Assignment 1 COMP1100

Abhaas Goyal

September 6, 2020

1 Introduction

This report presents a summary of Haskell program made using CodeWorld API to draw colourful shapes on the screen and present their area. The program is made using the MVC (Model-View-Controller) Architecture which is widely used in building GUI Desktop/Web Apps.

Note that the program has one additional feature of having the area being calculated for Polygon using the shoelace formula.[1]

2 Contents

2.1 Program Design

When the `Main` procedure is run, it calls the function `activityOf` relating to the three parts of MVC paradigm:

1. `emptyModel` (The Initialized Model)
2. `handleEvent` (Handles Events and makes the respective changes)
3. `modelToPicture` (Content actually been viewed on the website)

Model

Initially, we construct the structure of the `Model`. It is constructed in the following manner :

1. `Shape` : List of Shapes with required specification
2. `ColourShape` : a type Combining `Shape` and `ColourName` to give a single `ColourShape`
3. `Tool` : List of tools with specification to build Shape
4. `ColourName` : List of colours

We now define `Model` data type itself which combines and stores the information as a singular point of access for actions in the Controller/View. It's implementation w.r.t `[Shapes]` in the context of the current program can be thought of as a Stack data structure.
Finally `emptyModel` (The main data) is initialized to `Model` with default parameters.

Controller

The dependency tree is given as follows:

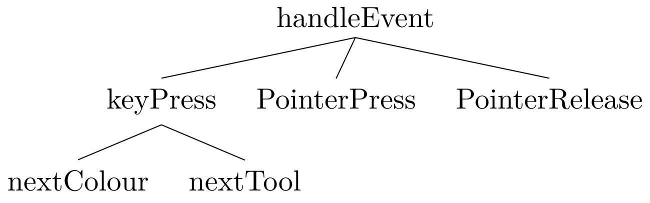

Figure 1 : Controller Dependency

1. Separate functions for changing colour/tool (`nextColour` and `nextTool` respectively) in `KeyPress` is made for readability.

2. Rotated parameter is initialized to zero on drawing `Rectangle` and `Ellipse` and when rotated it's changed by $\frac{\pi}{180}$. (Since conversion of $degree \leftrightarrow radians$)

3. While drawing the shape parameters of the current `tool` are changed and in the end, the new shape is pre-pended to the `[Shape]` with the current colour on the basis of other parameters.

4. In cases of `LineTool`, `CircleTool`, `RectangleTool`, and `EllipseTool` combination of both `PointerPress` and `PointerRelease` is important, while in other cases (`ParallelogramTool` and `PolygonTool`) only `PointerPress` determines the overall shape.

View

The dependency tree is given as follows:

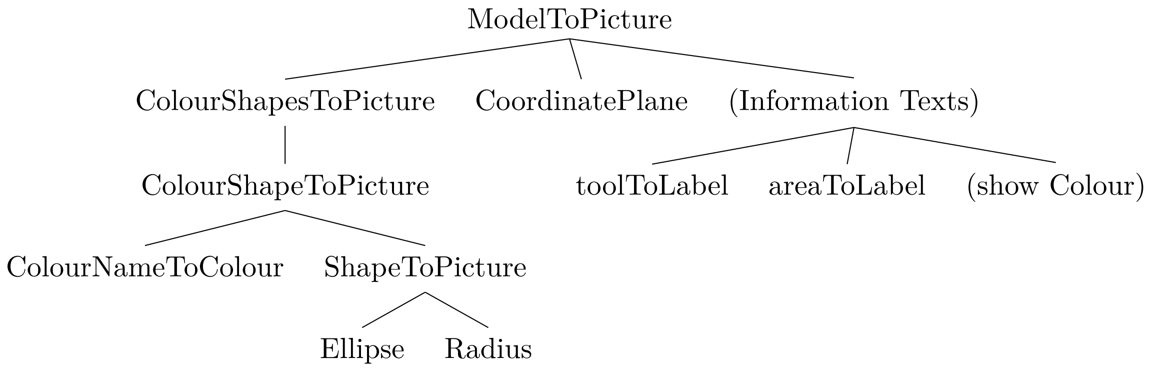

Figure 2 : View Dependency

1. For the whole program, `coordinatePlane` needs to be shown and Information texts have to be displayed for the current colour and tool with usage instructions.

2. `ColourShapesToPicture` acts as a higher order function to `ColourShapeToPicture` for making a list out of individual Shapes and passing to Model.

3. `ColourShapeToPicture` itself takes `ColourNameToColour` and `ShapeToPicture` as functions.

4. Below table gives implementation of `ShapeToPicture` and `areaToLabel` for basic shapes

Shape Name	**Picture Conversion**	**Area**
Line x y	`PolyLine` function on x, y	By Default 0
Polygon [points]	`SolidPolygon` function w.r.t p $\forall p \in points$	Calculated using Shoelace formula w.r.t all points
Rectangle (x1,y1) (x2, y2)	Calculate l,b by calculating difference between x and y components and use in `solidRectangle` function	Calculate l, b and use $abs(lb)$
Circle c e	Calculate r using distance formula on centre (c) and edge (e) points and use `solidCircle`	Calculate r and use πr^2

Table 1: Techniques Used for shapeToPicture and areaShapes

In addition to the listed things above:

- `rotated` has to used with the parameters of Rectangle and Parallelogram
- To build Rectangle, Circle, Ellipse and Parallelogram `translated` has to be used after finding out the center in the respective shapes. (After calculating their centres)

For more complex shapes :-

- **Ellipse:** Let the opposite corners be (x_1, y_1) and (x_2, y_2). Initially, we calculate the length (l) and breadth (b) of major and minor axis by calculating difference between x and y components. Alongside we calculate the center to where translation needs to be done.

  After implementing the above there are three cases
  Case 1 `Length > Breadth` : Draw Circle with Radius b and scale l by l/b
  Case 2 `Length < Breadth` : Draw Circle with Radius l and scale b by b/l
  Case 3 `(Length || Breadth) == 0` : Make the Circle with Radius 0

  For **area** we use the formula as πlb. (l and b are calculated with `abs`)

- **Parallelogram:** Given four adjacent points in terms of x_i, y_i and using the fact that diagonals of a parallelogram bisect each other, we get the following relation:

$$\left(\frac{x_1 + x_3}{2}, \frac{y_1 + y_3}{2}\right) = \left(\frac{x_2 + x_4}{2}, \frac{y_2 + y_4}{2}\right)$$

  Hence if the fourth point is unknown we can derive it and use `solidPolygon` on four points-

$$(x_4, y_4) = (x_1 + x_3 - x_2, y_1 + y_3 - y_2)$$

  For **area** of parallelogram we can think of three points as a linear transformation w.r.t taking it as basis vectors and take it's determinant in the following manner -

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + y_2y_3 + x_3y_1 - x_1y_3 - x_2y_1 - x_3y_2$$

2.2 Assumptions

The general assumptions is that the end user needs to know a lot of controls so he may experiment a lot with the app. So a lot of testing has been done w.r.t combinations of events.

2.3 Testing

The Testing of the program has been done in the following manner -

Mystery Image The mystery image exactly matches the one given in the specification, hence we can say with a high degree of certainty that `shapesToPicture` works fine since mystery image contained implementation of all `shapesToPicture`.

Test Images for Area `Doctests` for all the shapes in area for which the area was already known were written which came out to be correct. (Written above `areaShapes` for reference).

Boundary Cases Tested to work in all conditions with boundary cases on combinations of keypress and pointers and removed errors such as:

1. Picture when [Shapes] is given as an empty list to `ColourShapesToPicture` -> return `mempty`
2. Press `BackSpace/Esc` when there are no shapes drawn -> return [] w.r.t `[Shapes]`
3. Drawing when length or breadth is 0 in ellipse -> Can't scale by ∞ so make radius 0 in drawing
4. Pressing `Spacebar` in Polygon before clicking on three points shouldn't do anything
5. Not changing `tool` when the user is halfway through an operation in any shape

2.4 Inspiration/ External Content

Inspiration was taken from various topics such as

1. CodeWorld API documentation [2]
2. Event Listeners in JavaScript and relating it's working to the current project
3. Shoelace formula in a Polygon (Works in a non-intersecting polygon) [1]

3 Reflection

3.1 Conceptual Issues

- As my first moderately sized software project in Haskell, I was having trouble on connecting all the bits and pieces of the program. To overcome this, I drew a dependency graph of functions and study `Model.hs` in depth to understand the basic structure of the data being transformed before working on other parts.
- Changing the model stack and it's relation to `PointerPress` , `PointerRelease`, `Tool`, and `Colour` was difficult as well. To solve this and use the right combination, I read about event listeners in JavaScript.

3.2 Things to do differently

- I would not keep the whole model to be changed everytime on changing events. For example, separate models for `Tool` and the `[Shapes]` is more easily understandable and would do a faster operation than the current implementation.
- I would also have the rotate parameter on shapes other than Rectangle/Ellipse for user friendly interaction.

References

[1] B. Braden, “The surveyor’s area formula,” *The College Mathematics Journal*, vol. 17, no. 4, pp. 326–337, 1986.

[2] Google, “Codeworld api,” Available at `https://hackage.haskell.org/package/codeworld-api-0.6.0/docs/CodeWorld.html`, version 0.6.0.