Thinking Inside the Box: Converting Encapsulated PostScript to Scalable Vector Graphics

Trevor L. Davis
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

Part of the Computer Sciences Commons

Recommended Citation

This Thesis (Undergraduate) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Undergraduate Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.
Thinking Inside the Box:

Converting Scalable Vector Graphics to Encapsulated PostScript


By Trevor Davis
Table of Contents

1. Introduction .................................................................................................................. 3
   1.1. Background ............................................................................................................ 3
   1.2. Primary Tasks ........................................................................................................ 4
   1.3. Other Tasks ............................................................................................................ 4

   2.1. Choice of Programming Languages ...................................................................... 6
   2.2. The Electron Framework ....................................................................................... 7
   2.3. React and Redux ..................................................................................................... 8
   2.4. Scalable Vector Graphics ....................................................................................... 10
   2.5. Additional Libraries ............................................................................................. 11

3. Rendering Shapes ......................................................................................................... 12
   3.1. A Note on PostScript ........................................................................................... 12
   3.2. DartDraw’s Export Landing Point ......................................................................... 12
   3.3. Line-Based Shapes ............................................................................................... 14
   3.4. Ellipses .................................................................................................................. 21
   3.5. Arcs ....................................................................................................................... 24
   3.6. Rounded Rectangles ............................................................................................ 26
   3.7. Bezier Curves ....................................................................................................... 27
   3.7. Text ...................................................................................................................... 28

4. Computing Bounding Boxes ........................................................................................ 31
   4.1. The Bounding Box Object ..................................................................................... 31
   4.2. Lines and Freehand Paths .................................................................................... 32
   4.3. Rectangles ............................................................................................................ 34
   4.4. Polygons ............................................................................................................... 35
   4.5. Ellipses ............................................................................................................... 37
   4.6. Arcs ....................................................................................................................... 39
   4.7. Rounded Rectangles ............................................................................................ 41
   4.8. Bezier Curves ....................................................................................................... 41

5. File Saving and Opening .............................................................................................. 41
   5.1. Saving files with IPC ............................................................................................ 42
   5.2. Opening files ........................................................................................................ 43

6. Future Work .................................................................................................................. 44
   6.1. Complex Shape Intersection ............................................................................... 44
   6.2. Color Flattening .................................................................................................... 45
   6.3. File Compression ................................................................................................... 46

7. Reflection ...................................................................................................................... 47

8. References .................................................................................................................... 49
1. Introduction

1.1. Background

In 1984, a vector graphic drawing application called MacDraw was released as part of the first Apple Macintosh systems. Following its initial success, MacDraw underwent several iterations until it became ClarisDraw 1.0v4 in 1994, which continued to function on Mac OS X for over a decade. When Mac OS X 10.5 Leopard debuted in October 2007, however, it did not support Classic applications, including MacDraw. Users of Intel based Macs have been unable to use MacDraw on their machines ever since, prompting a call to create a compatible version of MacDraw by the authors of Introduction to Algorithms.1

The call went unanswered until autumn 2017 when Professor Tom Cormen of Dartmouth College assigned several undergraduates the task of creating DartDraw, a replica of MacDraw with a few key improvements. These six undergraduates, including myself, were given approximately nine months to complete their task, broken down into individual responsibilities.

As one of the authors of Introduction to Algorithms, Cormen had previously used MacDraw to create the figures for the book by exporting drawings from the drawing application and embedding them within his LaTeX documents. As a member of the

---

1 https://en.wikipedia.org/wiki/MacDraw
DartDraw team, my task was to handle the creation of EPS files from the DartDraw application.

1.2. Primary Tasks

PostScript is a stack based, procedural programming language created by Adobe for page description. Printers can print directly from PostScript files. There is a similar file type known as Encapsulated PostScript (EPS), which allows for thumbnail images of the resulting PDF as well as additional metadata. In particular, every EPS file contains information about the enclosing rectangle of the image created by the file called the **Bounding Box**. This feature is particularly useful for creating images that can be embedded in other files such as text documents.

My primary task for this thesis was twofold:

- Enable DartDraw to export EPS files that describe the image created by a user.
- Ensure that these EPS files contain an ink-based bounding box of the image.

The core of this document describes my work on these two primary tasks.

1.3. Other Tasks

In addition to handling the EPS export functionality; I managed two other related tasks in the creation of DartDraw.
DartDraw needed a file save format so that users can save drawings and return to them later. When we created DartDraw, we leveraged the Redux predictable state container (see section 2.3), which tracks the state of a user’s drawing in an object tree called the store. I used the store as the core of a standard file save format with the extension “dart.”

Lastly, I created an inter-process communication component in the ReactJS framework to handle events passed between the main process and window processes in DartDraw. See section 5.1 for more information.

2. Design Decisions, Frameworks, and Libraries

Creating an application like DartDraw requires an understanding of many existing utilities that make a programmer’s life easier. In the early stages of this project, the DartDraw team spent a considerable amount of time deciding upon the base structure of our project. The sections below discuss the benefits and tradeoffs of our decisions.

An online GitHub repository is available for a more in-depth look at the project.
2.1. Choice of Programming Languages

Given the plethora of programming languages that exist today, it was a daunting task for our team to decide which one to use to create DartDraw. Eventually, we narrowed down our choices to two languages: Swift and JavaScript.

Swift is Apple’s proprietary programming language that is specifically designed for easy use with iOS and OS X applications. It works hand in hand with Objective-C and has many custom libraries that make things like menu options and styling trivial for developers. Furthermore, it works with Apple’s advanced Xcode IDE that even simplifies certain tasks such as layout design down to a drag-and-drop procedure. On the other hand, many undergraduate students do not know Swift, and it has a significant learning curve. Swift is also exclusively used for Apple applications.

JavaScript by itself is primarily a scripting language for web pages, but it has recently become more versatile. Now, libraries such as ReactJS make it easy to use JavaScript in desktop applications. JavaScript is one of the most popular programming languages with one of the fastest growing user bases. It is fast, multi-platform, and expressive. Because of its prevalence in web applications, there is a JavaScript library for virtually everything. Many of these libraries reside in the npm package ecosystem, an enormous collection of JavaScript libraries controlled by the Node.js JavaScript runtime (see 2.2).
Both of these programming languages present excellent options for applications like DartDraw, but our project team was much more familiar with JavaScript than Swift, making JavaScript the clear choice for a low entrance barrier to this project. As an added bonus, the use of JavaScript and certain libraries and frameworks described below allowed us to make DartDraw a cross platform application, including the potential for a web-based version of the application.

2.3. The Electron Framework

The goal of this project was to create a user friendly, vector-based, robust graphics desktop application. To that end, we did not want to spend excessive time worrying about boilerplate coding for app frames, menus, and the like. However, we realized that our goal could not be realized without considering how these basic components affect the user experiences. The Electron framework provided a solution to this conflict.

According to electronjs.org, “Electron is a framework for creating native applications with web technologies like JavaScript, HTML, and CSS. It takes care of the hard parts so you can focus on the core of your application.” To be more specific, Electron uses Chromium, an open-source web browser project started by Google. The browser structure acts as a container of sorts for an offline desktop

---

application. Consequentially, Electron-based applications are compatible with Mac, Windows, and Linux out of the box. Electron even handles native menus.

A key feature of Electron is its separation between the main process and the browser processes. Creation, manipulation, and destruction of browser processes are handled from the main process, which also handles app start up and menu options. Browser processes are the parent processes for each window of an Electron application. Importantly, this allows our team to create a multi-window application using a framework like ReactJS that is intended for single page applications. Each window of DartDraw is handled as a separate instance of the application that is controlled by the main process.

Electron runs on the Node.js runtime. As a result, Electron-based apps can leverage the entire body of the npm package ecosystem, including the ones described below.

2.3. React and Redux

React – often called ReactJS – is an open source JavaScript library used for creating interactive user interfaces. The key feature of ReactJS is its breakdown into components. A component is rendered independently according to input data and a state for each component. Each component can be further broken down into sub components, which means that applications built with ReactJS are very scalable. Importantly, ReactJS components can render according to developer-specified rules and any kind of data that is passed through the app.
Redux is another open source JavaScript library, and it is used to manage the state of the application. Redux stores the state of an application in an object tree called the store. Suppose a DartDraw user created a drawing to explain the Breadth-First Search algorithm. The user might have a graph with vertices and edges, and they might include that graph several times to demonstrate various stages of the algorithm. It is easy to imagine an object tree with the drawing state at the root, and groups representing the graphs as children. Each of the groups would have vertices, edges, and labels as children, and each of those would have children representing information like dimensions, transformations, and a unique id. Although this is a simplified explanation of DartDraw’s Redux state, it is sufficient to explain the basics of the internal structure. When the Redux state needs to be changed, an action is dispatched to do so. This idea is explored further below.

Although React and Redux are separate libraries, they are often used together. There is even another JavaScript library called React-Redux that seamlessly integrates the two. The state of the drawing is maintained by Redux while the visualization of the drawing is handled by React. When we want to change the drawing, we need to change both. React-Redux handles this with its connect module. When an action is dispatched by Redux to change the Redux state, the connect module maps the changing parts of the state to input of the appropriate React components, thereby causing them to re-render.
By utilizing these three libraries in conjunction with one another, DartDraw only changes the parts of the internal state that it needs to. At the same time, individual components of the visual drawing re-render independently. A DartDraw user therefore experiences a fast and lightweight application that lets them change the drawing without any noticeable delay.

### 2.4. Scalable Vector Graphics

Because the Electron framework is based on the Chromium project, our app is able to leverage Chrome browser support for rendering Scalable Vector Graphics (SVG), an XML-based standard for representing two-dimensional graphics that was adopted by the World Wide Web Consortium. Notably, SVG supports interactivity, making it ideal for a drawing app within a Chromium shell.

In order to utilize SVG as our graphics standard, we based the internal structure of the Redux store on the same standard. In other words, the parameters for various SVG shapes and figures are the same as the parameters for DartDraw figures in the Redux store. When a user manipulates a shape using the interactive components of SVG such as click or drag, they also manipulate the shape’s representation in the Redux store, which in turn re-renders the corresponding React components.
2.5. Additional Libraries

An application of this size and complexity required that we avoid reinventing the wheel when possible. As a whole, our team used many different libraries along with React, Redux, and React-Redux. For the purposes of exporting EPS files and handling inter-process communication, I personally relied on two distinct libraries:

- **math.js** is a math library for Node.js. Its core functions do not differ in any meaningful way from JavaScript's built-in math library except that math.js allows for calculations with arbitrary precision. For complex calculations involving large, irrational, or complex numbers (e.g. solving quadratic equations), math.js is not only helpful but essential as well.

- **transformation-matrix-js** is “an affine transformation matrix class for JavaScript that performs various such as rotate, scale, translate [...] and converting to and from an SVG/DOM matrix.”\(^3\) In DartDraw, this library is useful in converting user actions on shapes (transforms) to transformation matrices stored as child nodes of shapes in the Redux store. These transforms can propagate down through the shapes in the Redux drawing state when performed on groups. When computing bounding boxes, it is sometimes useful to work with canonical forms of shapes before applying the appropriate transforms via this library.

---

3. Rendering Shapes

3.1. A Note on PostScript

A distinct advantage when it comes to exporting EPS files from DartDraw is that EPS is a vector graphics format, allowing it to easily represent the scalable vector graphics of our app. EPS also has the ability to define procedures and apply geometric transformations.

However, there are notable differences between the representations of figures rendered by EPS and DartDraw figures described in the application’s Redux store. Most of these differences arise from the stack-based paradigm of PostScript. All literals such as integers and floats are placed by the (Encapsulated) PostScript interpreter on the stack. PostScript operands pop their required number of arguments off of the stack before executing.

Section 3.2 describes the conversion process from Redux representations of SVG to figures rendered by EPS. Following that section, the remainder of chapter 3 describes how PostScript operands are used to render the different types of shapes supported by DartDraw.

3.2. DartDraw’s Export Landing Point

The React components that handle menu actions are found within DartDraw/src/components. The file LeftMenu.js handles actions related to tool
selection, shape manipulation, and exporting EPS files. Each of these actions is handled by buttons within the LeftMenu render method, which binds button clicks to various methods of the LeftMenu component.

When the export button is clicked by a user, the `generateEps` function from `DartDraw/src/eps/eps.js` is called with the drawing state node of the Redux store as an argument. This simple function looks through all figures in the Redux store by ID and handles the postscript generation for each figure. Then it appends the generated EPS to an Adobe EPS preset template. It also creates an instance of the `bbox` class to compute the bounding box of the total drawing, which it writes into the Adobe preset template. Finally, the function invokes the save dialog and saves the above information to a file that is named by the user.

For each figure in the drawing state of the Redux store, a new instance of the `epsShape` class is created. This class - found in `DartDraw/src/eps/epsShape.js` - is responsible for parsing the figures and generating the corresponding EPS. This abstract class contains methods such as `getStrokeWidth`, `getCoords`, and `produceEps`, all of which are implemented by the corresponding classes for each shape. The `produceEps` method returns EPS, and it is called from within `eps.js` to generate the content for each saved file.

Shape-specific classes are discussed below.
3.3. Line-Based Shapes

PostScript was created and is primarily used as a standard for printing documents. Consequentially, the PostScript graphics commands can be thought of as descriptions for a pen’s movement on paper.

For example, the PostScript \texttt{moveto} command takes two arguments: an \texttt{x} coordinate and a \texttt{y} coordinate. When this command is executed, it takes the top two values off of the stack and treats them as the \texttt{y} value and \texttt{x} value, respectively before “picking up the pen” and moving it to the specified location. It is worth noting that PostScript always has a notion of the current location relative to the origin of the page at the bottom left-hand corner, whereas the DartDraw app describes coordinates in relation to the origin at the top left-hand corner.

Another command called \texttt{lineto} also takes an \texttt{x} coordinate and a \texttt{y} coordinate. It puts the pen on the page at the current location and draws a straight line to the given coordinate.

This is where drawing commands such as \texttt{lineto} become slightly more nuanced. Ink is never rendered in PostScript-generated documents unless the command \texttt{stroke} or \texttt{fill} is called. Instead, the PostScript interpreter maintains a \textbf{current path} that is unrealized until a drawing command is called. This particular feature of PostScript allows DartDraw to create several lines with a \texttt{moveto} command and several \texttt{lineto} commands before rendering them all at once with the \texttt{stroke} command.
Two other commands are worth mentioning before discussing specific shapes. The `newpath` command resets the current path, a feature it shares in common with the `moveto` command. The `closepath` command draws a line in the current path from the current point to the start of the current path. If the rendered lines do not make a cohesive outline, PostScript automatically fills in the gap. This command is often used before calling `fill`.

The following shapes are rendered in a very similar fashion. As we shall see in chapter 4, however, their bounding boxes are rather more complex.

**Lines**

EPS for lines is created in `DartDraw/src/eps/epsLine.js`. This `epsLine` object’s constructor takes a shape object as a parameter to the constructor and assumes that the shape given represents a line. It takes the first point in the line’s `points` attribute and applies the proper transformations utilizing the `transform-matrix.js` library and the line’s `transform` attribute. Then, it accounts for the discrepancy between PostScript’s coordinate system origin and DartDraw’s coordinate system origin by subtracting the resulting `y` coordinates from the application’s canvas height.

After obtaining the required coordinates, the `produceEps` method of the object parses the `stroke` attribute of the line to get the stroke color and the parses the `strokeWidth` attribute to get the width of the line.
Finally, the `produceEps` method takes advantage of JavaScript's template feature that allows variable values to be substituted into a string. It creates an EPS snippet similar to the following:

```
0 0 0 setrgbcolor
5 setlinewidth
newpath
10 10 moveto
20 20 lineto
stroke
```

(Figure 1)

Which would create a black line that is 5 points wide from (10,10) to (20,20).

**Rectangles and Polygons**

Both rectangles and polygons are created in the same manner as lines with three key differences.

First, a wrap-around loop through the set of points in a rectangle or general polygon allows the shape outline to be closed.

Second, the commands shown in Figure 1 are repeated twice. In the first of the two identical sections, the `setlinewidth` command is omitted and the `stroke` command is replaced with `fill`. This sequence tells PostScript to render the fill of the shape before overlaying the outline.
Third, the `closepath` command is called after the end of each of the above sections. When the lines do not form a cohesive path (as in Figure 2), PostScript fills it in automatically (as in Figure 3). Though the difference is slight, we can see that the upper left-hand corner of Figure 2 is not quite right because the path is not closed.

(Figure 2)  (Figure 3)

**Free-Hand Paths**

Free-hand paths are represented as an array of very small lines between control points. They are rendered in the same way as polygons except that they are neither closed nor filled. Therefore, the `fill` and `closepath` commands are not used.

**Arrows**

SVG represents arrows differently than it represents other shapes. When drawing an arrowhead on a line, SVG first renders the arrow separately in an abstract viewing box, which is visualized in a user-friendly display within the DartDraw application. After the arrow is rendered, it is attached to the appropriate line by connecting a marker on the arrowhead to a marker on the line, thus attaching it and displaying an arrow in the front end graphics.
Rendering arrows therefore requires one additional step in addition to the normal process of identifying and manipulating points as with other shapes.

First, we determine the arrowhead’s height and length `arrowLength` and `arrowHeight`, which is a matter of subtraction of points. In the case of barbed arrowheads, the arrowhead representation in the Redux store has an additional point to account for the indentation in the barbed arrowhead. We use this point to compute a value called `minorLength` that denotes the Euclidian distance between the tip of the arrow and the intersection of the line with the arrowhead. This distinction is shown in Figure 4.

![Diagram of arrow with minorLength and arrowLength](image)

After computing these values, it is necessary to consider the line to which the arrowhead will be attached in order to continue with rendering calculations. First, suppose that the line is horizontal and that the arrowhead is pointing towards (1,0) on the unit circle. Then we note that the tip of the arrowhead is at the end of the line.
We can call this point $P_{tip}$. The other two points on a triangular arrowhead (or two of the three remaining points on a barbed arrowhead) require that we first compute an intermediary point, which we can call $P_1$. Still assuming that the line is horizontal:

$$P_1 = (P_{tip}.x - \text{arrowLength}, P_{tip}.y)$$

Lastly, the remaining points $P_1$ and $P_2$ are computed relative to $P_1$.

$$P_1 = (P_1.x, P_1.y - \text{arrowHeight})$$
$$P_2 = (P_1.x, P_1.y + \text{arrowHeight})$$

Note that the only difference between a barbed arrowhead and a triangular arrowhead is that a barbed arrowhead has an additional point $P_3$, where

$$P_3 = (P_{tip}.x - \text{minorLength}, P_{tip}.y).$$

Now we must transform these points according to the actual position of the line. Fortunately, it is easy to get the angle of rotation $\theta$ of the line to which the arrowhead is attached. Then we only need to rotate the points computed above by the angle $\theta$ to get the actual coordinates of the arrowhead.

Finally, we use a moveto command and multiple lineto commands to connect these points in the same way as we would with a rectangle or other polygon. In the case of a triangular arrowhead, the order of these points is $P_{tip}$, $P_1$, $P_2$, and $P_{tip}$. If we have a barbed arrowhead, then the order is $P_{tip}$, $P_1$, $P_3$, $P_2$, and $P_{tip}$. 
In the case of an arrow “tail” as they are called within the app, we only need to set $P_i$ equal to the starting point of the line (points 0 and 1 in the line’s points attribute), compute the coordinates of the arrow’s tip relative to $P_i$ using the arrowLength, and process as above.

If arrows are flipped (according to the Boolean flip attribute in the arrow’s Redux state), we begin in the same fashion as above, swap $P_i$ and $P_{\text{tip}}$, and proceed as above, keeping in mind that we need to invert the way that we compute $P_1$, $P_2$, and $P_3$ relative to $P_i$ and $P_{\text{tip}}$ since the direction of the arrows is flipped.

While the logic above is all that is needed to render the arrowheads, we are not done. As it stands, we would be rendering arrowheads on a line with some specified stroke thickness. If we do not alter the line after rendering the arrowheads, then the thickness of the line will cause it to “poke out” past the tip of the arrow. Therefore, we need to alter the points attribute of the line before making it into a line object that can be rendered and influence the bounding box of the drawing.

As luck would have it, we already have the points that we need. By getting the produceEps method of the arrow class to return an object that includes data about the arrow’s points in addition to the EPS that we need to export, we can change the line to which the arrow is attached when we first parse the shape within eps/eps.js. Suppose that we have a line with a classic triangular arrowhead that is not flipped. The point at which we want the line to end is the point at which the line and the
arrowhead shape intersect, which is of course $P_t$. We can make similar comparisons using $P_t, P_{tip}$ or $P_3$ depending on whether we have an arrowhead or an arrow “tail” and whether or not they are flipped.

For a line to be modified in this way, we must first copy the entirety of the original line shape using DartDraw’s own `deepCopy` function. This way we can alter the points in the copy and use that to render the appropriate line and affect the bounding box. If we instead modified the original line object or modified a reference to the original line, it would change the points that the arrowhead uses as references for where to render itself, thus changing the location of the arrowhead and causing the same issue of the line “sticking out” that we were trying to fix in the first place.

In order to make sure that arrowheads and arrow “tails” are rendered on the correct lines, each line’s Redux state has four important attributes: `arrowHeadId` and `arrowTailId`, which determine which arrowheads within the arrows node of the Redux store to attach to the line, and `arrowHeadShown` and `arrowTailShown`, which are Boolean values that determine whether or not the arrowhead or arrow “tail” is rendered and affects the bounding box of the drawing.

### 3.4. Ellipses

PostScript has a very useful command called `arc` that takes a center point, a radius, a starting angle, and an ending angle as an argument. For the most part, this command
is useful in drawing circles or partial circles. However, with a couple of clever modifications, it is possible to use the arc command to draw general ellipses.

The first modification that we make is to apply PostScript coordinate system transformations. The commands translate, rotate, and scale can be used to affect the PostScript coordinate system as if the paper being drawn upon was being shifted, turned, or stretched.

The second modification that we make is to define a procedure in PostScript. Procedures are a powerful component of the language that allows users to modularize common tasks. Here, we use it to specify the order of transformations that are needed to draw complex shapes. The following procedure is used in drawing ellipses within src/eps/epsEllipse.js:

```
/ellipse 6 dict begin
  /angle exch def
  /yradius exch def
  /xradius exch def
  /yC exch def
  /xC exch def
  /savematrix currentmatrix def
  xC yC translate
  angle rotate
  xradius yradius scale
  0 0 1 360 arc
  savematrix setmatrix
(Figure 5)
```

The procedure shown in Figure 5 begins by declaring a dictionary for the variables, a necessary syntactical component of PostScript procedures. The items in the dictionary are the angle of rotation of the ellipse, its y radius, its x radius, the x
coordinate of its center point, the $y$ coordinate of its center point, and the current matrix. The \texttt{currentmatrix} command returns the current 3x3 transformation matrix of the PostScript coordinate system. It allows transformations to be applied and saved prior to a procedure such as \texttt{/ellipse} so that the same transformation matrix can be restored when the procedure exits.

As we can see from Figure 5, the \texttt{/ellipse} procedure applies transformations after defining its dictionary. Ellipses in the DartDraw Redux store are represented in the canonical form, so the procedure first makes a translation to reset the origin of the PostScript coordinate system, followed by a scaling in the $x$ and $y$ directions to stretch the circle into an ellipse. Finally, a rotation in the reverse direction of the rotation of the ellipse is applied so that the actual “ink” drawing on the PostScript coordinate system results in an ellipse rotated by the intended amount in the correct angular direction. Lastly, the procedure draws a canonical circle.

In order for this procedure to work, we first need to obtain the appropriate data for the ellipse. Each ellipse in the drawing.shapes.byId node of the Redux store has the following data: \{cx, cy, fill, id, rx0, ry0, stroke, strokeWidth, transform, type\}. The cx and cy variables are the coordinate to translate to after being transformed by the \texttt{transform-matrix.js} library. The transformed rx variable (call it $rx_1$) is found by transforming the center point’s $x$ coordinate, transforming the point on the edge of the ellipse with base coordinates ($center.x + rx$, $center.y$), and the transformed ry
variable \( ry_1 \) is found analogously. Finally, we have to obtain the angle of rotation, which we can do with the following method (shown in pseudo code).

Find the point that is \( rx_1 \) pixels right of the ellipse’s center; call this \( p_0 \)
Transform that point using \texttt{transform-matrix.js}: call this \( p_1 \)
Compute the vector from the center to \( p_1 \); call this \( v \)
The cosine of the angle of rotation for \( v \) is equal to \( \frac{v_1.y \cdot ry}{ry^2} \)
The above simplifies to \( \frac{v_1.y}{ry} \). Call this value \( c \)
Let \( \alpha = \arccos(c) \)
If \( v.y < 0 \), \( \alpha = -\alpha \)
Return \( \alpha \)

(Figure 6)

When all of the variables have been computed, the ellipse procedure is called. At the end of the procedure, the \texttt{setmatrix} command restores the previous transformation matrix of the PostScript coordinate system. This step is crucial because it occurs before the \texttt{fill} command is called. If it did not, then the stroke would be affected by the ellipse transformation. In the final drawing, the outline of the ellipse would be inconsistent in width.

3.5. Arcs

Arcs in DartDraw can be viewed as partial outlines of ellipses. In fact, we only need three additional variables to render arcs – though as we will see, bounding boxes require several additional considerations.

The first two variables are the start and end angles of the arc, measured by counterclockwise rotation from \((1,0)\) on the unit circle. The start and endpoints of the rotation of the arc are given by the points attribute. Using these, we can calculate
the two angles in the same way that we calculated the angle of rotation of an ellipse. However, we must be sure to calculate these angles before applying any transformations. Otherwise, transformed points may be mistaken for non-transformed points, resulting in inaccurate angles.

The third additional variable that is needed to render arcs is \texttt{flipArc}. This is a special SVG component of the arc object that tells us which direction the arc is drawn. By default, \texttt{flipArc} is false, which means that the arc is drawn counterclockwise between the start and the end angle of the arc. If \texttt{flipArc} is true, the arc is drawn clockwise. To represent this difference in PostScript, the \texttt{arcn} command is used instead of \texttt{arc}, which tells PostScript to draw the arc in the correct direction.

The PostScript procedure for arcs is as follows:

\texttt{/arc 8 dict begin
    /rotateAngle exch def
    /secondAngle exch def
    /firstAngle exch def
    /yradius exch def
    /xradius exch def
    /yC exch def
    /xC exch def
    /savematrix currentmatrix def
    xc yc translate
    rotateAngle rotate
    xradius yradius scale
    0 0 1 secondAngle firstAngle arcDirection
    savematrix setmatrix
}

(Figure 7)

Here, \texttt{arcDirection} is equal to either “arc” or “arcn.”
3.6. Rounded Rectangles

Rounded rectangles are the same as normal rectangles in almost every way except that they have an $x$ radius and a $y$ radius at the corner. These values are equal to distances in the $x$ and $y$ direction from each corner towards the inside of the rectangle, where the resulting points are the centers of ellipses with the $x$ radius and $y$ radius of the rectangle as its own radii.

It is easy to render these shapes with a clever trick. First, we note that each corner of a rounded rectangle is a quadrant of an ellipse. If we remove each of those quadrants, a cross-like shape remains (Figure 8).

We can render this shape with two rectangles without a stroke. Getting the rest of the shape filled in requires that we render the ellipses first, then the cross-like figure over them to overlap the part of the ellipses’ outline that should not be showing inside the figure. Lastly, we render the four lines that connect the outlines of the ellipse quadrants.
As luck would have it, we already know how to render each of these shapes, and we can transform all of them simultaneously when needed using the transformation matrix of the rounded rectangle shape in the Redux store.

3.7. Bezier Curves

In DartDraw, Bezier curves are cubic rather than quadratic. Cubic Bezier curves are represented by two endpoints \( P_0 \) and \( P_3 \) along with two intermediary control points \( P_1 \) and \( P_2 \). They are described by the following equation:

\[
B(t) = (1 - t)^3 P_0 + 3(1 - t)^2 t P_1 + 3(1 - t)t^2 P_2 + t^3 P_3, \quad 0 \leq t \leq 1
\]

Bezier objects in the Redux Store have a points attribute that specifies points \( P_0, P_1, P_2, \) and \( P_3 \) for each curve. Once we rotate and translate these points according to the transform attribute of each curve, we can simply plug them into the curveto command of PostScript, as in the following example:

```
0 0 moveto
25 50 75 50 100 0 curveto
```

(Figure 9)

The above would move the pen to \((0,0)\) then make a semicircle curve to \((100,0)\) using the points \((25,50)\) and \((75,50)\) as control points.
3.7. Text

PostScript has a special *show* command that displays text on the screen. For it to work specified text must be on the stack directly on top of a specified font, as follows:

```
/Helvetica findfont
24 scalefont
setfont
1 1 1 setrgbcolor
newpath
0 0 moveto
(hello world) show
```

(Figure 10)

The EPS shown above uses the *findfont* command to look up the appropriate font in the PostScript dictionary. Just as the user-defined dictionaries we saw earlier to access the variables used in the */ellipse* and */arc* procedures, PostScript has built-in dictionaries for a variety of purposes, including pre-defined fonts. It also has a fallback procedure in the event that the user-specified font is not found; in this case, it typically prints in Helvetica.

The above EPS then scales the font to the appropriate size in points and sets the stroke color. It begins a new path, moves to the upper left-hand corner of the text to be written, and shows the specified text using the *show* command.
Scaling, translating, and rotating text is fairly straightforward as well, using commands that we have already seen in earlier procedures. For example, if we would like to rotate text by 90 degrees, we can do so as follows:

```latex
/Helvetica findfont
24 scalefont
setfont
1 1 1 setrgbcolor
gsave
e
newpath
0 0 translate
90 rotate
(hello world) show
grestore
```

(Figure 11)

Note that this procedure is largely the same as before with three key differences. First, we add a `rotate` command. Second, we use the `translate` command instead of the `moveto` command so that the origin of the coordinate system is set to the upper left-hand corner of the text and can be rotated around that point. Lastly, we use the `gsave` and `grestore` commands to save a reference to the current transformation matrix of the coordinate system prior to any transformations and to restore it after the text has been rendered.

The structure of the text box object in DartDraw is such that there is no automatic expansion of the text box to accommodate text that runs over. Instead, the user sees text wrap around to the next line. **Because there is no way to detect this wrap around behavior from the back end, it cannot be handled by the export functionality.** Therefore, we instead impose a couple of constraints upon the user.
First, any text that a user types before a carriage return is assumed to be on the same line. If the user sees the text wrap around in the DartDraw display, this is only a product of the fact that he or she has not made the textbox wide enough to accommodate the text. The corresponding EPS file will not render any text that is meant to be on the same line but does not fit within the text box. Second, the backend export functionality uses the newline character “\n” as a delimiter when parsing. The parser creates an array of strings to display as text, after which it applies any necessary transformations and stacks these strings on a line.

For example, to display the following string of text in black Helvetica, starting at coordinate (100,100), and rotated by 45 degrees, the export functionality would create the EPS fragment shown below it:

"Good morning, sunshine!  
The Earth says:  
Hello!"

(Figure 12)
4. Computing Bounding Boxes

The most important separating factor between EPS files and other PostScript files is the concept of a bounding box in EPS. The bounding box is specified in terms of the lower left-hand x coordinate (llx), the lower left-hand y coordinate (lly), the upper right-hand x coordinate (urx), and the upper right-hand y coordinate (ury). A large portion of my role in the DartDraw project was to compute ink-based bounding boxes for exported EPS files.

4.1. The Bounding Box Object

Using JavaScript allowed me to handle the creation of bounding boxes in an object-oriented fashion. The file src/eps/eps.js creates a bounding box object, which initially has arbitrarily high values for llx and lly and arbitrarily low values for urx and ury; that is, there is no bounding box as no figures have yet been rendered.

The main job of eps/eps.js is then to parse through a deep copy of the drawing state within the DartDraw Redux store to parse each shape. Each of these shapes is made into an epsShape object, where epsShape is an abstract class that defines several methods including `produceEPS`, `getAngle`, and `getCoords`. As it parses the drawing state and creates each epsShape object, eps/eps.js calls the `updateBoundsIfNecessary` method of the bounding box class, which takes an epsShape as an argument.

The bounding box object itself is defined within src/eps/boundingBox.js. The object’s `updateBoundsIfNecessary` method consists of a switch statement based on
the type of the given epsShape. For example, if the type of the shape is “ellipse,” the method appropriately calls its helper method `updateBoundsEllipseHelper`. All such helper methods have the ability to manipulate the llx, lly, urx, and ury values of the bounding box object based on the parameters of the epsShape object that it considers.

Below, I discuss how the bounding box of each figure in DartDraw is computed.

### 4.2. Lines and Freehand Paths

For lines and for all shapes, it is important to first note that a stroke in PostScript is divided on either side of a point. For example, a vertical line of stroke width 5, increasing in y value and ending at (10,10) would spread on either side to corner points of (7.5,10) and (12.5,10) (see Figure 9).

(Figure 13)
Second, we must remember that the origin of the DartDraw coordinate system is in the upper left-hand corner while the origin of the PostScript coordinate system is in the lower left-hand corner. It’s therefore necessary to subtract the $y$ value of a shape from the canvas height of the DartDraw window whenever that $y$ value comes from the Redux state representation and we wish to convert it an EPS representation.

The bounding boxes of lines are fairly easy to compute and form the basis for the bounding box computations of other figures. Given two points $P_1$ and $P_2$, we can assume WLOG that the vector of the line is towards $P_2$. We can get the $x$ and $y$ components of this vector by subtracting the corresponding points. Then, we can find the angle of rotation of the vector from $(1,0)$ on the unit circle as follows:

\[
\Delta x = P_2.x - P_1.x \\
\Delta y = P_2.y - P_1.y \\
\theta = \tan^{-1}(\Delta y, \Delta x)
\]

Note that the implementation of this equation within the DartDraw back end uses the $\texttt{atan2}$ function in place of $\tan^{-1}$, the arctangent. That is because $\texttt{atan2}$ can distinguish between diametrically opposed angles and has a range of $[-\pi, \pi]$ while $\tan^{-1}$ cannot do so and is limited to a range of $(\frac{-\pi}{2}, \frac{\pi}{2})$.

The ends of the lines are flat, which means that the edge of this flat line segment is orthogonal to $\theta$. We can call this angle $\phi$, and $\phi = \theta + \frac{\pi}{2}$ for the point of the line end that is rotationally clockwise relative to $P_2$. We can use this point as a reference to
compute the $x$ and $y$ components of the vector between line endpoints and the
corners of the non-zero stroke line. Calling this vector $\nu$, we set:

$$v_x = \cos(\varphi) \times \frac{\text{strokeWidth}}{2} \quad \text{and} \quad v_y = \sin(\varphi) \times \frac{\text{strokeWidth}}{2}.$$ 

Our four corner points are thus equal to:

$$(P_1.x + v_x, P_1.y + v_y)$$
$$(P_1.x - v_x, P_1.y - v_y)$$
$$(P_2.x + v_x, P_2.y + v_y)$$
$$(P_2.x - v_x, P_2.y - v_y)$$

Then we need only compare these values to the bounding box coordinates and
update the bounding box accordingly.

4.3. Rectangles

Because rectangles are rendered using lines, computing the bounding box of
rectangles is a very similar process to the one used for computing the bounding box
of lines. However, as we noted in section 3, PostScript fills in the corners of closed
shapes rendered with lines after closepath is called, and we must account for this.

Because lines only intersect at angles of $\frac{\pi}{2}$ in rectangles, we know that the additional
ink filled in by PostScript between rendered lines is a square. Furthermore, we
know that the sides are of length $\frac{\text{strokeWidth}}{2}$ and that the diagonal is of length

$$\frac{\text{strokeWidth}}{2} \times \sqrt{2}.$$ If we can find the angle of the vector $\nu$ from each intersection of
lines to the corresponding corner of the additional filled-in box, then we can find the coordinates of the corner of each filled-in box.

In fact, we can say that this angle is \( \varphi = \theta - \frac{\pi}{4} \) (where \( \theta \) is computed in the same way as it is for lines) because the hypotenuse of the filled-in square bisects the right angle. Then we set \( v_x = \cos(\varphi) \times \frac{strokeWidth}{2} \times \sqrt{2} \) and \( v_y = \sin(\varphi) \times \frac{strokeWidth}{2} \times \sqrt{2} \). For each line in the rectangle from \( P_1 \) to \( P_2 \), there is only one such corner point of a filled in box. Once we have \( v_x \) and \( v_y \), this point is:

\[
(P_2.x + v_x, P_2.y + v_y)
\]

Then we compare this point to the bounding box coordinates and update the bounding box.

### 4.4. Polygons

Just as the we needed to add an extra step to move from the bounding boxes of lines to the bounding boxes of rectangles, we must add yet another step to move from the bounding boxes of rectangles to the bounding boxes of polygons.

Unlike rectangles where the corners are always angles of \( \frac{\pi}{2} \), the corners of polygons can be any angle. If this angle is greater than \( \pi \), it is not an extremum of the polygon, so it can be ignored. Suppose that points \( P_1 \), \( P_2 \), and \( P_3 \) make a corner at \( P_2 \). If the
vector from $P_1$ to $P_2$ has the same sign for its $x$ and $y$ components as the vector from $P_2$ to $P_3$, then the angle of the corner is greater than $\pi$, meaning that we don’t have to consider this corner when computing the bounding box.

Otherwise, we must compute the angle $\omega$ between these two vectors for use in the computation of the length and angle of the vector from the actual corner to the corner of the filled-in space. We can use the dot product rule, which states that:

$$A \cdot B = |A| \times |B| \times \cos (\omega),$$

so

$$\omega = \cos^{-1} \left( \frac{A \cdot B}{|A| \times |B|} \right)$$

The filled-in region for the corner of a polygon is a quadrilateral that can be bisected into two triangles. This dividing line is the vector $\nu$ that we are looking for. Since we know the length of one of the sides is $\frac{\text{strokeWidth}}{2}$, and the angle opposite this side is $\frac{\omega}{2}$, we can find $|\nu|$.

$$|\nu| = \frac{\text{strokeWidth}}{2} \times \csc \left( \frac{\omega}{2} \right)$$

Then we find the angle of rotation of $\nu$ from the point (1,0) on the unit circle, and we call this angle $\varphi$. After finding the angle $\theta$ of the vector from $P_1$ to $P_2$ (computed in the same way as it was for lines), we can compute $\varphi$ using the parallelogram rule.
Finally, \( v_x = \cos(\varphi) \times |v| \) and \( v_y = \sin(\varphi) \times |v| \), so we check whether or not we need to update the bounding box based on the point \((P_2.x + v_x, P_2.y + v_y)\).

### 4.5. Ellipses

The following derivation of the ellipse bounding box equation is based on the work of Mark C. Hendricks, PhD.

To begin, we assume that the ellipse is centered at the origin since we can offset the resulting bounding box by the appropriate amount after the fact. Suppose that we have such an ellipse that has been rotated around the origin by an angle \( \alpha \). Then we recall that equation for a standard ellipse with horizontal radius \( h \) and vertical radius \( v \):

\[
\frac{x_0^2}{h^2} + \frac{y_0^2}{v^2} = 1
\]

Here, \((x_0, y_0)\) is a point on the boundary of the ellipse after the ellipse has been rotated in reverse by angle \( \alpha \) to make a standard ellipse. Now we can find \( x_0 \) and \( y_0 \) in terms of the points \( x \) and \( y \) on the boundary of the original non-rotated ellipse.

\[
x_0 = x \cos(\alpha) + y \sin(\alpha)
\]
\[ y_0 = y \cos(\alpha) - x \sin(\alpha) \]

Substitution gets us the equation of a rotated ellipse.

\[
\frac{(x \cos(\alpha) + y \sin(\alpha))^2}{h^2} + \frac{(y \cos(\alpha) - x \sin(\alpha))^2}{v^2} = 1
\]

Expanding and setting the right-hand side of the equation to 0 gives the following.

\[
v^2x^2\cos^2(\alpha) + 2v^2xy\cos(\alpha)\sin(\alpha) + v^2y^2\sin^2(\alpha) + h^2y^2\cos^2(\alpha) - 2h^2xy\cos(\alpha)\sin(\alpha) + h^2x^2\sin^2(\alpha) - h^2v^2 = 0
\]

When we hold \( y \) constant, we get a quadratic equation for \( x \) with the following coefficients:

\[
a = v^2\cos^2(\alpha) + h^2\sin^2(\alpha) \\
b = 2v\cos(\alpha)\sin(\alpha)(v^2 - h^2) \\
c = y^2(v^2\sin^2(\alpha) + h^2\cos^2(\alpha)) - h^2v^2
\]

Using the quadratic formula, we get one solution when \( b^2 - 4ac = 0 \). Using the variables defined above and a little bit of algebra to solve for \( x \), we find that:

\[
x^2 = \frac{-ah^2v^2}{\cos^2(\alpha)\sin^2(\alpha)(v^2 - h^2)^2 - a(v^2\cos^2(\alpha) + h^2\sin^2(\alpha))}
\]
We get the x coordinates of the bounding box for the rotated ellipse by taking the square root of both sides of the above equation. Lastly, we must offset the two x coordinates by the x coordinate of the ellipse’s true center to account for our initial assumption that the ellipse is centered at the origin.

The y values of the bounding box of the rotated ellipse are computed analogously. The parameters used in the above equation are derived according to the process described in section 3.

4.6. Arcs

An arc is simply a partial outline of an ellipse. The logic for finding the extrema is fairly simple while the math to do so is rather cumbersome.

The procedure is as follows:

Find the minX and maxX of the base ellipse
For each of these values:

Find the corresponding y coordinate using the implicit quadratic equation
Find the angle of rotation of this x,y pair
Determine if this angle lies between the start and end angle of the arc

If it does:
    This is an extremum. Update the bounding box
If it does not:
    This is not an extremum.

Repeat the above for minY and maxY.
Test the coordinates at the start and end angle of the arc, accounting for stroke width

(Figure 14)
Finding the corresponding y coordinate to minx (for example) is the difficult part of this procedure and the only one with math that we have not already examined.

To solve this, we note that we can also represent ellipses using the implicit quadratic equation:

\[ Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0 \]

The values for the coefficients are as follows, with \( h, v, \) and \( \alpha \) defined as before.

\[
\begin{align*}
A &= h^2 \sin^2(\alpha) + v^2 \cos^2(\alpha) \\
B &= 2(v^2 - h^2) \sin(\alpha) \cos(\alpha) \\
C &= h^2 \cos^2(\alpha) + v^2 \sin^2(\alpha) \\
D &= -2Ax_{\text{center}} - By_{\text{center}} \\
E &= -Bx_{\text{center}} - 2Cy_{\text{center}} \\
F &= Ax_{\text{center}}^2 + Bx_{\text{center}}y_{\text{center}} + Cy_{\text{center}}^2 - h^2v^2
\end{align*}
\]

Then the coefficients of the quadratic equation to solve for the x coordinate, for example, are:

\[
\begin{align*}
a &= A \\
b &= By + d \\
c &= Cy^2 + Ey + F
\end{align*}
\]

We have one solution for \( y \) when \( b^2 - 4ac = 0 \). We can solve for \( x \) given the \( y \) coordinate of an extremum in a similar fashion, though we need not show all of the equations.
4.7. Rounded Rectangles

Since rounded rectangles are created from DartDraw rectangles and ellipses, the bounding box of a DartDraw drawing is altered separately by each of the component shapes.

4.8. Bezier Curves

In section 3, we saw that four points in the coordinate system define each cubic Bezier curve used in DartDraw. Although Bezier curves are fairly easy to render, computing the bounding boxes of these shapes is a difficult process that is beyond the scope of this thesis.

However, these four points are easy to access within the Bounding Box class. The updateBoundsBezierHelper method leverages the getCoords method of the epsShape class to get the four points needed to define a Bezier curve. Every cubic Bezier curve is bound by the polygon described by its four control points. Therefore, the Bounding Box class checks each point with an offset of half of the strokeWidth of the Bezier curve to see if this bounding polygon pushes past the edges of the current DartDraw bounding box. While not precise, this method is accurate.

5. File Saving and Opening

As discussed throughout this thesis, the internal structure of the DartDraw application rests entirely upon the Redux framework. What makes the Redux
framework so useful is that it stores exactly what we need to know about the state of a drawing with no missing or superfluous information. Therefore, it is the basis of the DartDraw file save format.

5.1. Saving files with IPC

In the introduction to this thesis, I discussed the relationship between the Electron main process and the browser processes. Because electron processes handle menu events and browser processes can render separate drawings from each other, this relationship becomes important in the file save process.

The menu is created using the Electron Menu module, which allows a developer to create a template menu format and set that as the native menu. In this case, we have OS X menu options for saving and opening, along with other functionality not discussed here.

When a user decides to save a file, they click on File > Save within the Electron menu, which then handles the appropriate click event described in DartDraw/electron.js. In the case of a file save, the click event begins by getting the currently focused browser process – seen by the user as a window with a rendered drawing – using the command Browser.getFocusedWindow(). Next, it takes advantage of inter-process communication (IPC) to communicate between the Electron main process and this specific browser process.
Electron’s IPC module enables the processes to commute in this way. The idea is that the main process side “pings” a particular channel on the browser side using the ipcMain sub module, sending a message and an event along with the ping. The browser side uses the ipcMain sub module to listen for messages on the file save channel. When it receives a ping, it interprets the message as the main process asking for a file save. The browser process then accesses the Redux store, converts it from JSON to a string, and sends it back to the main process over the correct channel using the event that was sent by the main process.

When the main process receives a response from a browser process over the file save channel, it saves whatever it was given using Electron’s file save module. Specifically, this module opens an OS X native save dialog, allowing the user to input a filename. The main process saves the given string to this filename, checking to see if such a filename already exists and exiting gracefully when it encounters an error.

5.2. Opening files

DartDraw opens files in much the same way as it saves them, but in reverse. When a user opts to open a file from the DartDraw menu, the main process shows a native OS X open dialog through which the user may select one or multiple files. For each file that is selected, the main process attempts to open the file and the text into a JSON object representing the drawing state portion of a DartDraw Redux store. Upon each success, the main process creates a new browser process, instantiates the IPC file save channel in that process, and sends it a Redux store representing a
drawing. Finally, each new browser process parses its drawing state and renders it accordingly.

6. Future Work

Impressive as the DartDraw application may be, there are a few important ways in which the application could be improved. Below, I focus on the potential improvements that would fall under my responsibility.

6.1. Complex Shape Intersection

One useful feature of DartDraw allows a user to select a color of their choice as the canvas color. Whenever shapes are the same color as the background, we avoid expanding the bounding box of the total drawing based on that shape. We would not, for example, want an EPS file to represent a drawing of significant dimensions when the only figures present in the drawing are the same color as the canvas.

Without loss of generality, let's assume that the canvas is white. A white rectangle drawn by the user should not result in a call to any bounding box helper function because the bounding box should not expand. However, it may well be that this rectangle overlaps a red ellipse. An ink-based bounding box should account for all parts of this red ellipse that are not covered by the white rectangle, which means that we would have to compute the points of intersection between the two shapes.
Additionally, we have to make an extra pass through the list of shapes every time we consider updating the bounding boxes. If we were to draw the red ellipse first, the bounding box would have the necessary dimensions to encapsulate the red ellipse. If we were to then draw the white rectangle over the red ellipse, the bounding box would not update because the white rectangle is the same color as the canvas. This is a problem; we know have a bounding box that is not ink-based because it encompasses the area in which the part of the red ellipse that is no covered by the white rectangle used to be visible. Therefore, we must instead “look ahead” through the list of shapes in the Redux store each time that we consider updating the bounding boxes. If we take this approach, we would consider the red ellipse and realize that it is going to be overlapped by the white rectangle down the road, so we can update the bounding box according to the points of intersection between the two shapes instead of naively considering each shape only in isolation.

Currently, DartDraw export functionality accounts for intersections between lines, rectangles, rounded rectangles, and ellipses. Future work would include accounting for intersection between more complex shapes including arcs, Bezier curves, freehand paths, and text.

6.2. Color Flattening

Each shape type in DartDraw has an associated context menu within the app that allows users to select different features of that shape. One feature that is common to all shapes is color, which may apply to the fill and/or the stroke of that shape. The
user can choose whether the color is represented by RGB or CMYK. In either case, the color in PostScript is represented in CMYK.

Fortunately for the front end of the application, the user may also manipulate the alpha channel of the color palette when styling their drawings. Unfortunately for the back end of the application – including exporting EPS files – this behavior is not supported. That means that there is no automatic way to include opacity for shapes dictated by EPS files.

Adobe handles this issue with a technique known as color flattening. The idea is that multiple shapes with some degree of opacity can overlap in any manner of ways, creating a new color at the areas of overlap. Of course this new color can be calculated with relative ease, but the areas of overlap can be very difficult to compute for an arbitrary number of overlaps. While the problem is solvable, it is beyond the scope of this thesis.

6.3. File Compression

The current version of DartDraw is capable of opening DartDraw files, recognizing when files are not DartDraw files and should not be opened, and can save files in the DartDraw file format. Furthermore, Jean Zhou has written a standalone Python script to convert MacDraw files into the DartDraw file format.
However, DartDraw does not currently support compressed files. One potential method that would be easily implemented in a scripting language like JavaScript is Huffman encoding, a compression technique that can greatly reduce file size when used on a known and finite set of symbols such as the ASCII characters used to represent the Redux Store in the DartDraw file format.

7. Reflection

Working on this project has been both challenging and extremely rewarding. From the beginning, I was opposed to the idea of the JavaScript-based React-Redux design that the majority of the group favored as it meant that I would have to learn a new programming language in addition to PostScript.

While getting up to speed was quite challenging, I can now confidently say that my colleagues made a wise choice to set up our project this way. The Redux API is incredibly easy to use and makes translation between the application’s information and the exported PostScript files much less painless than it might have been. Furthermore, I see the runtime benefits of using the React framework to individually apply changes to figures on the front end. Compared to Swift-based applications that I have made in the past using Xcode, DartDraw is quite a speedy and user-friendly graphics application.
There are certainly decisions that I would have made differently if I had the opportunity to go back and do this project again. Mostly, I would do all of the math that I ended up needing ahead of time. After learning how to use the React-Redux framework, reading up on some basic PostScript, and computing the bounding boxes for a figure or two, I was convinced that I was ready to start coding in earnest. In reality, there is no code written before a couple of months ago that I did not change at least a little bit in the past few weeks. As I realized more logical and efficient ways to organize this project – such as the connection between rendering arcs and ellipses, or how to apply quadratic solvers to both arcs and Bezier curves – I came to admit to myself that my initial efforts belied the lack of planning I did for the long term.

Even so, I am proud of the application that we have made. There are certainly bugs to work out and features to add, but I genuinely believe that users could enjoy their experience with DartDraw as it stands now. Whether we polish what we've made after we turn in these theses or it has to wait until another undergraduate fits together the final pieces, it's clear to me that we accomplished an impressive feat in software engineering with limited time, resources, and initial knowledge.

I am impressed by my colleagues Collin, Elisabeth, Emma, Jean, Luisa, and Michelle, and I am thankful to my advisor, Thomas Cormen.
8. References


“Electron | Build Cross Platform Desktop Apps with JavaScript, HTML, and CSS.”

Hendricks, Mark C. Rotated Ellipses and Their Intersections with Lines. PDF. March 8, 2012.


Wikipedia, s.v., “Ellipse,” last modified May 23, 2018,
   https://en.wikipedia.org/wiki/MacDraw

Wikipedia, s.v., “MacDraw,” last modified February 17, 2018,
   https://en.wikipedia.org/wiki/MacDraw