Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU44012 Topics in Functional Programming

Project 1 - Image Server

Prathamesh Sai
4th year Integrated Computer Science
Student ID: 19314123

School of Computer Science and Statistics,
Trinity College Dublin

Contents

1 Introduction

This report covers the design decisions that I made while completing the image server (project 1) for CSU44012. There are sections for each of the project tasks, and also a reflection at the end. All the design choices are spread out over this report, and are located under the specific project task that they fall under. All the project deliverables have been satisfactorily completed, and are explained at depth below. To run the project, use the bash script *render.sh*. You can follow this video [1] that I've made to see a project demo.

```
bash render.sh
```

2 Designing a suitable drawing eDSL

To design a suitable drawing eDSL, I had to satisfy a number of requirements which are listed below.

2.1 Providing basic shapes

To provide basic shapes, I extended the Shape.hs file from the week 5 weekly assignment. I did this because I was comfortable using it from before, and I found it easy to use and understand. The shapes do not need any size or location for them when created, as it defaults to a size of 1 unit located at the origin (0,0). This means that the shapes can be declared with just their name which was nice. To specify a size or location, we can use transformations. I implemented the circle shape using the code provided by Glenn Strong in the week 5 assignment.

```
p `insides` Circle = distance p <= 1
distance :: Point -> Double
distance (Vector x y) = sqrt ( x**2 + y**2 )
```

With the ellipse, I use a hard coded ratio (which is stated as a fraction of 16/9) to change the width of the shape and eventually make it into an ellipse (using the equation $(x^2/ratio^2) + (y^2/1^2)$).

```
Vector x y `insides` Ellipse = (x**2/(16/9)**2) + (y**2/1**2) <= 1
```

Similarly with a rectangle, I use a hard coded ratio of 16 : 9 to change the width of the shape and eventually make it into a rectangle.

```
Vector x y `insides` Rectangle = (sqrt (y**2) <= 1) && (sqrt (x**2) <=
(16/9))
```

The polygon is instantiated with a series of points listed in clockwise order. The user can simply make the shape by listing "polygon" and a list of points straight after it. To do this, I found the "halfLine" function from the week 4 slides [2] very helpful in implementing a list of points that would help carry out this functionality.

```
Vector x y `insides` Polygon (first:(second:[])) =
                                halfLine (Vector x y) first second
Vector x y `insides` Polygon (first:(second:rest)) =
                    if (halfLine (Vector x y) first second == False)
                        then False
                        else insides ((Vector x y)) (Polygon (second:rest))
-- Function below is from lecture slide 5 from Week 4 - Designing an eDSL
(04.02 - Geometric regions and shapes)
halfLine :: (Vector) -> (Vector) -> (Vector) -> Bool
-- We subtract the x from x1, and y from y1, then subtract x1 from x2 and
y1 from y2.
halfLine (Vector x y) (Vector x1 y1) (Vector x2 y2) =
    zcross (Vector (x1-x) (y1-y)) (Vector (x2-x1) (y2-y1)) < 0
    where zcross (Vector a b) (Vector c d) = a*d - b*c
```

Finally, I added the mandelbrot as an additional shape as I had implemented it already from the week 5 weekly assignment, and thought it would be interesting to include it with more trivial shapes. The functionality of this was inspired by the code from the research paper "Composing Fractals" by Mark P. Jones [3].

2.2 Providing basic affine transformations

To provide basic affine transformations, I used the transformations for scale, rotate and transform from the week 5 assignment. For implementing shear, I used the affine transformations webpage from the Algorithm Archive website [4] which was extremely helpful. To use shear, I implemented it such that you have to pass in 2 parameters. Firstly, the amount that you want to shear in the x-axis, and secondly the amount that you want to shear in the y-axis. I felt that this would be intuitive to the user, and also easy to use rather than having separate shearX and shearY transformations. To implement the shear functionality, I gathered inspiration from the rotate transformation which had logic for the transformation matrix as "$(cos\ angle)(-sin\ angle)(sin\ angle)(cos\ angle)$".

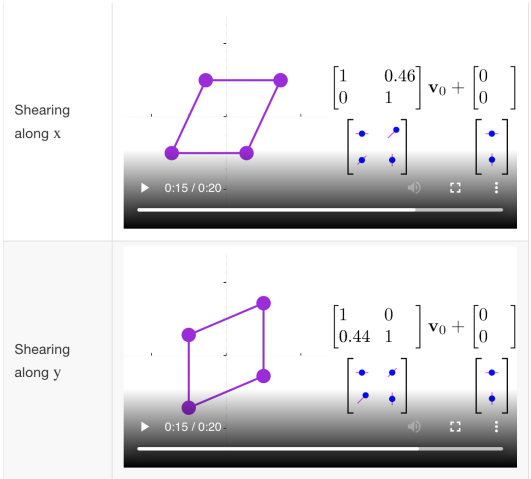

Figure 1: The transformation matrix changing while shearing in the x and y direction [4].

As seen above when shearing along x, the only thing that changes is row 1 column 2. Hence the logic for the transformation matrix for shearX was (1) $(-sin\ angle)(0)(1)$. When shearing along y, the only thing that changes is row 2 column 1. Hence the logic for the transformation matrix for shearY was $(1)(0)(sin\ angle)(1)$. For the final shear transformation, it simply uses both shearX and shearY definitions depending on the 2 values passed into it.

2.3 Providing a way to specify colours for shapes

To provide a way to specify colours for shapes, you add a color to the tuple (transformation, shape) to end up with (transformation, shape, colour). This made the most sense to me, as it is clear to any user that a drawing can consist of a shape which has a transformation and a colour applied to it. I implemented colours as a datatype. I ended up making logic to abstract-away the specifics of colours so that a user could simply state "orange" at the end of the tuple, hence simplifying the DSL (a sort of 'syntactic sugar'). I used Word8 types for the rgba values for the colours as they are 8 bits long (they allow values from 0-255) which was perfect for specifying numbers for rgba.

2.4 Providing a way to mask images

To provide a way to mask images, my maskImages function allows users to mask 2 images together, and gives the user the capability of deciding how much of the masked image they want to see. The maskImages function takes a Word8 (0-255) to let the user specify the opacity of the image being masked over another, and 2 coloured drawings to mask. I felt that this gave the user more flexibility to change how much of the masked image they wanted to see (if at all). This was great as the user could specify the opacity of the masked image and not simply a boolean of showing the masked image (255) or not (0).

3 Developing a Scotty application to render sample images

I had to develop a Scotty application that renders some hard coded images that demonstrate the features mentioned in the previous section. The images are rendered using JuicyPixels [5] and returned as a PNG. As mentioned before, the commands to run the project are inside the script *render.sh*, which can be ran using "*bash render.sh*". Under each image, I include the code for producing the image so the user of the web app can see how the image was made. I used Scotty [6] and Blaze [7] to make multiple pages that contain different images. How the application works is that the images are rendered first when the project is run. After these are rendered, they are stored in an internal "*/img*" directory. When the user clicks a button on the homepage to see a particular image, it directs them to another web page which contains the image (that is accessed by the app by making a call to localhost:3000/image-name [where image-name is the image in

question] which gets it from the */img* directory) with code under it. I also used CSS to make the webpage look a bit nicer, and stored it in a function that returns it in CSS.hs. I then hard-coded code for images at the top of Main.hs. These pages contain those images:

3.1 Page 1: Home page

This page contains text and buttons leading to all the different pages.

3.2 Page 2: Showing all shapes with colours

This page contains 5 images with text under each showing all the types of shapes available.

3.3 Page 3: Showing affine transformations

This page contains 1 image showing multiple shapes fitted into the same image with the use of all the transformations available.

3.4 Page 4: Showing overlapping shapes

This page contains 1 image showing multiple shapes that overlap over each other, which re-iterates to the user that the shape that is instantiated last is the shape that will overlap the shapes instantiated before it.

3.5 Page 5: Showing simple masking

This page contains 5 images. Image 1 is a black circle, Image 2 is a yellow rectangle, and Image 3 shows "fake masking" where one could simply overlap Image 1 and Image 2 so the rectangle is on top of the circle. Image 4 and 5 show real masking taking place, where the rectangle is masked over the circle.

3.6 Page 6: Showing more complex masking

This page contains 5 images showcasing the masking of images from Page 3 and Page 4 mentioned in this report while decreasing the opacity to let the user see how the maskImages function works.

4 Optimisations to the DSL program

I had to complete at least one optimisation to the DSL program that is run before it is rendered. I added 5 optimisations that consist of optimising the transformations before images are rendered.

4.1 Optimisation #1

If a user is shearing and passes in a value of 0 for shearing in the x or y direction (or both), that will be an extra function call that will have no difference in the result. To optimise this, we simply check if any parameters are 0, and avoid the extra function call. If both are 0, we apply the identity transformation which does nothing.

4.2 Optimisation #2

If a user does Translate (Vector 5 5) < + > Translate (Vector 5 5), that is equivalent to Translate (Vector 10 10). So, instead of having 2 separate transformations, we can simply get the summation of the x and y in the Vectors, and have one function call.

4.3 Optimisation #3

If a user does Scale (Vector 2 2) < + > Scale (Vector 2 2), that is equivalent to Scale (Vector 4 4). Once again, instead of having 2 separate transformations, we can simply get the summation of the x and y in the Vectors, and have one function call.

4.4 Optimisation #4

If a user does Rotate 45 < + > Rotate 45, that is equivalent to Rotate 90. Instead of having 2 separate transformations, we can simply get the summation of the angles and have one function call.

4.5 Optimisation #5

When masking, if a user passes in a Word8 value of 0 into the mask, this increases the opacity of the masked image so much that it makes it invisible. This means that we can simply return the original image without the other image that was meant to be masked over it, instead of computing anything.

5 Reflection

5.1 How challenging were the design choices?

I found the process of coming up with the design choices enjoyable but challenging at times. In particular, I found making the design choice for how to implement colours quite challenging. I also found the design choices to implement the shear transformation quite challenging too. Finally, I found the design choices to implement the masking quite difficult. Below, I go into these challenges in depth.

5.2 Did you revise the language design at any point?

I revised the language design at mainly 3 different points. Firstly, I had implemented the colours within each shape itself (not adding it to the tuple that describes a drawing, but rather a part of the shape itself). This meant that I had to do a lot of pattern matching and initially I was going to use generics [8]. This was a long-winded solution and I then decided on adding the colour to the tuple that describes a drawing. Secondly, I had implemented the rgba as Integers which worked but wasn't an accurate representation of the values from 0-255 that could go into the rgba entries. I then changed this to Word8 after I remembered that they can store 8 bits (hence in decimal 0-255). Finally, I had implemented masking that takes both drawings and just a boolean that specified if you

want to see the masked image over the original one. I felt that this didn't give the user as much flexibility as it should, so I changed this to a Word8 value for the user to have plenty of flexibility to determine how much of the masked image they want to see over the original image.

5.3 How well did your design choices fit with the needs of the rendering library?

The design choices I made fit well with the JuicyPixels library. This is because the use of optimised values for rgba, simple tuples to describe drawings and flexible functionality with masking all worked well with the JuicyPixels library. Also, adding the O(1) function improvements from the week 5 assignment improved it further, as I was able to speedup the rendering with the JuicyPixels library.

5.4 What changes are there to design choices if the final rendering target was instead an SVG produces by Blaze?

The changes to the decision choices for rendering an SVG with Blaze would be potentially representing other shapes with vectors (since SVG = Scalable Vector Graphics). Furthermore, a polygon would work well with a list of vectors instead of a list of points. For example, if you wanted to render shapes with the blaze-svg combinator library [9], you could simply "import qualified Text.Blaze.Svg11 as S", and use "S.rect" for a rectangle for example which might work well in terms of simplicity.

6 References

[1]: Demo video published on Youtube
[2]: Week 4 slides explaining halfLine function from CSU44012
[3]: Composing Fractals (research paper) by Mark P. Jones
[4]: Algorithm Archive - Affine transformations
[5]: JuicyPixels - Rendering library
[6]: Scotty - Web framework
[7]: Blaze - HTML combinator library
[8]: Generics
[9]: Blaze-SVG - SVG combinator library