ChalkBoard Tutorial

Kevin Matlage and Andy Gill

Information Technology and Telecommunication Center
Department of Electrical Engineering and Computer Science
The University of Kansas
2335 Irving Hill Road
Lawrence, KS 66045
{kmatlage,andygill}@ittc.ku.edu

ChalkBoard Version 1.9.0.15

Released December 1st, 2009

This is a tutorial on how to use ChalkBoard, a domain specific language for describing and creating images being developed at the University of Kansas. The aim of this tutorial is to familiarize the user with some of the basic syntax and structure possibilities that can be used within ChalkBoard.

This is an early version of ChalkBoard. Everything might change! The primary concepts (functor based transformations of images, OpenGL acceleration) will remain, but we are still trying to balance and tune our observable sub-language. Applicative functors are also sure to follow, and much more experimentation is needed.

Please let us know if there is anything we can add, and/or give feedback. Thank you for looking at ChalkBoard.

Kevin Matlage, Andy Gill
Dec 2009

Contents

List of Figures

1 Installing ChalkBoard

Installing ChalkBoard is easy, using the `cabal install` command.

```
$ cabal install ChalkBoard
```

ChalkBoard depends on a number of Haskell packages, all available on Hackage. `cabal install ChalkBoard` should take care of this.

1.1 Possible Issues

We have tested ChalkBoard on Linux and OSX. Your milage may vary on Windows. Please tell us how you get on.

1.1.1 Mac OSX

One that might cause an issue is the DevIL library, `Codec-Image-DevIL`. We found the underlying C library, as loaded via the `port` OSX package manager, was unable to load .gif files properly. This may have been fixed.

2 Building A Standalone ChalkBoard Binary

To build your first ChalkBoard example, type in (or cut and paste) this example into a file `Example.hs`

Example.hs

```
import Graphics.ChalkBoard

main = startChalkBoard [] $ \ cb -> do
            -- ChalkBoard commands go here
            drawChalkBoard cb (boardOf blue)
```

Compiling this file with GHC gives a binary that uses ChalkBoard. We use the notation `$` to signify a command to be typed in.

```
$ ghc --make Example.hs
```

This binary can be executed.

```
$ ./Example
```

This brings up the ChalkBoard viewer on the screen, with the default size of 400x400, and a rather bland blue background, as shown in figure 1.

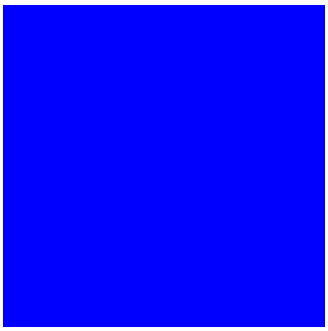

Figure 1: Blue ChalkBoard

This example gives the first flavor of ChalkBoard. We

- initialize a ChalkBoard viewer, giving us a ChalkBoard handle inside a scope,
- then issue a sequence of ChalkBoard commands, in this case a single `drawChalkBoard`, using the `cb` handle.

This completes the trivial example. In our next example, we introduce some basic shapes.

2.1 ChalkBoard and GHCi

For technical reasons, OpenGL (which ChalkBoard uses) and GHCi do not interact well. To mitigate this, we have provided the ChalkBoard server, which accepts commands from the GHCi command line. We discuss using this a special ChalkBoard Server section. All the commands that we present in this and the following sections can equally be used in stand alone or server modes; the only difference is in initialization.

3 ChalkBoard Examples

To begin, consider this example. This example may also be found in the cabal distribution, in `tutorial/basic/Main.hs`.

Main.hs

```
module Main where

import Graphics.ChalkBoard

main = startChalkBoard [] cbMain

cbMain cb = do
        let example1 = boardOf blue
        drawChalkBoard cb example1
```

First, notice that the only module you need to import to use ChalkBoard is the Graphics.ChalkBoard module. This module gives you all of the functions you need to use ChalkBoard.

The function to begin running ChalkBoard is `startChalkBoard`. This function takes two parameters. The first is a list of options, and the second is another driver function that takes a ChalkBoard handle. For now, we leave the list of options empty and pass our driver function `cbMain` for the second parameter, defining our main as:

```
main = startChalkBoard [] cbMain
```

Now, for this tutorial, `cbMain` has been defined a little bit unusually. For clarity, a lot of examples have been defined, where each example represents a Board of RGB, or `Board RGB` type. At the bottom of `cbMain` is the line of code that actually draws one of those boards. This line is:

```
drawChalkBoard chalkboard example1
```

This `drawChalkBoard` function takes in two parameters. The first one is the ChalkBoard handle `cb`, which is the only argument passed to `cbMain`. The second parameter is a board of type `Board RGB`. In order to display any of the examples in this tutorial, simply change which example board is being passed to `drawChalkBoard`.

The rest of this section will walk through the different examples, explaining how each of these different boards is created. This is mainly done to show the syntax and structure that can be used while defining images in ChalkBoard. For a little more insight on how to start creating more complicated boards, see some of the other ChalkBoard tutorials and tests.

3.1 Example 1 – Blue ChalkBoard

In example 1, we simply draw a board filled entirely with the color blue. In ChalkBoard, the `boardOf` function is kind of similar to `pure` in that it takes a color and creates an entire board of that color. Many common colors, like blue, are already defined in ChalkBoard and can be used by name. Others can easily be created by the user, but this will be described a little bit later.

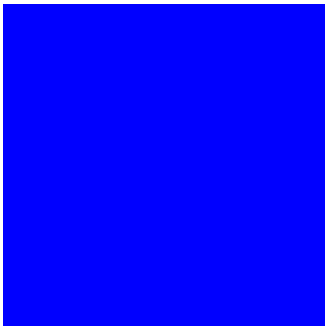

Figure 2: Blue ChalkBoard

```
example1 = boardOf blue
```

3.2 Example 2 - Red Square

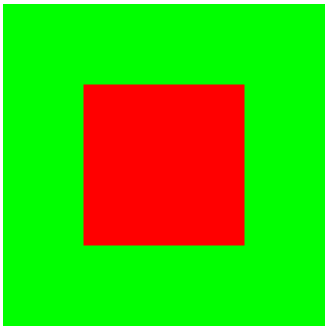

Figure 3: Red Square

In the second example, we start to introduce some of the basic functions of ChalkBoard. Starting on the right, `square` is a predefined region, or board of booleans. It's type, as you may expect, is `Board Bool`. The `choose` function is one of the most important functions in ChalkBoard. It takes two colors and a region (`Board Bool`) and maps the first color to the parts of the region that are true and the second color to the parts of the region that are false. In this instance, the color red is applied to the true parts of the board (the square) and green is applied to the false parts of the board (outside the square). Finally, the board is scaled by 0.5 in much the way that you would expect, reducing the sides of the square by a factor of 2 while still keeping it centered.

This final scaling step is taken here because `square` and the default backing board in ChalkBoard are both unit squares. While we can consider all the area outside of the inner square to be green, only a certain portion of this infinite board can be displayed. Because the default backing board in ChalkBoard is a unit square centered at the origin, the same as the region generated by `square`, we wouldn't be able to see any of this green portion without first scaling the image or changing the size of the default backing board.

```
example2 = scale 0.5 $ choose red green <$> square
```

3.3 Example 3 - Overlaying

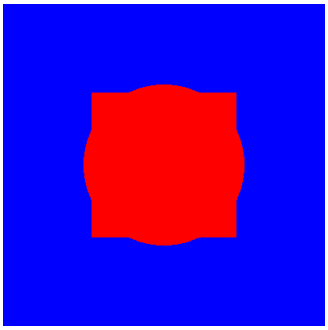

Figure 4: Overlaying

Example 3 is much the same as example 2, but with a few notable exceptions. The first is the addition of a new function, `over`. What this function does is simply overlay two boards of the same type. While this function works over boards of any type, it is used here over boards of bool. This way, when the choose function is applied, the region which is used is the resulting `Board Bool` that comes from combining the circle region with the scaled square region. This brings me to the second change worth noting, that the `scale` function is now also being used over the `square` region. This shows how `scale` can be applied not just to the final image, but also to boards of different types during many parts of the image specification process. In this particular instance, it keeps the square from encompassing the circle, allowing the resulting region to be a much more interesting shape.

```
example3 = scale 0.5 $ choose red blue <$> circle
                                           `over`
                                           (scale 0.9 square)
```

3.4 Example 4 - Alpha Triangle

Example 4 is another one that introduces a lot of new functions available to the user. The most basic of these functions is `triangle`, which creates a region (`Board Bool`) similar to `square`. The main difference with `triangle` is that it allows the user to specify the 3 points of the triangular region that is created. While all squares can be achieved by moving, rotating, and scaling another square, this is not true of triangles, and thus the vertices must be specified. Note that, while the default backing board in ChalkBoard is still a unit board, the points specified aren't touching the edges because the triangle is later scaled by 0.5.

Figure 5: Alpha Triangle

The next set of new functions all have to do with adding an alpha channel to our image. The `withAlpha` function adds this alpha channel to a normal RGB color with the specified value. In this example, an alpha of 0.2 is added to the RGB color `blue`, resulting in an RGBA value equivalent to (0 0 1 0.2). The `transparent` function is very similar, but adds a set alpha value of 0 to the color, making it entirely transparent. Because it is entirely transparent, it doesn't really matter what initial color is given, but in this case, the resulting RGBA value would be (1 1 1 0). Lastly, the `unAlpha` function is used to take the `Board RGBA` that comes after the `<$>` operator, and turn it into a `Board RGB` by removing the alpha channel. The image itself will not be changed by this operation. It will look exactly as it did when it had an alpha channel, but with the blending of colors already done so that it can be stored as RGB.

```
example4 = unAlpha <$> scale 0.5 ( choose (withAlpha 0.2 blue)
                                          (transparent white)
                                          <$> triangle (-0.5,-0.5)
                                                       (0.5,-0.5)
                                                       (0,0.5) )
```

3.5 Example 5 - Alpha Blending

In our fifth example, we show a little bit more how boards can start to be built up. The `Board RGBA` boards that are built up in the where clause are combined using the `over` operator. Because the `cir` board has a partially transparent circle on it, this will be visible over the polygon on the `poly` board. Again, `scale` is used on `cir` before it is combined so that the circle will not completely cover the polygon, allowing for a slightly more interesting combination.

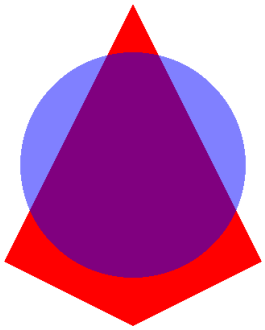

Figure 6: Alpha Blending

There are also a couple new functions used in this example that should be pointed out. The first of these is `polygon`, which takes in a list of points to use in creating an arbitrary polygon. This allows for general shapes to be created much more quickly and easily. It should be noted, however, that like OpenGL, the specified polygon must be convex in order to be guaranteed to display correctly. Concave polygons can be created using combinations of convex polygons. The other new function is `alpha`, which is used to turn an RGB color into an RGBA color with an alpha value of one. This would be the exact same as using the `withAlpha` function with an argument of 1, and in this instance will create an opaque red with RGBA value (1 0 0 1).

```
example5 = unAlpha <$> ((scale 0.7 cir) `over` poly)
               where
                   cir = choose (withAlpha 0.5 blue)
                                (transparent white)
                                <$> circle
                   poly = choose (alpha red)
                                 (transparent white)
                                 <$> polygon [(0,-0.5), (-0.4,-0.3),
                                              (0,0.5), (0.4,-0.3)]
```

3.6 Using Existing Images

The examples from this point forward will all be using an image that is read in from a file. The ChalkBoard command to create a board with an image from a file on it is `readBoard`. This is an IO function, however, and so therefore must be used outside of the let clauses that we have been using so far to specify individual boards. At the moment, this image is always read in as an `Board RGBA` strictly following the data in the file. In order to get the image board to be visible on the backing board, however, there is some scaling and moving that needs to be done. This is shown in the following segment of code, which can be used to read in an image board from the file lambda.png and fit it to the default backing board. After this segment of code is executed, `img` can be used as a `Board RGBA` in creating other boards.

```
(w,h,imgBrd) <- readBoard ("lambda.png")
let wh = fromIntegral $ max w h
    sc = 1 / wh
    wd = fromIntegral w / wh
    hd = fromIntegral h / wh
    img = move (-0.5 * hd,-0.5 * wd)  (scale sc imgBrd)
```

While this method allows for the most freedom, giving the user back the actual image board and the w and h dimensions of this board, it is not always the easiest to work with. In order to get back a board that has already been fit to the backing board, simply use the `readNormalizedBoard` function instead. This function does the exact operations listed above and returns the original w, h and `img` so that the board you get back is already fit to the ChalkBoard backing board. From there the image board can be additionally modified by the user, but starts in a much more usable state. An example of reading an image in with this method is:

```
(w2,h2,img2) <- readNormalizedBoard ("lambda.png")
```

3.7 Example 6 - Displaying Existing Images

Example 6 doesn't really contain any new information. It simply shows how one would go about using the image board that was read in from a file. Because the image is read into a `Board RGBA`, it just needs to have the alpha channel removed in order to be displayed.

Figure 7: Existing Image

```
example6 = unAlpha <$> img2
```

3.8 Example 7 - Image Overlaying

Example 7 is another fairly simple extension. It shows how one can overlay the image board above another `Board RGBA` as long as the image board contains transparency. Naturally, building up other RGBA boards and placing them over the image board would then draw on top of the image instead.

Figure 8: Image Overlaying

There is one other interesting change in this example, however, which is the ability to work with user-defined colors instead of just the built in colors. Creating a new color is simple, one just uses the `RGB` function with arguments of the red, green, and blue values for the new color. In this way, colors can be created in much the same way as in other graphics applications. One difference, however, is that in ChalkBoard everything must be observable in order to be compiled down into the ChalkBoard Intermediate Representation for OpenGL translation. Because of this, the `o` function must be used to lift the new color into this observable world, in much the same way that `return` works for monads.

```
example7 = unAlpha <$> img2 `over` (boardOf (alpha (o (RGB 0.5 1 0.8))))
```

3.9 Example 8 - Transformations

In example 8, the other transformation functions are introduced. These functions are `move` and `rotate`. Note that these functions, like `scale`, also work on any type of board. The arguments to `rotate` are the number of radians that you wish to rotate the board clockwise, and then the board itself. The `move` function takes an ordered pair for its first argument, which corresponds to the amount the board should be moved in the x and y directions, respectively. This movement is in relation to the ChalkBoard backing board and can therefore easily move things off the screen, if desired.

Figure 9: Transformations

In this particular instance, the image is scaled and then rotated before being placed over the pure board of red. The combined board is then moved to the right and down. Because the boards are thought to be infinite though and the entire background is red, more red just comes in from the top left and this move actually ends up having the same effect as it would have had if it was done prior to the boards being overlaid (in that only the image itself appears to move).

```
example8 = unAlpha <$> move (0.25, -0.25) ( (rotate 1 (scale 0.7 img2))
                                             `over`
                                             (boardOf (alpha red)) )
```

3.10 Examples 9 and 10

Examples 9 and 10 will not be discussed in depth in this tutorial. These examples don't present any new functionality, but rather show some basic extensions to what has already been given. They are included in order to hint at some of the simplest possibilities of using ChalkBoard to functionally create images and animations.

In order to display either of these examples, simply switch the last two lines in the tutorial's Main.hs file, commenting out the simple `drawChalkBoard` function and uncommenting the version of `drawChalkBoard` that uses list comprehensions. Then just compile and run the program like you usually would, with either example9 or example10 being called from this line. Note that while most of this tutorial has used 1 `drawChalkBoard` call, you can actually use as many as you want, as demonstrated by these last couple of examples.

Keep in mind that all of these examples are extremely simple. Even these last two still only use 1 line of code to create the board and 1 variable to change some of the features over the list comprehension. Once you start to build up examples that are a little bit more sophisticated, ChalkBoard clearly becomes capable of doing a lot more than has been demonstrated here. In order to see a couple examples that are at least a little bit more complicated, try checking out some of the test programs provided with ChalkBoard.

4 ChalkBoard `cabal` package

ChalkBoard is packaged with cabal, and is shipped with a number of tests which are disabled by default (if you want to just **use** the library, then you do not need to build these tests, obviously). The ChalkBoard server is always built.

There are a number of extra binaries provided inside the cabal distribution, most of them disabled by default. This page lists the various options for the cabal package, as well as listing the tests available.

4.1 Examples

There are two examples provided, `example` and `simple`.

```
$ cabal configure -fexample
```

`example`, in `examples/example`, gives a trivial example of spinning boxes.

```
$ cabal configure -fsimple
```

`simple`, in `examples/simple`, gives a small number of tests, and was use as material for this tutorial.

4.2 Tests

There are two test suits, one for the front end, and one for the backend.

```
$ cabal configure -ftest1 -fcbbe1
```

`test1`, in `tests/test1`, is our primary testing system. Typing

```
$ make test
```

inside `tests/test1` does a basic sanity check for ChalkBoard. `cbbe1` should only be used if you are developing ChalkBoard.

4.3 Benchmark

ChalkBoard ships with `chalkmark`, a basic timing test. At some point in the future it will output a number, the chalkmark. It lives in `tests/chalkmark`.

5 ChalkBoard Server

The server, called `chalkboard-server-1.9.0.15`, is found in the `server` directory. The binary name is version specific, and can not be mixed and matched with other ChalkBoard releases. Typically, its usage is transparent.

To revisit our original example, we can use the server (rather than a standalone binary) use

ServerExample.hs

```
import Graphics.ChalkBoard

main = do cb <- openChalkBoard []
          -- ChalkBoard commands go here
          drawChalkBoard cb (boardOf blue)
```

`openChalkBoard` takes the same options as `createChalkBoard`, but instead of opening up a OpenGL window (using GLUT), it spawns a child that has responsibility to open the window, and instead simply returns the ChalkBoard handle. `openChalkBoard` can be used from inside GHCi or GHC, and accepts exactly the same ChalkBoard language.

The only disadvantage is speed, because every ChalkBoard command needs to be serialized into bits, and piped to the server.

When using the server, is completely reasonable to have multiple instances of the server interacting with a single ChalkBoard client program.

6 ChalkBoard API

This is the API, as transliterated from haddock. You only need to `import Graphics.ChalkBoard`, which imports the following modules.

Note: This API might change at any time. ChalkBoard is experimental.

6.1 Graphics.ChalkBoard.Board

Contents

- The `Board`
- Ways of manipulating `Board`.
- Ways of creating a new `Board`.

6.1.1 Synopsis

```
data Board a
(<$>) :: (O a -> O b) -> Board a -> Board b
move :: (R, R) -> Board a -> Board a
rotate :: Radian -> Board a -> Board a
scaleXY :: (R, R) -> Board a -> Board a
boardOf :: O a -> Board a
circle :: Board Bool
box :: (Point, Point) -> Board Bool
square :: Board Bool
triangle :: Point -> Point -> Point -> Board Bool
polygon :: [Point] -> Board Bool
readBoard :: String -> IO (Int, Int, Board RGBA)
readNormalizedBoard :: String -> IO (Int, Int, Board RGBA)
```

6.1.2 The `Board`

data **Board** a

Instances

Show (Board a)
Scale (Board a)
Over a => Over (Board a)

6.1.3 Ways of manipulating `Board`.

(<$>) :: (O a -> O b) -> Board a -> Board b

`fmap` like operator over a `Board`.

move :: (R, R) -> Board a -> Board a

`move` moves the contents of `Board`

rotate :: Radian -> Board a -> Board a

`rotate` rotates a `Board` clockwise by a radian argument.

scaleXY :: (R, R) -> Board a -> Board a

`scaleXY` scales the contents of `Board` the X and Y dimension. See also `scale`.

6.1.4 Ways of creating a new `Board`.

boardOf :: O a -> Board a

pure like operator for `Board`.

circle :: Board Bool

Generate a unit circle (radius .5) centered on origin

box :: (Point, Point) -> Board Bool

`box` generate a box between two corner points)

square :: Board Bool

Generate a unit square (1 by 1 square) centered on origin

triangle :: Point -> Point -> Point -> Board Bool

Generate an arbitary triangle from 3 points.

polygon :: [Point] -> Board Bool

Generate a (convex) polygon from a list of points. There must be at least 3 points, and the points must form a convex polygon.

readBoard :: String -> IO (Int, Int, Board RGBA)

read a file containing a common image format (jpg, gif, etc.), and create a 'Board RGBA', and the X and Y size of the image.

readNormalizedBoard :: String -> IO (Int, Int, Board RGBA)

6.2 Graphics.ChalkBoard.O

Contents

6.2.1 Synopsis

```
data O o
class Obs a where
  o :: a -> O a
unO :: O o -> o
true :: O Bool
false :: O Bool
choose :: O o -> O o -> O Bool -> O o
alpha :: O RGB -> O RGBA
withAlpha :: UI -> O RGB -> O RGBA
unAlpha :: O RGBA -> O RGB
transparent :: O RGB -> O RGBA
red :: O RGB
green :: O RGB
blue :: O RGB
white :: O RGB
black :: O RGB
cyan :: O RGB
purple :: O RGB
yellow :: O RGB
```

6.2.2 The Observable datatype

data **O** o

Instances

Show o => Show (O o)

class **Obs** a where

Methods

o :: a -> O a

Instances

Obs Bool
Obs RGBA
Obs RGB

unO :: O o -> o

project into an unobservable version of O.

6.2.3 The Observable language

true :: O Bool

Observable `True`.

false :: O Bool

Observable `False`.

choose :: O o -> O o -> O Bool -> O o

choose between two Observable alternatives, based on a Observable `Bool`

alpha :: O RGB -> O RGBA

Observable function to add an alpha channel.

withAlpha :: UI -> O RGB -> O RGBA

Observable function to add a preset alpha channel.

unAlpha :: O RGBA -> O RGB

Observable function to remove the alpha channel.

transparent :: O RGB -> O RGBA

Observable function to add a transparent alpha channel.

red :: O RGB

green :: O RGB

blue :: O RGB

white :: O RGB

black :: O RGB

cyan :: O RGB

purple :: O RGB

yellow :: O RGB

6.3 Graphics.ChalkBoard.Shapes

6.3.1 Description

This module contains some basic shape generators, expressed as `Board Bool`.

6.3.2 Synopsis

```
straightLine :: (Point, Point) -> R -> Board Bool
pointsToLine :: [Point] -> R -> Board Bool
dotAt :: Point -> R -> Board Bool
functionLine :: (R -> Point) -> R -> Int -> Board Bool
```

6.3.3 Documentation

straightLine :: (Point, Point) -> R -> Board Bool

A straight line, of a given width, between two points.

pointsToLine :: [Point] -> R -> Board Bool

dotAt :: Point -> R -> Board Bool

place dot at this location, with given diameter.

functionLine :: (R -> Point) -> R -> Int -> Board Bool

A line generated by sampling a function from `R` to `Point`s, with a specific width. There needs to be at least 2 sample points.

6.4 Graphics.ChalkBoard.Types

Contents

6.4.1 Description

This module contains the types used by chalkboard, except Board itself.

6.4.2 Synopsis

```
type UI = R
type R = Float
type Point = (R, R)
type Radian = Float
class Over c where
  over :: c -> c -> c
stack :: Over c => [c] -> c
class Scale c where
  scale :: R -> c -> c
class Lerp a where
  lerp :: UI -> a -> a -> a
class Average a where
  average :: [a] -> a
nearZero :: R
type Gray = UI
data RGB = RGB !UI !UI !UI
data RGBA = RGBA !UI !UI !UI !UI
```

6.4.3 Basic types

type **UI** = R

Unit Interval: value between 0 and 1, inclusive.

type **R** = Float

A real number.

type **Point** = (R, R)

A point in R2.

type **Radian** = Float

Angle units

6.4.4 Overlaying

class **Over** c where

For placing a value literally *over* another value. The 2nd value *might* shine through. The operation *must* be associative.

Methods

over :: c -> c -> c

Instances

Over Bool
Over RGBA
Over RGB
Over Gray
Over (Maybe a)
Over a => Over (Board a)

stack :: Over c => [c] -> c

`stack` stacks a list of things over each other, where earlier elements are `over` later elements. Requires non empty lists, which can be satisfied by using an explicitly transparent `Board` as one of the elements.

6.4.5 Scaling

class **Scale** c where

`Scale` something by a value. scaling value can be bigger than 1.

Methods

scale :: R -> c -> c

Instances

Scale RGB
Scale R
Scale (Board a)
(Scale a, Scale b) => Scale ((,) a b)

6.4.6 Linear Interpolation

class **Lerp** a where

Linear interpolation between two values.

Methods

lerp :: UI -> a -> a -> a

Instances

Lerp RGB
Lerp R
Lerp a => Lerp (Maybe a)
(Lerp a, Lerp b) => Lerp ((,) a b)

6.4.7 Averaging

class **Average** a where

`Average` a set of values. weighting can be achived using multiple entries.
Methods
average :: [a] -> a
average is not defined for empty list
Instances
Average RGB
Average R
(Average a, Average b) => Average ((,) a b)

6.4.8 Constants

nearZero :: R

Close to zero; needed for `Over (Alpha c)` instance.

6.4.9 Colors

type **Gray** = UI

`Gray` is just a value between 0 and 1, inclusive. Be careful to consider if this is pre or post gamma.

data **RGB**

`RGB` is our color, with values between 0 and 1, inclusive.
Constructors
RGB !UI !UI !UI
Instances
Show RGB
Binary RGB
Average RGB
Lerp RGB
Scale RGB
Over RGB
Obs RGB

data **RGBA**

`RGBA` is our color, with values between 0 and 1, inclusive. These values are *not* prenormalized

Constructors

RGBA !UI !UI !UI !UI

Instances

Show RGBA
Binary RGBA
Over RGBA
Obs RGBA

6.5 Graphics.ChalkBoard.Main

6.5.1 Synopsis

```
data ChalkBoard
drawChalkBoard :: ChalkBoard -> Board RGB -> IO ()
writeChalkBoard :: ChalkBoard -> FilePath -> IO ()
updateChalkBoard :: ChalkBoard -> (Board RGB -> Board RGB) -> IO ()
drawRawChalkBoard :: ChalkBoard -> [Inst BufferId] -> IO ()
exitChalkBoard :: ChalkBoard -> IO ()
startChalkBoard :: [Options] -> (ChalkBoard -> IO ()) -> IO ()
openChalkBoard :: [Options] -> IO ChalkBoard
chalkBoardServer :: IO ()
```

6.5.2 Documentation

data **ChalkBoard**

drawChalkBoard :: ChalkBoard -> Board RGB -> IO ()

Draw a board onto the ChalkBoard.

writeChalkBoard :: ChalkBoard -> FilePath -> IO ()

Write the contents of a ChalkBoard into a File.

updateChalkBoard :: ChalkBoard -> (Board RGB -> Board RGB) -> IO ()

modify the current ChalkBoard.

drawRawChalkBoard :: ChalkBoard -> [Inst BufferId] -> IO ()

Debugging hook for writing raw CBIR code.

exitChalkBoard :: ChalkBoard -> IO ()

quit ChalkBoard.

startChalkBoard :: [Options] -> (ChalkBoard -> IO ()) -> IO ()

Start, in this process, a ChalkBoard window, and run some commands on it.

openChalkBoard :: [Options] -> IO ChalkBoard

Open, remotely, a ChalkBoard windown, and return a handle to it. Needs CHALKBOARD_SERVER set to the location of the ChalkBoard server.

chalkBoardServer :: IO ()

create an instance of the ChalkBoard. Only used by the server binary.

6.6 Graphics.ChalkBoard.Utils

Contents

- Point Utilties.
- Utilties for `R`.

6.6.1 Description

This module has some basic, externally visable, definitions.

6.6.2 Synopsis

```
insideRegion :: (Point, Point) -> Point -> Bool
insideCircle :: Point -> R -> Point -> Bool
distance :: Point -> Point -> R
intervalOnLine :: (Point, Point) -> Point -> R
circleOfDots :: Int -> [Point]
insidePoly :: [Point] -> Point -> Bool
innerSteps :: Int -> [R]
outerSteps :: Int -> [R]
fracPart :: R -> R
fromPolar :: (R, Radian) -> Point
toPolar :: Point -> (R, Radian)
angleOfLine :: (Point, Point) -> Radian
```

6.6.3 Point Utilties.

insideRegion :: (Point, Point) -> Point -> Bool

is a `Point` inside a region?

insideCircle :: Point -> R -> Point -> Bool

is a `Point` inside a circle, where the first two arguments are the center of the circle, and the radius.

distance :: Point -> Point -> R

What is the `distance` between two points in R2? This is optimised for the normal form `distance p1 p2 <= v`, which avoids using `sqrt`.

intervalOnLine :: (Point, Point) -> Point -> R

`intervalOnLine` find the place on a line (between 0 and 1) that is closest to the given point.

circleOfDots :: Int -> [Point]

circleOfDots generates a set of points between (-1..1,-1..1), inside a circle.

insidePoly :: [Point] -> Point -> Bool

6.6.4 Utilties for `R`.

innerSteps :: Int -> [R]

innerSteps takes n even steps from 0 .. 1, by not actually touching 0 or 1. The first and last step are 1/2 the size of the others, so that repeated innerSteps can be tiled neatly.

outerSteps :: Int -> [R]

outerSteps takes n even steps from 0 .. 1, starting with 0, and ending with 1, returning n+1 elements.

fracPart :: R -> R

Extract the fractional part of an `R`.

fromPolar :: (R, Radian) -> Point

toPolar :: Point -> (R, Radian)

angleOfLine :: (Point, Point) -> Radian

6.7 Graphics.ChalkBoard.Options

6.7.1 Documentation

data **Options**

Constructors

NoFBO

DebugFrames

DebugAll not supported (yet!)

DebugBoards [BufferId] not supported (yet!)

BoardSize Int Int default is 400x400.

FullScreen not supported (yet!)

Instances

Eq Options

Show Options

Binary Options