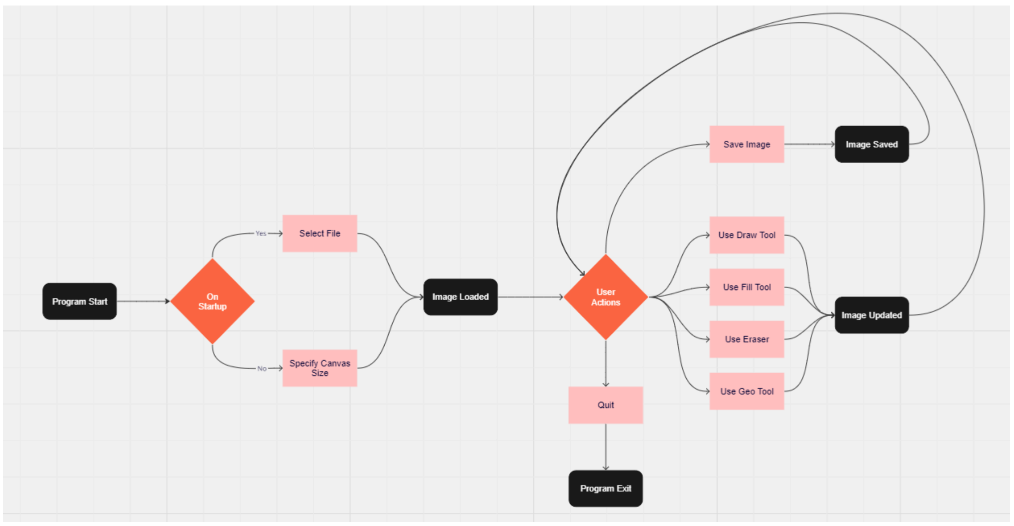
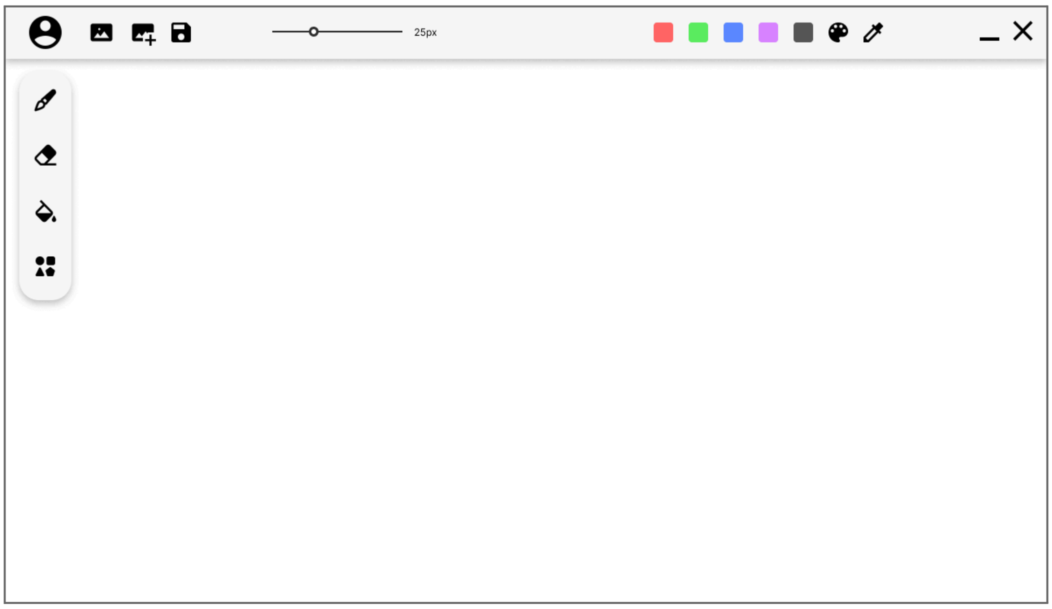
Introduction

This project aimed to create a Raster Graphics Editor / Paint Program in C++ using SDL. As I enjoy UI design and making desktop applications, I felt that PAINter would be a good challenge for developing my programming skills. To get inspiration for the project, I looked at programs such as MSPaint, Photoshop, and Krita. Once I got to grips with those programs, I created a flow chart diagram in Miro to see how *PAINter* would work.



User Interface

The user interface is fairly standard with slider and button controls. I jumped into Figma to make a simple mockup of what I felt would work for the application.

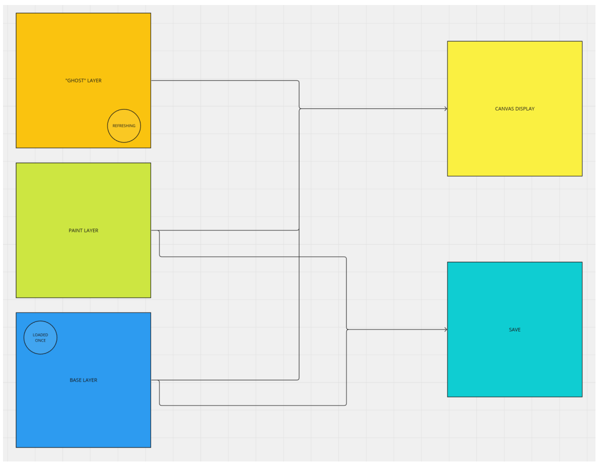


Loading, Creating, and Saving

I aimed for loading, creating, and saving to be done within the user interface. However, I decided to use command arguments instead due to text-displaying issues. Users could load a pre-existing image and save it in any format.

The Paint Tools

Layers

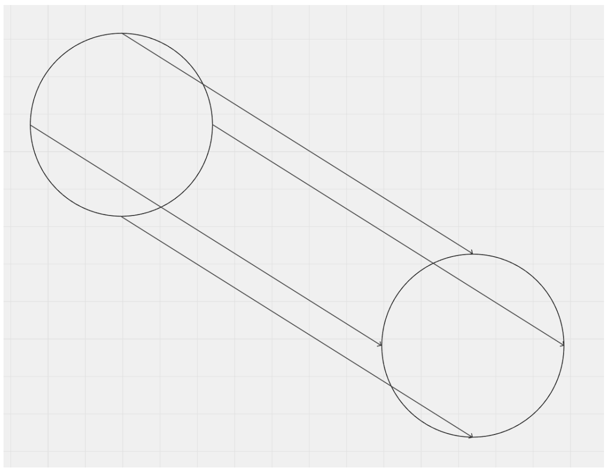


For painting, I implemented a layer-based approach. The 1st layer contained the base image, which would remain untouched throughout the program. The 2nd layer was where painting would be done, and the 3rd layer was used for any rubber-banding or non-permanent canvas drawing. On saving, the application would combine the first 2 layers and output it as a complete image. The 3rd layer would be continuously wiped at every draw call.

Fill Tool

The fill tool uses a stack-based flood fill algorithm. Since recursion causes a stack overflow, a custom array storing the pixel positions allows the program to fill large amounts of space without dealing with memory issues.

Line Tool



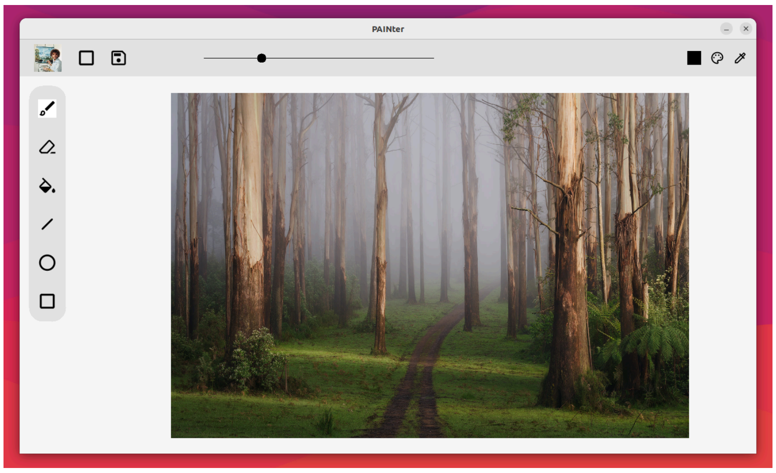
The line tool is built on the Bresenham line drawing algorithm. However, to get line thickness working, I took an extremely CPU-intensive, convoluted approach. If I were to continue working on the application, this would be the first thing to fix. For line thickness, I generate 2 circles around the endpoints of the line. I then draw lines between the points with corresponding angles around their origin, making a perfectly filled-in line. Once again, *very* slow, but it does the job.

Circle & Square Tools

The circle and square tools are built on the line tool. The square tool takes a starting corner and ending corner and draws a box using 4 lines. For the circle, 360 lines are generated around a centre point to give the illusion of a perfectly smooth circle. However, due to the already slow algorithm for the line tool, drawing circles can be very computationally expensive.

Implementation

User Interface



For the user interface, I decided to go with an object-oriented structure so that I could easily change buttons, sliders, and other UI elements. This way of designing a UI is very scalable as I could (theoretically) have multiple types of windows, buttons, and sliders.

While there were numerous challenges in creating the UI, I've highlighted some of the major ones below.

Bevelled Rectangles

A common trend in modern UIs is to have bevelled rectangle shapes. Because SDL2 has no built-in UI, I had to write the bevelling myself. This was done by creating 3 smaller rectangles, and ¼ circles to place on the corners of the main rectangle. I used a "radius" detector that would draw a normal rectangle if the radius was 0. I could then re-call this function for the 3 small rectangles.

```
// renderPanel can be used to draw both sharp and beveled rectangles. Since it's designed for UIs, renderPanel has not been designed to draw onto surfaces
(however with some changes, it could be).
void PNTR_Panel::renderPanel(SDL_Renderer *renderer, SDL_Rect rect, SDL_Color color, int radius)
{
  if (radius) // If the panel is bevelled
  {
  SDL_Rect r1 = {rect.x, rect.y+radius, rect.w, rect.h-radius*2}; // make small top rectangle
  SDL_Rect r2 = {rect.x+radius, rect.y, rect.w-radius*2, radius}; // make large middle rectangle
  SDL_Rect r3 = {rect.x+radius, rect.y+rect.h-radius, rect.w-radius*2, radius}; // make small bottom rectangle

  // the 3 rectangles make sure that the corners are empty.
  /*    - - - - -
       |  r1  |
   - - - - - - - - - - -
   |            |
   |            |
   |    r2      |
   |            |
   |            |
   - - - - - - - - - - -
      |  r3  |
      - - - - - -
  */

  // renderPanel is re-called without no radius.
  renderPanel(renderer, r1, color, 0);
  renderPanel(renderer, r2, color, 0);
  renderPanel(renderer, r3, color, 0);

  // draw the 4 quarter circles using the static function renderCircle from PNTR_Circle

  PNTR_Circle::renderCircle(renderer, new PNTR_Vector2I(rect.x, rect.y), &color, radius, new PNTR_Vector2I(-radius, -radius));
  PNTR_Circle::renderCircle(renderer, new PNTR_Vector2I(rect.x+rect.w, rect.y), &color, radius, new PNTR_Vector2I(radius, -radius));
  PNTR_Circle::renderCircle(renderer, new PNTR_Vector2I(rect.x+rect.w, rect.y+rect.h), &color, radius, new PNTR_Vector2I(radius, radius));
  PNTR_Circle::renderCircle(renderer, new PNTR_Vector2I(rect.x, rect.y+rect.h), &color, radius, new PNTR_Vector2I(-radius, radius));

  } else { // Otherwise draw a rectangle using the fillRect function.
    SDL_Surface *surface = SDL_CreateRGBSurfaceWithFormat(0, rect.w, rect.h, 32, SDL_PIXELFORMAT_RGBA8888);
    SDL_SetSurfaceBlendMode(surface, SDL_BLENDMODE_BLEND);

    SDL_FillRect(surface, NULL, SDL_MapRGBA(surface->format, color.r, color.g, color.b, color.a));
    SDL_Texture *texture = SDL_CreateTextureFromSurface(renderer, surface);
    SDL_FreeSurface(surface);

    SDL_RenderCopy(renderer, texture, NULL, &rect);
    SDL_DestroyTexture(texture);
  }
}
```

Getting the ¼ circles involves an entirely different function.

```
// circleToSurface() creates a trimmable circle surface.
SDL_Surface *PNTR_Circle::circleToSurface(int radius, SDL_Color *color, PNTR_Vector2I *trim)
{
    SDL_Surface *surface = SDL_CreateRGBSurfaceWithFormat(0, radius * 2, radius * 2, 32, SDL_PIXELFORMAT_RGBA8888);

    PNTR_Vector2I index = PNTR_Vector2I();

    // It iterates through every point on the surface and checks if the point is within the circle.
    // This has the potential to be slow on larger images as it would be checking areas that the circle doesnt fill.

    for (index.y = -radius; index.y <= radius; ++index.y)
    {
        for (index.x = -radius; index.x <= radius; ++index.x)
        {
            if (index.x * index.x + index.y * index.y <= radius * radius)
            {
                PNTR_Vector2I change = PNTR_Vector2I(radius + index.x - trim->x, radius + index.y - trim->y);

                // If the point is within the trimmed image bounds, draw it onto the surface

                if (change.x >= 0 && change.x < surface->w && change.y >= 0 && change.y < surface->h)
                {
                    setSurfacePixel(surface, color, change);
                }
            }
        }
    }
    return surface;
}
```

Button Actions

As I was using an OOP structure, I wanted to make functions assignable to buttons so that when a button press occurs it would run the assigned function. I had to use std::function to store the functions as void function pointers didn't work.

```
// Assign button actions. Since these actions must adjust the gui, and therefore must be class functions, they have to be assigned using std::function and std::bind.
std::function<void()> clearFunc = bind(&PNTR_Window::clearPaintLayerAction, this);
gui->clearImageButton->setAction(clearFunc);
```

Another useful function was *isMouseOver(),* which helped me know when the mouse was over or clicking any of my UI elements.

```
// The isMouseOver function is extremely important in detected hover and pressEvents. It checks if the mouse position is within the bounds of the bbox.
/*
    (bbox->x, bbox->y) - - - - - - - - - - - - - - - - - (bbox->x + bbox->w, bbox->y)
          |                                              |
          |                o (mouse-x, mouse->y)         |
          |                                              |
          |                                              |
    (bbox->x, bbox->y + bbox->h) - - - - - - - (bbox->x + bbox->w, bbox->y + bbox->h)
*/
bool PNTR_Widget::isMouseOver(PNTR_Vector2I* mouse) {
    return (
        (mouse->x >= bbox->x && mouse->x <= bbox->x + bbox->w) &&
        (mouse->y >= bbox->y && mouse->y <= bbox->y + bbox->h)
    );
}
```

Viewport Controls

The first issue with viewport controls was getting the canvas to pan and zoom. I did this by finding the position of the mouse click. Then, when dragging, calculate the distance between the two and assign that to the bounding box of the canvas.

```
 // Otherwise, if the middle mouse is down, move the canvas.
else if (middleMouseDown)
{
    // The code sets the canvas location to the sum of its previous location
    // and the difference between the mouse and its last location.
    // This works on a per-call basis and gives a smooth-looking translation.
    gui->canvas->getBBox()->x = lastCanvasPos->x + mousePos->x - lastPos->x;
    gui->canvas->getBBox()->y = lastCanvasPos->y + mousePos->y - lastPos->y;
}
```

The other issue I came across was converting “Mouse Coordinates” to “Canvas Coordinates”. Since the position of the mouse in the application was different from the position of the mouse in image coordinates, I had to write a formula that mapped the two together.

```
// Map the window mouse position to the canvas.
// (mouse.x - canvas.x) / (canvas.w) * (image.w)

PNTR_Vector2I(
                (int)(((mousePos->x - gui->canvas->getBBox()->x) / (float)gui->canvas->getBBox()->w) * gui->canvas->getImageLayer()->w),
                (int)(((mousePos->y - gui->canvas->getBBox()->y) / (float)gui->canvas->getBBox()->h) * gui->canvas->getImageLayer()->h)
);
```

Sliders

The biggest problem that came with sliders was figuring out how to process the slide. I found that I could detect when the slider was active and whether the mouse was down, and then process the slider value from that.

```
// Check if the user is currently moving a slider. If so, update the slider based on the mouse position.
    if (leftMouseDown)
    {
        if (activeSlider != nullptr)
        {
            activeSlider->setValue((float(mousePos->x) - float(activeSlider->getBBox()->x)) / float(activeSlider->getBBox()->w));
        }
    }


// setValue takes an input and clamps it between 0.0f and 1.0f.
void PNTR_Slider::setValue(float v) { value = SDL_min(SDL_max(0.0f, v), 1.0f); };
```

Loading, Creating, Saving

For loading and creating, that was dependant on the command line arguments. I processed them like you would in any standard application.

```
    // Process command line arguments
    for (int c = 0; c < argc; c++)
    {
        if (strcmp(argv[c], "-f") == 0)
        {
            readFilePath = argv[c + 1];
        }

        if (strcmp(argv[c], "-o") == 0)
        {
            writeFilePath = argv[c + 1];
        }
    }
    // It's ok for readFilePath and writeFilePath to remain empty.
    // They are null-checked later on in the application.
```

For the saving function, I combined the 2 surfaces in my canvas and exported it as a .png.

```
// Combine the layers for the fill tool or save the image.
SDL_Surface *PNTR_Canvas::combineLayers()
{
    SDL_Surface *output = SDL_CreateRGBSurface(0, sourceSize.w, sourceSize.h, 32, 0x00ff0000, 0x0000ff00, 0x000000ff, 0xff000000);
    SDL_BlitSurface(imageLayer, NULL, output, NULL);
    SDL_BlitSurface(paintLayer, NULL, output, NULL);
    return output;
};


// Save the image by combining the images and saving them out as PAINting.png
void PNTR_Canvas::saveImage()
{
    SDL_Surface *output = combineLayers();
    // if writeFilePath is not initialized, it will default to PAINting.png
    if (writeFilePath != NULL)
    {
        IMG_SavePNG(output, writeFilePath);
    }
    else
    {
        IMG_SavePNG(output, "./PAINting.png");
    }

    SDL_ShowSimpleMessageBox(SDL_MESSAGEBOX_INFORMATION, "File Saved!", "Your PAINting has been saved!", NULL);

    SDL_FreeSurface(output);
}
```

Painting

Fill Tool

The fill tool takes a pixel from the stack, if the pixel is fillable, it adds the neighbouring pixels to the stack and repeats until done.

```
case PNTR_PaintMode::FILL:
        // Check that click was in the image bounds
        if ((mouseOnCanvas.x < 0 || mouseOnCanvas.x > getSourceSize().w) || (mouseOnCanvas.y < 0 || mouseOnCanvas.y > getSourceSize().h))
            break;

        // Get the clicked pixel and add it to the queue.
        pixel = getSurfacePixel(output, PNTR_Vector2I(mouseOnCanvas.x, mouseOnCanvas.y));
        fillStack.clear();
        fillStack.push_back(PNTR_Vector2I({mouseOnCanvas.x, mouseOnCanvas.y}));

        // The flood fill uses an array called "visit_array", which checks if a pixel has already been added to the queue. This sets the size of the array to fit the image.
        visit_array.resize(getImageLayer()->w, std::vector<bool>(getImageLayer()->h, false));
        while (!fillStack.empty()) // While the fillstack isn't empty,
        {
            // Get the first element in the array and take it out, then run the flood fill on it.
            currentPos = fillStack.front();
            fillStack.erase(fillStack.begin());
            fillStack.shrink_to_fit();
            PNTR_Canvas::floodFill(currentPos, output, getPaintLayer(), activeColor, pixel);
        }
        // Once finished, clear the array and stack.
        fillStack.clear();
        fillStack.shrink_to_fit();
        visit_array.clear();
        visit_array.resize(getImageLayer()->w, std::vector<bool>(getImageLayer()->h, false));
        paintChanged = true;
        break;
```

```
// isValid() checks the pixel during flood fill. If the pixel is within the bounds and needs to be filled, return True
bool PNTR_Canvas::isValid(PNTR_Vector2I position, SDL_Surface *read, SDL_Color *fill, SDL_Color *pixel)
{
    return (position.x >= 0 && position.x < getSourceSize().w) && (position.y >= 0 && position.y < getSourceSize().h) && !compare(fill, pixel) && compare(getSurfacePixel(read,
position), pixel);
}


// The main flood fill algorithm
void PNTR_Canvas::floodFill(PNTR_Vector2I pos, SDL_Surface *read, SDL_Surface *write, SDL_Color *fill, SDL_Color *pixel)
{
    // Check if the current pixel is valid for filling
    if (!isValid(pos, read, fill, pixel) && !visit_array[pos.x][pos.y])
        return;

    // Fill the pixel
    setSurfacePixel(write, fill, pos);

    // Set the visit_array at the pixel to true, so that no other fill instances will go over the same point.
    visit_array[pos.x][pos.y] = true;

    // Then, add the pixel positions next to the current pixel (as long as they're valid).
    if (isValid(PNTR_Vector2I(pos.x + 1, pos.y), read, fill, pixel) && !visit_array[pos.x + 1][pos.y])
    {
        fillStack.push_back(PNTR_Vector2I(pos.x + 1, pos.y));
        visit_array[pos.x + 1][pos.y] = true;
    }
    if (isValid(PNTR_Vector2I(pos.x - 1, pos.y), read, fill, pixel) && !visit_array[pos.x - 1][pos.y])
    {
        fillStack.push_back(PNTR_Vector2I(pos.x - 1, pos.y));
        visit_array[pos.x - 1][pos.y] = true;
    }
    if (isValid(PNTR_Vector2I(pos.x, pos.y + 1), read, fill, pixel) && !visit_array[pos.x][pos.y + 1])
    {
        fillStack.push_back(PNTR_Vector2I(pos.x, pos.y + 1));
        visit_array[pos.x][pos.y + 1] = true;
    }
    if (isValid(PNTR_Vector2I(pos.x, pos.y - 1), read, fill, pixel) && !visit_array[pos.x][pos.y - 1])
    {
        fillStack.push_back(PNTR_Vector2I(pos.x, pos.y - 1));
        visit_array[pos.x][pos.y - 1] = true;
    }
    // Eventually, all avaliable pixels will be added to the stack, and the stack size will eventually return to 0.
}
```

Geometric Tools

I split the code into the original Bresenham line drawing algorithm, and then the thick line algorithm. This was done so that I could adjust later on without having to piece them apart.

```
void PNTR_Canvas::drawThickLine(SDL_Surface *surface, SDL_Rect bounds, PNTR_Vector2I p1, PNTR_Vector2I p2, int thickness, SDL_Color *color)
{
    // iterate through every angle
    for (float theta = 0; theta < 360; theta++)
    {
        float radians = (float)(theta * M_PI) / 180.0f;
        // Iterate through every level of thickness (THIS IS WHY IT'S SLOW)
        for (int t = 0; t < thickness / 2; t++)
        {
            // Get the new points to draw a line between.
            PNTR_Vector2I c1 = PNTR_Vector2I(p1.x + SDL_cosf(radians) * t, p1.y + SDL_sinf(radians) * t);
            PNTR_Vector2I c2 = PNTR_Vector2I(p2.x + SDL_cosf(radians) * t, p2.y + SDL_sinf(radians) * t);
            // Check if the points are within the image bounds (yes, there are issues when drawing lines at boundaries)
            if ((c1.x >= 0 && c1.x < bounds.w) && (c1.y >= 0 && c1.y < bounds.h) && (c2.x >= 0 && c2.x < bounds.w) && (c2.y >= 0 && c2.y < bounds.h))
            {
                // Draw the line
                drawLine(surface, bounds, c1, c2, color);
            }
        }
    }
}
/* Source based on Eike Anderson starts here : https://brightspace.bournemouth.ac.uk/d2l/le/lessons/345037/topics/1968571 */
void PNTR_Canvas::drawLine(SDL_Surface *surface, SDL_Rect bounds, PNTR_Vector2I p1, PNTR_Vector2I p2, SDL_Color *color)
{
    PNTR_Vector2I change = PNTR_Vector2I(abs(p2.x - p1.x), abs(p2.y - p1.y));

    PNTR_Vector2I scale = PNTR_Vector2I(p1.x < p2.x ? 1 : -1, p1.y < p2.y ? 1 : -1);

    int error = (change.x > change.y ? change.x : -change.y) / 2, e2;

    while (true)
    {
        /* draw point only if coordinate is valid */
        if (p1.x >= 0 && p1.x < bounds.w && p1.y >= 0 && p1.y < bounds.h)
        {
            setSurfacePixel(surface, color, p1);
        }

        if (p1.x == p2.x && p1.y == p2.y)
            break;
        e2 = error;
        if (e2 > -change.x)
        {
            error -= change.y;
            p1.x += scale.x;
        }
        if (e2 < change.y)
        {
            error += change.x;
            p1.y += scale.y;
        }
    }
}
/* Source based on Eike Anderson ends here*/
```

The square and circle functions then call *drawThickLine()*.

```
void PNTR_Canvas::drawSquare(SDL_Surface *surface, SDL_Rect bounds, PNTR_Vector2I tl, PNTR_Vector2I br, int thickness, SDL_Color *color)
{
    // Get the 4 corners of the square
    PNTR_Vector2I p1 = PNTR_Vector2I(tl.x, tl.y);
    PNTR_Vector2I p2 = PNTR_Vector2I(br.x, tl.y);
    PNTR_Vector2I p3 = PNTR_Vector2I(br.x, br.y);
    PNTR_Vector2I p4 = PNTR_Vector2I(tl.x, br.y);
    // Draw 4 thick lines between them
    drawThickLine(surface, bounds, p1, p2, thickness, color);
    drawThickLine(surface, bounds, p2, p3, thickness, color);
    drawThickLine(surface, bounds, p3, p4, thickness, color);
    drawThickLine(surface, bounds, p4, p1, thickness, color);
}

void PNTR_Canvas::drawCircle(SDL_Surface *surface, SDL_Rect bounds, PNTR_Vector2I *center, SDL_Color *color, int radius, int thickness)
{
    // Iterate through every degree
    for (float theta = 0; theta < 361; theta++)
    {
        float r1 = (float)(theta * M_PI) / 180.0f;
        float r2 = (float)((theta - 1) * M_PI) / 180.0f;

        // Draw a line between each degree. Because it is only 1 degree, nobody can see that a segmented circle is being drawn.
        drawThickLine(surface, bounds, PNTR_Vector2I(center->x + radius * cos(r1), center->y + radius * sin(r1)), PNTR_Vector2I(center->x + radius * cos(r2), center->y + radius *
sin(r2)), thickness / 2, color);
    }
}
```

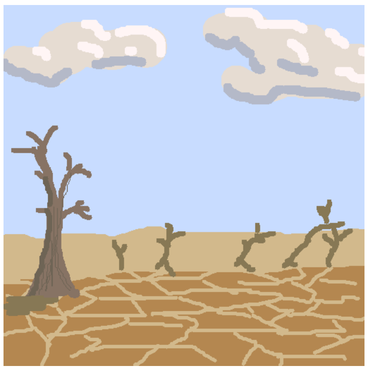
To get rubber banding to work, I drew the shapes to the ghost layer and then to the paint layer on mouse release.

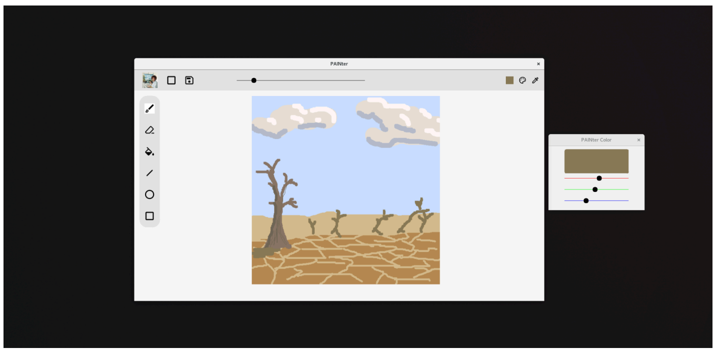
```
if (!middleMouseDown)
        {
            if (!leftMouseDown) // Check if the left mouse button isnt down.
            {
                // Draw the line onto the paint layer
                drawThickLine(paintLayer, sourceSize, *shapeStart, mouseOnCanvas, drawSize, activeColor);
                paintChanged = true;
            }
            else
            {
                // Draw the line onto the ghost layer
                drawThickLine(ghostLayer, sourceSize, *shapeStart, mouseOnCanvas, drawSize, activeColor);
                ghostChanged = true;
            }
        }
```

Results

Overall I am quite happy with this project. However, If I were to continue working on it, these are the things that I would address next.

- Faster line drawing algorithm
- Reusable window / Gui classes with individual event management
- Buttons for loading and creating images





An example of *PAINter* in use.

Bibliography

Keltar, 2016. How to set a GUI Button in the win32 window using SDL C++? [online]. Stack Overflow. Available from: https://stackoverflow.com/questions/39133873/how-to-set-a-gui-button-in-the-win32-window-using-sdl-c [Accessed 21 Dec 2023].

Olevgard, 2018. Drawing a rectangle with SDL2 [online]. Stack Overflow. Available from: https://stackoverflow.com/questions/21890627/drawing-a-rectangle-with-sdl2 [Accessed 21 Dec 2023].

OpenAI, 2023. ChatGPT [online]. chat.openai.com. Available from: https://chat.openai.com/.

Stack Overflow, 2022. Stack Overflow - Where Developers Learn, Share, & Build Careers [online]. Stack Overflow. Available from: https://stackoverflow.com/.

Theicfire, 2023. Makefile Tutorial by Example [online]. Makefile Tutorial. Available from: https://makefiletutorial.com/.