Image server
Topics in functional programing

David Esteso Calatrava

November 13, 2024

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

School of Computer Scinece and Statics

Contents

1 Design of the Drawing eDSL

1.1 Basic Shapes

First, let's recall that the decision to define shapes based on point inclusion is driven by our rendering approach. Each pixel maps to a specific point in the drawing space, and its color depends on whether this point falls inside a shape. To implement new shapes, we need to define a function for each one to determine if a point lies within its boundaries. For the ellipse and rectangle, we decided to follow a similar approach to the circle and square, recognizing that these are particular cases of the first two shapes. For polygons, however, we had multiple options, including ray casting, winding number, angle summation, or triangulation methods, each offering different ways to verify if a point is inside the shape. In the end, we opted for the cross-product method, that works by checking if the point is consistently on the "correct" side of each edge of the polygon; essentially, if it is always on the left or right side of every edge, relative to the polygon's orientation. The choice of this method is based more on preference than efficiency.

1.2 Affine Transformations

In this case, there's no notable design decision involved beyond adhering to the mathematical definition of the new operation. The implementation simply follows the established formula.

1.3 Color Specification

To implement gradients, we redefined the Color data type to include a new ColorType that can be either a Solid color or a Gradient between two colors along the y-axis. This approach is considered the most flexible, allowing us to handle coloring a pixel as two distinct cases within the same data type. Additionally, we declared "color constants" in order make the hardcoding of shapes more readable and maintainable. Now, we can say "circle, red" instead of "circle, Color 255 0 0 255".

The main challenge with the gradient is that, to interpolate the color, we need to operate within the range of 0 to 1. Since we don't know the height of the shape in advance, we must adapt to its dimensions. This becomes straightforward thanks to the *bounding box* (which we will explain later), as it gives us the height directly. This way, we can normalize using $t = \frac{y - \text{minY}}{\text{maxY} - \text{minY}}$ to scale within the range of 0 to 1.

1.4 Combination of Shapes

In our design, we had to redefine what constitutes a drawing.

```
data Operation = Over | LeftOf | RightOf | Above | Below

data ColorDrawing
    = SingleDrawing (Transform, Shape, ColorType)
    | Combine ColorDrawing Operation ColorDrawing
```

I'm aware that this definition is not exactly what was provided in the original specification, as it allows for one drawing to be combined with another. However, I believe this is the most suitable design decision for the following reasons:

1. Solving this issue is as simple as using a function to check if the user is trying to combine a drawing with another, and then providing an informative error message.

2. If this were a project that could be continued by either me or another programmer, this definition makes future extensions easier. We wouldn't need to change the existing code, as it allows for further flexibility and expansions in the design. It avoids needing to alter existing functionality and documentation, thus facilitating maintenance and future work.

Now, the decision of how to implement operations for combining shapes is based on the idea that each shape is enclosed within a bounding box. Then, when using a combination operation, you are essentially modifying the final position of a shape. For example, consider the `LeftOf` operation between a shape and a drawing. To determine the exact position of the new shape's center, we need to know:

- Where the drawing is located.
- The size of the drawing.
- The size of the shape.

The center of the new shape (without translating the shape) will be at:

$$\left(\text{center of the drawing}_x - \frac{\text{width of the drawing}}{2} - \frac{\text{width of the shape}}{2}, \text{center of the drawing}_y\right)$$

This calculation is straightforward if we have the bounding boxes for both shapes. Additionally, we need to adapt these combinations to transformations such as translations, scaling, or rotations on both shape and drawing. This can be easily achieved, as we have transformed the four points of the bounding box. With the definition above, it's also easy to see how this could adapt to other operations; for example, in an "Over" operation, we wouldn't consider widths, or for an "Above" operation, we would work only with heights.

2 Implementation

2.1 Rendering with JuicyPixels

In the original `render` function, it simply checked if a point was inside the image and, if it was, assigned it the color white. Now, in `renderWithColor`, we added a function that not only determines if the pixel is inside but also what color it should be. Additionally, we introduced transparency using the RGBA color model, which enhances the visual design of the website by allowing for smoother and more appealing visuals. This decision just aims to improve the overall look of the website.

To associate coordinates with points in the window space, we used a `Map` data structure. This allows for constant-time lookups instead of linear searches, improving the efficiency of coordinate mapping.

2.2 Web Application with Scotty and Blaze

There are no design decisions beyond the fact that Scotty handles routing and request management, while Blaze handles HTML content generation.

3 Optimizations and Performance Considerations

Additionally, on top of the optimization in the render function using `Map`, I further improved my program by enabling parallel image rendering with the `mapConcurrently` function from Haskell's `Control.Concurrent.Async` module. `mapConcurrently` takes two arguments: a function to apply to each element of the list and the list itself to process concurrently. The function is a lambda that takes a tuple `(path, window, illustration)`. It calls `renderWithColor` to generate and save the image to the file using the specified window. It "Maps an IO-performing function over any Traversable data type, performing all the IO actions concurrently". The parallel point comes because "async will try to immediately spawn a thread for each element of the Traversable". I am quoting `Hackage`. On my machine, with this parallel improvement, the render time has decreased from approximately 55 seconds to 25 seconds—a reduction of over 55%.

4 Reflection on the DSL Design Process

4.1 Bounding Box

Since much of the program's logic relies on the bounding box, I believe it deserves a dedicated section. To calculate the bounding box, we first define one for each shape and then transform the points of that bounding box. However, this operation cannot be generalized for all shapes, as each has its own characteristics. For instance, rotating a circle doesn't change its bounding box, but rotating the bounding box itself does. This illustrates that each shape requires individual treatment when calculating its bounding box after a transformation.

Additionally, choosing points to transform for the bounding box calculation can be straightforward for shapes like a square, but it's more complex for curved shapes like circles. Operations like shear, for example, don't work well if we approximate a circle with only four points. Because of this, we avoid approximating ellipses and circles with just a few points and instead use functions for approximating these shapes. I decided to approximate with 15 points because, although using more would provide slightly more pixel accuracy, I chose not to use extra execution time.

It's important to note that calculating the bounding box first and then applying a rotation would be a mistake. We want the bounding box to always remain upright.

4.2 Design Challenges

One of the biggest challenges was deciding to use the bounding box and realizing that calculating it couldn't be handled generically for every shape, as explained in the Bounding Box section. The specific details and particularities mentioned there, such as the need to account for each shape's unique characteristics, like how rotation affects circles differently from other shapes, led to multiple revisions in the code. Another coding challenge was making sure to reuse the code as much as possible, since there are many cases (for example, operations like over, left, etc.) and a lot of repetitive logic in each case.

Regarding to optimization, The first thing I tried to parallelize was using the parallel function, but I think it didn't work because par and pseq only work with pure values and don't create separate threads for IO actions. IO actions, like rendering images, are inherently sequential because they depend on the system's state, so they need explicit thread management to run in parallel.

Another thing that took me a while to realize is that when translating a point using `transform`, I was subtracting the coordinates, but for my calculations, I wanted to add them. This applies to the rest of the operations as well, so I had to implement an `inverseTransform` function that does the opposite.

4.3 Rendering Considerations

Current implementation is based on finding out the color of a pixel at runtime. If the format were to change to SVG, then this core rendering should change. SVG stands for Scalable Vector Graphics, and in these kinds of formats, the concept of vectors is used. It does not store information concerning pixels but geometrical data of shapes. In SVG, we work with paths and coordinates, dealing with vector shapes instead of pixel-level computations. Although the bounding box principle my solution uses could be generalized to handle SVG, I am not allowed to make a conclusion on that without having hands-on experience with it. What is for sure, though, is that the actual rendering methodology would definitely change, since this constitutes a core aspect of the system.

4.4 Workflow and iterations

One of the biggest challenges was figuring out where to start to avoid unnecessary iterations. At first, I thought it made the most sense to calculate the gradient before the operations, as it seemed like the logical order of difficulty. However, I soon realized that I had to redo the gradient after working on the operations, since they led me to focus on the bounding box. If I could go back, I would have tackled the operations first, before the gradient.

5 Project Deliverables and Completion

To conclude this report, I believe all the deliverables requested for the project have been successfully completed. It is possible to combine all types of shapes, including those with every possible transformation. However, I must admit that the wide range of possible combinations makes it nearly impossible to test all of them to ensure everything works correctly. I have tried to showcase as much variety as possible on the website, but possibilities are almost infinite. Therefore, while it seems everything is functioning as expected, it's important to acknowledge that there may still be some specific combination with an anomaly. Nonetheless, this is a common challenge in projects of this nature. Overall, I am confident that the system is robust and performs as expected.