CSU44012 Assignment 1

For this project, we were tasked with combining a drawing eDSL with web eDSLs to create a web application that displays custom-generated drawings. For the drawing eDSL we used JuicyPixels, and for the web eDSLs we used Scotty and Blaze.

1. Drawing eDSL

1.1 Shapes

I based my initial drawing eDSL code off of the Mandelbrot3 and ShapeExample2 examples. With this, I was able to generate simple shapes in black and white. To create extra shapes, I first took the existing shapes Circle and Square and modified them. I created a new shape Rect, which took in a width and height. To draw this, I simply drew and scaled a Square to the width and height. The same was done to create the Ellipse. Due to this, the Rect and Ellipse became (Transform, Shape) tuples instead of just Shapes. Therefore, to add more Transforms to these Shapes, I provided a new infix function `<++>` to Compose a (Transform, Shape) tuple with a Transform.

1.1.1 Polygons

To create a polygon, I started by making a new Polygon shape which took in a list of points. For all other shapes, I needed to be able to define whether or not a particular point was in that shape. Since Rect and Ellipse extend Square and Circle respectively, they didn't need new checks, but the Polygon shape did. The “Point in Polygon” problem has many solutions, some of them for specific cases and others for more general ones, but since we were told that the polygon had to be a closed convex polygon with points in clockwise order, I chose a more general solution.

My solution is an implementation of Brian Adkins’s code for point in polygon detection in Haskell[1]. In his code, he uses a variant of the dot product (the perp dot product[2]) of two vertices to calculate the normal of each edge in the polygon, where an edge is two adjacent points in the list. If the normal is greater than or equal to 0, the point is on the left (or below), and vice versa.[3]

Since we want the polygon to be clockwise, we use the opposite of this and determine a point to be in a polygon if the normal is less than or equal to 0. Checking for 0 means that the point is considered to be in a polygon if it lies on one of its edges. I also took the cross product function from Adkin’s code and replaced the pre-existing code, as it wouldn’t work with polygon detection. Finally, I added this new Polygon shape to the Shape type and the `pointInPoly` function to `insides` for shape detection.

1.2 Transforms

Scaling, rotation, and translation were all already provided by both ShapeExample2 and Mandelbrot3, so I only had to implement shearing myself.

The algorithm for shearing is largely used for matrices (affine transformation), so it had to be re-written for vectors. The formula is:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x + my \\ nx + y \end{pmatrix} = \begin{pmatrix} 1 & m \\ n & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

which can be implemented using linear transformations (vectors).

I also added a function `deg2Rad`, which converted degrees into radians, the angle format the rotation function used.

1.3 ColourShapes

Next, I worked on providing colours for each shape. The pre-existing code in ShapeExample2 had code for colouring a shape completely white, which wasn't enough in terms of variety.

I began by defining a Colour type, which was a tuple of Color8s, and a ColourShape, which was a tuple of (Shape, Colour). I then redefined Drawing to be a list of (Transform, ColourShape) tuples and modified the rest of the code. The function `inside1` now returns a Colour instead of a Bool, since the Colour (0, 0, 0) does not display. I took the `colourAtPoint` function from Mandelbrot3 (originally `insidecolour`) and modified it to return a Colour tuple instead of just an Int.

Then, in Render.hs, the `colorForImage` function was modified to use the colour given to the point by the Drawing. The pixRenderer function also flipped the area of the image I intended to colour on the x-axis, so I modified the `defaultWindow` function to use regular Cartesian coordinates.

Now, to draw a shape, you had to specify both the Shape and its parameters, and the Colour you wanted to use with the shape.

1.4 Masks

A mask creates a new shape by returning the parts of two ColourShapes that intersect. I added an ImageMask type which took in two Drawings (a list of (Transform, ColourShape) tuples) and returned a new Shape (not a ColourShape). The `insides` check for this shape simply checked the overlapping pixels of either drawing. To make it a valid ColourShape, I can just add a Colour tuple at the end.

2. Scotty

The Scotty webpage used code from the original Scotty example, with modifications.

Displaying the webpage was simple enough. I used multiple route patterns to add webpages the user could be taken to. On these webpages, I displayed the generated image next to the code used to generate it. I used style attributes to spruce things up a bit, and kept the code in a separate container so the user wouldn't have to scroll down the entire page.

To load the images from my computer, I used the wai-middleware-static library. This library serves static files based on filters, which I would supply. The only thing I needed from the library was to add the files I generated to the webpage. The generated files were sent to the "static" folder, so I used the middleware library to add that folder as a "base" – that is, I added a route between my computer and the webpage to display the images stored locally.

3. Optimisations

One minor optimisation I made was changing Compose to add values of adjacent transformations instead of doing them recursively.

If I had more time, I would consider using matrices to their full extent and use affine transformations everywhere instead of just rotation. Affine transformations can combine arbitrary Transforms more efficiently than linear transformations, which need recursion to achieve the same result. My current optimization only makes sense if I, for whatever reason, decided to place two of the same Transform next to each other instead of just combining them immediately.

I would also further attempt to implement the `pointInPoly` function in `O(log(N))` time. I found a paper[4] describing it in C++ and tried to implement it, but couldn't quite manage it and stayed with the original implementation by Adkin. While it is complicated, I feel like I could eventually implement this given enough time with my current ability with Haskell.

Reflection

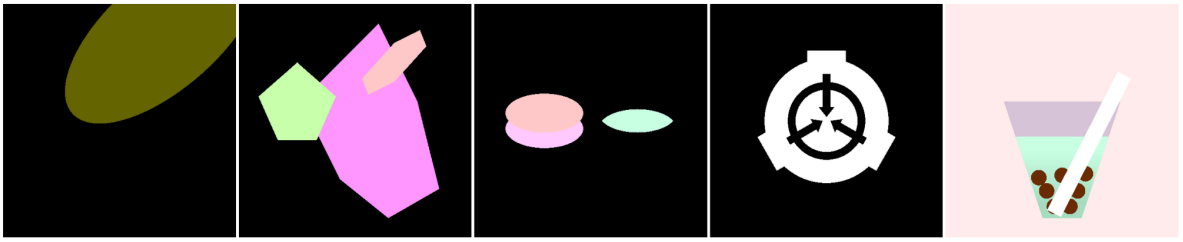
Overall, I enjoyed this project. I found implementing Scotty and Blaze very easy given the template I started from, so I spent a bit more time adding CSS to the webpage. This time could have been spent adding more optimisations, but I had spent a lot of time struggling with that, so I left it alone.

The choices I made in regards to the design of the shapes were difficult to figure out, but simple in hindsight. Creating a separate ColourShape instead of modifying the existing Shape type didn't occur to me until I realised I had to change the way I implemented the Rect and Ellipse. Because of this, you can use a Square or Circle to return a ColourShape, which I do in one of my examples.

Blaze SVGs seem to work very similarly to my current implementation. I wouldn't have to change much if I decided to use SVGs, other than the fact that you can close a polygon using the `closepath` function, so I wouldn't have to specify the final point when drawing.

Pictures

CSU44012 Assignment 1



Shear Example | Polygon Example | Mask Example | Complex Example 1 | Complex Example 2

Iris Onwa - 20332400

You chose Image 4

Complex Example 1

Code used to generate this:

```
-- arrows
-- -- arrow heads
(translate (point 0 0.05) <+> rotate (deg2Rad 90) <+> scale (point 0.1 0.1), (polygon [
    point 0 0,
    point 1 (-1.25),
    point (-1) (-1.25),
    point 0 0
], (1, 1, 1))),

(translate (point (-0.05) 0) <+> rotate (deg2Rad 210) <+> scale (point 0.1 0.1), (polygon [
    point 0 0,
    point 1 (-1.25),
    point (-1) (-1.25),
    point 0 0
], (1, 1, 1))),

(translate (point 0.05 0) <+> rotate (deg2Rad 330) <+> scale (point 0.1 0.1), (polygon [
    point 0 0,
    point 1 (-1.25),
    point (-1) (-1.25),
    point 0 0
], (1, 1, 1))),

-- -- arrow bodies
translate (point (-0.3) (-0.15)) <+> rotate (deg2Rad 30) <+> scale (point 0.05 0.05) <++> rect 1 4.5 (1, 1, 1),
translate (point 0.3 (-0.15)) <+> rotate (deg2Rad (-30)) <+> scale (point 0.05 0.05) <++> rect 1 4.5 (1, 1, 1),
translate (point 0 0.375) <+> scale (point 0.05 0.05) <++> rect 1 4.5 (1, 1, 1),

-- inner circle
```

Go back

You chose Image 5

Complex Example 2

Go back

```
translate (point (-0.1) (-1.1)) <++> ellipse 0.1 0.1 (107, 43, 0),
translate (point 0.1 (-1.1)) <++> ellipse 0.1 0.1 (107, 43, 0),
-- drink
{- mask gradients for drink -}
    (translate (point 0 0), (imageMask (translate (point 0 (-0.30)) <++> rect 1 0.1 (1, 1, 1)) (translate (point 0.25 0.25) <+> rotate
(deg2Rad 180) <+> scale (point 0.5 0.5), (polygon [ point 0 0, point 0 1, point 3 0, point 0 0], (215, 195, 215))), (200, 255, 227))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.30)) <++> rect 1 0.1 (1, 1, 1)) (translate (point (-0.25) 0.25) <+> rotate
(deg2Rad 180) <+> scale (point 0.5 (-0.5)), (polygon [ point 0 0, point 0 1, point 3 0, point 0 0], (215, 195, 215))), (200, 255, 227))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.30)) <++> rect 1 0.1 (1, 1, 1)) (translate (point 0 (-0.5)) <++> rect 0.25
0.75 (215, 195, 215)), (200, 255, 227))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.32)) <++> rect 1 0.1 (1, 1, 1)) (translate (point 0.25 0.25) <+> rotate
(deg2Rad 180) <+> scale (point 0.5 0.5), (polygon [ point 0 0, point 0 1, point 3 0, point 0 0], (215, 195, 215))), (199, 254, 226))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.32)) <++> rect 1 0.1 (1, 1, 1)) (translate (point (-0.25) 0.25) <+> rotate
(deg2Rad 180) <+> scale (point 0.5 (-0.5)), (polygon [ point 0 0, point 0 1, point 3 0, point 0 0], (215, 195, 215))), (199, 254, 226))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.32)) <++> rect 1 0.1 (1, 1, 1)) (translate (point 0 (-0.5)) <++> rect 0.25
0.75 (215, 195, 215)), (199, 254, 226))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.34)) <++> rect 1 0.1 (1, 1, 1)) (translate (point 0.25 0.25) <+> rotate
(deg2Rad 180) <+> scale (point 0.5 0.5), (polygon [ point 0 0, point 0 1, point 3 0, point 0 0], (215, 195, 215))), (198, 253, 225))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.34)) <++> rect 1 0.1 (1, 1, 1)) (translate (point (-0.25) 0.25) <+> rotate
(deg2Rad 180) <+> scale (point 0.5 (-0.5)), (polygon [ point 0 0, point 0 1, point 3 0, point 0 0], (215, 195, 215))), (198, 253, 225))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.34)) <++> rect 1 0.1 (1, 1, 1)) (translate (point 0 (-0.5)) <++> rect 0.25
0.75 (215, 195, 215)), (198, 253, 225))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.36)) <++> rect 1 0.1 (1, 1, 1)) (translate (point 0.25 0.25) <+> rotate
(deg2Rad 180) <+> scale (point 0.5 0.5), (polygon [ point 0 0, point 0 1, point 3 0, point 0 0], (215, 195, 215))), (197, 252, 224))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.36)) <++> rect 1 0.1 (1, 1, 1)) (translate (point (-0.25) 0.25) <+> rotate
(deg2Rad 180) <+> scale (point 0.5 (-0.5)), (polygon [ point 0 0, point 0 1, point 3 0, point 0 0], (215, 195, 215))), (197, 252, 224))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.36)) <++> rect 1 0.1 (1, 1, 1)) (translate (point 0 (-0.5)) <++> rect 0.25
0.75 (215, 195, 215)), (197, 252, 224))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.38)) <++> rect 1 0.1 (1, 1, 1)) (translate (point 0.25 0.25) <+> rotate
(deg2Rad 180) <+> scale (point 0.5 0.5), (polygon [ point 0 0, point 0 1, point 3 0, point 0 0], (215, 195, 215))), (196, 251, 223))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.38)) <++> rect 1 0.1 (1, 1, 1)) (translate (point (-0.25) 0.25) <+> rotate
(deg2Rad 180) <+> scale (point 0.5 (-0.5)), (polygon [ point 0 0, point 0 1, point 3 0, point 0 0], (215, 195, 215))), (196, 251, 223))),
    (translate (point 0 0), (imageMask (translate (point 0 (-0.38)) <++> rect 1 0.1 (1, 1, 1)) (translate (point 0 (-0.5)) <++> rect 0.25
```

References

[1]: *Point in convex polygon detection in Haskell*, Brian Adkin
https://gist.github.com/lojic/6734952
[2]: Weisstein, Eric W. "Perp Dot Product." From *MathWorld*--A Wolfram Web Resource.
https://mathworld.wolfram.com/PerpDotProduct.html
[3]: *Game Math: "Cross Product" of 2D Vectors*, Allen Chou
https://allenchou.net/2013/07/cross-product-of-2d-vectors/
[4]: Check if point belongs to the convex polygon in O(log N)
https://cp-algorithms.com/geometry/point-in-convex-polygon.html