Róisín Ní Bhriain - 18326577

Image Server

Basic Shapes - Done

I implemented all the basic shapes. The rectangle was an implementation of the square function with some extra division to make the edges different lengths.

```
maxnorm1 :: Point0 -> Double
maxnorm1 (Vector0 x y) = max (abs x / 0.7)  (abs y/ 0.2)
```

The ellipse was an implementation of the circle function with some extra division similar to the rectangle.

```
distance1 :: Point0 -> Double
distance1 (Vector0 x y ) = sqrt ( x**2/0.5**2 + y**2/0.3**2 )
```

The polygon was the toughest, after extensive research, I decided to use a library with a function to see if a point is inside the polygon and a function to ensure the polygon is convex.

```
import Data.Geometry
import Data.Geometry.Polygon
import Data.Geometry.Polygon.Convex
import Data.Ext
```

I used the insidePolygon to see if the current point was in the polygon and isConvex to ensure the polygon was convex.

```
(Vector0 l r) `insides` Polygon = insidePolygon (Point2 l r) simplePoly && isConvex
simplePoly

simplePoly :: SimplePolygon ( ) Double
simplePoly = fromPoints $ map ext $
                        [ Point2 0.0 0.25
                        , Point2 0 0.75
                        , Point2 0.25 1
                        , Point2 0.5 1
                        , Point2 0.75 0.75
                        , Point2 0.75 0.25
                        , Point2 0.5 0
                        , Point2 0.25 0
                        ]
```

Róisín Ní Bhriain - 18326577

Affine Transformations - Done

The affine transformations weren't too tough, the only one I had to implement myself was the shear function. I decided to do a shear in the x.

```
transform (Shear m) (Vector0 tx ty) = Vector0 (m*ty + tx)  ty
```

Colours - Done

I added a Colour of three Pixel8 to the tuple for drawings.

```
type Drawing = [(Transform,Shape,Colour)]

type Colour = [(Pixel8,Pixel8,Pixel8)]
```

I then used this in the render where instead of colouring each of the RGB pixels with 255 I used the colour provided by the drawing. This was fairly simple.

```
pixRenderer x y = PixelRGB8 r g b where  [(r,g,b)] = (colorForImage $ mapPoint win (x,y))
```

Masking Images - Done

I did both types of masking. I implemented a version to combine both drawings and get a mixed-colour combination. I also implemented a mask version where the second drawing would indicate which parts of the first drawing should be implemented.

I used an averaging function to mix the colours of the two images that were to be combined. This uses the combine function rather than the render function.

```
mixColours :: Drawing -> Drawing -> Colour
mixColours [(_,_,c)] [(_,_,c1)] = averageColours c c1


averageColours :: Colour -> Colour -> Colour
averageColours [(r,g,b)] [(r1,g1,b1)] = [(r+1`div`2,g+g1`div`2,b+b1`div`2)]
```

For masking the first image with the second image I simply changed the colorForImage function slightly. This uses the mask function rather than the render function.

```
colorForImage :: Point0 -> Colour
colorForImage p | p `inside` sh && p `inside` th  =p `colour` sh
                | otherwise     = [(0,0,0)]
```

Róisín Ní Bhriain - 18326577

Scotty Application - Done

My application has a main page with all the DL programs for each of the shapes and then pages with the images.

```
scotty 3000 $ do
get "/" $ do
    html  response
get "/circle" $ do
       setHeader "Content-Type" "image/png"
       file "./circle.png"
get "/combining" $ do
       setHeader "Content-Type" "image/png"
       file "./combining.png"
get "/square" $ do
        setHeader "Content-Type" "image/png"
        file "./square.png"
get "/mandlebrot" $ do
        setHeader "Content-Type" "image/png"
        file "./mandlebrot.png"
get "/ellipse" $ do
        html "hello"
        setHeader "Content-Type" "image/png"
        file "./ellipse.png"
get "/rectangle" $ do
        setHeader "Content-Type" "image/png"
        file "./rectangle.png"
get "/polygon" $ do
        setHeader "Content-Type" "image/png"
        file "./polygon.png"
get "/masking" $ do
        setHeader "Content-Type" "image/png"
        file "./masking.png"
```

Optimisation

The optimisation I did was merging transformations that are the same if they are consecutive. I chose to do this by making a separate merge for each of the transformations if they were the same.

```
transform (Compose ( Translate (Vector0 tx ty))( Translate (Vector0 tx1 ty1)) ) p =
transform (translate (Vector0 (tx + tx1) (ty + ty1))) p
transform (Compose ( Scale (Vector0 tx ty))( Scale (Vector0 tx1 ty1)) ) p = transform (scale
(Vector0 (tx + tx1) (ty + ty1))) p
transform (Compose ( Shear m)( Shear m1) ) p = transform (shear (m+m1)) p
transform (Compose ( Rotate m)( Rotate m1) ) p = transform (Rotate (matrixAdd m m1)) p
transform (Compose t1 t2) p = transform t2 $ transform t1 p
```

Reflection

Some of the design choices I made such as using a library for rendering the polygon made the process a lot easier. My method for combining and masking images required a slight change of the render function which was pretty easy. I found it tough to get the optimisation done with the merging transformations. I tried to figure out to get the merging of transformations done so that it would work for the same transformations even if they weren't in consecutive order. When making the combined drawings I had to change the render function a bit so which was frustrating as I had three very similar functions. There would be no need for the render function and instead, the scalable vector graphics can return the points that just need to be rendered.