REPORT DELIVERABLE 1: The Framebuffer

Team Group: G - 10
Names:

- Ander Barnadas Mendizabal u233729
  ander.barnadas01@estudiant.upf.edu
- Sergi Cases Alonso u232159
  sergi.cases01@estudiant.upf.edu

1- Drawing Lines

Algorithm Explanation (Input, Output, Description) [5-10 lines]:

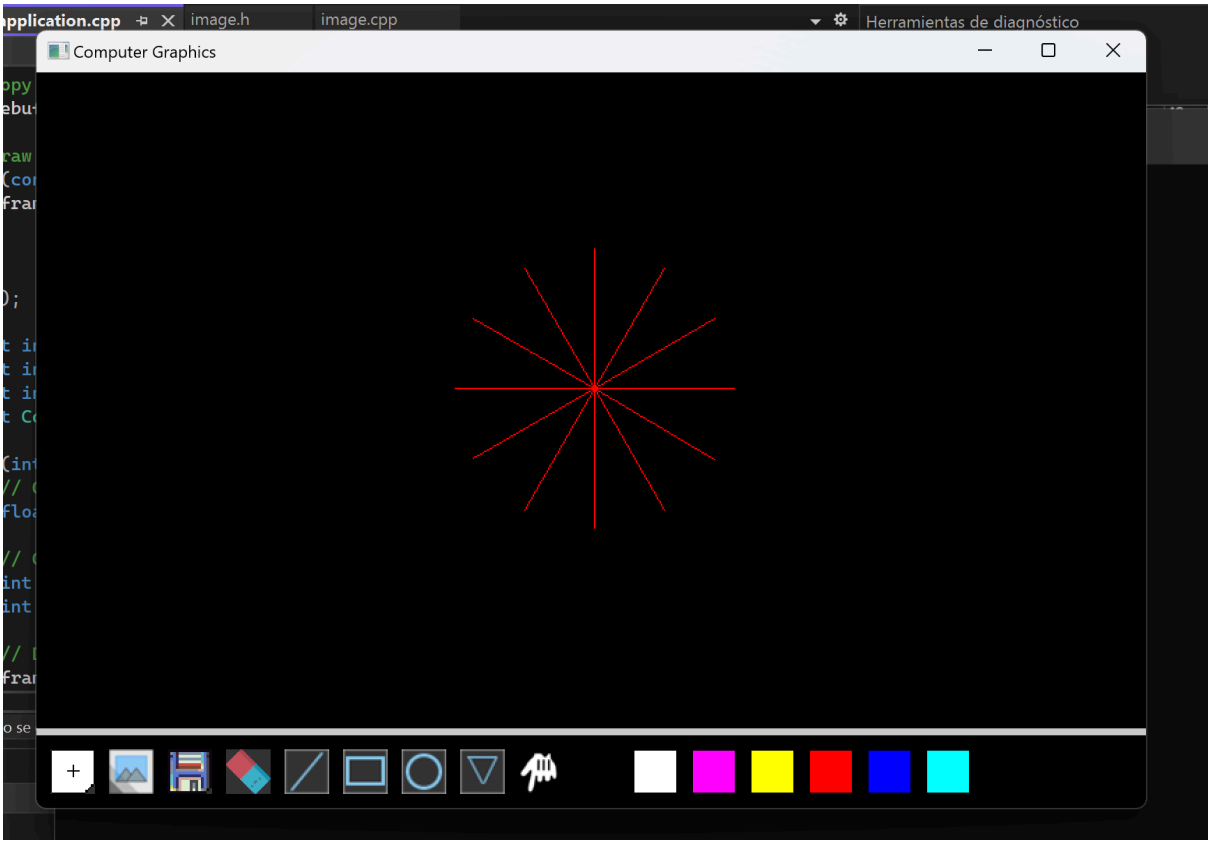
(Provide a step-by-step description in your own words of how the DDA algorithm works, specifying the input required and the output produced.)

As inputs of this function we have: Starting point (x0,y0)(x_0, y_0)(x0,y0): The coordinates of the line's start. Ending point (x1,y1)(x_1, y_1)(x1,y1): The coordinates of the line's end. A Color object (c): The color to use for drawing the line.
As an output we get the pixels of a straight line drawn from (x0,y0)(x_0, y_0)(x0,y0) to (x1,y1)(x_1, y_1)(x1,y1) on an image.
First the changes in the x-coordinate and the y-coordinate are computed. The number of steps required to draw the line is the maximum of the absolute values of dx and dy. This ensures the line is drawn pixel by pixel with uniform spacing. The increments for x and y per step are calculated as x = dx/steps and y = dy/steps.

Code Snapshot:

```cpp
void Image::DrawLineDDA(int x0, int y0, int x1, int y1, const Color& c) {
    int dx = x1 - x0;
    int dy = y1 - y0;

    int steps = std::max(abs(dx), abs(dy));

    // Avoid division by zero
    if (steps == 0) {
        SetPixel(x0, y0, c);
        return;
    }

    float x_inc = dx / (float)steps;
    float y_inc = dy / (float)steps;

    float x = x0;
    float y = y0;

    for (int i = 0; i <= steps; i++) {
        if (x >= 0 && x < width && y >= 0 && y < height) {
            SetPixel(round(x), round(y), c);
        }
        x += x_inc;
        y += y_inc;
    }
}
```

Screenshots/Visual Output:



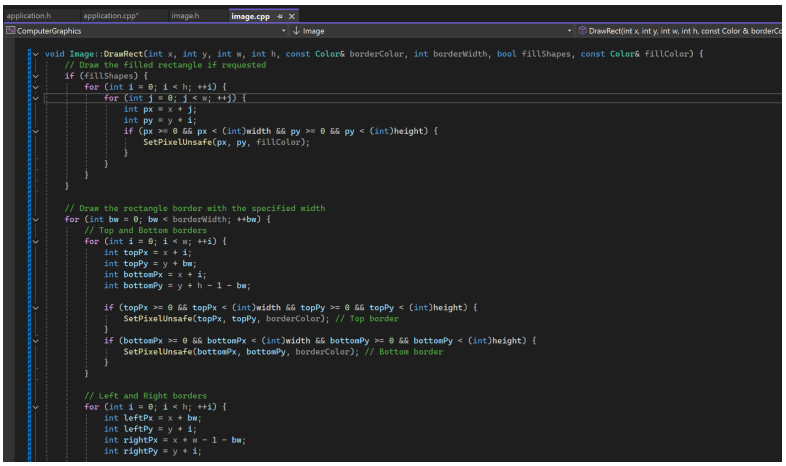
2- Drawing Rectangles

Algorithm Explanation (Input, Output, Description) [5-10 lines]:

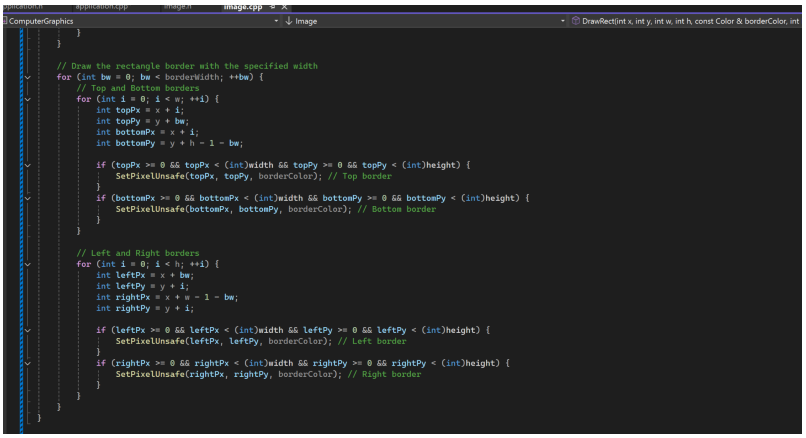
As input we have (x,y): The top-left corner coordinates of the rectangle. w,hw, hw,h: Width and height of the rectangle. borderColor, borderWidth, fillShapes (Boolean): Whether the rectangle should be filled and fillColor. As output we get a rectangle drawn on the image, either filled or just with a border, based on the specified parameters.

Fill the Rectangle (if fillShapes is true): Loops through every pixel inside the rectangle (from the top-left corner to the bottom-right corner). Checks if the pixel is within the image boundaries. Colors the pixel with the fillColor. Draw the Border: Iteratively draws the top and bottom borders for the specified borderWidth. Ensures the pixels are within the image boundaries. Iteratively draws the left and right borders for the specified borderWidth. Ensures the pixels are within the image boundaries.

Code Snapshot:

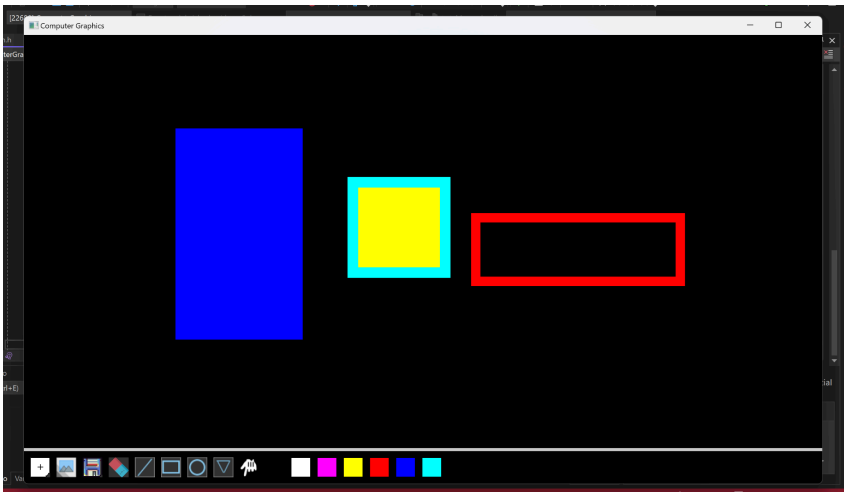


```
void Image::DrawRect(int x, int y, int w, int h, const Color& borderColor, int borderWidth, bool fillShapes, const Color& fillColor) {
    // Draw the filled rectangle if requested
    if (fillShapes) {
        for (int i = 0; i < h; ++i) {
            for (int j = 0; j < w; ++j) {
                int px = x + j;
                int py = y + i;
                if (px >= 0 && px < (int)width && py >= 0 && py < (int)height) {
                    SetPixelUnsafe(px, py, fillColor);
                }
            }
        }
    }

    // Draw the rectangle border with the specified width
    for (int bw = 0; bw < borderWidth; ++bw) {
        // Top and Bottom borders
        for (int i = 0; i < w; ++i) {
            int topPx = x + i;
            int topPy = y + bw;
            int bottomPx = x + i;
            int bottomPy = y + h - 1 - bw;

            if (topPx >= 0 && topPx < (int)width && topPy >= 0 && topPy < (int)height) {
                SetPixelUnsafe(topPx, topPy, borderColor); // Top border
            }
            if (bottomPx >= 0 && bottomPx < (int)width && bottomPy >= 0 && bottomPy < (int)height) {
                SetPixelUnsafe(bottomPx, bottomPy, borderColor); // Bottom border
            }
        }

        // Left and Right borders
        for (int i = 0; i < h; ++i) {
            int leftPx = x + bw;
            int leftPy = y + i;
            int rightPx = x + w - 1 - bw;
            int rightPy = y + i;
```



```
    }

    // Draw the rectangle border with the specified width
    for (int bw = 0; bw < borderWidth; ++bw) {
        // Top and Bottom borders
        for (int i = 0; i < w; ++i) {
            int topPx = x + i;
            int topPy = y + bw;
            int bottomPx = x + i;
            int bottomPy = y + h - 1 - bw;

            if (topPx >= 0 && topPx < (int)width && topPy >= 0 && topPy < (int)height) {
                SetPixelUnsafe(topPx, topPy, borderColor); // Top border
            }
            if (bottomPx >= 0 && bottomPx < (int)width && bottomPy >= 0 && bottomPy < (int)height) {
                SetPixelUnsafe(bottomPx, bottomPy, borderColor); // Bottom border
            }
        }

        // Left and Right borders
        for (int i = 0; i < h; ++i) {
            int leftPx = x + bw;
            int leftPy = y + i;
            int rightPx = x + w - 1 - bw;
            int rightPy = y + i;

            if (leftPx >= 0 && leftPx < (int)width && leftPy >= 0 && leftPy < (int)height) {
                SetPixelUnsafe(leftPx, leftPy, borderColor); // Left border
            }
            if (rightPx >= 0 && rightPx < (int)width && rightPy >= 0 && rightPy < (int)height) {
                SetPixelUnsafe(rightPx, rightPy, borderColor); // Right border
            }
        }
    }
}
```

Screenshots/Visual Output:



3- Rasterizing Triangles

Algorithm Explanation (Input, Output, Description) [5-10 lines]:

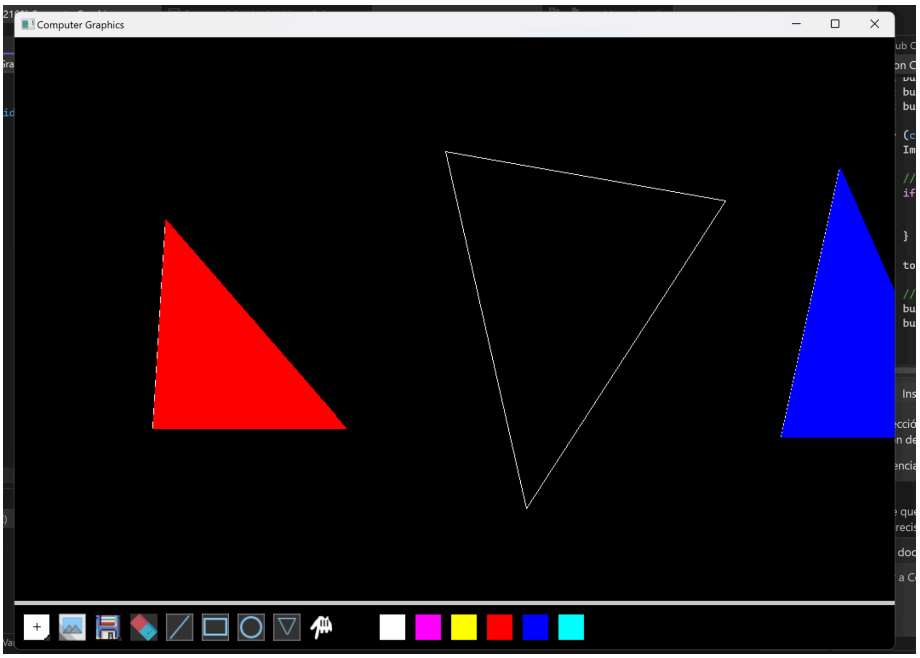
As input: Triangle vertices: (p0,p1,p2)(p_0, p_1, p_2)(p0,p1,p2), borderColor, isFilled: Boolean indicating whether the triangle should be filled, fillColor.
As output: A triangle with a specified border (and optionally filled interior) drawn on the image.
It first uses the DDA line algorithm to draw lines connecting the three vertices forming the triangle's perimeter. The ScanLineDDA function calculates the intersection points (x) of the scanline with the edges of the triangle. For each scanline at y, pixels between the intersection points are colored with fillColor. This ensures the triangle interior is filled pixel by pixel.

Code Snapshot:

```cpp
void Image::ScanLineDDA(int x0, int y0, int x1, int y1, int& xIntersect, int y) {
    if (y1 == y0) return; // Prevent division by zero

    float slope = float(x1 - x0) / float(y1 - y0);  // Calculate the slope

    // Calculate the intersection point on the scanline at y
    xIntersect = x0 + int(slope * (y - y0));
}

void Image::DrawTriangle(const Vector2& p0, const Vector2& p1, const Vector2& p2, const Color& borderColor, bool isFilled, const Color& fillColor) {
    // Create non-const local copies of the vertices
    Vector2 v0 = p0;
    Vector2 v1 = p1;
    Vector2 v2 = p2;

    // Draw the triangle's border
    DrawLineDDA(v0.x, v0.y, v1.x, v1.y, borderColor);
    DrawLineDDA(v1.x, v1.y, v2.x, v2.y, borderColor);
    DrawLineDDA(v2.x, v2.y, v0.x, v0.y, borderColor);

    // If the triangle should be filled
    if (isFilled) {
        // Sort vertices by y-coordinate (ascending order)
        Vector2 sorted[3] = { v0, v1, v2 };
        std::sort(sorted, sorted + 3, [](const Vector2& a, const Vector2& b) { return a.y < b.y; });

        // Extract sorted vertices
        v0 = sorted[0];
        v1 = sorted[1];
        v2 = sorted[2];

        for (int y = v0.y; y <= v2.y; ++y) {
            // For the current scanline, calculate intersections with edges
            int x1, x2;
            ScanLineDDA(v0.x, v0.y, v1.x, v1.y, x1, y); // v0 to v1
            ScanLineDDA(v1.x, v1.y, v2.x, v2.y, x2, y); // v1 to v2
            ScanLineDDA(v2.x, v2.y, v0.x, v0.y, x1, y); // v2 to v0
```

```cpp
}

void Image::DrawTriangle(const Vector2& p0, const Vector2& p1, const Vector2& p2, const Color& borderColor, bool isFilled, const Color& fillColor) {
    // Create non-const local copies of the vertices
    Vector2 v0 = p0;
    Vector2 v1 = p1;
    Vector2 v2 = p2;

    // Draw the triangle's border
    DrawLineDDA(v0.x, v0.y, v1.x, v1.y, borderColor);
    DrawLineDDA(v1.x, v1.y, v2.x, v2.y, borderColor);
    DrawLineDDA(v2.x, v2.y, v0.x, v0.y, borderColor);

    // If the triangle should be filled
    if (isFilled) {
        // Sort vertices by y-coordinate (ascending order)
        Vector2 sorted[3] = { v0, v1, v2 };
        std::sort(sorted, sorted + 3, [](const Vector2& a, const Vector2& b) { return a.y < b.y; });

        // Extract sorted vertices
        v0 = sorted[0];
        v1 = sorted[1];
        v2 = sorted[2];

        for (int y = v0.y; y <= v2.y; ++y) {
            // For the current scanline, calculate intersections with edges
            int x1, x2;
            ScanLineDDA(v0.x, v0.y, v1.x, v1.y, x1, y); // v0 to v1
            ScanLineDDA(v1.x, v1.y, v2.x, v2.y, x2, y); // v1 to v2
            ScanLineDDA(v2.x, v2.y, v0.x, v0.y, x1, y); // v2 to v0

            // Fill pixels between x1 and x2 for the current y
            for (int x = x1; x <= x2; ++x) {
                SetPixel(x, y, fillColor);
            }
        }
    }
}
```

Screenshots/Visual Output:



4- Extra - Rasterizing Circles

Algorithm Explanation (Input, Output, Description) [5-10 lines]:

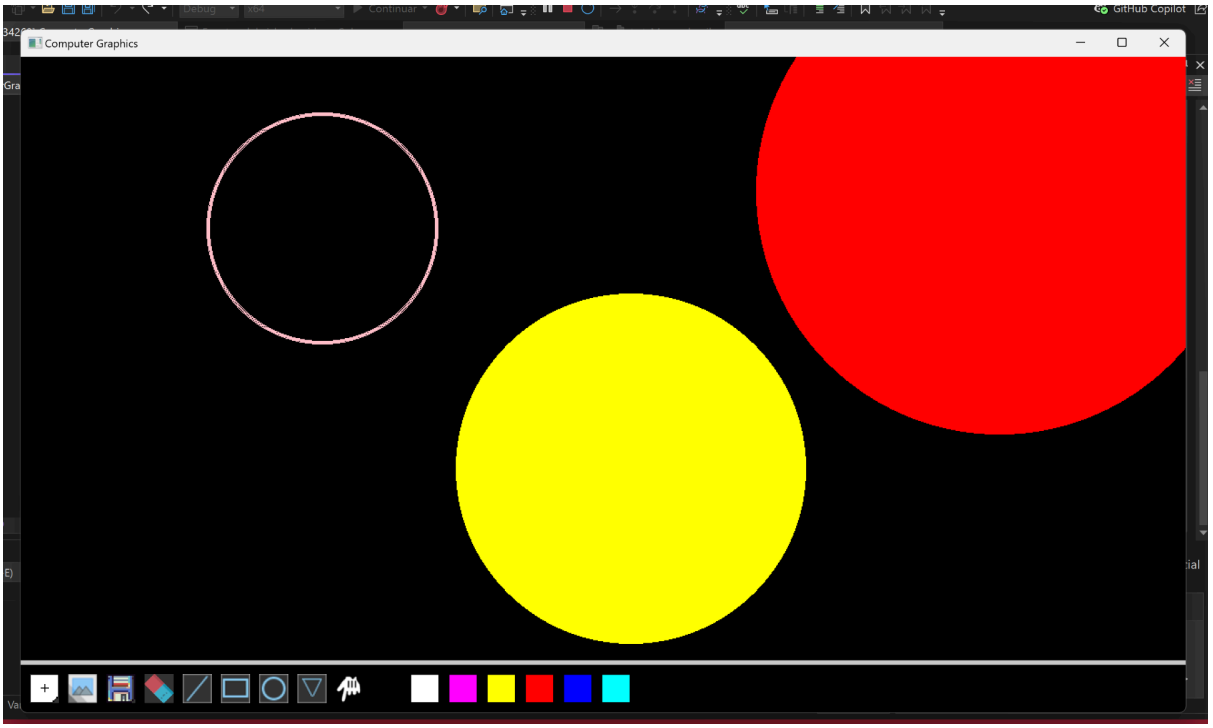
As input: (x,y), radius, borderColor, borderWidth, isFilled, fillColor.
As output: A rasterized circle with an optional border and filled interior, drawn on the image.
Midpoint Circle Drawing Algorithm: The circle's perimeter is drawn using the Midpoint Circle Algorithm, which calculates points incrementally based on a decision variable. Drawing the Circle Border: The algorithm plots points for the circle perimeter at the given radius. If a thicker border is required, concentric circles with decreasing radius are drawn up to the specified borderWidth. Filling the Circle (if isFilled is true): For each vertical slice across the circle, the height of the circle at that x is calculated . Horizontal lines spanning the calculated y-values are drawn at each vertical slice, filling the circle pixel by pixel.

Code Snapshot:

```cpp
void Image::DrawCircle(int x, int y, int r, const Color& borderColor, int borderWidth, bool isFilled, const Color& fillColor)
{
    // Midpoint Circle Drawing Algorithm to draw the perimeter
    int cx = x, cy = y; // Circle center
    int radius = r;
    int d = 1 - radius;  // Midpoint decision variable
    int x0 = 0, y0 = radius;

    // Function to plot points
    auto plotCirclePoints = [&](int cx, int cy, int x, int y, const Color& color) {
        SetPixel(cx + x, cy + y, color); // Octant 1
        SetPixel(cx - x, cy + y, color); // Octant 2
        SetPixel(cx + x, cy - y, color); // Octant 3
        SetPixel(cx - x, cy - y, color); // Octant 4
        SetPixel(cx + y, cy + x, color); // Octant 5
        SetPixel(cx - y, cy + x, color); // Octant 6
        SetPixel(cx + y, cy - x, color); // Octant 7
        SetPixel(cx - y, cy - x, color); // Octant 8
        };

    // Draw the border (perimeter) of the circle
    plotCirclePoints(cx, cy, x0, y0, borderColor);
    while (x0 < y0)
    {
        if (d < 0)
        {
            d += 2 * x0 + 3;
        }
        else
        {
            d += 2 * (x0 - y0) + 5;
            --y0;
        }
        ++x0;
        plotCirclePoints(cx, cy, x0, y0, borderColor);
    }

    // Fill the circle if the isFilled flag is true
```

```cpp
    // Fill the circle if the isFilled flag is true
    if (isFilled)
    {
        for (int i = -radius; i <= radius; ++i) {
            int height = static_cast<int>(sqrt(radius * radius - i * i)); // Calculate the y-distance for each x (i)
            for (int j = cx - height; j <= cx + height; ++j) {
                SetPixel(j, cy + i, fillColor); // Fill horizontal lines inside the circle
            }
        }
    }

    // Handle borderWidth by drawing concentric circles
    for (int width = 1; width < borderWidth; ++width) {
        int newRadius = radius - width;
        if (newRadius <= 0) break; // Avoid negative radius
        int d = 1 - newRadius;  // Reset decision variable
        int x0 = 0, y0 = newRadius;

        // Draw the concentric circle
        plotCirclePoints(cx, cy, x0, y0, borderColor);
        while (x0 < y0)
        {
            if (d < 0)
            {
                d += 2 * x0 + 3;
            }
            else
            {
                d += 2 * (x0 - y0) + 5;
                --y0;
            }
            ++x0;
            plotCirclePoints(cx, cy, x0, y0, borderColor);
        }
    }
}
```

Screenshots/Visual Output:

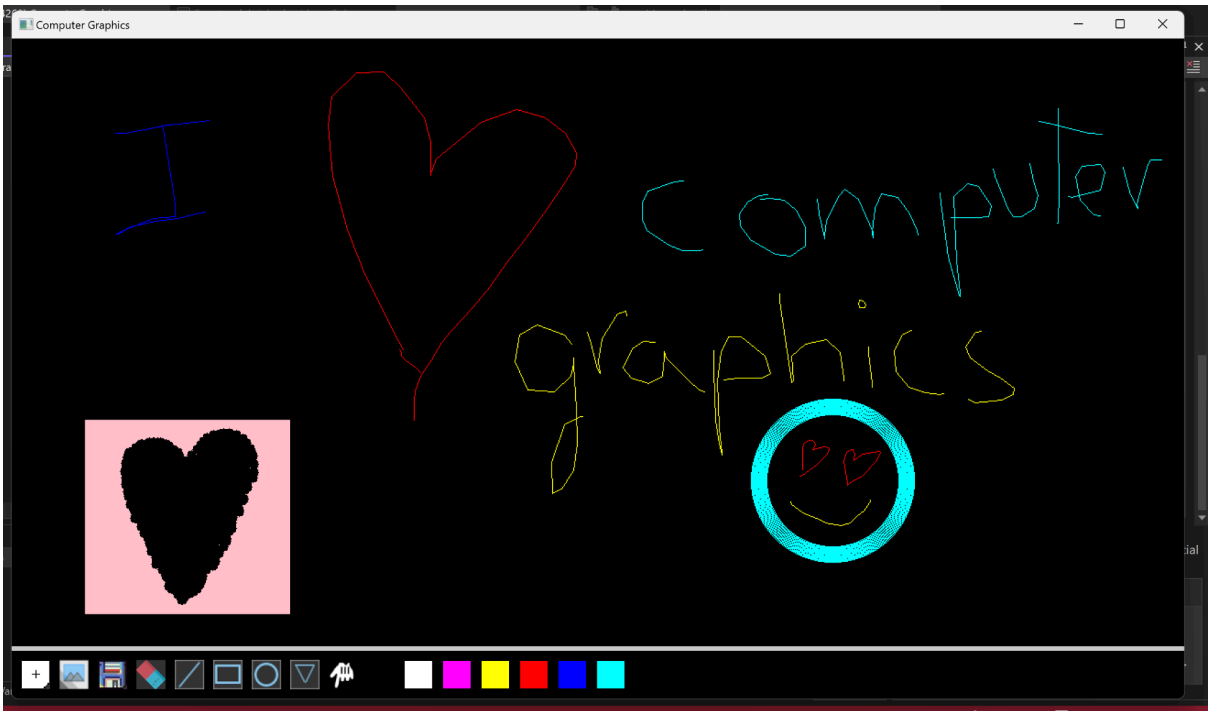


5- Drawing Tool

Interaction Explanation [5-10 lines]:

Firstly, to draw a line the user has to click on a coordinate of the screen and the drag the mouse to another coordinate where the line ends. For the rect it is the same, you click the mouse and drag it to another coordinate and it draws a rectangle, if you want it to be filled, you press key F. For a circle you click on a coord and this will be the center and then drag the mouse in any direction until you reach your wanted radius. In the triangle just click with the mouse your desired 3 vertices. For free drawing, maintain the mouse click and drag it in any direction. You can also make the drawing lines bigger by clicking the + buttons. Also all the keys 1,2,3,4,5… work if you don't want to select each shape with the mouse.

Screenshots/Visual Output:



6- Animation

Algorithm Explanation (Initialization and Update) [5-10 lines]:

Each particle is initialized with a random position across the screen's width and height, simulating scattered snow. A random downward velocity is assigned to mimic falling snow, ensuring variability in speed. Particle color is randomly generated, adding visual diversity. All particles start as active and visible at initialization.Each particle's position is updated based on its velocity and the elapsed time dtdtdt, making it fall downward.
TTL Update: TTL decreases over time. If a particle's TTL reaches 0 or it moves beyond the screen bounds, it becomes inactive. Respawning: Inactive particles are respawned at the top of the screen with new random properties (position, velocity, TTL, color, and size), ensuring a continuous animation.

Code Snapshot (init, render, update):

```cpp
//COlorful snowing-like animation but upwards
void Application::InitParticles() {
    srand(static_cast<unsigned int>(std::time(nullptr))); // Seed RNG

    for (int i = 0; i < MAX_PARTICLES; ++i) {
        // Start at a random position
        particles[i].position = Vector2(rand() % framebuffer.width, rand() % framebuffer.height);


        float speed = static_cast<float>(rand() % 200) ;
        particles[i].velocity = Vector2(0.0f, speed);
        particles[i].color = Color(rand() % 256, rand() % 256, rand() % 256);
        particles[i].acceleration = 1.0f;
        particles[i].ttl = static_cast<float>(rand() % 5000) / 1000.0f + 5.0f;
        particles[i].inactive = false;

        // Random size (2-6)
        particles[i].size = static_cast<float>(rand() % 5 + 2);
    }
}
```

```cpp
void Application::RenderParticles(Image* framebuffer) {
    for (int i = 0; i < MAX_PARTICLES; ++i) {
        if (!particles[i].inactive) {
            // Draw filled circles based on particle size
            framebuffer->DrawCircle(
                static_cast<int>(particles[i].position.x),
                static_cast<int>(particles[i].position.y),
                static_cast<int>(particles[i].size),
                particles[i].color,    // Border color
                0,                     // No border width
                true,                  // Filled circle
                particles[i].color     // Fill color
            );
        }
    }
}
```

```cpp
void Application::UpdateParticles(float dt) {
    for (int i = 0; i < MAX_PARTICLES; ++i) {
        if (!particles[i].inactive) {
            // Update position
            particles[i].position = particles[i].position + particles[i].velocity * dt;

            // Decrease TTL
            particles[i].ttl -= dt;

            // Mark as inactive if TTL expires
            if (particles[i].ttl <= 0) {
                particles[i].inactive = true;
            }
            // Mark as inactive if the particle leaves the screen bounds (y >= screen height)
            if (particles[i].position.y >= framebuffer.height) {
                particles[i].inactive = true;
            }
        }
        // Respawn inactive particles
        if (particles[i].inactive) {
            // Respawn at a random position
            particles[i].position = Vector2(rand() % framebuffer.width,rand()% framebuffer.height);

            // Randomize downward velocity
            float speed = static_cast<float>(rand() % 200) / 100.0f + 1.5f;
            particles[i].velocity = Vector2(0.0f, speed);

            // Reset properties
            particles[i].ttl = static_cast<float>(rand() % 5000) / 1000.0f + 5.0f;
            particles[i].inactive = false;

            // Randomize color
            particles[i].color = Color(rand() % 256, rand() % 256, rand() % 256);
        }
    }
}
```

Screenshots/Visual Output :