle at (x, y) sized wxh colored c
method createRectangleAt(x : Number, y : Number)
          sized(w : Number, h : Number)
         colored(c : Color) -> Rectangle
```

```
// CIRCLES

// Circle. Has default values: Black, around (50, 50), radius 25
method createCircle -> Circle

// Circle around (x, y) radius r colored c
method createCircleAround(x : Number, y : Number)
                  radius(r : Number)
                 colored(c : Color) -> Circle

// OVALS

// Oval. Has default values: Black, around (50, 50) width 50 height 25
method createOval -> Oval

// Oval around (x, y) size wxh colored c
method createOvalAt(x : Number, y : Number)
             sized(w : Number, h : Number)
           colored(c : Color) -> Oval

// SECTORS

// Sector. Has default values: Black, around (50, 50), radius 25, from 0 to pi
method createSector -> Sector

// Sector around (x, y) radius r from f to t colored c
method createSectorAround(x : Number, y : Number)
                    from(f : Number)
                      to(t : Number)
                  radius(r : Number)
                 colored(c : Color) -> Sector

// ARCS

// Arc. Has default values: Black, around (50, 50), radius 25, width 10 from 0 to 2pi
method createArc -> Arc

// Arc around (x, y) radius r width w from f to t colored c
method createArcAround(x : Number, y : Number)
                 from(f : Number)
                   to(t : Number)
               radius(r : Number)
                width(w : Number)
              colored(c : Color) -> Arc
```

Graphics continued on next page. . .

```
// LINE

// Line. Has default values: Black, from (25, 25) to (75, 75)
method createLine −> Line

// Line from (x1, y1) to (x2, y2) colored b
method createLineFrom(x1 : Number, y1 : Number)
                    to(x2 : Number, y2 : Number)
               colored(c : Color) −> Line

// Line at (x, y) with length l and anti−clockwise angle a in radians
method createLineAt(x : Number, y : Number)
              length(l : Number)
               angle(a : Number)
             colored(c : Color) −> Line

// TEXT

// Text. Has default values: Black, at (45, 48) saying "Text"
method createText −> Text

// Text saying t at (x, y) colored c
method createTextSaying(t : String)
                      at(x : Number, y : Number)
                 colored(c : Color) −> Text


// IMAGE

// Image at (x, y) sized wxh from file: path
method createImageAt(x : Number, y : Number)
                sized(w : Number, h : Number)
                 from(path : String) −> Image
```

3 Window

This component is a top level graphical frame, which is required for anything that needs to be displayed. It is also of type container as it can contain other components which get displayed in the window. Whenever you add a new component to the window it gets displayed at the bottom, as the window uses a vertical box. You can add other boxes to the window if you want components to be displayed in different directions and orders.

3.1 Interface

Listing 2 shows the interface for a Grace-Phix window.

Listing 2: Grace-Phix window interface

```
type Window = Container & {

    // Displays the window and all components in the window
    // then starts the main loop.
    display -> Done

    // Return the title of this window
    title -> String

    // Sets the title of this window
    title:= (t : String) -> Done

    // Return the size of this window with a vector
    // First component is width, second is height
    size -> Vector2

    // Set the size of this window with a vector
    // First component is width, second is height
    size:= (s : Vector2) -> Done

    // Return the width of this window
    width -> Number

    // Set the width of this window
    width:= (w' : Number) -> Done

    // Return the height of this window
    height -> Number

    // Set the height of this window
    height:= (h' : Number) -> Done

    // Return the position of this window with a vector
    // First component is width, second is height
    position -> Vector2

    // Set the posistion of this window with a vector
    // First component is width, second is height
    position:= (p : Vector2) -> Done
```

```
    // Set what happens when the mouse is pressed in this window
    mousePressed:= (b : Block) -> Done

    // Set what happens when the mouse is released in this window
    mouseReleased:= (b : Block) -> Done

    // Set what happens when the mouse is dragged in this window
    mouseDragged:= (b : Block) -> Done

    // Sets what happens when the key is pressed
    onKey(key : String) do(b : Block) -> Done

    asString -> String
}
```

4 Canvas

This component is a display buffer. It contains a list of drawable objects which get painted to the buffer and can be manipulated. Dispite appearing to have similar methods as a container, it is not as it doesn't contain components, only drawables. The canvas can have external drawables objects added to it, or it can create it's own basic drawables directly.

4.1 Interface

Listing 3 shows the interface for a Grace-Phix canvas.

Listing 3: Grace-Phix canvas interface

```
type Canvas = Component & {

    // Adds a drawable object d to the canvas.
    // The same drawable object cannot be added more than once, so clone it before adding a new one.
    // A set could work here, but it would make the ordering methods much more complicated.
    // Lists are also very readable
    add(d : Drawable) -> Boolean

    // Adds the list l of drawables to the canvas.
    // If one of the drawables is already on the canvas then the method
    // will break and return false, with the rest of the drawables in l
    // not being added.
    addAll(l : List<Drawable>) -> Boolean

    // Remove the drawable d from the canvas.
    // Returns true if one instance was found and removed, or false otherwise.
    remove(d : Drawable) -> Boolean

    // Removes the drawable at the index ind.
    // Returns false if the index was out of bounds.
    removeWithIndex(ind : Number) -> Boolean

    // Returns the drawable at index ind
    getWithIndex(ind : Number) -> Drawable

    // Returns the index of the top drawable that contains (x, y) or
    // 0 if none are found. Note that it searchs from the end of the list
    // to the beggining of the list, as when painting the canvas paints from
    // the start of the list.
    findDrawableAt(x : Number, y : Number) -> Number

    // Sends the drawable d to the back of the display buffer
    sendToBack(d : Drawable) -> Done

    // Sends the drawable at index ind to the back of the display buffer
    sendIndexToBack(ind : Number) -> Done

    // Brings the drawable d to the front of the display buffer
    bringToFront(d : Drawable) -> Done
```

Canvas continued on next page. . .

```
// Brings the drawable at the index ind to the front of the display buffer
bringIndexToFront(ind : Number) -> Done


// Sends the drawable d back one position in the draw order
sendBack(d : Drawable) -> Done

// Sends the drawable at index ind back one position in the draw order
sendIndexBack(ind : Number) -> Done

// Brings the drawable d forward one position in the draw order
bringForward(d : Drawable) -> Done

// Brings the drawable at index ind forward one position in the draw order
bringIndexForward(ind : Number) -> Done

// Gets the size of this canvas as a 2d Vector
// First component is width, second is height
size -> Vector2

// Sets the size of this canvas with a 2d Vector s'
// First component is width, second is height
size:= (s : Vector2) -> Done

// Returns the width of this canvas
width -> Number

// Set the width of this canvas
width:= (w' : Number) -> Done

// Returns the height of this canvas
height -> Number

// Set the height of this canvas
height:= (h' : Number) -> Done

// Set whether this canvas can actually paint things. True by default
setPaintable(b : Boolean) -> Done

// Asks the canvas to repaint
paint -> Done

// Returns a string with infomation about this canvas
asString -> String
```

Canvas continued on next page...

```
// ------------------
//  DRAWNING METHODS
// ------------------

// The color that shapes will be drawn with when using the canvas draw methods
color -> col.Color

// Defines whether or not the shapes drawn by the canvas will be filled
fill  -> Boolean

// Sets what happens when the mouse is pressed inside the canvas
mousePressed:=  (b : Block) -> Done

// Sets what happens when the mouse is released inside the canvas
mouseReleased:= (b : Block) -> Done

// Sets what happens when the mouse is clicked inside the canvas
mouseClicked:=  (b : Block) -> Done

// Rectangle at (x, y) sized wxh colored c
drawRectangleAt(x : Number, y : Number) sized(w : Number, h : Number) -> Drawable

// Circle around (x, y) radius r colored c
drawCircleAround(x : Number, y : Number) radius(r : Number) -> Drawable

// Oval around (x, y) size wxh colored c
drawOvalAt(x : Number, y : Number) sized(w : Number, h : Number) -> Drawable

// Sector around (x, y) radius r from f to t colored c
drawSectorAround(x : Number, y : Number)
            from(f : Number)
              to(t : Number)
          radius(r : Number) -> Drawable


// Arc around (x, y) radius r width w from f to t colored c
drawArcAround(x : Number, y : Number)
         from(f : Number)
           to(t : Number)
       radius(r : Number)
        width(w : Number) -> Drawable


// Line from (x1, y1) to (x2, y2) colored b
drawLineFrom(x1 : Number, y1 : Number) to(x2 : Number, y2 : Number) -> Drawable

// Line at (x, y) with length l and anti-clockwise angle a in radians
drawLineAt(x : Number, y : Number) length(l : Number) angle(a : Number) -> Drawable

// Text saying t at (x, y) colored c
drawTextSaying(t : String) at(x : Number, y : Number) -> Drawable
```

```
// Image at (x, y) sized wxh from file: path
drawImageAt(x : Number, y : Number)
    sized(w : Number, h : Number)
     from(path : String) -> Drawable

}
```

5 Drawable

This module contains all of the basic objects that can be painted on a canvas. The canvas can paint any object that is of type Drawable. You can create your own object that is made up of lots of these basic drawable types, make sure it is of type Drawable, then simply add it to the canvas to be drawn.

5.1 Interface

Listing 4 shows the interface for a Grace-Phix drawable.

Listing 4: Grace-Phix drawable interface

```
// Abstract super class for drawable objects
type Drawable = {

  // Cartesian coordinates for the location on the 2d plane
  location -> Vector2

  // If true, this drawable will be drawn
  visible -> Boolean

  // Returns the x cartesian coordinate
  x -> Number

  // Returns the y cartesian coordinate
  y -> Number

  // Sets the x cartesian coordinate
  x := (x' : Number) -> Done

  // Sets the y cartesian coordinate
  y := (y' : Number) -> Done

  // Sets the x,y cartesian coordinate with a vector
  moveTo := (l : Vector2) -> Done

  // Moves the object by dx in the x direction, dy in the y direction
  moveBy(dx : Number, dy : Number) -> Done

  // Paints this object to a canvas, using the graphical object gfx
  // of that canvas
  draw(gfx) -> Done

  // Checks if (x, y) is inside the drawable
  contains(x : Number, y : Number) -> Boolean

  // Returns an identical object
  clone -> Drawable
}
```

```
// Drawable rectangle type
type Rectangle = Drawable & {

    // Vector holding the width and height of this rectangle
    // First component is width, second is height
    size  -> Vector2

    // The color of this rectangle
    color -> Color

    // Defines whether this rectangle should filled
    fill  -> Boolean

    // If not filled, this is the width of the outline
    lineWidth -> Number

    // Returns the width of this rectangle
    width -> Number

    // Sets the width of this rectangle
    width := (w' : Number) -> Done

    // Returns the height of this rectangle
    height -> Number

    // Sets the height of this rectangle
    height := (h' : Number) -> Done
}


// Drawable circle type
// This is a sector, but the "from" and "to" values are constants: 0 -> 2pi
type Circle = Drawable & {

    // The radius of this circle
    radius -> Number

    // The color of this circle
    color  -> Color

    // Defines whether this circle should filled
    fill      -> Boolean

    // If not filled, this is the width of the outline
    lineWidth -> Number
}
```

Drawable continued on next page...

```
// Drawable oval type
type oval = Drawable & {

    // Vector holding the width and height of this oval
    // First component is width, second is height
    size -> Vector2

    // The color of this oval
    color  -> Color

    // Defines whether this oval should be filled
    fill     -> Boolean

    // If not filled, this is the width of the outline
    lineWidth -> Number

    // Returns the width of the oval
    width -> Number

    // Sets the width of the oval
    width := (w' : Number) -> Done

    // Returns the height of the oval
    height -> Number

    // Sets the height of the oval
    height := (h' : Number) -> Done
}


// Drawable sector type. Like a circle but you can define where
// (on the complex plane) it starts being drawn, and where it ends
type Sector = Drawable & {

    // The radius of the sector
    radius -> Number

    // The color of this sector
    color  -> Color

    // Defines whether this sector should be filled
    fill     -> Boolean

    // If not filled, this is the width of the line
    lineWidth -> Number

    // Angle the sector starts drawing from (in radians)
    from -> Number

    // Angle the sector stops drawing at (in radians)
    to   -> Number
}
```

Drawable continued on next page. . .

```
// Drawable arc type. This is like a sector except you can define a certain
// ammount of the inside to not be filled in
type Arc = Drawable & {

    // The radius of the arc
    radius -> Number

    // The width of the arc
    width  -> Number

    // The color of the arc
    color  -> Color

    // Defines whether this arc should be filled
    fill      -> Boolean

    // If not filled, this is the width of the outline
    lineWidth -> Number

    // Angle the arc starts drawing from (in radians)
    from   -> Number

    // Angle the arc stops drawing at (in radians)
    to     -> Number
}


// Drawable line type
type Line = Drawable & {

    // The color of the line
    color -> Color

    // The width of the line
    lineWidth -> Number

    // Catesian coordinates of the start of the line
    from -> Vector2

    // Cartesian coordinates of the end of the line
    to   -> Vector2
}
```

```
// Drawable text type
type Text = Drawable & {

    // The color of the text
    color -> Color

    // The string that this text writes
    text  -> String

    // The font size of the text
    size  -> Number
}


// Drawable image type. Takes a string pathway to the image file,
// which must be a png
type Image = Drawable & {

    // Path to the image, relative to the file that is using this object
    filename -> String

    // Returns the width of the image
    width -> Number

    // Sets the width of the image
    width := (w' : Number) -> Done

    // Returns the height of the image
    height -> Number

    // Sets the height of the image
    height := (h' : Number) -> Done
}
```

6 Components

This module has the graphical components that are displayed on the screen. Other than utility objetcs and drawables, all objects in Grace-Phix are components at the top level. For example, a verticale box is a component and a container, so it can contain other components, including other containers. Only minor components are defined in this module, the large ones such as window and canvas have their own modules.

6.1 Interface

Listing 5 shows the interface for Grace-Phix components.

Listing 5: Grace-Phix components interface

```
// Top level type for graphical user interface objects.
type Component = {

    parent -> Component

    // TODO
    // This type also has a getComponent method which returns the
    // actual graphics object this type wraps up. Not sure how to
    // approach this as the type would be different for different
    // backend graphics libraries.
}



// Type for components that can contain other components
type Container = Component & {

    // Adds component c to this container.
    // Returns true if added, false if c was already in the container
    add(c : Component)         -> Boolean

    // Adds every component in the list l to this container.
    // Returns false if one of the components was already in the container
    addAll(l : List<Component>) -> Boolean

    // Attempts to remove the component c from this container
    remove(c : Component)      -> Boolean

    // Returns a list of all the components in this container
    getChildren -> List<Component>
}
```

```
// Type for a button, which is a component that can be clicked
// to perform actions
type Button = Component & {

    // Returns the label on this button
    label -> String

    // Sets the label on this button
    label:= (s : String) -> Done

    // Returns a vector of the width and height of this button
    // First coordinate is width, second is height
    size -> Vector2

    // Sets the size of this button, with a vector
    // First coordinate is width, second is height
    sized:= (s' : Vector2) -> Done

    // Returns the width of this button
    width -> Number

    // Sets the width of this button
    width:= -> Done

    // Returns the height of this button
    height -> Number

    // Sets the height of this button
    height:= -> Done

    // Sets what happens when this button is clicked
    clicked(b : Block)  -> Done

    // Sets what happens when this button is pressed
    pressed(b : Block)  -> Done

    // Sets what happens when this button is released
    released(b : Block) -> Done
}
```

7 Utilities

The following are the utility modules that the Grace-Phix library uses.

7.1 GMath

Note that this module doesn't create an object, so it doesn't have any type definitions. These are all the methods that the math module provides.

Listing 6 shows the interface for GMath.

Listing 6: GMath interface

```
// Constants
def pi : Number is readable      = 3.14159265358979323846
def half_pi : Number is readable = 1.57079632679489661923
def two_pi : Number is readable  = 6.28318530717958647692


//METHODS

// Return the absolute value
method abs(value : Number) -> Number

// Returns the square root of the given value
method sqrt(value : Number) -> Number

// Returns the minimum value of the two given values
method min(value : Number, value' : Number) -> Number

// Returns the maximum value of the two given values
method max(value : Number, value' : Number) -> Number

// Clamps the given value between the upper and lower thresholds
method clamp(value : Number) between(lower : Number, upper : Number) -> Number

// Clamps the given value so that it is not greater then the given threshold
method clamp(value : Number) below(threshold : Number) -> Number

// Clamps the given value so that it is not less than the given threshold
method clamp(value : Number) above(threshold : Number) -> Number

// Calculates the factorial of the given input
method fact(value : Number) -> Number

// Converts degrees to radians
method toRadians(value : Number) -> Number

// Converts radians to degrees
method toDegrees(value : Number) -> Number

// Normalizes radians to be between 0 and 2pi
method normalizeRadians(value : Number) -> Number
```

Utilities continued on next page...

```
// Binomial Coefficient
method n(n : Number) choose(k : Number) -> Number

// Returns the sine of value
method sin(value : Number) -> Number

// Returns the cosine of value
method cos(value : Number) -> Number

// Returns the tangent of value
// Undefined on pi/2 and 3pi/2 so will throw an error on these values
method tan(value : Number) -> Number

// Returns the arcsine of value
method asin(value : Number) -> Number

// Returns the arccosine of value
method acos(value : Number) -> Number

// Returns the arctangent of value
method atan(value : Number) -> Number
```

7.2 Color

This color module provides a color object and many constructor methods.

Listing 7 shows the interface for Color.

Listing 7: Color interface

```
// Object that holds the rgba components of a color.
// Values are between 0 and 1
type Color = {

    // Returns the red component of this color
    r -> Number

    // Returns the green component of this color
    g -> Number

    // Returns the blue component of this color
    b -> Number

    // Returns the alpha component of this color
    a -> Number

    // Sets the red component of this color
    r := (r' : Number) -> Done

    // Sets the green component of this color
    g := (g' : Number) -> Done
```

```
    // Sets the blue component of this color
    b := (b' : Number) -> Done

    // Sets the alpha component of this color
    a := (a' : Number) -> Done


    // Additive binary operator for this color.
    +(other : Color) -> Color

    // Returns a brighter copy of this color, by a factor of 0.7
    brighter -> Color

    // Returns a darker copy of this color, by a factor of 0.7
    darker -> Color

    // Returns an inverted copy of this color
    invert -> Color

    // Returns a greyscale copy of this color
    greyscale -> Color

    // Returns a string of this color, format: rgb(r, g, b)
    asString -> String
}

// -----------------
// CONSTRUCTOR METHODS
// -----------------

// Constructs and returns a black color object
method black -> Color

// Constructs and returns a blue color object
method blue -> Color

// Constructs and returns a cyan color object
method cyan -> Color

// Constructs and returns a doge color object
method doge -> Color

// Constructs and returns a magenta color object
method magenta -> Color

// Constructs and returns a grey color object
method grey -> Color

// Constructs and returns a green color object
method green -> Color

// Constructs and returns a orange color object
method orange -> Color
```

Utilities continued on next page. . .

```
// Constructs and returns a pink color object
method pink -> Color

// Constructs and returns a red color object
method red -> Color

// Constructs and returns a white color object
method white -> Color

// Constructs and returns a yellow color object
method yellow -> Color

// Constructs and returns a color object from input r,g,b values
method fromRGB(r : Number, g : Number, b : Number) -> Color

// Constructs and returns a color object from input r,g,b,a values
method fromRGB(r : Number, g : Number, b : Number) withAlpha(a : Number) -> Color
```

7.3 Vector2

Vectors are used throughout Grace-Phix, not just for cartesian coordinates but for storing any variables that can be conceptually grouped together. For example, width and height could be stored together in a two dimensional vector.

This module provides a Vector2 object and many constructor methods.

Listing 8 shows the interface for Vector2.

Listing 8: Vector2 interface

```
// Two dimensional Vector type.
// Ie is has an x and y coordinate.
type Vector2 = {

    // The first coordinate of this vector
    x -> Number

    // The second coordinate of this vector
    y -> Number

    // OPERATORS

    // Returns true if this vector has the same values as the given vector v
    ==(v : Vector2) -> Boolean

    // Returns true if this vector has different values from the given vector v
    !=(v : Vector2) -> Boolean

    // Creates a new 2d vector as the result of adding this vector with
    // the given vector v
    +(v : Vector2) -> Vector2
```

```
    // Creates a new 2d vector as the result of subtracting this vector
    // from the the given vector v
    -(v : Vector2) -> Vector2

    // Creates a new 2d vector as the result of multiplying the components
    // of this vector by the given scalar s
    *(s : Number) -> Vector2

    // Creates a new 2d vector as the result of dividing the components
    // of this vector by the given scalar s
    /(s : Number) -> Vector2

    // Returns if the magnitude of this vector is less than the
    // magnitude of the given vector
    <(v : Vector2) -> Boolean

    // Returns if the magnitude of this vector is greater than the
    // magnitude of the given vector
    >(v : Vector2) -> Boolean

    // Returns the inversion of this vector
    prefix- -> Vector2

    // METHODS

    // Returns the magnitude of this vector
    magnitude -> Number

    // Returns a normalised version of this vector
    normalize -> Vector2

    // Returns a new vector that is this vector clamped between the two values
    clampBetween(lower : Number, upper : Number) -> Vector2

    // Returns a new vector that is this vector clamped above the given threshold
    clampAbove(threshold : Number) -> Vector2

    // Returns a new vector that is this vector clamped below the given threshold
    clampBelow(threshold : Number) -> Vector2

    // Creates a new 2d vector as the result of adding the given scalar s to this vector
    addScalar(s : Number) -> Vector2

    // Creates a new 2d vector as the result of subtracting the given scalar s from this vector
    subScalar(s : Number) -> Vector2

    // Returns the vector as a list
    toList -> List

    // Returns the vector as a string
    asString -> String
}
```

Utilities continued on next page...

```
// CONSTRUCTOR METHODS

// Creates a new zero 2d vector
method zero -> Vector2

// Creates a new 2d vector and initialises it with the given x and y values
method setCoord(x : Number, y : Number) -> Vector2

// Creates a new 2d vector and initialises it with the values of the given
// 2d vector
method setVector2(v : Vector2) -> Vector2

// Creates a new 2d vector and initialises it with the x and y values of the
// given 3d vector and truncates the z value
method setVector3(v : Vector3) -> Vector2

// Creates a new 2d vector and initialises it with the x and y values of the
// given 4d vector and truncates the z and w values
method setVector4(v : Vector4) -> Vector2

// Creates a new 2d vector initialised with x: 1, y: 0
method xAxis -> Vector2

// Creates a new 2d vector initialised with x: 0, y: 1
method yAxis -> Vector2


// VECTOR MATH

// Returns the dot product of this vector with the other given vector
method dot(v1 : Vector2, v2 : Vector2) -> Number is public

// Returns the distance from this vector to the other vector
method distanceBetween(v1 : Vector2, v2 : Vector2) -> Number is public

// Returns the angle between this vector and the other vector, in radians
method angleBetween(v1 : Vector2, v2 : Vector2) -> Number is public
```