Index

Abstract

This project report presents the design and development of a comprehensive paint application built using HTML, CSS, and JavaScript. The primary objective of this project was to create a user-friendly and feature-rich web-based application that allows users to create and manipulate digital artwork with ease.

The paint application offers a range of essential drawing tools, including **brushes**, **lines**, **rectangles**, and **circles**, empowering users to express their creativity through various stroke sizes and colors. Additionally, the application supports multiple layers, enabling users to work on complex compositions while maintaining flexibility and control over individual elements.

The project's implementation leveraged the power of the **Canvas API**, a built-in HTML5 feature that facilitates dynamic rendering of graphics and animations. Through careful planning and efficient coding practices, the application delivers a seamless and responsive user experience, ensuring smooth drawing and real-time visual feedback.

Furthermore, the report delves into the project's planning and development phases, including requirements gathering, system design, and implementation details. It also outlines the testing methodologies employed to ensure the application's functionality, usability, and performance.

The paint application not only serves as a practical tool for digital artists and hobbyists but also demonstrates the capabilities of web technologies in creating interactive and visually appealing applications. The project showcases the integration of various programming concepts and techniques, making it a valuable educational resource for aspiring web developers and designers.

Objective of the Project

Develop a versatile paint application

The primary objective of the project is to create a feature-rich paint application that caters to the needs of both casual users and professional artists. The application aims to provide a wide range of drawing tools, editing features, and customization options to enhance user creativity and productivity.

Intuitive user interface

One of the key objectives is to design an intuitive and user-friendly interface that allows users to navigate seamlessly through the application. Emphasis is placed on accessibility and ease of use, ensuring that users of all skill levels can comfortably utilize the software.

Modular design for scalability

The project aims to adopt a modular design approach, dividing the application into distinct modules or components. This facilitates scalability and maintainability, allowing for easier implementation of new features and enhancements in the future.

System Analysis

Identification of Need

In today's digital age, there is a significant demand for simple yet powerful graphic design tools that cater to a broad range of users, from beginners to professionals. Traditional paint applications often come with a steep learning curve and a cluttered interface, which can be overwhelming for new users. This project aims to address these issues by developing a minimalist paint application that focuses on reducing clutter while providing essential functionality.

Simplicity and Ease of Use

A straightforward, intuitive interface that allows users to start drawing immediately without needing extensive tutorials or prior experience.
Clear and easily accessible tools and options to ensure users can find and use the features they need without confusion.

Essential Tools and Features

Providing fundamental drawing tools such as **brush**, **line**, **rectangle**, and **circle tools**. Ensuring these tools are effective and user-friendly, allowing users to create various types of artwork effortlessly.Allowing users to adjust key settings like **canvas size**, **brush size**, and **color** options. Ensuring that customization options are easy to find and use, enhancing the user experience without adding complexity.

Preliminary Investigation

The preliminary investigation phase is crucial in laying the groundwork for the Paint Application project. This phase involves gathering relevant information to understand the requirements and feasibility of the project. The preliminary investigation helps in identifying potential challenges and opportunities, ensuring that the project is viable and aligns with user needs and technological capabilities.

Feasibility Study

The feasibility study is a critical step in project planning, conducted to determine the viability of the Paint Application project. It involves assessing various factors that can affect the project's success, including technical, operational, and economic feasibility. This study helps in identifying potential risks and ensures that the project can be successfully developed and implemented.

Technical Feasibility

The technical feasibility assesses whether the project can be completed with the available technology and resources. For the Paint Application, the following aspects were considered:

Technology Stack

JavaScript (ES6+): The entire functionality of the paint application is implemented using vanilla JavaScript (ES6+), without the use of any external frameworks or libraries. This approach emphasizes native browser capabilities and ensures lightweight, efficient code execution.

HTML5 and CSS3 are utilized for structuring the application's user interface and styling elements, respectively. **HTML5** provides the foundation for creating interactive web content, while **CSS3** enhances the visual presentation and layout of the paint application.

Git: Git is used for version control, enabling collaborative development, code management, and tracking of project changes. GitHub is utilized for hosting the project repository.

Visual Studio Code: Visual Studio Code (VS Code) is chosen as the code editor for its lightweight yet powerful features, extensive plugin ecosystem, and support for JavaScript and web development. It provides a comfortable environment for writing, debugging, and testing JavaScript code.

Operational Feasibility

Operational feasibility evaluates how well the project meets the identified needs and whether it can be integrated into current operations. For the Paint Application, the following factors were considered:

User Acceptance

Usability: The application is designed with a minimalist interface to reduce clutter and enhance the user experience. It includes essential drawing tools (brush, line, rectangle, circle) with easy-to-use controls.

Accessibility: The application is accessible via web browsers without requiring additional software installation, making it convenient for users.

Project Planning & Scheduling

Overview

Project scheduling involves the planning and organization of tasks, resources, and timelines to ensure the successful completion of the paint application development. This section presents the scheduling methodology i.e. the GANTT chart representation.

GANTT Chart

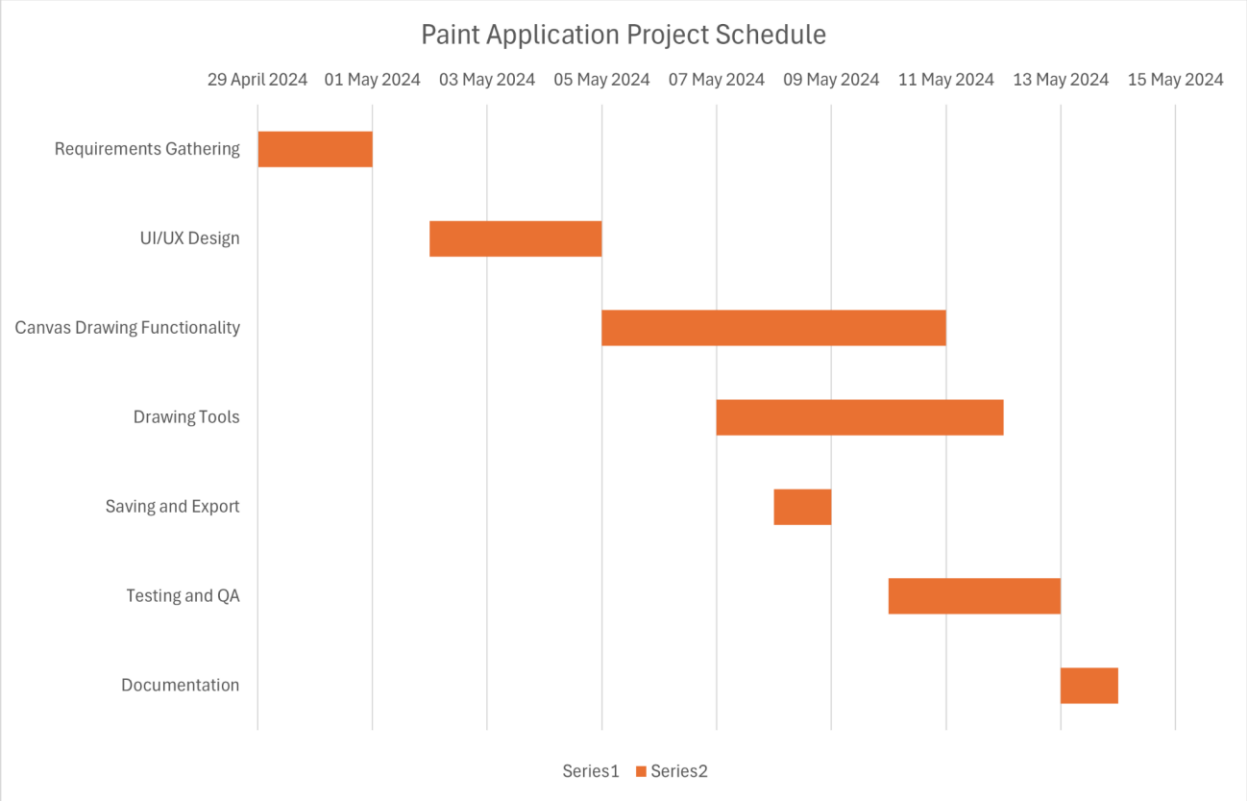
A GANTT chart is a visual representation of project tasks, timelines, and dependencies, displayed as horizontal bars on a time scale. The GANTT chart for the paint application development outlines the following key elements.

1. **Task List**: A list of project tasks, including their names, durations, start dates, and end dates.

2. **Timeline**: A horizontal time scale representing the project duration, typically divided into days, weeks, or months.

3. **Task** Bars: Horizontal bars representing the duration of each task, positioned along the timeline according to their start and end dates.

The Chart



Conclusion

Project scheduling using the GANTT chart provides a systematic approach to planning, organizing, and tracking the paint application development. By identifying critical tasks, dependencies, and timelines, project managers can allocate resources effectively, mitigate risks, and ensure timely project delivery.

Software Requirement Specification (SRS)

Introduction to SRS

The Software Requirement Specification (SRS) outlines the functional and non-functional requirements of the paint application. It serves as a blueprint for the development team, guiding the design, implementation, and testing phases of the project.

Functional Requirements

Drawing Tools

The application shall provide a variety of drawing tools, including brushes, pencils, erasers, shapes (e.g., lines, rectangles, circles), and text tools. Users shall be able to select different brush sizes, shapes, and colors for drawing and editing artwork.

Canvas Management

The application shall offer canvas management features, allowing users to create new canvases, save artwork, and export images in various formats (e.g., PNG, JPEG, SVG).

Layer Support

The application shall support multiple layers, enabling users to organize artwork into separate layers for better organization and editing flexibility. Users shall be able to create, delete, rearrange, and merge layers, as well as adjust layer opacity and blending modes.

User Interface

The user interface shall be intuitive and user-friendly, featuring menus, toolbars, dialogs, and panels for easy navigation and access to drawing tools and features. Tooltips, hover effects, and contextual menus shall be utilized to provide guidance and enhance the user experience.

Non-Functional Requirements

Performance

The application shall exhibit responsive performance, with smooth rendering and minimal latency during drawing operations. Load times for opening and saving files shall be optimized to ensure efficient user interaction.

Compatibility

The application shall be compatible with modern web browsers, including Google Chrome, Mozilla Firefox, Safari, and Microsoft Edge. The application shall adhere to web accessibility standards (WCAG) to ensure accessibility for users with disabilities.

Constraints

Software Requirement

Browser Limitations

The application's functionality may be limited by the capabilities and constraints of web browsers, particularly in terms of performance, memory usage, and compatibility with certain features (e.g., file system access).

Hardware Requirements

Users may require hardware components such as a mouse, touchpad, or stylus for optimal interaction with the application, especially during drawing and editing tasks.

Process Model

For this project, the process model suitable is the Iterative Waterfall Model.
The iterative waterfall model was chosen for the paint application's development due to its structured yet adaptable framework. By breaking the process into sequential phases, it provided clear direction while allowing for iterative refinement. This flexibility accommodated evolving requirements and design iterations, ensuring the project remained focused and responsive to change. Ultimately, the model's balance between structure and adaptability contributed to the successful development of the paint application.

Data Flow Diagram (DFD)

Introduction To DFD

A DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions. In the DFD terminology, it is useful to consider each function as a process that consumes some input data and produces some output data. The DFD (also known as the bubble chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system).

Types of DFD

1. **Logical DFD**: It concentrates on the system process, and data flow. It deals with abstract concepts rather than the actual implementation of data flow and processes.

2. **Physical DFD**: This type of DFD shows how data flow is implemented in the system. It is more specific and close to the implementation.

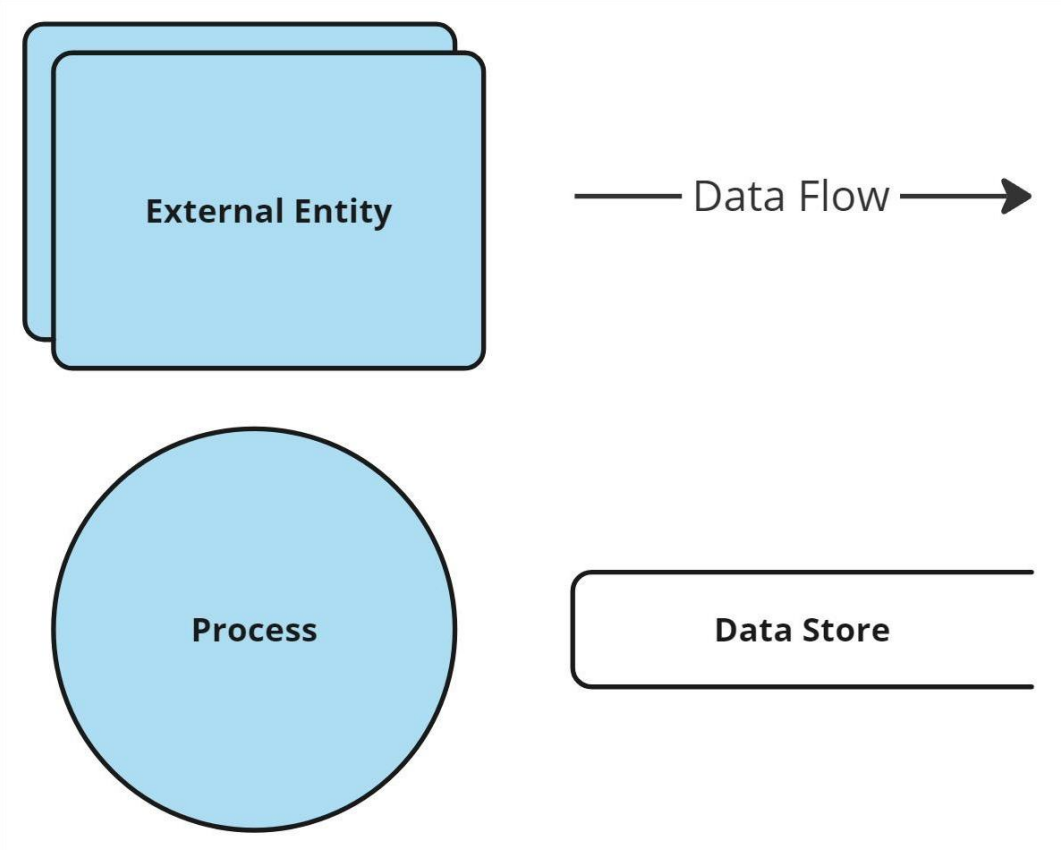
Different Components and Symbols Used In DFD

A Data Flow Diagram (DFD) is a graphical representation of the flow of data through a system or process. It uses a set of standard symbols to represent different elements of the system. Here are the main elements of a DFD and their corresponding symbols:

1. **External Entity**: Symbol: A rectangle or a square Represents an external source or destination of data that interacts with the system but is outside

of its boundaries. Examples include users, other systems, databases, or external devices.

2. **Process**: Symbol: A circle or a rounded rectangle Represents a transformation or manipulation of data within the system. Processes receive input data, perform operations on it, and produce output data.
3. **Data Flow**: Symbol: An arrow Represents the movement or flow of data between processes, external entities, and data stores. The direction of the arrow shows the direction of data flow.
4. **Data Store**: Symbol: A parallel line or an open-ended rectangle Represents a repository or a place where data is stored and retrieved. Examples include databases, files, or temporary storage areas.

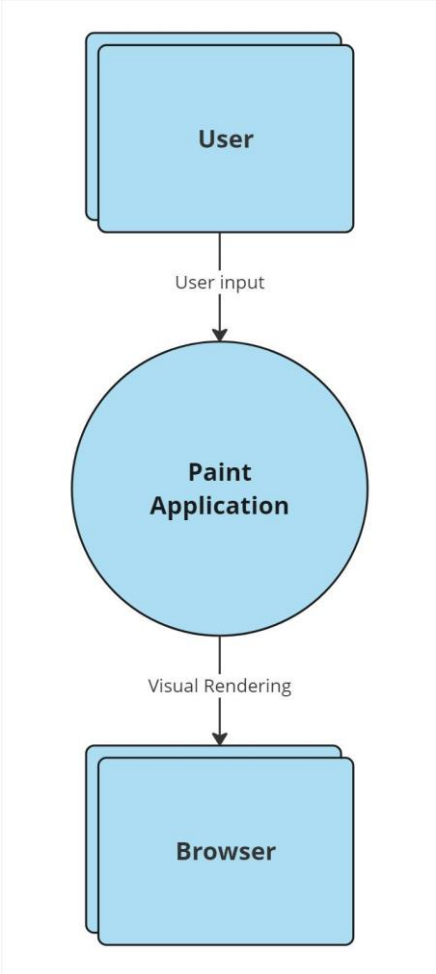


Context Level Diagram For Paint Application

Intro To Context Diagrams

A context-level Data Flow Diagram (DFD) is a high-level, single-process DFD that represents the entire system or application as a single process and shows its interactions with external entities. It provides an overview of the system's boundaries and relationship with the external environment.

DFD (Context)



In this context-level DFD:

- The single process "Paint Application" represents the entire paint application system.
- The external entity "User" interacts with the paint application by providing input, such as mouse or keyboard events.
- The data flow "User Input (mouse/keyboard events)" represents the user's input flowing into the paint application process.
- The external entity "Browser" interacts with the paint application for visual rendering purposes.
- The data flow "Visual Rendering" represents the rendered canvas or visual output flowing from the paint application process to the browser for display.

Level 1 DFD For The Paint Application

A Level 1 Data Flow Diagram (DFD) is the second level of abstraction in the hierarchy of DFDs. It provides a more detailed view of the system by breaking down the main process shown in the context-level DFD into its sub-processes and data flows.

In the context-level DFD, the entire system is represented as a single process, with the focus being on the interactions between the system and external entities. The Level 1 DFD takes that single process and decomposes it into its major sub-processes, revealing the internal structure and data flows within the system.

Purpose of a Level 1 DFD

1. Decompose the main process: The single process shown in the context-level DFD is broken down into its major sub-processes, giving a better understanding of the system's internal structure and functionality.

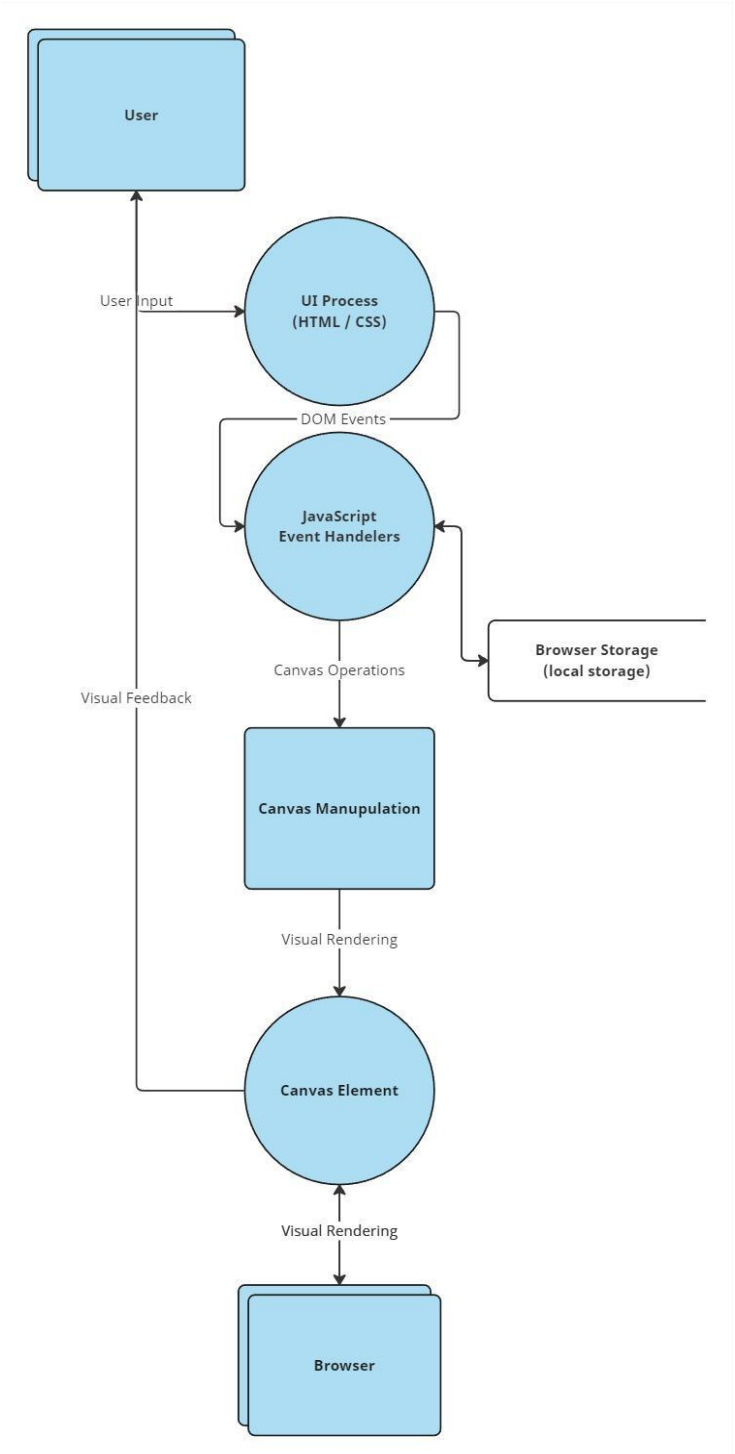
2. Show internal data flows: The data flows between the sub-processes are represented, depicting the movement of data within the system.
3. Identify additional external entities: While the context-level DFD shows the interactions with external entities, the Level 1 DFD may reveal additional external entities that interact with specific sub-processes within the system.
4. Provide a more detailed view: The Level 1 DFD provides a more detailed representation of the system compared to the context-level DFD, allowing for a better understanding of the system's internal workings.

In the context of the paint application, the Level 1 DFD decomposes the "Paint Application" process into sub-processes like UI handling, event processing, canvas manipulation, and rendering. It shows the data flows between these sub-processes and also includes external entities like the browser's local storage.

The Level 1 DFD serves as a bridge between the high-level context-level DFD and the more detailed lower-level DFDs that may further decompose specific sub-processes. It provides a balanced view of the system, showing both its internal structure and its interactions with external entities.

By creating a Level 1 DFD, analysts and developers can better understand the system's components, their relationships, and the flow of data within the system. This understanding is crucial for designing, developing, and maintaining complex systems effectively.

DFD (Level 1)



In this Level 1 DFD:

1. **The "User"** external entity and the "User Input (mouse/keyboard events)" data flow remain the same as in the context-level DFD.
2. **The "UI Process"** represents the process that handles the user interface and captures user interactions, generating DOM events.
3. **The "Event Handlers"** process represents the JavaScript code that listens for DOM events and performs the necessary canvas operations.
4. **The "Canvas Manipulation"** process handles the actual drawing and updating of the canvas based on the user's actions and the selected tool.
5. **The "Canvas Element"** process represents the HTML canvas element where the drawings are rendered.
6. **The "Browser Storage (localStorage)"** external entity represents the browser's localStorage, which can be used to store and retrieve application data or user preferences.
7. The data flows between the processes represent the movement of data, such as DOM events, canvas operations, and visual rendering data.
8. **The "Visual Rendering"** data flow remains the same as in the context-level DFD, representing the rendered canvas or visual output flowing to the browser for display.

This Level 1 DFD provides a more detailed view of the paint application's internal processes and data flows, while still maintaining a high-level perspective. It shows the decomposition of the "Paint Application" process into sub-processes like UI handling, event processing, canvas manipulation, and rendering.

Entity-Relationship Diagram

Introduction

The Entity-Relationship (ER) diagram illustrates the entities, attributes, and relationships within the paint application's database schema. It provides a visual representation of the data model, helping identify the entities and their attributes, and the relationships between them.

The entity-relationship data model is based on a perception of a real-world that consists of a collection of basic objects called entities and of relationships among these objects. An entity is an “object” in the real world that is distinguishable from other objects. E.g. each customer is an entity and rooms can be considered to be entities. Entities are described by a set of attributes. E.g. the attributes Room no. and Room type describe a particular Room in a hotel. The set of all entities of the same type and the set of all relationships of the same type are termed as an entity set and relationship set respectively

ER Diagram Components

1. **Entity** (User, Canvas, Layer, Drawing Tool, Color Palette)
    a. Entities represent the real-world objects or concepts that the database stores information about. They are typically nouns and serve as the main building blocks of the ER diagram. Examples of entities in a paint application may include User, Canvas, Layer, Drawing Tool, and Color Palette.

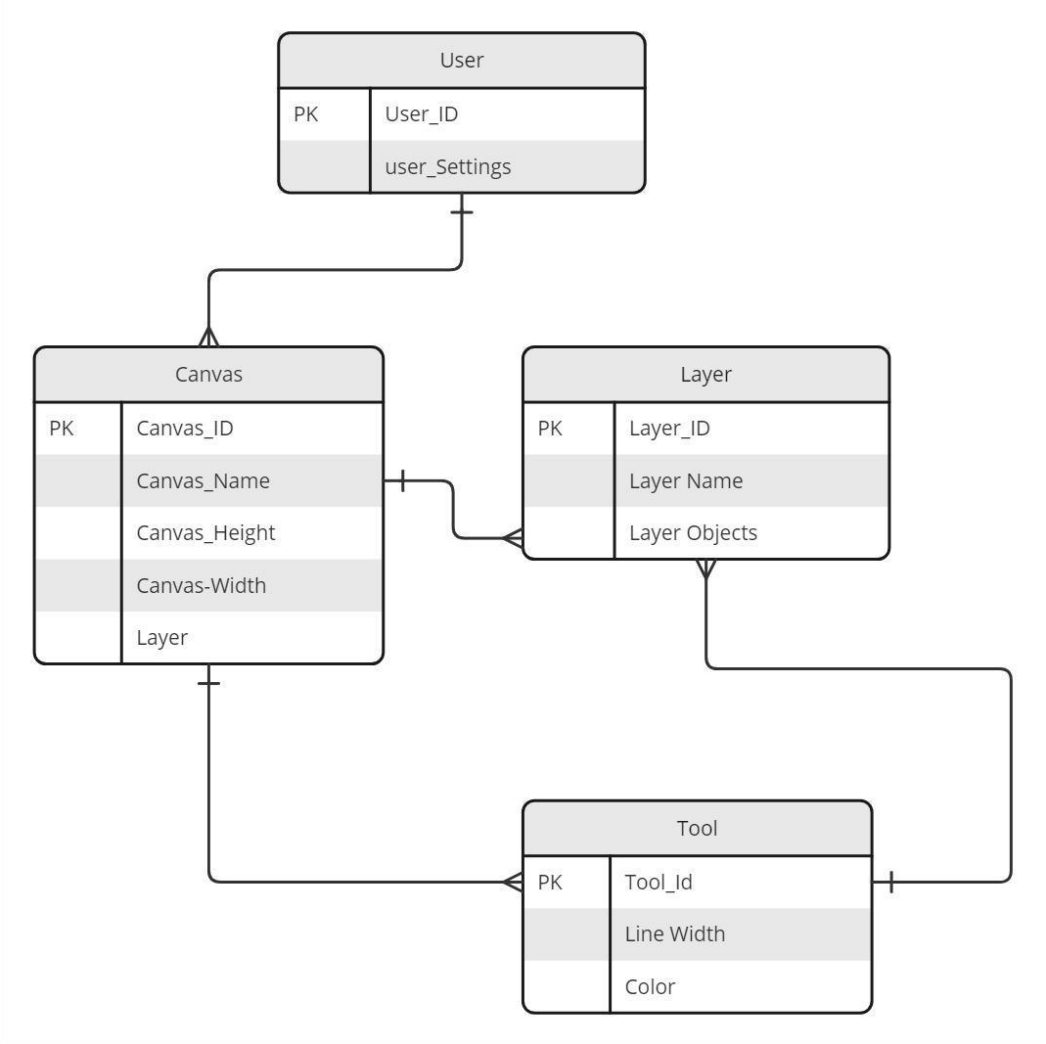
2. **Attributes**
    a. Attributes are the properties or characteristics of entities. They describe the details or qualities of an entity. Each entity has one or more attributes that define it. Attributes are typically represented

as ovals connected to their respective entity by a line. Examples of attributes for the User entity might include UserID, Username, Email, and Password.

3. **Relationships**
    a. Relationships represent the connections or associations between entities. They define how entities are related to each other and how they interact. Relationships are typically represented as lines connecting two entities, with labels indicating the nature of the relationship (e.g., one-to-one, one-to-many, many-to-many). For example, a User may have multiple Canvases, indicating a one-to-many relationship between User and Canvas entities

4. **Keys:**
    a. Keys are attributes or combinations of attributes that uniquely identify each instance of an entity. They ensure data integrity and help establish relationships between entities. In an ER diagram, keys are often designated as primary keys (PK) or foreign keys (FK). The primary key uniquely identifies each record in an entity, while foreign keys establish relationships between entities by referencing the primary key of another entity.

ER Diagram For Paint Application



The above Entity Relationship (ER) diagram represents the data model for a paint application. Here are some key observations and conclusions about this ER diagram:

1. **User Entity:** The User entity represents the application's users. It has attributes like User_ID and User_Settings, which can store user-specific preferences or settings.

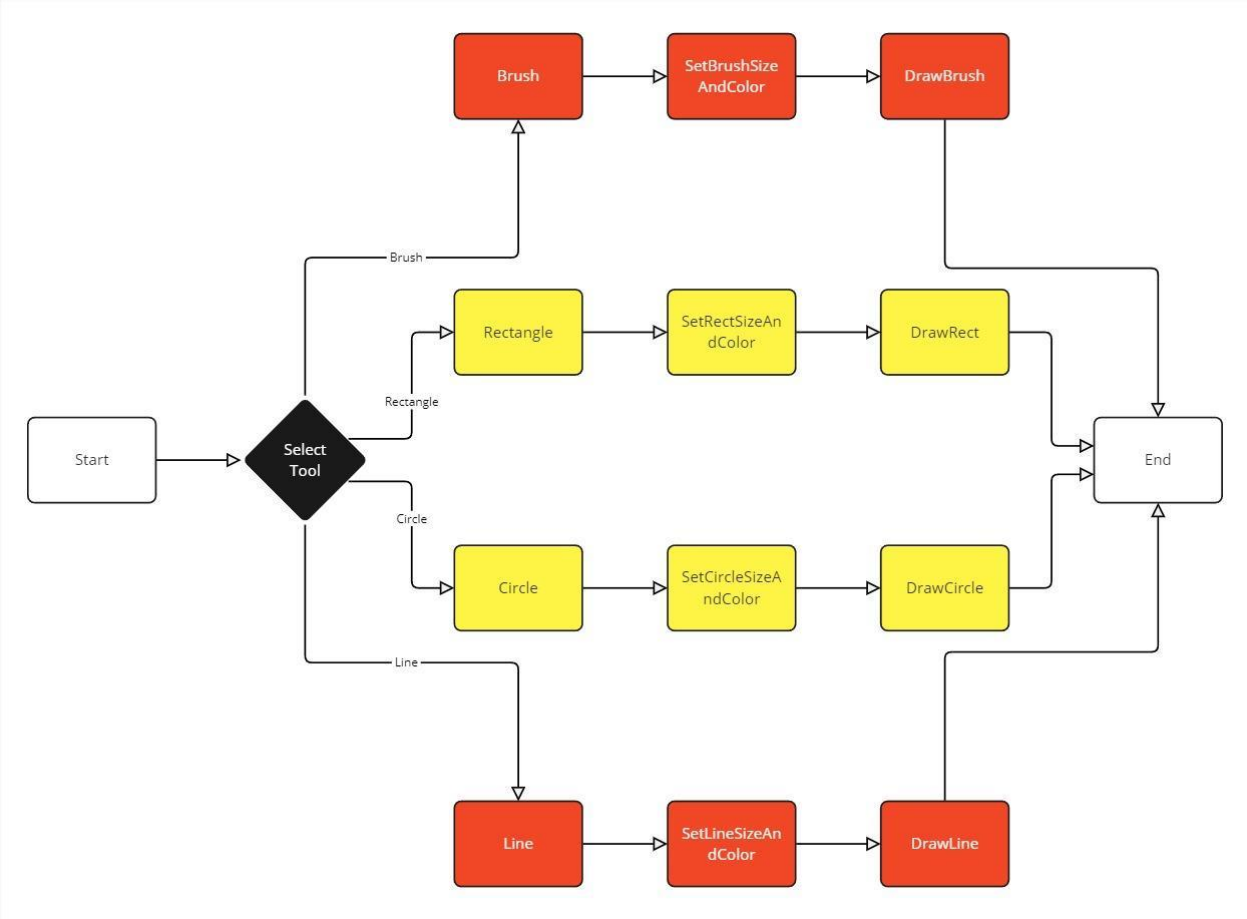
2. **Canvas Entity:** The Canvas entity is central to the application, as it represents the drawing area or canvas. It has attributes like Canvas ID, Canvas Name, Canvas Height, and Canvas Width to define the canvas dimensions. Notably, it has a one-to-many relationship with the Layer entity, indicating that a canvas can have multiple layers.
3. **Layer Entity:** The Layer entity represents individual layers within the canvas. Each layer has a unique Layer ID, Layer Name, and Layer Objects attribute. The Layer Objects attribute likely stores the drawing objects (e.g., shapes, lines, brush strokes) present on that particular layer.
4. **Tool Entity:** The Tool entity represents the various drawing tools available in the application, such as brushes, pencils, shapes, etc. It has attributes like Tool_Id, Line Width, and Color, which define the properties of the selected tool.
5. **Relationships:**
    a. The User entity has a one-to-many relationship with the Canvas entity, indicating that a user can have multiple canvases.
    b. The Canvas entity has a one-to-many relationship with the Layer entity, meaning a canvas can have multiple layers.
    c. The Layer entity has a one-to-many relationship with the Tool entity, suggesting that a layer can have multiple drawing objects created with different tools.
6. **Normalization:** The ER diagram is normalized, with entities having their own unique identifiers (primary keys) and relationships established between them to avoid data redundancy.

Control Flow Diagram

A Control Flow Diagram (CFD) illustrates the sequence of operations and the control logic that dictates the flow of the application. For a Paint Application,

the CFD will demonstrate the user interactions with the application and how different functionalities are triggered in response.

Control Flow Diagram For Paint Application



Use Case Diagram

A use case diagram is a visual representation of the interactions between users (actors) and the system. For a simple paint application with basic drawing tools, here's a breakdown of the use cases and the actors involved.

Actors

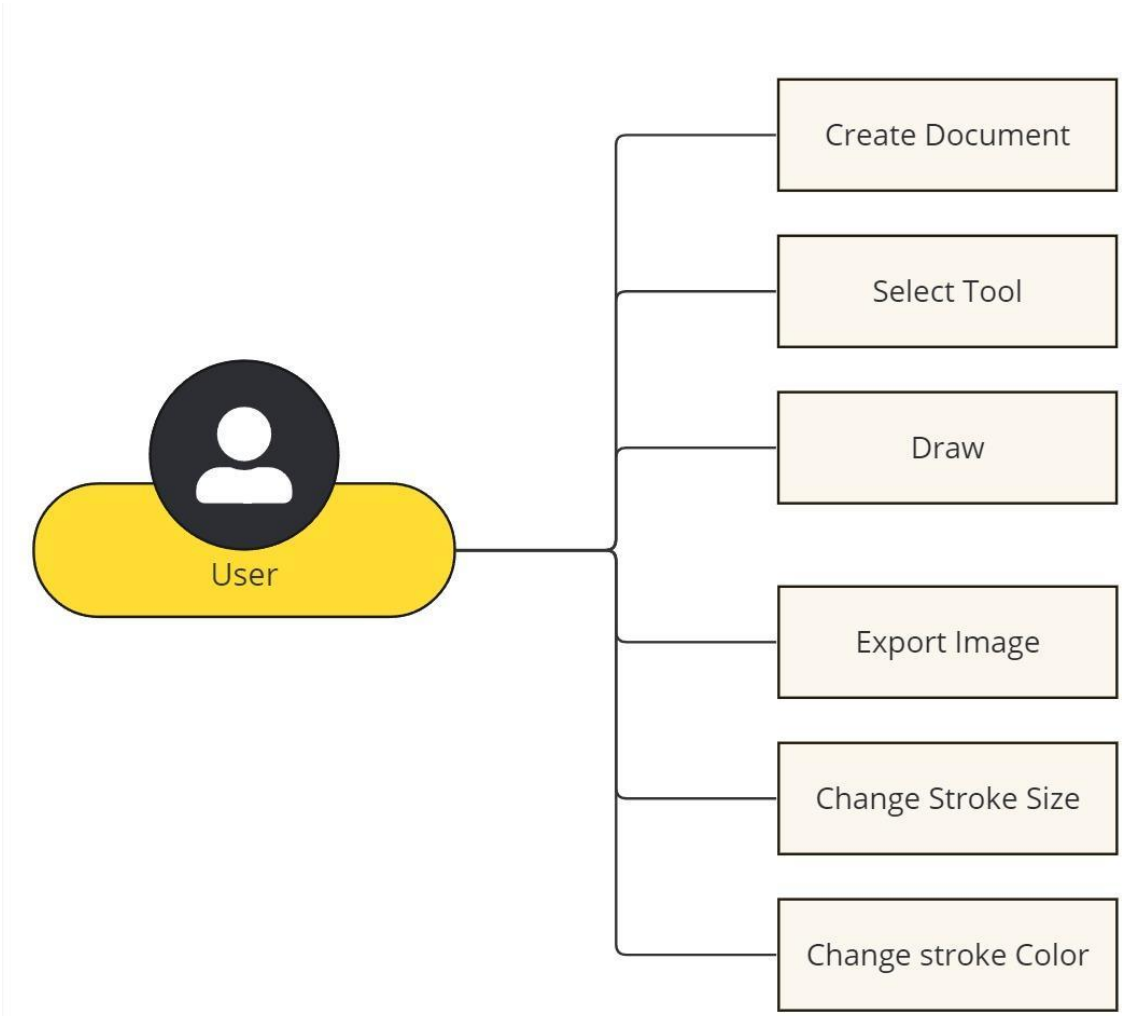
1. **User**: The person using the paint application.

Use Cases

Following are the use cases of the Paint Application.

1. **Creating Document**: the user has the ability to create a new document.
2. **Select Tool**: the user has the ability to select the tool of his liking from the provided 4 tools (brush tool, line tool, rect tool, circle tool).
3. **Draw**: the user can draw on the created canvas.
4. **Export Image**: the user can export the drawn image in the form of a .png file.
5. **Change Stroke Size**: the user can change brush size.
6. **Change Stroke Colo**r: the user can change brush color.

Use Case Diagram for Paint Application



System Design

Introduction to System Design

System Design involves detailing the architecture and structure of the application, including its modules, databases, user interface, and other components.

Modularization Details

Modularization involves breaking down the system into smaller, manageable modules that can be developed, tested, and maintained independently. In the context of the Paint Application, following are the modules.

Drawing Canvas Module

This module forms the core of the paint application, providing users with a blank canvas to draw and create artwork. It includes functionalities such as drawing tools, color selection, brush customization, and layer management.

Toolbox Module

The toolbox module encompasses various drawing tools and editing features, including brushes, pencils, erasers, shapes, text, and image manipulation tools. Each tool is designed to serve a specific purpose and enhance user creativity.

User Interface Module

This module focuses on the design and implementation of the graphical user interface (GUI) of the application. It includes components such as menus,

toolbars, dialogs, and panels, all designed to enhance user interaction and experience.

User Interface Design

User Interface (UI) Design involves crafting the visual and interactive aspects of digital products like websites, mobile apps, or software applications. It aims to produce interfaces that are intuitive and easy to use, fostering seamless communication between users and the system. Through a blend of aesthetic considerations and user-centric design approaches, UI designers strive to create interfaces that optimize the overall user experience, encouraging positive interactions with the product.

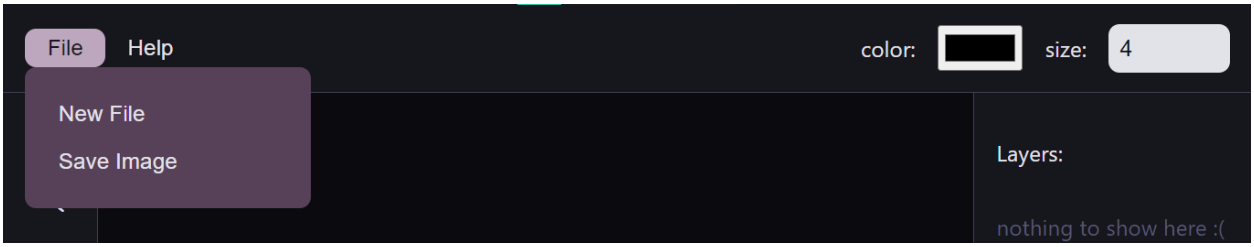
UI Design for Paint Application

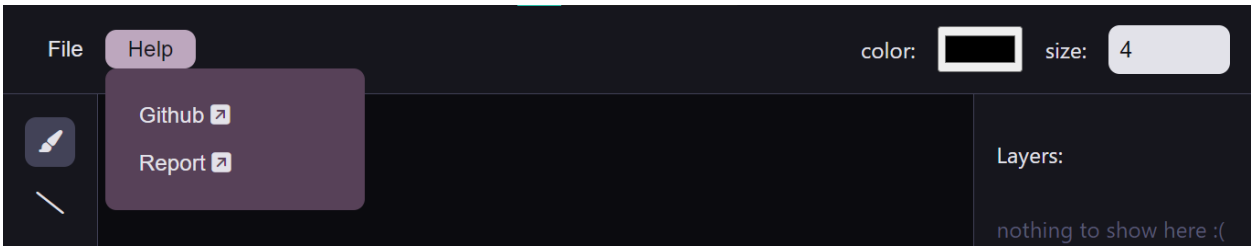
Following are the different types of user interfaces a user might encounter, while using the Paint Application.

1. The Menu bar and Submenus.
2. Documentation Creation Screen.
3. The Tool Bar and Side bar.
4. With no Document
5. With Document

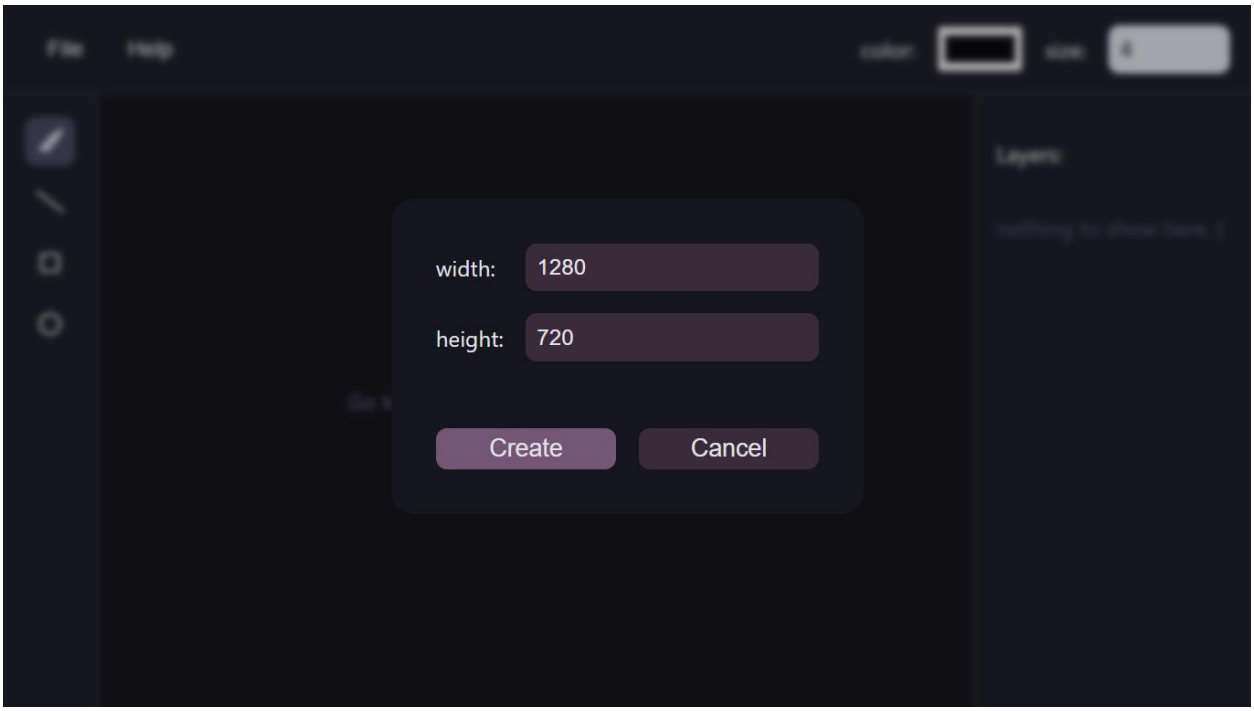
The upcoming images are at **175% zoom** level and may not truly represent the Paint Application at its fullest

The Menu Area

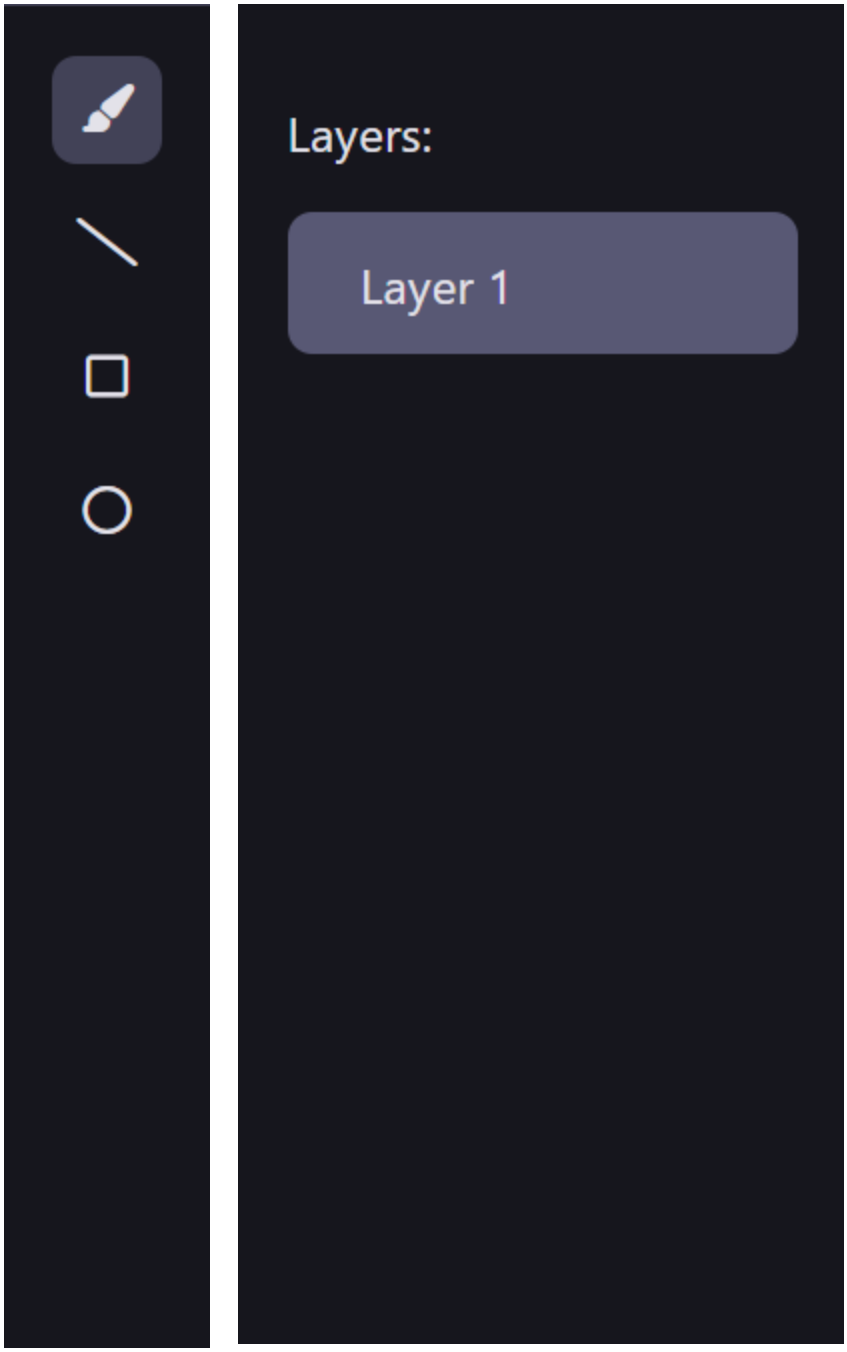




Documentation Creation Screen



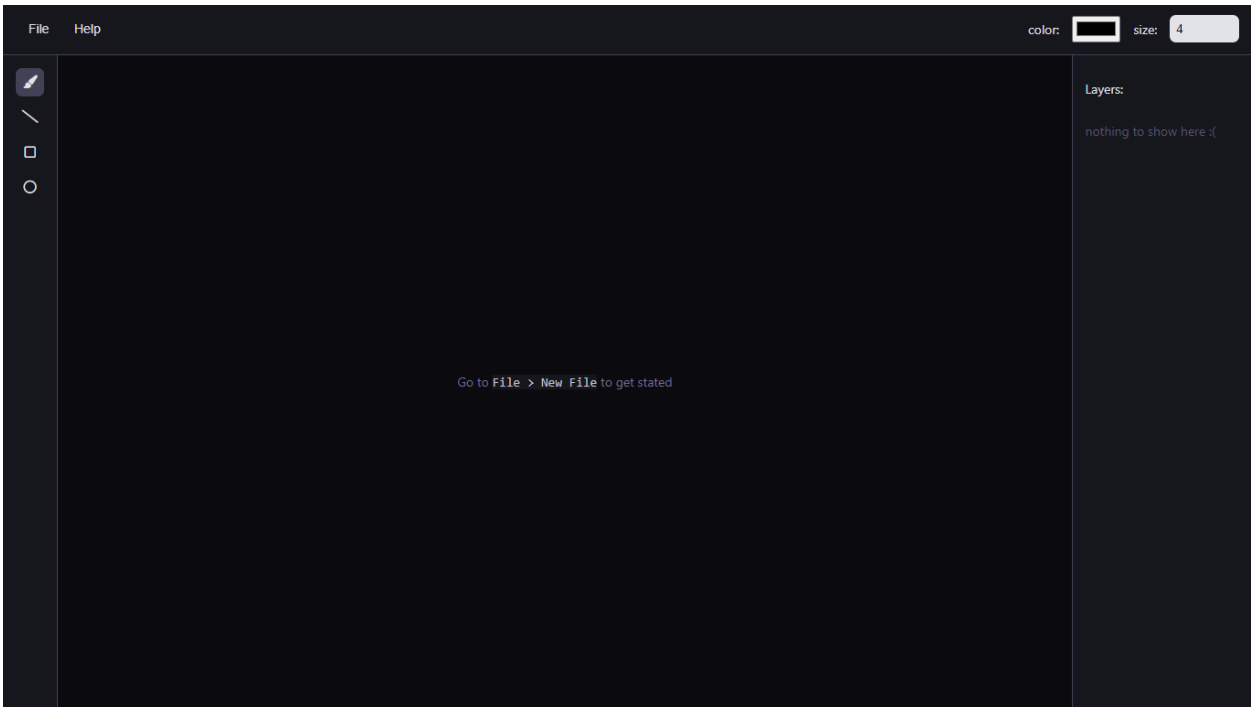
Toolbar & Sidebar



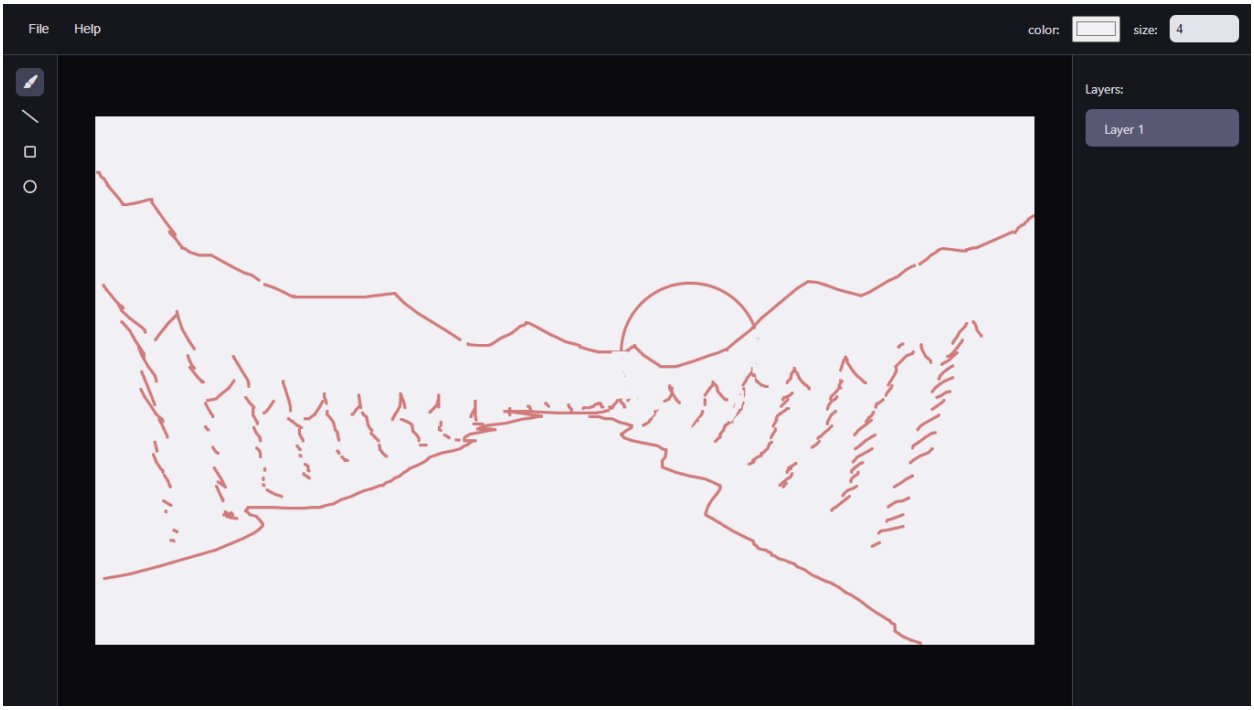
The **toolbar** typically appears at the top of the application interface and contains icons or buttons representing different drawing tools and actions. The toolbar contains buttons to select different tools, namely the brush, the circle, the rectangle and the line tool, when the user selects a tool by clicking on the button, the javascript event is fired to identify the tool selected.

The **sidebar** typically appears on one side of the application interface (usually the left or right) and contains additional options, settings, or navigation controls

Without Document



With Document



Coding

Coding is the process of transforming software design and specifications into a functional program. This phase involves writing the source code in a programming language and ensuring it meets the defined requirements and design specifications. In the context of the paint application, coding encompasses the development of features such as the toolbar, canvas, drawing tools, and more.

Comments and Description of Coding Segments

Effective commenting and detailed descriptions of code segments are crucial for maintaining code readability and facilitating future modifications. Below are the coding segments with comments and descriptions for key functionalities in the paint application.

Comments in HTML: In HTML, comments are written between **<!-- and -->** tags are not displayed in the browser. They are used to add notes, explanations, or reminders to the code.

Comments in CSS: In CSS, comments are written between **/* and */**. Similar to HTML comments, CSS comments are not rendered by browsers and are used to provide context or explanations for CSS rules

Comments in JavaScrip: Here we have two ways to type comments:

1. Single Line Comment: Start with **//** and continue until the end of the line.
2. Multi-line comments: Enclosed between **/* and */,** allowing for comments spanning multiple lines.

Comments For **index.html**

```
<!--link to styling and fontawesome script -->
<!-- body start (body is grid) -->
  <!-- New File Modal -->
  <!-- Header containing the menu bar -->
  <!-- Main Start-->
    <!-- Toolbar -->
    <!-- Sidebar -->
    <!-- Canvas Wrapper that contains all HTML5 Canvases -->
  <!-- Main end -->
<!-- body end -->
```

Standardization of Coding

Adhering to coding standards ensures consistency, readability, and maintainability across the codebase. Below are some standard practices followed in the paint application development:

Consistent Naming Conventions: Use clear and descriptive names for variables, functions, and classes.Follow a consistent naming pattern, such as camelCase for variables and functions, and PascalCase for classes.

Modular Code: Break down the code into smaller, reusable modules and functions.Ensure each module has a single responsibility and can be easily tested and maintained.

Validation Checks

Validation checks ensure the inputs and data used in the application are valid and within acceptable ranges. Below are the validation checks in the paint application:

Validation in Document Dimensions Input

When prompting the user to input their desired document dimensions, it is essential to validate their entries for correct syntax and valid numerical values. The following code segment handles this validation in the Paint Application.

```
docWidthInput.addEventListener("input", function (event) {
  this.value = this.value.replace(/[^\d]/g, "");
  if (Number.parseInt(this.value) > 9999) {
    this.value = 9999;
  }
});

docHeightInput.addEventListener("input", function (event) {
  this.value = this.value.replace(/[^\d]/g, "");
  if (Number.parseInt(this.value) > 9999) {
    this.value = 9999;
  }
});
```

Code For JavaScript

Entry Point (index.js)

```
import DrawingCanvas from "./src/js/DrawinfCanvas.mjs";
import { onMenuHoverEffect } from "./src/js/menuHover.mjs";
import { init } from "./src/js/meunBtnClick.mjs";
import { initNewFileModal } from "./src/js/newFileModal.mjs";
import { docWidthInput, docHeightInput } from
"./src/js/newFileModal.mjs";
import { NEW_DOC_CREATED } from "./src/js/newFileModal.mjs";

const wrapper = document.getElementById("canvas-wrapper");
const ACTIVE_CANVAS_CHANGED_EVENT = "activeCanvasChanged";
const activeCanvasChanged = new
Event(ACTIVE_CANVAS_CHANGED_EVENT);

onMenuHoverEffect();
init(); //initialize menu button functionality
initNewFileModal();

let canvasArray = [];
let baseCanvas = null;
let docDimentions = {};
let scale = 1;
let activeCanvas = null;

const tools = {
  brush: "brush",
  line: "line",
  rect: "rect",
  circle: "circle",
};
let currentTool = tools.brush;
```

```
const brushBtn = document.getElementById("brush-tool-btn");
const lineBtn = document.getElementById("line-tool-btn");
const rectBtn = document.getElementById("rect-tool-btn");
const circleBtn = document.getElementById("circle-tool-btn");

brushBtn.classList.add("active-tool");

const toolBtnArray = [brushBtn, lineBtn, rectBtn, circleBtn];

brushBtn.addEventListener("click", () => {
  currentTool = tools.brush;
  toolBtnArray.forEach((btn) => {
    if (btn != brushBtn) {
      btn.classList.remove("active-tool");
    } else {
      btn.classList.add("active-tool");
    }
  });
});
lineBtn.addEventListener("click", () => {
  currentTool = tools.line;
  toolBtnArray.forEach((btn) => {
    if (btn != lineBtn) {
      btn.classList.remove("active-tool");
    } else {
      btn.classList.add("active-tool");
    }
  });
});
rectBtn.addEventListener("click", () => {
  currentTool = tools.rect;
  toolBtnArray.forEach((btn) => {
    if (btn != rectBtn) {
```

```
      btn.classList.remove("active-tool");
    } else {
      btn.classList.add("active-tool");
    }
  });
});
circleBtn.addEventListener("click", () => {
  currentTool = tools.circle;
  toolBtnArray.forEach((btn) => {
    if (btn != circleBtn) {
      btn.classList.remove("active-tool");
    } else {
      btn.classList.add("active-tool");
    }
  });
});

//events

let initX = 0;
let initY = 0;
let currentX = 0;
let currentY = 0;
let isDrawing = false;
let isMouseDown = false;

let tempLayer = null;

const saveBtn = document.getElementById("save-file-btn");
saveBtn.addEventListener("click", saveImage);

window.addEventListener(NEW_DOC_CREATED, (e) => {
  resetAppState();
  docDimentions = {
```

```
    width: Number.parseInt(docWidthInput.value),
    height: Number.parseInt(docHeightInput.value),
  };
  baseCanvas = new DrawingCanvas(
    wrapper,
    docDimentions,
    "canvas",
    canvasArray.length
  );
  canvasArray.push(baseCanvas);
  scale = dertermineScale(docDimentions.width,
docDimentions.height);
  activeCanvas = canvasArray[0];
  activeCanvas.canvas.style.scale = scale;
  window.dispatchEvent(activeCanvasChanged);
  const layerWrapper = document.getElementById("layers-
wrapper");
  layerWrapper.innerHTML = `<article class="layer" id="0">
      <p class="layer-name ">Layer 1</p>
    </article>`;
});

window.addEventListener(ACTIVE_CANVAS_CHANGED_EVENT, (e) => {
  activeCanvas.canvas.style.scale = scale;
  activeCanvas.canvas.addEventListener("mousedown", (e) => {
    if (currentTool != tools.brush && !tempLayer) {
      tempLayer = new DrawingCanvas(
        wrapper,
        docDimentions,
        "temp",
        canvasArray.length
      );
      canvasArray.push(tempLayer);
      activeCanvas = canvasArray[canvasArray.length - 1];
```

```
      window.dispatchEvent(activeCanvasChanged);
    }

    initX = e.offsetX;
    initY = e.offsetY;
    isDrawing = true;
  });

  activeCanvas.canvas.addEventListener("mousemove", (e) => {
    currentX = e.offsetX;
    currentY = e.offsetY;
  });

  activeCanvas.canvas.addEventListener("mouseup", (e) => {
    if (tempLayer && currentTool != tools.brush) {
      const last = canvasArray[canvasArray.length - 1];
      const secondLast = canvasArray[canvasArray.length - 2];

      secondLast.ctx.drawImage(last.canvas, 0, 0);
      last.ctx.clearRect(0, 0, docDimentions.width,
docDimentions.height);

      canvasArray.pop();
      activeCanvas.remove();
      tempLayer = null;

      activeCanvas = canvasArray[canvasArray.length - 1];
      window.dispatchEvent(activeCanvasChanged);
    }

    isDrawing = false;
    activeCanvas.ctx.beginPath();
  });
```

```
  activeCanvas.canvas.addEventListener("mouseleave", (e) => {
    if (currentTool == tools.brush) {
      isDrawing = false;
      activeCanvas.ctx.beginPath();
    }
  });

  activeCanvas.canvas.addEventListener("mouseenter", () => {
    if (currentTool == tools.brush && isMouseDown) {
      isDrawing = true;
      activeCanvas.ctx.beginPath();
    }
  });
});

window.addEventListener("mousedown", () => {
  isMouseDown = true;
});
window.addEventListener("mouseup", () => {
  isMouseDown = false;
});

updateFrame();
function updateFrame() {
  requestAnimationFrame(updateFrame);
  if (!activeCanvas) return;
  switch (currentTool) {
    case tools.brush:
      drawBrush(activeCanvas.ctx);
      break;
    case tools.line:
      drawLine(activeCanvas.ctx);
      break;
    case tools.rect:
```

```
      drawRect(activeCanvas.ctx);
      break;
    case tools.circle:
      drawCircle(activeCanvas.ctx);
      break;
  }
}

const colorInput = document.getElementById("brush-color-
input");
const bruhSizeInput = document.getElementById("brush-size-
input");

function drawBrush(ctx) {
  if (!isDrawing) return;

  ctx.globalCompositeOperation = "source-over";
  ctx.lineTo(currentX, currentY);
  ctx.lineCap = "round";
  ctx.lineWidth = bruhSizeInput.value;
  ctx.strokeStyle = colorInput.value;
  ctx.stroke();
  ctx.beginPath();
  ctx.moveTo(currentX, currentY);
}

function drawLine(ctx) {
  if (!isDrawing) return;
  ctx.clearRect(0, 0, docDimentions.width,
docDimentions.height);
  ctx.beginPath();
  ctx.moveTo(initX, initY);
  ctx.lineWidth = bruhSizeInput.value;
  ctx.strokeStyle = colorInput.value;
```

```
  ctx.lineTo(currentX, currentY);
  ctx.stroke();
}

function drawRect(ctx) {
  if (!isDrawing) return;
  ctx.clearRect(0, 0, docDimentions.width,
docDimentions.height);
  ctx.beginPath();

  ctx.lineWidth = bruhSizeInput.value;
  ctx.strokeStyle = colorInput.value;
  ctx.rect(initX, initY, currentX - initX, currentY - initY);
  ctx.stroke();
}

function drawCircle(ctx) {
  if (!isDrawing) return;

  const mag = Math.sqrt(
    Math.pow(currentX - initX, 2) + Math.pow(currentY - initY,
2)
  );

  ctx.clearRect(0, 0, docDimentions.width,
docDimentions.height);
  ctx.beginPath();
  ctx.lineWidth = bruhSizeInput.value;
  ctx.strokeStyle = colorInput.value;
  ctx.arc(initX, initY, mag, 0, 2 * Math.PI);
  ctx.stroke();
}

function dertermineScale(width, height) {
```

```
  if (width > height) {
    return 0.75 * (1280 /
Number.parseInt(docWidthInput.value));
  } else if (width == height) {
    return 0.75 * (720 /
Number.parseInt(docHeightInput.value));
  } else {
    return 0.75 * (720 /
Number.parseInt(docHeightInput.value));
  }
}

function saveImage() {

  if (activeCanvas == null) {
    console.error("cannot save when no document is present");
    return;
  }

  let saveCanvas = new DrawingCanvas(
    wrapper,
    docDimentions,
    "temp",
    "saving-canvas"
  );
  saveCanvas.canvas.style.scale =
  dertermineScale(docDimentions.width, docDimentions.height);
  saveCanvas.ctx.fillStyle = "white";
  saveCanvas.ctx.fillRect(0, 0, docDimentions.width,
  docDimentions.height);
  saveCanvas.ctx.drawImage(activeCanvas.canvas, 0, 0);
  let canvasImage = saveCanvas.canvas.toDataURL("image");

  let xhr = new XMLHttpRequest();
```

```
  xhr.responseType = "blob";
  xhr.onload = function () {
    let a = document.createElement("a");
    a.href = window.URL.createObjectURL(xhr.response);
    a.download = "image_name.png";
    a.style.display = "none";
    document.body.appendChild(a);
    a.click();
    a.remove();
    saveCanvas.remove();
    saveCanvas = null;
  };
  xhr.open("GET", canvasImage);
  // This is to download the   canvas Image
  xhr.send();
}

function resetAppState() {
  isDrawing = false;
  isMouseDown = false;
  wrapper.innerHTML = "";
  canvasArray = [];
  activeCanvas = null;
  tempLayer = null;
}
```

The DrawingCanvas Object (DrawingCanvas.mjs)

```
class DrawingCanvas {
  constructor(parent, {width, height}, className, id) {
    this.canvas = document.createElement("canvas");
    this.width = width;
    this.height = height;

    this.canvas.height = height;
    this.canvas.width = width;
    this.canvas.classList.add(className);
    this.canvas.id = id;
    this.id = id;

    this.ctx = this.canvas.getContext("2d");

    parent.append(this.canvas);
    this.remove = () => {
      parent.removeChild(this.canvas);
    };
  }
}

export default DrawingCanvas;
```

Function for Creating a New Document (NewFileModal.mjs)

```
const bruhsSizeInput =    document.getElementById("brush-size-input");
const newFileModal = document.getElementById("new-file-modal");

const docWidthInput = document.getElementById("doc-width-input");
const docHeightInput = document.getElementById("doc-height-input");

const createBtn = document.getElementById("new-file-create-btn");
const cancelBtn = document.getElementById("new-file-cancel-btn");

const NEW_DOC_CREATED = "newDocCreated";
const newDocCreated = new Event(NEW_DOC_CREATED);

function initNewFileModal() {
  //validate only text

  bruhsSizeInput.addEventListener("input", function (event) {
    this.value = this.value.replace(/[^\d]/g, "");
    if (Number.parseInt(this.value) > 75) {
      this.value = 75;
    }
    if (Number.parseInt(this.value) <= 0) {
      this.value = 1;
    }
  });
```

```
docWidthInput.addEventListener("input", function (event) {
    this.value = this.value.replace(/[^\d]/g, "");
    if (Number.parseInt(this.value) > 9999) {
      this.value = 9999;
    }
  });

  docHeightInput.addEventListener("input", function (event) {
    this.value = this.value.replace(/[^\d]/g, "");
    if (Number.parseInt(this.value) > 9999) {
      this.value = 9999;
    }
  });

  createBtn.addEventListener("click", () => {
    window.dispatchEvent(newDocCreated);
    newFileModal.classList.remove("modal-active");
  });

  cancelBtn.addEventListener("click", () => {
    newFileModal.classList.remove("modal-active");
  });
}

export { initNewFileModal };
export { docWidthInput, docHeightInput };
export { NEW_DOC_CREATED, newDocCreated };
```

Menu Hover Effect (menuHover.mjs)

```
function onMenuHoverEffect() {
  const parents = document.querySelectorAll(".submenu-parent");

  parents.forEach((parent) => {
    const subMenu = parent.querySelector(".submenu");
    parent.addEventListener("mouseenter", (e) => {
      subMenu.classList.add("activate-submenu");
    });

    parent.addEventListener("mouseleave", (e) => {
      subMenu.classList.remove("activate-submenu");
    });

    parent.addEventListener("click", () => {
      subMenu.classList.add("activate-submenu");
    });
  });
}

export {onMenuHoverEffect};
```

Testing

Testing is a crucial phase in the software development lifecycle, aimed at ensuring the functionality, performance, and reliability of the application. It involves systematically evaluating the application to identify and rectify defects, ensuring that the final product meets the specified requirements and provides a seamless user experience.

Types of Testing

Functional Testing

Verifies that the application operates according to the specified functional requirements. It focuses on user interactions and the functionality of the features.
Examples: Testing the brush tool, eraser functionality, color picker, and shape drawing tools to ensure they perform as expected.

Non-Functional Testing

Evaluates aspects such as performance, usability, reliability, and security, which are not related to specific functions but are crucial for overall system quality.
Examples: Testing the application's response time, load handling, and user interface consistency across different devices.

Testing Strategies Used

Unit Testing

Involves testing individual components or units of the application to ensure they function correctly in isolation.
Examples: Testing individual functions within the brush tool to ensure each performs correctly.

Integration Testing

Focuses on testing the interaction between different components or modules of the application.
Examples: Ensuring that the color picker works seamlessly with the brush and shape tools.

System Testing

Evaluates the complete and integrated application to ensure it meets the specified requirements.
Examples: Testing the entire drawing application, including all tools and features, to ensure it functions as a cohesive unit.

In summary, thorough testing is essential to ensure the paint application is reliable, efficient, and user-friendly. By employing various testing techniques and strategies, the application can be refined to meet user expectations and function seamlessly.

Future Scope

When considering the future scope of the paint application, it's important to think about potential enhancements, features, or improvements that could be implemented to enhance its functionality, usability, or performance. Here are some ideas for future scope that I have in mind:

Additional Drawing Tools

Introducing new drawing tools such as new shapes, text tools, and an eraser tool to provide users with more options for creating artwork. Layer Support: Implement support for layers, allowing users to organize their drawings into separate layers and manipulate them independently. This can enhance flexibility and enable more complex compositions.

Undo/Redo Functionality

Adding undo and redo functionality to allow users to easily revert or repeat their actions while drawing, providing a more seamless and user-friendly experience.

Customizable Brushes

Allowing users to customize brush properties such as opacity, and hardness to tailor the drawing experience to their preferences.

Import/Export Options

Enabling users to import images as backgrounds or reference images for their drawings and export their artwork in various formats (e.g., PNG, SVG) for sharing or further editing.

Collaborative Drawing

Introducing collaborative drawing features that allow multiple users to work on the same canvas simultaneously, enabling real-time collaboration and creative exploration.

Touchscreen Support

Optimizing the application for touchscreen devices to provide a more intuitive and responsive drawing experience for users on tablets or touch-enabled laptops.

Accessibility Features

Implementing accessibility features such as keyboard shortcuts, screen reader support, and high contrast modes to ensure that the application is usable by a diverse range of users.

Performance Optimization

Continuously optimizing the application's performance to ensure smooth rendering, responsive interactions, and efficient use of system resources, particularly for larger or more complex drawings.

Bibliography

1. Smith, J. (2023). "**Introduction to HTML5 Canvas.**" W3Schools. https://www.w3schools.com/html/html5_canvas.asp

2. Johnson, A. (2021). "**JavaScript: The Definitive Guide.**" O'Reilly Media.

3. Garcia, M., & Lee, S. (2020). "**Enhancing User Experience in Web-Based Drawing Applications.**" Journal of Human-Computer Interaction, 15(2), 123-135. https://doi.org/10.1080/07370024.2020.1802010

4. Brown, T. (2022). "**Getting Started with HTML Canvas.**" MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Getting_started_with_canvas.