COMP1130 Assignment 1

Manindra de Mel, u7156805

Tutor: James, LAB: Tuesday 13:00 - 15:00

Introduction

Assignment one is a simplistic design program, which initially is missing key functionalities that have to be implemented in the programming language Haskell. This report will consist of an intuitive approach to implementing the lacking functionality throughout the program, some limitations faced and attempted methods taken to overcome them. Furthermore, such implementations consists of a concise and readable codebase, which correctly reacts to all the user inputs that the assignment requires. Additionally, concepts taught throughout the semester have been effectively recognised and applied appropriately throughout the program, and often the ideas behind these implementations are extremely intuitive.

The Helper functions

The first main problems posed in this program were the colours and the tools. Currently, there was no functionality provided for the user to easily change between which type of shape to draw or what colour to draw them in. Additionally, the defined tools in the base program didn't have any implemented descriptions for the user to follow. Once implemented, these helper functions were precisely defined.

```
nextColour :: ColourName → ColourName
nextColour colour = case colour of
 White → Black
 _ → succ colour
```

As shown, this simple function, which changes the current colour, takes advantage that the `ColourName` type is enumerable and thus can be succeeded upon. This implementation saves several lines of unnecessary repeated code, which can often be prone to error or unreadable.

Creating the shapes

With the helper functions finalised, the main functionality of drawing the shapes onto the screen was the final accumulation of all the preceding tasks asked by the assignment. The code created here must both be intuitive for the user to use and meet the parameters of the sample shape provided by the task. Additionally, the code should be as succinct as possible, providing as much mathematical and logical intuition behind how these shapes are drawn onto the screen.

Lines, Circles, Triangles and Polygons

More simplistic shapes were extremely easy to create. For example, a Line simply consists of two points A and B with a line drawn between them. A circle, given a centre (A) and a radius point (B) can be first created with `solidCircle` $\sqrt{(A_x - B_x)^2 + (X_y - B_y)^2}$ to create the shape, and then translated from the origin to the user defined centre A. Triangles are also simply defined, since the triangle is isosceles we know that the unknown point must be on the same y level as the other base point and the x coordinate would simply be $2 \times A_x - B_x$ where A is the Apex of the triangle and B is the known base coordinate. Finally, polygons are simply a list of points which can be drawn with the inbuilt `solidPolygon [(Point)]` function.

Drawing more complex shapes

Rectangles

Initially, drawing a rectangle seemed like a simple task. However, the parameters defined by the assignment made drawing this shape significantly harder. The task provided us with a scale factor, which can be interpreted as the ratio between the width and length of the rectangle. The next two parameters provide us with two vertices of the rectangle. The created implementation currently satisfies the correct dimensions and general position that the user defined. However, the rectangle's centre lies on the line AB rather than AB being one of the sides. Unfortunately, the plausible solution to this problem consists of guards and repeated code, which is deemed as unnecessary and inconsistent with the rest of the program. Hence, it was disregarded, as the current implementation, is both precise and mostly accurate.

Cap

A cap is simply a semicircle. Therefore, a simple implementation of this shape would be to draw a circle and then clip it with another shape at a given y-coordinate. The in-built `clipped` function provides a rectangle to clip off another shape. Then we can define the rectangles dimensions as such:

```
(2 * radius) (2 * (b - cutoff + radius)
```

This rectangle's dimensions are then applied to a newly created circle that has been translated about the origin.

```
translated 0 (-radius) $ solidCircle radius
```

Finally, we move the semicircle back to the centre provided, resulting in a concise definition for a cap.

```
translated a (b + radius) clipped (2 * radius) (2 * (b - cutoff + radius))
(translated 0 (-radius) $ solidCircle radius)
```

Handling user input

There are two forms of user input present within the program. Key presses and mouse based input. Key presses are often used to change tool or colour.

```
| k == "T" -> Model shapes (nextTool tool) colour saves -- change tool
```

Whilst other key presses are used for modifying certain shapes parameters like the rectangle or undoing a shape that was just drawn:

```
(k == "Backspace" || k == "Delete") && not (null shapes) -> Model (init shapes)
tool colour saves
```

Although key presses are important. Mouse based input is the main source of interaction between the user and the program. There are two types of mouse input. `PointerPress` when the user presses down on the mouse and `PointerRelease` is when the user then releases the mouse button. `PointerPress` can be referred to as the first coordinate and is often the centre of a shape. Subsequently, `PointerRelease` is the second coordinate, which may be a vertex of a shape. Due to how the tools and shapes are both similar in definitions, both `PointerPress` and `PointerRelease` were implemented consistently with respect to each shape.

Generally, for `PointerPress` we simply append the defined point to the tool the user is currently using as defined below

```
 Tool _ ⟶ Model shapes (Tool FirstPoint Nothing) colour
```

Moreover, in `PointerRelease`, the tool is also matched. However, since this is the last point the user is defining *(this is exempt for polygon and cap)*. We append the associated shape with the tool to the `shapes` list, and finally clear the tool of its parameters for later use.

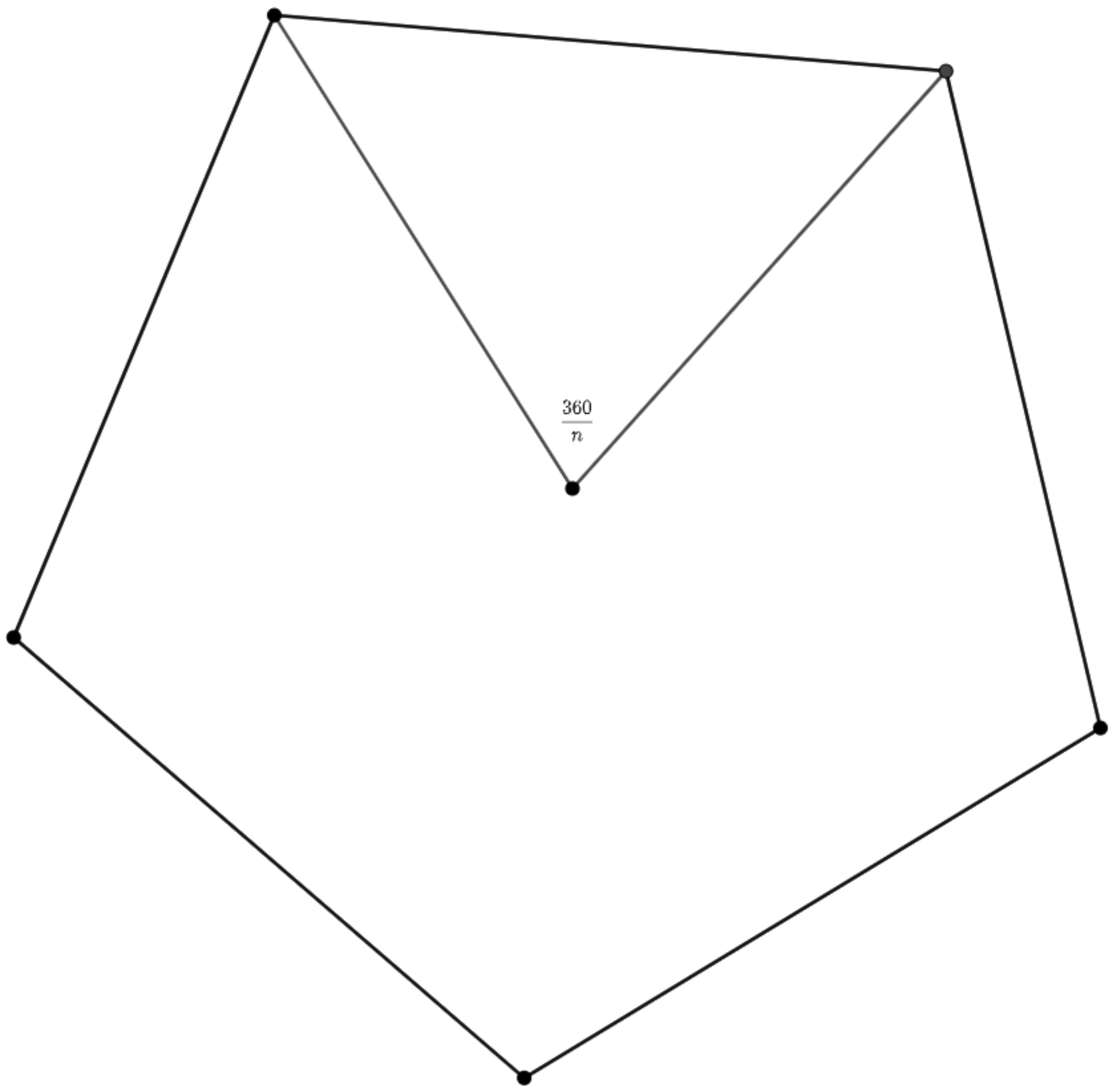
```
Tool PreviousPoint ⟶ Model (shapes ++ [(Shape (fromJust PreviousPoint)
NewlyDefinedPoint, colour)]) (Tool Nothing)
```

Extension tasks

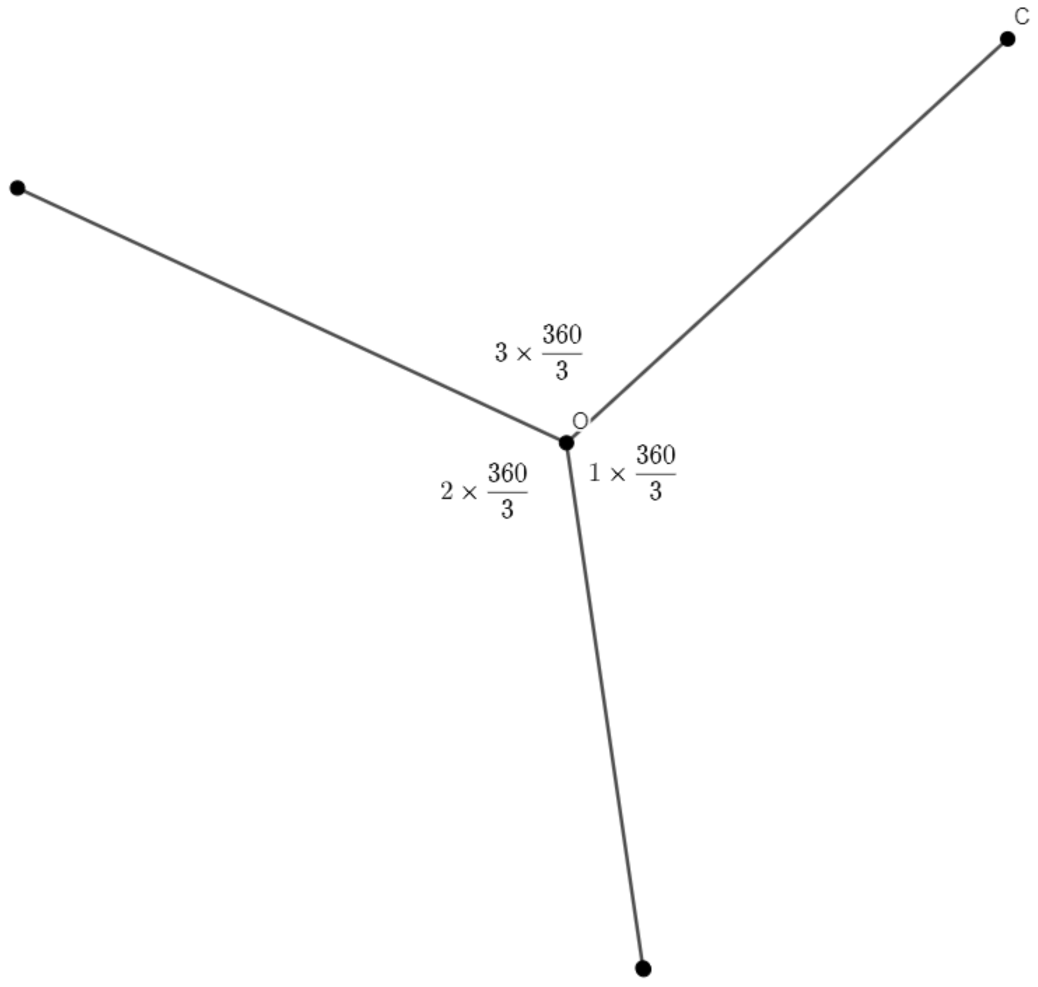
Extension 4.1 was the addition of saving the state of the current shapes on the screen and having the ability to return to any one of those saves. Approaching this task, it was immediately recognised that the data structure to contain the current state of shapes, would be a multidimensional list. Specifically, a list with type `[[ColourShapes]]`. This extension functions as such: On a key press, the user will append the current Shapes list to another list. Then on another key press the most recent save will be chosen from the saves list and then dropped so the user can recursively travel back throughout saves.

Extension 4.4 was the second extension that was decided to be implemented into the program. This extension requires an additional tool which, given a number of vertices, a centre and a single vertex, should draw the appropriate polygon for the amount of vertices given at the user defined location.

I first began implementing the `generalPolygon` tool. A tool and shape that is extremely similar to both `RectangleTool` and `Rectangle` respectively. However, once completed, generalising polygons and the relation between the vertices and the type of polygon was the difficult logic behind the tool. The solution to this problem is a property of polygons, where the angle between the vertices is as given $360/\text{number of vertices}$.



As an example, given a triangle with 3 vertices and a centre O and a vertex C we can generate the other points with this angle-vertex relation.



Hence, list comprehension can be used to encapsulate this idea:

```
[rotatedPoint ((nVertex * (360 / vertices)) * (pi/180)) relativePoint | nVertex
← [1..vertices]]
```

Where `nVertex` simply increases with respect to which point we are calculating and `relativePoint` is the difference between the user defined vertex and the centre of the polygon. Finally, we then move the shape back to the centre using `translated centreX centreY`.

Conclusion

Assignment One now has the functionality to provide a user with some basic tools to enact a design program. The addition of these tools and shapes were both implemented in a clear manner, such that both the user using the program or another programmer reading through the software, can easily understand the function of each tool or shape. Extension problems also added a lot of functionality, whilst also retaining the intuitive and consistent programming that was present within the rest of the software. Although some functions such as the rectangles did not meet the required sample image, the shape still functions as a rectangle, meeting all the requirements such as the dimensions and general position of the user-defined input. Overall, the program was implemented in a succulent, effective and easy to understand manner, minimising the amount of code required for added functionality.