PalettePro Report/Documentation

Introduction

The purpose of this report is to explain the development of the PalettePro paint program, a 2D raster graphics editor, in the style of Microsoft Paint, MacPaint and GIMP. The project brief was that the program be written in C++ and use the SDL (Simple Directmedia Layer) 2 Library. The minimum requirements were that the program should allow the user to load and save selected images and the user interaction be conducted via a graphical interface. The required image editing operations were for the user to be able to select from a number of common pixel editing tools such as paint brush, eraser and a flood fill as well as being able to draw a number of simple geometric shapes. It is also necessary that the user can select different drawing colours.

Development

The initial development of the program was helped by the used of Gigi lab's SDL2 tutorials (D'Agostino, 2021) and focused on simple tasks such as generating an empty window then moved on to loading a specified image which could then be transferred to surface then to a texture before being copied to the renderer. The next early task was to create a texture from scratch (using SDL_CreateTexture) then pass a pixel array to this texture. This then allowed individual pixel colours to be changed via x- and y-coordinates.

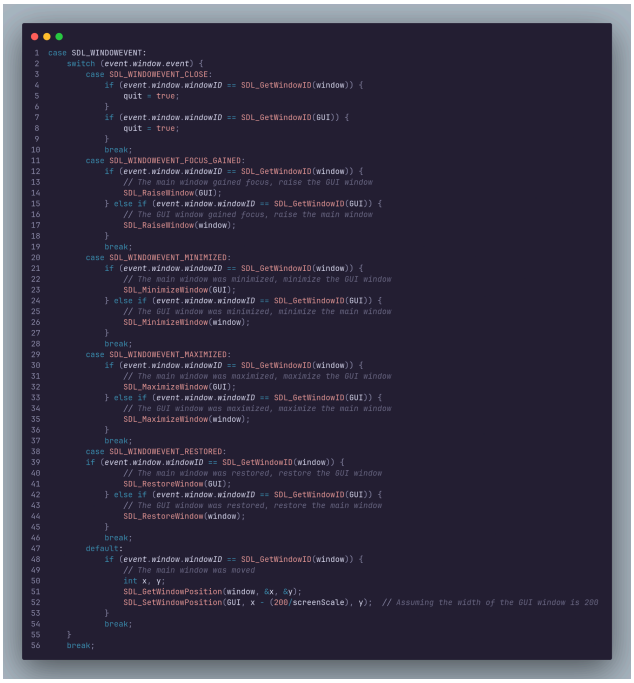
Once this basic understanding of images, surfaces, textures, pixel arrays and renderers had been gained, it was possible to start implementing the handling of mouse and keyboard events. A simple 'brush' painting algorithm was written using mouse button and motion events and this was extended using hotkeys to change stroke colour. Drawing of simple rectangles via mouse events followed and this also allowed for the inclusion of a rectangular fill option and the alteration of stroke width (again, via hotkeys). At this point, a decision was made to include a fill option on any geometric shapes as this would be more efficient than always relying on the inefficient flood-fill algorithm.

The development then progressed to the drawing functions that required more complex algorithms, beginning with Bresenham's straight line algorithm. Initially, an algorithm was written for the basic case where the line went from left to right and had a gradient of between 0 and 1. This was then extended to account for the other three low gradient cases in the same algorithm. An equivalent algorithm was also written for cases where the gradient was greater than 1 and an if statement was used to choose the appropriate algorithm. Whilst the line drawing tool worked perfectly and it was very useful in understanding the issues that arose in the non-basic cases, it was decided that Dr Anderson's much neater implementation [cite] would be used in the final program. A midpoint ellipse algorithm was written with the help of a javaTpoint (javaTpoint) description of the algorithm. This was chosen over Besenham's circle algorithm as it allowed both circle and ellipse drawing and is more accurate. As with the rectangle drawing

algorithm, this was implemented with the option for colour fill. At this point, it became clear that altering the stroke width on these shapes might be difficult. Initially this was done by calling the ellipse algorithm multiple times via a for loop but this resulted in shape boundaries that were not completely solid and looked unsatisfactory. An alternative approach was then adopted by developing an algorithm that drew a filled circle of diameter equal to the stroke width and centre at the current pixel point. This could then be used in the line, circle and ellipse algorithms.

All of the above functionality was added together into the main program which, as a result, became incredibly unreadable and contained several repetitions of chunks of code. At this point, the program was refactored using functions for all the drawing tools which were called via a single ToolAction handling function. This meant that the main program and its event handler became much easier to read and allowed easier maintenance and bug fixing within the tools.

Up to this point, the user interaction was done through the use of hotkeys rather than a GUI. The initial plan for the GUI implementation was to generate a separate GUI window that was attached to the side of the main drawing window. This involved a lot of work around SDL's multiple window support and difficulties ensuring that window event actions were mirrored between the two windows so that they appeared to be linked. Figure 1 shows the code that was added to the event handler to deal with this.



```
case SDL_WINDOWEVENT:
    switch (event.window.event) {
        case SDL_WINDOWEVENT_CLOSE:
            if (event.window.windowID == SDL_GetWindowID(window)) {
                quit = true;
            }
            if (event.window.windowID == SDL_GetWindowID(GUI)) {
                quit = true;
            }
            break;
        case SDL_WINDOWEVENT_FOCUS_GAINED:
            if (event.window.windowID == SDL_GetWindowID(window)) {
                // The main window gained focus, raise the GUI window
                SDL_RaiseWindow(GUI);
            } else if (event.window.windowID == SDL_GetWindowID(GUI)) {
                // The GUI window gained focus, raise the main window
                SDL_RaiseWindow(window);
            }
            break;
        case SDL_WINDOWEVENT_MINIMIZED:
            if (event.window.windowID == SDL_GetWindowID(window)) {
                // The main window was minimized, minimize the GUI window
                SDL_MinimizeWindow(GUI);
            } else if (event.window.windowID == SDL_GetWindowID(GUI)) {
                // The GUI window was minimized, minimize the main window
                SDL_MinimizeWindow(window);
            }
            break;
        case SDL_WINDOWEVENT_MAXIMIZED:
            if (event.window.windowID == SDL_GetWindowID(window)) {
                // The main window was maximized, maximize the GUI window
                SDL_MaximizeWindow(GUI);
            } else if (event.window.windowID == SDL_GetWindowID(GUI)) {
                // The GUI window was maximized, maximize the main window
                SDL_MaximizeWindow(window);
            }
            break;
        case SDL_WINDOWEVENT_RESTORED:
        if (event.window.windowID == SDL_GetWindowID(window)) {
                // The main window was restored, restore the GUI window
                SDL_RestoreWindow(GUI);
            } else if (event.window.windowID == SDL_GetWindowID(GUI)) {
                // The GUI window was restored, restore the main window
                SDL_RestoreWindow(window);
            }
            break;
        default:
            if (event.window.windowID == SDL_GetWindowID(window)) {
                // The main window was moved
                int x, y;
                SDL_GetWindowPosition(window, &x, &y);
                SDL_SetWindowPosition(GUI, x - (200/screenScale), y);  // Assuming the width of the GUI window is 200
            }
            break;
    }
    break;
```

Figure 1. Code excerpt showing event handler for multiple window GUI

Whilst the approach worked well on Mac and UNIX systems, problems arose when testing on a Windows set-up with the mouse event detection not being fully functional (less responsive) on the canvas window. It was decided that an alternative approach should be used in the final set-up for the sake of simplicity and being more robust across different platforms. A new GUI panel image was designed in Photoshop and imported directly into the left side of the drawing

window. A simple drawing flag was created to ensure that the drawing tools could be selected when the mouse was in the GUI section and the drawing tools could only be used when the mouse was in the drawing section. Interaction with the GUI panel was initially implemented using a long series of nested if statements inside the event handler. This resulted in the event handler becoming messy again and so an approach using a struct named Button was used to identify which GUI icon had been selected via the mouse position and then take the appropriate action (change tool, stroke width, colour etc).

Up to this point for development purposes, when the program was launched, the user was prompted with a (Y/N) option to load an image. The program would then stop responding until it received a user response. If the user chose Y, they were then prompted for the image file path. If the image loaded correctly, the blank canvas stage, where the pixel array was memset to white, would be skipped. Instead, the image was stretched to fill the canvas. This approach had several issues: 1. Potential image distortion, 2. User interface was via terminal, 3. Images could only be loaded on program initiation and 4. All the logic required for this approach was inside the main function leaving it very difficult to read.

It was clear that a better image loading approach was required and that this should be done via the GUI rather than the terminal. After some research, it was decided that the best option would be to open a file dialogue once the GUI button was pressed. However, to my knowledge, this functionality does not exist in the SDL library alone and so it was decided that the easiest external library for file dialogues in C++ was the tinyfiledialogs library (Granick). This was used for both image loading and saving and will be discussed in more detail in the Implementation section along with the resolution of the image stretching issues.

This section discussed the overall development journey of the project. The next section will talk through the structure of the final program, the implementation of the algorithms and how it all comes together.

Implementation

Main program implementation

To understand the core of how the program flows, the overall program structure will first be discussed. After that, the supplementary functions will be discussed in more detail.

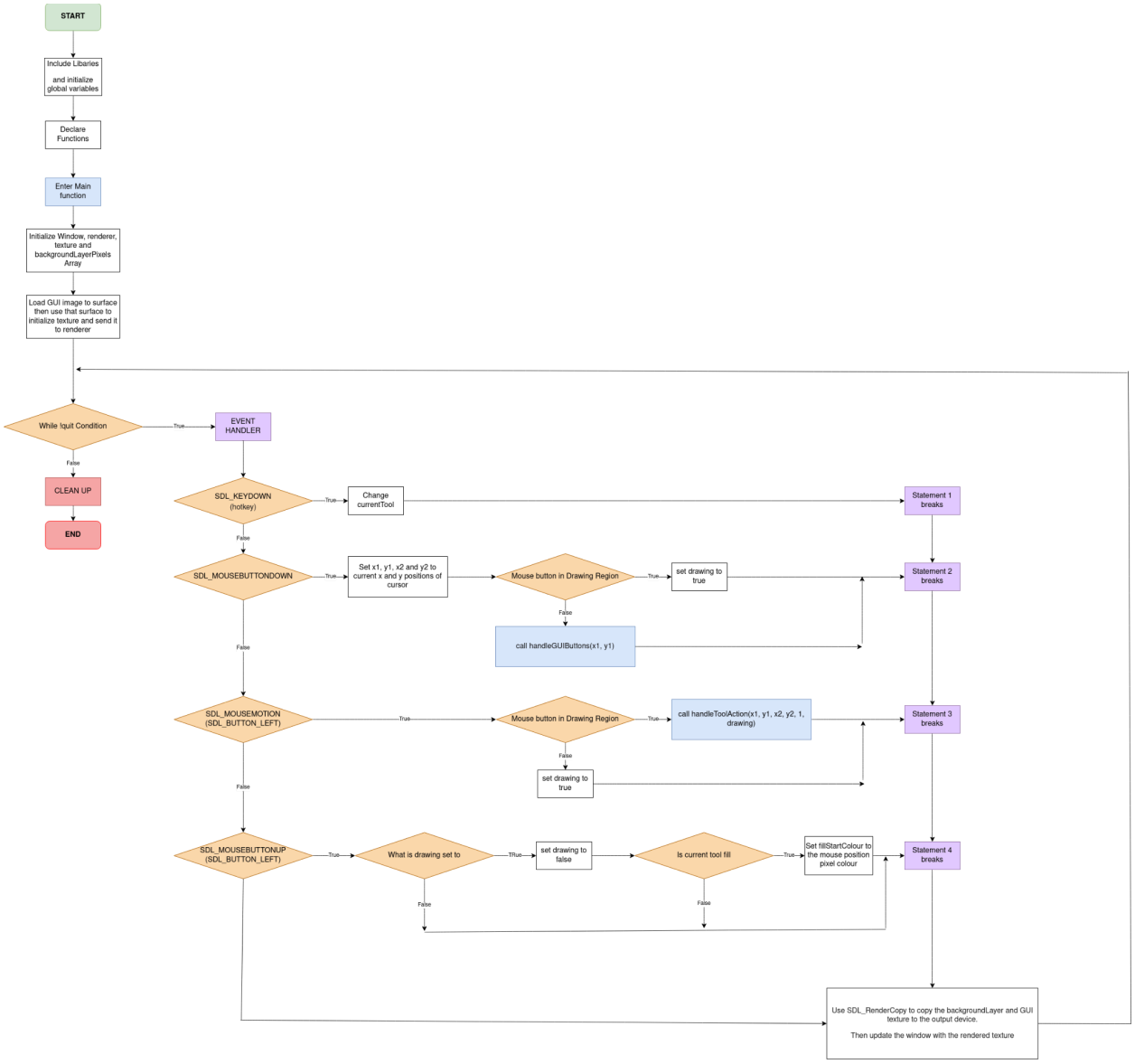


Figure 2. Flowchart showing PalettePro main flow (Full res image can be found in source file)

Above is a figure showing a flowchart for the PalettePro program. This is the main flow of the program and can be thought of as the foundations or core for how the program behaves.

The journey begins at the program start (green rectangle at top left of Figure 2). All required libraries are then included for use later. Next, a number of global variables are declared - these consist of Window/Renderer variables and Tool Selection/Settings variables. The number of global variables was kept as low as possible, reserving them for those required for core functionality across the software. This was to try to ensure that variables were kept to their required scope where possible. All functions are then declared in the next section of code and, as discussed above, will be investigated in detail later.

At this point, we have arrived at the main function. At the start of this function, a number of initialisations occur as can be seen in Figure 3.

```
int main(int argc, char **argv)
{
    // Init SDL with video sub-system
    SDL_Init(SDL_INIT_VIDEO);

    // Creating Main Window
    SDL_Window *window = SDL_CreateWindow("BRUSH SELECTED, STROKE WIDTH 1, STROKE COLOUR (255,255,255), FILL COLOUR (-1, -1, -1)",
        SDL_WINDOWPOS_UNDEFINED, 30, screenWidth, screenHeight, SDL_WINDOW_ALLOW_HIGHDPI);
    SDL_Renderer *renderer = SDL_CreateRenderer(window, -1, 0);
    SDL_Texture *backgroundLayer = SDL_CreateTexture(renderer, SDL_PIXELFORMAT_RGBA8888, SDL_TEXTUREACCESS_STREAMING, screenWidth, screenHeight);

    // Creating Blank Canvas
    backgroundLayerPixels = new Uint32[screenWidth * screenHeight];
    memset(backgroundLayerPixels, 255, screenWidth * screenHeight * sizeof(Uint32));
    SDL_UpdateTexture(backgroundLayer, NULL, backgroundLayerPixels, screenWidth * sizeof(Uint32));

    // Load GUI
    SDL_Surface *gui_image = IMG_Load("GUI.png");
    SDL_Texture *guiTexture = SDL_CreateTextureFromSurface(renderer, gui_image);
    SDL_FreeSurface(gui_image);
    SDL_Rect dstrect = { 0, 0, guiWidth, screenHeight };
    SDL_RenderCopy(renderer, guiTexture, NULL, &dstrect);

    //SDL_RenderPresent commits texture to video memory
    SDL_RenderPresent(renderer);
```

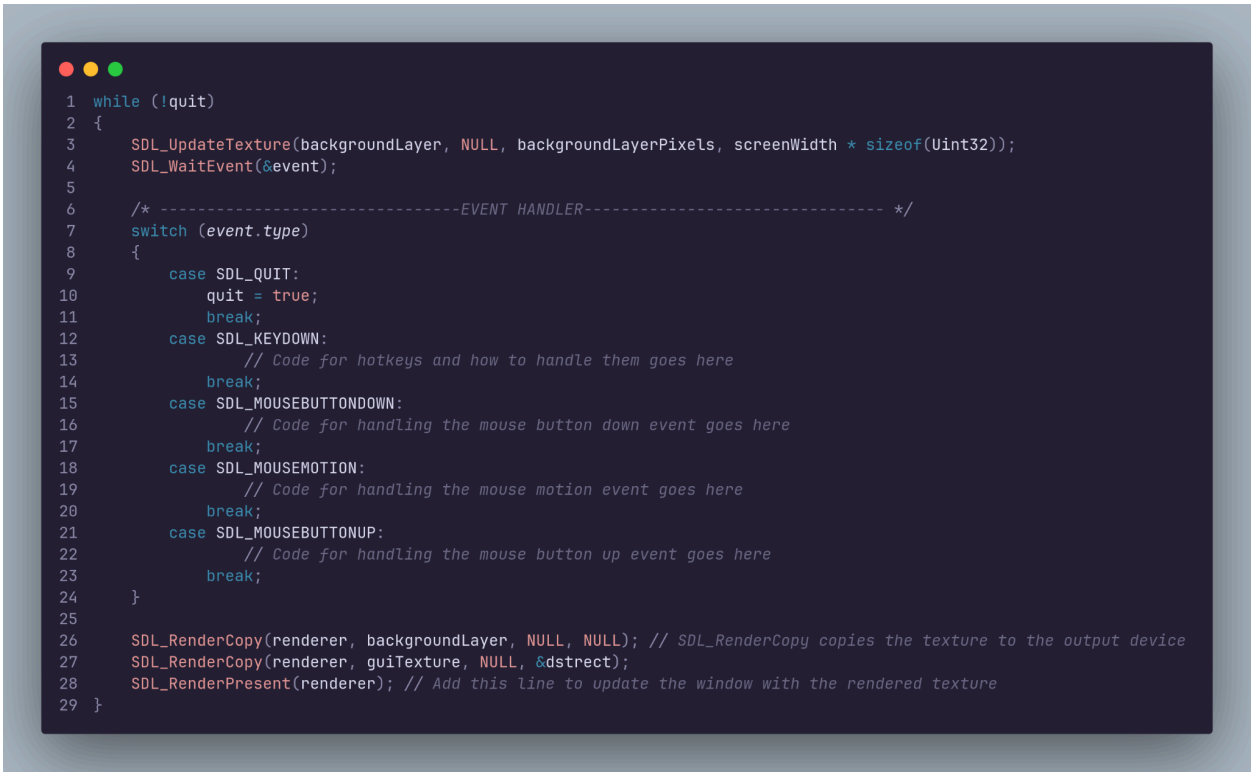
Figure 3. Initialisation section at start of the main function

This section begins with initialisation of the video subsystem using the previously-included SDL library.

The main window is then created in steps:

1. The window, renderer and texture for the whole window of the program are created.
2. A pixel array (backgroundLayerPixels) is created and all values are memset to 255 (to make the colour white). Then the backgroundLayer texture is updated with this pixel array resulting in a blank white canvas.
3. The GUI image is then loaded into a gui_image surface which is then used to create a new GUI texture. The now-redundant image surface is freed from memory. The GUI texture is then rendered to the left side of the main canvas with the desired size.
4. Finally, both textures are committed to video memory via the RenderPresent command.

After the window creation, the program enters the main loop.



```
while (!quit)
{
    SDL_UpdateTexture(backgroundLayer, NULL, backgroundLayerPixels, screenWidth * sizeof(Uint32));
    SDL_WaitEvent(&event);

    /* ------------------------------EVENT HANDLER------------------------------ */
    switch (event.type)
    {
        case SDL_QUIT:
            quit = true;
            break;
        case SDL_KEYDOWN:
                // Code for hotkeys and how to handle them goes here
            break;
        case SDL_MOUSEBUTTONDOWN:
                // Code for handling the mouse button down event goes here
            break;
        case SDL_MOUSEMOTION:
                // Code for handling the mouse motion event goes here
            break;
        case SDL_MOUSEBUTTONUP:
                // Code for handling the mouse button up event goes here
            break;
    }

    SDL_RenderCopy(renderer, backgroundLayer, NULL, NULL); // SDL_RenderCopy copies the texture to the output device
    SDL_RenderCopy(renderer, guiTexture, NULL, &dstrect);
    SDL_RenderPresent(renderer); // Add this line to update the window with the rendered texture
}
```

Figure 4. Main program loop with reduced version of event handler

Figure 4 shows the main program loop with a stripped back version (for readability purposes) of the event handler. This while loop can only be exited via the quit event being triggered.

Upon entering the while loop, the backgroundLayer texture is updated to reflect any changes to the backgroundLayerPixels array before using the SDL_WaitEvent function to wait for an event to occur (e.g. hotkey events and mouse-related events).

As can be seen, aside from the quit event, the event handlers job is to look for events in order to alter the code flow appropriately. To understand the non-stripped back version of the event handler please refer to the bottom right side of the flowchart in Figure 2 or the source code.

After handling all event cases the main loop moves onto rendering both backgroundLayer and GUI textures to the output window.

Implementation of particular functions

In this subsection, a brief description of some of the functions that are not based upon well-known documentation or algorithms will be given. The four functions that will be discussed here will be the Load and Save Image functions, the handleToolAction function and the handleGUIButtons function.

```
void handleToolAction(int x1, int y1, int x2, int y2, bool isMouseDown)
// Function to handle which tool to pass the variables based on the current tool selected
{
    switch (currentTool) {
        case BRUSH:
            drawBrush(x2, y2, isMouseDown);
            break;
        case ERASER:
            eraser(x2, y2, isMouseDown);
            break;
        case LINE:
            drawLine(x1, y1, x2, y2, isMouseDown);
            break;
        case RECTANGLE:
            drawRect(x1, y1, x2, y2, isMouseDown);
            break;
        case CIRCLE:
            drawEllipse(x1, y1, x2, y2, isMouseDown);
            break;
        case FILL:
            bucketFill(x1, y1, isMouseDown);
            break;
        default:
            break;
    }
}
```

Figure 5. HandleToolAction function

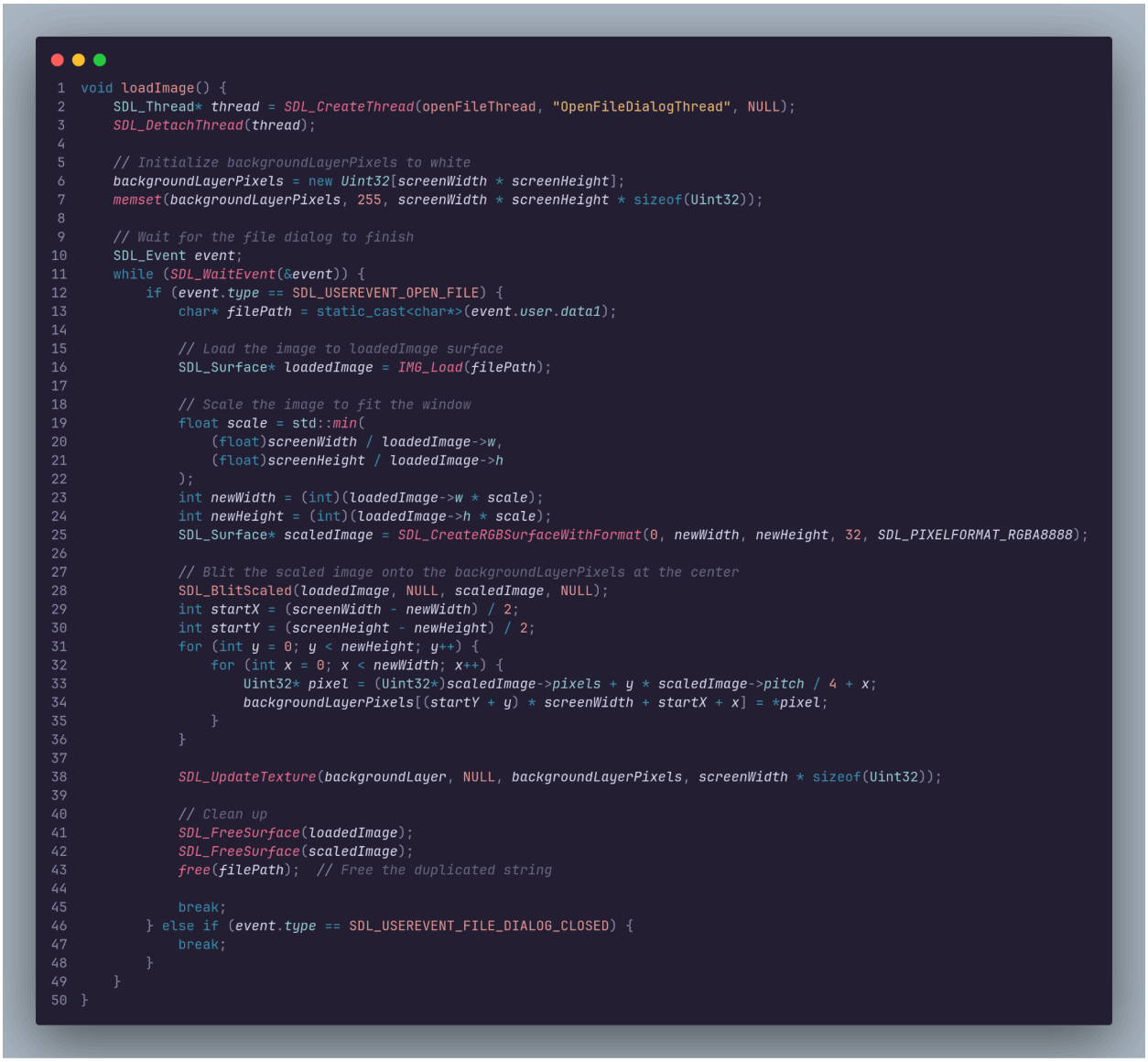
This is the first function that will be covered as it is one of the only two functions that get called inside the event handler of our program. Its only purpose is to make the code more readable by handling the switch case for where to send the variables x1, y1, x2, y2 and isMouseDown variables in a separate function instead of inside the event handler.

As can be seen in the Figure 5 above, different tool functions will only be passed whichever variables they need to work.

```
void handleGUIButtons(int x1, int y1)
// Function to handle the buttons on the GUI
{
    struct Button {
        int x1, y1, x2, y2;
        std::function<void()> onClick;

        bool contains(int x, int y) const {return x >= x1 && x < x2 && y >= y1 && y < y2;}
    };
    std::vector<Button> buttons = {
        // Tool selection buttons
        {0, 0, 50, 50, [](){ currentTool = BRUSH; }},
        {50, 0, 100, 50, [](){ currentTool = ERASER; }},
        {0, 50, 50, 100, [](){ currentTool = FILL; }},
        {50, 50, 100, 100, [](){ currentTool = LINE; }},
        {0, 100, 50, 150, [](){ currentTool = CIRCLE; }},
        {50, 100, 100, 150, [](){ currentTool = RECTANGLE; }},

        // Image save and load buttons
        {0, 170, 50, 210, [](){ saveImage(); }},
        {50, 170, 100, 210, [](){ loadImage(); }},

        // Stroke width buttons Y = 210 - 230
        {0, 230, 20, 250, [](){ strokeWidth = 1; }},
        {20, 230, 40, 250, [](){ strokeWidth = 3; }},
        {40, 230, 60, 250, [](){ strokeWidth = 5; }},
        {60, 230, 80, 250, [](){ strokeWidth = 7; }},
        {80, 230, 100, 250, [](){ strokeWidth = 9; }},

        // --- Stroke colour buttons ---

        // --- Fill colour buttons ---
    };

    for (const Button& button : buttons) {
        if (button.contains(x1, y1)) {
            button.onClick();
            break;
        }
    }
}
```

Figure 6. handleGUIButtons function (slightly simplified for readability)

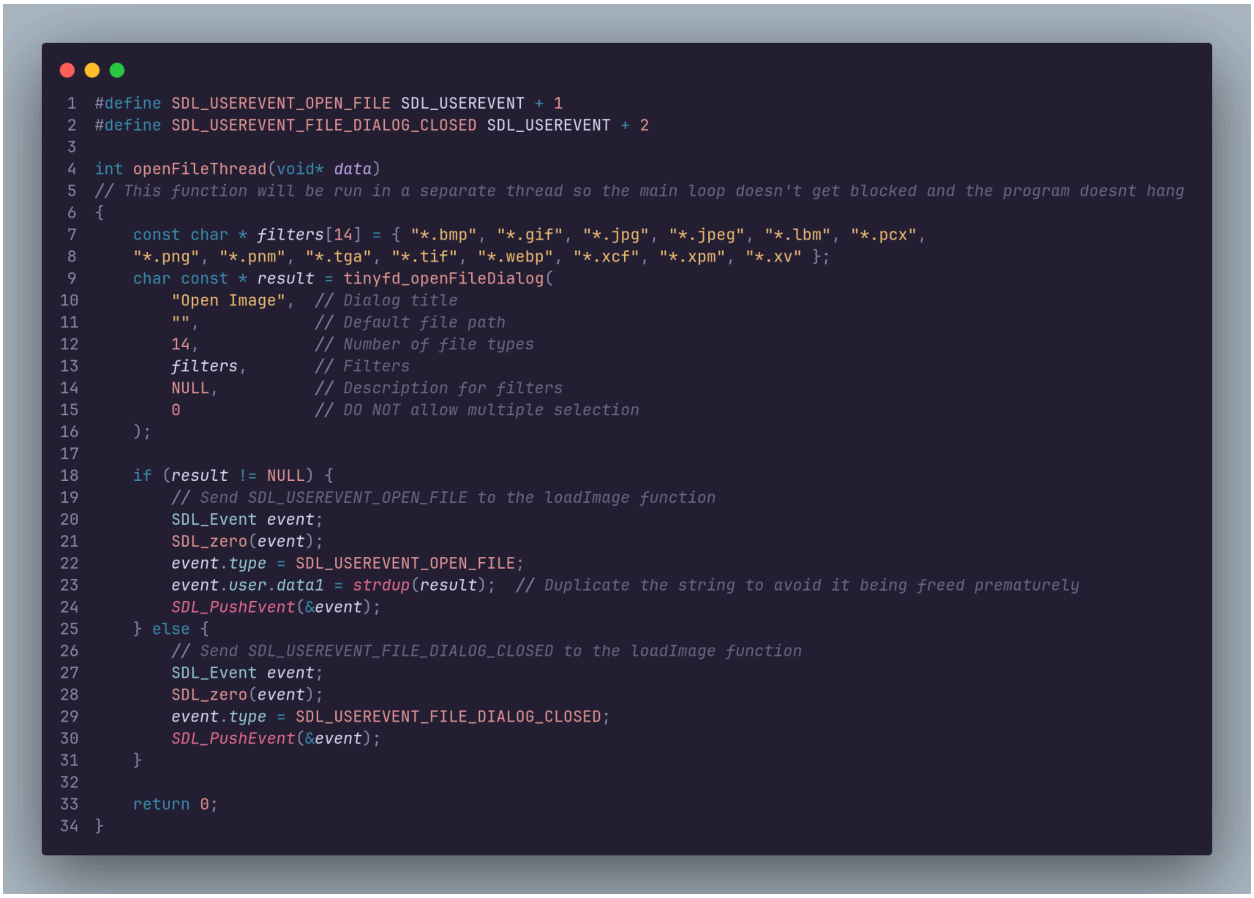
The purpose of the handleGUIButtons function is also to increase the event handler's readability by removing the large button struct. Inside this function, a struct for the buttons that sit 'above' the GUI icons is created. The struct has the button coordinates, a function onClick that is called when a button is clicked and a contains function that returns a boolean value depending upon whether the mouse coordinates are within the button coordinates. A button vector is created with all of the button coordinates and associated onClick function. The function ends by iterating over all of the buttons to check which, if any, contains the mouse coordinates and executes the appropriate onClick function.



```
void loadImage() {
    SDL_Thread* thread = SDL_CreateThread(openFileThread, "OpenFileDialogThread", NULL);
    SDL_DetachThread(thread);

    // Initialize backgroundLayerPixels to white
    backgroundLayerPixels = new Uint32[screenWidth * screenHeight];
    memset(backgroundLayerPixels, 255, screenWidth * screenHeight * sizeof(Uint32));

    // Wait for the file dialog to finish
    SDL_Event event;
    while (SDL_WaitEvent(&event)) {
        if (event.type == SDL_USEREVENT_OPEN_FILE) {
            char* filePath = static_cast<char*>(event.user.data1);

            // Load the image to loadedImage surface
            SDL_Surface* loadedImage = IMG_Load(filePath);

            // Scale the image to fit the window
            float scale = std::min(
                (float)screenWidth / loadedImage->w,
                (float)screenHeight / loadedImage->h
            );
            int newWidth = (int)(loadedImage->w * scale);
            int newHeight = (int)(loadedImage->h * scale);
            SDL_Surface* scaledImage = SDL_CreateRGBSurfaceWithFormat(0, newWidth, newHeight, 32, SDL_PIXELFORMAT_RGBA8888);

            // Blit the scaled image onto the backgroundLayerPixels at the center
            SDL_BlitScaled(loadedImage, NULL, scaledImage, NULL);
            int startX = (screenWidth - newWidth) / 2;
            int startY = (screenHeight - newHeight) / 2;
            for (int y = 0; y < newHeight; y++) {
                for (int x = 0; x < newWidth; x++) {
                    Uint32* pixel = (Uint32*)scaledImage->pixels + y * scaledImage->pitch / 4 + x;
                    backgroundLayerPixels[(startY + y) * screenWidth + startX + x] = *pixel;
                }
            }

            SDL_UpdateTexture(backgroundLayer, NULL, backgroundLayerPixels, screenWidth * sizeof(Uint32));

            // Clean up
            SDL_FreeSurface(loadedImage);
            SDL_FreeSurface(scaledImage);
            free(filePath);  // Free the duplicated string

            break;
        } else if (event.type == SDL_USEREVENT_FILE_DIALOG_CLOSED) {
            break;
        }
    }
}
```

Figure 7. loadImage function (full)

Figure 7 above shows the loadImage function. When this function is called by clicking on the Load button in the GUI panel, the first thing that it does is calls the openFileThread function on a new thread using SDL_CreateThread. The reason this is done is so that the program does not hang whilst the file dialog is open. The SDL_DetachThread tells the computer that we no longer need this thread once the function is finished and to automatically release the thread back to the system. A pixel array (backgroundLayerPixels) is created and all values are memset to 255 (to make the colour white).



```c
#define SDL_USEREVENT_OPEN_FILE SDL_USEREVENT + 1
#define SDL_USEREVENT_FILE_DIALOG_CLOSED SDL_USEREVENT + 2

int openFileThread(void* data)
// This function will be run in a separate thread so the main loop doesn't get blocked and the program doesnt hang
{
    const char * filters[14] = { "*.bmp", "*.gif", "*.jpg", "*.jpeg", "*.lbm", "*.pcx",
    "*.png", "*.pnm", "*.tga", "*.tif", "*.webp", "*.xcf", "*.xpm", "*.xv" };
    char const * result = tinyfd_openFileDialog(
        "Open Image",  // Dialog title
        "",            // Default file path
        14,            // Number of file types
        filters,       // Filters
        NULL,          // Description for filters
        0              // DO NOT allow multiple selection
    );

    if (result != NULL) {
        // Send SDL_USEREVENT_OPEN_FILE to the loadImage function
        SDL_Event event;
        SDL_zero(event);
        event.type = SDL_USEREVENT_OPEN_FILE;
        event.user.data1 = strdup(result);  // Duplicate the string to avoid it being freed prematurely
        SDL_PushEvent(&event);
    } else {
        // Send SDL_USEREVENT_FILE_DIALOG_CLOSED to the loadImage function
        SDL_Event event;
        SDL_zero(event);
        event.type = SDL_USEREVENT_FILE_DIALOG_CLOSED;
        SDL_PushEvent(&event);
    }

    return 0;
}
```

Figure 8. openFileThread function (called at start of loadImage function)

The openFileThread function calls the function shown above in Figure 8. This is composed of 3 sections. The first section utilizes and calls the tinyfd_openFileDialog function included in the tiny file dialog library that was included at the start of the program. It is set up following the documentation found in the tiny file dialog github page: [REF] . A char variable named result is created for the storage of the image path that the user wants to open. If the result is a valid file, the event that was defined before this function, SDL_USEREVENT_OPEN_FILE, will be sent back to the loadImage function otherwise the event SDL_USEREVENT_DIALOG_CLOSED is returned.

Returning to Figure 7, when the wait event detects the SDL_USEREVENT_OPEN_FILE event, it retrieves the file path that is returned by the openFileThread function and assigns it to a char variable named filePath. The image is then loaded to the loadedImage surface using the IMG_Load function and is scaled to fit the screen dimensions based upon the ratio of screen to image width and heights and given the RGBA888 pixel format. This is then loaded into the scaledImage surface. The pixel values of the scaledImage surface are then copied to the central area of the backgroundPixelsLayer before the backgroundLayer texture is updated to reflect the pixel changes. Image surfaces and file paths are cleaned up at the end.

```
void saveImage() {
    SDL_Thread * thread = SDL_CreateThread(saveFileThread, "SaveFileDialogThread", NULL);
    SDL_DetachThread(thread);

    // Wait for the file dialog to finish
    SDL_Event event;
    while (SDL_WaitEvent(&event)) {
        if (event.type == SDL_USEREVENT_SAVE_FILE) {
            char* filePath = static_cast<char*>(event.user.data1);

            // Create an SDL_Surface from backgroundLayerPixels
            SDL_Surface* surface = SDL_CreateRGBSurfaceFrom(
                backgroundLayerPixels,  // Pixels
                screenWidth,            // Width
                screenHeight,           // Height
                32,                     // Depth
                screenWidth * 4,        // Pitch
                0xff000000,             // Rmask
                0x00ff0000,             // Gmask
                0x0000ff00,             // Bmask
                0x000000ff              // Amask
            );

            const char* extension = strrchr(filePath, '.');

            if (strcmp(extension, ".bmp") == 0) {
                SDL_SaveBMP(surface, filePath);
            } else if (strcmp(extension, ".jpg") == 0 || strcmp(extension, ".jpeg") == 0) {
                IMG_SaveJPG(surface, filePath, 100);
            } else if (strcmp(extension, ".png") == 0) {
                IMG_SavePNG(surface, filePath);
            } else {
                printf("Error: Unsupported file format.\n");
            }

            // Clean up
            SDL_FreeSurface(surface);
            free(filePath);  // Free the duplicated string
            break;
        } else if (event.type == SDL_USEREVENT_FILE_DIALOG_CLOSED) {
            break;
        }
    }
}
```

Figure 9. saveImage function (full)

Figure 9 above shows the saveImage function. This follows a very similar pattern to the loadImage function in that it creates a thread, calls the tinyfd_openFileDialog function to obtain a filepath. A surface is then created from the current values of backgroundLayerPixels and this surface is then saved in the requested .bmp, .jpg or .img format. The surface and filepath are then freed.

Results

This section briefly explains some typical outputs that arise from the program.

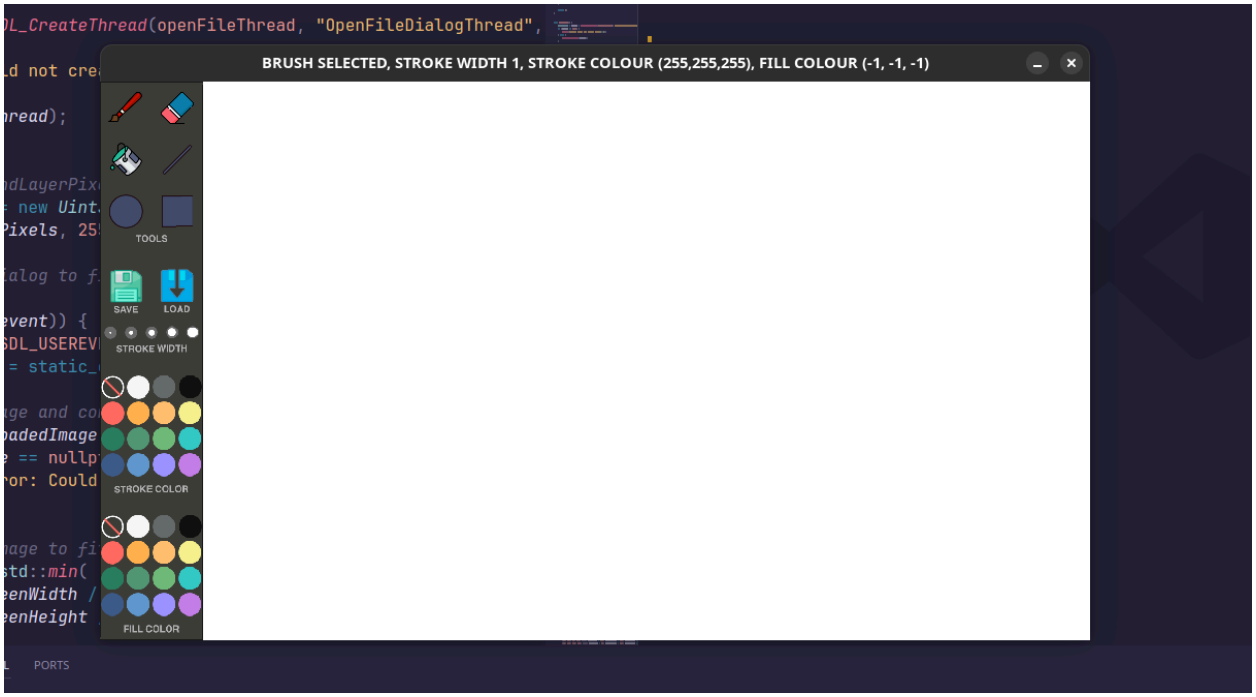


Figure 10. PalettePro on initialisation

Figure 10 above shows how the program appears when it is run automatically from the executable. The full white canvas background layer appears on launch and, by default, the brush tool is selected, the stroke colour is set to black, the stroke width is set to 1 and the fill color turned off (signified by the -1,-1,-1 RGB values). The user is able to see the tool information displayed in the window title. The user simply clicks on any of the drawing tools with desired stroke width, stroke colour and fill colour before using the mouse events to draw on the canvas.

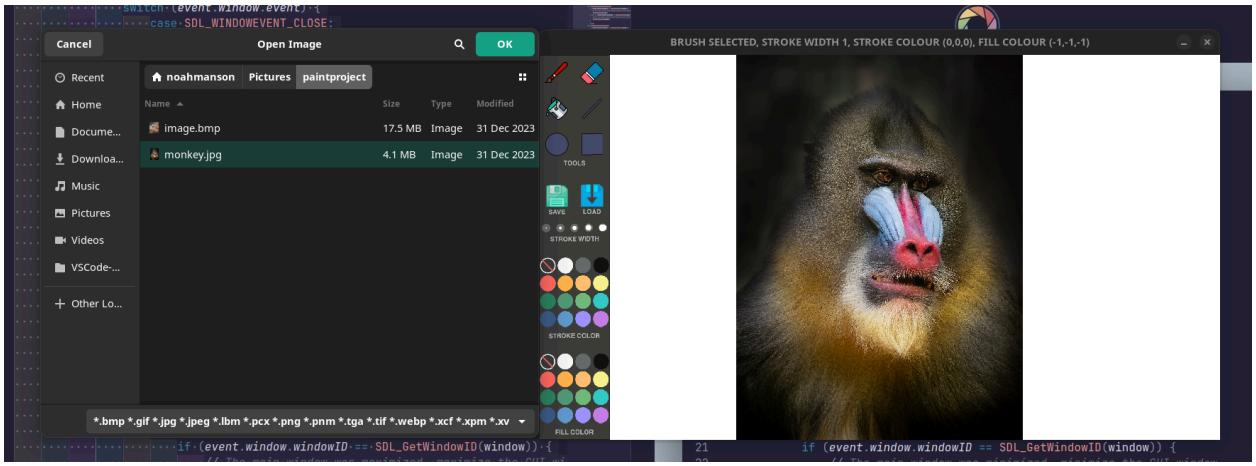


Figure 11. File dialogue to allow user to import images

If the user wants to import an image to use as a background layer, they can click the load button on the GUI. This will launch a file dialogue (as seen on left in Figure 11 above) for them to navigate to the image they want to import. All 14 of SDL2 image's supported formats can be loaded.

As shown on the right side of Figure 11, the image will automatically be scaled to fit the window and placed on top of a blank white canvas.

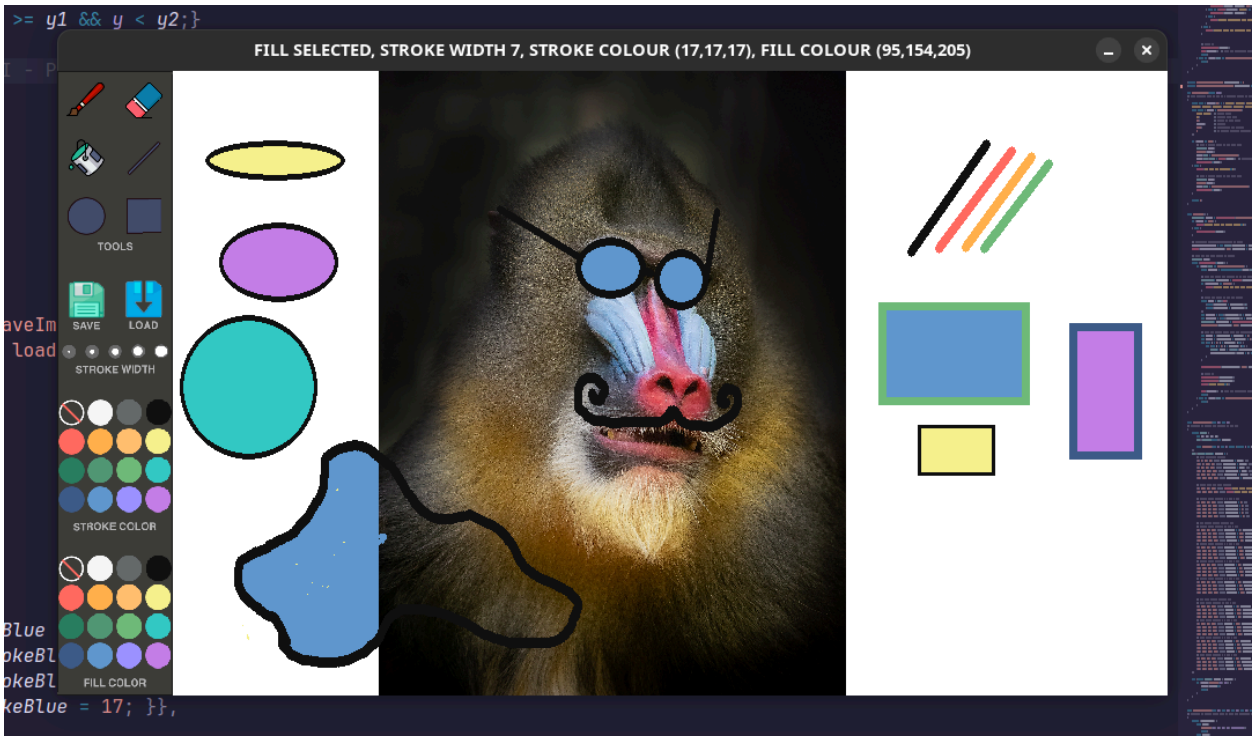


Figure 12. Example of use of all drawing tools on canvas with imported image.

The user can now freely draw using any of the tools that they wish to use by selecting them in the GUI panel. The tools work in an intuitive manner with the exception of the circle tool which will work similar to the one in Photoshop. If the user holds shift down, a circle is drawn, otherwise an ellipse is drawn. Figure 12 shows an example of the output of drawing using each of the tools functionality to alter the image and background canvas.

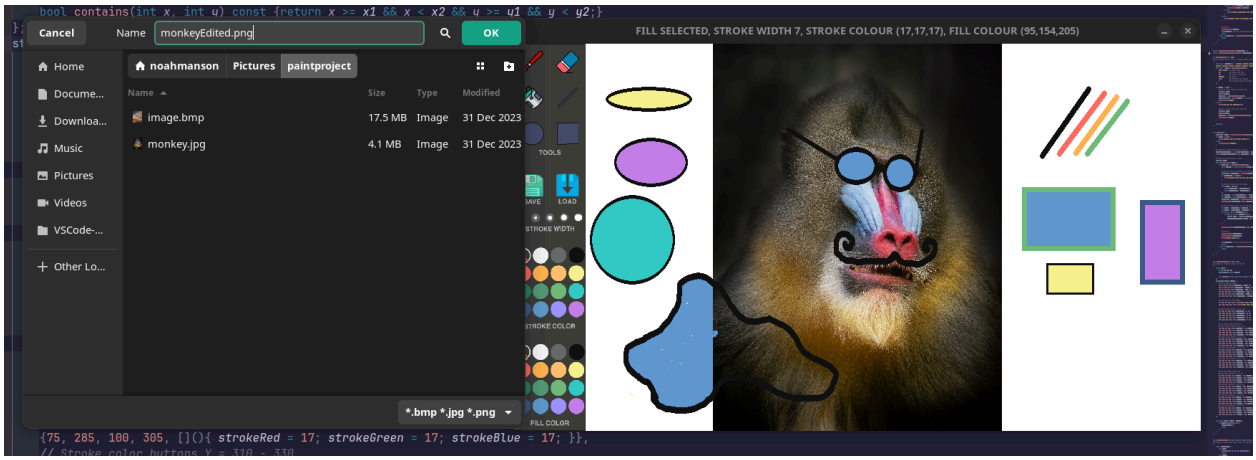


Figure 13. File dialogue to allow user to save image

Finally, to save the resulting image, the user needs only to click the save button on the GUI. Similar to the image load feature, a file dialogue will pop up allowing the user to choose an appropriate file name and format as shown in Figure 13. The user must save in one of the three supported file formats in the SDL2_image library and have their file end in either .png .bmp or .jpg. Once the user presses ok the image is saved to the location they selected. Figure 14 below shows the resulting png image file that was saved to disk from the previously-created masterpiece.

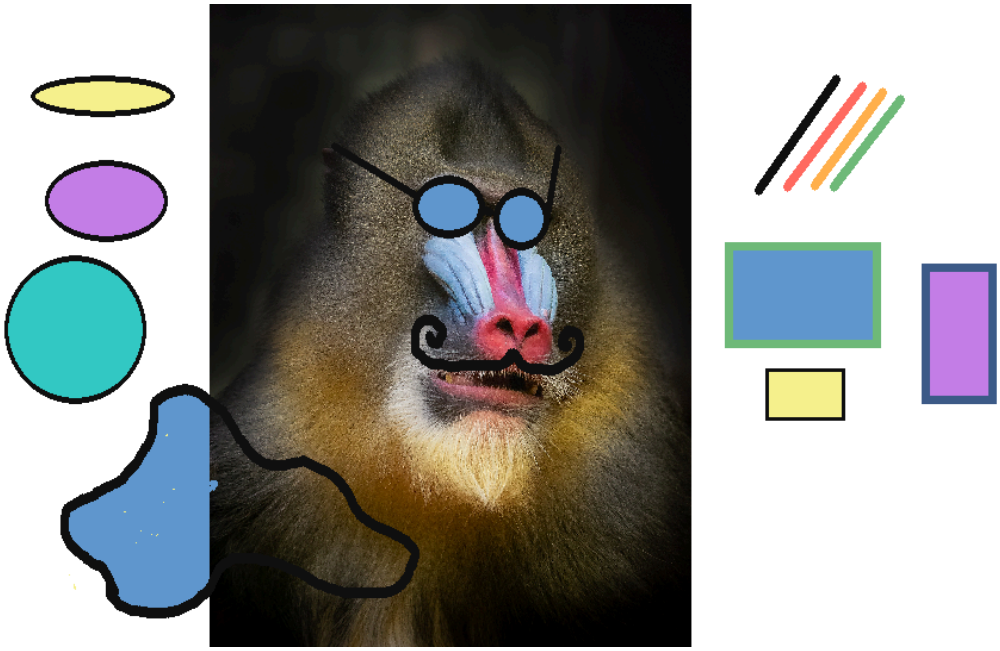


Figure 14. Saved monkeyEdited.png image

Demo video of PalettePro software usage can be found in source file.

References

D'Agostino, D. (2021) *SDL2 tutorials*, *Gigi Labs*. Available at: https://gigi.nullneuron.net/gigilabs/writing/sdl2-tutorials/

javatpoint (no date) *Computer graphics midpoint ellipse algorithm - javatpoint*, *www.javatpoint.com*. Available at: https://www.javatpoint.com/computer-graphics-midpoint-ellipse-algorithm

Granick, J. (no date) *Native-toolkit/Libtinyfiledialogs*, *GitHub*. Available at: https://github.com/native-toolkit/libtinyfiledialogs