Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

A Handwriting Math Expression Input Component For Online Learning System

Author:
Jiawei Zhou

Supervisor:
Jason Bailey
Andrea Callia D'Iddio

Submitted in partial fulfillment of the requirements for the MSc degree in MSc Computing Science of Imperial College London

August 30, 2022

Abstract

Department of engineering of the college is developing an online learning system, Lambda Feedback, to enhance user experience on online assignment and assessment. The system can check the correctness of an answer to a question. The format of the answer includes mathematical expressions, but the system only supports keyboard input for equations. This project thus develops a handwriting mathematical expression input component for Lambda Feedback, and integrates it to the system. The component allows users to write mathematical expressions with mouse or pointer, recognises them and submits the result to the system. This report records the process of development, presents results and gives an evaluation of the project.

Acknowledgments

I would like to express my deepest appreciation to my supervisors, Jason Bailey and Andrea Callia D'Iddio, who guided me throughout the project. I would also like to thank the technical support provided by the Lambda Feedback team, especially Pierre Tharreau. Lastly, I wish to acknowledge my family and friends for their emotional support.

Contents

Chapter 1

Introduction

The application of remote learning in higher education has increased massively since the explosion of COVID-19 [1]. Many institutions including Imperial College London moved their courses online during lockdowns. These courses are taught based on online systems. However, some online systems are not capable of conducting the digitalisation of teaching tasks such as lecture recording/playing and assignment marking etc. For example, some departments of Imperial College London were using Möbius for online assignment and assessment, but students and staff found it cumbersome. This thus leads to a demand of updating or developing new systems so that students and teachers can get better experience.

Therefore, department of engineering of the college has decided to develop a new online learning system named Lambda Feedback. This system is a web application which allows students to do assignments posted by teachers and get feedback immediately. Comparing to its predecessor Möbius, Lambda Feedback has modern user interface, friendly interaction and much more features. One of the features is the expression input. In many assignment questions, solutions are math expressions. In Lambda Feedback, users can input math expressions in a text field, and the system is able to understand the expression and check if it is correct. Figure 1.1 is a screenshot of a demo in Lambda Feedback. The webpage displays a navigation sidebar, the question, and the answer area, where users can input the expression.

However, the current system only supports keyboard input for math expressions. Users need to type the whole expression including symbols and operators, which can be very inconvenient and complex especially for complicated equations. Consequently, an extension with a new method of input is needed for the system.

Handwriting input is considered appropriate for the system because of its convenience and simplicity. Although there are other input methods such as equation editor and image upload, they either require users to learn how to use, or need some extra steps. This means they are not suitable as the major approach of expression input, but can be a good supplement to handwriting input in some special situations.

This project thus aims to develop a component with handwriting input and equa-

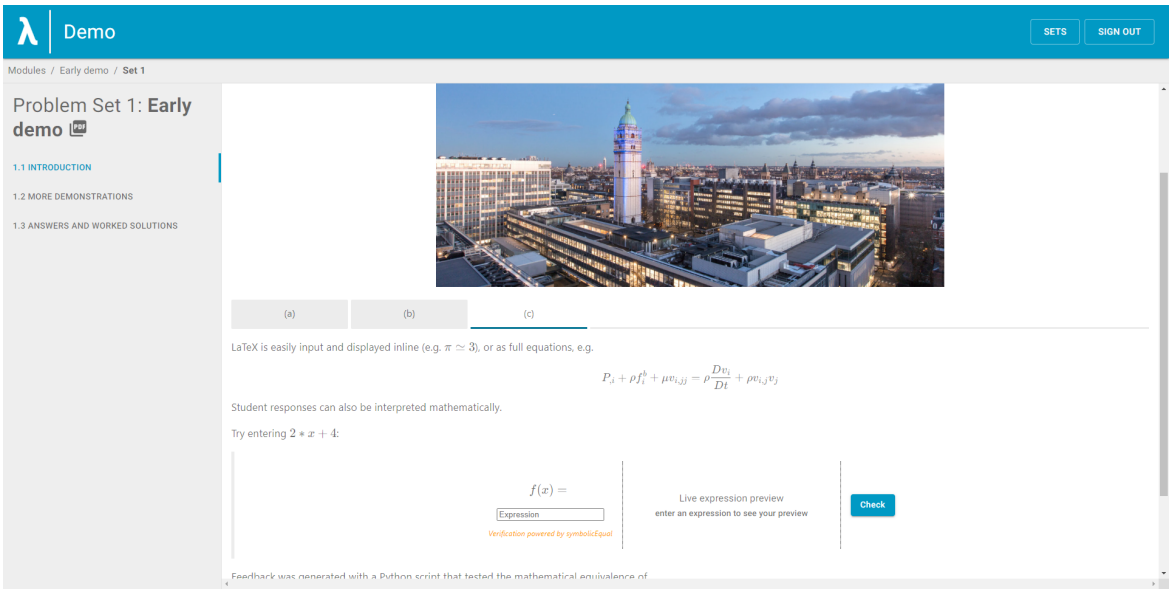


Figure 1.1: Screenshot of a Demo in Lambada Feedback System

tion recognition, and integrate it to the Lambda Feedback system. The component shall enable the users to write on it either via mouse/touchpad on computers or fingers/pens on mobile devices. The component should be able to recognise the equations written by the user, and display the text result alongside the writing area. Live update is needed, which means when the user finishes a stroke (i.e. a path is drawn and the user releases the pointer), the component should automatically conducts the recognition. Integration with the Lambda Feedback system should be carried out after the component is tested available.

This report records the development of the project. Chapter 2 investigates several approaches to realise the software component and analyses their advantages and drawbacks. Chapter 3 states the process of development and explains the implementation of the software component. In Chapter 4, progressive results are demonstrated and the final achievement is shown. An evaluation of the results are included in this chapter as well. Finally, in chapter 5, a conclusion of the project is drawn, and possible future improvements are listed.

Chapter 2

Background

This chapter records the background research, which includes general reviews on articles, tutorials and existing projects that have helped the development of this project, as well as potential choices that could be taken and why they were discarded, along with the reasons of selecting current tools. An overview of the project is made below before diving into details.

Overview

The component to develop is a TypeScript (strong typed version of JavaScript) web application which has two major functionalities:

1. Have a canvas where users can write equation
2. Ability to recognise the equation written on canvas and display the result

In addition, the application should include some functions for editing the user's drawing, such as undo, redo, delete all and remove stroke.

Moreover, the component should be integrated into Lambda Feedback, so it is necessary to introduce the system because they will use same frameworks and language.

Lambda Feedback is a web application which has a frontend and a backend. The frontend is built upon NextJS (a framework built on React), and the backend is built upon NestJS. Both are written in Typescript (These frameworks will be introduced in Chapter 2.1). Most code of the project will be installed in a component of the frontend named `ResponseArea`, which is responsible for receiving input and giving result to users.

It is difficult to develop the component directly upon Lambda Feedback, as it already had much code when this project launched. Therefore, it was decided to build a web application independent of Lambda Feedback to save time on reading code and testing functionality.

Although the final independent web application must be written under the same framework with Lambda Feedback, i.e. React, there is another way, Vanilla JavaScript (also referred to as plain JS), to start with. Table 2.1 compares the two methods:

Table 2.1: Comparison of Two Methods for Independent Web Application

Method	Advantages	Disadvantages
Vanilla JS	Easy to start & Require no extra configuration	Does not fit Lambda Feedback
React	Uses same framework as Lambda Feedback - does not need extra steps for integration	Need to study React before start

After careful consideration, it was decided to start by using Vanilla JS to get familiar with functionalities, and then converting the application to React. This makes the procedure more fluent and only need to focus on one emphasis for every update.

Overall, the procedure of the project is:

1. Develop a web application with Vanilla JS which enables users to draw equations, and recognise the equation
2. Move the web application under React framework
3. Integrate the application as a component to Lambda Feedback

Despite using different frameworks during the procedure, the approaches to realise functionalities are the same. Hence, the background research mainly focuses on three parts:

- React framework and integration with the Lambda Feedback
- A canvas where users can draw equations using mouse or pointer
- Functionality of mathematical equation recognition

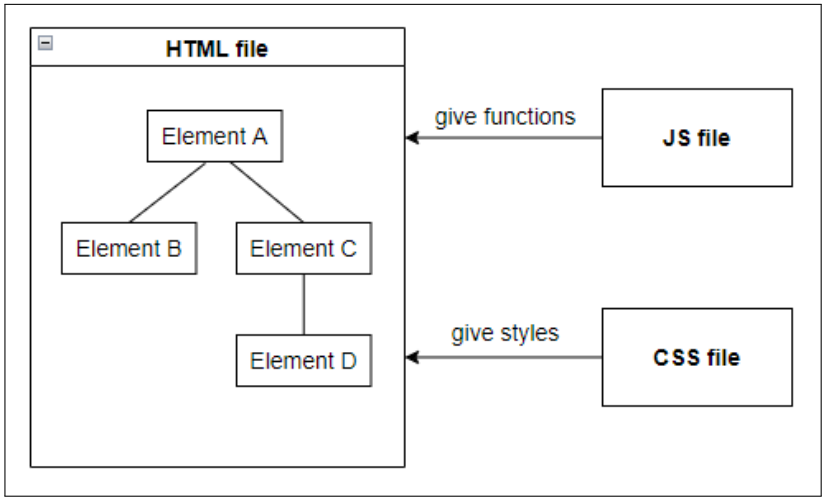
Therefore, the remaining part of this chapter is divided into three sections, each introducing one part.

2.1 React and Lambda Feedback

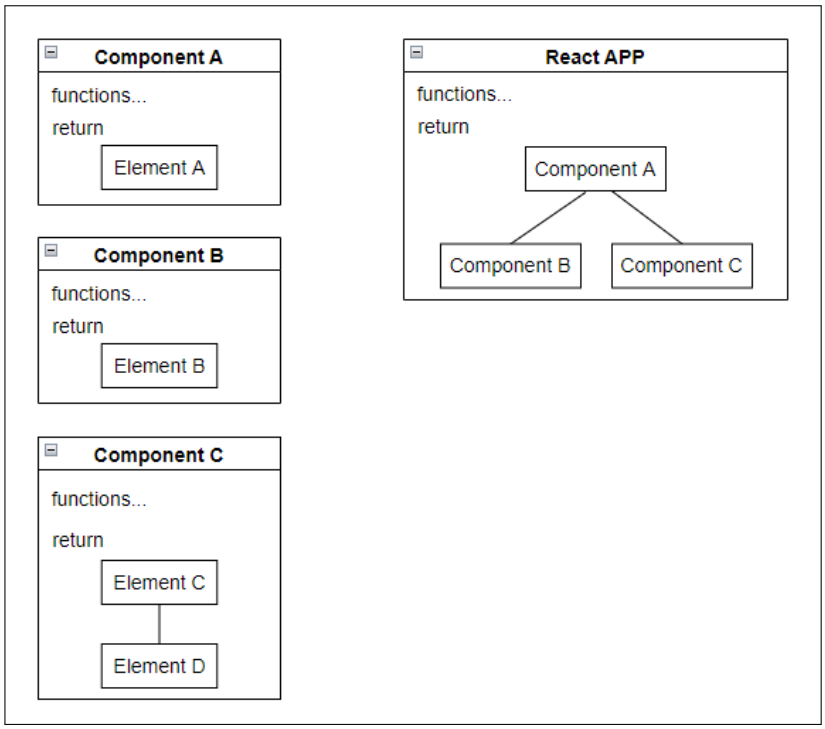
Lambda Feedback's frontend is built upon NextJS, a framework based on React. Distinct features of NextJS mainly focus on page level optimization, and the method of building a component remains the same as React. Therefore, it is necessary to learn React framework before building the component for Lambda Feedback, but only need to have a general understanding of NextJS.

React is an open-source frontend library deployed to develop reusable user interface (UI) components. Unlike Vanilla JS which structures all the layouts of a webpage in one file, React focuses on the idea of components [2].

In Vanilla JS, layouts, functions and styles of a webpage are written in HTML, JS and CSS file respectively. However, in React, the webpage is assembled by React components. This makes component the basic unit in React. Layout and functions of a component are in the same file whose type is JSX. Even styles can be embedded in it as well. This means only one file is needed for one component [2]. Figure 2.1 shows the design pattern of VanillaJS and React. It can be found from Figure 2.1



(a) Design Pattern of Vanilla JS



(b) Design Pattern of React

Figure 2.1: Comparison of Vanilla JS and React

that React is more flexible than Vanilla JS as its components can be plugged into other React APPs conveniently, whereas the structure of Vanilla JS is fixed.

The method of writing a React component in JSX is similar with writing a class. The class contains functions needed for the component, and has a return function which returns the layout of the component in HTML. Styles can either be imported from a CSS file, or specified in the JSX file.

A React component can have attributes, which can either be constants or functions (presented as function typed constant). Attributes of a component are defined by external classes, normally its parent component, when the component is used. Examples of attribites can be:

- Height, width...
- onClick - function happened in parent component when this component is clicked

The class can be simplified to a function typed constant when feature Hooks is introduced. The constant behaves exactly the same as a function but can be exported so that other components can use it.

Apart from the idea of JSX and component, Hooks is another important feature used in this project.

Without Hooks, React components were written in classes and had to contain many React features, which had made the development complicated. To solve this, Hooks were published. Hooks are essentially functions which encapsulate React features by using existing JavaScript features, so that developers do not need to learn extra syntaxes of React, but only need to focus on functions needed for the component [3].

There are three Hooks used in the project: `useState`, `useEffect` and `useRef`.

Hook: `useState`

`useState` function declares a state variable. An example of `useState` is as following:

```
const [count, setCount] = useState(0)
```

In this line of code, variable `count` is declared as a state of the component, and it can only be modified using function `setCount`. It is initialised to 0 which is a parameter of `useState`.

The state variables are useful in the component they are monitored by React, so that the component can 'react' to any change of a state variable. For example, the `count` variable is used in an element:

```
return (<div>{count}</div>)
```

This line returns the structure of the component, which simply displays the `count` variable. When `count` changes, React will automatically re-render this element so that the updated `count` is displayed.

Hook: `useEffect`

`useEffect` allows the developer to add side effect to the change of a state variable. An example is as following:

```
useEffect(() => {do something}, [state])
```

When `state` changes, function in `useEffect` is invoked.

Hook: `useRef`

`useRef` creates a reference to an element in the component, so that APIs of the element can be used, and other functions can refer to this element.

These are the main React features of that will be utilised in this project for the React component.

To integrate the component into Lambda Feedback, context in Lambda Feedback should be examined.

The component is a handwriting input tool. In Lambda Feedback, there is a component `ResponseArea`, where users can input their result and get feedback from backend. Inside `ResponseArea`, there is a type folder containing all the types of input. One of them is `ExpressionInput`. The handwriting input component will be installed inside the `ExpressionInput` component as an extension input method of expression input. It was considered firstly to add a new input type in `ResponseArea`, but this needs to change 15 files in frontend and 15 files in backend and can be difficult testing as well. Therefore, it was dicided to install it upon an existing type, `ExpressionInput`, so that minimal changes will be done to the big project, and is much easier to test.

Details of `ExpressionInput` and the process of integration will be presented in Chapter 3.2.

Moreover, Lambda Feedback has used many components from Material UI to structure and style the webpage. This project thus uses MUI as well for adaption to Lambda Feedback and convenience.

Chapter 2.1 has investigated features of React and the component where this project is installed in Lambda Feedback. The next section will then discover different choices to realise a handwriting canvas.

2.2 Canvas

A handwriting canvas is needed for the component, which should do the following:

- Users can draw and edit stuffs
- The canvas can output the content, either by image or stroke data

Here, stroke data means positions that represent points on the strokes. For example, stroke `{1, 1}, {2, 2}, {3, 3}` means a line starting at point (1, 1), passing point (2, 2) and ends at (3, 3). All paths drawn on the canvas can be expressed as stroke data.

Two HTML elements - `canvas` and `SVG` - were investigated for this part.

The `canvas` element is an HTML container used to draw graphics on web page. Graphics are drawn by APIs, which has several methods for drawing, including paths, boxes, circles, text, and adding images [4].

The component can utilize its paths method to display the user's strokes: once the user drags the pointer, the component fetches its position in the canvas, and draws a path to the next position the pointer moves to.

The `canvas` element also provides APIs to output content as images, while stroke data is generated when recording positions of the pointer.

Drawing on `canvas` is easy because all graphics is drawn by APIs, so the component only needs to 'tell' `canvas` how to draw. However, there are two drawbacks of this element:

1. To use `canvas` APIs, the HTML element should be referred. This in React will be using `React Refs`. Unlike in Vanilla JS, API callings in React should be carefully considered with *React Component Lifecycle* [5]. This will affect the performance of the component and brings extra work.
2. The `canvas` element is represented as a Bitmap in webpage. This means when transforming the size of the canvas, graphics might look blurry [6]. This will be further discussed in Chapter 3.

Figure 2.2 is a good example showing how graphics get blurry in `canvas` element when zooming in.

(a) Canvas Image with Initial Size **(b)** Canvas Image with Zoomed Size

Figure 2.2: Comparison of Unzoomed and Zoomed Canvas

In contrast with `canvas`, the `SVG` element does not have listed problems. The term SVG (Scalable Vector Graphics) means an XML based vector image format for defining 2D graphics. Therefore, as an element in HTML, `SVG` simply represents a container for SVG graphics.

Because it is 'vector graphic', regardless of the ratio of rendered size and programmed size, users can always get a clear graph of what they draw. This thus solve the blurry

problem of `canvas`.

`SVG` has several sub-element including `Path`. Data of a `Path` is a string containing commands and points, and is attached to its property 'd'.

To show strokes that a user draws, instead of containing them in one `canvas` element, multiple `Path`s are used to display the user's drawing, each for one stroke. It is more flexible for the programme to monitor and control them.

This feature fits *React Component Lifecycle* perfectly as any change of path data causes React to 'react' to it, and render the path with updated data.

Overall, `SVG` is more suitable than `canvas` for this project due to its vector property and compatibility with React.

Before converting the Vanilla JS application to React component, an existing React component, *React-Sketch-Canvas* [7], was investigated. It has most features this project needs as a canvas, and can be directly installed in Lambda Feedback. However, it only allows the user to erase on a stroke but cannot remove the whole stroke the pointer points. The remove stroke function is considered important as the canvas focuses more on writing than on drawing, and operations on strokes are more important than on the graph. Besides, the existing component has many functions that are not needed but still need to be configured when invoking. Therefore, it was decided to develop a custom canvas with `SVG` so that necessary functions are included and the size of the component is minimised.

2.3 Equation Recognition

An equation recognition function is required for the project. The function needs to correctly recognize the content which users draw as an equation. High accuracy is required for this project to ensure good user experience. In addition, response to recognition request should be quick so that live update is available.

Initially, building a model with machine learning was considered, and some tools such as TensorFlow and Pytorch were preliminarily explored. It then turned out that this approach is excessively time-consuming if a high-accuracy model is wanted. Wenqi Zhao, etc.'s project *Handwritten Mathematical Expression Recognition with Bidirectionally Trained Transformer* [8] in 2021 shows that after 8 months of development and training, their model achieved scores of 57.91%, 54.49%, and 56.88% on three different datasets. This apparently does not meet the high accuracy requirement. Although improvements can be made upon existing projects, it is still impossible to train a model with sufficient accuracy within 2 months, and this approach is discarded as a consequence.

As a result, the project finally uses an existing service for mathematical equation recognition: *Mathpix*.

Mathpix is a company that focuses on mathematical equation recognition. It provides OCR (Optical Charactor Recognition) APIs for developers so that they can embed this

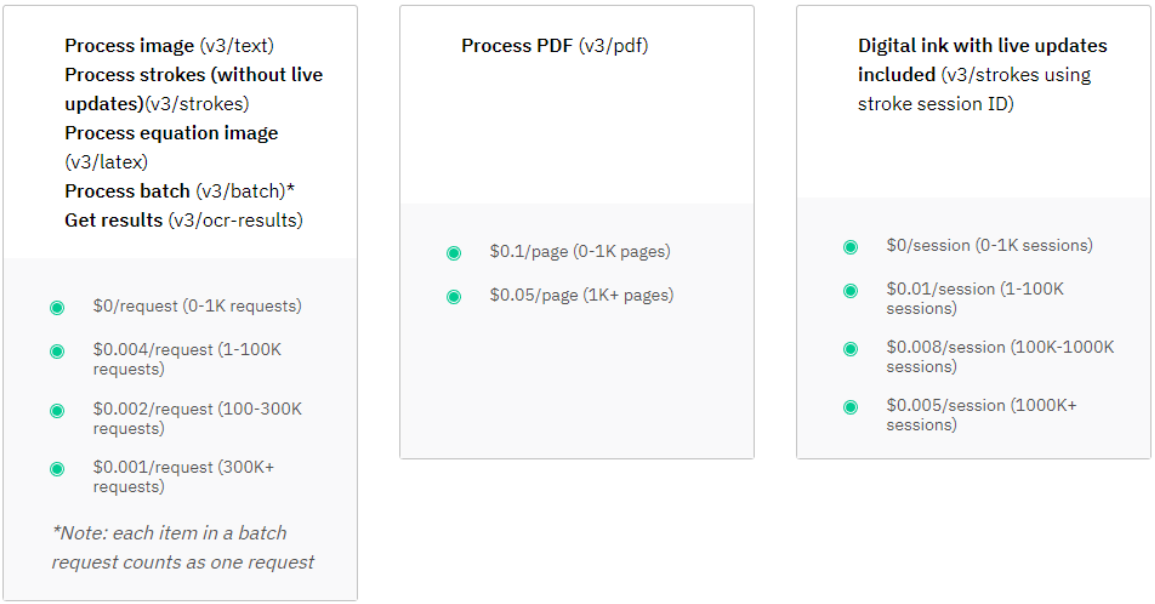


Figure 2.3: Mathpix OCR API Pricing [9]

feature in their project. Figure 2.3 shows the pricing of the service.

Mathpix receives image data and recognizes the content in the image. If one or multiple equations are found, it can return the result in several styles like Latex, MathML and Asciimath etc. There are also options such as `confidence`, `data type` to assist the user for further process on the reponse data. In addition, it provides complete error messages which is helpful for development.

Although there are other companies providing similar services such as *MyScript*, *Mathpix* is advantageous at its *App Token* and *digital ink* method.

To request recognition, *Mathpix* requires the *API Key* to verify if the access is authorised. Therefore, the *API Key* must not be included in the client end, or they can easily be stolen by hackers by inspecting the app binary. This means that *API Key* and corresponding functions should be implemented in backend, and every time a recognition request is made by client end, it is passed to the backend. The backend then makes the request to *Mathpix* server and returns the response to the client end. This could waste time transmitting data, and bring much load to the backend server when multiple client ends make requests at the same time.

To solve the problem, *Mathpix* provides an *App Token* function, which creates a short live client side token so that client apps can make request directly to *Mathpix* server. In this approach, the clients no longer need to proxy their requests through the backend, but only need to request an *App Token* from the backend. It largely reduces the time on data transmition and load of the backend server.

Figure 2.4 illustrates the procedures of two methods mentioned above.

While other services require the image to be uploaded in their API call, *Mathpix*'s

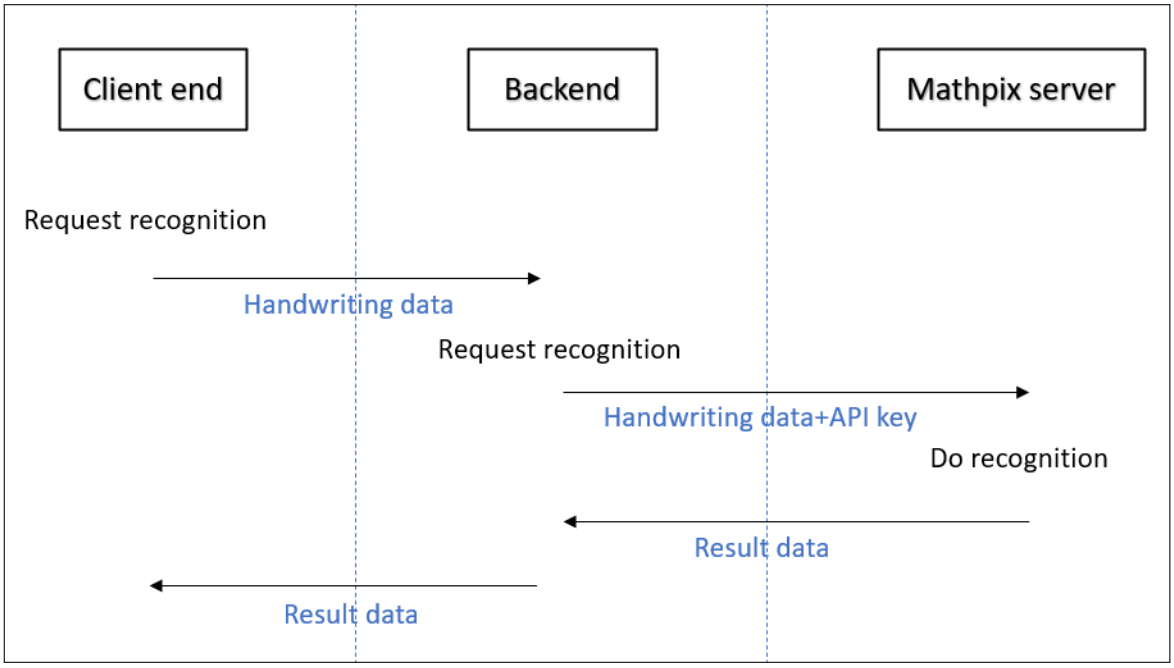
digital ink method allows users to upload stroke data, i.e. coordinates of points on strokes, in their API call in order to reduce network load and response time. Moreover, a live processing session for 5 minutes can be created in *digital ink* method. In a session, users can invoke APIs as many times as they but only counts once in pricing, which is much more cost-effective.

In conclusion, *Mathpix* is the best choice regarding accuracy, cost and time and development.

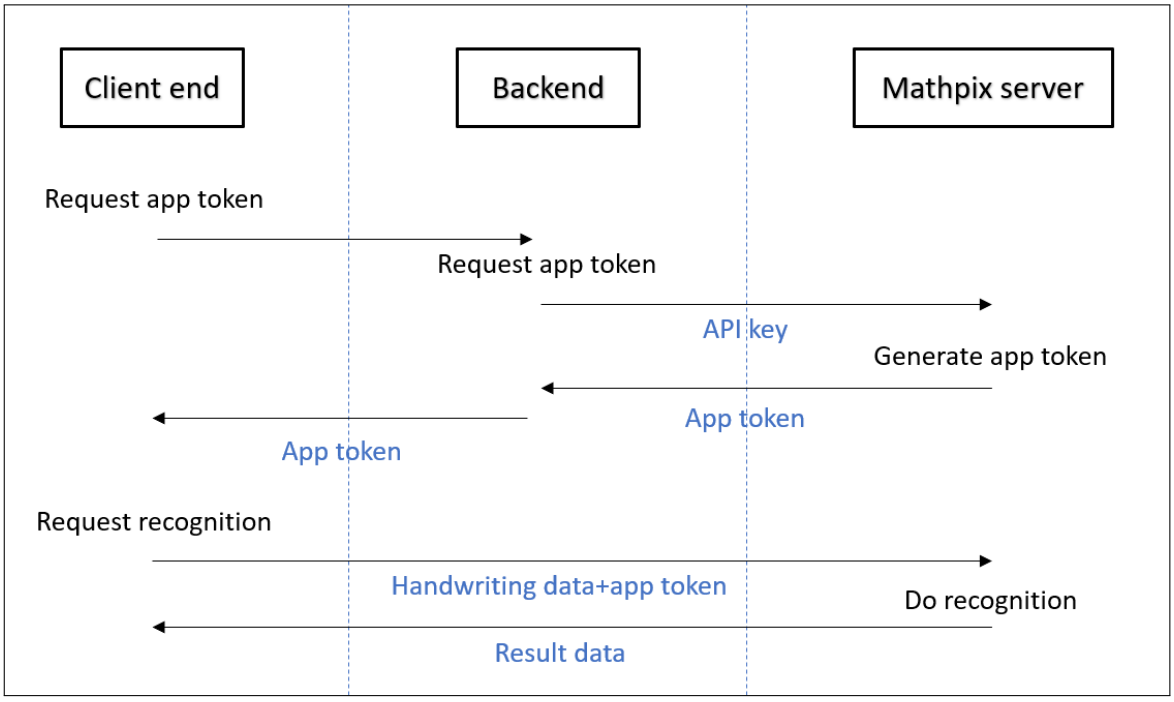
Another thing to be noted is the equation rendering. *Mathpix* can return a latex styled string of the result, but the string cannot be displayed as an equation in web page. Therefore, a rendering tool is needed to parse the latex string and display the equation in the web page.

This project uses `Katex`, which is the same tool used in Lambda Feedback. It can render a latex styled equation in one line of code [10]. Details will be explained in Chapter 3.1.

This chapter records the background research on React framework, handwriting canvas to use and method of equation recognition, and givs reasons why `SVG` element and *Mathpix* are used in the project. In Chapter 3, the implementation of the Vanilla JS application, conversion to React component and integration with Lambda Feedback is explained.



(a) Use API Key for Recognition



(b) Use App Token for Recognition

Figure 2.4: Comparison of Using API Key and App Token for Recognition

Chapter 3

Implementation

Following the three steps mentioned in Chapter 2, this chapter is divided into three sections as well, each explaining the implementation of one step.

It should be noted that before installing the component into Lambda Feedback, the frontend part and backend part are not seperated for the convenience of development and test. This means that the Vanilla JS web application and React component contain the *API Key* for requesting *App Token*.

3.1 Vanilla JS Web Application

The first stage of the project is to develop a web application in Vanilla JS. This is to figure out the structure of the application and the working process of the programme, and focus on the JavaScript code but not React features.

The application requests an *App Token* when being launched. It has a handwriting area for the user to draw, and when the user finishes a stroke, i.e. the pointer goes up, it should automatically send the stroke data to *Mathpix* server. After receiving response from the serverm it should display the equation recognised in text format. In addition, there should be a toolbar to edit the content.

This application consisits of three parts:

- Structure of the web pate (HTML)
- Functions (JavaScript)
- Styles of the web page (css file)

In this application, styles are not concerned as they will be specified in another approach in React.

The structure of the web page is designed as Figure 3.1

The HTML code can thus be written easily based on the structure. It should be noted that this application uses `canvas` element because `SVG` element was not discovered

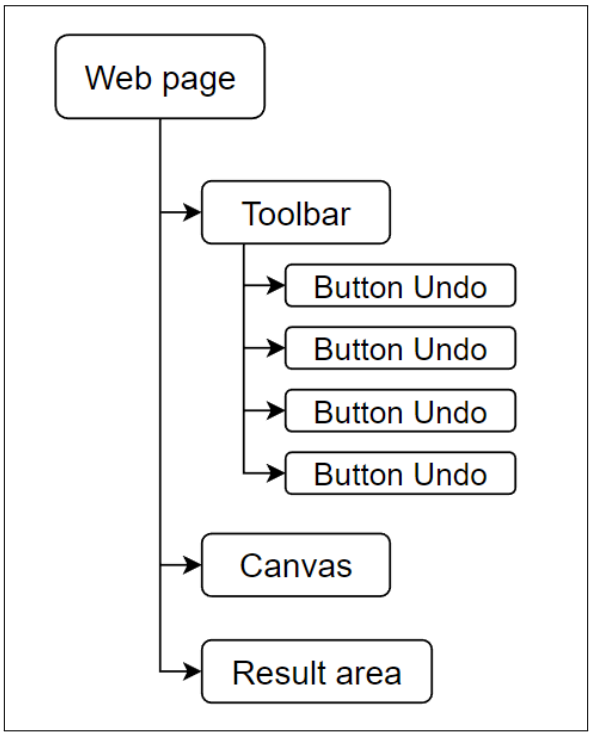


Figure 3.1: Structure of Vanilla JS Web Page

yet when building this prototype. The next step is to add functions to the application.

Different from developing in other languages such as Python and C++, test functions are normally not available for frontend development with JavaScript/TypeScript. Tests are usually carried in the console of a browser. Therefore, the application was developed progressively. Functions were divided into several parts and each part was added after previous part passed tests:

1. Users can draw on the canvas, and stroke data is recorded
2. The application can send stroke data to *Mathpix* and display the result
3. Undo, redo, delete and remove functions are available

Following sections will explain how each function was implemented.

3.1.1 Drawing on the canvas

Before start, the `canvas` should be initialised and referred. This is done in a `setupCanvas` function, which sets up variables such as size, stroke color and stroke width etc.

To let the users draw on the canvas and display the strokes they draw, function `addEventListener` was used. This function can detect specified HTML event (e.g. `onClick`, `onMouseDown` etc.) on an HTML element, and bind the event to a custom function so that once the event happens on the element, the corresponding function is called. The function is used in `setupCanvas` as well.

In `canvas` element, three events are being detected:

- `onMouseDown`: the mouse button is pressed down
- `onMouseMove`: the mouse is moving
- `onMouseUp`: the mouse is released

In this application, three events are connected to function `handleMouseDown`, `handleMouseMove` and `handleMouseUp` respectively. Operations in each function are shown in flowchart in Figure 3.2.

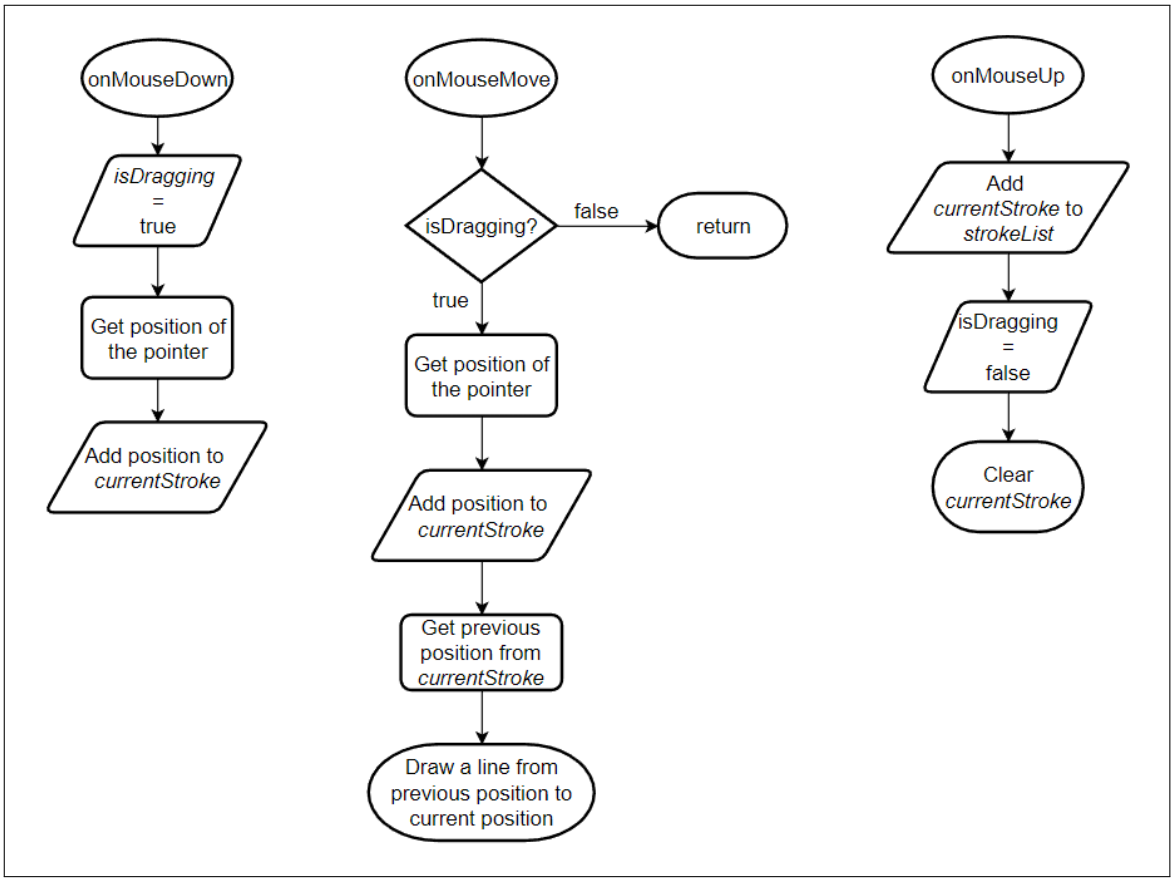


Figure 3.2: Event Flowcharts for Drawing on Canvas and Data Recording

Where:

- `isDragging` is a state flag checking if the mouse is down
- `currentStroke` is a list of points representing the current stroke the user is drawing
- `strokeList` is a list of strokes representing all content the user has drawn, i.e. the stroke data

The function for `onMouseDown` sets a start point for the canvas to draw, and marks the pointer state down. Event `onMouseMove` is checked every browser's event loop.

When it is detected, and mouse is down, the position of the pointer is recorded to `currentStroke` and the `canvas` draws a line from the previous point obtained from `currentStroke` and the current position. This is done by methods `moveto(previousPoint)` and `lineto(currentPoint)` of element `canvas`.

The user can now draw paths on the canvas, and stroke data is recorded in this process. The next section will explain how stroke data is sent to *Mathpix* and how result is fetched and displayed.

3.1.2 Send stroke data and display result

In this application, there are two requests to *Mathpix*: *App Token* request and recognition request. Both requests are made by `cURL`. In JS/TS, this can be done by `fetch` function. It can send a request through HTTP pipeline and fetch response asynchronously across the network [11]. Using `fetch`, it is easy to make requests for *App Token* and recognition.

App Token should be obtained when the web page launches. Therefore, the request is added in `setupCanvas` function. According to *Mathpix* document [12], the request header should include the *API Key* for authorisation and content type of `application/json` for data transmission. The data to transmit is to ask the server to return session ID so that a live session is created. The *Mathpix* server will then return an object containing *APP Token* and *session id*. The object is then parsed and data is stored.

As to recognition request, to realise live update, the content should be recognised every time the user finishes a stroke, which corresponds to the event `onMouseUp`. Therefore, the request is added to function `handleMouseUp`.

Instead of *API Key*, recognition request uses *APP Token*. This is for seperation of backend and frontend in further development. The data to submit is the stroke data (`strokeList`) and *session id*. Moreover, options for the response can be specified in the data as well. In this application, `latex_styled` is included in the option so that a latex styled string is included in the result. *Mathpix* will process the data and return the result as an object containing required data format. The object is then parsed and the `latex_styled` result is stored.

After the result is obtained, it has to be displayed. As introduced in Chapter 2.3, `Katex` can render a latex styled string to equation in a specified element. In this application, it is rendered in the element `result area`. This is done after the result is obtained, which is in function `handleMouseUp` as well.

Now, the flowchart of function `handleMouseUp` becomes Figure 3.3

Another thing to notice is that, the live update session has a time limit of 300 seconds. To ensure the user does not need to refresh the page when the session is expired, the application sends another *APP Token* requests when it receives an

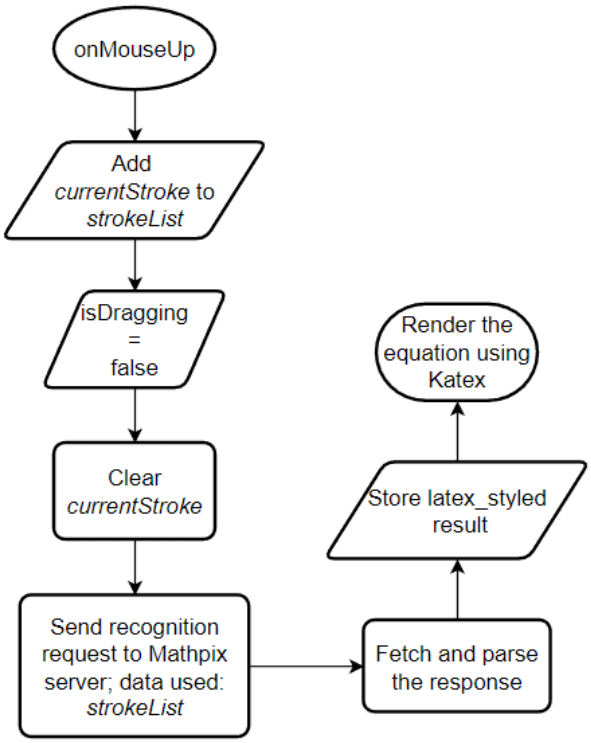


Figure 3.3: Flowchart of `handleMouseUp` for Recognition

`unauthorized` error in recognition request. After the *APP Token* is updated, the recognition request is resent.

This section records the implementation of recognition and render. The next section will explain how undo, redo, delete and remove functions are implemented.

3.1.3 Toolbar functions

Four edit functions, `undo`, `redo`, `delete` and `remove` are implemented in the toolbar. As `remove` function affects the development of `undo` and `redo` functions, it is implemented firstly.

The `remove` function is bound to the remove button. When the button is clicked, the application enters the 'remove' mode: when the pointer presses on or is dragged over a stroke, the stroke is removed. This is done by traversing all points in all strokes, and compare them to the position where the pointer is down. If horizontal and vertical difference is below a specific value, typically `3*stroke width`, the stroke is moved from `strokeList` to `undoStrokeList`, the canvas redraws all strokes and stroke data is submitted for recognition. This function is named `checkCollision`, and the flowchart is shown in Figure 3.4:

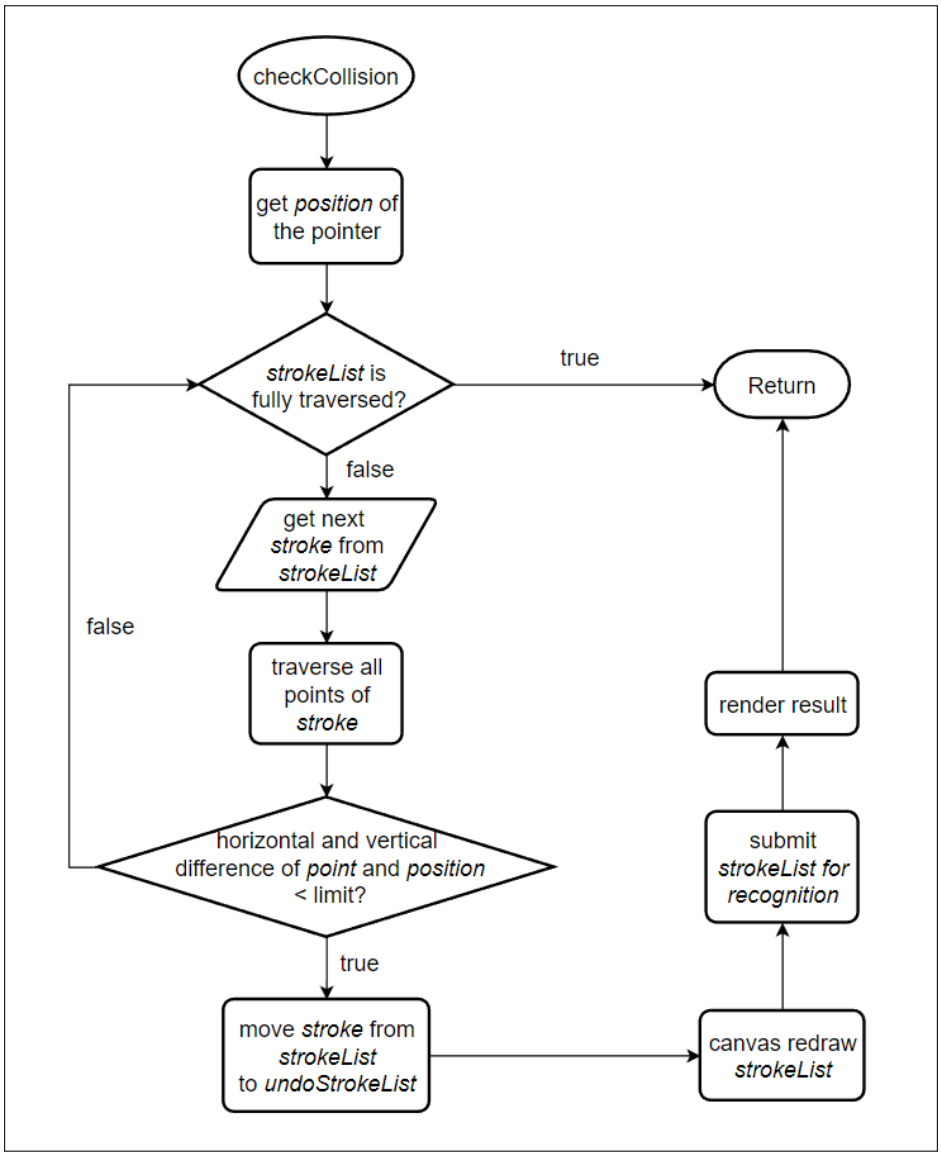


Figure 3.4: Flowchart of Function `checkCollision`

It should be noted that when 'remove' mode is added to the application, a state flag `isDrawing` is added to check if it is in 'remove' mode. Every time `handleMouseDown` or `handleMouseMove` is invoked, `isDrawing` is checked to determine following operations. The flowchart of `handleMouseDown` and `handleMouseMove` thus becomes Figure 3.5:

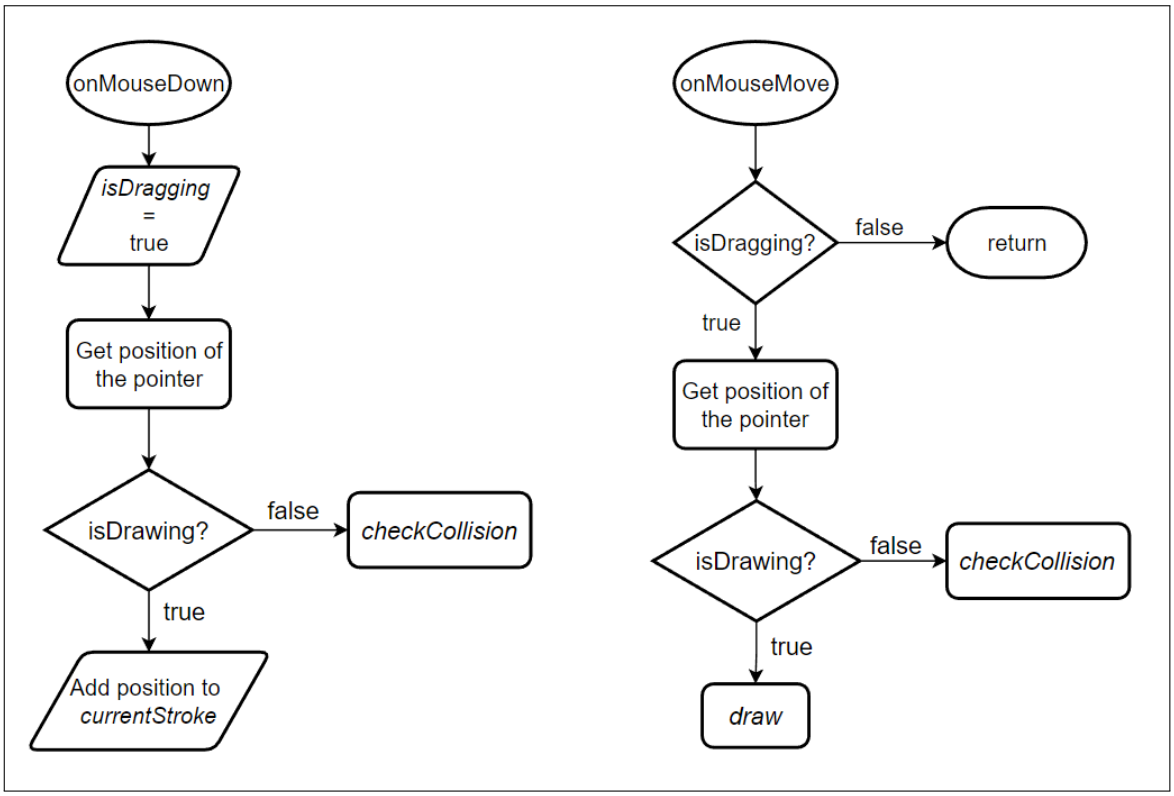


Figure 3.5: Flowchart of `handleMouseDown` and `handleMouseMove` with 'remove' mode

Where `draw` is illustrated in Figure 3.2

Function `delete` simply clears the canvas and the stroke data, i.e. `strokeList`. Note that the result area is cleared as well.

Function `undo` depends on the operation done to determine whether to remove or restore a stroke. Therefore, a variable named `operations` is needed to record all operations. When a stroke is added to `strokeList`, string 'draw' is added to `operations`; when a stroke is removed from `strokeList`, string 'remove' is added to `operations`. When `undo` is invoked, it checks the last operation in `operations` and moves it to `undoOperations`. If the last operation is 'add', the last stroke in `strokeList` is moved to `undoStrokeList`; else, the last stroke in `undoStrokeList` is moved to `strokeList`.

Two assist functions, `removeLastStroke` and `restoreLastStroke`, are used. Flowcharts are shown in Figure 3.6.

Function `redo` has opposite logic of `undo`, and can be easily implemented from it. Flowchart of `undo` and `redo` is shown in Figure 3.7

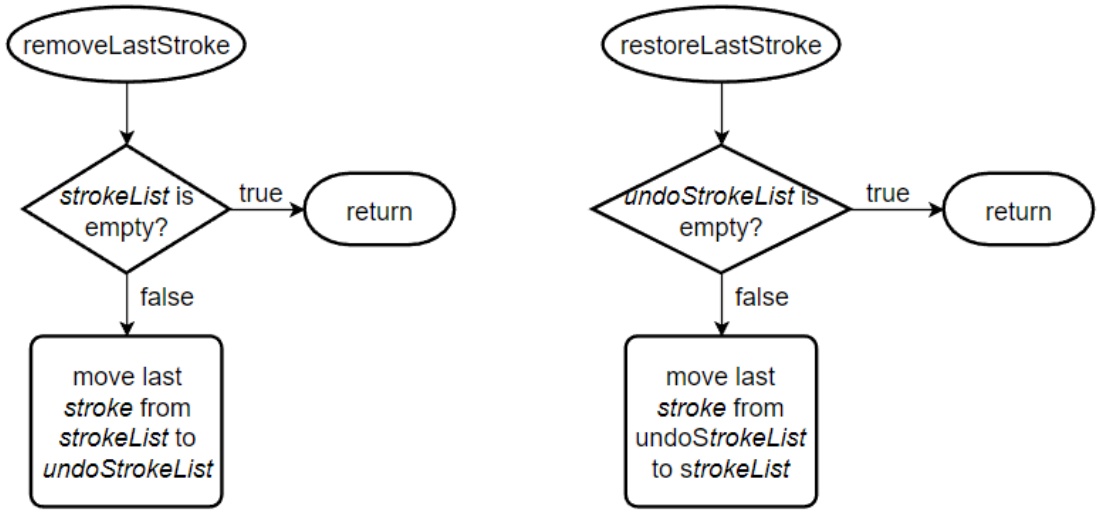


Figure 3.6: Flowchart of `removeLastStroke` and `restoreLastStroke`

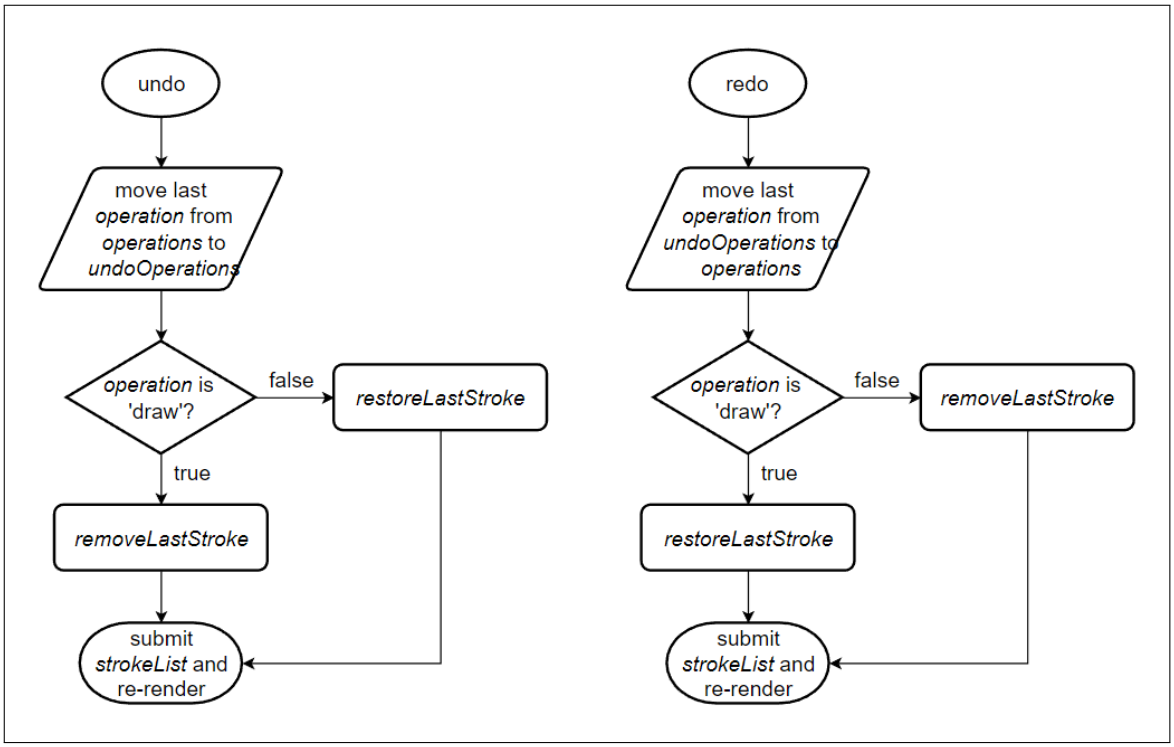
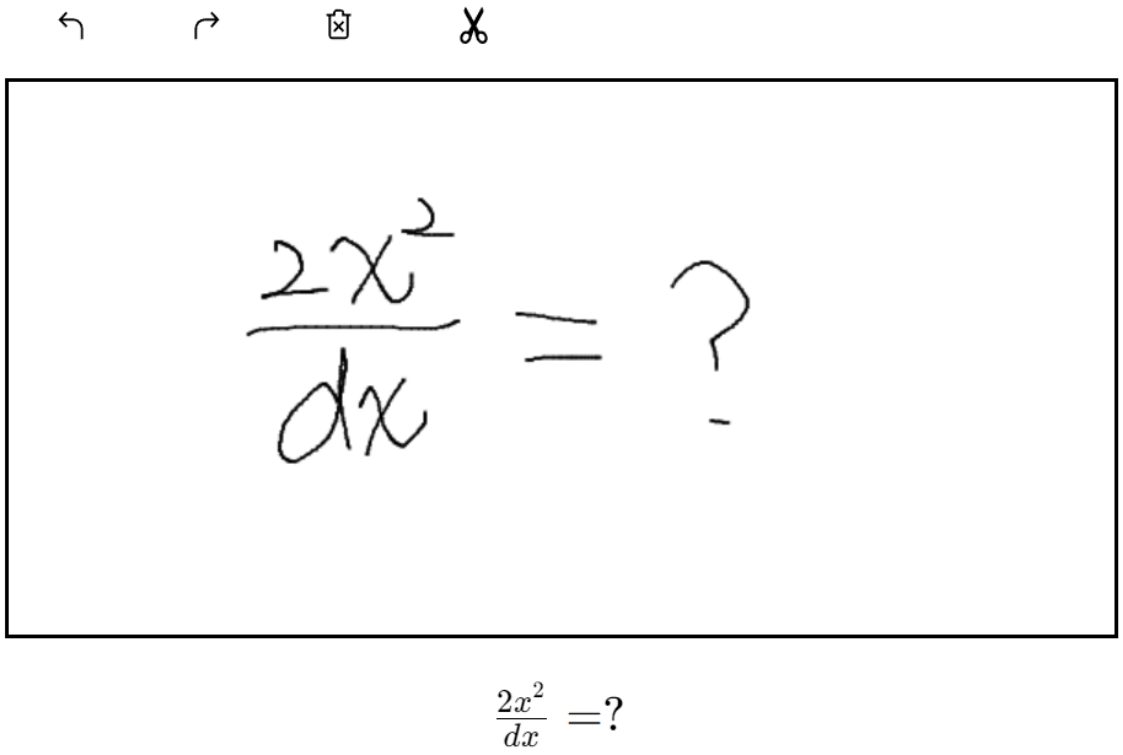


Figure 3.7: Flowchart of `undo` and `redo`

The implementation of the Vanilla JS web application is completed, and a test is carried out to validate that all functions are working. Figure 3.8 is a screenshot of the application.

It can be seen from Figure 3.8 that though the application has realised all functions,



$$\frac{2x^2}{dx} = ?$$

Figure 3.8: Screenshot of Vanilla JS Application

the content in the writing area (`canvas`) is blurry. This will be fixed in the next step.

After examining the functionality of the application, the project moves to the next stage - conversion to React component.

3.2 Convert to React component

Now that the Vanilla JS application has implemented the required functions, this chapter will then discuss the conversion to React component to adapt to Lambda Feedback.

To convert from Vanilla JS to React component, the programme is restructured into sub-components according to functions:

- `Canvas` : the area where users draw.
- `FullScreenDialog` : a container to wrap the component so that it can be installed in Lambda Feedback.
- `Navbar` : the toolbar to place function buttons.
- `Path` : display one stroke

Besides, a type `Point` is created to represent the position of a point.

All sub-components are wrapped in a parent component `HandwritingInput`, where most functions are deployed. It receives events from sub-components, executing corresponding functions to the events and distribute the changes to sub-components. Variables in Vanilla JS application are included in the parent component as React state variables. Figure shows the structure of the component.

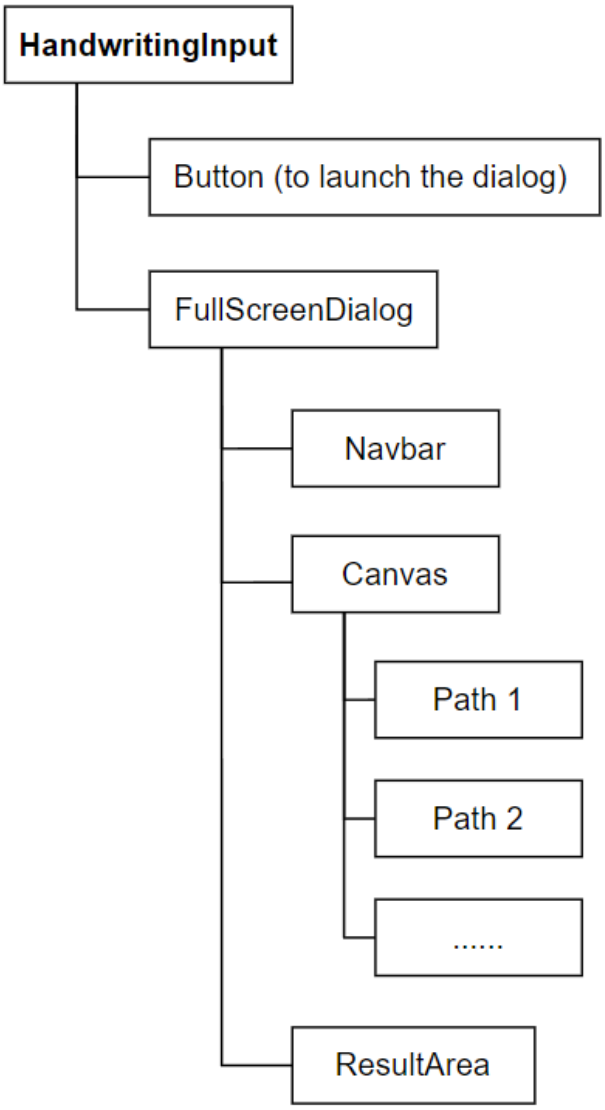


Figure 3.9: Structure of the React Component

It should be noted that for the handwriting area of the React component, the `canvas` element is replaced with `SVG` element. This thus leads to a new sub-component `Path`. This is the only change in the conversion in terms of implementation of functionalities.

Similar to Vanilla JS, the React component is built in the order of functions.

Handwriting Area

The logic of adding strokes to `strokeList` remains the same, but instead of using `addEventListner`, events and functions are bound via attributes.

Component `HandwritingInput` has three functions, `handlePointerDown`, `handlePointerMove` and `handlePointerUp`, which have same functionalities as handling functions in Vanilla JS app except for getting the position of the pointer, as this happens in `Canvas`. These functions are passed to component `Canvas` as function typed attributes `onPointerDown`, `onPointerMove` and `onPointerUp` respectively. This allows `Canvas` to let `HandwritingInput` execute handling functions by calling the 'on' functions inside `Canvas` component.

In `Canvas`, three 'handle' functions are created, which gets the position of the pointer, and invoke the 'on' functions with the position as the parameter. Now, calling a 'handle' function inside `Canvas` does all functionalities same in Vanilla JS app.

To call 'handle' functions in response to a pointer event, they have to be bound to the `SVG` element. The `SVG` element is created in the `return` command. It also has `onPointerDown`, `onPointerMove` and `onPointerUp` attributes, which can be bound to a function so that everytime the 'on' attribute triggers (pointer down/move/up), the corresponding function is called. 'Handle' functions in `Canvas` are then passed to attributes of the `SVG` element created inside. Thus, all functions that should react to an event are chained.

Figure 3.10 shows the structure of the handwriting area.

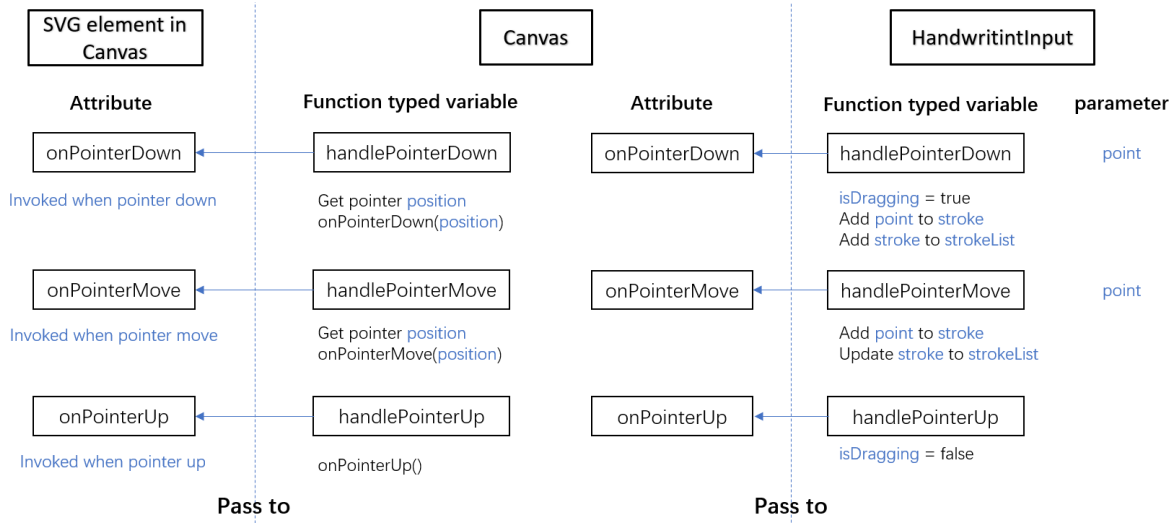


Figure 3.10: Structure of Handwriting Area in React Component

It is worth noting that handlers in `HandwritingInput` does not draw strokes like Vanilla JS app. This is because strokes in this component are displayed by `Path`

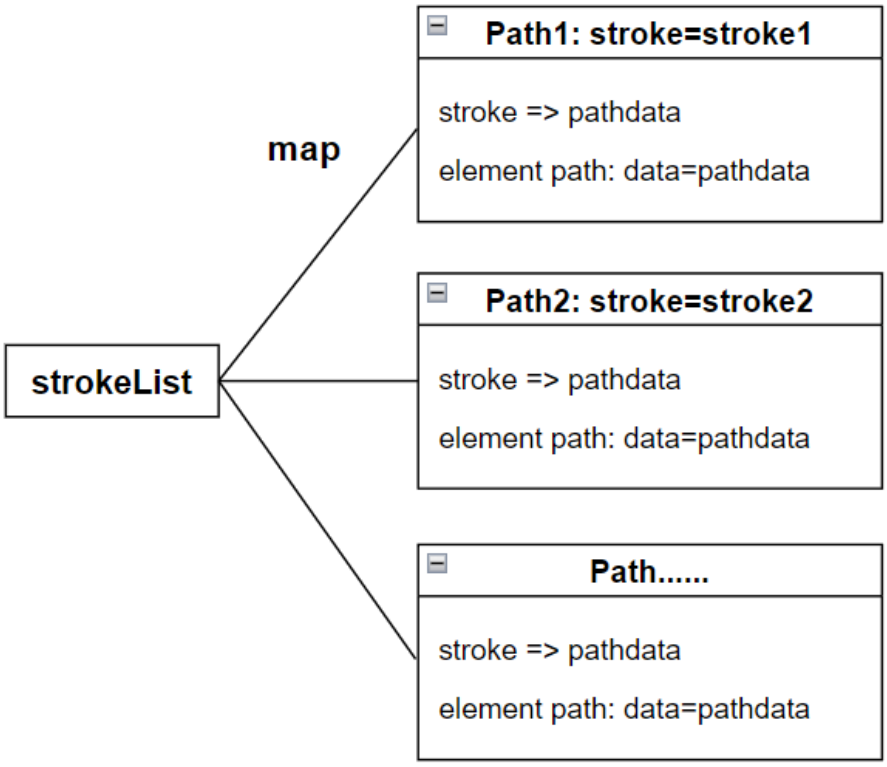


Figure 3.11: Displaying Strokes

components inside `SVG` element, and when `strokeList` is updated, all `Path`s update automatically. This is done by mapping `strokeList` to `Path`s, each `Path` component is distributed a stroke from `strokeList` with a key equal to the index of the stroke. Therefore, any change made to `strokeList` will make React rerender the `Path`s.

`Path` component has an attribute named `stroke`, which receives an array of type `Point`. Inside `Path`, `stroke` is parsed and a string of path data is generated. Path data is then attached to the attribute of element `path` so that the stroke is drawn.

Figure 3.11 illustrates how `Path`s are generated and strokes are displayed.

So far, the handwriting area has been converted to React component. The next step is to implement the equation recognition function.

Equation Recognition

Same as Vanilla JS app, requests are made by `fetch` function. However, stroke data is stored differently from Vanilla JS app in `strokeList`. In previous stage, stroke data is stored in the method required by *Mathpix*, an example of one stroke `strokeList` is:

```
{x:  [x1, x2, x3...], y:[y1, y2, y3...]}
```

However, to distribute strokes to `Path`, stroke data must be stroed like:

`[Point1, Point2, Point3...]`

Hence, a function `transformData` is written to transform `strokeList` to `strokeUpload` with the data structure required by *Mathpix*. This function is invoked in `undo`, `redo`, `remove` and `handleMouseUp`.

Another change made is that the request function is no longer called `handleMouseUp` and three button functions. Instead, the React feature `useEffect` on `strokeUpload`: when this variable changes, the request function is called. A benifit of doing so is decreasing the couple between the request function and the component, so that when the function needs to be isolated, only a few code has to be maintained.

As to rendering the result, there is no difference with Vanilla JS app. The component `ResultArea` receives a string of latex styled result as its attribute, and uses Katex to render the string inside a `span` element. The result string `renderString` is parsed from the response of recognition request in `HandwritingInput` component.

Now that recognition part is finished, the toolbar and related functions are to be installed.

Toolbar Implementation

Methods `undo`, `redo`, `delete` and `remove` remain the same as Vanilla JS app. At this step, it is only necessary to consider how to transplant the methods to the component `Navbar`.

The `Navbar` has a MUI component `Appbar`, which aligns its sub-components in a toolbar. Here, four MUI component `Button`s are added, each representing one function.

Methods are implemented in `HandwritingInput` as they need data from the parent component. Therefore, to bind them with corresponding buttons, they are passed to the corresponding attribute, `onClick`, provided by each button. After that, each method can be called by clicking its button.

Functions are all implemented in the React component. However, it is still a web page as it is not wrapped. To work in Lambda Feedback, it must be wrapped so that it can be used as a component but not a web page.

Wrap the Component

The component is wrapped in component `FullScreenDialog`. It contains a MUI component `Dialog`, and sets it to type 'fullScreen'. Using full screen is to provide enough space for the users to draw. A slide animation is added when opening the dialog so that it looks more fluent.

The `FullScreenDialog` component wraps `Navbar`, `Canvas` and `ResultArea`, and is under `HandwritingInput`. A button is created to open the dialog.

The implementation of the React component is completed. A test was carried out to check if the component works correctly. Figure 3.12 shows that only a button displays when opening the React project - the component is wrapped.

Figure 3.12: React Component Web Page

Figure 3.13 is a screenshot when the dialog is opened. It can be found that the strokes in the canvas is not blurry due to the use of `SVG` element.

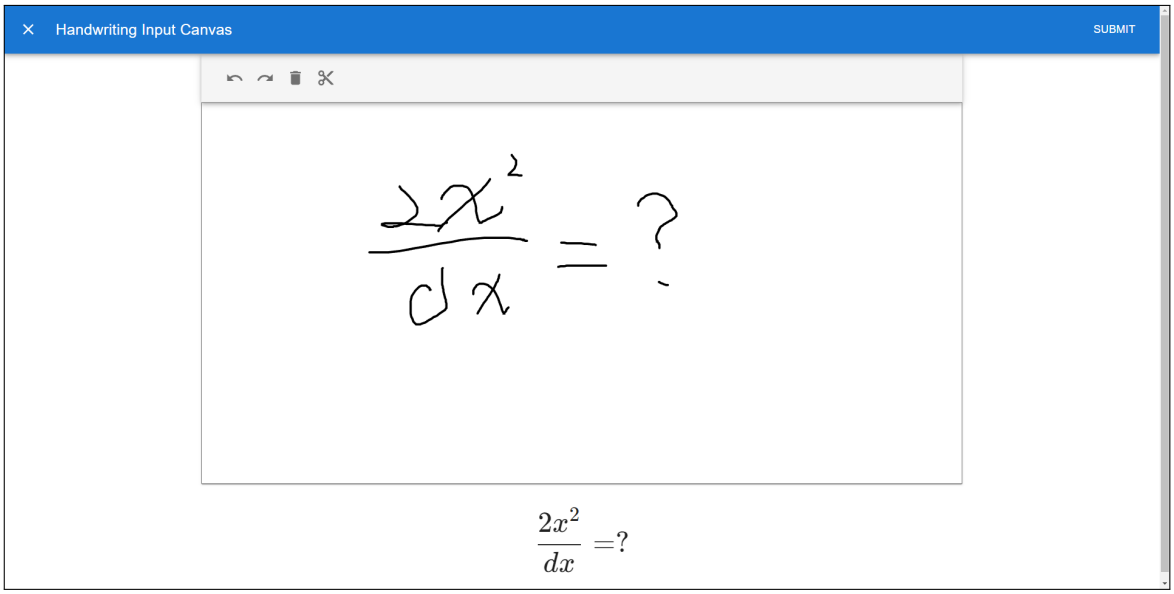


Figure 3.13: React Component

The Vanilla JS application is converted to a React component. However, it is still in JavaScript, whereas Lambda Feedback is developed in TypeScript. Therefore, the whole programme was refactored to TypeScript.

The process is simple and took only two hours. After this, it should be integrated into Lambda Feedback.

3.3 Integration to Lambda Feedback

As mentioned in Chapter 2.1, the handwriting component is added as an extension of type `ExpressionInput` in component `ResponseArea`.

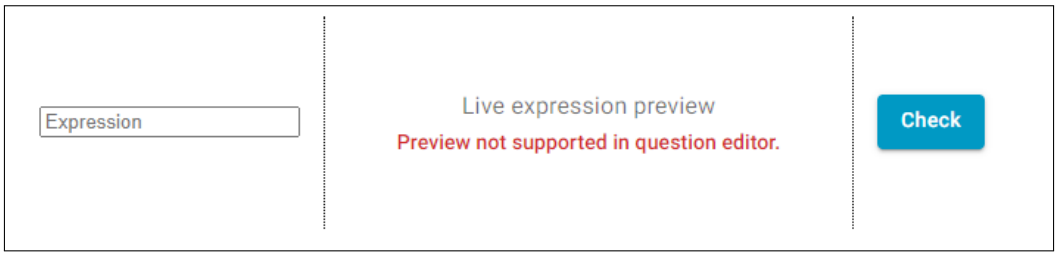


Figure 3.14: `ExpressionInput` before Integration

`ResponseArea` is a component for the user to input their answer and check if it is correct. `ExpressionInput` is a component of `ResponseArea` that has a textbox for the user to input math expression and a preview box for previewing the expression. Figure 3.14 is a screenshot of the component `ExpressionInput` before handwriting input is integrated.

To install handwriting input component, a folder named 'HandwritingInput' is created in `ExpressionInput`, and the code is moved in.

In the handwriting input component, all sub-components are controlled internally. Hence, only the parent component `HandwritingInput` needs to be modified to work in `ExpressionInput`.

There is only one variable that `ExpressionInput` needs from `HandwritingInput`, which is the string of latex styled result. The original text input box uses an anonymous function, where a syntax of `target: value` is used to fetch the value in the input box. However, a compile warning is suggesting that this 'target' config is deprecated and will be removed in future version. Therefore, the `HandwritingInput` component did not use the same approach. Instead, several attributes were exported from `HandwritingInput` so that `ExpressionInput` can pass functions that require the latex string to the attributes. The functions are:

- `handleChange`: the function to react to changes in the input. It is passed to attribute `submitResult` of `HandwritingInput`
- `previewResult`: the function that submits the result to the preview section. This is passed to attribute `previewResult` of `HandwritingInput`.
- `setLatexString`: this function sets the state variable `latexString`, which is created to save the latex result string to update the result to the input box, so that when the user want to slightly modify the expression, they can switch back to text input and edit the expression by keyboard. This is passed to attribute `returnResult` of `HandwritingInput`

Now that three functional attributes are added to `HandwritingInput`, it should be clarified where they are used. There is a submit button on the right top of the component. A function `handleSubmit` is passed to its attribute `onClick`, so that when

clicking on the button, the function is called. Inside `handleSubmit`, all the three functions plus a function to close the dialog are invoked. The three functions are given the parameter `latexString` parsed from the result, so that data is transmitted to `ExpressionInput`. It should be noted that `previewResult` has a type of 'function OR undefined', so it is only invoked when it is defined.

Apart from the three functional attributes, there are other attributes needed to be defined by `ExpressionInput`:

- `height`: the height of the writing area. Defaultly 500px
- `width`: the width of the writing area. Defaultly 1000px
- `linewidth`: the width of the strokes. Defaultly 3px
- `open`: boolean flag that controls if the dialog is open or close.

The connection between `HandwritingInput` and `ExpressionInput` is established. To use handwriting input, there should be an approach to open it. The purpose of this stage is to add the handwriting component to `ExpressionInput`, and its original functions should be retained. Therefore, a toggle is used so that the user can switch between handwriting input and keyboard input. MUI component `Switch` is used as the toggle. The toggle controls a state variable `useHandwrite`. When the toggle is on, `useHandwrite` equals false; when off, `usHandwrite` equals true.

The `&&` operator is used to control if a component is rendered. Handwriting input is controlled by:

```
useHandwrite && (<HandwritingInput>)
```

when `useHandwrite` equals true, handwriting input is enabled. Text input is controlled by:

```
!usehandwrite && (<textInput>)
```

when `useHandwrite` equals false, text input is enabled.

Beside the toggle, a button is set to open the handwriting input dialog. The button also controls a state variable `dialogOpen`, which is passed to attribute `open` to open or close the handwriting input interface.

After this, the handwriting input tool is installed in Lambda Feedback, and works well. There is then some updates made to the component to imporve user experience.

A grid paper styled background is added to the handwriting area. This is done by `makeStyle` function provided by Lambda Feedback code, which allows creating styles in component. The background is generated by `linear-gradient`, which is a CSS function creating an image consisting of progressive transition [13].

Another update is re-aligning sub-components in `HandwritingInput`. The handwriting area and the result area were aligned vertically, and the result might require

scrolling down the page to view. They were then re-aligned horizontally so that it can be seen directly.

So far, the handwriting input component has been successfully integrated into Lambda Feedback, and the implementation part is finished. In Chapter 4, a complete test was done on the component in terms of functionalities and fitness to Lambda Feedback. The results are shown and an evaluation is given as well.

Chapter 4

Experimental Results

Chapter 3 records the process of the implemention of this project, and in this chapter, a series of tests is carried out and results along with an evaluation is provided.

4.1 Tests and Results

Appearence of Expression Input

Figure 4.1 is a screenshot of the expression input. Comparing with Figure 3.14, it can be seen that a button and a toggle is added over the text input to open and switch to handwriting input.

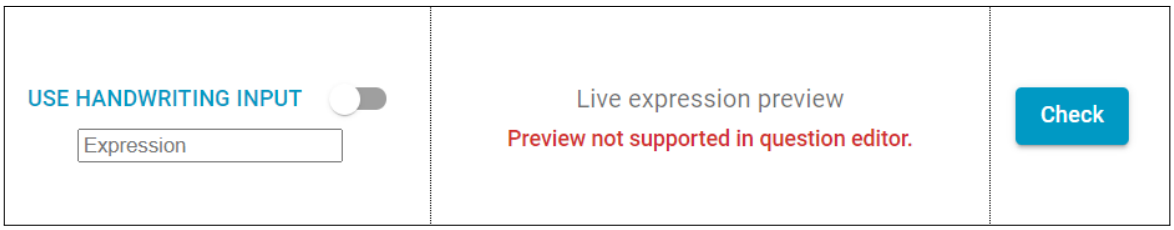


Figure 4.1: Appearence of Expression Input Component

Either clicking the button or turning on the toggle opens the handwriting input dialog shown in Figure 4.2

Handwriting Input Tests

The handwriting input component is tested to verify its functionalities.

Figure 4.3 shows that the component is able to recognise and display the equation written in writing area.

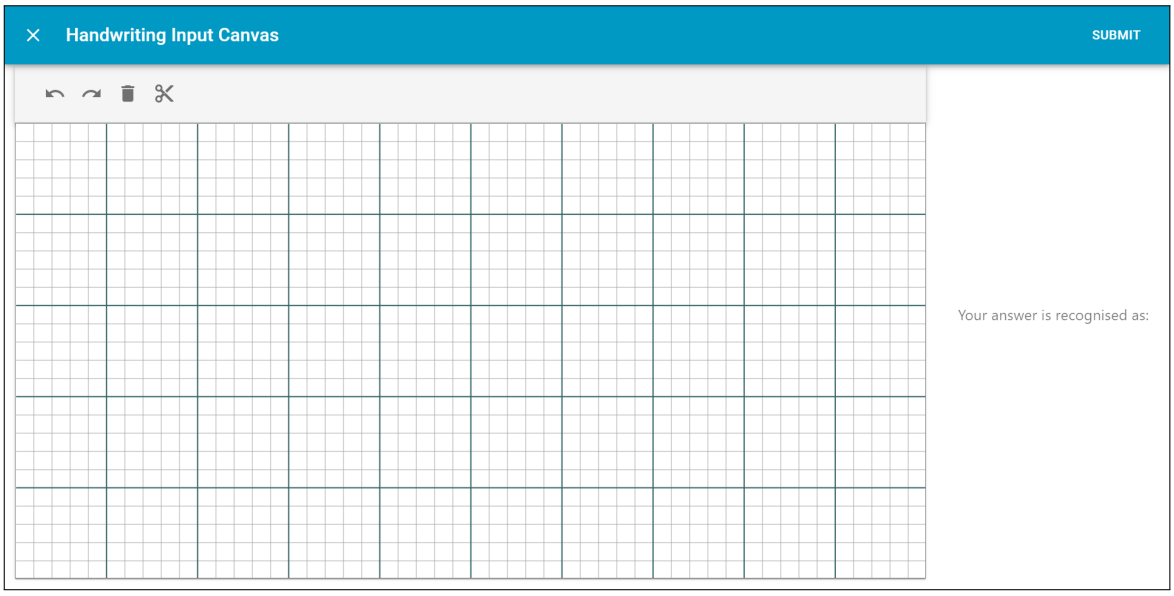


Figure 4.2: Handwriting Input Dialog

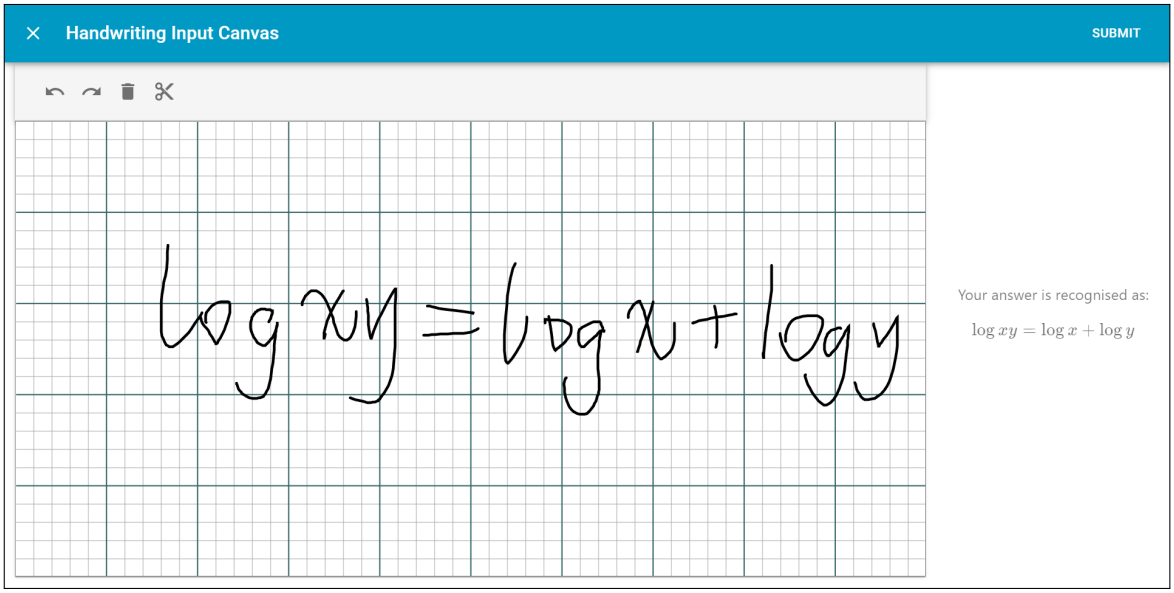


Figure 4.3: Writing Test

Undo, redo, delete and remove functions works as well. Figure 4.4 is an example of the remove function, where the '+' operator is removed, and the result updates.

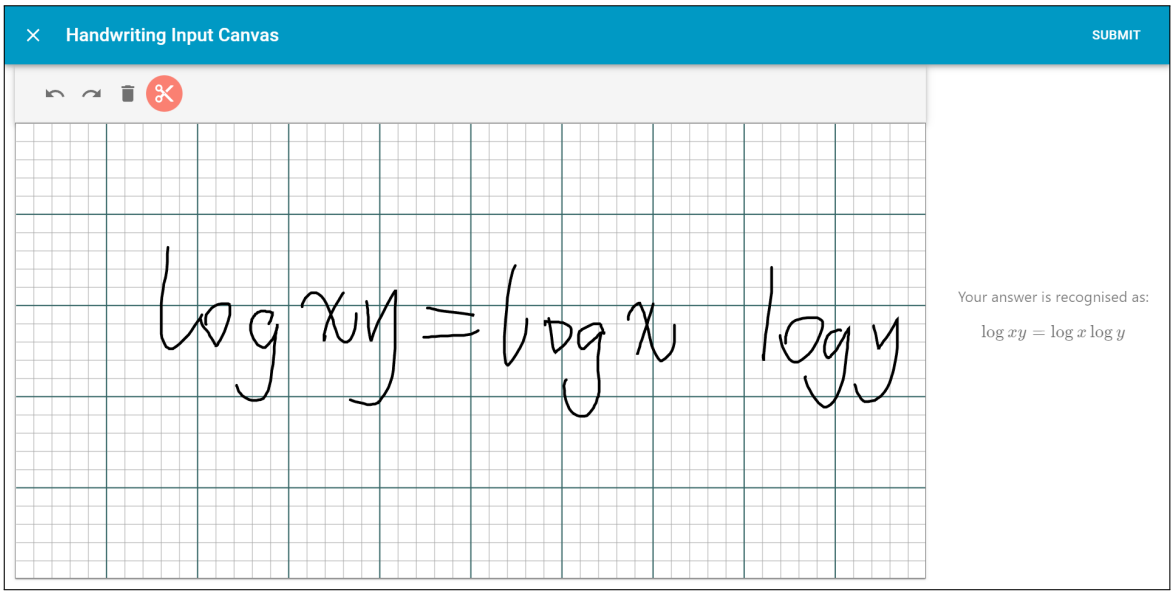


Figure 4.4: Remove Function Test

Integration Tests

After the tests on handwriting input, the result is checked to see if it is submitted, and the component is integrated successfully.

The answer is set to $\log xy = \log x + \log y$. Figure 4.5 shows that the result is evaluated correct, and the preview session works as well.

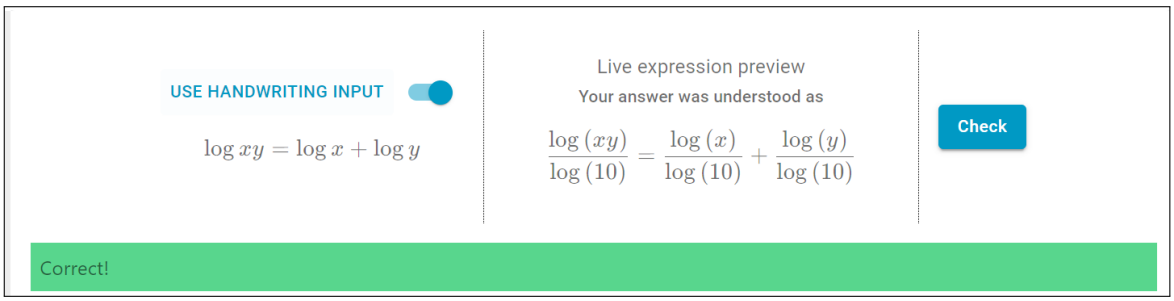


Figure 4.5: Check a Correct Answer

An incorrect answer is tested as well, and the result is shown in Figure 4.6.

When switching back to text input, the equation automatically exists in the box in latex styled text. This is shown in Figure 4.7.

As mentioned in Chapter 3.1, the session has a time limit of 300 seconds. To check if the session is automatically extended, a test is carried out. Handwriting input was done 6 minutes after the dialog was opened, and the component is still able to recognise and display the equation with little lag.

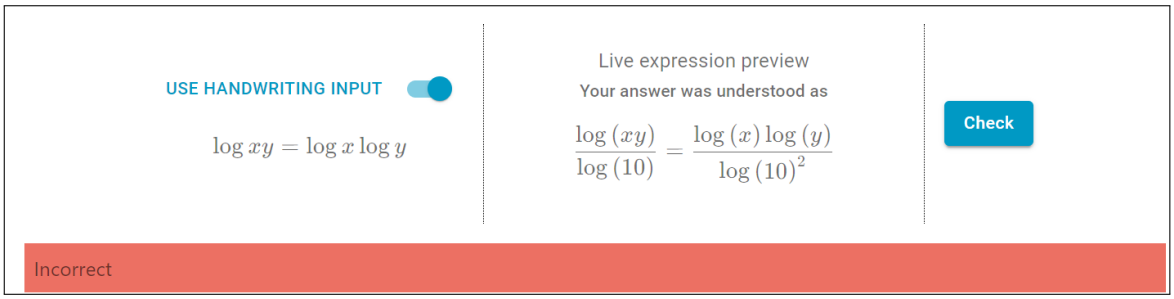


Figure 4.6: Check an Incorrect Answer

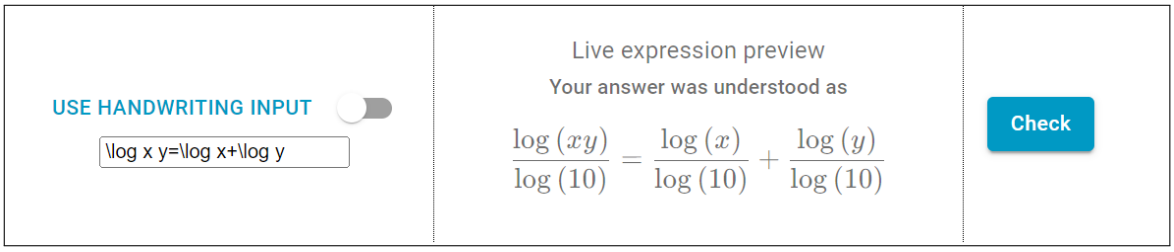


Figure 4.7: Switch Back To Text Input

One concern of the component is about the latex string obtained from *Mathpix*. Although *Mathpix* service focuses on mathematical equation recognition, the latex string it returns is not guaranteed to be a latex math string. Instead, it has several types, including `text`, `math` and `tabel` etc. Although non-math types can be filtered when obtaining the result and thus give a gentle warning to the user that the handwriting is not recognised as an equation, it sometimes happens that some simple equations, such as $2x + 1$, are returned with the type of `text` as it does not contain any special mathematical operator. These latex strings can still be evaluated by Lambda Feedback, but they are not `math` type. As other types are very different to an equation, which would not affect the result, the response obtained from *Mathpix* is not filtered.

To ensure this does not worsen user experience, a test was carried out. Many equations, from simple to complicated, were input and checked if they display in the live expression preview, as they are resolved in this session. It turned out that most equations can be successfully resolved by the preview session. The exceptions are equations that contain the lambda symbol (λ) - the result is shown in Figure 4.8. Other symbols including greek letters were tested and resolved normally, whereas they are not all symbols that might appear in an equation so there is no guarantee this phenomenon would not occur on other symbols. This is considered as a problem related to the evaluation function, `latexEqual`, and needs further imporvement.

Consequently, in most cases, type of the latex string has minor impact to the user experience. The lambda symbol problem is probably not related to the types.

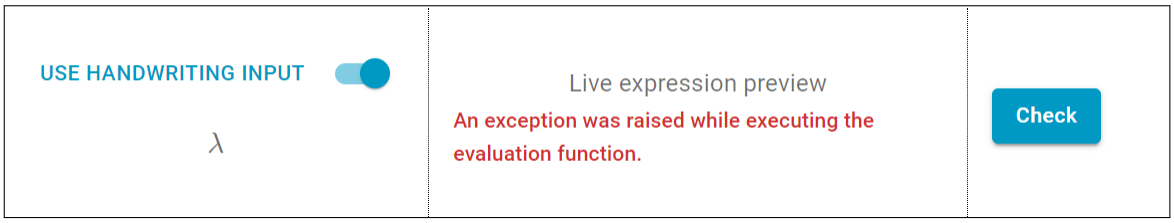


Figure 4.8: Test Lambda Symbol

4.2 Evaluation

In Chapter 4.1, a series of tests were carried out. The results show that the component is able to provide a complete handwriting input approach for the user to input mathematical expressions. The recognition is accurate, live update is available and response is quick. The answer can be passed to the parent component and checked by Lambda Feedback's evaluation function. It can also be displayed in the text input box in the format of latex string. Although there are many possible types of latex string returned by *Mathpix*, it is considered having insignificant impact to user experience. However, the problem of the lambda symbol still needs to be solved.

Despite the functionalities being generally good in Lambda Feedback, the handwriting input component has not seperated its backend and frontend, i.e. the *API Key* for *Mathpix* is still in frontend code. It means that the update cannot be published and put into practical use yet. This is because of limited time of the project. In spite of this, the backend code is simple: make a request for *APP Key* to *Mathpix* surver. Therefore, it should be quick for a developer who is familiar with the backend of Lambda Feedback to implement this.

Overall, the handwriting input component works well in terms of functionality and integration with Lambda Feedback. Regarding the code writing, only one file of programme in Lambda Feedback frontend is modified, and one folder is added in the `ExpressionInput` folder, which makes maintaince and debugging easier. However, the backend still needs to be seperated.

Chapter 5

Conclusion

In this project, a handwriting input component for mathematical expression is developed and added to the online learning system Lambda Feedback.

Firstly, a web application is developed in Vanilla JavaScript to learn the problem and figure out solutions. Then, the application is converted to a React component in TypeScript to fit Lambda Feedback. Finally, it is integrated into the system, and several updates were done afterwards.

A series of tests were conducted to verify the functionality and robustness of the component in Lambda Feedback. The results indicate that its functionalities are generally good, but has a problem with the lambda symbol. Despite this, the project did not manage to seperate its frontend and backend, and the update cannot be published yet.

Future Improvements

There are several future improvements planned:

1. Seperate the backend. This is necessary for the project. The token request function should be implemented into the backend and the function in client end should be requesting the backend for the token.
2. Add new method for removing stroke. Users might find it unconvenient to switch between 'draw' mode and 'remove' mode. A new method that drawing a circle would remove strokes inside, or black out a symbol to remove it is planned to be developed.
3. Error handling. Although current programme has some error handlings, they do not cover all errors such as an error object is returned from *Mathpix*. This rarely happens but still needs to be done.

Bibliography

[1] Ali W. Online and remote learning in higher education institutes: A necessity in light of COVID-19 pandemic. Higher education studies. 2020;10(3):16-25. pages 1

[2] Rawat P, Mahajan A. ReactJS: A Modern Web Development Framework. International Journal of Innovative Science and Research Technology. 2020;5(11):16. pages 4, 5

[3] Bugl D. Learn React Hooks: Build and refactor modern React. js applications using Hooks. Packt Publishing Ltd; 2019. pages 6

[4] Canvas API - web apis: MDN;. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API. pages 8

[5] React Component Lifecycle;. Available from: https://reactjs.org/docs/react-component.html#the-component-lifecycle. pages 8

[6] Spiess P. How to build a freehand drawing using react;. Available from: https://pspdfkit.com/blog/2017/how-to-build-free-hand-drawing-using-react/. pages 8

[7] Vinothpandian. Vinothpandian/react-sketch-canvas: Freehand vector drawing component for react using SVG as canvas;. Available from: https://github.com/vinothpandian/react-sketch-canvas. pages 9

[8] Zhao W, Gao L, Yan Z, Peng S, Du L, Zhang Z. Handwritten mathematical expression recognition with bidirectionally trained transformer. In: International Conference on Document Analysis and Recognition. Springer; 2021. p. 570-84. pages 9

[9] Mathpix products pricing;. Available from: https://mathpix.com/pricing. pages 10

[10] API · KATEX;. Available from: https://katex.org/docs/api.html. pages 11

[11] Using the fetch API - web apis: MDN;. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch. pages 16

[12] Mathpix. Mathpix Developer Document;. Available from: https://docs.mathpix.com/. pages 16

[13] Linear-gradient() - CSS: Cascading style sheets: MDN;. Available from: `https://developer.mozilla.org/en-US/docs/Web/CSS/gradient/linear-gradient`. pages 28