The Hugs Graphics Library
(Draft)

Alastair Reid
Yale University
Department of Computer Science
New Haven, CT 06520
reid-alastair@cs.yale.edu

1 April, 1997

1 Introduction

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

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

```
> module Hello where
>
> import Graphics
>
> helloWorld :: IO ()
> helloWorld = runGraphics (do
>         w <- openWindow "Hello World Window" (300, 300)
>         draw w (text (100, 100) "Hello")
>         draw 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 4 subactions) and cleans everything up at the end.[1]

[1] The description of `runGraphics` is rather vague because of our promise to protect you from the full horror of Win32 programming. If you really want to know, we highly recommend Charles Petzold's book "Programming Windows 95" which does an excellent job with a difficult subject.

- `openWindow :: String -> Point -> IO Window` opens a window specifying the window title "Hello World Window" and the size of the window (300 pixels $\times$ 300 pixels).
- `draw :: Window -> Picture -> IO ()` draws a `Picture` on a `Window`.
- `text :: Point -> String -> Picture` creates a `Picture` consisting of a `String` at a given screen location.
- `getKey :: Window -> IO ()` 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 organised as follows:

- Section 2 describes `Picture`s.
- Section 3 describes `Window`s.
- Section 4 describes `Event`s.
- Section 5 describes the Concurrent Haskell primitives which you need to create complex interfaces.

2 Pictures

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

```
> draw :: Window -> Picture -> IO ()
> text :: Point  -> String  -> Picture
```

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

2.1 Primitive Pictures

Here's a list of the primitive operations

```
> empty        :: Picture
> ellipse      :: Point -> Point          -> Picture
> shearEllipse :: Point -> Point -> Point -> Picture
> line         :: Point -> Point          -> Picture
> polyline     :: [Point] -> Picture
> polygon      :: [Point] -> Picture
> polyBezier   :: [Point] -> Picture
> text         :: Point -> String -> Picture
```

`empty` is a blank `Picture`.

`ellipse` is a filled ellipse which fits inside a rectangle defined by two `Points` on the window. `shearEllipse` is a filled ellipse inside a parallelogram defined by three `Points` on the window.

ToDo: *Is there a way to draw unfilled ellipses?*

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

`text` is a rendered `String`.

2.2 Picture Modifiers

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

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

The effect of these "modifiers" is to modify the way in which a picture 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 picture. For example, the picture

```
> 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)`;

- coloured red
- displayed in 10 point Courier font

Modifiers nest in the obvious way — so

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

will produce green text, as expected.

Aside:

As you write more and more complex pictures, you'll quickly realise 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 pictures. Using the `($)` *operator, the above example can be rewritten like this*

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

End aside.

2.3 Combining Pictures

The other useful property of `Picture`s is that they can be combined using the `over` combinator

```
> over :: Picture -> Picture -> Picture
```

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

```
> over
>   (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 picture — modifiers applied to one part of a picture have no effect on other parts of the picture. For example the above picture could be rewritten like this.

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

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

```
> overMany :: [Picture] -> Picture
> overMany = foldr over empty
```

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

2.4 Attribute Generators

The picture 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 $\alpha \rightarrow$ `Picture` and return a `Picture`.

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

For example, the following program uses a 50×50 pixel, non-bold, italic, Arial 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" (5,5)
>   draw w $
>     withTextColor (RGB 255 0 0)              $
>     mkFont (50,100) (pi/4) False True "Arial" $ \ font ->
>     withFont font                            $
>     withBkColor (RGB 0 255 0)                $
>     withBkMode  Opaque                       $
>     text (0.5,4.5) "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 (eg for `System` font).

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 -> Picture) -> Picture
> mkPen         :: Style -> Int -> RGB -> (Pen   -> Picture) -> Picture
> withTextColor ::                 RGB -> Picture            -> Picture
> withBkColor   ::                 RGB -> Picture            -> Picture
> withRGB       ::                 RGB -> Picture            -> Picture
```

What do these different modifiers and attributes control?

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

Pens are used when drawing lines — so the pen colour is used when drawing lines, polylines and polyBeziers. Surprisingly, the pen is also used to draw a line round all the filled shapes — so the pen colour also affects how polygons, ellipses and regions are drawn.

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
```

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.

ToDo: *Should I split* `mkPen` *into two separate functions? One to do wide pens and one to do patterned pens?*

TextColor is used as the foreground colour when drawing text.

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

ToDo: *Should I expand the name* `BkColor` *to* `BackgroundColor`*?*

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

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

2.6 Named Colours

Working with RGB triples is a pain in the neck so the `GraphicsColor` module provides these built in colours 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 colour in the `colorTable` and uses that colour for the brush, pen and text colour.

```
> withColor :: Color -> Picture -> Picture
```

It's worth pointing out that there's nothing "magical" about the `Color` type or our choice of colours. If you don't like our choice of colours, 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

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

Portability Note: `shearBitmap` *is not supported on Win'95.*

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

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

(but be sure that the current `Picture` 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)
```

ToDo: *Describe the other bitmap operations — clean them up a bit first though!*

2.8 Regions

The most complicated graphics objects are `Region`s. `Region`s can be viewed as sets of pixels on the screen. They are created from rectangles, ellipses, polygons and combined using set operations (union, intersection, difference and xor).

The main difficulty is that, in order to make sure that `Region`s are allocated and deallocated correctly, all `Region` operations have a type of the form

$\alpha \rightarrow$ (`Region` $\rightarrow$ `Picture`) $\rightarrow$ `Picture`

That is, all `Region` operations are applied to a "continuation" which will use the `Region` to generate a `Picture`.

These are the operations available

```
> mkEmpty     ::                     (Region -> Picture) -> Picture
> mkRectangle :: Point -> Point   -> (Region -> Picture) -> Picture
> mkEllipse   :: Point -> Point   -> (Region -> Picture) -> Picture
> mkPolygon   :: [Point]          -> (Region -> Picture) -> Picture
>
> andRegion   :: Region -> Region -> (Region -> Picture) -> Picture
> orRegion    :: Region -> Region -> (Region -> Picture) -> Picture
> xorRegion   :: Region -> Region -> (Region -> Picture) -> Picture
> diffRegion  :: Region -> Region -> (Region -> Picture) -> Picture
>
> region      :: Region -> Picture
```

ToDo: *Should the combining operations have more setlike names?*

Portability Note: *We'd like to port most of this library over to X11; however, we suspect that the* `Region` *operations will be hard to port.*

2.9 The Picture Algebra

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

- The triple ⟨`Picture`, `over`, `empty`⟩ forms a "monoid." If this wasn't true, we wouldn't find the `overMany` function very useful.
- Modifiers and generators all distribute over `over`. That is,

  ```
  > mkFoo <args> (p1 `over` p2) = (mkFoo <args> p1) `over` (mkFoo <args> p2)
  > withFoo foo  (p1 `over` p2) = (withFoo foo  p1) `over` (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 colour 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 -> prim) = prim
  ```

  for any primitive operation.)

ToDo: *Add tables describing which modifiers are relevant and which ones are independent. Or just list all the identities.*

The practical upshot of all this is that there are many ways to rearrange a picture 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 pictures — but at the cost of ignoring efficiency. For example, this innocent looking picture

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

will take about 10 times as long to draw as this equivalent picture

```
> mkBrush (colorTable ! Red) $ \ redBrush  ->
> overMany
>   [ 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 colour of the brush, the pen and the text but ellipses only use the brush colour; 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 colours on the screen at once but allow you to specify any one of 16777216 different colours when selecting a drawing colour — finding a close match to the requested colour can be as expensive as drawing the primitive object itself.

ToDo: *Time both pictures — check that my estimate is about right.*

This doesn't matter much for a picture of this size — but if you're drawing several thousand picture 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.

Aside:

The (lazy) functional programming community has a bad habit of ignoring these kinds of considerations; with the result that C programmers have acquired the notion that functional programs will necessarily *run* several orders of magnitude *more slowly than equivalent C programs. On the basis of this notion, they* quite rightly *regard functional languages as toys which have no relevance to Real Programming.*

We'd like to dispel that belief and so, in designing the graphics library, we have made a serious attempt to expose enough of the underlying machinery that we can tackle this sort of efficiency consideration.

That said, it's worth emphasising that **Hugs** *is an interpreter which makes it run 10–100 times more slowly than compiled implementations of* **Haskell** *such as* **GHC**. *If you really are wanting to animate thousands of objects, you probably shouldn't be relying on* **Hugs**.

End aside.

2.11 Eliminate calls to `withRGB` and `withColor`

At the risk of pointing out the obvious, the first step in optimising 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 optimisation to the above `Picture`, we obtain this (which should run about 3 times faster).

```
> overMany
>   [ mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (000,000) (100,100)
>   , mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (100,100) (200,200)
>   , mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (200,200) (300,300)
>   ]
```

2.12 Lifting generators to the top of `Pictures`

Another important optimisation 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 picture. For example, this picture

```
> overMany
>   [ mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (000,000) (100,100)
>   , mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (100,100) (200,200)
>   , mkBrush red $ \ redBrush -> withBrush redBrush $ ellipse (200,200) (300,300)
>   ]
```

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

```
> mkBrush red  $ \ redBrush  ->
> overMany
>   [ withBrush redBrush $ ellipse (000,000) (100,100)
>   , withBrush redBrush $ ellipse (100,100) (200,200)
>   , withBrush redBrush $ ellipse (200,200) (300,300)
>   ]
```

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

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

```
>   [ withBrush (palette ! Red)  $ ellipse (000,000) (100,100)
>   , withBrush (palette ! Blue) $ ellipse (100,100) (200,200)
>   , withBrush (palette ! Red)  $ ellipse (200,200) (300,300)
>   ]
```

ToDo: *Write the obvious function with type* `[RGB] -> ([Brush] -> Picture) -> Picture` *(and similarily, for Pens, Fonts, etc).*

2.13 Lifting generators out of pictures

Even this program has room for improvement: every time the picture is redrawn (eg whenever the window is resized), it will create fresh brushes with which to draw the picture. 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 picture.

Maybe things will be better by the time we do an official release...

End outline.

3 Windows

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

```
> getPicture   :: Window -> IO Picture
> setPicture   :: Window -> Picture -> IO ()
```

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

```
> draw         :: Window -> Picture -> IO ()
> draw w p = do
>         oldPicture <- getPicture w
>         setPicture w (p `over` oldPicture)
>
> clearWindow :: Window -> IO ()
> clearWindow w = setPicture w empty
```

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 `getEvent`

```
> getEvent     :: 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
>  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.

  ToDo: *Say more about what the keycode is — in the meantime, users will just have to try a few experiments to find out which code each key produces.*

- `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.

  Portability Note: *When we support X11 as well, we'll have to change this datatype to support up to 5 buttons (with 3 being the usual number).*

- `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.

  ```
  > windowRect   :: Window -> IO (Point, Point)
  > windowSize   :: Window -> IO Point
  ```

Portability Note: *Programmers should assume that this 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.*

As examples of how `getEvent` 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 <- getEvent 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 <- getEvent w
>         case e of
>         Button {pt,isLeft,isDown}
>           | isLeft == left && isDown == down
>           -> return pt
>         _ -> loop
```

4.4 Using Timers

Portability Note: *Timers are not very well integrated with the rest of the library at the moment. We plan to improve this situation in future versions.*

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

```
> openWindowEx :: String -> Maybe Point -> Maybe Point ->
>                 (Picture -> DrawFun) -> Maybe Int ->
>                 IO Window
```

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

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

The drawing function can be either `drawBufferedPicture` which uses a "double buffer" to reduce flicker or `drawPicture` which draws directly to the window and runs slightly faster. You should probably use `drawBufferedPicture` for animations.

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

```
> getTick      :: 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

```
timerDemo = do
  w <- openWindowEx
         "Timer demo"          -- title
         (Just (500,500))      -- initial position of window
         (Just (100,100))      -- initial size of window
         drawBufferedPicture   -- draw function - see above
         (Just 50)             -- tick rate
  let
    loop x = do
      setPicture w $ text (0,50) $ show x
      getTick 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 `MVar` 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` generalises `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 this paper

```
"Concurrent Haskell"
Simon Peyton Jones, Andrew Gordon and Sigbjorn Finne.
In Proceedings of the ACM Symposium on Principles of Programming
Languages,St Petersburg Beach, Florida, January 1996.
http://www.dcs.gla.ac.uk/fp/authors/Simon_Peyton_Jones/
  concurrent-haskell.ps
```

to which we refer the enthusiastic reader.

Aside: *Programmers should be aware that there is one significant difference between* **Hugs** *implementation of concurrency and* **GHC***s.*

GHC *uses preemptive multitasking.*

Context switches can occur at any time (except if you call a C function (like "getchar") which blocks the entire process while waiting for input.

Hugs *uses cooperative multitasking.*

Context switches only occur when you use one of the primitives defined in this module. This means that programs such as:

```
> main = forkIO (write 'a') >> write 'b'
>  where
>   write c = putChar c >> write c
```

will print either "aaaaaaaaaaaaaa..." or "bbbbbbbbbbbb..." instead of some random interleaving of 'a's and 'b's.

Cooperative multitasking is sufficient for writing coroutines and simple graphical user interfaces.

End aside.

A Quick Reference

This section consists of a quick summary of every module.

ToDo: *It would be very cool to turn this appendix into an index containing the type of each function, the definition of each transparent type and the page numbers on which it is discussed. Sadly, I have no idea how to do it.*

A.1 Module GraphicsPicture

```
> -- Reexports everything exported by the following modules:
> --
> --    GraphicsTypes
> --    GraphicsText
> --    GraphicsRGN
> --    GraphicsFont
> --    GraphicsBrush
> --    GraphicsPen
> --    GraphicsBitmap
>
> empty        :: Picture
> over         :: Picture -> Picture -> Picture
> overMany     :: [Picture] -> Picture
>
> ellipse      :: Point -> Point          -> Picture
> shearEllipse :: Point -> Point -> Point -> Picture
> line         :: Point -> Point          -> Picture
>
> polyline     :: [Point] -> Picture
> polygon      :: [Point] -> Picture
> polyBezier   :: [Point] -> Picture
>
> withRGB      :: RGB -> Picture -> Picture
```

A.2 Module GraphicsTypes

```
> type Angle     = Double
> type Dimension = Int
> type Point     = (Dimension,Dimension)
>
> data RGB = RGB Int Int Int
>
> type Picture = ...
```

A.3 Module GraphicsText

```
> 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
>
```

```
> data BkMode = Opaque | Transparent
>
> text               :: Point     -> String  -> Picture
> withTextColor      :: RGB       -> Picture -> Picture
> withTextAlignment :: Alignment -> Picture -> Picture
> withBkColor        :: RGB       -> Picture -> Picture
> withBkMode         :: BkMode    -> Picture -> Picture
```

A.4 Module GraphicsFont

```
> newtype Font = ...
>
> mkFont     :: Point -> Angle -> Bool -> Bool -> String ->
>               (Font -> Picture) -> Picture
> withFont   :: Font -> Picture -> Picture
>
> createFont :: Point -> Angle -> Bool -> Bool -> String -> IO Font
> deleteFont :: Font -> IO ()
```

A.5 Module GraphicsBrush

```
> newtype Brush = ...
>
> mkBrush     :: RGB   -> (Brush -> Picture) -> Picture
> withBrush   :: Brush -> Picture -> Picture
>
> createBrush :: RGB   -> IO Brush
> deleteBrush :: Brush -> IO ()
```

A.6 Module GraphicsPen

```
> newtype Pen = ...
>
> data Style
>   = Solid
>   | Dash          -- "-------"
>   | Dot           -- "......."
>   | DashDot       -- "_._._._"
>   | DashDotDot    -- "_.._.._"
>   | Null
>   | InsideFrame
>
> withPen   :: Pen -> Picture -> Picture
> mkPen     :: Style -> Int -> RGB -> (Pen -> Picture) -> Picture
```

```
> createPen :: Style -> Int -> RGB -> IO Pen
> deletePen :: Pen -> IO ()
```

A.7 Module GraphicsBitmap

```
> newtype Bitmap = ...
>
> loadBitmap    :: String -> IO (Bitmap, (Int, Int))
> readBitmap    :: String -> IO Bitmap
> deleteBitmap  :: Bitmap -> IO ()
>
> getBitmapSize :: Bitmap -> IO (Int, Int)
>
> -- Bitmaps can be drawn in three ways:
> -- a) with no transformation at  a point
> -- b) stretched to fit           a rectangle
> -- c) rotated and sheared to fit a parallelogram
> --
> -- Sadly, the latter isn't supported in Win'95
>
> bitmap        :: Point                      -> Bitmap -> Picture
> stretchBitmap :: Point  -> Point            -> Bitmap -> Picture
> shearBitmap   :: Point  -> Point  -> Point  -> Bitmap -> Picture
>
> withBitmap             :: Bitmap -> Picture -> Picture
> createCompatibleBitmap :: Win32.HDC -> Int -> Int -> IO Bitmap
> withCompatibleBitmap   :: Int -> Int -> (Bitmap -> Picture) -> Picture
> withCompatibleDC       :: (Win32.HDC -> Picture) -> Picture
```

A.8 Module GraphicsRGN

```
> newtype Region = MkRegion Win32.HRGN
>
> mkEmpty     ::                     (Region -> Picture) -> Picture
> mkRectangle :: Point -> Point   -> (Region -> Picture) -> Picture
> mkEllipse   :: Point -> Point   -> (Region -> Picture) -> Picture
> mkPolygon   :: [Point]          -> (Region -> Picture) -> Picture
>
> andRegion   :: Region -> Region -> (Region -> Picture) -> Picture
> orRegion    :: Region -> Region -> (Region -> Picture) -> Picture
> xorRegion   :: Region -> Region -> (Region -> Picture) -> Picture
> diffRegion  :: Region -> Region -> (Region -> Picture) -> Picture
>
> region      :: Region -> Picture
```

A.9 Module GraphicsColor

```
> 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 -> Picture -> Picture
```

A.10 Module GraphicsPicture (again)

```
> -- These are defined in GraphicsPicture, but don't seem to belong there
> type DrawFun = Win32.HWND -> Win32.HDC -> IO ()
> drawPicture         :: Picture -> DrawFun
> drawBufferedPicture :: Picture -> DrawFun
```

A.11 Module Graphics

```
> -- Reexports everything exported by the following modules:
> --
> --   GraphicsPicture
> --   GraphicsWindows
> --   GraphicsUtils
```

A.12 Module GraphicsWindows

```
> data Window = ...
>
> data Event
>   = Key       { char :: Char, isDown :: Bool }
>   | Button    { pt :: Point, isLeft, isDown :: Bool }
>   | MouseMove { pt :: Point }
>   | Resize
>  deriving Show
>
> runGraphics  :: IO () -> IO ()
```

```
> openWindow   :: String -> Point -> IO Window
> openWindowEx :: String -> Maybe Point -> Maybe Point ->
>                 (Picture -> DrawFun) -> Maybe Int ->
>                 IO Window
> closeWindow  :: Window -> IO ()
> redraw       :: Window -> IO ()
> windowRect   :: Window -> IO (Point, Point)
> getPicture   :: Window -> IO Picture
> setPicture   :: Window -> Picture -> IO ()
> getEvent     :: Window -> IO Event
> getTick      :: Window -> IO ()
```

A.13 Module GraphicsUtils

```
> clearWindow  :: Window -> IO ()
> draw         :: Window -> Picture -> IO ()
> windowSize   :: Window -> IO Point
> getLBP       :: Window -> IO Point
> getRBP       :: Window -> IO Point
> getButton    :: Window -> Bool -> Bool -> IO Point
> getKey       :: Window -> IO Char
> getKeyEx     :: Window -> Bool -> IO Char
```