5-1-2018

DartDraw: The Design and Implementation of Global State Management, User Interaction Management, and Text in a React-Redux Drawing Application

Collin M. McKinney
Dartmouth College

Dartmouth College Computer Science

Technical Report TR2018-850

DartDraw: The Design and Implementation of Global State Management, User Interaction Management, and Text in a React-Redux Drawing Application

Collin M. McKinney

Collin.M.McKinney.18@dartmouth.edu

Introduction

This paper outlines the design and implementation of the DartDraw application. Our client required a solution that could import, edit, and export his textbook figures that are saved in the MacDraw file format. The client also had specific requirements for the features and user interface of the application, requesting that they emulate the simplicity and functionality of the original MacDraw application. A team of six students created DartDraw to fulfill the need for an intuitive drawing application that can import MacDraw files and export EPS files.

I will focus specifically on my contributions to the project, including a discussion of the application's core technologies, the application's design, and the implementation of the user interaction engine, the contextual menu, and the text engine.

Technologies

There were two main development platforms to choose from for developing this application. The first option was to use Swift and Xcode, Apple's own native development environment. On its developer page [2], Apple promotes Swift as a modern and powerful programming language and, indeed, many modern native iOS and MacOS applications are programmed in Swift. Developing an application on a native platform brings the benefits of performance and facilitating environment configuration for new developers. Swift apps are not cross-platform, but since this app was intended to be only for MacOS, we did not consider this factor. Swift also boasts a number of graphics and text libraries that would be useful for building a drawing application.

The second option was to build an Electron app. Electron is a library for building cross-platform applications using web technologies such as JavaScript, HTML, or CSS. Electron is a multi-platform implementation of Google's web engine, Chromium, built to receive data from a local web server [4]. An app built with Electron is just as platform-agnostic as a modern web app. Again, we did not prioritize cross-platform compatibility, but using Electron carries other benefits. Web technologies are actively growing and improving, and there are many libraries available that would facilitate the development of an application, such as React, AngularJS, or jQuery. For each of those frameworks, there exist thousands of open-source third-party libraries built for well-defined tasks such as animation, mathematical operations, or UI components. These libraries are valuable time-savers for a small team that has under a year to build an entire application. Moreover, half of the development team on this project had previous web development experience, but not a single team member had previous Swift experience. It was this factor that ultimately drove the decision to use Electron. With so little time, it was essential to minimize the learning curve and make full use of the development experience that we already had as a team.

After deciding to use Electron as a platform, we had to decide which technologies to build the actual web application with. Because the application required no network support, we were able to eliminate many options. Technologies such as AngularJS or jQuery would be excessive, as they are full web application frameworks, built to handle tasks like routing, form validation, and asynchronous API calls. At a high level, we were looking for libraries that could help us build the two main components of this application: drawing state management and a responsive user interface for displaying and interacting with the drawing state.

User Interface

For building our user interface (UI), React was a clear choice. Open-source and actively supported by Facebook, React is a UI framework that's declarative and component-based [3].

Because it’s component-based, we could easily reuse code and factor it out into its functional pieces. And being declarative, a UI component built in React needs no internal knowledge of the application’s state at any given time. The component receives data and callback functions as properties, displaying the data as specified and calling callback functions for the corresponding user interactions. Again, all of these traits are useful, but, most importantly, half of the team had at least some experience with React in the past, thereby reducing the learning curve and allowing the team to begin development as soon as possible.

State Management

Any drawing application needs a way to manage its drawing state, which consists of all of the shapes in a drawing and their properties, including position, dimensions, color, and more. In theory, the drawing state could be entirely managed within React, as React components are capable of maintaining their own state. The canvas component could maintain a list of all current shapes and the properties of each shape could be stored on the shape’s corresponding React component. This solution isn’t ideal because, in this application, state needs to be able to be both loaded and exported. In order to load state under a decentralized state model, each component would need to be initialized with its corresponding state, and in order to export, the state would have to be retrieved from each state and then synthesized to write to a file. This model is not fit for the purposes of our application, and it doesn’t take advantage of React’s declarative nature, which allows our UI to function without the components storing their state internally. The declarative nature of React and the nature of our drawing application called for a global state model.

Global state and React are a common combination, and the most common library for managing global state in JavaScript is Redux [8]. Redux can work with any UI framework, but the Redux documentation notes that it works especially well with React because of React's declarative nature [7]. React can define a UI in terms of a global state, and Redux can manage the storing, updating, and broadcasting of that state. Redux is relatively simple, consisting of three parts: the store, actions, and reducers. The store is simply a JavaScript object that contains the application's global state. Actions are a representation of UI events, which are dispatched from within the application. Reducers are responsible for responding to actions by altering the state depending on the action type and the payload included in the action. In practice, upon a user interaction, the application dispatches a Redux action, and the reducer responds to that action by altering the global state and rebroadcasting it to the rest of the application.

Design

After establishing our technology stack, the next step was to design our application in a way that accounts for all of the features requested by the client. There were two parts of our application that needed designing. The first, the component structure, determined the hierarchy of our React UI components. The second, the Redux state, determined the structure of our global state, including the drawing state and the application's UI state.

Component Structure

At a high level, the user interface of our application is split into two parts, the canvas and the menu. The canvas is a single React element that displays the drawing state and allows the user to interact with it directly by dragging or clicking either itself or its contents. The menu is composed of a few different React elements (the left menu and the top menu, for example), but they all serve the same purpose: providing the user with an interface to alter the drawing state without the interface itself needing any knowledge of the drawing state.

The canvas component iterates over all shapes in the drawing and renders them within its coordinate system as children of the canvas element. The canvas element receives properties

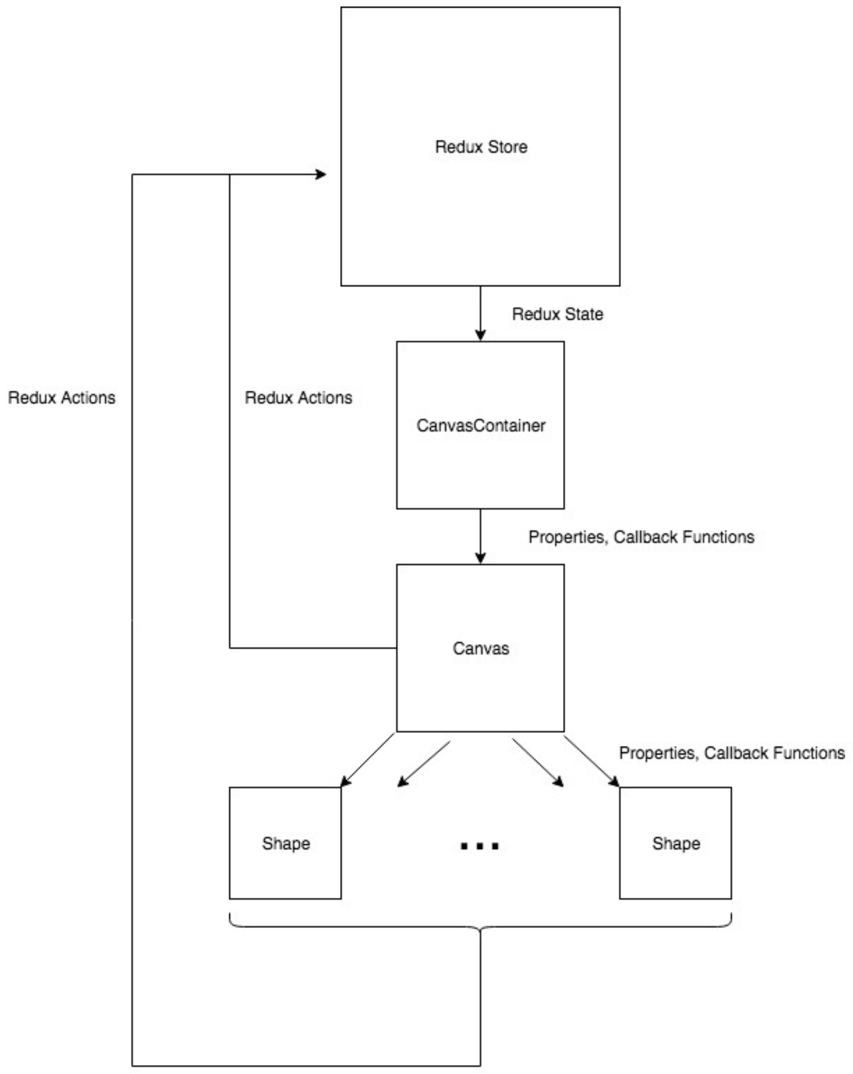

Figure 1: The design of the drawing state management flow relies on the propagation of data from the Redux store and the dispatching of actions from the canvas and shapes.

such as its dimensions and zoom scale, as well as callback functions for clicking and dragging the canvas. Every shape element is contained within the canvas element and receives its corresponding attributes, such as its location and dimensions, as well as its callback functions.

In order for the canvas component to receive data about the drawing state, it needs to connect to the Redux store via a container component. In this case, the canvas container component connects to the canvas component, a presentational component. The presentational components, or “dumb” components, need no knowledge of how the application state works; they need only to render their attributes and call the functions that are passed in as properties by the container components. The attributes passed by the container are derived from the global state, and the callback functions dispatch Redux actions, which in turn update and rebroadcast the global state. Figure 1 illustrates this flow. This design pattern allows us to achieve a separation between the user interface and the application state. Separating these concerns facilitates the team’s development process, as it allows the presentational components to be developed and tested without the finished Redux reducers and actions.

The menu components are also connected to the Redux store via container components, but they only receive menu state information (such as selected tool or selected fill color) and callback functions to dispatch Redux actions. Again, separating the management and displaying of the application state allows us to develop the entire user interface without the Redux logic, and then to later add the appropriate callback functions to the container components.

Redux State

The design of the Redux state, specifically the drawing state, was a critical step in the design process. The drawing state is the interface with which the other two components of this project, the importing and exporting applications, interact. The importing application converts MacDraw files from raw binary into the drawing state data structure, and the exporting application converts the drawing state data structure which is exported into PostScript. Because of these dependencies, it was essential to get it right the first time, in order to minimize future changes to the structure.

There are a few key pieces of information that need to be captured within the drawing state: every shape with its type and corresponding attributes, the z-index ordering of the shapes, and the shape groupings. The data structure also needs to offer performant retrieval of shapes by ID, as well as the ability to iterate through the shapes in the correct z-index order. The Redux documentation offers a solution: maintain a map of IDs to shape objects, as well as an array of shape IDs in the correct z-index order [6]. The Redux state also needs to support grouping, including the ability to nest groups. To address the issue of grouping, groups are stored in the same manner as shapes, with an ID that is stored in the array of shape IDs and a corresponding shape object. The group's shape object includes a list of member IDs, which can include other group IDs, allowing for nested groups. This state structure accounted for all the requested features.

Because the drawing state structure would also serve as the file format for the application, we strove to make it as human-readable as possible. We decided to emulate the SVG API as closely as possible, and tried to keep the naming conventions of every shape aligned with the SVG design specification. This decision was in the interest of keeping the file format human-readable, but also to ease the development of the importing and exporting pipelines, which could then use the SVG specification as a reference for the file format. With the application design completed, the team was able to begin the implementation.

Implementation

User Interaction Engine

a. Draggable component

The Draggable component is a higher-order component, which takes three functions as properties: onStart, onDrag, and onStop, calling them at the corresponding stages of a user mouse drag. The component is an extension of the DraggableCore component from the third-party NPM module, React-Draggable. Wrapping a React element with the Draggable component enables it to respond to a user dragging interaction. Each callback function receives a data structure which includes the *x* and *y* position of the mouse, the change in *x* and *y* position from the last function call, and the *x* and *y* position of the drag's origin. This component wraps the canvas and every shape within it, so that the user can interact with the canvas and all its shapes.

b. Canvas component

The Canvas component is the specification for the element that contains the entire drawing. As specified in *Component Structure*, the canvas needs to be able to handle user interactions with both itself and with its child elements, the shapes. We meet these requirements by making the Canvas component an SVG element wrapped by a Draggable element, which handles all drag events. The Canvas receives all of its properties through a container component, CanvasContainer, which maps the current Redux state to the Canvas's properties. The most important properties are all canvas display properties, the array of shapes to be rendered, and all of the callback functions. The canvas display properties include the width and height of the canvas, the zoom scale, and the view box, which are saved in the Redux state and altered through the menu. These properties are mapped directly from the Redux state into the Canvas component via the CanvasContainer. The array of shapes is generated in the CanvasContainer component by iterating through the z-indexed ordered array of shape IDs stored in the Redux state. The container maps each ID to its corresponding shape object, which is retrieved from the ID to shape dictionary also stored in the Redux state. Shapes are mapped from the Redux state to the shapes array through a recursive formatting function, so that if the shape is a group, the function calls itself recursively on the group's array of member IDs. The callback functions, such as onDrag or onShapeDrag, are generated in the container component, and they are wired to generate Redux actions which alter the drawing state. In the Canvas component, upon rendering, the canvas and each shape's element are provided with their corresponding callback functions.

c. Shape component

The Canvas component accepts different shape components as children. Shape components are a class of components that accept the following properties: onDragStart, onDrag, onDragStop, onClick, and propagateEvents, all of which are callback functions except for propagateEvents, which is a Boolean value. This application supports the following shape components: Arc, Arrow, Bezier, Ellipse, FreehandPath, Line, Path, Polygon, Polyline, Rectangle, and TextInput. A shape component consists of an SVG element, such as ellipse or rect element, wrapped by the Shape higher-order component, which enforces the required shape properties and also wraps the child element with a Draggable component to respond to user interactions.

d. Group component

In addition to accepting shape elements, the Canvas also accepts Group elements as children. The Group component accepts any number of shape elements as children and overrides their onDragStart, onDrag, onDragStop, and onClick callback functions. This allows the application reducer to process user interactions with the shapes in the group as an interaction with the entire group rather than an interaction with the individual shape.

e. Event propagation

Another detail that arose during the implementation of user interaction management is event propagation. The default behavior in React is for parent elements to respond to interactions with their child elements. In our case, this behavior meant that the canvas component reacted to drags or clicks that occurred within a shape. This behavior caused undesirable results, such as new shapes being drawn when a user tried to drag a shape to a different position. The React event API offers a solution to this. An event can be prevented from propagating to its parent elements with the stopPropagation function, which is stored on the event object. Thus, in our Draggable higher-order component, we can prevent propagation for any user interactions by calling stopPropagation in the callback function. However, in some cases, propagation is desirable. If a user wants to draw a new shape, for example, and clicks or begins dragging on an existing shape, the canvas rather than the shape should handle that event and create a new shape. Whether or not events should propagate to the parent elements depends on the currently selected tool.

To implement our solution to this problem, we created a propagateEvents property for the Canvas component. This is a Boolean value which is evaluated in the CanvasContainer, and depends on the selected menu tool. If the selected tool is for drawing shapes or panning, propagateEvents is set to true. If the tool is for interacting with shapes, such as repositioning, resizing, or rotating, then the property is set to false. The propagateEvents property is passed down to each shape component and into its Draggable element, where interactions are stopped from propagating if propagateEvents is false.

Contextual menu implementation

The contextual menu is distinct from other menus in this application because it depends on the drawing state. While the top and left menus connect only to the menu state portion of the

Redux state to display selected tools or selected colors, the contextual menu displays a selected shape's properties. The contextual menu's container component, ContextualMenuContainer, maps each ID in the Redux state's selected shape IDs array to its corresponding data object. If there are more than one selected shapes, the default behavior is to display the data of the first selected shape.

The contextual menu essentially acts as a form for a shape's properties. Whether it be width, height, stroke width, rotation, or position, the exact value should be available to view and edit in the contextual menu. This allows the user to meet exact specifications for shape attributes.

To implement this design, a single Redux action is needed: the edit shape action. The edit shape action takes a shape ID and an updated shape data object as payload, and then overwrites any attributes which differ between the updated shape data and the data that is stored in the Redux state. This action is connected to a callback function which is passed to the contextual menu via the ContextualMenuContainer. The callback function is then called whenever the user submits a change to the attributes form.

The overwriting of shape data with a series of attribute inputs may seem trivial, but SVG transformations complicate the implementation. Although the exact implementation of shape transformations is beyond the scope of this paper, the property that defines a shape's dimensional attributes in this application is the SVG transformation matrix, which encodes scale, translation, and rotation. In order to view and edit more human-interpretable properties, such as x and y position or width and height, we need functions that convert the SVG transformation matrix into these properties as well as their inverses. The contextual menu has been implemented for some shapes, but future work is still needed.

Text implementation

Text provided a unique challenge given the design of this application. With our design specification, the ideal text component would receive its text content and styles as properties, and process any user interactions by sending the updated text content to a callback function, allowing our Redux reducers to update the corresponding shape component and propagate the changes in the drawing state. There are many open-source text editor packages in the JavaScript ecosystem as well as the standard HTML text inputs, but none of them are designed to be explicitly controlled in this fashion. For the most part, web-based text editing libraries internally store text data, such as the content and style, and allow access either imperatively or through a callback function, but cannot be driven declaratively by properties.

To solve this problem, we used Draft.js, a text editing framework for React, designed to solve the problem of controlled text inputs. Draft.js's Editor component accepts two important properties, an editor state and a callback function for handling changes to the editor state. The editor state contains all information about the text content, the text styles, and the selection state, and upon any change to the text input, the callback function receives the updated editor state. With this component at our disposal, we simply had to devise a way to interface the Editor component's editor state with our Redux drawing state.

Although we could have easily solved this problem by storing an exact replica of each text input's editor state in Redux, that would go against our initial design decision to maintain a human-readable Redux state that closely resembled the SVG specification. Instead, we chose to develop an interface between our chosen text data format and the Draft.js editor state. By wrapping Draft.js's Editor component in our own custom TextInput component, we were able to manipulate the Draft.js editor state whenever any of the TextInput props changed. In the TextInput's componentWillReceiveProps lifecycle method, if the text's style or any of its tspans, which specify the style within a given interval, change, we alter the text input's editor state to match. Specifically, we iterate over all of the text input's tspans, and apply the styles using the applyInlineStyle function from the Draft.js Modifier module. This interface allows us to have a controlled text input, without compromising our human-readable, SVG-based Redux state.

Normally, a text input responds to click and drag interactions by changing the cursor position and highlighting text. However, the user also needs to be able to click and drag the text input box to change its position as they can with any other shape in the drawing. In order to resolve this conflict, we implemented a user interface in which the user can double-click the text box to focus the input. When the user first creates a text object, the input is automatically focused, but any click away from the text will blur the input. When the text box isn't focused, the user can select it and drag it around the canvas. To refocus, the user just has to double click the text box. To implement this dual behavior, we added an internal editing state flag to the TextInput component, which is set to true upon a double-click, and set to false when the component loses focus. This state flag switches the element that is rendered by this component. When it is set to true, the component renders the Draft.js Editor element, enabling standard text input interactions. When it is set to false, the component renders a standard HTML div with the appropriate text and styling. This div responds to drag and click events in the same way that any other shape on the canvas does.

Conclusion

This paper has outlined the design and implementation of some key components of the DartDraw application, including the global state, the user interaction engine, and the text engine.

References

[1] "API Basics." Draft.js. Accessed May 26, 2018. https://draftjs.org/docs/quickstart-api-basics.html.

[2] Apple Inc. "Swift." Apple Developer. Accessed May 26, 2018. https://developer.apple.com/swift/.

[3] Facebook. "React." React. Accessed May 26, 2018. https://reactjs.org/.

[4] Github. "About Electron." Electron. Accessed May 26, 2018. https://electronjs.org/docs/tutorial/about.

[5] Mzabriskie. "React-Draggable." GitHub. Accessed May 26, 2018. https://github.com/mzabriskie/react-draggable.

[6] "Normalizing State Shape." Redux. Accessed May 26, 2018. https://Redux.js.org/recipes/structuring-reducers/normalizing-state-shape.

[7] "Usage with React." Redux. Accessed May 26, 2018. https://Redux.js.org/basics/usage-with-react.

[8] "Redux." Redux. Accessed May 26, 2018. https://Redux.js.org/.