5-31-2018

Reflections on Building DartDraw: A React + Redux Vector-Based Graphics Editor

Elisabeth G. Pillsbury
Dartmouth College

Dartmouth College Computer Science
Technical Report TR2018-853

Reflections on Building DartDraw: A React + Redux Vector-Based Graphics Editor

Elisabeth Pillsbury
May 31, 2018

1 Introduction

In the spring of 2017, Professor Thomas Cormen sent an email to the Computer Science Majors in the Dartmouth Class of 2018. He proposed a group senior thesis project for the 2017-2018 academic year: create a vector-based graphics editor for Mac OS X inspired by MacDraw, a drawing application released by Apple in 1984 and discontinued in 1997.

MacDraw is desirable for two reasons. It is simple; frequent user actions - such as drawing a rectangle or changing ruler divisions - are accessible and easy. Secondly, the PostScript produced by MacDraw integrates seamlessly with LaTeX using the PSfrag package. Both of these attributes are essential for Professor Cormen, who is in the process of updating hundreds of MacDraw figures for the fourth edition of his textbook, *Introduction to Algorithms*, produced in LaTeX. However, no MacDraw replacement for Mac OS X currently exists that satisfies both of the above requirements.

Six ‘18s signed onto this project and, in the fall of 2018, began to design and build a MacDraw replacement for Mac OS X. We called it DartDraw.

We divided the project into three parts:

- Jean Zhou took on the task of decrypting old MacDraw files for the purpose of importing them into our application,
- Trevor Davis tackled exporting graphics: converting the application’s JSON output into Encapsulated PostScript,
- The remaining four students, Emma Oberstein, Collin McKinney, Luisa Vasquez, and myself, built the drawing application itself. We chose Electron as the framework for our application in order to leverage our past experience and use a web development stack (HTML, CSS, JavaScript, React, Redux) to build a native desktop application. We harnessed Scalable Vector Graphics (SVG) to define vector-based graphics in XML format.

2 My Contributions

I was the primary contributor to and took responsibility for three project components: Zoom and Pan, Grid and Ruler, and the Arrowhead Editor.

Zoom & Pan:

- Zoom to custom scale
- Zoom to marquee selection

- Pan canvas with mouse click and drag

Grid & Ruler:

- Show / hide grid
- Show / hide rulers
- Set grid preferences (dimensions, units)
- Set ruler preferences (divisions per unit - i.e. halves, quarters, tenths, etc)
- Customize canvas presets (i.e. save a set of grid and ruler preferences in a preset that can be easily applied later on)
- Track the mouse's exact xy coordinates with corresponding marks on the rulers
- Snap objects to grid (mouse trackers will also snap to grid when this option is turned on)

Arrowhead Editor:

- Drag out desired arrowhead shape in GUI or enter numerical arrowhead dimensions
- Customize arrowhead presets (i.e. save arrowhead preferences in a preset that can be easily applied to any line later on)
- Select and edit one or more arrowheads
- Lock arrowhead aspect ratio
- Flip arrowhead across the vertical axis
- Place arrowheads at the start or end of a line

3 Design Challenges

Set Pan Coordinates While Navigating The Canvas: As I tested the zoom and pan features, I repeatedly discovered unexpected responses to edge cases I did not originally account for. It took me many iterations to refine our application's zoom and pan behavior until it functioned as a user would expect. I discovered three guiding principles which govern correct pan behavior at any given zoom scale. Consider the following three cases as illustration:

We define pan coordinates to be the upper-left-hand corner of the `viewBox`, the viewing window through which all or part of the canvas is visible.

> *1) The entire canvas is fully visible within the `viewBox`.* In this scenario, panning is disabled and the pan coordinates are (0, 0). This makes sense; there is no need to pan if you can see the whole canvas.

> *2) The canvas extends off the screen in one direction but not the other.* The canvas does not necessarily have the same aspect ratio as the `viewBox` and so the x and y pan coordinates must be handled separately. For example, consider a very tall and skinny canvas; the x pan coordinate would be locked at 0, but the y pan coordinate would be variable to allow for the user to pan to, but not past, the bottom of the canvas.

3) The canvas fills the `viewBox` *and extends off the screen in both directions.* In this scenario, we enable pan in both directions but never allow the user to pan off the canvas in any direction.

When I first implemented zoom functionality, I did not update the pan coordinates; the `viewBox` was anchored to the upper left-hand-corner even as the zoom scale changed. Only after testing did I realize that this is unexpected behavior. In most drawing programs, the user zooms out from and in to the center of the `viewBox`. Implementing this zoom-to-center feature was more difficult than I anticipated because it requires that we heed the above `viewBox` constraints at two zoom scales. A `viewBox` might be in a valid position but zooming out from the center would place it in violation of one of the above cases. This edge case is best illustrated by a simple example shown below.

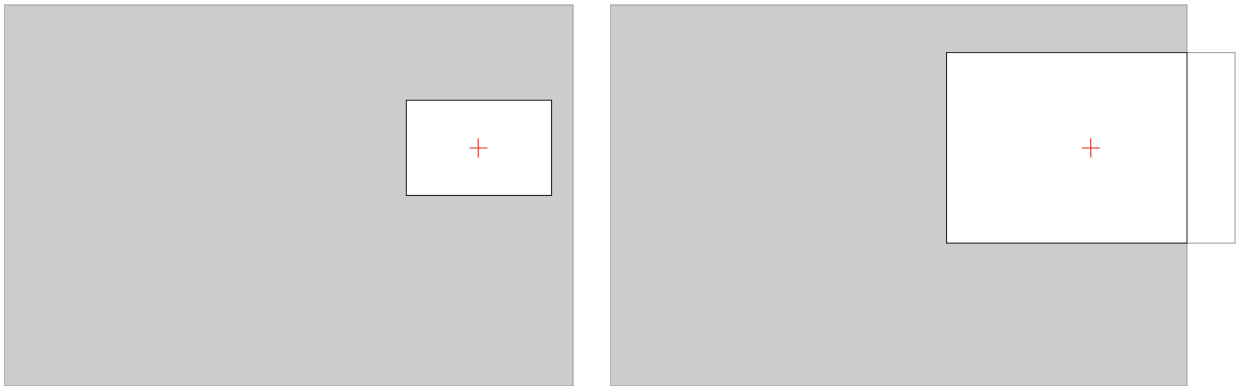

Figure 1: Invalid "zoom-out" behavior.

The above behavior is unexpected and thus, undesirable. Instead, we must adjust the pan coordinates, as shown below, so that our `viewBox` is in compliance with the cases posed above.

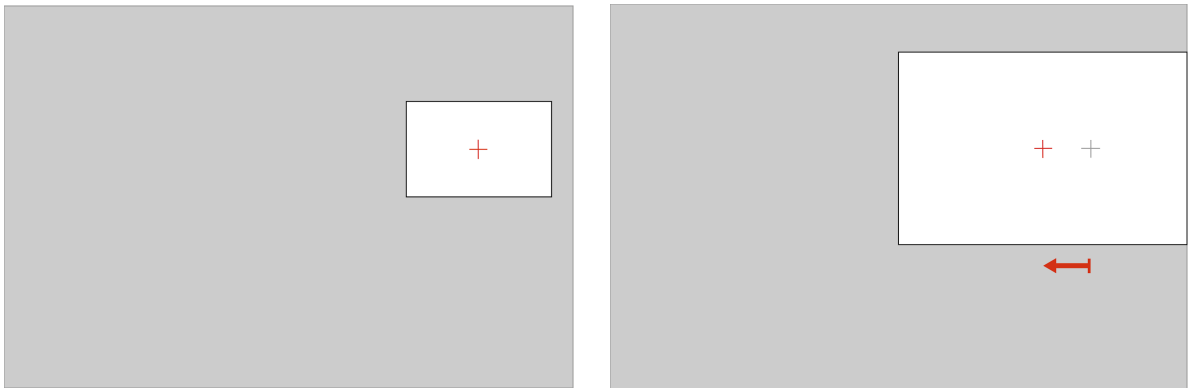

Figure 2: Valid “zoom-out” behavior with appropriate pan coordinate adjustment.

Calculating Ruler Tick Length and Spacing: I initially thought I would treat the ruler components - which lie along the left and top edges of the canvas component - similarly to the canvas. By applying the same transformations, the ruler would zoom and pan in tandem with the canvas. But I quickly discovered that this was not feasible for a few reasons. Scale would have to be applied in one direction only, effectively stretching each ruler lengthwise and distorting the rendered ticks and unit labels. I could not settle on a clean way of counteracting this skew effect. Additionally, in many circumstances, the level of ruler detail and granularity increases as we zoom in. For example, we might show the half-inch ticks at 100% zoom but quarter-inch ticks at 200% zoom and so on. This effect requires that we update the ruler component in addition to applying a transform during the render. Because of these issues and their messy solutions, I discarded this implementation and decided to manually calculate each tick’s location and length to create the illusion of a ruler that scrolls and zooms.

This approach presented its own set of challenges. Depending upon the zoom scale and ruler division preference, we calculate the appropriate location of each tick. But what if we’ve

zoomed out and the density of the ticks leaves the ruler illegible? The complexity of managing the spacing between unit ticks, sub-unit ticks, and unit labels is best demonstrated by example.

Consider a scenario in which we've set the ruler unit to be centimeters. But perhaps we've decided that visually, ticks must be at least 21 pixels apart. At our current zoom scale, however, rendering every tick would require one pixel between ticks. My initial solution was to find a skip interval such that we render ticks as close together as possible without violating the minimum spacing constraint of 21 pixels. The resulting ruler is shown below:

Figure 3: Unexpected tick spacing in a previous version of DartDraw.

Only after rendering did I realize how jarring and unintuitive this solution was - instead of choosing the first interval that satisfies the minimum spacing constraint, I had to be more judicious in choosing the a skip interval that is a factor of the base. In this example, choosing a skip interval of 25 for a ruler with units in the hundreds results in a clearer, more intuitive spacing interval.

Figure 4: Expected tick spacing in the most recent version of DartDraw.

If the number of pixels per centimeter as determined by our zoom scale (`pixelsPerUnit`) equals our minimum spacing constraint (`minSpacing`), we can safely leave our skip interval (`interval`) equal to 1. However, if `pixelsPerUnit` is less than `minSpacing`, we must find an `interval` such that

`interval * pixelsPerUnit >= minSpacing`.

But how do we choose an `interval` such as 25 (which we know to be more expected) when 21 is the first (lowest) number which satisfies the inequality above? I solved this problem by requiring `interval` to be the product of one of the values in `[2, 2.5, 5, 10]` and `scalingFactor` which is a power of 10. The only caveat is that `interval` may not equal 2.5 because whole units should not be split unnecessarily.

This function for finding the spacing between ticks, described in words above, is conveyed in pseudocode on the following page.

```
findSpacingInterval(minSpacing, pixelsPerUnit)
      factorArray = [2, 2.5, 5, 10]
      scalingFactor = 1
      interval = 1

      if pixelsPerUnit < minSpacing:
            interval = ceiling of (minSpacing / pixelsPerUnit)

            while interval > factorArray[factorArray.length - 1] *
                  scalingFactor:
                  scalingFactor = scalingFactor * 10

            for i from 0 to factorArray.length - 1:
                  if interval < (factorArray[i] * scalingFactor):
                        interval = factorArray[i] * scalingFactor

                        if interval !== 2.5
                              return interval
      return interval
```

Arrowhead Miters: When selecting an arrow object, we expect to see two selection handles, one at the butt end of the line, one at the tip of the arrowhead. To do this, we calculate how far the arrowhead extends past the end of the line. This value varies depending upon potentially three factors: the dimensions of the arrowhead and, if there is a stroke, the stroke width and angle of the arrowhead point. The first factor is obvious but the addition of the last two is less intuitive. As the arrowhead becomes pointier, the angle shrinks, and the length of the miter (the seam between the inner and outer corners of the joint) increases. This is because the stroke extends past the point of intersection. To prevent the miter from becoming too long and unwieldy, we impose a miter limit (made easy to implement with SVG) and bevel the point of intersection as shown below.

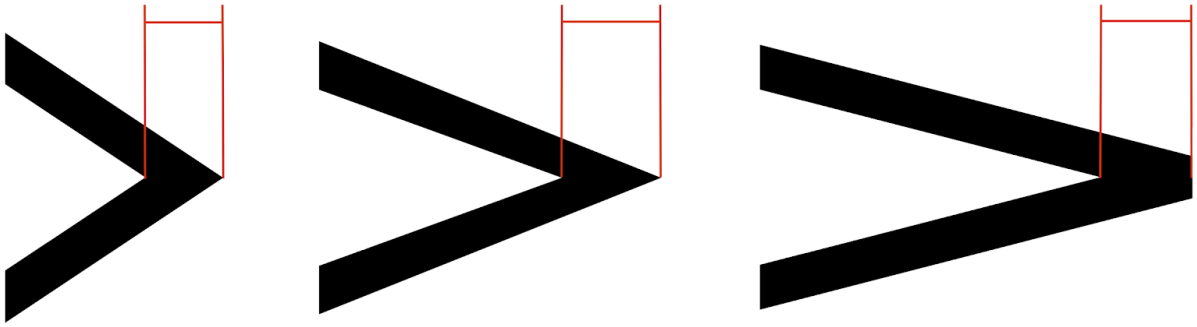

Figure 5: Demonstration of the miter limit.

Using the arrowhead's dimensions (represented as the points of a polygon) and stroke width, and the angle of the arrowhead tip, we are able to calculate the length of the arrowhead in order to render the selection handles correctly.

4 Acknowledgments

Thank you to my fellow teammates, Trevor, Luisa, Collin, Jean, and Emma, for their patience, support, and friendship throughout this process. Professor Cormen served as both our advisor and client for this project; his dedication, guidance, and insight were invaluable while developing DartDraw and drafting this paper.

5 Resources

Figures 1, 2, and 5 were created and exported to PDF using our application, DartDraw. Figures 3 and 4 are screenshots from DartDraw.