# The Hugs Graphics Library
# (Version 2.0)

Alastair Reid
Department of Computer Science
University of Utah
reid@cs.utah.edu

January 16, 2009

## 1 Introduction

The Hugs Graphics Library is designed to give the programmer access to most interesting parts of the Win32 Graphics Device Interface and X11 library without exposing the programmer to the pain and anguish usually associated with using these interfaces.

To give you a taste of what the library looks like, here is the obligatory "Hello World" program:

```
> module Hello where
>
> import GraphicsUtils
>
> helloWorld :: IO ()
> helloWorld = runGraphics (do
>   w <- openWindow "Hello World Window" (300, 300)
>   drawInWindow w (text (100, 100) "Hello")
>   drawInWindow w (text (100, 200) "World")
>   getKey w
>   closeWindow w
>   )
```

Here's what each function does:

- `runGraphics :: IO () -> IO ()` get Hugs ready to do graphics, runs an action (here, the action is a sequence of 5 subactions) and cleans everything up at the end.[1]
- `openWindow :: Title -> Point -> IO Window` opens a window specifying the window title "Hello World Window" and the size of the window (300 pixels × 300 pixels).
- `drawInWindow :: Window -> Graphic -> IO ()` draws a `Graphic` on a `Window`.

[1] The description of `runGraphics` is rather vague because of our promise to protect you from the full horror of Win32/X11 programming. If you really want to know, we highly recommend Charles Petzold's book "Programming Windows" [2] which does an excellent job with a difficult subject or Adrian Nye's "Xlib Programming Manual" [1] which is almost adequate.

- `text :: Point -> String -> Graphic` creates a `Graphic` consisting of a `String` at a given screen location.
- `getKey :: Window -> IO Char` waits for the user to press (and release) a key. This is necessary to prevent the window from closing before you have a chance to read what's on the screen.
- `closeWindow :: Window -> IO ()` closes the window.

The rest of this document is organized as follows: Section 2 describes the `Graphic` type (a declarative way of drawing pictures); Section 3 describes `Windows`; Section 4 describes `Events`; Section 5 describes the Concurrent Haskell primitives which you need to create complex interfaces; and Section 6 describes the `Draw` monad (a more imperative side to the `Graphic` type).

# 2 Graphics

In section 1, we used these two functions to draw to a window

```
> drawInWindow :: Window -> Graphic -> IO ()
> text         :: Point  -> String  -> Graphic
```

This section describes other ways of creating graphics that can be drawn to the screen.

## 2.1 Atomic Graphics

Here's a list of the atomic operations

```
> emptyGraphic ::                                   Graphic
> ellipse      :: Point -> Point                 -> Graphic
> shearEllipse :: Point -> Point -> Point        -> Graphic
> arc          :: Point -> Point -> Angle -> Angle -> Graphic
> line         :: Point -> Point                 -> Graphic
> polyline     :: [Point]                        -> Graphic
> polygon      :: [Point]                        -> Graphic
> polyBezier   :: [Point]                        -> Graphic
> text         :: Point -> String                -> Graphic
```

`emptyGraphic` is a blank `Graphic`.

`ellipse` is a filled ellipse which fits inside a rectangle defined by two `Point`s on the window. `shearEllipse` is a filled ellipse inside a parallelogram defined by three `Point`s on the window. `arc` is an unfilled elliptical arc which fits inside a rectangle defined by two `Point`s on the window. The angles specify the start and end points of the arc — the arc consists of all points from the start angle counter-clockwise to the end angle. Angles are in degrees [0..360] rather than radians [$0..2\pi$].

`line` is a line between two `Point`s. `polyline` is a series of lines through a list of `Point`s. `polyBezier` is a series of (unfilled) bezier curves defined by a list of $3n+1$ control `Point`s. `polygon` is a filled polygon defined by a list of `Point`s.

`text` is a rendered `String`.

**Portability Note:**

- `polyBezier` *is not provided in the X11 implementation of this library.*
- `shearEllipse` *is implemented by polygons on both Win32 and X11.*

**End Portability Note.**

## 2.2 Graphic Modifiers

One of the most useful properties of `Graphics` is that they can be modified in various ways. Here is a selection of the modifiers available

```
> withFont          :: Font      -> Graphic -> Graphic
> withTextColor     :: RGB       -> Graphic -> Graphic
> withTextAlignment :: Alignment -> Graphic -> Graphic
> withBkColor       :: RGB       -> Graphic -> Graphic
> withBkMode        :: BkMode    -> Graphic -> Graphic
> withPen           :: Pen       -> Graphic -> Graphic
> withBrush         :: Brush     -> Graphic -> Graphic
> withRGB           :: RGB       -> Graphic -> Graphic
```

The effect of these "modifiers" is to modify the way in which a graphic will be drawn. For example, if `courier :: Font` is a 10 point Courier font, then drawing `withFont courier (text (100,100) "Hello")` will draw the string `"Hello"` on the window using the 10 point Courier font.

Modifiers are cumulative: a series of modifiers can be applied to a single graphic. For example, the graphic

```
> withFont courier (
>   withTextColor red (
>     withTextAlignment (Center, Top) (
>       text (100,100) "Hello World"
>     )
>   )
> )
```

will be

- horizontally aligned so that the centre of the text is at `(100, 100)`;
- vertically aligned so that the top of the text is at `(100, 100)`;
- colored red
- displayed in 10 point Courier font

Modifiers nest in the obvious way — so

```
> withTextColor red (
>   withTextColor green (
>     text (100,100) "What Color Am I?"
>   )
> )
```

will produce green text, as expected.

**Aside**

*As you write more and more complex graphics, you'll quickly realize that it's very tedious to insert all those parentheses and to keep everything indented in a way that reveals its structure.*

*Fortunately, the Haskell Prelude provides a right associative application operator*

```
> ($) :: (a -> b) -> a -> b
```

*which eliminates the need for almost all parentheses when defining* `Graphics`*. Using the* `($)` *operator, the above example can be rewritten like this*

```
> withTextColor red   $
> withTextColor green $
> text (100,100) "What Color Am I?"
```

**End aside.**

## 2.3 Combining Graphics

The other useful property of `Graphics` is that they can be combined using the `overGraphic` combinator

```
> overGraphic :: Graphic -> Graphic -> Graphic
```

For example, drawing this graphic produces a red triangle "on top of" (or "in front of") a blue square

```
> overGraphic
>   (withBrush red  $ polygon [(200,200),(400,200),(300,400)])
>   (withBrush blue $ polygon [(100,100),(500,100),(500,500),(100,500)])
```

Notice that modifiers respect the structure of a graphic — modifiers applied to one part of a graphic have no effect on other parts of the graphic. For example the above graphic could be rewritten like this.

```
> withBrush blue $
> overGraphic
>   (withBrush red  $ polygon [(200,200),(400,200),(300,400)])
>   (polygon [(100,100),(500,100),(500,500),(100,500)])
```

The `overGraphics` function is useful if you want to draw a list of graphics. It's type and definition are

```
> overGraphics :: [Graphic] -> Graphic
> overGraphics = foldr overGraphic emptyGraphic
```

Notice that graphics at the head of the list are drawn "in front of" graphics at the tail of the list.

## 2.4 Attribute Generators

The graphic modifiers listed at the start of Section 2.2 use attributes with types like `Font`, `RGB` and `Brush`, but so far we have no way of generating any of these attributes.

Some of these types are *concrete* (you can create them using normal data constructors) and some are *abstract* (you can only create them with special "attribute generators"). Here's the definitions of the concrete types.

```
> type Angle     = Double
> type Dimension = Int
> type Point     = (Dimension,Dimension)
> data RGB       = RGB Int Int Int
>
> -- Text alignments
> type Alignment = (HAlign, VAlign)
> -- names have a tick to distinguish them from Prelude names (blech!)
> data HAlign = Left' | Center   | Right'
> data VAlign = Top   | Baseline | Bottom
>
> -- Text background modes
> data BkMode = Opaque | Transparent
```

The attributes `Font`, `Brush` and `Pen` are *abstract,* and are a little more complex because we want to delete the font, brush, or pen once we've finished using it. This gives the attribute generators a similar flavour to the modifiers seen in section 2.2 — these functions are applied to an argument of type $\alpha \rightarrow$ `Graphic` and return a `Graphic`.

```
> mkFont  :: Point -> Angle -> Bool -> Bool -> String ->
>                                     (Font  -> Graphic) -> Graphic
> mkBrush ::                  RGB -> (Brush -> Graphic) -> Graphic
> mkPen   :: Style -> Int -> RGB -> (Pen   -> Graphic) -> Graphic
```

For example, the following program uses a $50 \times 50$ pixel, non-bold, italic, courier font to draw red text on a green background at an angle of 45 degrees across the screen.

```
> fontDemo = runGraphics $ do
>   w <- openWindow "Font Demo Window" (100,100)
>   drawInWindow w $
>     withTextColor (RGB 255 0 0)                $
>     mkFont (50,100) (pi/4) False True "courier" $ \ font ->
>     withFont font                              $
>     withBkColor (RGB 0 255 0)                  $
>     withBkMode  Opaque                         $
>     text (50,50) "Font Demo"
>   getKey w
>   closeWindow w
```

A default font is substituted if the requested font does not exist. The rotation angle is ignored if the font is not a "TrueType" font (e.g., for `System` font on Win32).

**Portability Note:**

- *X11 does not directly support font rotation so* `mkFont` *always ignores the rotation angle argument in the X11 implementation of this library.*
- *Many of the font families typically available on Win32 are not available on X11 (and vice-versa). In our experience, the font families "courier," "helvetica" and "times" are somewhat portable.*

**End Portability Note.**

## 2.5 Brushes, Pens and Text Colors

If you were counting, you'll have noticed that there are five separate ways of specifying colors

```
> mkBrush       ::                  RGB -> (Brush -> Graphic) -> Graphic
> mkPen         :: Style -> Int -> RGB -> (Pen   -> Graphic) -> Graphic
> withTextColor ::                  RGB -> Graphic            -> Graphic
> withBkColor   ::                  RGB -> Graphic            -> Graphic
> withRGB       ::                  RGB -> Graphic            -> Graphic
```

What do these different modifiers and attributes control?

**Brushes** are used when filling shapes — so the brush color is used when drawing polygons, ellipses and regions.

**Pens** are used when drawing lines — so the pen color is used when drawing arcs, lines, polylines and polyBeziers.

Pens also have a "style" and a "width". The `Style` argument is used to select solid lines or various styles of dotted and dashed lines.

```
> data Style
>   = Solid
>   | Dash        -- "-------"
>   | Dot         -- "......."
>   | DashDot     -- "_._._._"
>   | DashDotDot  -- "_.._.._"
>   | Null
>   | InsideFrame
```

**TextColor** is used as the foreground color when drawing text.

**BkColor** is used as the background color when drawing text with background mode `Opaque`. The background color is ignored when the mode is `Transparent`.

Finally, `withRGB` is a convenience function which sets the brush, pen and text colors to the same value. Here is its definition

```
> withRGB :: RGB -> Graphic -> Graphic
> withRGB c g =
>   mkBrush c       $ \ brush ->
>   withBrush brush $
>   mkPen Solid 2 c $ \ pen ->
>   withPen pen     $
>   withTextColor c $
>   g
```

**Portability Note:**

- *On Win32, the pen is also used to draw a line round all the filled shapes — so the pen color also affects how polygons, ellipses and regions are drawn.*
- *One of the Win32 "gotchas" is that the choice of* `Style` *only applies if the width is 1 or less. With greater widths, the pen style will always be* `Solid` *no matter what you try to select. This problem does not apply to X11.*

**End Portability Note.**

## 2.6 Named Colors

Working with RGB triples is a pain in the neck so the `GraphicsUtils` module provides these built in colors as convenient "abbreviations."

```
> data Color
>   = Black
>   | Blue
>   | Green
>   | Cyan
>   | Red
>   | Magenta
>   | Yellow
>   | White
>  deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)
```

This type is useful because it may be used to index an array of RGB triples.

```
> colorTable :: Array Color RGB
```

For example, we provide this function which looks up a color in the `colorTable` and uses that color for the brush, pen and text color.

```
> withColor :: Color -> Graphic -> Graphic
```

It's worth pointing out that there's nothing "magical" about the `Color` type or our choice of colors. If you don't like our choice of colors, our names, or the way we mapped them onto RGB triples, you can write your own! To get you started, here's our implementation of `withColor` and `colorTable`.

```
> withColor c = withRGB (colorTable ! c)
>
> colorTable = array (minBound, maxBound) colorList
>
> colorList  :: [(Color, RGB)]
> colorList =
>   [ (Black   , RGB   0   0   0)
>   , (Blue    , RGB   0   0 255)
>   , (Green   , RGB   0 255   0)
>   , (Cyan    , RGB   0 255 255)
>   , (Red     , RGB 255   0   0)
>   , (Magenta , RGB 255   0 255)
>   , (Yellow  , RGB 255 255   0)
>   , (White   , RGB 255 255 255)
>   ]
```

## 2.7 Bitmaps

`Bitmap`s can be displayed in three ways:

1. with no transformation at a point
2. stretched to fit a rectangle
3. rotated and sheared to fit a parallelogram

Rectangles are specified by a pair of points: the top-left, and bottom-right corners of the rectangle.

```
> bitmap        :: Point                       -> Bitmap -> Graphic
> stretchBitmap :: Point  -> Point             -> Bitmap -> Graphic
> shearBitmap   :: Point  -> Point  -> Point   -> Bitmap -> Graphic
```

`Bitmap`s are read in from files and disposed of using

```
> readBitmap    :: String -> IO Bitmap
> deleteBitmap  :: Bitmap -> IO ()
```

(but be sure that the current `Graphic` on a `Window` doesn't contain a reference to a `Bitmap` before you delete the `Bitmap`!)

This operation gets the size of a bitmap.

```
> getBitmapSize :: Bitmap -> IO (Int, Int)
```

**Portability Note:**

- *The Bitmap functions are not currently provided in the X11 implementation of this library.*
- `shearBitmap` *is supported on Win'NT but not Win'95.*

**End Portability Note.**

## 2.8 Regions

`Regions` can be viewed as an efficient representation of sets of pixels. They are created from rectangles, ellipses, polygons and combined using set operations (intersection, union, difference and xor (symmetric difference)).

These are the operations available:

```
> emptyRegion     ::                    Region
> rectangleRegion :: Point -> Point -> Region
> ellipseRegion   :: Point -> Point -> Region
> polygonRegion   :: [Point]        -> Region
>
> intersectRegion :: Region -> Region -> Region
> unionRegion     :: Region -> Region -> Region
> subtractRegion  :: Region -> Region -> Region
> xorRegion       :: Region -> Region -> Region
>
> regionToGraphic :: Region -> Graphic
```

`withBrush` affects the color of `regionToGraphic`.

**Portability Note:**

- `emptyRegion` *is not provided in the Win32 implementation of this library. It is possible to use an empty rectangle region instead*
- `ellipseRegion` *is implemented using polygons in the X11 implementation of the library.*

**End Portability Note.**

## 2.9 The `Graphic` Algebra

The Graphic modifiers satisfy a large number of useful identities. For example,

- The triple $\langle$`Graphic`, `overGraphic`, `emptyGraphic`$\rangle$ forms a "monoid." If this wasn't true, we wouldn't find the `overGraphics` function very useful.
- Modifiers and generators all distribute over `overGraphic`. That is,

  ```
  > mkFoo <args> (p1 `overGraphic` p2)
  >  = (mkFoo <args> p1) `overGraphic` (mkFoo <args> p2)
  > withFoo foo  (p1 `overGraphic` p2)
  >  = (withFoo foo  p1) `overGraphic` (withFoo foo  p2)
  ```

  (These laws are especially useful when trying to make programs more efficient — see section 2.10.)

- "Independent" modifiers commute with each other. For example,

  ```
  > withTextColor c (withTextAlignment a p)
  >  = withTextAlignment a (withTextColor c p)
  ```

- Generators commute with modifiers. For example,

  ```
  > mkBrush c (\ b -> withBrush b' p) = withBrush b' mkBrush c (\ b -> p)
  ```

  if `b` and `b'` are distinct.

- Generators commute with other generators. For example

  ```
  > mkBrush c (\ b -> mkBrush c' (\ b' -> p))
  >  = mkBrush c' (\ b' -> mkBrush c (\ b -> p))
  ```

  if `b` and `b'` are distinct.

- "Irrelevant" modifiers and generators can be added or removed at will. For example, the text color has no effect on line drawing

  ```
  > withTextColor c (line p0 p1) = line p0 p1
  ```

  and there's no need to create a brush if you don't use it

  ```
  > mkBrush c (\ b -> p) = p, if b does not occur in p
  ```

  This last law can also be stated in the form

  ```
  > mkBrush c (\ b -> atomic) = atomic
  ```

  for any atomic operation.

The practical upshot of all this is that there are many ways to rearrange a graphic so that it will be drawn more (or less) efficiently. We explore this topic in the next section.

## 2.10 Efficiency Considerations

The other sections provide a very simple set of functions for creating graphics — but at the cost of ignoring efficiency. For example, this innocent looking graphic

```
> overGraphics
>   [ withColor Red $ ellipse (000,000) (100,100)
>   , withColor Red $ ellipse (100,100) (200,200)
>   , withColor Red $ ellipse (200,200) (300,300)
>   ]
```

will take longer to draw than this equivalent graphic

```
> mkBrush (colorTable ! Red) $ \ redBrush  ->
> overGraphics
>   [ withBrush redBrush $ ellipse (000,000) (100,100)
>   , withBrush redBrush $ ellipse (100,100) (200,200)
>   , withBrush redBrush $ ellipse (200,200) (300,300)
>   ]
```

Briefly, the problems are that `withColor` sets the color of the brush, the pen and the text but ellipses only use the brush color; and we're calling `withColor` 3 times more than we have to. This wouldn't matter if brush creation was cheap and easy. However, most typical workstations can only display at most 256 or 65536 different colors on the screen at once but allow you to specify any one of 16777216 different colors when selecting a drawing color — finding a close match to the requested color can be as expensive as drawing the primitive object itself.

This doesn't matter much for a graphic of this size — but if you're drawing several thousand graphic elements onto the screen as part of an animation, it can make the difference between a quite respectable frame rate of 20–30 frames per second and an absolutely unusable frame rate of 2–3 frames per second.

### 2.10.1 Eliminate calls to `withRGB` and `withColor`

At the risk of pointing out the obvious, the first step in optimizing a program in this way is to expand all uses of the `withRGB` and `withColor` functions and eliminating unnecessary calls to `mkBrush`, `mkPen` and `withTextColor`. Applying this optimization to the above `Graphic`, we obtain this (which should run about 3 times faster).

```
> overGraphics
>   [ mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (00,00) (10,10)
>   , mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (10,10) (20,20)
>   , mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (20,20) (30,30)
>   ]
```

### 2.10.2 Lifting generators to the top of `Graphics`

Another important optimization is to avoid creating many identical brushes, pens or fonts when one will do. We do this by "lifting" brush creation out to the top of a graphic. For example, this graphic

```
> overGraphics
>   [ mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (00,00) (10,10)
>   , mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (10,10) (20,20)
>   , mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (20,20) (30,30)
>   ]
```

creates three red brushes. It would be more efficient to rewrite it like this

```
> mkBrush red  $ \ redBrush  ->
> overGraphics
>   [ withBrush redBrush $ ellipse (00,00) (10,10)
>   , withBrush redBrush $ ellipse (10,10) (20,20)
>   , withBrush redBrush $ ellipse (20,20) (30,30)
>   ]
```

If your program uses a lot of brushes, it may be more convenient to store the brushes in a "palette" (i.e., an array of brushes)

```
> mkBrush red  $ \ redBrush  ->
> mkBrush blue $ \ blueBrush ->
> let palette = array (minBound, maxBound)
>                     [(Red, redBrush), (Blue, blueBrush)]
> in
```

```
> overGraphics
>   [ withBrush (palette ! Red)  $ ellipse (00,00) (10,10)
>   , withBrush (palette ! Blue) $ ellipse (10,10) (20,20)
>   , withBrush (palette ! Red)  $ ellipse (20,20) (30,30)
>   ]
```

### 2.10.3 Lifting generators out of graphics

Even this program has room for improvement: every time the graphic is redrawn (e.g., whenever the window is resized), it will create fresh brushes with which to draw the graphic. The graphics library provides a way round this — but it's more difficult and fraught with danger.

**Outline:** *This section will talk about using explicit creation and deletion functions to create brushes, fonts, etc.*

*The situation isn't very happy at the moment because it's easy to forget to deallocate brushes before you quit or to deallocate them before you change the graphic.*

**End outline.**

# 3 Windows

In section 1 we saw the function `drawInWindow` for drawing a `Graphic` on a `Window`. It turns out that `drawInWindow` is not a primitive function but, rather, it is defined using these two primitive functions which read the current `Graphic` and set a new `Graphic`.

```
> getGraphic   :: Window -> IO Graphic
> setGraphic   :: Window -> Graphic -> IO ()
```

Here's how these functions are used to define the function `drawInWindow` (which we used in section 1) and another useful function `clearWindow`.

```
> drawInWindow :: Window -> Graphic -> IO ()
> drawInWindow w p = do
>   oldGraphic <- getGraphic w
>   setGraphic w (p `over` oldGraphic)
>
> clearWindow :: Window -> IO ()
> clearWindow w = setGraphic w emptyGraphic
```

# 4 Events

The graphics library supports several different input devices (the mouse, the keyboard, etc) each of which can generate several different kinds of event (mouse movement, mouse button clicks, key presses, key releases, window resizing, etc.)

## 4.1 Keyboard events

In section 1 we saw the function `getKey` being used to wait until a key was pressed and released. The function `getKey` is defined in terms of a more general function `getKeyEx`

```
> getKeyEx     :: Window -> Bool -> IO Char
```

which can be used to wait until a key is pressed (`getKeyEx w True`) or until it is released (`getKeyEx w False`). The definition of `getKey` using this function is trivial:

```
> getKey       :: Window -> IO Char
> getKey w = do{ getKeyEx w True; getKeyEx w False }
```

## 4.2 Mouse events

As well as waiting for keyboard events, we can wait for mouse button events. We provide three functions for getting these events. `getLBP` and `getRBP` are used to wait for left and right button presses. Both functions are defined using `getButton` which can be used to wait for either the left button or the right button being either pressed or released.

```
> getLBP       :: Window -> IO Point
> getRBP       :: Window -> IO Point
> getButton    :: Window -> Bool -> Bool -> IO Point
>
> getLBP w = getButton w True  True
> getRBP w = getButton w False True
```

## 4.3 General events

The functions `getKeyEx` and `getButton` described in the previous sections are not primitive functions. Rather they are defined using the primitive function `getWindowEvent`

```
> getWindowEvent :: Window -> IO Event
```

which waits for the next "event" on a given `Window`. `Event`s are defined by the following data type.

```
> data Event
>    = Key       { char :: Char, isDown :: Bool }
>    | Button    { pt :: Point, isLeft, isDown :: Bool }
>    | MouseMove { pt :: Point }
>    | Resize
>    | Closed
>  deriving Show
```

These events are:

- `Key{char, isDown}` occurs when a key is pressed (`isDown==True`) or released (`isDown==False`). `char` is the "keycode" for the corresponding key. This keycode can be a letter, a number or some other value corresponding to the shift key, control key, etc.
- `Button{pt, isLeft, isDown}` occurs when a mouse button is pressed (`isDown==True`) or released (`isDown==False`). `pt` is the mouse position when the button was pressed and `isLeft` indicates whether it was the left or the right button.
- `MouseMove{pt}` occurs when the mouse is moved inside the window. `pt` is the position of the mouse after the movement.

- `Resize` occurs when the window is resized. The new window size can be discovered using these functions.

```
> getWindowRect   :: Window -> IO (Point, Size)
> getWindowSize   :: Window -> IO Size
> getWindowSize w = do
>   (pt,sz) <- getWindowRect w
>   return sz
```

- `Resize` occurs when the window is closed.

**Portability Note:**

- *Programmers should assume that the* `Event` *datatype will be extended in the not-too-distant future and that individual events may change slightly. As a minimum, you should add a "match anything" alternative to any function which pattern matches against* `Event`*s.*
- *X11 systems typically have three button mice. Button 1 is used as the left button, button 3 as the right button and button 2 (the middle button) is ignored.*

**End Portability Note.**

As examples of how `getWindowEvent` might be used in a program, here are the definitions of `getKeyEx` and `getButton`.

```
> getKeyEx     :: Window -> Bool -> IO Char
> getKeyEx w down = loop
>  where
>   loop = do
>         e <- getWindowEvent w
>         case e of
>           Key{ char = c, isDown }
>             |  isDown == down
>             -> return c
>           _ -> loop
```

```
> getButton    :: Window -> Bool -> Bool -> IO Point
> getButton w left down = loop
>  where
>   loop = do
>         e <- getWindowEvent w
>         case e of
>           Button{pt,isLeft,isDown}
>             | isLeft == left && isDown == down
>             -> return pt
>           _ -> loop
```

## 4.4 Using Timers

If you want to use a timer, you have to open the window using `openWindowEx` instead of `openWindow`

```
> openWindowEx :: Title -> Maybe Point -> Size ->
>                 RedrawMode -> Maybe Time -> IO Window
>
> data RedrawMode
>   = Unbuffered
>   | DoubleBuffered
```

This *extended* version of `openWindow` takes extra parameters which specify

- the initial position of a window;
- how to display a graphic on a window; and
- the time between ticks (in milliseconds).

The function `openWindow` is defined using `openWindowEx`

```
> openWindow name size = openWindowEx name Nothing size Unbuffered Nothing
```

The drawing mode can be either `DoubleBuffered` which uses a "double buffer" to reduce flicker or `Unbuffered` which draws directly to the window and runs slightly faster but is more prone to flicker. You should probably use `DoubleBuffered` for animations.

The timer generates "tick events" at regular intervals. The function `getWindowTick` waits for the next "tick event" to occur.

```
> getWindowTick :: Window -> IO ()
```

**Aside**

*With normal events, like button presses, we store every event that happens until you remove that event from the queue. If we did this with tick events, and your program takes a little too long to draw each frame of an animation, the event queue could become so swamped with "ticks" that you'd never respond to user input. To avoid this problem, we only insert a tick into the queue if there's no tick there already.*

**End aside.**

Here's a simple example of how to use timers. Note the use of `setGraphic` instead of `drawInWindow`.

```
> timerDemo = do
>   w <- openWindowEx
>          "Timer demo"          -- title
>          (Just (500,500))      -- initial position of window
>          (100,100)             -- initial size of window
>          DoubleBuffered        -- drawing mode - see above
>          (Just 50)             -- tick rate
>   let
>     loop x = do
>       setGraphic w $ text (0,50) $ show x
>       getWindowTick w          -- wait for next tick on window
>       loop (x+1)
>   loop 0
```

# 5 Concurrent Haskell

If you want to use multiple windows or each window contains a number of essentially independent components, it is convenient to use separate threads for handling each window. Hugs provides a simple mechanism for doing that.

The simplest concurrency primitives are `par` and `par_`

```
> par  :: IO a -> IO b -> IO (a,b)
> par_ :: IO a -> IO b -> IO ()
```

(These are both exported from the `GraphicsUtils` module.)

These run two `IO` actions in parallel and terminate when both actions terminate. The function `par_` discards the results of the actions.

**Aside**

*The underscore in the name* `par_` *is derived from the use of the underscore in the definition of* `par_`.

```
> par_ p q = (p `par` q) >>= \ _ -> return ()
```

*This naming convention is also used in the Haskell Prelude and standard libraries (*`mapM_`*,* `zipWithM_`*, etc.).*

**End aside.**

The function `parMany` generalizes `par_` to lists.

```
> parMany :: [ IO () ] -> IO ()
> parMany = foldr par_ (return ())
```

Of course, you'll quickly realise that there's not much point in being able to create concurrent threads if threads can't communicate with each other. Hugs provides an implementation of the "Concurrent Haskell" primitives described in the Concurrent Haskell paper [3] to which we refer the enthusiastic reader.

# 6 The `Draw` monad

The `Graphic` type, operations and combinators provide a flexible, efficient and convenient way of drawing images on a window and encapsulate good programming practice by cleaning up any changes they must make to the state of the window. In some applications though, it is appropriate to use a lower-level, more error-prone interface for drawing images. For example, when building a library on top of the Graphics library, one might want to build on a slightly more efficient, less secure interface. Or, when teaching courses on computer graphics, it would not be possible to demonstrate low-level aspects of graphics using an interface which hides those aspects. This section describes the `Draw` monad (an imperative graphics interface) and describes how this is used to implement the `Graphic` type (a declarative graphics interface). This section can be ignored by most readers.

## 6.1 The `Draw` monad and the `Graphic` type

The `Graphic` type lets you describe what an image should look like; the `Draw` monad lets you describe how to build an image. These views intersect for atomic graphics. For example, the function to draw a line can serve both as a description and as the implementation. This is exploited in the graphics library by defining `Graphic` as an instance of the `Draw` monad. Thus, all `Graphic` types and operations listed in section 2 can also be used with the `Draw` monad.

```
> data Draw a = ...
> instance Functor Draw where ...
> instance Monad   Draw where ...
>
> type Graphic = Draw ()
```

The `emptyGraphic` and `overGraphic` functions are implemented using this monad. Their definitions should not be surprising.

```
> emptyGraphic        = return ()
> g1 `overGraphic` g2 = g2 >> g1
```

## 6.2 `Draw` modifiers and generators

The difference between the `Draw` monad and the `Graphic` type is that the `Graphic` modifiers and combinators respect the structure of the graphic (see section 2.3). For example, the `withBrush` modifier only affects the color of the `Graphic` it is applied to, it does not affect the color of the `Graphic` it is embedded in. In contrast, the `Draw` monad provides operations which change the effect of subsequent drawing operations. The following operations correspond to the graphics modifiers described in section 2.2.

```
> selectFont       :: Font          -> Draw Font
> setTextColor     :: RGB           -> Draw RGB
> setTextAlignment :: Alignment     -> Draw Alignment
> setBkColor       :: RGB           -> Draw RGB
> setBkMode        :: BkMode        -> Draw BkMode
> selectPen        :: Pen           -> Draw Pen
> selectBrush      :: Brush         -> Draw Brush
```

These operations all have a type of the form $\alpha \rightarrow$ `Draw` $\alpha$. The value returned is the old value of the attribute being changed and can be used to restore the attribute to its previous value. For example, the `withFont` modifier could be implemented like this:

```
> withFont new g = do
>   old <- selectFont new
>   g
>   selectFont old
>   return ()
```

**Aside**

*This pattern of use is very common in imperative programs so the Haskell* `IO` *library provides two combinators which encapsulate this behavior. The* `bracket` *function takes three operations as arguments: a pre-operation* `left`*, a post-operation* `right` *and an operation* `middle` *and performs them in the order* `left`*;* `middle`*;* `right`*. The arguments are provided in the order* `left`*,* `right`*,* `middle` *because the* `left` *and* `right` *operations are often "inverses" of each other such as* `openFile` *and* `closeFile`*. The* `bracket_` *function is similar and is used when the* `middle` *operation does not require the result of the* `left` *operation.*

```
> bracket  :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
> bracket_ :: IO a -> (a -> IO b) -> IO c        -> IO c
>
> bracket left right middle = do
>   a <- left
>   c <- middle a
>   right a
>   return c
```

```
>
> bracket_ left right middle = bracket left right (const middle)
```

**End aside.**

The graphics library provides similar combinators for the `Draw` monad:

```
> bracket  :: Draw a -> (a -> Draw b) -> (a -> Draw c) -> Draw c
> bracket_ :: Draw a -> (a -> Draw b) -> Draw c        -> Draw c
```

**Aside**

*In fact, the* `bracket` *and* `bracket_` *functions do slightly more than the above description suggests. Those provided in the* `IO` *library use Haskell's error-catching facilities to ensure that the* `right` *operation is performed even if the* `middle` *operation raises an* `IOError` *whilst those in the Graphics library use Hugs' exception-handling facilities to ensure that the* `right` *operation is performed even if the* `middle` *operation raises an exception.*

**End aside.**

Using these combinators, it is trivial to implement the modifiers described in section 2.2.

```
> withFont          x = bracket_ (selectFont       x) selectFont
> withTextColor     x = bracket_ (setTextColor     x) setTextColor
> withTextAlignment x = bracket_ (setTextAlignment x) setTextAlignment
> withBkColor       x = bracket_ (setBkColor       x) setBkColor
> withBkMode        x = bracket_ (setBkMode        x) setBkMode
> withPen           x = bracket_ (selectPen        x) selectPen
> withBrush         x = bracket_ (selectBrush      x) selectBrush
```

# References


[1] A. Nye. *Xlib Programming Manual*. O'Reilly and Associates, Inc., 1988. ISBN 0-937175-26-9.

[2] C. Petzold. *Programming Windows*. Microsoft Press, 1999. ISBN 1-57321-995-X (hardback).

[3] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, FL, January 1996. ACM press.


# A Quick Reference

The exported (stable) interface of the library consists of all symbols exported from `GraphicsCore` and `GraphicsUtils`. `GraphicsUtils` reexports all symbols exported by `GraphicsCore` and it is expected that most users will only import `GraphicsUtils`; the `GraphicsCore` interface is aimed solely at those wishing to use the graphics library as a base on which to build their own library or who find the `GraphicsUtils` interface inappropriate for their needs.

## A.1 Module GraphicsCore

```
> type Title = String
> type Point = (Int,Int)
> type Size  = (Int,Int)
> type Angle = Double
> type Time  = Word32 -- milliseconds
> data RGB = RGB Word8 Word8 Word8
> data BkMode = Opaque | Transparent
>
> type Alignment = (HAlign, VAlign)
> -- names have a tick to distinguish them from Prelude names (blech!)
> data HAlign = Left' | Center   | Right'
>  deriving (Enum, Eq, Ord, Ix, Show)
> data VAlign = Top   | Baseline | Bottom
>  deriving (Enum, Eq, Ord, Ix, Show)
>
> data Style
>   = Solid
>   | Dash        -- "-------"
>   | Dot         -- "......."
>   | DashDot     -- "_._._._"
>   | DashDotDot  -- "_.._.._"
>   | Null
>   | InsideFrame
>
> runGraphics      :: IO () -> IO ()
> getTime          :: IO Time
>
> data Window
> openWindowEx     :: Title -> Maybe Point -> Size ->
>                     RedrawMode -> Maybe T.Time -> IO Window
>
> closeWindow      :: Window -> IO ()
> getWindowRect    :: Window -> IO (Point,Point)
> getWindowEvent   :: Window -> IO Event
> getWindowTick    :: Window -> IO ()
> maybeGetWindowEvent :: Window -> IO (Maybe Event)
>
> type Graphic = Draw ()
> setGraphic       :: Window -> Graphic -> IO ()
> getGraphic       :: Window -> IO Graphic
> modGraphic       :: Window -> (Graphic -> Graphic) -> IO ()
> directDraw       :: Window -> Graphic -> IO ()
>
> selectFont       :: Font          -> Draw Font
> setTextColor     :: RGB           -> Draw RGB
> setTextAlignment :: Alignment     -> Draw Alignment
> setBkColor       :: RGB           -> Draw RGB
> setBkMode        :: BkMode        -> Draw BkMode
> selectPen        :: Pen           -> Draw Pen
```

```
> selectBrush      :: Brush         -> Draw Brush
>
> bracket          :: Draw a -> (a -> Draw b) -> (a -> Draw c) -> Draw c
> bracket_         :: Draw a -> (a -> Draw b) -> Draw c -> Draw c
>
> data Font
> createFont       :: Point -> Angle -> Bool -> Bool -> String -> IO Font
> deleteFont       :: Font -> IO ()
>
> data Brush
> mkBrush          :: RGB                -> (Brush -> Draw a) -> Draw a
>
> data Pen
> mkPen            :: Style -> Int -> RGB -> (Pen   -> Draw a) -> Draw a
> createPen        :: Style -> Int -> RGB -> IO Pen
>
> arc              :: Point -> Point -> Angle -> Angle -> Graphic  -- unfilled
> line             :: Point -> Point                   -> Graphic  -- unfilled
> polyline         :: [Point]                          -> Graphic  -- unfilled
> ellipse          :: Point -> Point                   -> Graphic  -- filled
> shearEllipse     :: Point -> Point -> Point          -> Graphic  -- filled
> polygon          :: [Point]                          -> Graphic  -- filled
> text             :: Point -> String                  -> Graphic  -- filled
>
> data Region
> emptyRegion      :: Region
> rectangleRegion  :: Point -> Point -> Region
> ellipseRegion    :: Point -> Point -> Region
> polygonRegion    :: [Point] -> Region
> intersectRegion  :: Region -> Region -> Region
> unionRegion      :: Region -> Region -> Region
> subtractRegion   :: Region -> Region -> Region
> xorRegion        :: Region -> Region -> Region
> regionToGraphic  :: Region -> Graphic
>
> data Event
>   = Key       { char :: Char, isDown :: Bool }
>   | Button    { pt :: Point, isLeft, isDown :: Bool }
>   | MouseMove { pt :: Point }
>   | Resize
>   | Closed
>  deriving Show
```

## A.2 Module GraphicsUtils

Note that this document repeats the definitions of all the functions defined in `GraphicsUtils`.

```
> -- Reexports GraphicsCore
>
> openWindow        :: Title -> Size -> IO Window
> clearWindow       :: Window -> IO ()
```

```
> drawInWindow      :: Window -> Graphic -> IO ()
>
> getWindowSize     :: Window -> IO Size
> getLBP            :: Window -> IO Point
> getRBP            :: Window -> IO Point
> getButton         :: Window -> Bool -> Bool -> IO Point
> getKey            :: Window -> IO Char
> getKeyEx          :: Window -> Bool -> IO Char
>
> emptyGraphic      :: Graphic
> overGraphic       :: Graphic -> Graphic -> Graphic
> overGraphics      :: [Graphic] -> Graphic
>
> withFont          :: Font      -> Graphic -> Graphic
> withTextColor     :: RGB       -> Graphic -> Graphic
> withTextAlignment :: Alignment -> Graphic -> Graphic
> withBkColor       :: RGB       -> Graphic -> Graphic
> withBkMode        :: BkMode    -> Graphic -> Graphic
> withPen           :: Pen       -> Graphic -> Graphic
> withBrush         :: Brush     -> Graphic -> Graphic
> withRGB           :: RGB       -> Graphic -> Graphic
>
> data Color
>   = Black
>   | Blue
>   | Green
>   | Cyan
>   | Red
>   | Magenta
>   | Yellow
>   | White
>  deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)
>
> colorList         :: [(Color, RGB)]
> colorTable        :: Array Color RGB
> withColor         :: Color -> Graphic -> Graphic
>
> par               :: IO a -> IO b -> IO (a,b)
> par_              :: IO a -> IO b -> IO ()
> parMany           :: [IO ()] -> IO ()
```

## A.3 Portability notes

- `polyBezier` is not provided in the X11 implementation of this library.
- `shearEllipse` is implemented by polygons on both Win32 and X11.
- X11 does not directly support font rotation so `mkFont` always ignores the rotation angle argument in the X11 implementation of this library.
- Many of the font families typically available on Win32 are not available on X11 (and *vice-versa*). In our experience, the font families "courier," "helvetica" and "times" are somewhat portable.

- On Win32, the pen is also used to draw a line round all the filled shapes — so the pen color also affects how polygons, ellipses and regions are drawn.
- One of the Win32 "gotchas" is that the choice of `Style` only applies if the width is 1 or less. With greater widths, the pen style will always be `Solid` no matter what you try to select. This problem does not apply to X11.
- The Bitmap functions are not currently provided in the X11 implementation of this library.
- `shearBitmap` is supported on Win'NT but not Win'95.
- `emptyRegion` is not provided in the Win32 implementation of this library. It is possible to use an empty rectangle region instead
- `ellipseRegion` is implemented using polygons in the X11 implementation of the library.
- Programmers should assume that the `Event` datatype will be extended in the not-too-distant future and that individual events may change slightly. As a minimum, you should add a "match anything" alternative to any function which pattern matches against `Event`s.
- X11 systems typically have three button mice. Button 1 is used as the left button, button 3 as the right button and button 2 (the middle button) is ignored.