**PaintX**

**User guide**

A screenshot of a computer

Description automatically generated

**Getting Started**

When you launch PaintX, you'll be greeted with a clean interface divided into the following areas:

1. **Shape Selector**: On the top bar, you'll find various shapes like rectangles, ellipses, lines, triangles. Click on any shape to select it for drawing on the canvas.
2. **Color Selector**: Next to the shapes, there's a color palette. Click on any color to change the drawing color. The selected color applies to both the outline and fill of shapes.
3. **Line Size**: Adjust the slider to set the thickness of the lines for your shapes
4. **Tools**:
   * **Eraser**: Removes parts(shapes) of your drawing.
   * **Mover**: Allows you to reposition shapes on the canvas.
   * **Resizer**: Lets you adjust the size of the selected shapes.
5. **Save/Load Buttons**: Use these to save your current drawing or load a previous one.

**Drawing Shapes**

1. Select a shape from the Shape Selector.
2. Choose your desired color from the Color Selector.
3. Adjust the line size if necessary.
4. Click and drag on the canvas to draw the shape.

**Editing Shapes**

* To erase, select the eraser tool and rub it over the area you want to clear.
* To move a shape, select it with the mover tool, then click and drag to the new location.
* To resize a shape, select it with the resizer tool and drag the handles that appear to adjust its dimensions.

**Saving and Loading**

* To save your drawing, click the save button. PaintX saves files in the **.pnx** format.
* To load a drawing, click the load button. Your previous work will appear on the canvas, ready for further editing.

**Program documentation**

The challenge was about to create a small program using only SFML and TGUI. It uses around 30-80 mb of RAM, it’s less than 1 mb and uses 0.1% of my processor.

**Window class**The **Window** class encapsulates the main application window functionality using SFML and TGUI libraries. It handles the creation and management of the main window, GUI elements, and rendering loop.

## Constructors and Destructor

### Window()

Constructs a **Window** object.

* Initializes the window to 80% of the desktop resolution.
* Sets up the TGUI **Gui** object and binds it to the SFML window.
* Loads and sets the window icon from an image file.
* Calculates the canvas size and initializes the **Canvas** object.
* Initializes the **UIBar** object which contains the GUI elements for the application.
* Ensures proper memory management by deleting allocated resources upon object destruction.

### ~Window()

Destructor for the **Window** class.

* Safely releases the dynamically allocated **Canvas**, **Gui**, and **UIBar** objects.

## Public Methods

### void run()

Begins the main event and rendering loop of the application window.

* Handles window events such as close and resize.
* Enforces minimum size constraints on window resize.
* Updates the view to the new size of the window when resized.
* Processes mouse events to determine interaction with the canvas or UI bar.
* Clears the window and draws all elements, updating the window at a specified frame rate.

### sf::RenderWindow& getWindow()

Provides access to the underlying **sf::RenderWindow** object.

* Returns a reference to the **sf::RenderWindow** object used by the **Window** class.

**Canvas**

The **Canvas** class is a part of the PaintX application and it manages the drawing area where users can create and manipulate graphical objects. It handles drawing operations, input events related to drawing, and the management of graphical objects.

## Constructors and Destructor

### Canvas(int width, int height)

Constructs a **Canvas** object with the specified width and height.

* Initializes a **sf::RenderTexture** to the given dimensions.
* Creates a **sf::Sprite** that will display the content of the render texture.
* Clears the render texture with a white color to start with a blank canvas.

### ~Canvas()

Destructor for the **Canvas** class.

* Safely releases the **sf::Sprite** and **sf::RenderTexture** resources.

## Public Methods

### void handleEvent(const sf::Event& event, sf::RenderWindow& window)

Handles input events for the canvas, such as mouse clicks and movements.

* Responds to mouse button presses to initiate drawing or object manipulation.
* Responds to mouse button release to finalize drawing or object manipulation.
* Responds to mouse movements to update drawing or object manipulation.

### void draw(sf::RenderWindow& window)

Renders the canvas and all contained graphical objects onto the provided window.

* Draws the canvas sprite.
* Iterates through and draws all objects managed by the **Controller**.
* Draws the currently active object, if any.

### void setPosition(float x, float y)

Sets the position of the canvas within the window.

* Updates the position of the canvas sprite.

### sf::Vector2f getPosition() const

Returns the current position of the canvas within the window.

* Retrieves the position of the canvas sprite.

### int getWidth() const

Returns the width of the canvas.

* Provides the width as specified during construction.

### int getHeight() const

Returns the height of the canvas.

* Provides the height as specified during construction.

**UIBar**The **UIBar** class represents a user interface (UI) bar in the PaintX application. It is responsible for aggregating and managing different UI components that provide functionality such as shape selection, color picking, line thickness adjustment, tool selection, and file operations.

## Constructors and Destructor

### UIBar(int width, int height, tgui::Gui& gui)

Constructs a **UIBar** object.

* Creates and initializes a render texture to the specified width and height.
* Attaches a sprite to the render texture for displaying the UI bar.
* Sets the UI bar's background color to a dark grey.
* Positions the UI bar at the top of the window (0, 0).
* Initializes various selectors (**ShapeSelector**, **ColorSelector**, **LineSelector**, **ToolSelector**) and **FileTool**, passing in the **tgui::Gui** reference and appropriate widths for positioning.

### ~UIBar()

Destructor for the **UIBar** class.

* Releases the dynamically allocated **sf::RenderTexture** and **sf::Sprite** objects.

## Public Methods

### void draw(sf::RenderWindow& window)

Renders the UI bar and its components.

* Calls the **display** method of **renderTexture** to update the texture.
* Draws the UI bar sprite onto the window.
* Invokes the draw methods of each UI component (**ShapeSelector**, **ColorSelector**, **LineSelector**, **ToolSelector**, and **FileTool**).

### void handleEvent(sf::Event& event)

Handles user input events related to the UI bar.

* Currently, this method is empty and can be implemented to respond to specific events affecting the UI bar.

### int getWidth()

Returns the width of the UI bar.

* Retrieves and returns the width of the **renderTexture**.

### int getHeight()

Returns the height of the UI bar.

* Retrieves and returns the height of the **renderTexture**.

### sf::Vector2f getPosition()

Returns the position of the UI bar.

* Retrieves and returns the position of the UI bar sprite.

**Controller**The **Controller** class serves as a central hub for managing the state and interactions within the PaintX application. It tracks the current drawing tool, color, line size, fill status, and holds the collection of drawn objects.

## Static Members

* **Shapes shape**: Holds the current shape to be drawn.
* **sf::Color color**: Holds the current color selected for drawing.
* **std::vector<Object\*> objects**: Stores all the objects that have been drawn.
* **Object\* currentObject**: Points to the object currently being drawn or manipulated.
* **LineSize lineSize**: Specifies the current line size for drawing.
* **bool isFilled**: Indicates whether shapes should be drawn filled or not.
* **Tool instrument**: Represents the current tool selected (e.g., pen, eraser, selector).

## Public Methods

### void setShape(Shapes shape)

Sets the current shape to be drawn.

* **shape**: The **Shapes** enum value representing the new shape to be drawn.

### Shapes getShape()

Returns the current shape to be drawn.

### void AddObject(Object\* object)

Adds a new object to the collection of drawn objects.

* **object**: A pointer to the **Object** to be added to the collection.

### const std::vector<Object\*> getObjects()

Returns a constant reference to the collection of drawn objects.

### void setCurrentObject(Object\* object)

Sets the current object being manipulated.

* **object**: A pointer to the **Object** that is currently being manipulated.

### Object\* getCurrentObject()

Returns the current object being manipulated.

### void setLineSize(LineSize lineSize)

Sets the current line size for drawing.

* **lineSize**: The **LineSize** enum value representing the new line size.

### sf::Color getColor()

Returns the current color selected for drawing.

### void setColor(sf::Color color)

Sets the current color for drawing.

* **color**: The **sf::Color** value representing the new color.

### void setIsFilled(bool isFilled)

Sets the fill status for shapes.

* **isFilled**: A boolean indicating whether shapes should be drawn filled.

### bool getIsFilled()

Returns the fill status for shapes.

### LineSize getLineSizeParameter()

Returns the current **LineSize** parameter.

### void setTool(Tool instrument)

Sets the current tool selected for interaction.

* **instrument**: The **Tool** enum value representing the new tool to be used.

### Tool getTool()

Returns the current tool selected for interaction.

### void RemoveObject(Object\* object)

Removes an object from the collection and deletes it from memory.

* **object**: A pointer to the **Object** to be removed.

### void ClearObjects()

Clears all objects from the collection and deletes them from memory.

### void setMovingObject(Object\* object)

Sets the current object being moved or resized.

* **object**: A pointer to the **Object** that is being moved or resized.

### Object\* getMovingObject()

Returns the current object being moved or resized.

### void ClearController()

Clears the controller state, including all objects and the current object.

### float getLineSize()

Returns the numerical value representing the current line size.

**ObjectFactory**

The **ObjectsFactory** is a static class designed to facilitate the creation of drawable objects based on the current settings specified in the **Controller**. It serves as a factory that abstracts the instantiation of various **Object** types like **Line**, **Rectangle**, **Ellipse**, and **Triangle**.

## Public Methods

### static Object\* createObject(const sf::Vector2f& start, const sf::Vector2f& end)

Creates and returns a pointer to a new **Object** instance based on the current shape selected in the **Controller**.

* **start**: A **sf::Vector2f** representing the starting point of the object.
* **end**: A **sf::Vector2f** representing the ending point of the object.

The method checks the current shape selected in the **Controller** and creates an object of the corresponding type. It sets the object's properties such as color, line size, and fill status based on the current settings in the **Controller**.

## Return Value

Returns a pointer to the newly created object, or **nullptr** if the shape is not recognized or no shape is selected.

**Saver**The **Saver** class is responsible for saving and loading the state of drawn objects to and from a file. It interacts with the **Controller** class to access the current set of objects and uses file I/O to persist their properties.

## Public Methods

### void Save(const std::string& filename)

Saves the current set of drawable objects to the specified file.

* **filename**: A string representing the path to the file where the objects will be saved.

The method opens the file, erasing its current contents (**std::ios::trunc**), and iterates over the objects managed by the **Controller**. It writes the type, start point, end point, color, line width, and fill status of each object to the file in a human-readable format.

### void Open(const std::string& filename)

Loads drawable objects from the specified file and adds them to the **Controller**'s management.

* **filename**: A string representing the path to the file from which to load the objects.

The method reads the file line by line, reconstructing each object's properties and creating a new instance of the corresponding drawable object. It then adds the new object to the **Controller**.

**Ellipse**The **Ellipse** class is a drawable object representing an ellipse. It extends from the base **Object** class and is designed to be used within a graphical application that utilizes SFML for rendering. The **Ellipse** class provides functionality to draw both filled and outlined ellipses, check if a point is inside the ellipse, and retrieve the type of shape.

## Public Methods

### void draw(sf::RenderWindow& window)

Draws the ellipse on the provided SFML **RenderWindow**.

* **window**: The SFML **RenderWindow** where the ellipse will be drawn.

The method calculates the size and center of the ellipse based on the current window size. It then creates a set of vertices that define the shape of the ellipse with the specified number of points for smoothness. If **isFilled** is **true**, it draws a filled ellipse using a **sf::ConvexShape**. If **isFilled** is **false**, it draws an outline of the ellipse using a series of quadrilaterals (quads) constructed from the vertices.

### bool isInside(sf::Vector2f point, sf::RenderWindow& window)

Determines if a given point is inside the ellipse.

* **point**: The point to check, specified in window coordinates.
* **window**: The SFML **RenderWindow** used for context in calculations.

The method adjusts the point's coordinates relative to the ellipse's center and checks if the point lies within the bounds of the ellipse using its equation. For outlined ellipses, it checks if the point is between the boundaries of the outer and inner ellipses.

### std::string getType()

Returns the type of the shape as a string.

* This method always returns **"Ellipse"**.

## Drawing and Calculation Explanation

### Filled Ellipse Drawing

When **isFilled** is **true**, the method draws a filled ellipse by creating a **sf::ConvexShape** with a set number of points, forming a smooth circular shape. Each point on this shape is calculated using trigonometric functions (**cos** and **sin**) to determine its position on the circumference based on the calculated radii **a** and **b**.

### Outlined Ellipse Drawing

For an outlined ellipse, the method calculates two sets of vertices: one for the outer edge and one for the inner edge of the outline. These vertices form quads that visually create the outline with the specified width (**realWidth**). The outline is constructed by connecting each consecutive pair of outer and inner vertices, forming a series of narrow shapes around the ellipse's perimeter.

### Point-Inside Calculation

The **isInside** function calculates whether a given point is inside the filled ellipse or near the boundary of the outlined ellipse. For a filled ellipse, the standard ellipse equation is used with the adjusted point to determine if it lies within the ellipse. For an outlined ellipse, the method checks if the point is within the outer ellipse and outside the inner ellipse, creating a band representing the outline width.

**Rectangle**The **Rectangle** class is a concrete implementation of the **Object** interface for drawing rectangles on the PaintX application canvas.

## Member Functions

### void draw(sf::RenderWindow& window)

Draws a rectangle on the given **sf::RenderWindow**.

* **Parameters**: **window** - the **sf::RenderWindow** object on which the rectangle will be drawn.
* **Functionality**:
  + For a filled rectangle (**isFilled** is **true**), it creates an **sf::RectangleShape** with the size calculated from **start** and **end**, sets the fill color, and draws the filled rectangle onto the window.
  + For an unfilled rectangle (**isFilled** is **false**), it calculates the vertices for the four sides of the rectangle's outline and uses **sf::Quads** to draw a thick outline by overlapping the extended sides at the corners.

### bool isInside(sf::Vector2f point, sf::RenderWindow& window)

Determines whether the given point is inside the rectangle.

* **Parameters**:
  + **point** - the point to check, in window coordinates.
  + **window** - the **sf::RenderWindow** object used for reference dimensions.
* **Returns**: **true** if the point is inside the rectangle, **false** otherwise.
* **Functionality**:
  + For a filled rectangle, it checks whether the point's coordinates fall within the rectangle's boundaries.
  + For an unfilled rectangle, it creates **sf::FloatRect** objects representing the area of each side including the line width and checks if the point falls within any of these areas.

### std::string getType()

Returns the type of the object, in this case, "Rectangle".

* **Returns**: A string representing the type of the drawable object.

**Triangle**

The **Triangle** class is a specific implementation of the **Object** interface tailored for drawing triangles on the PaintX application canvas.

## Member Functions

### void draw(sf::RenderWindow& window)

Draws a triangle on the provided **sf::RenderWindow**.

* **Parameters**: **window** - the **sf::RenderWindow** object on which the triangle will be drawn.
* **Functionality**:
  + For a filled triangle (**isFilled** is **true**), constructs an **sf::ConvexShape** with three points and draws it onto the window with the specified **color**.
  + For an unfilled triangle (**isFilled** is **false**), calculates the lines that form the edges of the triangle and draws thick lines to represent the triangle's outline using the **drawThickLine** helper function.

### bool isInside(sf::Vector2f point, sf::RenderWindow& window)

Determines whether the specified point is inside the triangle.

* **Parameters**:
  + **point** - the point to check, in window coordinates.
  + **window** - the **sf::RenderWindow** used for scaling reference.
* **Returns**: **true** if the point is inside the triangle, **false** otherwise.
* **Functionality**:
  + Uses barycentric coordinate conditions to determine if the point lies within the filled area of the triangle.
  + For an unfilled triangle, checks if the point is close to any of the triangle's edges within the set line width.

### std::string getType()

Returns the type of the drawable object, in this case, "Triangle".

* **Returns**: A string literal "Triangle".

**Line**  
The **Line** class is a specific implementation of the **Object** interface intended for drawing lines on the PaintX application canvas.

## Member Functions

### void draw(sf::RenderWindow& window)

Draws a line on the given **sf::RenderWindow**.

* **Parameters**: **window** - the **sf::RenderWindow** object on which the line will be drawn.
* **Functionality**:
  + Calculates the actual start and end positions of the line based on the current window size.
  + Determines the direction and perpendicular vectors from the start to the end of the line.
  + Creates a thick line by calculating the vertices of a rectangle that represents the line with the specified width.
  + Sets the color for each vertex and uses **sf::TriangleStrip** to draw the thick line on the window.

### bool isInside(sf::Vector2f point, sf::RenderWindow& window)

Determines whether the specified point is "inside" the line (within a certain proximity to the line).

* **Parameters**:
  + **point** - the point to check, in window coordinates.
  + **window** - the **sf::RenderWindow** used for reference dimensions.
* **Returns**: **true** if the point is within the thickness of the line, **false** otherwise.
* **Functionality**:
  + Computes whether the point lies within the width of the line by projecting it onto the line segment and checking the distance from the point to the line against half of the line's width.

### std::string getType()

Returns the type of the drawable object, in this case, "Line".

* **Returns**: A string literal "Line".

**Object**The **Object** class is an abstract base class designed to represent a drawable object within the PaintX application. It provides the foundational attributes and behaviors from which specific drawable objects like **Rectangle**, **Triangle**, etc., are derived.

## Member Variables

* **sf::Vector2f start**: The starting point of the object, represented as a percentage of the screen dimensions.
* **sf::Vector2f end**: The ending point of the object, also represented as a percentage of the screen dimensions.
* **sf::Color color**: The color of the object.
* **bool isFilled**: A boolean indicating whether the object is filled with color.
* **float width**: The line width of the object's outline.
* **bool isSelected**: A boolean indicating whether the object is currently selected.

## Constructor

### Object(sf::Vector2f start, sf::Vector2f end, sf::Color color, float width, bool isFilled)

Constructs an **Object** with specified start and end points, color, line width, and fill status.

* **start**: The starting point of the object, in screen percentage coordinates.
* **end**: The ending point of the object, in screen percentage coordinates.
* **color**: The color of the object.
* **width**: The width of the object's outline.
* **isFilled**: Indicates whether the object's shape is filled with color.

## Pure Virtual Functions

These functions must be implemented by derived classes:

### void draw(sf::RenderWindow& window)

Draws the object onto the given render window.

* **Parameters**: **window** - the **sf::RenderWindow** object on which the object will be drawn.

### bool isInside(sf::Vector2f point, sf::RenderWindow& window)

Determines if the provided point is inside the object's area.

* **Parameters**:
  + **point** - the point to check, in window coordinates.
  + **window** - the **sf::RenderWindow** object used for reference dimensions.

### std::string getType()

Returns a string representing the type of the drawable object.

## Public Member Functions

### sf::Color getColor()

Returns the color of the object.

### sf::Vector2f getStart()

Returns the starting point of the object.

### sf::Vector2f getEnd()

Returns the ending point of the object.

### float getLineWidth()

Returns the line width of the object's outline.

### bool getFilled()

Returns the fill status of the object.

### void setEnd(sf::Vector2f end)

Updates the ending point of the object.

### void resize(sf::Vector2f offset)

Alters the size of the object based on the provided offset.

### void move(sf::Vector2f offset)

Moves the object by the specified offset.

# **Selector Classes**

The selector classes (**ToolSelector**, **ColorSelector**, **LineSelector**, **ShapeSelector**, **FileTool**) provide user interface components for the PaintX application, allowing users to choose drawing tools, colors, line sizes, shapes, and file operations through a graphical interface. Each selector class is tailored to a specific type of selection but shares a common structure and functionality centered around TGUI buttons and textures.

## Common Structure

Each selector class typically contains:

* Private member variables for TGUI buttons and textures corresponding to the options available within the selector (e.g., different tools in **ToolSelector**, colors in **ColorSelector**).
* An **int** or **float** to store the position or height of the selector UI component, ensuring proper layout within the application window.
* A method to set common parameters for the buttons, such as size, position, texture, and callbacks, to reduce code duplication.

## Constructor

### Selector(tgui::Gui& gui, int height, int position)

Initializes a selector with references to the main GUI interface, its height, and its starting position.

* **gui**: Reference to the TGUI **Gui** object to which the selector's buttons will be added.
* **height**: The height of the selector, which may dictate the size of the buttons.
* **position**: The starting position of the selector, used to place it correctly within the UI layout.

## Public Methods

### void SetButtonParameters(int index, tgui::Button::Ptr button, tgui::Texture& texture)

Configures a button with the specified parameters, including setting its texture, position based on the **index**, and common styling.

* **index**: The index of the button, used to calculate its position within the selector.
* **button**: The TGUI button to configure.
* **texture**: The texture to apply to the button.

### float GetWidth()

Returns the total width of the selector, calculated based on the number of buttons and their sizes.

### void draw()

Renders the selector and its components. In the context of TGUI, this might be an implicit action handled by TGUI's drawing routines rather than a method directly called by the application.

## Common Functionality

* **Button Initialization**: Each selector initializes its set of buttons and assigns them textures corresponding to the options they represent (e.g., eraser, resize, move tools).
* **Event Handling**: Selectors listen for button press events to trigger actions, like changing the current drawing tool or color.
* **GUI Integration**: Selectors add their buttons to the main **tgui::Gui** object, ensuring they are rendered and can interact with user input.