COMPUTER GRAPHICS REPORT

Computer Graphics Lab

Name	**Roll Number**
Shinjan Roy	002310501083
Souradeep De	002310501084
Arko Dasgupta	002310501085
Arjeesh Palai	002310501086

Submission Date:	November 21, 2025
University:	Jadavpur University
Department:	Department of Computer Science and Engineering
Batch:	UG 3rd Year, 1st Semester
Repository:	https://github.com/shadowbeast0/Braking-Bad

Contents

Contributions

- **Shinjan Roy (002310501083):**
  - Circle Drawing Algorithms.
  - Ellipse Drawing Algorithms.
- **Souradeep De (002310501084):**
  - Line Drawing Algorithms.
  - Cubic Bézier Curve.
- **Arko Dasgupta (002310501085):**
  - Line Clipping
  - Polygon Clipping.
- **Arjeesh Palai (002310501086):**
  - Seed-Fill and Scanline Fill.
  - 2D Transformations.

Part 1: Laboratory Assignments

Assignment 1: Line Drawing Algorithms

Problem Statement

Implement a line drawing algorithm to draw lines between two end points in the raster grid using:

- Digital Differential Analyzer (DDA)
- Bresenham's line drawing algorithm

Measure and report the execution time for each algorithm in milliseconds (ms). Test and verify the implementations for line endpoints located in all four quadrants.

Code and Theoretical Explanation

Digital Differential Analyzer (DDA). In my implementation, the function `drawLineDDA()` realises the classical Digital Differential Analyzer algorithm in an incremental form over a raster grid.

Given two end points

$$P_0(x_0, y_0), \qquad P_1(x_1, y_1),$$

the continuous line is described by the slope–intercept form

$$y = mx + b, \qquad m = \frac{y_1 - y_0}{x_1 - x_0}, \qquad b = y_0 - mx_0.$$

Instead of directly evaluating $y = mx + b$ for each integer x (which would involve a multiplication and rounding in every step), DDA works with incremental updates. Let

$$\Delta x = x_1 - x_0, \qquad \Delta y = y_1 - y_0.$$

I choose the number of plotting steps as

$$\text{steps} = \max(|\Delta x|, |\Delta y|),$$

which means that I always step along the dominant direction (either x or y depending on $|m|$). The code fragment

```
int x0 = box0.x(), y0 = box0.y(), x1 = box1.x(), y1 = box1.y();
int dx = x1 - x0, dy = y1 - y0;
int steps = std::max(std::abs(dx), std::abs(dy));
```

implements these Δx, Δy and the choice of steps.

The parametric form of the line can be written as

$$x_k = x_0 + k \cdot \frac{\Delta x}{\text{steps}}, \qquad y_k = y_0 + k \cdot \frac{\Delta y}{\text{steps}}, \qquad k = 0, 1, \ldots, \text{steps}.$$

Hence the incremental updates are

$$x_{k+1} = x_k + x_{\text{inc}}, \qquad y_{k+1} = y_k + y_{\text{inc}},$$

where

$$x_{\text{inc}} = \frac{\Delta x}{\text{steps}}, \qquad y_{\text{inc}} = \frac{\Delta y}{\text{steps}}.$$

This directly matches the implementation:

```
double x_inc = static_cast<double>(dx) / steps;
double y_inc = static_cast<double>(dy) / steps;
double x = x0, y = y0;

for (int i = 0; i <= steps; i++) {
    to_fill.append(QPoint{static_cast<int>(std::round(x)),
                          static_cast<int>(std::round(y))});
    x += x_inc;
    y += y_inc;
}
```

The use of `std::round` corresponds to mapping the real-valued (x_k, y_k) to the nearest pixel centre,

$$(x_k, y_k) \longrightarrow (\text{round}(x_k), \text{round}(y_k)).$$

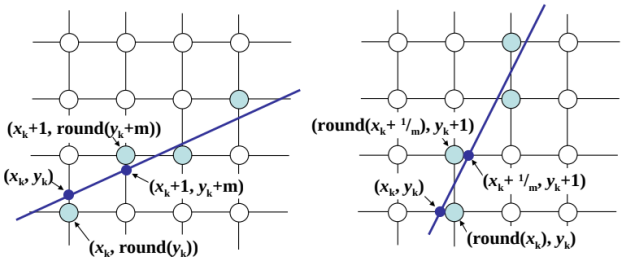
Successive additions of the floating-point increments x_{inc} and y_{inc} may accumulate small numerical deviations, which are then discretised by this rounding step.

The timing code via `QElapsedTimer` is kept strictly outside the core recurrence so that the measured time corresponds purely to the computational part of the algorithm (the incremental updates and rounding), not to the Qt drawing calls.

In summary, my DDA code is a direct implementation of the incremental equations

$$(x_{k+1}, y_{k+1}) = (x_k, y_k) + (x_{\text{inc}}, y_{\text{inc}})$$

with x_{inc} and y_{inc} derived from the geometric slope of the line and the dominant direction of traversal.



Bresenham's Line Drawing Algorithm. The function `drawLineBresenham()` implements an integer-only version of Bresenham's line algorithm, which improves over DDA by eliminating floating-point arithmetic and explicit rounding.

Again I start from the endpoints

$$P_0(x_0, y_0), \qquad P_1(x_1, y_1),$$

and define

$$\Delta x = |x_1 - x_0|, \qquad \Delta y = |y_1 - y_0|.$$

This corresponds to the code

```
int x0 = box0.x(), y0 = box0.y(), x1 = box1.x(), y1 = box1.y();
int dx = std::abs(x1 - x0);
int dy = std::abs(y1 - y0);
```

and matches the usual notation Δx, Δy in the standard derivation.

For the basic case $0 \leq m \leq 1$ and $\Delta x > 0$, Bresenham's algorithm considers, at each step, two candidate pixels:

$$E = (x_k + 1,\, y_k), \qquad NE = (x_k + 1,\, y_k + 1),$$

and chooses the one that is closer to the true line. This decision is encoded in a *decision parameter* p_k which is a scaled form of the difference of line-function values at two points. For a line with slope $m = \Delta y / \Delta x$ one obtains the recurrence

$$p_0 = 2\Delta y - \Delta x,$$

$$p_{k+1} = \begin{cases} p_k + 2\Delta y, & \text{if } p_k < 0 \quad (\text{choose } E), \\ p_k + 2\Delta y - 2\Delta x, & \text{if } p_k \geq 0 \quad (\text{choose } NE). \end{cases}$$

My implementation uses an equivalent but slightly more symmetric formulation that automatically handles all octants. After determining the step directions

$$s_x = \operatorname{sign}(x_1 - x_0), \qquad s_y = \operatorname{sign}(y_1 - y_0),$$

the code initialises

```
int sx = (x1 >= x0) ? 1 : -1;
int sy = (y1 >= y0) ? 1 : -1;

int err = dx - dy;
int x = x0;
int y = y0;
```

Here the variable `err` represents a scaled form of the decision parameter; for the shallow-slope case it is proportional to $(2\Delta y - \Delta x)$ from the classical recurrence. The main loop is

```
while (true) {
    to_fill.append(QPoint{x, y});
    if (x == x1 && y == y1)
        break;

    int e2 = 2 * err;

    if (e2 > -dy) {
        err -= dy;
        x += sx;
    }
    if (e2 < dx) {
        err += dx;
        y += sy;
    }
}
```

Mathematically, the quantity `err` accumulates the difference between the ideal continuous line and the current discrete raster position. At each iteration, I compute

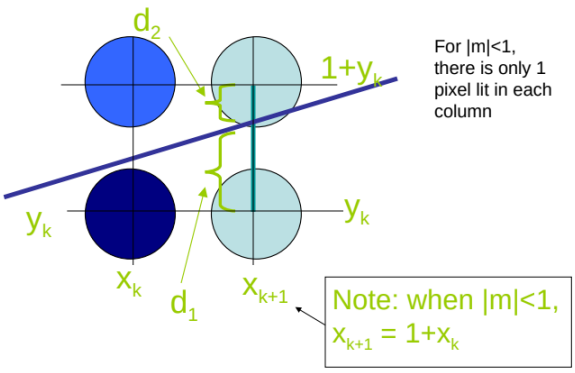
$$e_2 = 2\,\texttt{err},$$

and compare it against $-\Delta y$ and Δx. When $e_2 > -\Delta y$, the accumulated error indicates that I can move one step along the x-direction without deviating too much from the line; algebraically this corresponds to the case where the decision parameter prefers the horizontal (or E-like) advance. I then update

$$\texttt{err} \leftarrow \texttt{err} - \Delta y, \qquad x \leftarrow x + s_x.$$

Similarly, when $e_2 < \Delta x$ the error suggests that I need to move in y as well to stay close to the ideal line, which corresponds to the NE case in the standard formulation:

$$\texttt{err} \leftarrow \texttt{err} + \Delta x, \qquad y \leftarrow y + s_y.$$



If both conditions hold in the same iteration (for steep slopes or negative slopes), both coordinates are updated and a diagonal move is made. This is how the generalised Bresenham algorithm handles all eight octants with a single integer-based scheme.

Crucially, all these updates are integer additions and subtractions; there are no floating-point multiplications or explicit rounding steps. This realises the two key advantages:

- the algorithm is purely incremental (each step uses only the previous state), and
- it uses only integer arithmetic, avoiding the rounding-error accumulation and performance cost present in DDA.

As with DDA, the drawing of pixels is kept outside the timing section; the measured time therefore reflects just the integer decision logic of Bresenham's algorithm.

Code

DDA:

```
1     qint64 MainWindow::drawLineDDA(const QPoint& box0, const QPoint& box1, const
          bool draw) {
2     QVector<QPoint> to_fill;
```

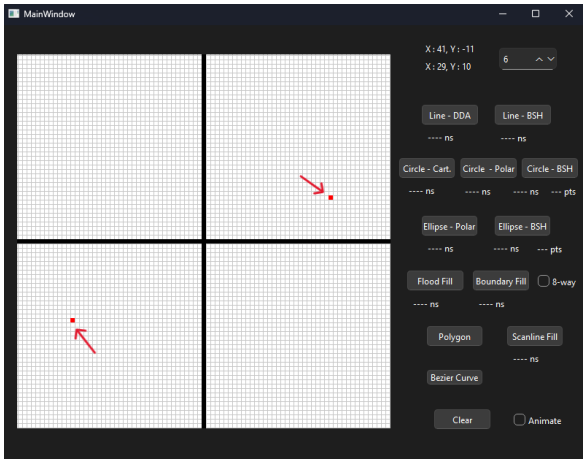
```

    int x0 = box0.x(), y0 = box0.y(), x1 = box1.x(), y1 = box1.y();
    int dx = x1 - x0, dy = y1 - y0;
    int steps = std::max(std::abs(dx), std::abs(dy));
    to_fill.reserve(steps + 1);

    QElapsedTimer timer;
    timer.start();  // Start timing only for computation

    if (steps == 0) {
        to_fill.append(box0);
    } else {
        double x_inc = static_cast<double>(dx) / steps;
        double y_inc = static_cast<double>(dy) / steps;
        double x = x0, y = y0;


        for (int i = 0; i <= steps; i++) {
            to_fill.append(QPoint{static_cast<int>(std::round(x)), static_cast<int
                >(std::round(y))});
            x += x_inc;
            y += y_inc;
        }
    }
    qint64 time = timer.nsecsElapsed();  // Stop timing after computation

    // Perform drawing outside the timed section
    if (draw) addPoints(to_fill, QColor(20, 120, 250));

    return time;
}
```

Bresenham Line Drawing:

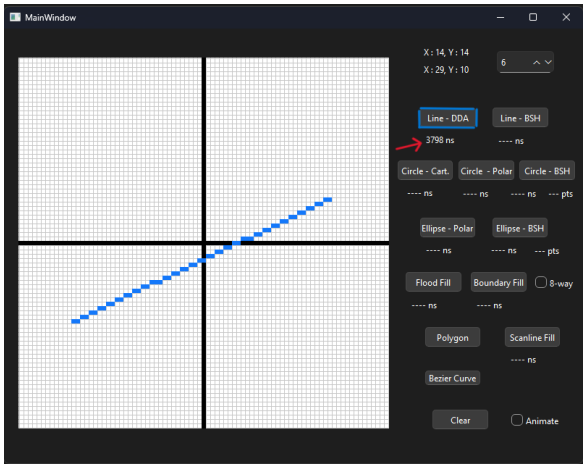
```
    qint64 MainWindow::drawLineBresenham(const QPoint& box0, const QPoint& box1,
        const bool draw) {
    QVector<QPoint> to_fill;

    int x0 = box0.x(), y0 = box0.y(), x1 = box1.x(), y1 = box1.y();
    int dx = std::abs(x1 - x0);
    int dy = std::abs(y1 - y0);
    int max_steps = std::max(dx, dy) + 1;
    to_fill.reserve(max_steps);

    QElapsedTimer timer;
    timer.start();  // Start timing only for computation

    int sx = (x1 >= x0) ? 1 : -1;
    int sy = (y1 >= y0) ? 1 : -1;

    int err = dx - dy;
    int x = x0;
    int y = y0;
```

```
19
20      while (true) {
21          to_fill.append(QPoint{x, y});
22
23          if (x == x1 && y == y1)
24              break;
25
26          int e2 = 2 * err;
27
28          if (e2 > -dy) {
29              err -= dy;
30              x += sx;
31          }
32          if (e2 < dx) {
33              err += dx;
34              y += sy;
35          }
36      }
37
38      qint64 time = timer.nsecsElapsed();  // Stop timing after computation
39
40      // Perform drawing outside the timed section
41      if (draw) addPoints(to_fill, QColor(30, 30, 100));
42
43      return time;
44  }
```

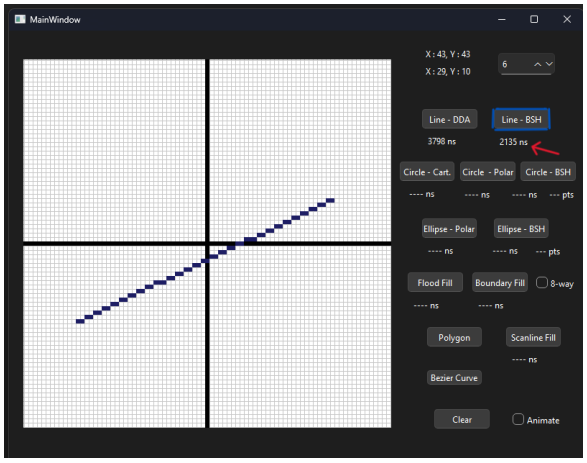
Outputs



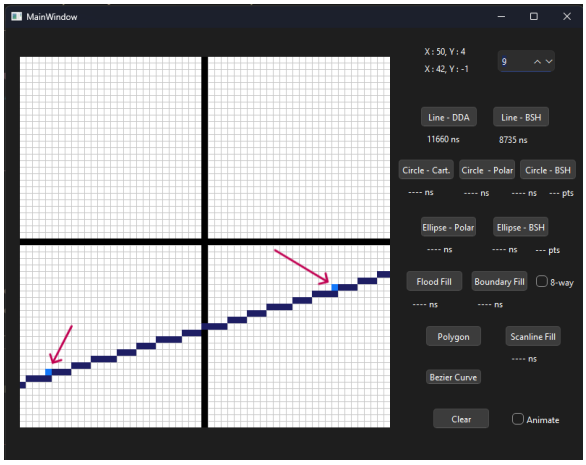
(a) Selecting endpoints for line



(b) DDA Line from selected endpoints



(a) Bresenham Line from selected endpoints



(b) Example where DDA and Bresenham differ

Output Explanation

To compare the behaviour of DDA and Bresenham fairly, I always draw both lines for the *same* pair of endpoints (x_0, y_0) and (x_1, y_1). In the first output screenshot I select these endpoints on the grid. Once the points are fixed, I invoke `drawLineDDA()` and `drawLineBresenham()` with the corresponding integer grid coordinates `box0` and `box1`. Both functions therefore approximate exactly the same underlying continuous line.

In the DDA output, the plotted pixels follow the incremental equations

$$x_{k+1} = x_k + \frac{\Delta x}{\text{steps}}, \qquad y_{k+1} = y_k + \frac{\Delta y}{\text{steps}},$$

with each (x_k, y_k) rounded to the nearest integer grid location. Visually, this produces a staircase pattern that is very close to the ideal straight line, but at the discrete level some decisions are biased by how the floating-point values happen to round.

In the Bresenham output, the same endpoints are connected using the integer decision parameter

described in the previous subsection. Here the next pixel is chosen by testing the accumulated error term rather than by rounding a floating-point position. As a result, the rasterised line is still an 8-connected path between the same endpoints, but the exact sequence of visited pixels is governed by the sign of the decision parameter instead of by $\mathrm{round}(\cdot)$.

To make this difference explicit, I also render an overlay where both DDA and Bresenham are drawn for the same segment using different colours. Most of the plotted pixels coincide, because both algorithms approximate the same ideal line and share the same initial point (x_0, y_0) and final point (x_1, y_1). However, in the middle portion of the segment there are two grid points where the DDA and Bresenham paths diverge: DDA chooses the pixel that is closest to its floating-point sample, whereas Bresenham, following its integer recurrence, steps to a neighbouring pixel on the other side of the ideal line. These non-overlapping pixels highlight the subtle difference between floating-point rounding and integer decision logic.

For performance comparison, each function measures only the core computation time using a `QElapsedTimer`. In both algorithms I start the timer just before the stepping loop and stop it immediately after the sequence of raster points has been generated; the actual drawing (the call to `addPoints`) is performed outside the timed region. The functions return the elapsed time in nanoseconds.

When the push buttons for drawing the lines are clicked, the algorithms are ran 8 times without invoking the drawing command, just to measure the time averaged over 8 iterations of the same algorithm so that variable factors do not affect the time measurement much. After that the actual drawing invocation is done.

In all of my experiments, the reported time for Bresenham is consistently smaller than for DDA for the same endpoints. This matches the theoretical expectation: DDA performs floating-point additions and rounding at every step, whereas Bresenham uses only integer additions, subtractions and comparisons, so its inner loop is lighter and avoids floating-point rounding overhead.

Assignment 2: Circle Drawing Algorithms

Problem Statement

Implement a circle drawing algorithm to draw a circle with a given radius in the raster grid using:

- Polar (parametric) method
- Bresenham's Midpoint circle drawing algorithm

Demonstrate eight-point symmetry of the circle using an animation.
Optional:

- Implement the Cartesian circle drawing method.
- Measure the execution time of each algorithm in milliseconds (ms).

Code and Theoretical Explanation

Polar (parametric) circle method. For a circle of centre $C(x_c, y_c)$ and radius r, the Cartesian equation

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

can be rewritten in polar (parametric) form as

$$x(\theta) = x_c + r\cos\theta, \qquad y(\theta) = y_c + r\sin\theta, \qquad 0 \le \theta < 2\pi.$$

Sampling this parametric curve with a suitable angular step $\Delta\theta$ gives a discrete set of points on the circle. To get roughly one pixel per step along the circumference, I use the approximation

$$\Delta\theta \approx \frac{1}{r},$$

because the arc length associated with a small angle is $r\,\Delta\theta$, so choosing $r\,\Delta\theta \approx 1$ keeps consecutive points about one pixel apart. :contentReference[oaicite:0]index=0

In the function `draw_circle_polar()`:

```
for (double theta = 0; theta <= M_PI/4; theta += 1.0 / radius) {
    int x = std::round(radius * std::cos(theta));
    int y = std::round(radius * std::sin(theta));
    ...
}
```

the loop parameter `theta` corresponds to θ above, and `x`, `y` are the rounded integer versions of $(r\cos\theta, r\sin\theta)$. I only iterate from 0 to $\pi/4$, i.e. over the first octant, and then explicitly exploit the eight-way symmetry of the circle:

$$(x_c \pm x,\ y_c \pm y), \qquad (x_c \pm y,\ y_c \pm x).$$

This is reflected in the eight `push_back` calls in the code, which generate all symmetric points from a single parametric sample. This reduces computation while still drawing the full circle.

The animation function `animate_circle_polar()` uses the same parametric equations, but draws only the current octant point (and its symmetric copies) for the present value of θ. A `QTimer` updates θ over time, giving a visual demonstration of how the eight-way symmetry gradually builds the circle.

Cartesian circle method. Starting from the same circle equation

$$(x - x_c)^2 + (y - y_c)^2 = r^2,$$

we can solve for x in terms of y:

$$x = x_c \pm \sqrt{r^2 - (y - y_c)^2}.$$

If the centre is taken as the origin in the integer grid coordinates ($x_c = 0, y_c = 0$), this simplifies to

$$x = \pm\sqrt{r^2 - y^2}.$$

The Cartesian method steps through integer y values and computes the corresponding x by evaluating this square root. To avoid redundant computation, I again restrict to the first octant and reproduce points in the remaining octants using symmetry.

In `draw_circle_cartesian()` I first compute an integer bound

$$r' = \left\lfloor \frac{r}{\sqrt{2}} \right\rfloor,$$

which corresponds to the intersection of the circle with the line $y = x$ (i.e. the boundary between the first and second octants). Then I iterate over

$$y = 0, 1, \ldots, r',$$

and for each y compute

$$x = \text{round}(\sqrt{r^2 - y^2}).$$

This matches the code:

```
int r = std::round(radius / std::sqrt(2));
...
for (int y = 0; y <= r; ++y) {
    int x = std::round(std::sqrt(radius * radius - y * y));
    ...
}
```

The subsequent eight insertions into `point` again realise the symmetric set

$$(\pm x_c \pm x,\ \pm y_c \pm y), \qquad (\pm x_c \pm y,\ \pm y_c \pm x).$$

Mathematically, this method is straightforward but uses costly square-root operations and floating-point rounding, so it is mainly included as an optional reference implementation.

The animation routine `animate_circle_cartesian()` runs the same recurrence for a single y value per timer tick and draws only that ring of eight symmetric pixels. This visually illustrates how the circle is traced as y moves from 0 to r'.

Bresenham midpoint circle algorithm. The Bresenham midpoint circle algorithm is an incremental, integer-only technique based on evaluating the implicit circle function

$$f_{\text{circle}}(x, y) = x^2 + y^2 - r^2.$$

For any grid point (x, y), the sign of $f_{\text{circle}}(x, y)$ classifies it as:

$f_{\text{circle}}(x, y) < 0 \Rightarrow$ inside the circle, $f_{\text{circle}}(x, y) = 0 \Rightarrow$ on the circle, $f_{\text{circle}}(x, y) > 0 \Rightarrow$ outside the circle.

The algorithm walks along the circle in the first octant with integer coordinates, maintaining a decision parameter p_k that approximates f_{circle} at the midpoint between two candidate pixels. When the current point is (x_k, y_k), the next x-coordinate is incremented by one, and there are two choices for the next pixel:

$$E = (x_k + 1,\ y_k), \qquad SE = (x_k + 1,\ y_k - 1).$$

The midpoint between these two candidates is

$$M = \left(x_k + 1,\ y_k - \tfrac{1}{2}\right),$$

and the decision parameter is defined as

$$p_k = f_{\text{circle}}(M) = \left(x_k + 1\right)^2 + \left(y_k - \tfrac{1}{2}\right)^2 - r^2.$$

If $p_k < 0$, the midpoint is inside the circle and E is closer to the ideal circle, so the next point is $(x_{k+1}, y_{k+1}) = (x_k + 1, y_k)$. If $p_k \geq 0$, the midpoint lies on or outside the circle and the better choice is $(x_{k+1}, y_{k+1}) = (x_k + 1, y_k - 1)$.

Algebraic manipulation yields an integer recurrence for the decision parameter. Starting with

$$x_0 = 0, \quad y_0 = r, \quad p_0 = 1 - r,$$

one can derive

$$p_{k+1} = \begin{cases} p_k + 2x_k + 3, & \text{if } p_k < 0 \quad \text{(E step)}, \\ p_k + 2x_k - 2y_k + 5, & \text{if } p_k \geq 0 \quad \text{(SE step)}. \end{cases}$$

Multiplying everything by 2 gives a form that matches the implementation constants used in my code. :contentReference[oaicite:1]index=1

In `draw_circle_bresenham()` I keep the circle centre (x_c, y_c) separate and iterate in a local coordinate system:

```
int x = 0, y = radius;
int d = 3 - 2 * radius;

while (x <= y) {
    point.push_back(QPoint(cx + x, cy + y));
    ...
    if (d < 0)
        d += 4 * x + 6;
```

```
9       else {
10          y--;
11          d += 4 * (x - y) + 10;
12      }
13      x++;
14  }
```

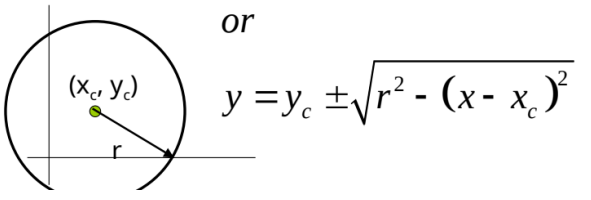
Here `d` is a scaled version of the decision parameter p_k. The initial value `d = 3 - 2r` is proportional to $4p_0 = 4(1 - r)$ plus a constant. The updates

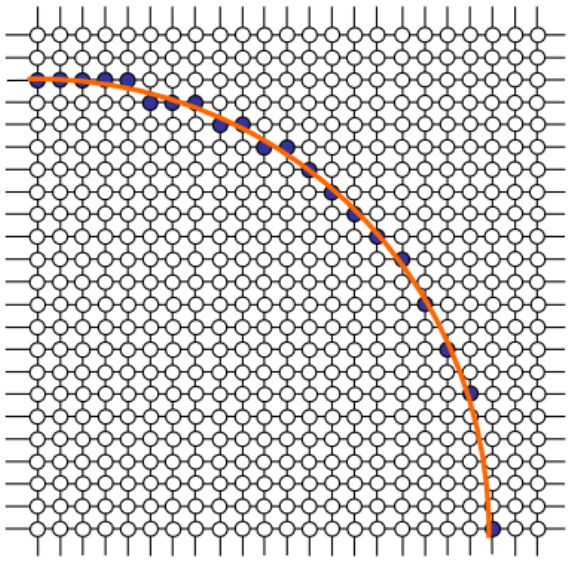
$$d \leftarrow d + 4x + 6 \quad \text{or} \quad d \leftarrow d + 4(x - y) + 10$$

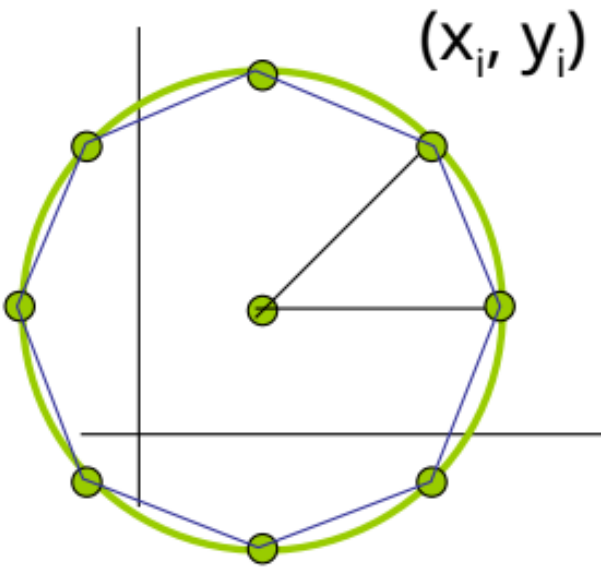
are the integer recurrences corresponding to the E and SE cases. Only additions and subtractions on integers are used; there are no multiplications by non-integer constants, no square roots, and no trigonometric functions.

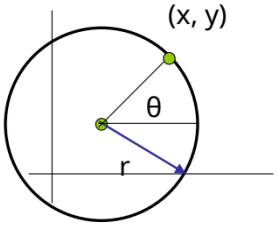
As in the other methods, I compute points only for the first octant where $0 \leq x \leq y$, and for each (x, y) I generate all eight symmetric points around the circle by adding and subtracting x and y to the centre coordinates. This symmetry is also used in `animate_circle_bresenham()`, which updates `x`, `y`, and `d` one step at a time, drawing eight new pixels per timer tick and thereby animating the growth of the circle.

In the non-animated path of `draw_circle()`, I benchmark the polar and Bresenham methods by repeating each algorithm 100 times and measuring the accumulated nanoseconds with `QElapsedTimer`. Dividing by 100 gives an average time per run, which is displayed in the labels `polarcircletime` and `bresenhamcircletime`. As expected from the absence of trigonometric and square-root operations, the Bresenham midpoint method consistently completes faster than the polar and Cartesian methods.





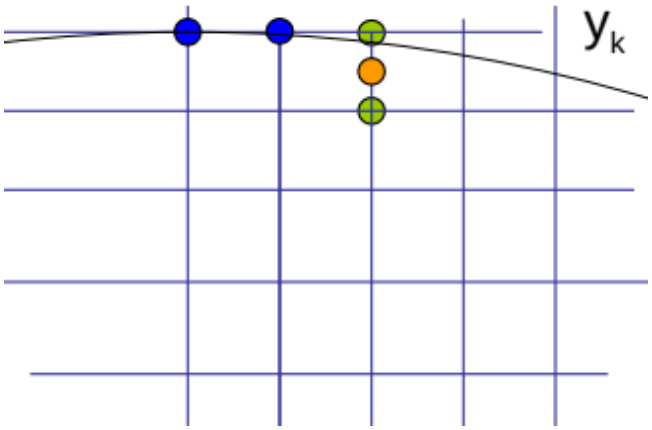




$$x = x_c + r\cos\theta$$
$$y = y_c + r\sin\theta$$

$$\Delta s = r\theta$$
$$\Delta s \approx 1$$
$$\Delta\theta \approx \frac{1}{r}$$

Simple method: plot directly from parametric equations



These images together support the theoretical discussion of the polar, Cartesian and Bresenham circle algorithms and make the behaviour of my implementation visually clear.

Code

circle.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

CONFIG += c++17

#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000

SOURCES += \
    main.cpp \
    mainwindow.cpp \
    my_label.cpp

```

```
HEADERS += \
    mainwindow.h \
    my_label.h

FORMS += \
    mainwindow.ui

qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QPoint>
#include <QSet>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void Mouse_Pressed();
    void showMousePosition(QPoint&);
    void on_clear_clicked();
    void on_draw_line_clicked();
    void draw_grid();
    std::vector<QPoint> draw_line_bresenham(QPoint, QPoint);

    void on_spinBox_valueChanged(int);
    void repaint();

    void paint(std::vector<QPoint>, QColor);
    void paint(QPoint, QColor, QPainter&);

    void on_draw_circle_clicked();
    void draw_circle(QPoint, int);
    std::vector<QPoint> draw_circle_polar(QPoint, int);
    std::vector<QPoint> draw_circle_bresenham(QPoint, int);
    std::vector<QPoint> draw_circle_cartesian(QPoint, int);

    void animate_circle_polar(QPoint, int, double);
```

```
41      void animate_circle_cartesian(QPoint, int, int);
42      void animate_circle_bresenham(QPoint, int&, int&, int&);
43
44      void on_circle_type_currentIndexChanged(int);
45
46      void on_undo_clicked();
47
48  private:
49      Ui::MainWindow *ui;
50      void addPoint(int x, int y, int c = 1);
51
52      QPoint lastPoint1, lastPoint2;
53      int sc_x, sc_y;
54      int org_x, org_y;
55      int width, height;
56
57      std::vector<std::pair<std::vector<QPoint>, QColor>> colormap;
58      int gap;
59      bool acircle;
60      int fcircle;
61  };
62
63  #endif // MAINWINDOW_H
```

my_label.h

```
1   #ifndef MY_LABEL_H
2   #define MY_LABEL_H
3
4   #include <QLabel>
5   #include <QMouseEvent>
6
7   class my_label : public QLabel
8   {
9       Q_OBJECT
10  public:
11      explicit my_label(QWidget *parent = nullptr);
12      int x, y;
13
14  protected:
15      void mouseMoveEvent(QMouseEvent *ev);
16      void mousePressEvent(QMouseEvent *ev);
17
18  signals:
19      void sendMousePosition(QPoint&);
20      void Mouse_Pos();
21  };
22
23  #endif // MY_LABEL_H
```

main.cpp

```
1   #include "mainwindow.h"
2
```

```
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

my_label.cpp

```
#include "my_label.h"

my_label::my_label(QWidget *parent) : QLabel(parent)
{
    this->setMouseTracking(true);
}

void my_label::mouseMoveEvent(QMouseEvent *ev)
{
    QPoint pos = ev->pos();
    if (pos.x() >= 0 && pos.y() >= 0 && pos.x() < this->width() && pos.y() < this
        ->height()) {
        emit sendMousePosition(pos);
    }
}

void my_label::mousePressEvent(QMouseEvent *ev)
{
    if (ev->button() == Qt::LeftButton) {
        x = ev->x();
        y = ev->y();
        emit Mouse_Pos();
    }
}
```

mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QPixmap>
#include <QImage>
#include <QPainter>
#include <QDebug>
#include <QTimer>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    lastPoint1 = QPoint(-100000, -100000);
    lastPoint2 = QPoint(-100000, -100000);
```

```
16
17      connect(ui->frame, SIGNAL(Mouse_Pos()), this, SLOT(Mouse_Pressed()));
18      connect(ui->frame, SIGNAL(sendMousePosition(QPoint&)), this, SLOT(
            showMousePosition(QPoint&)));
19
20      gap = 10;
21      fcircle = 0;
22      acircle = true;
23
24      width = 750;
25      height = 450;
26
27      draw_grid();
28      ui->polarcircletime->setText("Polar: 0 ns");
29      ui->bresenhamcircletime->setText("Bresenham: 0 ns");
30
31      ui->filled->setText("Filled: 0 pixels");
32
33      ui->undo->setEnabled(false);
34  }
35
36  MainWindow::~MainWindow()
37  {
38      delete ui;
39  }
40
41  void MainWindow::showMousePosition(QPoint &pos)
42  {
43      sc_x = pos.x();
44      sc_y = pos.y();
45      ui->mouse_movement->setText("X : " + QString::number((sc_x-sc_x%gap-((width/2)
            -(width/2)%gap))/gap) + ", Y : " + QString::number(-(sc_y-sc_y%gap-((
            height/2)-(height/2)%gap))/gap));
46  }
47
48  void MainWindow::Mouse_Pressed()
49  {
50      org_x = sc_x;
51      org_y = sc_y;
52
53      int x = (sc_x-sc_x%gap-((width/2)-(width/2)%gap))/gap;
54      int y = (sc_y-sc_y%gap-((height/2)-(height/2)%gap))/gap;
55
56      ui->mouse_pressed->setText("X : " + QString::number(x) + ", Y : " + QString::
            number(y));
57
58      lastPoint1 = lastPoint2;
59
60      lastPoint2 = QPoint(x, y);
61  }
62
63  void MainWindow::on_clear_clicked()
```

```
64 {
65     lastPoint1 = lastPoint2 = QPoint(-100000, -100000);
66     vertices.clear();
67     edges.clear();
68     colormap.clear();
69     draw_grid();
70     ui->polarcircletime->setText("Polar: 0 ns");
71     ui->bresenhamcircletime->setText("Bresenham: 0 ns");
72 
73     ui->filled->setText("Filled: 0 pixels");
74 }
75 
76 
77 void MainWindow::draw_grid()
78 {
79     QPixmap pix(ui->frame->width(), ui->frame->height());
80     pix.fill(QColor(255, 255, 255));
81     ui->frame->setPixmap(pix);
82 
83     QPixmap pm = ui->frame->pixmap();
84     QPainter painter(&pm);
85     painter.setPen(QPen(QColor(200, 200, 200)));
86     for(int i=0; i<=width; i+=gap) painter.drawLine(QPoint(i,0), QPoint(i, height)
        );
87     for(int i=0; i<=height; i+=gap) painter.drawLine(QPoint(0, i), QPoint(width, i
        ));
88 
89     painter.end();
90     ui->frame->setPixmap(pm);
91 
92     paint(draw_line_bresenham(QPoint(-100000, 0), QPoint(100000, 0)), QColor(0, 0,
         0));
93     paint(draw_line_bresenham(QPoint(0, -100000), QPoint(0, 100000)), QColor(0, 0,
         0));
94 }
95 
96 
97 void MainWindow::paint(std::vector<QPoint> points, QColor color){
98     QPixmap pm = ui->frame->pixmap();
99     QPainter painter(&pm);
100     for(QPoint p:points){
101         int x = p.x()*gap, y = p.y()*gap;
102         painter.fillRect(QRect(x+((width/2) - (width/2)%gap), y+((height/2) - (
            height/2)%gap), gap, gap), color);
103     }
104     painter.end();
105     ui->frame->setPixmap(pm);
106     ui->filled->setText("Filled: " + QString::number(points.size()) + " pixels");
107 }
108 
109 void MainWindow::paint(QPoint point, QColor color, QPainter& painter){
110     int x = point.x()*gap, y = point.y()*gap;
```

```
111     painter.fillRect(QRect(x+((width/2) - (width/2)%gap), y+((height/2) - (height
            /2)%gap), gap, gap), color);
112 }
113
114
115 std::vector<QPoint> MainWindow::draw_line_bresenham(QPoint a, QPoint b) {
116     int x1 = a.x(), y1 = a.y();
117     int x2 = b.x(), y2 = b.y();
118
119     int dx = abs(x2 - x1);
120     int dy = abs(y2 - y1);
121
122     int sx = (x1 < x2) ? 1 : -1;
123     int sy = (y1 < y2) ? 1 : -1;
124
125     int err = dx - dy;
126
127     std::vector<QPoint> point;
128
129     while (true) {
130         point.push_back(QPoint(x1, y1));
131
132         if (x1 == x2 && y1 == y2) break;
133
134         int e2 = 2 * err;
135         if (e2 > -dy) { err -= dy; x1 += sx; }
136         if (e2 < dx)  { err += dx; y1 += sy; }
137     }
138
139     return point;
140 }
141
142 void MainWindow::on_spinBox_valueChanged(int value)
143 {
144     gap = value;
145     repaint();
146 }
147
148
149 void MainWindow::repaint(){
150     draw_grid();
151     QPixmap pm = ui->frame->pixmap();
152     QPainter painter(&pm);
153     for(const auto& pair : colormap){
154         for(QPoint p : pair.first){
155             paint(p, pair.second, painter);
156         }
157     }
158     painter.end();
159     ui->frame->setPixmap(pm);
160 }
161
```

```
162
163 void MainWindow::on_draw_circle_clicked()
164 {
165     int cx = lastPoint1.x();
166     int cy = lastPoint1.y();
167     int dx = lastPoint2.x() - lastPoint1.x();
168     int dy = lastPoint2.y() - lastPoint1.y();
169     int radius = std::round(std::sqrt(dx * dx + dy * dy));
170
171     QPoint centre = QPoint(cx, cy);
172
173     draw_circle(centre, radius);
174 }
175
176
177 void MainWindow::draw_circle(QPoint centre, int radius){
178     if(acircle && ui->animate_circle->isChecked()){
179         if(fcircle == 0){
180             auto theta = std::make_shared<double>(0);
181             QTimer *timer = new QTimer(this);
182             connect(timer, &QTimer::timeout, this, [this, timer, theta, centre,
                    radius]() {
183                 if (*theta > M_PI/4) {
184                     timer->stop();
185                     timer->deleteLater();
186                     ui->undo->setEnabled(true);
187                 }
188                 animate_circle_polar(centre, radius, *theta);
189                 *theta += 1.0/radius;
190             });
191             timer->start(100);
192         }
193         else if(fcircle == 1){
194             QTimer *timer = new QTimer(this);
195             auto x = std::make_shared<int>(0);
196             auto y = std::make_shared<int>(radius);
197             auto d = std::make_shared<int>(3 - 2*radius);
198             connect(timer, &QTimer::timeout, this, [this, timer, centre, x, y, d
                    ]() {
199                 if (*x > *y) {
200                     timer->stop();
201                     timer->deleteLater();
202                 }
203                 animate_circle_bresenham(centre, *x, *y, *d);
204             });
205             timer->start(100);
206         }
207         else{
208             int r = std::round(radius / std::sqrt(2));
209             QTimer *timer = new QTimer(this);
210             auto y = std::make_shared<int>(0);
211             connect(timer, &QTimer::timeout, this, [this, timer, y, r, centre,
```

```
                radius]() {
212                 if (*y > r) {
213                     timer->stop();
214                     timer->deleteLater();
215                     ui->undo->setEnabled(true);
216                 }
217                 animate_circle_cartesian(centre, radius, *y);
218                 (*y)++;
219             });
220             timer->start(100);
221         }
222         std::vector<QPoint> point = fcircle ? draw_circle_cartesian(centre, radius
                ) : draw_circle_bresenham(centre, radius);
223         colormap.push_back(std::pair(point, fcircle? QColor(150, 0, 0) : QColor
                (255, 150, 0)));
224         ui->undo->setEnabled(true);
225     }
226
227
228     else{
229         QElapsedTimer timer;
230         qint64 time = 0;
231         std::vector<QPoint> point;
232         if(fcircle == 0){
233             for(int i=0; i<100; i++){
234                 timer.start();
235                 point = draw_circle_polar(centre, radius);
236                 time += timer.nsecsElapsed();
237             }
238             ui->polarcircletime->setText("Polar: " + QString::number(time/100) + "
                     ns");
239         }
240         else if(fcircle == 1){
241             for(int i=0; i<100; i++){
242                 timer.start();
243                 point = draw_circle_bresenham(centre, radius);
244                 time += timer.nsecsElapsed();
245             }
246             ui->bresenhamcircletime->setText("Bresenham: " + QString::number(time
                    /100) + " ns");
247         }
248         else{
249             point = draw_circle_cartesian(centre, radius);
250         }
251         colormap.push_back(std::pair(point, fcircle==0? QColor(150, 0, 0) :
                fcircle==1? QColor(255, 150, 0) : QColor(255, 200, 0)));
252         paint(point, fcircle==0? QColor(150, 0, 0) : fcircle==1? QColor(255, 150,
                0) : QColor(255, 200, 0));
253     }
254 }
255
256
```

```
257 std::vector<QPoint> MainWindow::draw_circle_polar(QPoint centre, int radius){
258     int cx = centre.x();
259     int cy = centre.y();
260 
261     std::vector<QPoint> point;
262     for(double theta = 0; theta <= M_PI/4; theta += 1.0/radius){
263         int x = std::round(radius * cos(theta)), y = std::round(radius * sin(theta
            ));
264         point.push_back(QPoint(cx + x, cy + y));
265         point.push_back(QPoint(cx - x, cy + y));
266         point.push_back(QPoint(cx + x, cy - y));
267         point.push_back(QPoint(cx - x, cy - y));
268         point.push_back(QPoint(cx + y, cy + x));
269         point.push_back(QPoint(cx - y, cy + x));
270         point.push_back(QPoint(cx + y, cy - x));
271         point.push_back(QPoint(cx - y, cy - x));
272     }
273 
274     return point;
275 }
276 
277 
278 std::vector<QPoint> MainWindow::draw_circle_cartesian(QPoint centre, int radius){
279     int r = std::round(radius / std::sqrt(2));
280 
281     int cx = centre.x();
282     int cy = centre.y();
283 
284     std::vector<QPoint> point;
285     for(int y = 0; y<=r; y++){
286         int x = std::round(std::sqrt(radius * radius - y * y));
287         point.push_back(QPoint(cx + x, cy + y));
288         point.push_back(QPoint(cx - x, cy + y));
289         point.push_back(QPoint(cx + x, cy - y));
290         point.push_back(QPoint(cx - x, cy - y));
291         point.push_back(QPoint(cx + y, cy + x));
292         point.push_back(QPoint(cx - y, cy + x));
293         point.push_back(QPoint(cx + y, cy - x));
294         point.push_back(QPoint(cx - y, cy - x));
295     }
296 
297     return point;
298 }
299 
300 std::vector<QPoint> MainWindow::draw_circle_bresenham(QPoint centre, int radius){
301     int cx = centre.x();
302     int cy = centre.y();
303 
304     int x = 0, y = radius;
305     int d = 3 - 2 * radius;
306 
307     std::vector<QPoint> point;
```

```
308
309     while(x <= y){
310         point.push_back(QPoint(cx + x, cy + y));
311         point.push_back(QPoint(cx - x, cy + y));
312         point.push_back(QPoint(cx + x, cy - y));
313         point.push_back(QPoint(cx - x, cy - y));
314         point.push_back(QPoint(cx + y, cy + x));
315         point.push_back(QPoint(cx - y, cy + x));
316         point.push_back(QPoint(cx + y, cy - x));
317         point.push_back(QPoint(cx - y, cy - x));
318         if(d < 0) d += 4 * x + 6;
319         else{
320             y--;
321             d += 4 * (x - y) + 10;
322         }
323         x++;
324     }
325
326     return point;
327 }
328
329 void MainWindow::animate_circle_polar(QPoint centre, int radius, double theta){
330     int cx = centre.x();
331     int cy = centre.y();
332
333     QPixmap pm = ui->frame->pixmap();
334     QPainter painter(&pm);
335
336     int x = std::round(radius * cos(theta)), y = std::round(radius * sin(theta));
337
338     paint(QPoint(cx + x, cy + y), QColor(150, 0, 0), painter);
339     paint(QPoint(cx - x, cy + y), QColor(150, 0, 0), painter);
340     paint(QPoint(cx + x, cy - y), QColor(150, 0, 0), painter);
341     paint(QPoint(cx - x, cy - y), QColor(150, 0, 0), painter);
342     paint(QPoint(cx + y, cy + x), QColor(150, 0, 0), painter);
343     paint(QPoint(cx - y, cy + x), QColor(150, 0, 0), painter);
344     paint(QPoint(cx + y, cy - x), QColor(150, 0, 0), painter);
345     paint(QPoint(cx - y, cy - x), QColor(150, 0, 0), painter);
346     painter.end();
347     ui->frame->setPixmap(pm);
348 }
349
350 void MainWindow::animate_circle_cartesian(QPoint centre, int radius, int y){
351     int cx = centre.x();
352     int cy = centre.y();
353
354     QPixmap pm = ui->frame->pixmap();
355     QPainter painter(&pm);
356     int x = std::round(std::sqrt(radius * radius - y * y));
357     paint(QPoint(cx + x, cy + y), QColor(255, 200, 0), painter);
358     paint(QPoint(cx - x, cy + y), QColor(255, 200, 0), painter);
359     paint(QPoint(cx + x, cy - y), QColor(255, 200, 0), painter);
```

```
360     paint(QPoint(cx - x, cy - y), QColor(255, 200, 0), painter);
361     paint(QPoint(cx + y, cy + x), QColor(255, 200, 0), painter);
362     paint(QPoint(cx - y, cy + x), QColor(255, 200, 0), painter);
363     paint(QPoint(cx + y, cy - x), QColor(255, 200, 0), painter);
364     paint(QPoint(cx - y, cy - x), QColor(255, 200, 0), painter);
365     painter.end();
366     ui->frame->setPixmap(pm);
367 }
368
369 void MainWindow::animate_circle_bresenham(QPoint centre, int& x, int& y, int& d){
370     int cx = centre.x();
371     int cy = centre.y();
372
373     QPixmap pm = ui->frame->pixmap();
374     QPainter painter(&pm);
375     paint(QPoint(cx + x, cy + y), QColor(255, 150, 0), painter);
376     paint(QPoint(cx - x, cy + y), QColor(255, 150, 0), painter);
377     paint(QPoint(cx + x, cy - y), QColor(255, 150, 0), painter);
378     paint(QPoint(cx - x, cy - y), QColor(255, 150, 0), painter);
379     paint(QPoint(cx + y, cy + x), QColor(255, 150, 0), painter);
380     paint(QPoint(cx - y, cy + x), QColor(255, 150, 0), painter);
381     paint(QPoint(cx + y, cy - x), QColor(255, 150, 0), painter);
382     paint(QPoint(cx - y, cy - x), QColor(255, 150, 0), painter);
383     painter.end();
384     ui->frame->setPixmap(pm);
385     if(d < 0) d += 4 * x + 6;
386     else{
387         y--;
388         d += 4 * (x - y) + 10;
389     }
390     x++;
391 }
392
393 void MainWindow::on_circle_type_currentIndexChanged(int index)
394 {
395     fcircle = index;
396 }
397
398
399 void MainWindow::on_undo_clicked()
400 {
401     colormap.pop_back();
402     if(colormap.empty()) ui->undo->setEnabled(false);
403     repaint();
404 }
```

Outputs

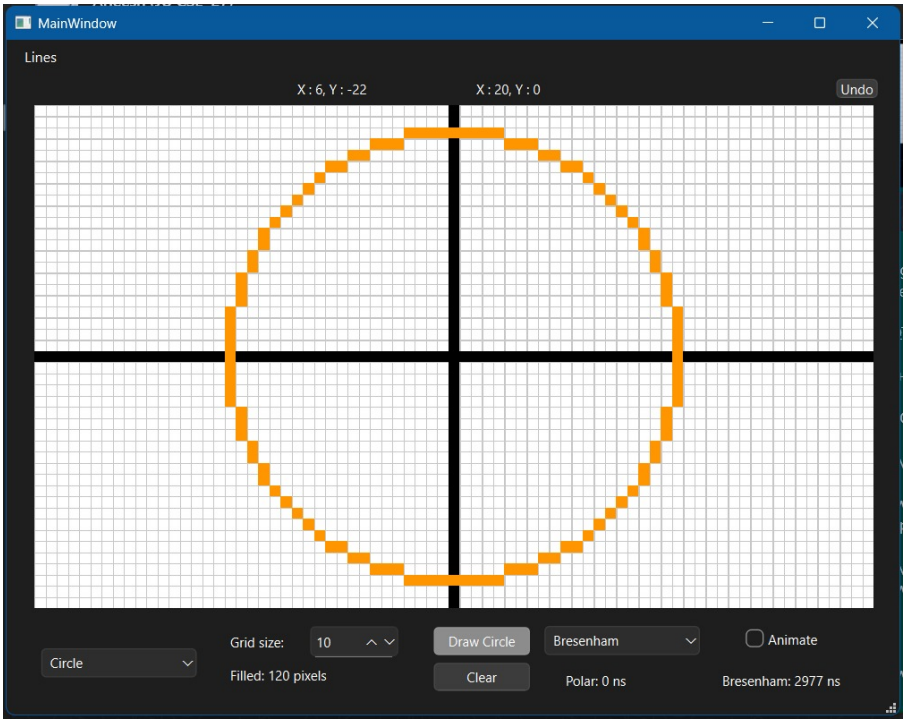


Figure 3: bresenham_circle

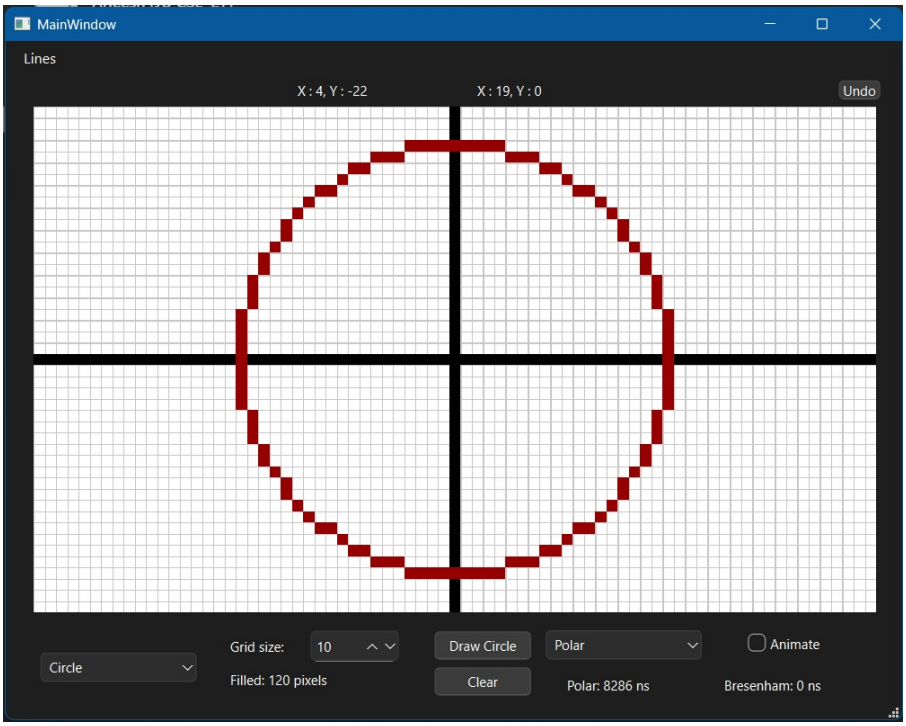


Figure 4: polar_circle

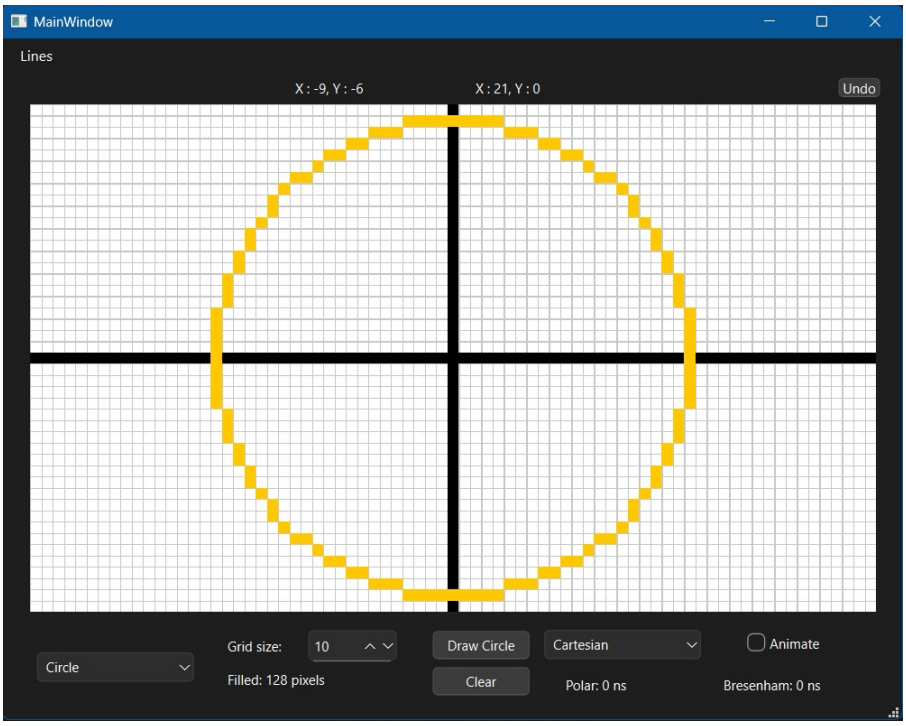


Figure 5: cartesian_circle

Output Explanation

For validating the three circle algorithms, I fixed the circle centre at the origin of the grid and chose a single radius r. Using the same two mouse clicks (first for the centre, second for a point on the circumference), the program first computes the integer radius

$$r = \left\lfloor \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \right\rceil$$

and then draws three circles of radius r using the polar, Cartesian and Bresenham midpoint methods.

Visually, all three circles are centred at the origin and exhibit correct eight-way symmetry. Because of different rounding strategies, a few pixels differ between methods, but the overall shapes are almost indistinguishable at this resolution. The polar and Cartesian methods both rely on floating-point $\cos\theta$, $\sin\theta$ or $\sqrt{\cdot}$ evaluations, while the Bresenham circle uses only integer additions and subtractions driven by the midpoint decision parameter.

To quantify this difference, I measured the execution time of the polar and Bresenham implementations using the built-in benchmarking in `draw_circle()`. For each method, the circle is generated 100 times in a tight loop and the elapsed time in nanoseconds is accumulated with `QElapsedTimer`; the average

$$T_{\text{avg}} = \frac{1}{100} \sum_{i=1}^{100} T_i$$

is then displayed in the UI as "Polar: 8286 ns" and "Bresenham: 2977 ns". Across repeated runs for the same radius, the reported average time for the Bresenham circle is consistently lower than that of the polar circle. This empirically confirms the theoretical expectation: avoiding trigonometric and square-root computations and using purely incremental integer arithmetic makes the Bresenham midpoint algorithm noticeably faster, while still producing a visually accurate rasterisation of the circle.

Assignment 3: Ellipse Drawing Algorithms

Problem Statement

Implement an ellipse drawing algorithm to draw an ellipse with given radii in the raster grid using:

- Polar (parametric) method
- Bresenham's Midpoint ellipse drawing algorithm

Measure and report the execution time for each algorithm in nanoseconds (ns).

Code and Theoretical Explanation

Polar (parametric) ellipse method. In my implementation, the function `draw_ellipse_polar()` realises the standard parametric form of an axis–aligned ellipse with centre

$$C = (x_c, y_c), \qquad \text{semi–major axis } a, \text{ semi–minor axis } b.$$

The continuous ellipse satisfies

$$\frac{(x - x_c)^2}{a^2} + \frac{(y - y_c)^2}{b^2} = 1$$

and can be written parametrically as

$$x(\theta) = x_c + a\cos\theta, \qquad y(\theta) = y_c + b\sin\theta, \qquad 0 \le \theta < 2\pi.$$

I first extract the integer centre coordinates and prepare a container of raster points:

```
int cx = centre.x();
int cy = centre.y();

std::vector<QPoint> point;
```

Instead of using a fixed angular increment, I adapt the step in θ so that the points stay roughly one pixel apart along the curve. For the parametric curve $\mathbf{r}(\theta) = (a\cos\theta, b\sin\theta)$, the speed is

$$\left\| \frac{d\mathbf{r}}{d\theta} \right\| = \sqrt{(-a\sin\theta)^2 + (b\cos\theta)^2} = \sqrt{a^2\sin^2\theta + b^2\cos^2\theta}.$$

If I target an arc–length step $\Delta s \approx 0.5$ pixels, then

$$\Delta\theta \approx \frac{\Delta s}{\sqrt{a^2\cos^2\theta + b^2\sin^2\theta}}.$$

This is encoded directly as

```
for (double theta = 0; theta < M_PI/2;
     theta += 0.5 / std::sqrt(a*a*std::cos(theta)*std::cos(theta) +
                              b*b*std::sin(theta)*std::sin(theta))) {
```

which samples only the first quadrant $0 \le \theta < \pi/2$. At each step I evaluate the parametric equations and round to the nearest pixel:

$$x = \lfloor x_c + a\cos\theta + 0.5 \rfloor, \qquad y = \lfloor y_c + b\sin\theta + 0.5 \rfloor.$$

In code,

```
    int x = std::round(a * std::cos(theta));
    int y = std::round(b * std::sin(theta));
```

The symmetry of the ellipse with respect to both axes allows me to generate all four quadrants from one computed point (x, y):

$$(x_c \pm x,\, y_c \pm y).$$

This four–way symmetry is implemented by pushing four points per sample:

```
    point.push_back(QPoint(cx + x, cy + y));
    point.push_back(QPoint(cx - x, cy + y));
    point.push_back(QPoint(cx + x, cy - y));
    point.push_back(QPoint(cx - x, cy - y));
```

Overall, `draw_ellipse_polar()` is therefore a direct discretisation of the continuous parametric equations $x(\theta), y(\theta)$, with an adaptive angular step chosen from the analytic expression of the ellipse's local speed and four–way symmetry used to cover the entire curve efficiently.

Cartesian ellipse method. The Cartesian method starts from the implicit equation of an ellipse with centre $C = (x_c, y_c)$:

$$\frac{(x - x_c)^2}{a^2} + \frac{(y - y_c)^2}{b^2} = 1.$$

Solving for y in terms of x gives

$$y(x) = \pm b\sqrt{1 - \frac{(x - x_c)^2}{a^2}},$$

and solving for x in terms of y gives

$$x(y) = \pm a\sqrt{1 - \frac{(y - y_c)^2}{b^2}}.$$

In my code I work in a local coordinate system with the centre at the origin and then shift by (x_c, y_c) when storing the points. The function `draw_ellipse_cartesian()` reflects exactly these formulas:

```
int cx = centre.x();
int cy = centre.y();

std::vector<QPoint> point;
for (int x = 0; x <= a; x++) {
    int y = std::round(b * std::sqrt(1 - (float)(x * x) / (a * a)));
    point.push_back(QPoint(cx + x, cy + y));
    point.push_back(QPoint(cx - x, cy + y));
    point.push_back(QPoint(cx + x, cy - y));
    point.push_back(QPoint(cx - x, cy - y));
}
```

For each integer x between 0 and a, I compute

$$y = \text{round}\left(b\sqrt{1 - \frac{x^2}{a^2}}\right),$$

and then replicate (x, y) to all four quadrants via $(\pm x, \pm y)$.

However, sampling only in x would undersample the portions of the ellipse where the curve is steep (near the top and bottom). To avoid gaps, I add a second pass where I sample in y and solve for x:

```
for (int y = 0; y <= b; y++) {
    int x = std::round(a * std::sqrt(1 - (float)(y * y) / (b * b)));
    point.push_back(QPoint(cx + x, cy + y));
    point.push_back(QPoint(cx - x, cy + y));
    point.push_back(QPoint(cx + x, cy - y));
    point.push_back(QPoint(cx - x, cy - y));
}
```

This corresponds to using

$$x = \text{round}\left(a\sqrt{1 - \frac{y^2}{b^2}}\right)$$

and again exploiting four–way symmetry. Thus the Cartesian method in my code numerically evaluates the explicit functions $y(x)$ and $x(y)$ derived from the ellipse equation, combining both to obtain a visually smooth rasterisation.

Bresenham (midpoint) ellipse algorithm. The midpoint ellipse algorithm is an extension of the midpoint circle method. It uses the implicit form

$$F(x, y) = b^2x^2 + a^2y^2 - a^2b^2 = 0$$

of the ellipse. For any integer point (x, y),

$F(x, y) < 0 \Rightarrow$ point inside ellipse, $F(x, y) = 0 \Rightarrow$ point on ellipse, $F(x, y) > 0 \Rightarrow$ point outside ellipse.

The method starts in the upper–middle point $(0, b)$ in the first quadrant and walks around the quadrant using only increments of 1 in x and/or y, deciding at each step which neighbouring pixel is closer to the true ellipse. This decision is based on evaluating F at a midpoint between candidate pixels, which leads to simple integer recurrences.

Region 1: $\boldsymbol{dx < dy}$.

In the first region, the slope of the ellipse satisfies $|dy/dx| \leq 1$ and we primarily increment x. The initial values in my code are

```
int x = 0, y = b;
int d1 = b*b - a*a*b + a*a/4;
int dx = 2 * b*b * x;
int dy = 2 * a*a * y;
```

The expression

$$d_1 = b^2 - a^2b + \frac{a^2}{4}$$

is the value of the decision function at the midpoint between the first two candidate pixels:

$$d_1 = F\left(x_0 + 1,\ y_0 - \tfrac{1}{2}\right) = b^2(x_0 + 1)^2 + a^2\left(y_0 - \tfrac{1}{2}\right)^2 - a^2b^2$$

with $(x_0, y_0) = (0, b)$. In order to avoid fractions, the derivation scales this by a constant factor, which yields the integer form used in the code.

At each step there are two possible moves:

- E: $(x + 1,\, y)$, - SE: $(x + 1,\, y - 1)$.

If $d_1 < 0$, the midpoint lies inside the ellipse, so the closer pixel is E; otherwise it is SE. The corresponding integer recurrences obtained from F are:

$$\text{if } d_1 < 0: \quad x \leftarrow x + 1, \quad d_1 \leftarrow d_1 + 2b^2x + b^2,$$

$$\text{else}: \quad x \leftarrow x + 1,\; y \leftarrow y - 1,$$
$$d_1 \leftarrow d_1 + 2b^2x - 2a^2y + b^2.$$

My implementation uses the incremental variables `dx` and `dy` to accumulate the terms $2b^2x$ and $2a^2y$:

```
while (dx < dy) {
    point.push_back(QPoint(cx + x, cy + y));
    point.push_back(QPoint(cx - x, cy + y));
    point.push_back(QPoint(cx + x, cy - y));
    point.push_back(QPoint(cx - x, cy - y));

    if (d1 < 0) {
        x = x + 1;
        dx += 2 * b*b;
        d1 += dx + b*b;
    } else {
        x++;
        y--;
        dx += 2 * b*b;
        dy -= 2 * a*a;
        d1 += dx - dy + b*b;
    }
}
```

The four calls to `push_back` implement the symmetry $(\pm x, \pm y)$ across both axes. The inequalities `dx < dy` and the updates of `dx`, `dy` follow directly from

$$dx = 2b^2x, \qquad dy = 2a^2y,$$

which are the partial derivatives of F with respect to x and y, scaled appropriately.

Region 2: $\boldsymbol{dx \geq dy}$.

When the condition $dx < dy$ no longer holds, the slope satisfies $|dy/dx| > 1$ and we primarily decrement y. The decision parameter is reinitialised using a midpoint appropriate for the second region:

$$d_2 = F\Big(x + \tfrac{1}{2},\, y - 1\Big) = b^2(x + \tfrac{1}{2})^2 + a^2(y - 1)^2 - a^2b^2,$$

which is encoded as

```
int d2 = b*b * (x + 0.5) * (x + 0.5)
       + a*a * (y - 1)   * (y - 1)   - a*a * b*b;
```

In this second region the two possible moves are
- S: $(x,\, y-1)$, - SE: $(x+1,\, y-1)$.
The corresponding integer recurrences are

$$\text{if } d_2 > 0: \quad y \leftarrow y-1, \quad d_2 \leftarrow d_2 - 2a^2y + a^2,$$

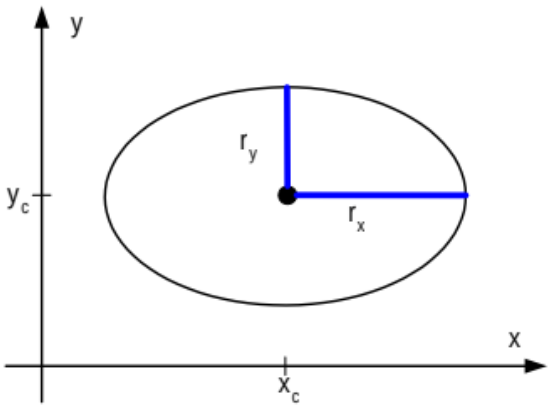
$$\text{else}: \quad x \leftarrow x+1,\ y \leftarrow y-1,$$
$$d_2 \leftarrow d_2 + 2b^2x - 2a^2y + a^2.$$

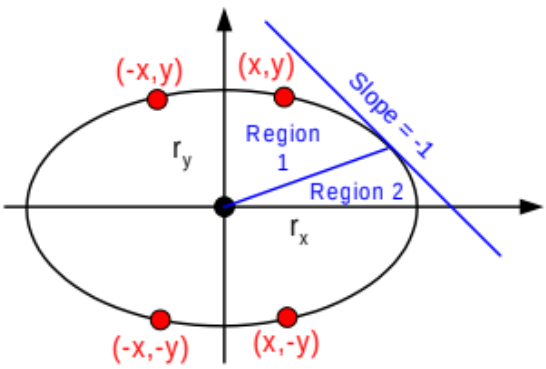
This is reflected in my implementation:

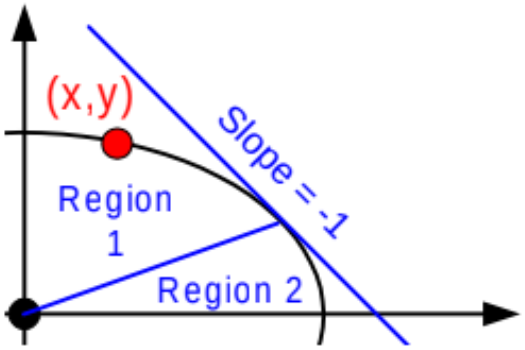
```
while (y >= 0) {
    point.push_back(QPoint(cx + x, cy + y));
    point.push_back(QPoint(cx - x, cy + y));
    point.push_back(QPoint(cx + x, cy - y));
    point.push_back(QPoint(cx - x, cy - y));

    if (d2 > 0) {
        y--;
        dy -= 2 * a*a;
        d2 += a*a - dy;
    } else {
        y--;
        x++;
        dx += 2 * b*b;
        dy -= 2 * a*a;
        d2 += dx - dy + a*a;
    }
}
```

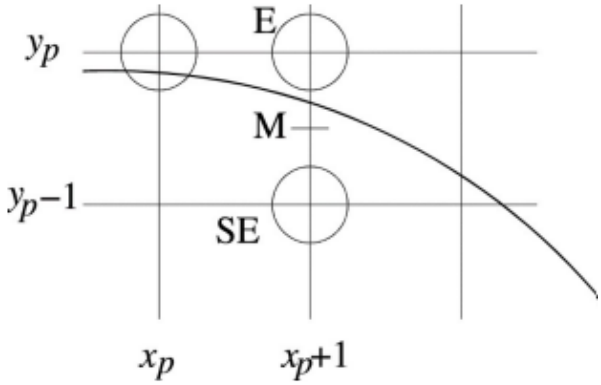
Again `dx` and `dy` maintain $2b^2x$ and $2a^2y$ so that the decision parameter updates remain purely additive.

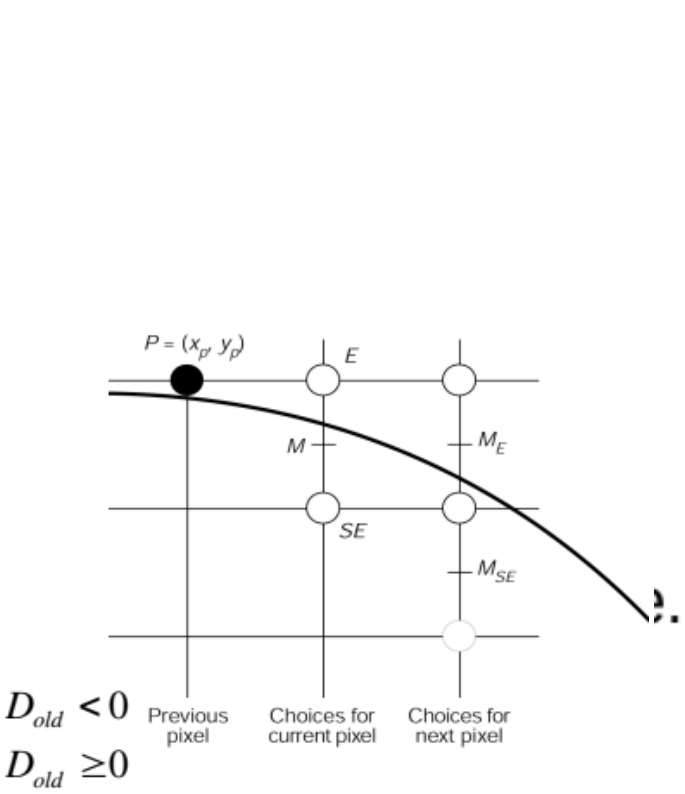
Throughout both regions, the algorithm uses only integer additions and subtractions; there are no calls to `sqrt`, `sin` or `cos`, and no explicit rounding inside the decision loop. This makes the midpoint ellipse significantly faster and more numerically stable than the purely trigonometric polar approach.

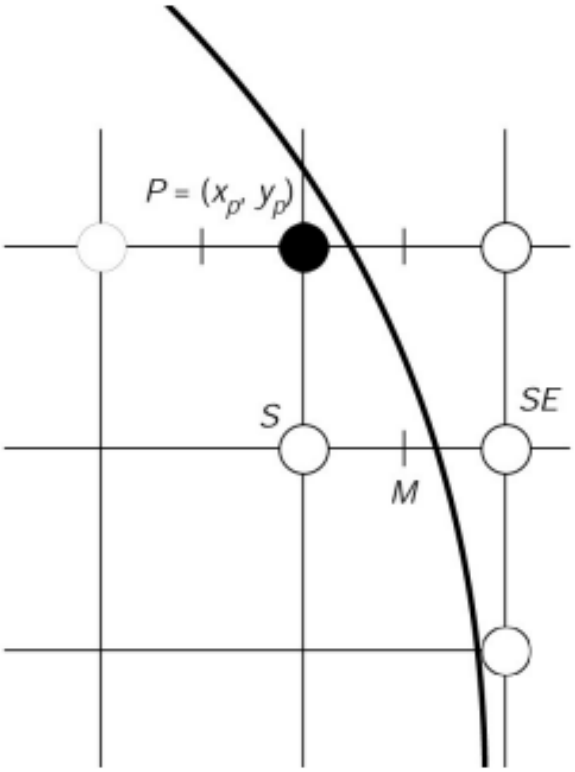












These figures complement the mathematical derivations above and visually connect the formulas with the discrete pixel trajectories produced by the three functions `draw_ellipse_polar()`, `draw_ellipse_cartesian()` and `draw_ellipse_bresenham()`.

Code

ellipse.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

CONFIG += c++17

```

```
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000

SOURCES += \
    main.cpp \
    mainwindow.cpp \
    my_label.cpp

HEADERS += \
    mainwindow.h \
    my_label.h

FORMS += \
    mainwindow.ui

qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target

}
```

my_label.h

```
#ifndef MY_LABEL_H
#define MY_LABEL_H

#include <QLabel>
#include <QMouseEvent>

class my_label : public QLabel
{
    Q_OBJECT
public:
    explicit my_label(QWidget *parent = nullptr);
    int x, y;

protected:
    void mouseMoveEvent(QMouseEvent *ev);
    void mousePressEvent(QMouseEvent *ev);

signals:
    void sendMousePosition(QPoint&);
    void Mouse_Pos();
};

#endif // MY_LABEL_H
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QPoint>

```

```
7  namespace Ui {
8  class MainWindow;
9  }
10
11 class MainWindow : public QMainWindow
12 {
13     Q_OBJECT
14
15 public:
16     explicit MainWindow(QWidget *parent = nullptr);
17     ~MainWindow();
18
19 private slots:
20     void Mouse_Pressed();
21     void showMousePosition(QPoint&);
22     void on_clear_clicked();
23     void draw_grid();
24     std::vector<QPoint> draw_line_bresenham(QPoint, QPoint);
25
26     void on_spinBox_valueChanged(int);
27     void repaint();
28
29     void paint(std::vector<QPoint>, QColor);
30     void paint(QPoint, QColor, QPainter&);
31
32     void draw_ellipse(QPoint, int, int);
33     std::vector<QPoint> draw_ellipse_polar(QPoint, int, int);
34     std::vector<QPoint> draw_ellipse_bresenham(QPoint, int, int);
35     std::vector<QPoint> draw_ellipse_cartesian(QPoint, int, int);
36
37     void on_draw_ellipse_clicked();
38
39     void on_ellipse_type_currentIndexChanged(int index);
40
41     void on_undo_clicked();
42
43 private:
44     Ui::MainWindow *ui;
45     void addPoint(int x, int y, int c = 1);
46
47     QPoint lastPoint1, lastPoint2;
48     int sc_x, sc_y;
49     int org_x, org_y;
50     int width, height;
51
52     std::vector<std::pair<std::vector<QPoint>, QColor>> colormap;
53     int gap;
54     int fellipse;
55 };
56
57 #endif // MAINWINDOW_H
```

main.cpp

```cpp
#include "mainwindow.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

my_label.cpp

```cpp
#include "my_label.h"

my_label::my_label(QWidget *parent) : QLabel(parent)
{
    this->setMouseTracking(true);
}

void my_label::mouseMoveEvent(QMouseEvent *ev)
{
    QPoint pos = ev->pos();
    if (pos.x() >= 0 && pos.y() >= 0 && pos.x() < this->width() && pos.y() < this
        ->height()) {
        emit sendMousePosition(pos);
    }
}

void my_label::mousePressEvent(QMouseEvent *ev)
{
    if (ev->button() == Qt::LeftButton) {
        x = ev->x();
        y = ev->y();
        emit Mouse_Pos();
    }
}
```

mainwindow.cpp

```cpp
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QPixmap>
#include <QImage>
#include <QPainter>
#include <QDebug>
#include <QTimer>
#include <QQueue>
#include <QSet>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
```

```
13      ui(new Ui::MainWindow)
14  {
15      ui->setupUi(this);
16      lastPoint1 = QPoint(-100000, -100000);
17      lastPoint2 = QPoint(-100000, -100000);
18
19      connect(ui->frame, SIGNAL(Mouse_Pos()), this, SLOT(Mouse_Pressed()));
20      connect(ui->frame, SIGNAL(sendMousePosition(QPoint&)), this, SLOT(
            showMousePosition(QPoint&)));
21
22      gap = 10;
23      fellipse = 0;
24
25      width = 750;
26      height = 450;
27
28      draw_grid();
29      ui->polarellipsetime->setText("Polar: 0 ns");
30      ui->bresenhamellipsetime->setText("Bresenham: 0 ns");
31
32      ui->filled->setText("Filled: 0 pixels");
33
34      ui->undo->setEnabled(false);
35  }
36
37  MainWindow::~MainWindow()
38  {
39      delete ui;
40  }
41
42  void MainWindow::showMousePosition(QPoint &pos)
43  {
44      sc_x = pos.x();
45      sc_y = pos.y();
46      ui->mouse_movement->setText("X : " + QString::number((sc_x-sc_x%gap-((width/2)
            -(width/2)%gap))/gap) + ", Y : " + QString::number(-(sc_y-sc_y%gap-((
            height/2)-(height/2)%gap))/gap));
47  }
48
49  void MainWindow::Mouse_Pressed()
50  {
51      org_x = sc_x;
52      org_y = sc_y;
53
54      int x = (sc_x-sc_x%gap-((width/2)-(width/2)%gap))/gap;
55      int y = (sc_y-sc_y%gap-((height/2)-(height/2)%gap))/gap;
56
57      ui->mouse_pressed->setText("X : " + QString::number(x) + ", Y : " + QString::
            number(y));
58
59      lastPoint1 = lastPoint2;
60
```

```
61      lastPoint2 = QPoint(x, y);
62  }
63
64  void MainWindow::on_clear_clicked()
65  {
66      lastPoint1 = lastPoint2 = QPoint(-100000, -100000);
67      colormap.clear();
68      draw_grid();
69      ui->polarellipsetime->setText("Polar: 0 ns");
70      ui->bresenhamellipsetime->setText("Bresenham: 0 ns");
71
72      ui->filled->setText("Filled: 0 pixels");
73  }
74
75  void MainWindow::draw_grid()
76  {
77      QPixmap pix(ui->frame->width(), ui->frame->height());
78      pix.fill(QColor(255, 255, 255));
79      ui->frame->setPixmap(pix);
80
81      QPixmap pm = ui->frame->pixmap();
82      QPainter painter(&pm);
83      painter.setPen(QPen(QColor(200, 200, 200)));
84      for(int i=0; i<=width; i+=gap) painter.drawLine(QPoint(i,0), QPoint(i, height)
            );
85      for(int i=0; i<=height; i+=gap) painter.drawLine(QPoint(0, i), QPoint(width, i
            ));
86
87      painter.end();
88      ui->frame->setPixmap(pm);
89
90      paint(draw_line_bresenham(QPoint(-100000, 0), QPoint(100000, 0)), QColor(0, 0,
             0));
91      paint(draw_line_bresenham(QPoint(0, -100000), QPoint(0, 100000)), QColor(0, 0,
             0));
92  }
93
94  void MainWindow::paint(std::vector<QPoint> points, QColor color){
95      QPixmap pm = ui->frame->pixmap();
96      QPainter painter(&pm);
97      for(QPoint p:points){
98          int x = p.x()*gap, y = p.y()*gap;
99          painter.fillRect(QRect(x+((width/2) - (width/2)%gap), y+((height/2) - (
                height/2)%gap), gap, gap), color);
100     }
101     painter.end();
102     ui->frame->setPixmap(pm);
103     ui->filled->setText("Filled: " + QString::number(points.size()) + " pixels");
104 }
105
106 void MainWindow::paint(QPoint point, QColor color, QPainter& painter){
107     int x = point.x()*gap, y = point.y()*gap;
```

```
108     painter.fillRect(QRect(x+((width/2) - (width/2)%gap), y+((height/2) - (height
            /2)%gap), gap, gap), color);
109 }
110
111 std::vector<QPoint> MainWindow::draw_line_bresenham(QPoint a, QPoint b) {
112     int x1 = a.x(), y1 = a.y();
113     int x2 = b.x(), y2 = b.y();
114
115     int dx = abs(x2 - x1);
116     int dy = abs(y2 - y1);
117
118     int sx = (x1 < x2) ? 1 : -1;
119     int sy = (y1 < y2) ? 1 : -1;
120
121     int err = dx - dy;
122
123     std::vector<QPoint> point;
124
125     while (true) {
126         point.push_back(QPoint(x1, y1));
127
128         if (x1 == x2 && y1 == y2) break;
129
130         int e2 = 2 * err;
131         if (e2 > -dy) { err -= dy; x1 += sx; }
132         if (e2 < dx)  { err += dx; y1 += sy; }
133     }
134
135     return point;
136 }
137
138 void MainWindow::on_spinBox_valueChanged(int value)
139 {
140     gap = value;
141     repaint();
142 }
143
144
145 void MainWindow::repaint(){
146     draw_grid();
147     QPixmap pm = ui->frame->pixmap();
148     QPainter painter(&pm);
149     for(const auto& pair : colormap){
150         for(QPoint p : pair.first){
151             paint(p, pair.second, painter);
152         }
153     }
154     painter.end();
155     ui->frame->setPixmap(pm);
156 }
157
158 void MainWindow::on_draw_ellipse_clicked()
```

```
159 {
160     if(lastPoint2 == QPoint(-100000, -100000)) return;
161     draw_ellipse(lastPoint2, ui->ellipse_a->value(), ui->ellipse_b->value());
162 }
163
164 void MainWindow::draw_ellipse(QPoint centre, int a, int b){
165     std::vector<QPoint> point;
166     QElapsedTimer timer;
167     qint64 time = 0;
168     if(fellipse == 0){
169         for(int i=0; i<100; i++){
170             timer.start();
171             point = draw_ellipse_polar(centre, a, b);
172             time += timer.nsecsElapsed();
173         }
174         ui->polarellipsetime->setText("Polar: " + QString::number(time/100) + " ns
                ");
175     }
176     else if(fellipse == 1){
177         for(int i=0; i<100; i++){
178             timer.start();
179             point = draw_ellipse_bresenham(centre, a, b);
180             time += timer.nsecsElapsed();
181         }
182         ui->bresenhamellipsetime->setText("Bresenham: " + QString::number(time
                /100) + " ns");
183     }
184     else{
185         point = draw_ellipse_cartesian(centre, a, b);
186     }
187     colormap.push_back(std::pair(point, fellipse==0? QColor(0, 150, 150) :
            fellipse==1? QColor(150, 0, 150) : QColor(255, 100, 200)));
188     paint(point, fellipse==0? QColor(0, 150, 150) : fellipse==1? QColor(150, 0,
            150) : QColor(255, 100, 200));
189     ui->undo->setEnabled(true);
190 }
191
192
193 std::vector<QPoint> MainWindow::draw_ellipse_polar(QPoint centre, int a, int b){
194     int cx = centre.x();
195     int cy = centre.y();
196
197     std::vector<QPoint> point;
198
199     for(double theta = 0; theta < M_PI/2; theta += 0.5/std::sqrt(a*a*cos(theta)*
            cos(theta) + b*b*sin(theta)*sin(theta))){
200         int x = std::round(a * cos(theta)), y = std::round(b * sin(theta));
201         point.push_back(QPoint(cx + x, cy + y));
202         point.push_back(QPoint(cx - x, cy + y));
203         point.push_back(QPoint(cx + x, cy - y));
204         point.push_back(QPoint(cx - x, cy - y));
205     }
```

```
206
207     return point;
208 }
209
210
211 std::vector<QPoint> MainWindow::draw_ellipse_cartesian(QPoint centre, int a, int b
        ){
212     int cx = centre.x();
213     int cy = centre.y();
214
215     std::vector<QPoint> point;
216     for(int x = 0; x<=a; x++){
217         int y = std::round(b * std::sqrt(1 - (float)(x * x)/(a * a)));
218         point.push_back(QPoint(cx + x, cy + y));
219         point.push_back(QPoint(cx - x, cy + y));
220         point.push_back(QPoint(cx + x, cy - y));
221         point.push_back(QPoint(cx - x, cy - y));
222     }
223     for(int y = 0; y<=b; y++){
224         int x = std::round(a * std::sqrt(1 - (float)(y * y)/(b * b)));
225         point.push_back(QPoint(cx + x, cy + y));
226         point.push_back(QPoint(cx - x, cy + y));
227         point.push_back(QPoint(cx + x, cy - y));
228         point.push_back(QPoint(cx - x, cy - y));
229     }
230
231     return point;
232 }
233
234
235 std::vector<QPoint> MainWindow::draw_ellipse_bresenham(QPoint centre, int a, int b
        ){
236     int cx = centre.x();
237     int cy = centre.y();
238
239     std::vector<QPoint> point;
240
241     int x = 0, y = b;
242     int d1 = b*b - a*a*b + a*a/4;
243     int dx = 2 * b*b * x;
244     int dy = 2 * a*a * y;
245
246
247     while(dx < dy){
248         point.push_back(QPoint(cx + x, cy + y));
249         point.push_back(QPoint(cx - x, cy + y));
250         point.push_back(QPoint(cx + x, cy - y));
251         point.push_back(QPoint(cx - x, cy - y));
252
253         if (d1 < 0){
254             x = x + 1;
255             dx += 2 * b*b;
```

```
256             d1 += dx + b*b;
257         }
258         else{
259             x++;
260             y--;
261             dx += 2 * b*b;
262             dy -= 2 * a*a;
263             d1 += dx - dy + b*b;
264         }
265     }
266 
267 
268     int d2 = b*b * (x+0.5)(x+0.5) + a*a * (y-1)(y-1) - a*a * b*b;
269 
270     while(y >= 0){
271         point.push_back(QPoint(cx + x, cy + y));
272         point.push_back(QPoint(cx - x, cy + y));
273         point.push_back(QPoint(cx + x, cy - y));
274         point.push_back(QPoint(cx - x, cy - y));
275 
276         if(d2 > 0){
277             y--;
278             dy -= 2 * a*a;
279             d2 += a*a - dy;
280         }
281         else{
282             y--;
283             x++;
284             dx += 2 * b*b;
285             dy -= 2 * a*a;
286             d2 += dx - dy + a*a;
287         }
288     }
289 
290     return point;
291 }
292 
293 void MainWindow::on_ellipse_type_currentIndexChanged(int index)
294 {
295     fellipse = index;
296 }
297 
298 void MainWindow::on_undo_clicked()
299 {
300     colormap.pop_back();
301     if(colormap.empty()) ui->undo->setEnabled(false);
302     repaint();
303 }
```

Outputs

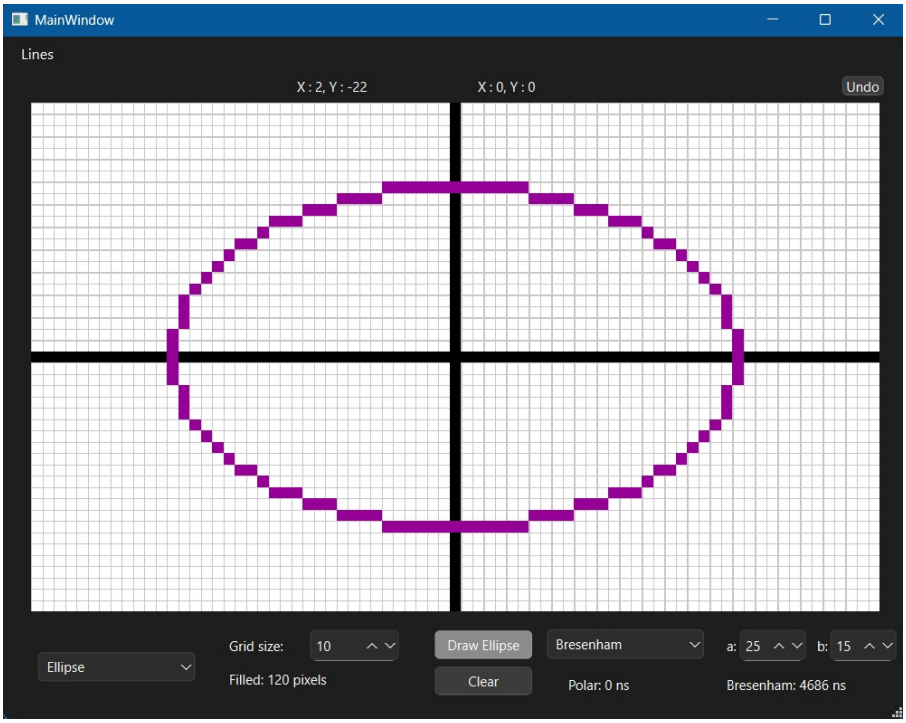


Figure 6: bresenham_ellipse

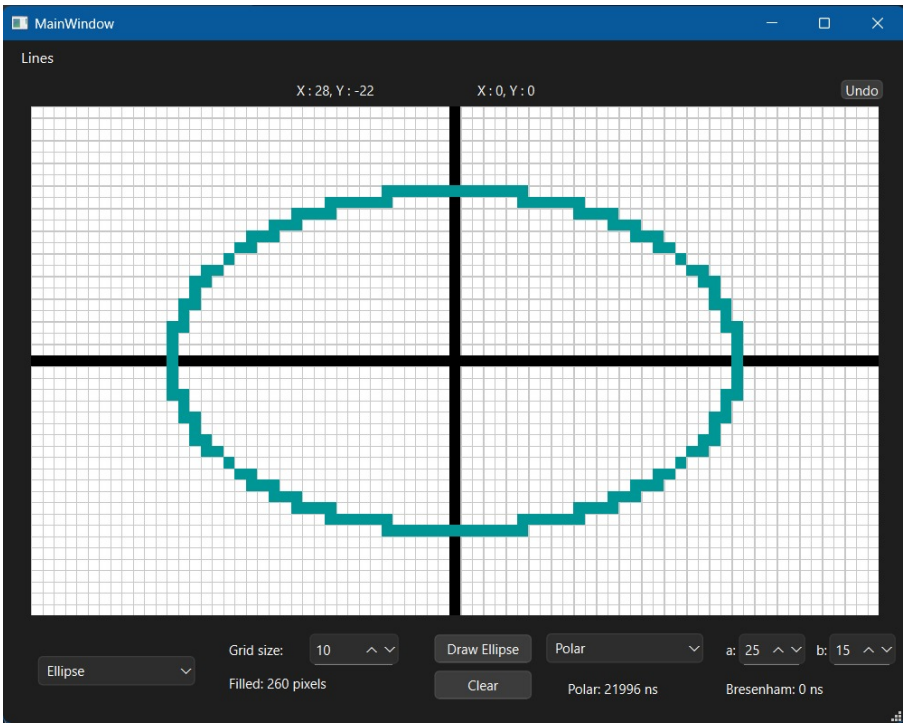


Figure 7: polar_ellipse

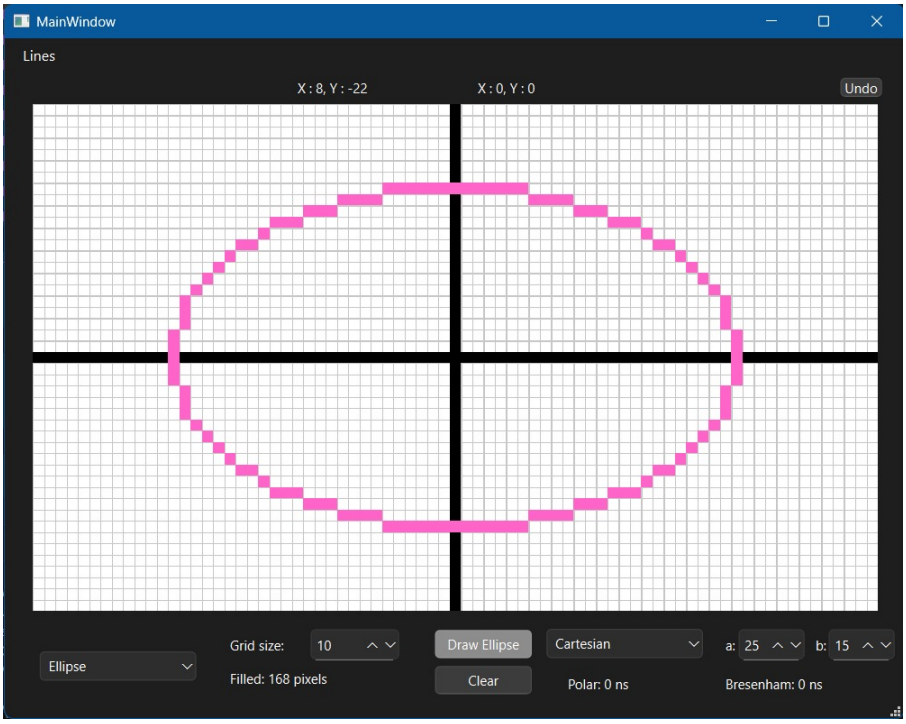


Figure 8: cartesian_ellipse

Output Explanation

For comparing the three ellipse drawing algorithms, I fixed the ellipse centre at the origin of the grid $(0, 0)$ and used the same pair of semi–axes (a, b) for all methods. Using the GUI, I first selected the origin as the centre and then specified identical values of a and b for:

- the polar (parametric) ellipse,
- the Cartesian ellipse,
- the Bresenham (midpoint) ellipse.

All three algorithms therefore approximate the same continuous ellipse

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1,$$

and any differences in the rasterised curves are purely due to their numerical treatment and rounding.

The timing is performed inside `draw_ellipse()` using `QElapsedTimer`. For each selected method, I call the corresponding routine (`draw_ellipse_polar()` or `draw_ellipse_bresenham()`) 100 times in a loop and accumulate the elapsed time in nanoseconds; the value displayed in the UI is the integer average over these 100 runs. The Cartesian variant is drawn for visual comparison.

With this setup, the measured average times for one ellipse of fixed (a, b) centred at the origin are:

$$T_{\text{polar}} \approx 21996 \text{ ns},$$
$$T_{\text{Bresenham}} \approx 4586 \text{ ns}.$$

Thus, the midpoint ellipse is roughly five times faster than the polar implementation. This directly reflects the algorithmic differences: the polar method repeatedly evaluates trigonometric functions and square roots, whereas Bresenham's ellipse uses only incremental integer additions and subtractions inside its main loop.

Minor pixel–level deviations are visible near regions of high curvature, where the polar and Cartesian methods round independently computed real coordinates, while the midpoint method follows a purely incremental decision process. Overall, the experiment shows that the Bresenham ellipse achieves comparable visual quality at a significantly lower computational cost compared to the polar method.

Assignment 4: Seed-Fill and Scanline Fill

Problem Statement

- Implement seed-fill algorithms:
  - Boundary fill
  - Flood fill
- Implement the scanline fill algorithm:
  - Draw a closed polygon and apply the scanline fill algorithm to fill the polygon.

Code and Theoretical Explanation

Boundary Fill Algorithm. In the boundary fill approach, I assume that the polygon boundary has already been rasterised with a distinct *boundary colour* and that the interior may contain arbitrary colours except this boundary colour and the chosen *fill colour*. Starting from a seed pixel

$$p_0 = (x_0, y_0),$$

the interior region to be filled is defined as the maximal connected set

$$R = \{(x, y) \,|\, \text{there exists a path from } p_0 \text{ to } (x, y) \text{ using only non–boundary, non–fill pixels}\}.$$

Connectivity can be either 4-connected,

$$N_4(x, y) = \{(x \pm 1, y), (x, y \pm 1)\},$$

or 8-connected,

$$N_8(x, y) = N_4(x, y) \cup \{(x \pm 1, y \pm 1)\},$$

as illustrated in the neighbourhood diagrams.:contentReference[oaicite:0]index=0

The classical recursive definition of boundary fill is:

$$\mathrm{BF}(x, y) = \begin{cases} \text{return}, & \text{if } C(x, y) = C_{\text{boundary}} \text{ or } C(x, y) = C_{\text{fill}}, \\ C(x, y) \leftarrow C_{\text{fill}}, & \text{otherwise, and recursively apply BF to neighbours.} \end{cases}$$

Here $C(x, y)$ denotes the current colour at pixel (x, y).

My implementation uses an explicit stack instead of recursion, but the logic is the same. Around the middle of `boundaryFill()` I build a work stack and repeatedly pop cells:

```
QSet<QPair<int,int>> visited;
QVector<QPoint> stack;
stack.append(QPoint(gx0, gy0));
...
while (!stack.isEmpty()) {
    QPoint p = stack.takeLast();
    int gx = p.x(), gy = p.y();
```

```
    ...
    QColor cur = img.pixelColor(cx, cy);
    if (cur == newColor) continue;       // already filled
    if (cur == boundaryColor) continue; // stop at boundary
    ...
    painter.fillRect(rect, QBrush(newColor, Qt::SolidPattern));
    ...
    if (connectivityMode == Connectivity::Eight) {
        push8(gx, gy);
    } else {
        push4(gx, gy);
    }
}
```

The predicates

$$C(x,y) = C_{\text{boundary}}, \qquad C(x,y) = C_{\text{fill}}$$

are implemented by the two early `continue` checks on `cur`. The functions `push4()` and `push8()` generate the appropriate neighbourhood N_4 or N_8, which matches the 4-connected / 8-connected patterns shown in the neighbourhood figures.

Because I operate in grid coordinates, each logical cell (g_x, g_y) is mapped to the rectangular pixel block

$$[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}], \quad x_{\min} = g_x \cdot \text{grid\_box}, \; y_{\min} = g_y \cdot \text{grid\_box},$$

and I fill this block via `painter.fillRect`. This maintains a consistent one–cell–one–colour model for analysis of interior versus boundary.

Flood Fill Algorithm. Flood fill differs conceptually from boundary fill in the region predicate. Instead of stopping at a fixed boundary colour, I define the region to be the maximal set of pixels having the same *old colour* as the seed:

$$R = \{(x,y) \,|\, C(x,y) = C_{\text{old}} \text{ and connected to } p_0\}.$$

The recursive specification is

$$\text{FF}(x,y) = \begin{cases} \text{return}, & \text{if } C(x,y) \neq C_{\text{old}}, \\ C(x,y) \leftarrow C_{\text{new}}, & \text{otherwise and recursively apply FF to neighbours.} \end{cases}$$

In my `floodFill()` implementation, I first determine the logical seed cell and read its colour from the image:

```
int gx0 = qFloor(x / (double)this->grid_box);
int gy0 = qFloor(y / (double)this->grid_box);
...
QImage img = pm.toImage();
int cx0 = gx0 * this->grid_box + this->grid_box / 2;
int cy0 = gy0 * this->grid_box + this->grid_box / 2;
QColor seed = img.pixelColor(cx0, cy0);
```

From this I derive a *region colour* (`regionColor`) that plays the role of C_{old}. All pixels that are part of

the connected component with this colour are candidates for replacement by `newColor`. The main loop contains the core test:

```
QColor cur = img.pixelColor(cx, cy);
...
if (cur == Qt::white) cur = regionColor;
if (isSeed && seedIsYellow) cur = regionColor;
if (cur != regionColor) continue;
```

This sequence implements $C(x, y) = C_{\text{old}}$: only if the effective colour equals `regionColor` does the algorithm fill the cell. The subsequent call

```
painter.fillRect(rect, QBrush(newColor, Qt::SolidPattern));
```

realises the assignment $C(x, y) \leftarrow C_{\text{new}}$.

As in boundary fill, I use either 4-connected or 8-connected neighbours:

```
if (connectivityMode == Connectivity::Eight) {
    push8(gx, gy);
} else {
    push4(gx, gy);
}
```

This matches the theoretical characterisation of flood fill as a region-growing process over a discrete lattice using a chosen connectivity.

Scanline Polygon Fill Algorithm. For scanline fill, the polygon is considered in terms of its edges. Let the vertices in Cartesian coordinates be

$$P_i = (x_i, y_i), \quad i = 0, \ldots, n-1,$$

with edges

$$E_i : P_i \to P_{i+1}, \qquad P_n \equiv P_0.$$

For a fixed integer scanline y, I find all intersections between this horizontal line and the polygon edges. For any non-horizontal edge from (x_1, y_1) to (x_2, y_2), parameterised as

$$x(t) = x_1 + t(x_2 - x_1), \quad y(t) = y_1 + t(y_2 - y_1), \quad t \in [0, 1],$$

the value of t at which the edge crosses the scanline is obtained by solving

$$y(t) = y + \tfrac{1}{2} \quad \Rightarrow \quad t = \frac{(y + 0.5) - y_1}{y_2 - y_1}.$$

The corresponding intersection x-coordinate is

$$x_{\text{int}} = x_1 + t(x_2 - x_1).$$

This is implemented in the loop over edges:

```
for (int i = 0; i < n; ++i) {
    const QPoint p1 = poly.vertices[i];
```

```
    const QPoint p2 = poly.vertices[(i + 1) % n];
    ...
    if (y >= yminEdge && y < ymaxEdge) {
        const double t  = ((y + 0.5) - y1) / double(y2 - y1);
        const double ix = x1 + t * (x2 - x1);
        interX.append(ix);
    }
}
```

The half-pixel offset $(y + 0.5)$ corresponds to evaluating the intersection through the centre of each scanline band, avoiding ambiguity at vertex positions.

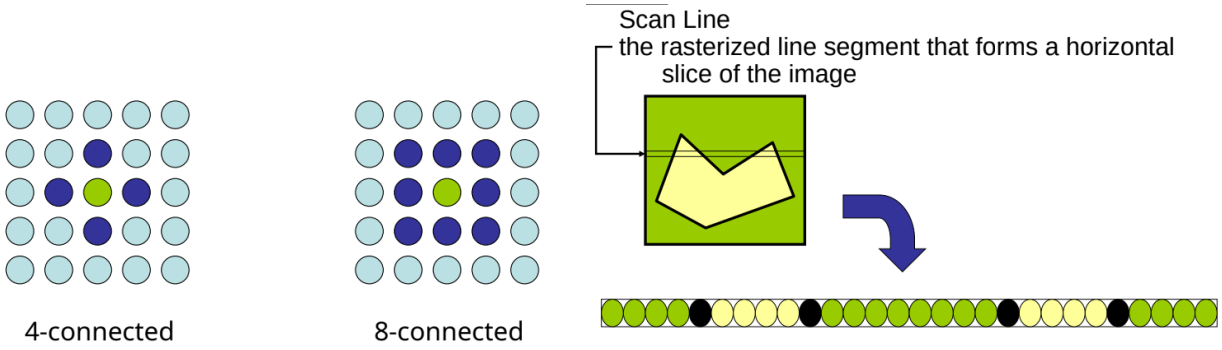
Once all intersections $\{x_k\}$ for that scanline are collected, they are sorted:

$$x_{(0)} \leq x_{(1)} \leq \cdots \leq x_{(m-1)}.$$

The even–odd rule is then applied: for each pair $(x_{(2k)}, x_{(2k+1)})$ I consider the interval between them as being inside the polygon and fill all cells whose centres lie strictly within that interval. This is reflected in the code:

```
std::sort(interX.begin(), interX.end());
for (int k = 0; k + 1 < interX.size(); k += 2) {
    const double L = std::min(interX[k],   interX[k+1]);
    const double R = std::max(interX[k],   interX[k+1]);

    int xLeft  = int(std::ceil (L + epsx));
    int xRight = int(std::floor(R - epsx));
    if (xRight < xLeft) continue;

    for (int lx = xLeft; lx <= xRight; ++lx) {
        int gx = lx + this->center;
        int gy = this->center - y;
        ...
        QRect rect(gx * this->grid_box, gy * this->grid_box,
                   this->grid_box, this->grid_box);
        painter.fillRect(rect, QBrush(fillColor, Qt::SolidPattern));
    }
    maybeAnimate();
}
```

The use of $\lceil L + \varepsilon \rceil$ and $\lfloor R - \varepsilon \rfloor$ corresponds to including only pixels whose centres lie strictly between two consecutive intersection points, which matches the theoretical requirement that we do not overwrite the boundary itself. The mapping from logical coordinates (l_x, y) to raster cells uses the same centre-based transform as in the line and circle assignments.



Together, these figures visually support the mathematical characterisation and the implemented algorithms described above.

Code

filling.pro

```
QT += core gui widgets
CONFIG += c++17

SOURCES += \
main.cpp \
mainwindow.cpp \
my_label.cpp

HEADERS += \
mainwindow.h \
my_label.h

FORMS += \
mainwindow.ui
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QPoint>
#include <QVector>
#include <QPixmap>
#include <QPainter>
#include <QColor>
#include <QMap>
#include <QComboBox>
#include <QLabel>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

struct Polygon {
```

```
19              QVector<QPoint> vertices;
20              Qt::GlobalColor col = Qt::blue;
21
22              bool   floodFilled  = false;
23              QColor floodCol     = Qt::red;
24              QPoint floodSeedPx  = QPoint(-1, -1);
25
26              bool   boundaryFilled  = false;
27              QColor boundaryCol     = Qt::green;
28              QPoint boundarySeedPx  = QPoint(-1, -1);
29
30              bool   scanFilled  = false;
31              QColor scanCol     = Qt::blue;
32          };
33
34          class MainWindow : public QMainWindow
35          {
36              Q_OBJECT
37
38              public:
39              explicit MainWindow(QWidget *parent = nullptr);
40              ~MainWindow();
41
42              private slots:
43              void showMousePosition(const QPoint& pos);
44              void Mouse_Pressed();
45
46              void on_draw_grid_clicked();
47              void on_clear_clicked();
48              void on_grid_size_valueChanged(int grid);
49
50              void on_start_polygon_clicked();
51              void on_close_polygon_clicked();
52              void on_fill_polygon_clicked();
53              void on_clear_fill_clicked();
54
55              void on_algo_select_currentIndexChanged(int index);
56
57              private:
58              Ui::MainWindow *ui;
59
60              int  grid_box;
61              int  grid_size;
62              int  center;
63              bool draw_clicked;
64
65              int sc_x, sc_y;
66              int org_x, org_y;
67              QPoint lastPoint1, lastPoint2;
68
69              bool polygonFlag;
70              Polygon currPoly;
```

```
71              QVector<Polygon> polygons;
72
73              QMap<QPair<int,int>, QColor> paintedCells;
74
75              enum class Algo { Flood, Boundary, Scan };
76              Algo currentAlgo;
77
78              enum class Connectivity { Four, Eight };
79              Connectivity connectivityMode = Connectivity::Four;
80              QComboBox* connectivitySelect = nullptr;
81
82              void draw_axes(QPainter& painter);
83              void refreshCanvas();
84              void redraw();
85              void drawPaintedCells(QPainter& painter);
86
87              void draw_line_grid(int X1, int X2, int Y1, int Y2, QPainter& painter,
                     QColor col);
88
89              void draw_polygon(QPainter& painter, const Polygon& poly);
90              void draw_polygon_preview(QPainter& painter, const QVector<QPoint>&
                    verts);
91              void draw_vertex_markers(QPainter& painter, const QVector<QPoint>&
                    verts);
92
93              void floodFill(int x_px, int y_px, const QColor& newColor);
94              void boundaryFill(int x_px, int y_px, const QColor& newColor, const
                    QColor& boundaryColor);
95              void scanlineFill(const Polygon& poly, const QColor& fillColor, bool
                    animate);
96
97              void paintCellAtPixel(int x_px, int y_px);
98          };
99
100         #endif
```

my_label.h

```
1           #ifndef MY_LABEL_H
2           #define MY_LABEL_H
3
4           #include <QLabel>
5           #include <QMouseEvent>
6
7           class my_label : public QLabel
8           {
9               Q_OBJECT
10              public:
11              explicit my_label(QWidget *parent = nullptr);
12              int x, y;
13
14              protected:
15              void mouseMoveEvent(QMouseEvent *ev) override;
```

```
16              void mousePressEvent(QMouseEvent *ev) override;
17
18              signals:
19              void sendMousePosition(const QPoint&);
20              void Mouse_Pos();
21          };
22
23          #endif
```

main.cpp

```
1           #include <QApplication>
2           #include "mainwindow.h"
3
4           int main(int argc, char *argv[]) {
5               QApplication a(argc, argv);
6               MainWindow w;
7               w.show();
8               return a.exec();
9           }
```

mainwindow.cpp

```
1           #include "mainwindow.h"
2           #include "ui_mainwindow.h"
3
4           #include <QPixmap>
5           #include <QImage>
6           #include <QPainter>
7           #include <QDebug>
8           #include <QtMath>
9           #include <QThread>
10          #include <QCoreApplication>
11          #include <QSet>
12          #include <QLabel>
13          #include <QComboBox>
14          #include <algorithm>
15
16          static inline QPair<int,int> cellOfPx(int x_px, int y_px, int grid_box) {
17              return qMakePair(qFloor(x_px / double(grid_box)), qFloor(y_px / double
                    (grid_box)));
18          }
19
20          MainWindow::MainWindow(QWidget *parent) :
21          QMainWindow(parent),
22          ui(new Ui::MainWindow)
23          {
24              ui->setupUi(this);
25
26              draw_clicked = false;
27
28              lastPoint1 = QPoint(-1, -1);
29              lastPoint2 = QPoint(-1, -1);
30
```

```
31
32              QPixmap pix(ui->frame->width(), ui->frame->height());
33              pix.fill(Qt::black);
34              ui->frame->setPixmap(pix);
35
36
37              connect(ui->frame, SIGNAL(Mouse_Pos()), this, SLOT(Mouse_Pressed()));
38              connect(ui->frame, SIGNAL(sendMousePosition(const QPoint&)),this, SLOT
                    (showMousePosition(const QPoint&)));
39
40
41              grid_box  = 10;
42              grid_size = ui->frame->frameSize().width();
43              center    = 0;
44
45
46              polygonFlag = false;
47              currPoly.col = Qt::blue;
48              currPoly.vertices.clear();
49
50
51              currentAlgo = Algo::Flood;
52              {
53                  auto *connLabel = new QLabel("Connectivity:", this);
54                  connectivitySelect = new QComboBox(this);
55                  connectivitySelect->setObjectName("connectivity_select");
56                  connectivitySelect->addItem("4-neighbour");
57                  connectivitySelect->addItem("8-neighbour");
58
59
60                  int idx = ui->verticalLayout->indexOf(ui->algo_select);
61                  ui->verticalLayout->insertWidget(idx + 1, connLabel);
62                  ui->verticalLayout->insertWidget(idx + 2, connectivitySelect);
63
64
65                  connect(connectivitySelect, &QComboBox::currentIndexChanged, this,
                        [this](int i){
66                      connectivityMode = (i == 1) ? Connectivity::Eight :
                            Connectivity::Four;
67                  });
68                  connectivitySelect->setCurrentIndex(0);
69              }
70
71
72              on_draw_grid_clicked();
73          }
74
75          MainWindow::~MainWindow()
76          {
77              delete ui;
78          }
79
```

```
80
81
82
83          void MainWindow::showMousePosition(const QPoint &pos)
84          {
85              sc_x = pos.x();
86              sc_y = pos.y();
87
88              int X = (sc_x / this->grid_box) - this->center;
89              int Y = -(sc_y / this->grid_box) + this->center;
90
91              ui->mouse_movement->setText("X : " + QString::number(X) + ", Y : " +
                    QString::number(Y));
92          }
93
94          void MainWindow::Mouse_Pressed()
95          {
96              org_x = sc_x;
97              org_y = sc_y;
98
99              int X = (sc_x / this->grid_box) - this->center;
100             int Y = -(sc_y / this->grid_box) + this->center;
101
102             lastPoint1 = lastPoint2;
103             lastPoint2 = QPoint(org_x, org_y);
104
105             if (polygonFlag) {
106
107                 currPoly.vertices.push_back(QPoint(X, Y));
108                 ui->mouse_pressed->setText("Vertex: X=" + QString::number(X) + ",
                        Y=" + QString::number(Y));
109
110
111                 redraw();
112                 QPixmap pm = ui->frame->pixmap();
113                 QPainter painter(&pm);
114                 draw_polygon_preview(painter, currPoly.vertices);
115                 draw_vertex_markers(painter, currPoly.vertices);
116                 painter.end();
117                 ui->frame->setPixmap(pm);
118             } else {
119
120                 ui->mouse_pressed->setText("Painted: X=" + QString::number(X) + ",
                         Y=" + QString::number(Y));
121                 paintCellAtPixel(org_x, org_y);
122             }
123         }
124
125
126
127
128         void MainWindow::on_draw_grid_clicked()
```

```
129         {
130             draw_clicked = true;
131 
132             QPixmap pm = ui->frame->pixmap();
133             QPainter painter(&pm);
134 
135             const int total_cells = this->grid_size / this->grid_box;
136             this->center = total_cells / 2;
137 
138 
139             draw_axes(painter);
140 
141 
142             painter.setPen(QPen(Qt::white, 1));
143             for (int i = 0; i <= total_cells; i++) {
144                 int pos = i * this->grid_box;
145                 painter.drawLine(QPoint(pos, 0), QPoint(pos, this->grid_size));
146                 painter.drawLine(QPoint(0, pos), QPoint(this->grid_size, pos));
147             }
148 
149             painter.end();
150             ui->frame->setPixmap(pm);
151         }
152 
153         void MainWindow::draw_axes(QPainter& painter)
154         {
155             int c = this->center;
156             draw_line_grid(-c,  c, 0, 0, painter, Qt::white);
157             draw_line_grid(0, 0, -c,  c, painter, Qt::white);
158         }
159 
160 
161 
162 
163         void MainWindow::draw_line_grid(int X1, int X2, int Y1, int Y2,
164         QPainter& painter, QColor col)
165         {
166             const int cells = this->grid_size / this->grid_box;
167 
168 
169             int gx1 = X1 + this->center;
170             int gy1 = this->center - Y1;
171             int gx2 = X2 + this->center;
172             int gy2 = this->center - Y2;
173 
174             QSet<QRect> rects;
175 
176             int dx = std::abs(gx2 - gx1);
177             int dy = std::abs(gy2 - gy1);
178             int sx = (gx1 < gx2) ? 1 : -1;
179             int sy = (gy1 < gy2) ? 1 : -1;
180             int err = dx - dy;
```

```
181
182             while (true) {
183                 if (gy1 >= 0 && gy1 < cells && gx1 >= 0 && gx1 < cells) {
184                     rects.insert(QRect(gx1 * this->grid_box, gy1 * this->grid_box,
185                     this->grid_box, this->grid_box));
186                 }
187                 if (gx1 == gx2 && gy1 == gy2) break;
188                 int e2 = 2 * err;
189                 if (e2 > -dy) { err -= dy; gx1 += sx; }
190                 if (e2 <  dx) { err += dx; gy1 += sy; }
191             }
192
193             for (const QRect &rect : rects) {
194                 painter.fillRect(rect, QBrush(col, Qt::SolidPattern));
195             }
196         }
197
198
199
200
201         void MainWindow::on_clear_clicked()
202         {
203             lastPoint1 = QPoint(-1, -1);
204             lastPoint2 = QPoint(-1, -1);
205             currPoly.vertices.clear();
206             polygons.clear();
207             paintedCells.clear();
208
209             QPixmap pix(ui->frame->width(), ui->frame->height());
210             pix.fill(Qt::black);
211             ui->frame->setPixmap(pix);
212
213             on_draw_grid_clicked();
214         }
215
216         void MainWindow::refreshCanvas()
217         {
218             QPixmap pix(ui->frame->width(), ui->frame->height());
219             pix.fill(Qt::black);
220             ui->frame->setPixmap(pix);
221             on_draw_grid_clicked();
222         }
223
224         void MainWindow::drawPaintedCells(QPainter& painter)
225         {
226             for (auto it = paintedCells.constBegin(); it != paintedCells.constEnd
                    (); ++it) {
227                 const QPair<int,int>& cell = it.key();
228                 const QColor& color = it.value();
229                 QRect rect(cell.first * this->grid_box, cell.second * this->
                        grid_box,
230                 this->grid_box, this->grid_box);
```

```
231                 painter.fillRect(rect, QBrush(color, Qt::SolidPattern));
232             }
233         }
234
235         void MainWindow::redraw()
236         {
237             QPixmap pm = ui->frame->pixmap();
238             QPainter painter(&pm);
239
240
241             drawPaintedCells(painter);
242
243
244             for (const auto &poly : std::as_const(polygons)) {
245                 if (poly.vertices.size() >= 2) {
246                     draw_polygon(painter, poly);
247                 }
248             }
249
250             painter.end();
251             ui->frame->setPixmap(pm);
252
253
254             for (const auto &poly : std::as_const(polygons)) {
255                 if (poly.boundaryFilled && poly.boundarySeedPx != QPoint(-1, -1))
                        {
256                     boundaryFill(poly.boundarySeedPx.x(), poly.boundarySeedPx.y(),
257                     poly.boundaryCol, QColor(poly.col));
258                 }
259                 if (poly.floodFilled && poly.floodSeedPx != QPoint(-1, -1)) {
260                     floodFill(poly.floodSeedPx.x(), poly.floodSeedPx.y(), poly.
                            floodCol);
261                 }
262                 if (poly.scanFilled) {
263                     scanlineFill(poly, poly.scanCol, false);
264                 }
265             }
266         }
267
268
269
270
271         void MainWindow::on_grid_size_valueChanged(int grid)
272         {
273             if (grid < 2) return;
274             this->grid_box = grid;
275
276             if (!draw_clicked) return;
277
278             refreshCanvas();
279             redraw();
280         }
```

```
281
282
283
284
285         void MainWindow::draw_polygon(QPainter& painter, const Polygon& poly)
286         {
287             if (poly.vertices.size() < 2) return;
288
289             for (int i = 0; i < poly.vertices.size() - 1; ++i) {
290                 const QPoint p1 = poly.vertices[i];
291                 const QPoint p2 = poly.vertices[i + 1];
292                 draw_line_grid(p1.x(), p2.x(), p1.y(), p2.y(), painter, QColor(
                        poly.col));
293             }
294
295             draw_line_grid(poly.vertices.back().x(), poly.vertices.front().x(),
296             poly.vertices.back().y(), poly.vertices.front().y(),
297             painter, QColor(poly.col));
298         }
299
300         void MainWindow::draw_polygon_preview(QPainter& painter, const QVector<
                QPoint>& verts)
301         {
302             if (verts.size() < 1) return;
303
304             for (int i = 0; i + 1 < verts.size(); ++i) {
305                 const QPoint p1 = verts[i];
306                 const QPoint p2 = verts[i + 1];
307                 draw_line_grid(p1.x(), p2.x(), p1.y(), p2.y(), painter, QColor(Qt
                        ::blue));
308             }
309         }
310
311         void MainWindow::draw_vertex_markers(QPainter& painter, const QVector<
                QPoint>& verts)
312         {
313             const int cells = this->grid_size / this->grid_box;
314             for (const QPoint &v : verts) {
315                 int gx = v.x() + this->center;
316                 int gy = this->center - v.y();
317                 if (gx < 0 || gy < 0 || gx >= cells || gy >= cells) continue;
318                 QRect rect(gx * this->grid_box, gy * this->grid_box, this->
                        grid_box, this->grid_box);
319                 painter.fillRect(rect, QBrush(Qt::cyan, Qt::SolidPattern));
320             }
321         }
322
323         void MainWindow::on_start_polygon_clicked()
324         {
325             polygonFlag = true;
326             currPoly = Polygon();
327             currPoly.col = Qt::blue;
```

```
            currPoly.vertices.clear();
        }

        void MainWindow::on_close_polygon_clicked()
        {
            if (currPoly.vertices.size() < 3) {
                qDebug() << "Need at least 3 points to form a polygon.";
            } else {
                polygonFlag = false;
                polygons.push_back(currPoly);


                QPixmap pm = ui->frame->pixmap();
                QPainter painter(&pm);
                draw_polygon(painter, currPoly);
                painter.end();
                ui->frame->setPixmap(pm);
            }
            currPoly.vertices.clear();
        }




        void MainWindow::paintCellAtPixel(int x_px, int y_px)
        {
            auto cell = cellOfPx(x_px, y_px, this->grid_box);
            const int gx = cell.first;
            const int gy = cell.second;

            const int cells = this->grid_size / this->grid_box;
            if (gx < 0 || gy < 0 || gx >= cells || gy >= cells) return;

            paintedCells[qMakePair(gx, gy)] = Qt::yellow;

            QPixmap pm = ui->frame->pixmap();
            QPainter painter(&pm);

            QRect rect(gx * this->grid_box, gy * this->grid_box,
            this->grid_box, this->grid_box);
            painter.fillRect(rect, QBrush(Qt::yellow, Qt::SolidPattern));
            painter.end();

            ui->frame->setPixmap(pm);
        }




        void MainWindow::floodFill(int x, int y, const QColor& newColor)
        {
            QPixmap pm = ui->frame->pixmap();
```

```
380             if (pm.isNull()) return;
381
382             const int cells = this->grid_size / this->grid_box;
383             int gx0 = qFloor(x / (double)this->grid_box);
384             int gy0 = qFloor(y / (double)this->grid_box);
385             if (gx0 < 0 || gy0 < 0 || gx0 >= cells || gy0 >= cells) return;
386
387             QImage img = pm.toImage();
388
389
390             int cx0 = gx0 * this->grid_box + this->grid_box / 2;
391             int cy0 = gy0 * this->grid_box + this->grid_box / 2;
392             QColor seed = img.pixelColor(cx0, cy0);
393             const bool seedIsYellow = (seed == Qt::yellow);
394
395             QColor regionColor;
396             if (seed == Qt::white || seedIsYellow) {
397                 regionColor = Qt::black;
398             } else {
399                 regionColor = seed;
400             }
401
402             if (regionColor == newColor) return;
403
404
405             paintedCells.remove(qMakePair(gx0, gy0));
406
407             QSet<QPair<int, int>> visited;
408             QVector<QPoint> stack;
409             stack.append(QPoint(gx0, gy0));
410
411             QPainter painter(&pm);
412
413             auto maybeAnimate = [&]() {
414                 ui->frame->setPixmap(pm);
415                 QCoreApplication::processEvents();
416                 QThread::msleep(15);
417             };
418
419             auto push4 = [&](int gx, int gy) {
420                 stack.append(QPoint(gx + 1, gy));
421                 stack.append(QPoint(gx - 1, gy));
422                 stack.append(QPoint(gx, gy + 1));
423                 stack.append(QPoint(gx, gy - 1));
424             };
425             auto push8 = [&](int gx, int gy) {
426                 push4(gx, gy);
427                 stack.append(QPoint(gx + 1, gy + 1));
428                 stack.append(QPoint(gx - 1, gy + 1));
429                 stack.append(QPoint(gx + 1, gy - 1));
430                 stack.append(QPoint(gx - 1, gy - 1));
431             };
```

```
432
433             while (!stack.isEmpty()) {
434                 QPoint p = stack.takeLast();
435                 int gx = p.x();
436                 int gy = p.y();
437
438                 if (gx < 0 || gy < 0 || gx >= cells || gy >= cells) continue;
439
440                 QPair<int, int> key(gx, gy);
441                 if (visited.contains(key)) continue;
442                 visited.insert(key);
443
444                 int cx = gx * this->grid_box + this->grid_box / 2;
445                 int cy = gy * this->grid_box + this->grid_box / 2;
446
447                 QColor cur = img.pixelColor(cx, cy);
448
449                 const bool isSeed = (gx == gx0 && gy == gy0);
450
451
452                 if (cur == Qt::yellow && !isSeed) continue;
453
454
455                 if (cur == Qt::white) cur = regionColor;
456
457
458                 if (isSeed && seedIsYellow) cur = regionColor;
459
460                 if (cur != regionColor) continue;
461
462                 QRect rect(gx * this->grid_box, gy * this->grid_box, this->
                        grid_box, this->grid_box);
463                 painter.fillRect(rect, QBrush(newColor, Qt::SolidPattern));
464
465                 maybeAnimate();
466
467                 if (connectivityMode == Connectivity::Eight) {
468                     push8(gx, gy);
469                 } else {
470                     push4(gx, gy);
471                 }
472             }
473
474             painter.end();
475             ui->frame->setPixmap(pm);
476         }
477
478
479         void MainWindow::boundaryFill(int x, int y, const QColor& newColor, const
                QColor& boundaryColor)
480         {
481             QPixmap pm = ui->frame->pixmap();
```

```
482             if (pm.isNull()) return;
483 
484             const int cells = this->grid_size / this->grid_box;
485             int gx0 = qFloor(x / (double)this->grid_box);
486             int gy0 = qFloor(y / (double)this->grid_box);
487             if (gx0 < 0 || gy0 < 0 || gx0 >= cells || gy0 >= cells) return;
488 
489             QImage img = pm.toImage();
490 
491             QSet<QPair<int,int>> visited;
492             QVector<QPoint> stack;
493             stack.append(QPoint(gx0, gy0));
494 
495             QPainter painter(&pm);
496 
497             auto maybeAnimate = [&]() {
498                 ui->frame->setPixmap(pm);
499                 QCoreApplication::processEvents();
500                 QThread::msleep(15);
501             };
502 
503             auto push4 = [&](int gx, int gy) {
504                 stack.append(QPoint(gx + 1, gy));
505                 stack.append(QPoint(gx - 1, gy));
506                 stack.append(QPoint(gx, gy + 1));
507                 stack.append(QPoint(gx, gy - 1));
508             };
509             auto push8 = [&](int gx, int gy) {
510                 push4(gx, gy);
511                 stack.append(QPoint(gx + 1, gy + 1));
512                 stack.append(QPoint(gx - 1, gy + 1));
513                 stack.append(QPoint(gx + 1, gy - 1));
514                 stack.append(QPoint(gx - 1, gy - 1));
515             };
516 
517             while (!stack.isEmpty()) {
518                 QPoint p = stack.takeLast();
519                 int gx = p.x();
520                 int gy = p.y();
521                 if (gx < 0 || gy < 0 || gx >= cells || gy >= cells) continue;
522 
523                 QPair<int,int> key(gx, gy);
524                 if (visited.contains(key)) continue;
525                 visited.insert(key);
526 
527                 int cx = gx * this->grid_box + this->grid_box / 2;
528                 int cy = gy * this->grid_box + this->grid_box / 2;
529 
530                 QColor cur = img.pixelColor(cx, cy);
531                 if (cur == newColor) continue;
532                 if (cur == boundaryColor) continue;
533 
```

```
534                 QRect rect(gx * this->grid_box, gy * this->grid_box, this->
                        grid_box, this->grid_box);
535                 painter.fillRect(rect, QBrush(newColor, Qt::SolidPattern));
536 
537                 maybeAnimate();
538 
539                 if (connectivityMode == Connectivity::Eight) {
540                     push8(gx, gy);
541                 } else {
542                     push4(gx, gy);
543                 }
544             }
545 
546             painter.end();
547             ui->frame->setPixmap(pm);
548         }
549 
550         void MainWindow::scanlineFill(const Polygon& poly, const QColor& fillColor
                , bool animate)
551         {
552             if (poly.vertices.size() < 3) return;
553 
554             int ymin = poly.vertices.front().y();
555             int ymax = ymin;
556             for (const QPoint &v : poly.vertices) {
557                 ymin = std::min(ymin, v.y());
558                 ymax = std::max(ymax, v.y());
559             }
560 
561             QPixmap pm = ui->frame->pixmap();
562             if (pm.isNull()) return;
563 
564             const int cells = this->grid_size / this->grid_box;
565             const int n = poly.vertices.size();
566 
567             QPainter painter(&pm);
568             QImage   img = pm.toImage();
569 
570             auto maybeAnimate = [&]() {
571                 if (animate) {
572                     ui->frame->setPixmap(pm);
573                     QCoreApplication::processEvents();
574                     QThread::msleep(20);
575                 }
576             };
577 
578             const double epsx = 1e-6;
579 
580             for (int y = ymin; y <= ymax; ++y) {
581                 QVector<double> interX;
582                 interX.reserve(n);
583 
```

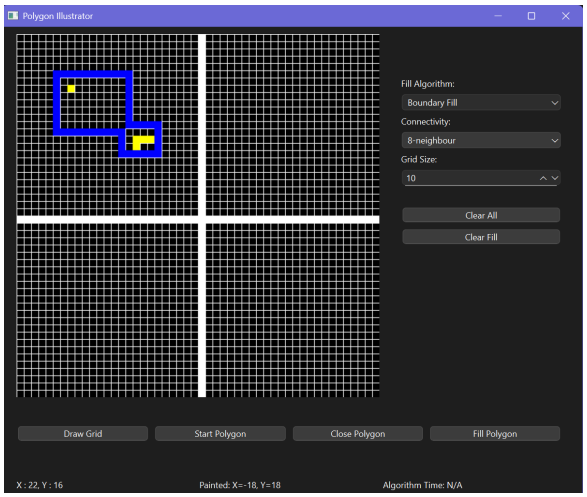
```
584                 for (int i = 0; i < n; ++i) {
585                     const QPoint p1 = poly.vertices[i];
586                     const QPoint p2 = poly.vertices[(i + 1) % n];
587                     const int x1 = p1.x(), y1 = p1.y();
588                     const int x2 = p2.x(), y2 = p2.y();
589 
590                     if (y1 == y2) continue;
591 
592                     const int yminEdge = std::min(y1, y2);
593                     const int ymaxEdge = std::max(y1, y2);
594                     if (y >= yminEdge && y < ymaxEdge) {
595                         const double t  = ((y + 0.5) - y1) / double(y2 - y1);
596                         const double ix = x1 + t * (x2 - x1);
597                         interX.append(ix);
598                     }
599                 }
600 
601                 if (interX.isEmpty()) continue;
602                 std::sort(interX.begin(), interX.end());
603 
604                 for (int k = 0; k + 1 < interX.size(); k += 2) {
605                     const double L = std::min(interX[k],   interX[k+1]);
606                     const double R = std::max(interX[k],   interX[k+1]);
607 
608                     int xLeft  = int(std::ceil (L + epsx));
609                     int xRight = int(std::floor(R - epsx));
610                     if (xRight < xLeft) continue;
611 
612                     for (int lx = xLeft; lx <= xRight; ++lx) {
613                         int gx = lx + this->center;
614                         int gy = this->center - y;
615                         if (gx < 0 || gy < 0 || gx >= cells || gy >= cells)
                                continue;
616 
617                         const int cx = gx * this->grid_box + this->grid_box / 2;
618                         const int cy = gy * this->grid_box + this->grid_box / 2;
619                         const QColor cur = img.pixelColor(cx, cy);
620 
621                         if (cur == QColor(poly.col) ) continue;
622 
623                         QRect rect(gx * this->grid_box, gy * this->grid_box,
624                         this->grid_box, this->grid_box);
625                         painter.fillRect(rect, QBrush(fillColor, Qt::SolidPattern)
                                );
626                     }
627                     maybeAnimate();
628                 }
629             }
630 
631             painter.end();
632             ui->frame->setPixmap(pm);
633         }
```

```
634
635
636         void MainWindow::on_fill_polygon_clicked()
637         {
638             if (polygons.isEmpty()) return;
639
640             Polygon &poly = polygons.last();
641
642             if (currentAlgo == Algo::Boundary) {
643                 if (lastPoint2 == QPoint(-1, -1)) return;
644                 poly.boundaryFilled = true;
645                 poly.boundaryCol = Qt::green;
646                 poly.boundarySeedPx = lastPoint2;
647                 boundaryFill(lastPoint2.x(), lastPoint2.y(), poly.boundaryCol,
                        QColor(poly.col));
648             } else if (currentAlgo == Algo::Flood) {
649                 if (lastPoint2 == QPoint(-1, -1)) return;
650                 poly.floodFilled = true;
651                 poly.floodCol = Qt::red;
652                 poly.floodSeedPx = lastPoint2;
653                 floodFill(lastPoint2.x(), lastPoint2.y(), poly.floodCol);
654             } else {
655                 poly.scanFilled = true;
656                 poly.scanCol = Qt::magenta;
657                 scanlineFill(poly, poly.scanCol, true);
658             }
659         }
660
661         void MainWindow::on_clear_fill_clicked()
662         {
663
664             for (auto &poly : polygons) {
665                 poly.floodFilled = false;     poly.floodSeedPx = QPoint(-1,-1);
666                 poly.boundaryFilled = false;  poly.boundarySeedPx = QPoint(-1,-1);
667                 poly.scanFilled = false;
668             }
669             refreshCanvas();
670             redraw();
671         }
672
673         void MainWindow::on_algo_select_currentIndexChanged(int)
674         {
675             const QString txt = ui->algo_select->currentText().toLower();
676             if (txt.contains("boundary")) { currentAlgo = Algo::Boundary; return;
                    }
677             if (txt.contains("flood"))    { currentAlgo = Algo::Flood;    return;
                    }
678             if (txt.contains("scan"))     { currentAlgo = Algo::Scan;     return;
                    }
679             currentAlgo = Algo::Flood;
680         }
```

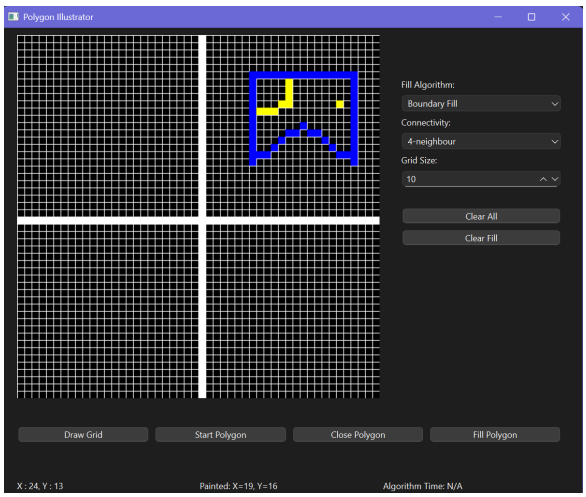
my_label.cpp

```
#include "my_label.h"

my_label::my_label(QWidget *parent) : QLabel(parent)
{
    this->setMouseTracking(true);
}

void my_label::mouseMoveEvent(QMouseEvent *ev)
{
    QPoint pos = ev->pos();
    if (pos.x() >= 0 && pos.y() >= 0 && pos.x() < this->width() && pos.y()
         < this->height()) {
        emit sendMousePosition(pos);
    }
}

void my_label::mousePressEvent(QMouseEvent *ev)
{
    if (ev->button() == Qt::LeftButton) {
        x = ev->x();
        y = ev->y();
        emit Mouse_Pos();
    }
}
```

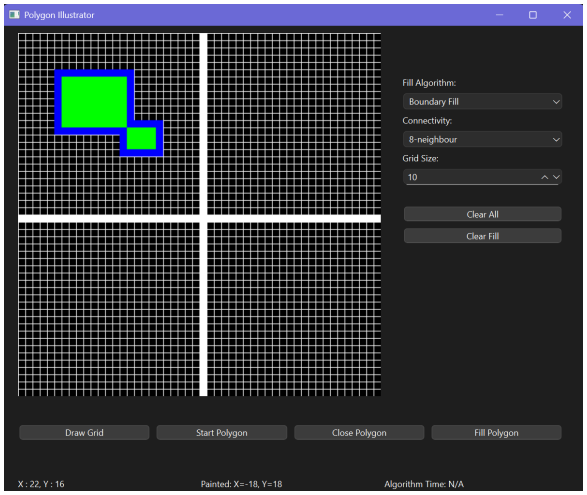
Outputs



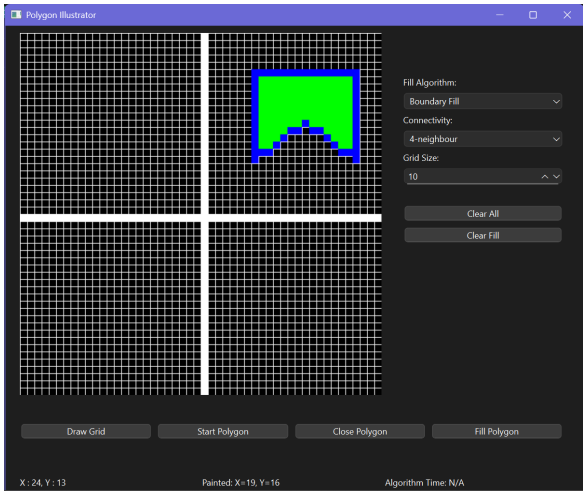
(a) boundaryfill_before_filling_8_neighbour



(b) boundaryfill_before_filling_4_neighbour

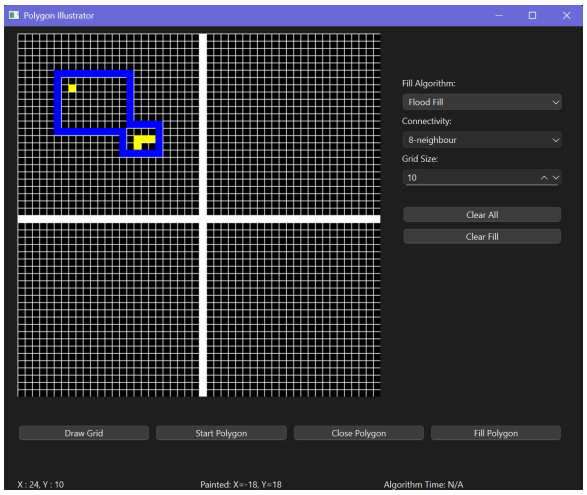


(c) boundaryfill_after_filling_8_neighbour

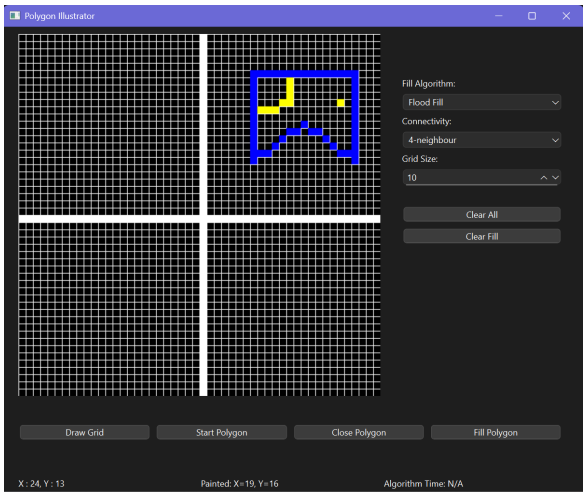


(d) boundaryfill_after_filling_4_neighbour

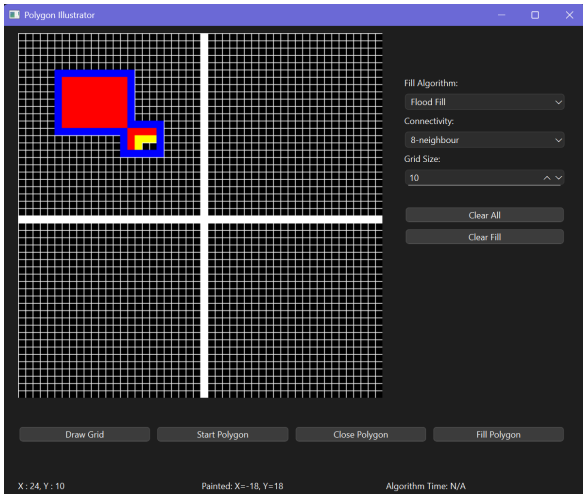
Figure 9: Boundary fill outputs.



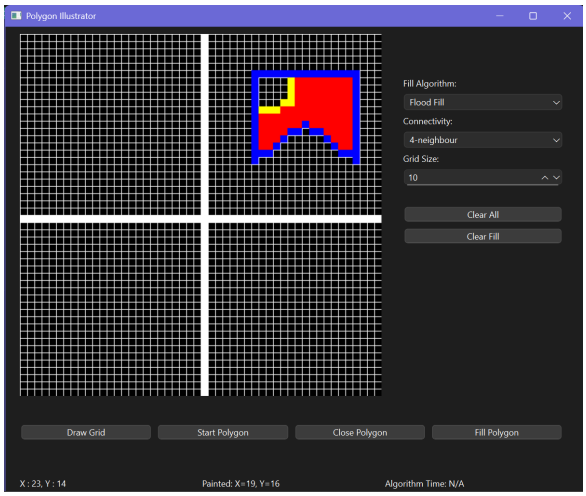
(a) floodfill_before_filling_8_neighbour



(b) floodfill_before_filling_4_neighbour

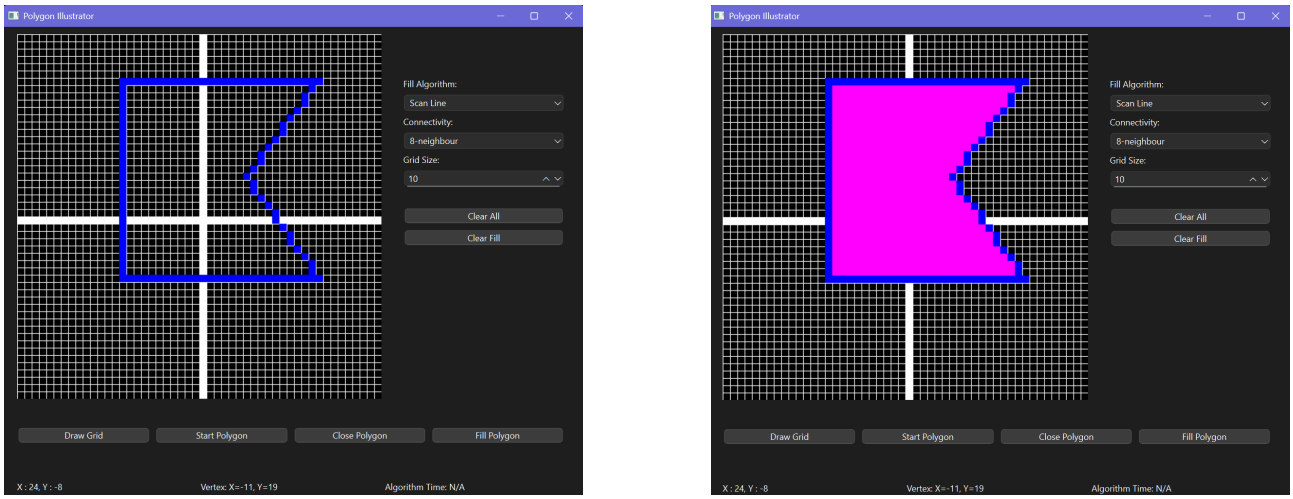


(c) floodfill_after_filling_8_neighbour



(d) floodfill_after_filling_4_neighbour

Figure 10: Flood fill outputs.



(a) scanline_before_filling (b) scanline_after_filling

Figure 11: Scanline fill outputs.

Output Explanation

For this assignment I interactively drew simple test polygons and regions on the grid and then applied the three implemented filling strategies: flood fill, boundary fill, and scanline fill. The behaviour of each algorithm is clearly visible in the rendered outputs.

Flood fill (4- and 8-neighbour connectivity). For flood fill I first painted a closed blue polygon and then manually drew an interior *yellow* barrier inside the polygon. I then chose a seed point in one of the blue interior regions and applied flood fill with red as the new region colour.

With **4-neighbour connectivity**, the fill spreads only through the 4-connected neighbours

$$N_4(x, y) = \{(x \pm 1, y), (x, y \pm 1)\},$$

and at every step the implementation checks that the current pixel belongs to the same region colour and is not yellow. As a result, the red region expands until it touches either the blue boundary or any yellow cell, and stops there. The yellow barrier therefore acts as a strict blocking curve, and the output shows the interior region subdivided into disjoint red components separated by yellow lines.

When I switch to **8-neighbour connectivity**,

$$N_8(x, y) = N_4(x, y) \cup \{(x \pm 1, y \pm 1)\},$$

the fill is allowed to propagate diagonally as well. In regions where there are diagonal gaps of size one cell between the yellow barrier and the boundary, the 8-connected flood fill can leak through those diagonal connections, and those subregions are now filled. However, any cell that is actually yellow is still treated as an obstacle, so the fill never overwrites the yellow colour. The final output clearly demonstrates that flood fill respects the interior barrier colour and cannot "cross" yellow, regardless of the chosen connectivity.

Boundary fill (4- and 8-neighbour connectivity). For boundary fill I used the same polygons but with a designated boundary colour (for example blue), and a seed point strictly inside the polygon. In this case the region predicate is different: the algorithm expands from the seed until it meets the boundary colour, and it does not treat yellow as an obstacle.

With **4-neighbour boundary fill**, the interior is filled with green by visiting all pixels whose colour is not the boundary colour and not already the fill colour. The yellow barrier is therefore overwritten just like any other interior colour, because the decision test only compares against the blue boundary. The resulting output shows a completely green interior with the blue outline intact and the original yellow barrier no longer visible.

With **8-neighbour boundary fill**, diagonal neighbours are also considered. As with flood fill, this allows the algorithm to reach diagonally connected interior cells more quickly, but again there is no special treatment of yellow: the fill propagates through, only stopping at the boundary pixels themselves. The two outputs together make it clear that, unlike flood fill, boundary fill can "cross" interior colours such as yellow and will fill the entire connected interior region up to the polygon boundary.

Scanline fill on a concave polygon. The algorithm computes, for each integer scanline y between the polygon's minimum and maximum y-coordinates, all intersections of that horizontal line with polygon edges,

sorts the intersection x-coordinates, and fills alternating intervals between successive pairs according to the even–odd rule.

In the output this produces horizontal magenta bands that correctly cover the entire interior of the concave polygon, including the indented regions, without spilling outside or leaving gaps. The scanline method correctly handles the non-convex shape because the alternating pairing of intersection points ensures that only the intended interior segments along each scanline are filled.

Assignment 5: 2D Transformations

Problem Statement

Draw a closed polygon and implement different transformation functions (with respect to the origin) on it:

- Translation
- Rotation
- Scaling
- Shear
- Reflection with respect to the x- and y-axes

Extend the algorithm to apply the transformations successively on the same object using homogeneous coordinates and matrix multiplication, including:

- Reflection with respect to an arbitrary line
- Rotation with respect to an arbitrary point

Code and Theoretical Explanation

In this assignment I first draw a polygon on a discrete Cartesian grid and then apply a sequence of 2D affine transformations to its vertices. Each vertex

$$\mathbf{p} = (x, y)$$

is internally represented in homogeneous coordinates as the column vector

$$\tilde{\mathbf{p}} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

so that every transformation can be written as a 3×3 matrix $\mathbf{T}$ and applied by a single matrix–vector multiplication

$$\tilde{\mathbf{p}}' = \mathbf{T}\,\tilde{\mathbf{p}}.$$

This matches the way `QTransform` stores transformations and the way I use it in `applyTransform(const QTransform& T)`, where each vertex is mapped via

```
QPointF r = T.map(QPointF(v.x(), v.y()));
out.push_back(QPoint(qRound(r.x()), qRound(r.y())));
```

corresponding exactly to multiplying $\tilde{\mathbf{p}}$ by the composed matrix $\mathbf{T}$.

Homogeneous matrices for basic 2D transformations. Using homogeneous coordinates, all basic 2D transformations are expressed as affine matrices of the form

$$\mathbf{T} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix}, \qquad \tilde{\mathbf{p}}' = \mathbf{T}\,\tilde{\mathbf{p}} = \begin{bmatrix} ax + by + t_x \\ cx + dy + t_y \\ 1 \end{bmatrix}.$$

Translation. A translation by (t_x, t_y) adds a constant offset to every vertex:

$$x' = x + t_x, \qquad y' = y + t_y.$$

In homogeneous form,

$$\mathbf{T}_{\text{trans}}(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}, \qquad \tilde{\mathbf{p}}' = \mathbf{T}_{\text{trans}}\tilde{\mathbf{p}}.$$

In code, this is implemented in `on_btnTranslate_clicked()`:

```
QTransform T;
T.translate(ui->spinTx->value(), ui->spinTy->value());
applyTransform(T);
```

which internally constructs the same matrix $\mathbf{T}_{\text{trans}}$.

Rotation about the origin. For a rotation by angle θ about the origin, the analytic equations are

$$x' = x\cos\theta - y\sin\theta,$$
$$y' = x\sin\theta + y\cos\theta.$$

The corresponding homogeneous matrix is

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Because the screen y-axis grows downwards, I supply a negative angle in `on_btnRotate_clicked()`:

```
QTransform T;
T.rotate(-ui->spinAngle->value());
applyTransform(T);
```

so that a positive angle in the UI still corresponds to a counterclockwise rotation in the mathematical xy-plane.

Scaling. Scaling by factors s_x and s_y along the x and y axes is given by

$$x' = s_x x, \qquad y' = s_y y,$$

and the matrix form is

$$\mathbf{S}(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

To avoid degenerate behaviour when the user sets negative slider values, I convert an input scale s into an effective factor

$$s_{\text{eff}} = \begin{cases} \dfrac{1}{1 + (-s)}, & s < 0, \\ s, & s \geq 0, \end{cases}$$

in `effectiveScale(double s)`, and then apply scaling about the current pivot (polygon centroid) in `on_btnScale_clicked()`:

```
QPointF c  = currentPivot();
double sx  = effectiveScale(ui->spinSx->value());
double sy  = effectiveScale(ui->spinSy->value());

QTransform T;
T.translate(c.x(), c.y());
T.scale(sx, sy);
T.translate(-c.x(), -c.y());
applyTransform(T);
```

Mathematically, scaling about the centroid $\mathbf{c} = (c_x, c_y)$ is the composition

$$\mathbf{T}_{\text{scale,centroid}} = \mathbf{T}(c_x, c_y)\, \mathbf{S}(s_x, s_y)\, \mathbf{T}(-c_x, -c_y).$$

Shear. A shear transformation keeps one axis fixed while sliding points along the other. With shear factors h_x (horizontal shear proportional to y) and h_y (vertical shear proportional to x), the equations are

$$x' = x + h_x y, \qquad y' = y + h_y x,$$

which correspond to

$$\mathbf{H}(h_x, h_y) = \begin{bmatrix} 1 & h_x & 0 \\ h_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

In code I again shear about the polygon centroid:

```
QPointF c  = currentPivot();
double shx = ui->spinShx->value();
double shy = ui->spinShy->value();

QTransform T;
T.translate(c.x(), c.y());
T.shear(shx, shy);
T.translate(-c.x(), -c.y());
applyTransform(T);
```

which implements $\mathbf{T}(c)\, \mathbf{H}(h_x, h_y)\, \mathbf{T}(-c)$.

Reflection about coordinate axes. Reflection about the x-axis maps $(x, y) \mapsto (x, -y)$. Its matrix is

$$\mathbf{R}_{x\text{-axis}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Similarly, reflection about the y-axis maps $(x, y) \mapsto (-x, y)$ with

$$\mathbf{R}_{y\text{-axis}} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

These are created directly in `on_btnReflectX_clicked()` and `on_btnReflectY_clicked()`:

```
QTransform T( 1, 0, 0,
              0,-1, 0, 0, 0, 1); // reflect about X
QTransform T(-1, 0, 0,
              0, 1, 0, 0, 0, 1); // reflect about Y
```

Rotation about an arbitrary point. To rotate about an arbitrary pivot $\mathbf{p}_0 = (p_x, p_y)$, I first translate the pivot to the origin, rotate, and then translate back:

$$\tilde{\mathbf{p}}' = \mathbf{T}(p_x, p_y)\, \mathbf{R}(\theta)\, \mathbf{T}(-p_x, -p_y)\, \tilde{\mathbf{p}}.$$

This composition is implemented in `on_btnRotateAboutPoint_clicked()` using the last grid point picked by the user as pivot:

```
QTransform T;
T.translate(lastPick.x(), lastPick.y());
T.rotate(-ui->spinAngle->value());
T.translate(-lastPick.x(), -lastPick.y());
applyTransform(T);
```

so each vertex is rotated around the chosen point without changing the pivot itself.

Reflection about an arbitrary line. For reflection with respect to an arbitrary line, I allow the user to draw a helper line through two points $a = (a_x, a_y)$ and $b = (b_x, b_y)$. For the special cases where the line has slope ± 1, I use closed-form formulas in grid coordinates. For example, for a line of slope $+1$ of the form $y = x + k$, each vertex (x, y) is reflected to

$$x' = y - k, \qquad y' = x + k,$$

as implemented in `reflectSpecialIfSlopePM1()` by constructing new integer points.

For general lines, I work with unit direction vector

$$\mathbf{u} = \frac{1}{\|\mathbf{b} - \mathbf{a}\|} \begin{bmatrix} b_x - a_x \\ b_y - a_y \end{bmatrix} = \begin{bmatrix} u_x \\ u_y \end{bmatrix}.$$

The 2×2 reflection matrix about the line through the origin with direction $\mathbf{u}$ is

$$\mathbf{M} = \begin{bmatrix} 2u_x^2 - 1 & 2u_x u_y \\ 2u_x u_y & 2u_y^2 - 1 \end{bmatrix},$$

which comes from reflecting the vector after decomposing it into components parallel and perpendicular to $\mathbf{u}$. For a line that does not pass through the origin, I first translate each vertex so that $\mathbf{a}$ moves to the origin, apply $\mathbf{M}$, and then translate back:

$$\mathbf{v}' = \mathbf{a} + \mathbf{M}(\mathbf{v} - \mathbf{a}).$$

This is exactly what I implement in `reflectPointAboutLine(const QPointF& v,...)`:

```
double ux = b.x() - a.x();
double uy = b.y() - a.y();
double invLen = 1.0 / std::sqrt(ux*ux + uy*uy);
ux *= invLen; uy *= invLen;

double r11 = 2.0*ux*ux - 1.0;
double r12 = 2.0*ux*uy;
double r21 = r12;
double r22 = 2.0*uy*uy - 1.0;

double wx = v.x() - a.x();
double wy = v.y() - a.y();

double rx = r11*wx + r12*wy;
double ry = r21*wx + r22*wy;

return QPointF(a.x() + rx, a.y() + ry);
```

and then apply it to each vertex in `on_btnReflectDrawnLine_clicked()`.

Matrix composition and reuse. All transformations above are composed as homogeneous matrices and then applied in one pass to the vertex list. Algebraically, if I want to scale, then rotate, then translate a polygon, the final transformation is

$$\mathbf{T}_{\text{final}} = \mathbf{T}_{\text{trans}}\,\mathbf{R}(\theta)\,\mathbf{S}(s_x, s_y),$$

and every vertex is updated as $\tilde{\mathbf{p}}' = \mathbf{T}_{\text{final}}\,\tilde{\mathbf{p}}$. `QTransform` internally performs this matrix multiplication whenever I chain operations using `translate`, `rotate`, `scale`, `shear`, etc., and then `applyTransform()` maps all vertices using the single composed matrix.

Code

transformation.pro

```
        TEMPLATE = app
        TARGET = transformation

```

```
QT += widgets core gui
CONFIG += c++17
CONFIG += warn_on
macx: CONFIG += app_bundle

SOURCES += \
main.cpp \
mainwindow.cpp \
my_label.cpp

HEADERS += \
mainwindow.h \
my_label.h

FORMS += \
mainwindow.ui

INCLUDEPATH += .
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QVector>
#include <QPoint>
#include <QColor>
#include <QTransform>
#include <QSet>
#include <QTimer>
#include <climits>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT
    public:
    explicit MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

    protected:
    bool eventFilter(QObject* obj, QEvent* event) override;

    private slots:
    void Mouse_Pressed();
    void showMousePosition(QPoint &pos);

    void on_btnDrawGrid_clicked();
    void on_spinGridSize_valueChanged(int value);
```

```
33              void on_btnClear_clicked();
34
35              void on_btnClosePolygon_clicked();
36
37              void on_btnRevertOriginal_clicked();
38
39              void on_btnTranslate_clicked();
40              void on_btnRotate_clicked();
41              void on_btnRotateAboutPoint_clicked();
42              void on_btnScale_clicked();
43              void on_btnShear_clicked();
44              void on_btnReflectX_clicked();
45              void on_btnReflectY_clicked();
46              void on_btnDrawLine_clicked();
47              void on_btnReflectDrawnLine_clicked();
48
49              void nudgeUp();
50              void nudgeLeft();
51              void nudgeDown();
52              void nudgeRight();
53
54              private:
55              Ui::MainWindow *ui = nullptr;
56
57              int  cellSize  = 20;
58              int  gridCols  = 0, gridRows = 0;
59              int  centerX   = 0, centerY = 0;
60              bool gridDrawn = false;
61
62              QColor bgColor    = QColor(11,21,64);
63              QColor gridColor  = QColor(240,240,240);
64              QColor axisColor  = QColor(255,255,255);
65              QColor polyColor  = QColor(255,208,96);
66              QColor origColor  = QColor(140,255,200);
67              QColor pickFill   = QColor(255,105,180);
68              QColor lineColor  = QColor(80,220,255);
69
70              int    scrX = 0, scrY = 0;
71              QPoint lastPick = QPoint(INT_MIN, INT_MIN);
72              QPoint prevPick = QPoint(INT_MIN, INT_MIN);
73
74              QVector<QPoint> polygon;
75              QVector<QPoint> originalPolygon;
76              bool hasOriginal   = false;
77              bool polygonClosed = false;
78
79              bool   hasDrawnLine = false;
80              QPoint lineP1, lineP2;
81
82              void repaintFreshGrid();
83              void drawAxes(class QPainter& p);
84              void refreshScene();
```

```
85              void fillPickedCellPx(int px, int py);
86
87              QPoint screenPxToGridXY(int px, int py) const;
88              QPoint gridXYToCellTL(int X, int Y) const;
89
90              void drawPolygon(class QPainter& p);
91              void drawPoly(const QVector<QPoint>& poly, class QPainter& p, const
                    QColor& color);
92              void drawHelperLine(class QPainter& p);
93              void rasterBresenham(int X1, int Y1, int X2, int Y2, QVector<QPoint>&
                    outCells);
94              inline void fillCellGrid(int gx, int gy, class QPainter& p);
95
96              void applyTransform(const QTransform& T);
97
98              QTransform reflectAboutTwoPointLine(const QPointF& p1, const QPointF&
                    p2);
99
100
101             bool hasValid(const QPoint& p) const { return p.x()!=INT_MIN && p.y()
                    !=INT_MIN; }
102             QPointF firstVertexPivot() const;
103             void updateOriginalUiState();
104
105             QPointF polygonCentroid(const QVector<QPoint>& poly) const;
106             QPointF currentPivot() const;
107
108
109             QPointF reflectPointAboutLine(const QPointF& v,
110             const QPointF& a,
111             const QPointF& b) const;
112             bool reflectSpecialIfSlopePM1(const QPoint& a, const QPoint& b,
113             QVector<QPoint>& out) const;
114
115             double effectiveScale(double s) const;
116
117             void translateByCells(int dx, int dy);
118
119             QSet<int> pressedKeys;
120             QTimer*   nudgeTimer = nullptr;
121             void scheduleNudge();
122             void performNudge();
123         };
124
125         #endif
```

my_label.h

```
1           #ifndef MY_LABEL_H
2           #define MY_LABEL_H
3
4           #include <QLabel>
5           #include <QMouseEvent>
```

```

        class my_label : public QLabel
        {
            Q_OBJECT
            public:
            explicit my_label(QWidget *parent = nullptr);

            protected:
            void mouseMoveEvent(QMouseEvent *ev) override;
            void mousePressEvent(QMouseEvent *ev) override;

            signals:
            void sendMousePosition(QPoint&);
            void Mouse_Pos();

            private:
            int lastX = -1, lastY = -1;
        };

        #endif
```

main.cpp

```
        #include "mainwindow.h"
        #include <QApplication>

        int main(int argc, char *argv[])
        {
            QApplication a(argc, argv);
            MainWindow w;
            w.show();
            return a.exec();
        }
```

mainwindow.cpp

```
        #include "mainwindow.h"
        #include "ui_mainwindow.h"

        #include <QPixmap>
        #include <QPainter>
        #include <QPen>
        #include <QtMath>
        #include <QApplication>
        #include <QKeyEvent>
        #include <QTimer>
        #include <climits>
        #include <cmath>

        MainWindow::MainWindow(QWidget *parent)
        : QMainWindow(parent), ui(new Ui::MainWindow)
        {
            ui->setupUi(this);

```

```
19              QPixmap pm(ui->frame->width(), ui->frame->height());
20              pm.fill(bgColor);
21              ui->frame->setPixmap(pm);
22
23              connect(ui->frame, SIGNAL(Mouse_Pos()), this, SLOT(Mouse_Pressed()));
24              connect(ui->frame, SIGNAL(sendMousePosition(QPoint&)), this, SLOT(
                    showMousePosition(QPoint&)));
25
26              cellSize = ui->spinGridSize->value();
27              ui->lblHover->setText("Hover: X: 0, Y: 0");
28              ui->lblInfo->setText("Click to add vertices. Close polygon to freeze '
                    original'.");
29              updateOriginalUiState();
30
31              qApp->installEventFilter(this);
32              nudgeTimer = new QTimer(this);
33              nudgeTimer->setSingleShot(false);
34              nudgeTimer->setInterval(80);
35              connect(nudgeTimer, &QTimer::timeout, this, &MainWindow::performNudge)
                    ;
36          }
37
38          MainWindow::~MainWindow()
39          {
40              delete ui;
41          }
42
43          bool MainWindow::eventFilter(QObject* obj, QEvent* event)
44          {
45              Q_UNUSED(obj);
46              if (event->type() == QEvent::KeyPress) {
47                  auto* e = static_cast<QKeyEvent*>(event);
48                  if (e->isAutoRepeat()) return true;
49                  const int k = e->key();
50                  if (k==Qt::Key_W || k==Qt::Key_A || k==Qt::Key_S || k==Qt::Key_D)
                        {
51                      pressedKeys.insert(k);
52                      scheduleNudge();
53                      return true;
54                  }
55              } else if (event->type() == QEvent::KeyRelease) {
56                  auto* e = static_cast<QKeyEvent*>(event);
57                  if (e->isAutoRepeat()) return true;
58                  const int k = e->key();
59                  if (k==Qt::Key_W || k==Qt::Key_A || k==Qt::Key_S || k==Qt::Key_D)
                        {
60                      pressedKeys.remove(k);
61                      if (pressedKeys.isEmpty())
62                      nudgeTimer->stop();
63                      return true;
64                  }
65              }
```

```
66              return false;
67          }
68
69          void MainWindow::scheduleNudge()
70          {
71              performNudge();
72              if (!nudgeTimer->isActive())
73              nudgeTimer->start();
74          }
75
76          void MainWindow::performNudge()
77          {
78              const int dx = (pressedKeys.contains(Qt::Key_D) ? 1 : 0)
79              - (pressedKeys.contains(Qt::Key_A) ? 1 : 0);
80              const int dy = (pressedKeys.contains(Qt::Key_W) ? 1 : 0)
81              - (pressedKeys.contains(Qt::Key_S) ? 1 : 0);
82
83              if (dx == 0 && dy == 0) {
84                  return;
85              }
86              translateByCells(dx, dy);
87          }
88
89          void MainWindow::showMousePosition(QPoint &pos)
90          {
91              scrX = pos.x();
92              scrY = pos.y();
93              if (!gridDrawn || cellSize <= 0) {
94                  ui->lblHover->setText("Hover: X: --, Y: --");
95                  return;
96              }
97              QPoint g = screenPxToGridXY(scrX, scrY);
98              ui->lblHover->setText(QString("Hover: X: %1, Y: %2").arg(g.x()).arg(g.
                    y()));
99          }
100
101         void MainWindow::Mouse_Pressed()
102         {
103             if (!gridDrawn) repaintFreshGrid();
104
105             QPoint g = screenPxToGridXY(scrX, scrY);
106
107             if (hasValid(lastPick)) prevPick = lastPick;
108             lastPick = g;
109
110             if (!polygonClosed) {
111                 polygon.push_back(g);
112                 ui->lblInfo->setText(QString("Vertices: %1 | Last pick: (%2,%3)")
113                 .arg(polygon.size()).arg(g.x()).arg(g.y()));
114             } else {
115                 ui->lblInfo->setText(QString("Last pick: (%1,%2)").arg(g.x()).arg(
                        g.y()));
```

```
116             }
117
118             fillPickedCellPx(scrX, scrY);
119         }
120
121         void MainWindow::on_btnDrawGrid_clicked()
122         {
123             repaintFreshGrid();
124             polygon.clear();
125             originalPolygon.clear();
126             hasOriginal   = false;
127             polygonClosed = false;
128
129             lastPick = prevPick = QPoint(INT_MIN, INT_MIN);
130             hasDrawnLine = false;
131
132             updateOriginalUiState();
133             ui->lblInfo->setText("Grid ready. Click to add vertices.");
134         }
135
136         void MainWindow::on_spinGridSize_valueChanged(int value)
137         {
138             if (value < 5) return;
139             cellSize = value;
140             refreshScene();
141         }
142
143         void MainWindow::on_btnClear_clicked()
144         {
145             polygon.clear();
146             originalPolygon.clear();
147             hasOriginal   = false;
148             polygonClosed = false;
149
150             lastPick = prevPick = QPoint(INT_MIN, INT_MIN);
151             hasDrawnLine = false;
152
153             refreshScene();
154             updateOriginalUiState();
155             ui->lblInfo->setText("Cleared. Click to add vertices.");
156         }
157
158         void MainWindow::repaintFreshGrid()
159         {
160             QPixmap pm(ui->frame->width(), ui->frame->height());
161             pm.fill(bgColor);
162
163             gridCols = qMax(1, pm.width()  / cellSize);
164             gridRows = qMax(1, pm.height() / cellSize);
165             centerX  = gridCols / 2;
166             centerY  = gridRows / 2;
167
```

```
168             QPainter p(&pm);
169             p.setRenderHint(QPainter::Antialiasing, false);
170 
171             QPen gridPen(gridColor);
172             gridPen.setWidth(1);
173             p.setPen(gridPen);
174             for (int c = 0; c <= gridCols; ++c)
175             p.drawLine(c * cellSize, 0, c * cellSize, gridRows * cellSize);
176             for (int r = 0; r <= gridRows; ++r)
177             p.drawLine(0, r * cellSize, gridCols * cellSize, r * cellSize);
178 
179             drawAxes(p);
180 
181             p.end();
182             ui->frame->setPixmap(pm);
183             gridDrawn = true;
184         }
185 
186         void MainWindow::drawAxes(QPainter& p)
187         {
188             p.setPen(Qt::NoPen);
189             p.setBrush(axisColor);
190             for (int r = 0; r < gridRows; ++r)
191             p.fillRect(QRect(centerX * cellSize, r * cellSize, cellSize, cellSize)
                    , axisColor);
192             for (int c = 0; c < gridCols; ++c)
193             p.fillRect(QRect(c * cellSize, centerY * cellSize, cellSize, cellSize)
                    , axisColor);
194         }
195 
196         void MainWindow::refreshScene()
197         {
198             repaintFreshGrid();
199 
200             QPixmap pm = ui->frame->pixmap();
201             if (pm.isNull()) return;
202 
203             QPainter p(&pm);
204             p.setRenderHint(QPainter::Antialiasing, false);
205 
206             drawHelperLine(p);
207             drawPolygon(p);
208 
209             p.end();
210             ui->frame->setPixmap(pm);
211         }
212 
213         void MainWindow::fillPickedCellPx(int px, int py)
214         {
215             QPixmap pm = ui->frame->pixmap();
216             if (pm.isNull() || cellSize <= 0) return;
217             QPainter painter(&pm);
```

```
218             painter.setRenderHint(QPainter::Antialiasing, false);
219             painter.setPen(Qt::NoPen);
220 
221             const int cx = (px / cellSize) * cellSize;
222             const int cy = (py / cellSize) * cellSize;
223             painter.fillRect(QRect(cx, cy, cellSize, cellSize), pickFill);
224             painter.end();
225             ui->frame->setPixmap(pm);
226         }
227 
228         QPoint MainWindow::screenPxToGridXY(int px, int py) const
229         {
230             const int gx = px / cellSize;
231             const int gy = py / cellSize;
232             const int X  = gx - centerX;
233             const int Y  = -(gy - centerY);
234             return QPoint(X, Y);
235         }
236 
237         QPoint MainWindow::gridXYToCellTL(int X, int Y) const
238         {
239             const int gx = X + centerX;
240             const int gy = centerY - Y;
241             return QPoint(gx * cellSize, gy * cellSize);
242         }
243 
244         inline void MainWindow::fillCellGrid(int gx, int gy, QPainter& p)
245         {
246             if (gx >= 0 && gx < gridCols && gy >= 0 && gy < gridRows)
247             p.fillRect(QRect(gx * cellSize, gy * cellSize, cellSize, cellSize), p.
                    brush());
248         }
249 
250         void MainWindow::rasterBresenham(int X1, int Y1, int X2, int Y2, QVector<
                QPoint>& out)
251         {
252             out.clear();
253             int gx1 = X1 + centerX;
254             int gy1 = centerY - Y1;
255             int gx2 = X2 + centerX;
256             int gy2 = centerY - Y2;
257 
258             int dx = std::abs(gx2 - gx1);
259             int dy = std::abs(gy2 - gy1);
260             int sx = (gx1 < gx2) ? 1 : -1;
261             int sy = (gy1 < gy2) ? 1 : -1;
262             int err = dx - dy;
263 
264             while (true) {
265                 out.push_back(QPoint(gx1, gy1));
266                 if (gx1 == gx2 && gy1 == gy2) break;
267                 int e2 = 2 * err;
```

```
268                 if (e2 > -dy) { err -= dy; gx1 += sx; }
269                 if (e2 <  dx) { err += dx; gy1 += sy; }
270             }
271         }
272
273         void MainWindow::drawPoly(const QVector<QPoint>& poly, QPainter& p, const
                QColor& color)
274         {
275             if (poly.isEmpty()) return;
276             p.setPen(Qt::NoPen);
277             p.setBrush(color);
278
279             for (const QPoint& v : poly) {
280                 const int gx = v.x() + centerX;
281                 const int gy = centerY - v.y();
282                 fillCellGrid(gx, gy, p);
283             }
284
285             QVector<QPoint> cells;
286             for (int i = 0; i + 1 < poly.size(); ++i) {
287                 rasterBresenham(poly[i].x(), poly[i].y(),
288                 poly[i+1].x(), poly[i+1].y(), cells);
289                 for (const QPoint& c : cells)
290                 fillCellGrid(c.x(), c.y(), p);
291             }
292         }
293
294         void MainWindow::drawHelperLine(QPainter& p)
295         {
296             if (!hasDrawnLine) return;
297             p.setPen(Qt::NoPen);
298             p.setBrush(lineColor);
299
300             QVector<QPoint> cells;
301             rasterBresenham(lineP1.x(), lineP1.y(), lineP2.x(), lineP2.y(), cells)
                    ;
302             for (const QPoint& c : cells)
303             fillCellGrid(c.x(), c.y(), p);
304         }
305
306         void MainWindow::drawPolygon(QPainter& p)
307         {
308             if (hasOriginal)
309             drawPoly(originalPolygon, p, origColor);
310             drawPoly(polygon, p, polyColor);
311         }
312
313         void MainWindow::on_btnClosePolygon_clicked()
314         {
315             if (polygon.size() >= 3 && polygon.front() != polygon.back())
316             polygon.push_back(polygon.front());
317
```

```
318             if (!hasOriginal && polygon.size() >= 4) {
319                 originalPolygon = polygon;
320                 hasOriginal     = true;
321                 polygonClosed   = true;
322                 ui->lblInfo->setText("Original frozen. You can transform the
                        working polygon.");
323                 updateOriginalUiState();
324             } else {
325                 polygonClosed = true;
326             }
327             refreshScene();
328         }
329
330         void MainWindow::on_btnRevertOriginal_clicked()
331         {
332             if (!hasOriginal) return;
333             polygon       = originalPolygon;
334             polygonClosed = true;
335
336             hasDrawnLine = false;
337
338             refreshScene();
339             ui->lblInfo->setText("Reverted to original polygon.");
340         }
341
342         void MainWindow::updateOriginalUiState()
343         {
344             ui->btnRevertOriginal->setEnabled(hasOriginal);
345         }
346
347         QPointF MainWindow::firstVertexPivot() const
348         {
349             if (!originalPolygon.isEmpty())
350             return QPointF(originalPolygon.front());
351             if (!polygon.isEmpty())
352             return QPointF(polygon.front());
353             return QPointF(0,0);
354         }
355
356         QPointF MainWindow::polygonCentroid(const QVector<QPoint>& poly) const
357         {
358             const int m = poly.size();
359             if (m == 0) return QPointF(0,0);
360
361             int n = m;
362             if (n > 1 && poly.front() == poly.back()) --n;
363
364             double A = 0.0, Cx = 0.0, Cy = 0.0;
365             for (int i = 0; i < n; ++i) {
366                 const QPoint& p0 = poly[i];
367                 const QPoint& p1 = poly[(i + 1) % n];
368                 const double cross = double(p0.x()) * p1.y() - double(p1.x()) * p0
```

```
                    .y();
369                 A  += cross;
370                 Cx += (p0.x() + p1.x()) * cross;
371                 Cy += (p0.y() + p1.y()) * cross;
372             }
373
374             if (qAbs(A) > 1e-9) {
375                 A *= 0.5;
376                 Cx /= (6.0 * A);
377                 Cy /= (6.0 * A);
378                 return QPointF(Cx, Cy);
379             }
380
381             int minx = poly[0].x(), maxx = poly[0].x();
382             int miny = poly[0].y(), maxy = poly[0].y();
383             for (int i = 1; i < n; ++i) {
384                 minx = qMin(minx, poly[i].x());
385                 maxx = qMax(maxx, poly[i].x());
386                 miny = qMin(miny, poly[i].y());
387                 maxy = qMax(maxy, poly[i].y());
388             }
389             return QPointF(0.5 * (minx + maxx), 0.5 * (miny + maxy));
390         }
391
392         QPointF MainWindow::currentPivot() const
393         {
394             if (!polygon.isEmpty())         return polygonCentroid(polygon);
395             if (!originalPolygon.isEmpty()) return polygonCentroid(originalPolygon
                    );
396             return QPointF(0,0);
397         }
398
399         double MainWindow::effectiveScale(double s) const
400         {
401             if (s < 0.0) return 1.0 / (1.0 + (-s));
402             return s;
403         }
404
405         void MainWindow::applyTransform(const QTransform& T)
406         {
407             if (polygon.isEmpty()) return;
408
409             QVector<QPoint> out;
410             out.reserve(polygon.size());
411
412             for (const QPoint& v : polygon) {
413                 QPointF r = T.map(QPointF(v.x(), v.y()));
414                 out.push_back(QPoint(qRound(r.x()), qRound(r.y())));
415             }
416             polygon = out;
417             refreshScene();
418         }
```

```
419
420         void MainWindow::on_btnTranslate_clicked()
421         {
422             QTransform T;
423             T.translate(ui->spinTx->value(), ui->spinTy->value());
424             applyTransform(T);
425         }
426
427         void MainWindow::on_btnRotate_clicked()
428         {
429             QTransform T;
430             T.rotate(-ui->spinAngle->value());
431             applyTransform(T);
432         }
433
434         void MainWindow::on_btnRotateAboutPoint_clicked()
435         {
436             if (!hasValid(lastPick)) {
437                 ui->lblInfo->setText("Pick a pivot on the grid (click) and try
                        again.");
438                 return;
439             }
440             const double deg = -ui->spinAngle->value();
441
442             QTransform T;
443             T.translate(lastPick.x(), lastPick.y());
444             T.rotate(deg);
445             T.translate(-lastPick.x(), -lastPick.y());
446             applyTransform(T);
447
448             ui->lblInfo->setText(QString("Rotated about (%1,%2)").arg(lastPick.x()
                    ).arg(lastPick.y()));
449         }
450
451         void MainWindow::on_btnScale_clicked()
452         {
453             const QPointF c  = currentPivot();
454             const double sxI = ui->spinSx->value();
455             const double syI = ui->spinSy->value();
456             const double sx  = effectiveScale(sxI);
457             const double sy  = effectiveScale(syI);
458
459             QTransform T;
460             T.translate(c.x(), c.y());
461             T.scale(sx, sy);
462             T.translate(-c.x(), -c.y());
463             applyTransform(T);
464
465             ui->lblInfo->setText(
466             QString("Scaled about centroid (%1,%2) | input(Sx,Sy)=(%3,%4) ->
                    effective=(%5,%6)")
467             .arg(c.x()).arg(c.y())
```

```
468             .arg(sxI,0,'g',4).arg(syI,0,'g',4)
469             .arg(sx,0,'g',4).arg(sy,0,'g',4));
470         }
471
472         void MainWindow::on_btnShear_clicked()
473         {
474             const QPointF c  = currentPivot();
475             const double shx = ui->spinShx->value();
476             const double shy = ui->spinShy->value();
477
478             QTransform T;
479             T.translate(c.x(), c.y());
480             T.shear(shx, shy);
481             T.translate(-c.x(), -c.y());
482             applyTransform(T);
483
484             ui->lblInfo->setText(QString("Sheared about centroid (%1,%2)").arg(c.x
                    ()).arg(c.y()));
485         }
486
487         void MainWindow::on_btnReflectX_clicked()
488         {
489             QTransform T(1, 0, 0, -1, 0, 0);
490             applyTransform(T);
491         }
492
493         void MainWindow::on_btnReflectY_clicked()
494         {
495             QTransform T(-1, 0, 0, 1, 0, 0);
496             applyTransform(T);
497         }
498
499         QPointF MainWindow::reflectPointAboutLine(const QPointF& v,
500         const QPointF& a,
501         const QPointF& b) const
502         {
503             double ux = b.x() - a.x();
504             double uy = b.y() - a.y();
505             const double len2 = ux*ux + uy*uy;
506             if (len2 < 1e-12) return v;
507
508             const double invLen = 1.0 / std::sqrt(len2);
509             ux *= invLen;  uy *= invLen;
510
511             const double r11 = 2.0*ux*ux - 1.0;
512             const double r12 = 2.0*ux*uy;
513             const double r21 = r12;
514             const double r22 = 2.0*uy*uy - 1.0;
515
516             const double wx = v.x() - a.x();
517             const double wy = v.y() - a.y();
518
```

```
519             const double rx = r11*wx + r12*wy;
520             const double ry = r21*wx + r22*wy;
521 
522             return QPointF(a.x() + rx, a.y() + ry);
523         }
524 
525         bool MainWindow::reflectSpecialIfSlopePM1(const QPoint& a, const QPoint& b
                ,
526         QVector<QPoint>& out) const
527         {
528             const int dx = b.x() - a.x();
529             const int dy = b.y() - a.y();
530             if (dx == 0 && dy == 0) return false;
531 
532             if (dy == dx) {
533                 const int k = a.y() - a.x();
534                 out.clear(); out.reserve(polygon.size());
535                 for (const QPoint& v : polygon) {
536                     const int xp = v.y() - k;
537                     const int yp = v.x() + k;
538                     out.push_back(QPoint(xp, yp));
539                 }
540                 return true;
541             }
542 
543             if (dy == -dx) {
544                 const int k = a.y() + a.x();
545                 out.clear(); out.reserve(polygon.size());
546                 for (const QPoint& v : polygon) {
547                     const int xp = -v.y() + k;
548                     const int yp = -v.x() + k;
549                     out.push_back(QPoint(xp, yp));
550                 }
551                 return true;
552             }
553 
554             return false;
555         }
556 
557         void MainWindow::on_btnDrawLine_clicked()
558         {
559             if (!hasValid(prevPick) || !hasValid(lastPick)) {
560                 ui->lblInfo->setText("Select two points (click twice) to define
                        the line.");
561                 return;
562             }
563             lineP1 = prevPick;
564             lineP2 = lastPick;
565             hasDrawnLine = true;
566             refreshScene();
567             ui->lblInfo->setText(QString("Helper line set: (%1,%2) -> (%3,%4)")
568             .arg(lineP1.x()).arg(lineP1.y())
```

```
569             .arg(lineP2.x()).arg(lineP2.y()));
570         }
571 
572         void MainWindow::on_btnReflectDrawnLine_clicked()
573         {
574             if (!hasDrawnLine) {
575                 ui->lblInfo->setText("No helper line. Click two points and press '
                        Draw Line' first.");
576                 return;
577             }
578 
579             const QPoint a = lineP1;
580             const QPoint b = lineP2;
581 
582             QVector<QPoint> out;
583 
584             if (reflectSpecialIfSlopePM1(a, b, out)) {
585                 polygon = out;
586                 refreshScene();
587                 ui->lblInfo->setText("Reflected about line with slope +/-1 (exact
                        swap).");
588                 return;
589             }
590 
591             out.clear(); out.reserve(polygon.size());
592             for (const QPoint& v : polygon) {
593                 const QPointF r = reflectPointAboutLine(QPointF(v), QPointF(a),
                        QPointF(b));
594                 out.push_back(QPoint(qRound(r.x()), qRound(r.y())));
595             }
596             polygon = out;
597             refreshScene();
598         }
599 
600         QTransform MainWindow::reflectAboutTwoPointLine(const QPointF& p1, const
                QPointF& p2)
601         {
602             const double dx = p2.x() - p1.x();
603             const double dy = p2.y() - p1.y();
604             const double len2 = dx*dx + dy*dy;
605             if (len2 < 1e-9) return QTransform();
606 
607             const double thetaRad = std::atan2(dy, dx);
608 
609             QTransform A; A.translate(-p1.x(), -p1.y());
610             QTransform B; B.rotateRadians(-thetaRad);
611             QTransform C(1, 0, 0, -1, 0, 0);
612             QTransform D; D.rotateRadians(thetaRad);
613             QTransform E; E.translate(p1.x(), p1.y());
614             return E * D * C * B * A;
615         }
616 
```

```
617         void MainWindow::translateByCells(int dx, int dy)
618         {
619             if (polygon.isEmpty()) return;
620             QTransform T;
621             T.translate(dx, dy);
622             applyTransform(T);
623             ui->lblInfo->setText(QString("Nudged by (%1, %2)").arg(dx).arg(dy));
624         }
625 
626         void MainWindow::nudgeUp()    { translateByCells(0, +1); }
627         void MainWindow::nudgeLeft()  { translateByCells(-1, 0); }
628         void MainWindow::nudgeDown()  { translateByCells(0, -1); }
629         void MainWindow::nudgeRight() { translateByCells(+1, 0); }
```

my_label.cpp

```
1           #include "my_label.h"
2 
3           my_label::my_label(QWidget *parent) : QLabel(parent)
4           {
5               setMouseTracking(true);
6           }
7 
8           void my_label::mouseMoveEvent(QMouseEvent *ev)
9           {
10              #if QT_VERSION >= QT_VERSION_CHECK(6,0,0)
11              QPointF pf = ev->position();
12              QPoint p = pf.toPoint();
13              #else
14              QPoint p = ev->pos();
15              #endif
16              if (p.x() >= 0 && p.y() >= 0 && p.x() < width() && p.y() < height()) {
17                  emit sendMousePosition(p);
18              }
19          }
20 
21          void my_label::mousePressEvent(QMouseEvent *ev)
22          {
23              if (ev->button() == Qt::LeftButton) {
24                  #if QT_VERSION >= QT_VERSION_CHECK(6,0,0)
25                  QPoint p = ev->position().toPoint();
26                  #else
27                  QPoint p = ev->pos();
28                  #endif
29                  lastX = p.x();
30                  lastY = p.y();
31                  emit Mouse_Pos();
32              }
33          }
```

Outputs

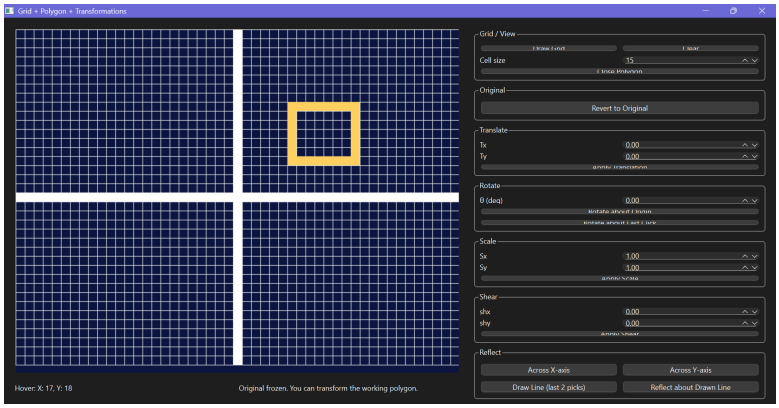


Figure 12: transformation_original_polygon

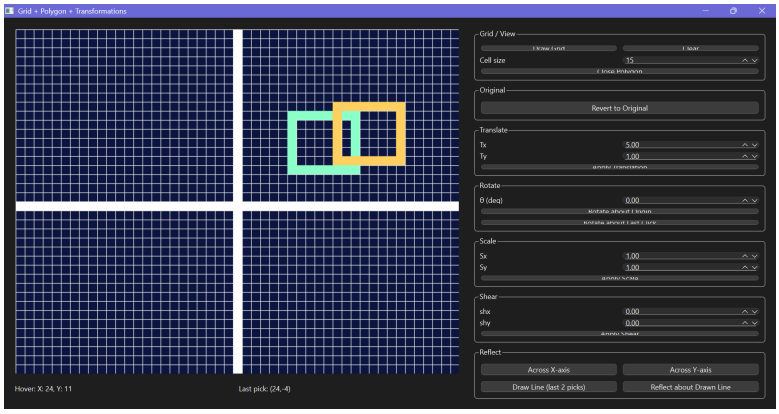


Figure 13: transformation_translation

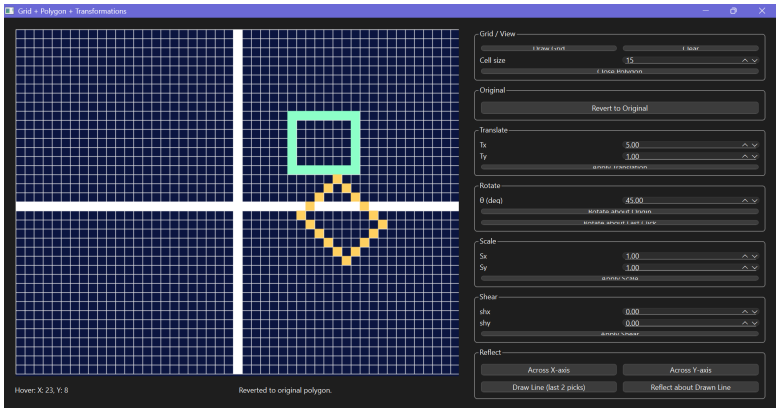


Figure 14: transformation_rotate_about_origin

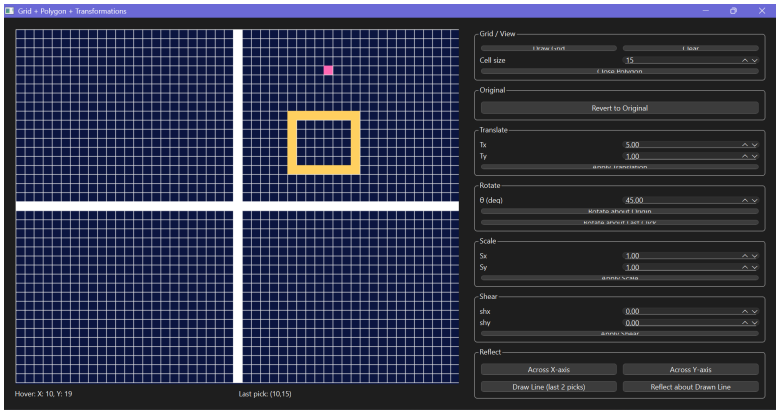


Figure 15: transformation_rotate_about_a_point_before_rotation

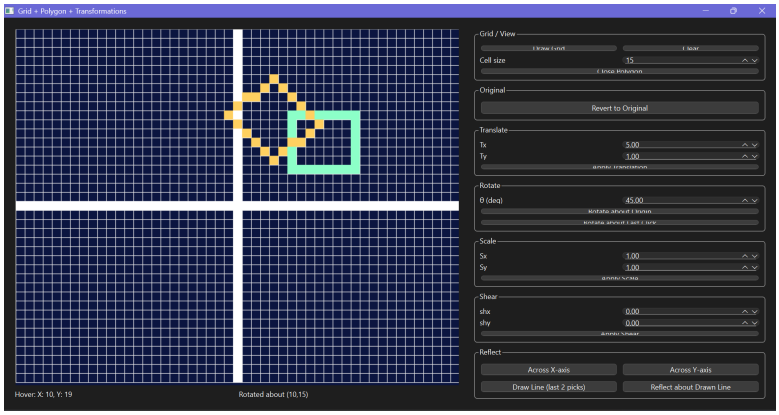


Figure 16: transformation_rotate_about_a_point_after_rotation

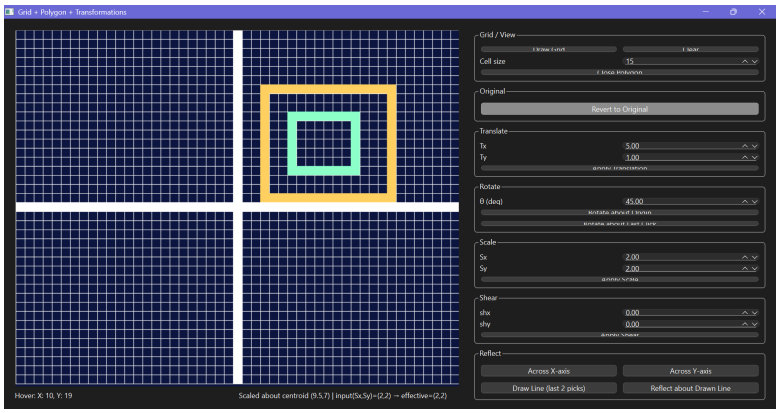


Figure 17: transformation_scaling_increased

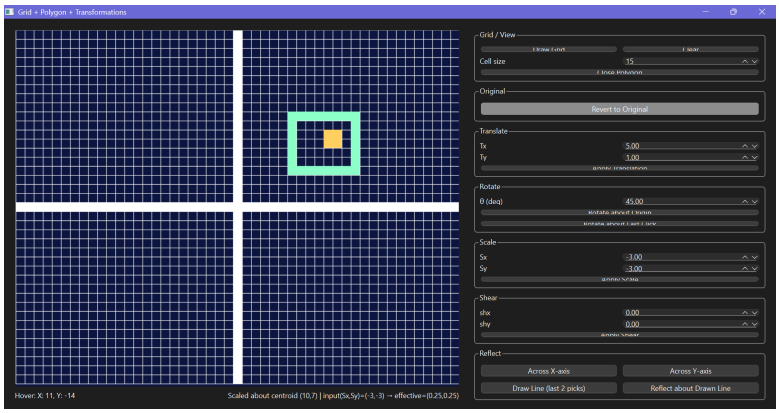


Figure 18: transformation_scaling_decreased

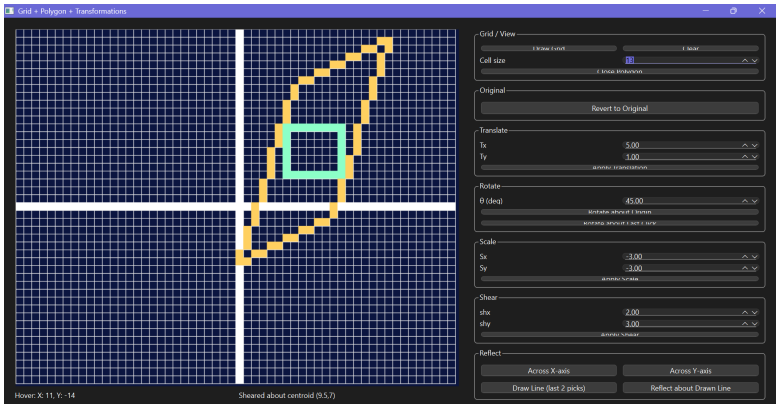


Figure 19: transformation_shearing

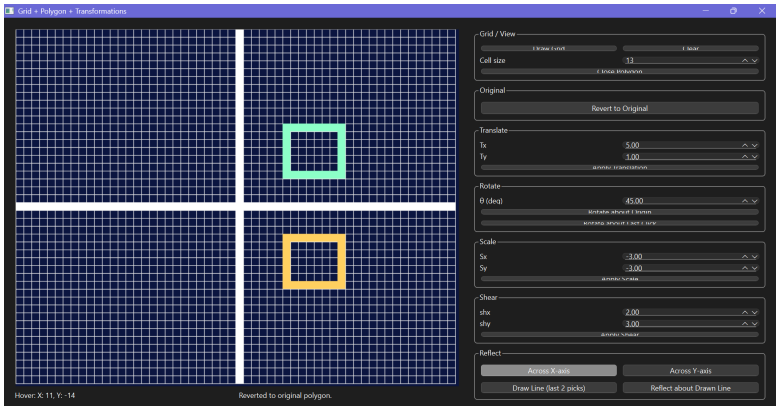


Figure 20: transformation_reflect_about_x_axis

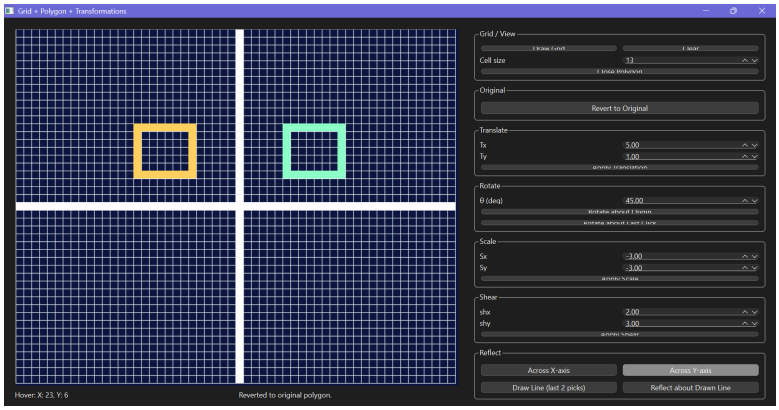


Figure 21: transformation_reflect_about_y_axis

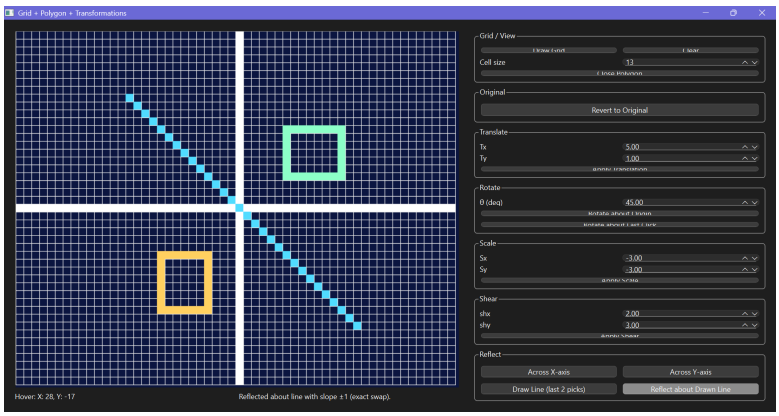


Figure 22: transformation_reflect_about_a_line

Output Explanation

For demonstrating the transformation functions, I used a simple rectangle as the base polygon. First I drew the original rectangle and then applied translation, rotation, scaling, shear, and reflections about the x- and y-axes, as well as rotation about an arbitrary picked point and reflection about a user–defined line. In each case, the transformed rectangle appeared exactly where predicted by the corresponding homogeneous transformation matrix (shifts for translation, correct angle and orientation for rotations, proportional changes for scaling, slanting for shear, and mirroring for reflections). Overall, the visual outputs closely matched the theoretical behaviour expected from the 2D transformation equations derived in this assignment.

Assignment 6: Line and Polygon Clipping

Problem Statement

1. Implement line clipping with respect to a rectangular clip window using:
   - Cohen–Sutherland algorithm
   - Liang–Barsky algorithm (optional)
2. Implement polygon clipping with respect to a rectangular clip window using:
   - Sutherland–Hodgeman algorithm
   - Weiler–Atherton algorithm (optional)

Code and Theoretical Explanation: Line Clipping

In this assignment I implemented interactive line clipping against an axis-aligned rectangular window using both Cohen–Sutherland and Liang–Barsky algorithms on a discrete grid.

Interactive grid and data structures The common infrastructure is provided by `GridScene` and `GridView`. The scene maintains a hash from integer grid coordinates to filled cells:

- `GridScene::paintCell(const QPoint& cell, const QBrush& brush)` draws a filled square of side `cellSize` at integer coordinates (x, y). This visually represents pixels on the conceptual Cartesian grid.
- `GridScene::drawBackground` draws the infinite grid and the coordinate axes, so that all clipping is done in integer grid space.

In the main window, the user selects two grid cells as endpoints of the line segment, which are stored in `linePoints`. The clipping window is defined by two opposite grid cells and stored as a `QRect clippingWindow`. The original line is rasterised using integer Bresenham:

```
void MainWindow::bresenhamLine(const QPoint& p1, const QPoint& p2,
                               const QBrush& brush)
{
    int x1 = p1.x(), y1 = p1.y(), x2 = p2.x(), y2 = p2.y();
    int dx = abs(x2 - x1), dy = abs(y2 - y1);
    int sx = (x1 < x2) ? 1 : -1;
    int sy = (y1 < y2) ? 1 : -1;
    int err = dx - dy;
    while (true) {
        scene->paintCell(QPoint(x1, y1), brush);
        if (x1 == x2 && y1 == y2) break;
        int e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x1 += sx; }
        if (e2 <  dx) { err += dx; y1 += sy; }
    }
}
```

After clipping, `drawPartialLine` recolours each pixel of the line as "inside" or "outside" the window, so that the clipped segment appears in green and the discarded part in gray:

```
void MainWindow::drawPartialLine(const QPoint& p1, const QPoint& p2,
                                 const QRect& window,
                                 const QBrush& insideBrush,
                                 const QBrush& outsideBrush,
                                 bool /*useCohenSutherland*/)
{
    int x1 = p1.x(), y1 = p1.y(), x2 = p2.x(), y2 = p2.y();
    int dx = abs(x2 - x1), dy = abs(y2 - y1);
    int sx = (x1 < x2) ? 1 : -1;
    int sy = (y1 < y2) ? 1 : -1;
    int err = dx - dy;
    while (true) {
        QPoint cell(x1, y1);
        if (isPointInsideWindow(cell, window))
            scene->paintCell(cell, insideBrush);
        else
            scene->paintCell(cell, outsideBrush);
        if (x1 == x2 && y1 == y2) break;
        int e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x1 += sx; }
        if (e2 <  dx) { err += dx; y1 += sy; }
    }
}
```

The actual clipping is performed in floating-point using the analytic algorithms below; the Bresenham loop is only for visualisation.

Cohen–Sutherland line clipping I consider an axis-aligned clipping window with boundaries

$$x_{\min},\ x_{\max},\ y_{\min},\ y_{\max}.$$

Each endpoint (x, y) is assigned a 4-bit region code based on its position relative to the window:

$$\text{code} = \begin{cases} \text{bit LEFT} = 1, & x < x_{\min}, \\ \text{bit RIGHT} = 1, & x > x_{\max}, \\ \text{bit BOTTOM} = 1, & y < y_{\min}, \\ \text{bit TOP} = 1, & y > y_{\max}. \end{cases}$$

Points inside the window have code 0000. This logic is implemented in

```
int MainWindow::computeOutCode(double x, double y, const QRect& rect)
{
    int code = INSIDE;
    if (x < rect.left())       code |= LEFT;
    else if (x > rect.right()) code |= RIGHT;
    if (y < rect.top())        code |= BOTTOM;
    else if (y > rect.bottom()) code |= TOP;
```

```
    return code;
}
```

Given a line segment from $P_1(x_1, y_1)$ to $P_2(x_2, y_2)$ with outcodes $\text{code}_1, \text{code}_2$, the algorithm iteratively applies three tests:

1. *Trivial acceptance:* if $\text{code}_1 \mid \text{code}_2 = 0$, both endpoints are inside and the line is fully visible.
2. *Trivial rejection:* if $\text{code}_1 \,\&\, \text{code}_2 \neq 0$, the line lies entirely in a common outside region and is fully invisible.
3. *Subdivision:* otherwise, exactly one endpoint is outside. I choose that endpoint and compute the intersection of the infinite line with the appropriate window boundary. I replace the outside endpoint with this intersection and recompute its outcode. The process repeats until case (1) or (2) holds.

In parametric form,

$$x(t) = x_1 + t(x_2 - x_1), \quad y(t) = y_1 + t(y_2 - y_1), \quad t \in [0, 1].$$

For each boundary, the intersection parameter is obtained by solving a single linear equation:

$$\begin{aligned}
\text{Top edge } (y = y_{\max}) &\Rightarrow t = \frac{y_{\max} - y_1}{y_2 - y_1},\ x = x_1 + t(x_2 - x_1), \\
\text{Bottom edge } (y = y_{\min}) &\Rightarrow t = \frac{y_{\min} - y_1}{y_2 - y_1},\ x = x_1 + t(x_2 - x_1), \\
\text{Right edge } (x = x_{\max}) &\Rightarrow t = \frac{x_{\max} - x_1}{x_2 - x_1},\ y = y_1 + t(y_2 - y_1), \\
\text{Left edge } (x = x_{\min}) &\Rightarrow t = \frac{x_{\min} - x_1}{x_2 - x_1},\ y = y_1 + t(y_2 - y_1).
\end{aligned}$$

The implementation follows this decision logic:

```
bool MainWindow::cohenSutherlandClip(double& x1, double& y1,
                                     double& x2, double& y2,
                                     const QRect& rect)
{
    int outcode1 = computeOutCode(x1, y1, rect);
    int outcode2 = computeOutCode(x2, y2, rect);
    bool accept = false;
    while (true) {
        if (!(outcode1 | outcode2)) { accept = true; break; }
        else if (outcode1 & outcode2) break;
        else {
            int outcodeOut = outcode1 ? outcode1 : outcode2;
            double x, y;
            if (outcodeOut & TOP) {
                x = x1 + (x2 - x1) *
                        (rect.bottom() - y1) / (y2 - y1);
                y = rect.bottom();
            }
            else if (outcodeOut & BOTTOM) {
                x = x1 + (x2 - x1) *
```

```
                        (rect.top() - y1) / (y2 - y1);
                y = rect.top();
            }
            else if (outcodeOut & RIGHT) {
                y = y1 + (y2 - y1) *
                        (rect.right() - x1) / (x2 - x1);
                x = rect.right();
            }
            else if (outcodeOut & LEFT) {
                y = y1 + (y2 - y1) *
                        (rect.left() - x1) / (x2 - x1);
                x = rect.left();
            }
            if (outcodeOut == outcode1) {
                x1 = x; y1 = y;
                outcode1 = computeOutCode(x1, y1, rect);
            } else {
                x2 = x; y2 = y;
                outcode2 = computeOutCode(x2, y2, rect);
            }
        }
    }
    return accept;
}
```

The GUI slot `onClipLineCohenSutherland()` checks that both the line and window exist, repaints the window and then calls `drawPartialLine` so the user can see which portion of the Bresenham raster falls inside the accepted segment.

Liang–Barsky line clipping Liang–Barsky uses the same parametric line representation, but instead of region codes it maintains a range of valid parameters $t \in [t_0, t_1]$ that satisfy all four window inequalities simultaneously.

The clipping window is described by

$$x_{\min} \le x(t) \le x_{\max}, \qquad y_{\min} \le y(t) \le y_{\max}.$$

Substituting $x(t) = x_1 + t\Delta x$ and $y(t) = y_1 + t\Delta y$, where $\Delta x = x_2 - x_1$, $\Delta y = y_2 - y_1$, we get four linear inequalities:

$$\begin{aligned}
x_{\min} \le x_1 + t\Delta x \quad &\Rightarrow \quad (-\Delta x)t \le x_1 - x_{\min}, \\
x_1 + t\Delta x \le x_{\max} \quad &\Rightarrow \quad (\ \Delta x)t \le x_{\max} - x_1, \\
y_{\min} \le y_1 + t\Delta y \quad &\Rightarrow \quad (-\Delta y)t \le y_1 - y_{\min}, \\
y_1 + t\Delta y \le y_{\max} \quad &\Rightarrow \quad (\ \Delta y)t \le y_{\max} - y_1.
\end{aligned}$$

In the compact form

$$p_i t \le q_i, \quad i = 0, \ldots, 3,$$

where the implementation uses

```
double dx = x2 - x1, dy = y2 - y1;
double p[4] = { -dx,  dx, -dy,  dy };
double q[4] = { x1 - xmin,  xmax - x1,
                y1 - ymin,  ymax - y1 };
```

For each constraint:

- If $p_i = 0$ and $q_i < 0$, the line is parallel to the corresponding boundary and entirely outside; the segment is rejected.
- If $p_i < 0$, the inequality tightens the lower bound:

$$t_{\text{enter}} = \max(t_{\text{enter}},\, q_i/p_i).$$

- If $p_i > 0$, the inequality tightens the upper bound:

$$t_{\text{leave}} = \min(t_{\text{leave}},\, q_i/p_i).$$

Initially $t_0 = 0$ and $t_1 = 1$. If at any time $t_0 > t_1$, the feasible interval is empty and the line is rejected. Otherwise, after all four constraints are processed, the clipped segment is simply

$$P'(t_0) = (x_1 + t_0\Delta x,\; y_1 + t_0\Delta y), \quad P'(t_1) = (x_1 + t_1\Delta x,\; y_1 + t_1\Delta y).$$

This logic is exactly reflected in the code:

```
bool MainWindow::liangBarskyClip(double& x1, double& y1,
                                 double& x2, double& y2,
                                 const QRect& rect)
{
    double xmin = rect.left(),  xmax = rect.right();
    double ymin = rect.top(),   ymax = rect.bottom();
    double dx = x2 - x1, dy = y2 - y1;
    double t0 = 0.0, t1 = 1.0;
    double p[4] = { -dx, dx, -dy, dy };
    double q[4] = { x1 - xmin, xmax - x1,
                    y1 - ymin, ymax - y1 };

    for (int i = 0; i < 4; i++) {
        if (p[i] == 0 && q[i] < 0) return false;
        double t = q[i] / p[i];
        if (p[i] < 0)      t0 = std::max(t0, t);
        else if (p[i] > 0) t1 = std::min(t1, t);
        if (t0 > t1) return false;
    }

    if (t0 > 0) { x1 += t0 * dx; y1 += t0 * dy; }
    if (t1 < 1) { x2 -= (1 - t1) * dx; y2 -= (1 - t1) * dy; }
    return true;
}
```

The slot `onClipLineLiangBarsky()` reuses the same GUI flow as Cohen–Sutherland, but the underlying slope-based clipping is now done using this parametric interval computation.

Code and Theoretical Explanation: Polygon Clipping

For polygon clipping I implemented both Sutherland–Hodgeman and Weiler–Atherton algorithms against the same axis-aligned rectangular window, again on the integer grid.

Polygon representation and drawing The interactive polygon is defined by a sequence of grid vertices `polygonVertices` (a `QList<QPoint>`). When the user finishes vertex selection and presses "Draw Polygon", the outline is rasterised by joining each pair of consecutive vertices (and finally the last to the first) using the same Bresenham routine:

```
void MainWindow::drawPolygonOutline(const QList<QPoint>& vertices,
                                    const QBrush& brush, bool collect)
{
    if (vertices.size() < 2) return;
    for (const QPoint& v : vertices) {
        scene->paintCell(v, brush);
        if (collect) polygonPixels.insert(v);
    }
    for (int i = 0; i < vertices.size(); i++) {
        int nextIndex = (i + 1) % vertices.size();
        bresenhamLine(vertices[i], vertices[nextIndex], brush, collect);
    }
}
```

The optional `collect` flag stores all edge pixels in a set `polygonPixels`, which lets me later recolour only those pixels that fall inside the clipping window. The window itself is again defined as a `QRect clippingWindow`.

Sutherland–Hodgeman polygon clipping Sutherland–Hodgeman clips a polygon by successively clipping it against each boundary of the window. The input polygon is described by a cyclic list of vertices $V = \{v_0, v_1, \ldots, v_{n-1}\}$. For each window edge, the algorithm walks along the polygon edges and constructs a new list according to inside/outside transitions.

For each edge of the clip window, I define a half-plane test:

$$\text{inside}(P) = \begin{cases} x \geq x_{\min}, & \text{for left boundary,} \\ x \leq x_{\max}, & \text{for right boundary,} \\ y \geq y_{\min}, & \text{for bottom boundary,} \\ y \leq y_{\max}, & \text{for top boundary.} \end{cases}$$

The code uses an enum `EdgePosition` and a single `isInside` routine:

```
bool MainWindow::isInside(const QPointF& point,
                          EdgePosition edge,
                          const QRect& clipRect)
{
```

```
    switch (edge) {
    case LEFT_EDGE:   return point.x() >= clipRect.left();
    case RIGHT_EDGE:  return point.x() <= clipRect.right();
    case BOTTOM_EDGE: return point.y() >= clipRect.top();
    case TOP_EDGE:    return point.y() <= clipRect.bottom();
    }
    return false;
}
```

Given an input vertex list V and a single clip edge, the *clip-against-edge* step considers each pair (S, P) of consecutive vertices (previous and current) and applies four cases:

- S inside, P inside: append P.
- S inside, P outside: append the intersection point I of segment SP with the boundary.
- S outside, P inside: append the intersection point I, then append P.
- S outside, P outside: append nothing.

This logic is implemented in `clipAgainstEdge`:

```
QList<QPointF> MainWindow::clipAgainstEdge(
        const QList<QPointF>& inputVertices,
        EdgePosition edge, const QRect& clipRect)
{
    QList<QPointF> outputVertices;
    if (inputVertices.isEmpty()) return outputVertices;

    QPointF previousVertex = inputVertices.last();
    for (const QPointF& currentVertex : inputVertices) {
        bool currentInside  = isInside(currentVertex, edge, clipRect);
        bool previousInside = isInside(previousVertex, edge, clipRect);
        if (currentInside) {
            if (!previousInside) {
                QPointF inter = getIntersection(previousVertex,
                                                currentVertex,
                                                edge, clipRect);
                outputVertices.append(inter);
            }
            outputVertices.append(currentVertex);
        }
        else if (previousInside) {
            QPointF inter = getIntersection(previousVertex,
                                            currentVertex,
                                            edge, clipRect);
            outputVertices.append(inter);
        }
        previousVertex = currentVertex;
    }
    return outputVertices;
}
```

The intersection point with a given edge is obtained by solving the parametric line equation, exactly as in the line clipping case. For example, for the left boundary $x = x_{\min}$,

$$x_1 + t(x_2 - x_1) = x_{\min} \;\Rightarrow\; t = \frac{x_{\min} - x_1}{x_2 - x_1}, \quad y = y_1 + t(y_2 - y_1).$$

The code handles all four boundaries:

```
QPointF MainWindow::getIntersection(const QPointF& p1,
                                    const QPointF& p2,
                                    EdgePosition edge,
                                    const QRect& clipRect)
{
    double x1 = p1.x(), y1 = p1.y();
    double x2 = p2.x(), y2 = p2.y();
    double x = 0, y = 0;

    switch (edge) {
    case LEFT_EDGE:
        x = clipRect.left();
        y = (x2 != x1) ? y1 + (y2 - y1) * (x - x1) / (x2 - x1) : y1;
        y = std::max<double>(clipRect.top(),
                             std::min<double>(clipRect.bottom(),
                                              std::round(y)));
        return QPointF(x, y);
    case RIGHT_EDGE:
        x = clipRect.right();
        y = (x2 != x1) ? y1 + (y2 - y1) * (x - x1) / (x2 - x1) : y1;
        y = std::max<double>(clipRect.top(),
                             std::min<double>(clipRect.bottom(),
                                              std::round(y)));
        return QPointF(x, y);
    case BOTTOM_EDGE:
        y = clipRect.top();
        x = (y2 != y1) ? x1 + (x2 - x1) * (y - y1) / (y2 - y1) : x1;
        x = std::max<double>(clipRect.left(),
                             std::min<double>(clipRect.right(),
                                              std::round(x)));
        return QPointF(x, y);
    case TOP_EDGE:
        y = clipRect.bottom();
        x = (y2 != y1) ? x1 + (x2 - x1) * (y - y1) / (y2 - y1) : x1;
        x = std::max<double>(clipRect.left(),
                             std::min<double>(clipRect.right(),
                                              std::round(x)));
        return QPointF(x, y);
    }
    return QPointF();
}
```

The full Sutherland–Hodgeman pipeline applies this step sequentially for the four window edges:

```
QList<QPointF> MainWindow::sutherlandHodgemanClip(
        const QList<QPointF>& polygon,
```

```
        const QRect& clipRect)
{
    QList<QPointF> output = polygon;
    output = clipAgainstEdge(output, LEFT_EDGE,   clipRect);
    if (output.isEmpty()) return output;
    output = clipAgainstEdge(output, RIGHT_EDGE,  clipRect);
    if (output.isEmpty()) return output;
    output = clipAgainstEdge(output, BOTTOM_EDGE, clipRect);
    if (output.isEmpty()) return output;
    output = clipAgainstEdge(output, TOP_EDGE,    clipRect);
    if (output.isEmpty()) return output;

    QList<QPointF> snapped;
    for (const QPointF& p : output) {
        QPointF q(qRound(p.x()), qRound(p.y()));
        if (snapped.isEmpty() || q != snapped.last())
            snapped.append(q);
    }
    if (snapped.size() >= 2 && snapped.first() == snapped.last())
        snapped.removeLast();
    return snapped;
}
```

The slot `onClipPolygonSutherHodge()` converts integer vertices to `QPointF`, calls `sutherlandHodgemanClip`, draws the original polygon in gray, the window in red with a semi-transparent fill, and finally rasterises the clipped polygon in green.

Weiler–Atherton polygon clipping Sutherland–Hodgeman works well when clipping a subject polygon by a convex window, but it is not ideal when the subject is concave or when multiple disjoint output regions or holes appear. To handle such general cases, I implemented Weiler–Atherton.

Weiler–Atherton operates on two closed polygonal loops:

- the *subject polygon* (the user-drawn polygon),
- the *clip polygon* (here, the rectangular window).

Both are represented as vertex lists that may be augmented with intersection vertices. I use the `VertexData` structure:

```
struct VertexData {
    QPointF pos;
    bool isIntersection = false;
    bool isStartNode    = false;
    int  link           = -1;
    bool visited        = false;
};
```

The algorithm proceeds in several stages:

1. **Initial lists.** I copy the subject polygon vertices into `subjectList` and the rectangular window vertices into `clipList`, both in cyclic order.

2. **Inside test.** I check whether all subject vertices lie inside the window (`isInsideClipWindow`). If so, the entire subject polygon is visible; if there are no intersections but the window lies inside the subject, the rectangular window itself becomes the clipped region.

3. **Intersection computation and insertion.** For every subject edge $[P_i, P_{i+1}]$ and every clip edge $[W_j, W_{j+1}]$, I compute their intersection (if any) using the analytic line intersection formula:

$$P(t) = P_1 + t(P_2 - P_1), \quad Q(u) = Q_1 + u(Q_2 - Q_1),$$

and solve for t, u such that $P(t) = Q(u)$. This is implemented in `getIntersectionWA`, which returns the intersection point only if it lies on both segments. For each valid intersection I, I record a `VertexData` node in both `subjectInserts` and `clipInserts`, marking whether it is a *start node* (transition from inside to outside):

```
QPointF I = getIntersectionWA(p1, p2, w1, w2);
if (!I.isNull()) {
    bool isStartNode = isInsideClipWindow(p1) &&
                       !isInsideClipWindow(p2);
    subjectInserts.append({ i + 1, { I, true,
                                     isStartNode, -1, false } });
    clipInserts.append   ({ j + 1, { I, true,
                                     isStartNode, -1, false } });
}
```

After computing all intersections, I insert these new vertices into `subjectList` and `clipList` at the correct positions along each edge. Sorting the insertion positions in descending order ensures that indices remain valid during insertion.

4. **Linking corresponding intersections.** Each geometric intersection appears once in the subject loop and once in the clip loop. I link the corresponding `VertexData` entries by setting their `link` fields so that traversals can jump between loops at intersection points:

```
for (int i = 0; i < subjectList.size(); ++i) {
    if (!subjectList[i].isIntersection) continue;
    for (int j = 0; j < clipList.size(); ++j) {
        if (clipList[j].isIntersection &&
            subjectList[i].pos == clipList[j].pos) {
            subjectList[i].link = j;
            clipList[j].link    = i;
            break;
        }
    }
}
```

5. **Tracing output polygons.** For each unvisited intersection in the subject list that is marked as a start node, I trace a complete output polygon by walking around the loops:

   - start from the chosen subject intersection,
   - follow vertices forward along the current loop (subject or clip), appending each `pos` to the output list and marking it visited,

- whenever an intersection vertex with a valid `link` is encountered, jump across to the other loop and continue,
- stop when I return to the starting vertex.

This is implemented by alternating between `subjectList` and `clipList` inside a do–while loop:

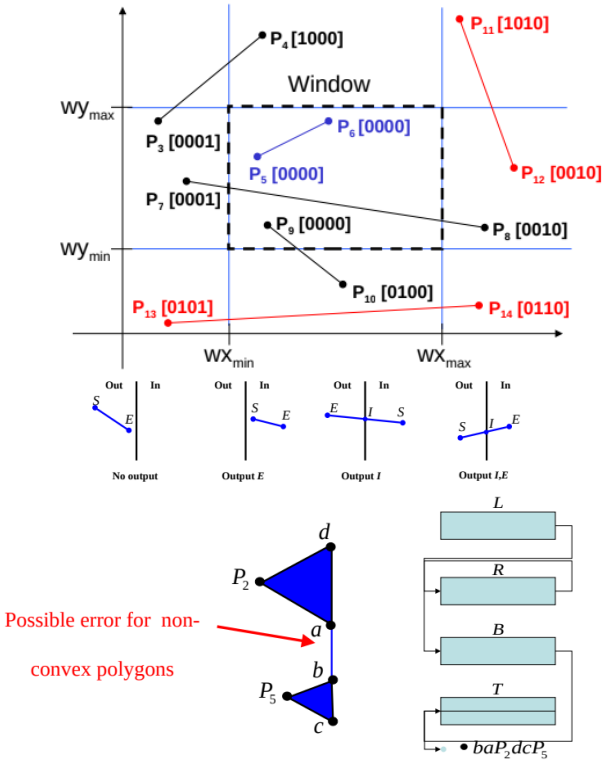
```
QList<QVector<QPointF>> resultPolygons;
for (int i = 0; i < subjectList.size(); ++i) {
    if (subjectList[i].isIntersection &&
        subjectList[i].isStartNode &&
        !subjectList[i].visited) {
        QVector<QPointF> cur;
        int idx = i;
        bool onSubject = true;
        do {
            VertexData* v = onSubject ?
                            &subjectList[idx] :
                            &clipList[idx];
            v->visited = true;
            cur.append(v->pos);
            if (v->isIntersection && v->link != -1) {
                onSubject = !onSubject;
                idx = v->link;
            }
            idx = (idx + 1) %
                  (onSubject ? subjectList.size()
                             : clipList.size());
        } while (cur.first() != cur.back() ||
                 cur.size() < 2);
        cur.pop_back();
        if (cur.size() >= 3)
            resultPolygons.append(cur);
    }
}
```

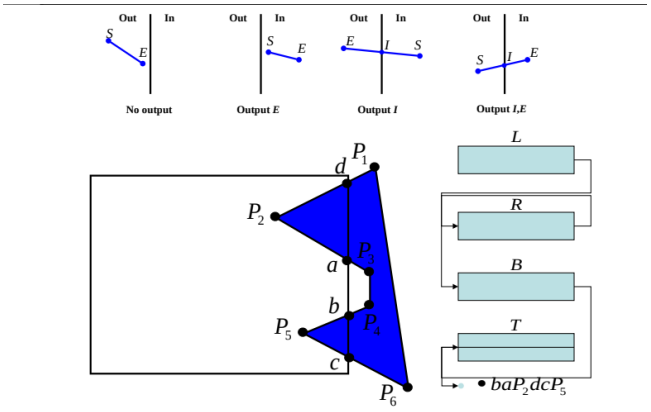
Each resulting list of points represents one clipped polygon component (there may be multiple for complex overlaps).

Finally, I rasterise all output polygons in green:

```
if (!resultPolygons.isEmpty()) {
    for (const auto& poly : resultPolygons) {
        QList<QPoint> edge;
        edge.reserve(poly.size());
        for (const QPointF& p : poly)
            edge.append(QPoint(qRound(p.x()),
                               qRound(p.y())));
        drawPolygonOutline(edge, QBrush(Qt::green));
    }
}
```

Before this, I draw the original polygon in light gray and the clip window in red with a translucent fill, so that the effect of Weiler–Atherton is visually clear.





Line Clipping Code

line_clipping.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

CONFIG += c++17

# You can make your code fail to compile if it uses deprecated APIs.
# In order to do so, uncomment the following line.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the
    APIs deprecated before Qt 6.0.0

SOURCES += \
gridscene.cpp \
gridview.cpp \
main.cpp \
mainwindow.cpp

HEADERS += \
gridscene.h \
gridview.h \
mainwindow.h

FORMS += \
mainwindow.ui

```

```
# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

gridscene.h

```
#ifndef GRIDSCENE_H
#define GRIDSCENE_H

#include <QGraphicsScene>
#include <QMap>
#include <QPoint>
#include <QBrush>
#include <QGraphicsRectItem>

class GridScene : public QGraphicsScene {
    Q_OBJECT

    public:
    explicit GridScene(QObject *parent = nullptr);

    void paintCell(const QPoint& cell, const QBrush& brush);
    void clearCells();
    void setCellSize(int size) { cellSize = size; }
    int getCellSize() const { return cellSize; }
    bool isCellFilled(const QPoint& cell) const;

    signals:
    void cellClicked(const QPoint& cell);
    void leftClick(const QPoint& cell);
    void rightClick(const QPoint& cell);

    protected:
    void mousePressEvent(QGraphicsSceneMouseEvent *event) override;
    void drawBackground(QPainter* painter, const QRectF& rect) override;

    private:
    int cellSize = 10;
    QHash<QPoint, QGraphicsRectItem*> cellItems;
};

#endif // GRIDSCENE_H
```

gridview.h

```
#ifndef GRIDVIEW_H
#define GRIDVIEW_H

#include <QGraphicsView>

class GridScene;

class GridView : public QGraphicsView {
```

```
    Q_OBJECT

    public:
    explicit GridView(QWidget *parent = nullptr);
    explicit GridView(GridScene *scene, QWidget *parent = nullptr);

    void zoomIn();
    void zoomOut();
    void resetZoom();
    double getZoomFactor() const;

    protected:
    void wheelEvent(QWheelEvent *event) override;

    private:
    void setZoom(double factor);

    double zoomFactor = 1.0;
    const double zoomIncrement = 0.1;
    const double minZoom = 0.2;
    const double maxZoom = 5.0;
};

#endif // GRIDVIEW_H
```

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QList>
#include <QPoint>
#include <QPointF>
#include <QRect>
#include "gridscene.h"

QT_BEGIN_NAMESPACE
namespace Ui {
    class MainWindow;
}
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

    public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

    private slots:
    void onDrawLine();
```

```
27              void onRestoreLine();
28              void onEraseLine();
29              void onDrawClippingWindow();
30              void onEraseClippingWindow();
31              void onClipLineCohenSutherland();
32              void onClipLineLiangBarsky();
33              void onClearAll();
34              void onCellClicked(const QPoint& cell);
35
36              private:
37              Ui::MainWindow *ui;
38              GridScene *scene;
39
40              QList<QPoint> linePoints;
41              QList<QPoint> originalLinePoints;
42              bool isDrawingLine;
43              int lineClickCount;
44
45              QRect clippingWindow;
46              bool hasClippingWindow;
47              bool isDrawingWindow;
48              int windowClickCount;
49              QPoint windowStart;
50
51              void bresenhamLine(const QPoint& p1, const QPoint& p2, const QBrush&
                    brush);
52              void drawRectangle(const QRect& rect, const QBrush& brush, int
                    thickness = 1);
53              void fillWindow(const QRect& rect, const QColor& color, int alpha);
54              void clearLine();
55              void clearWindow();
56              void drawLineWithClipping(const QPoint& p1, const QPoint& p2, const
                    QBrush& originalBrush, const QBrush& clippedBrush, bool
                    useCohenSutherland);
57
58              enum OutCode { INSIDE = 0, LEFT = 1, RIGHT = 2, BOTTOM = 4, TOP = 8 };
59
60              int computeOutCode(double x, double y, const QRect& rect);
61              bool cohenSutherlandClip(double& x1, double& y1, double& x2, double&
                    y2, const QRect& rect);
62              bool liangBarskyClip(double& x1, double& y1, double& x2, double& y2,
                    const QRect& rect);
63
64              void drawPartialLine(const QPoint& p1, const QPoint& p2, const QRect&
                    window, const QBrush& insideBrush, const QBrush& outsideBrush,
                    bool useCohenSutherland);
65              bool isPointInsideWindow(const QPoint& p, const QRect& window);
66          };
67
68          #endif // MAINWINDOW_H
```

gridscene.cpp

```
#include "gridscene.h"
#include <QGraphicsSceneMouseEvent>
#include <QPainter>
#include <cmath>
#include <QDebug>

GridScene::GridScene(QObject *parent)
: QGraphicsScene(parent) {
    setSceneRect(-5000, -5000, 10000, 10000);
}

void GridScene::paintCell(const QPoint& cell, const QBrush& brush) {
    if (!cellItems.contains(cell)) {
        auto *rect = addRect(cell.x() * cellSize, cell.y() * cellSize,
            cellSize, cellSize, Qt::NoPen, brush);
        cellItems[cell] = rect;
    } else {
        cellItems[cell]->setBrush(brush);
    }
}

void GridScene::clearCells() {
    for (auto *item : cellItems) removeItem(item);
    qDeleteAll(cellItems);
    cellItems.clear();
}

void GridScene::mousePressEvent(QGraphicsSceneMouseEvent *event) {
    QPoint cell(qFloor(event->scenePos().x() / cellSize),
    qFloor(event->scenePos().y() / cellSize));
    if (event->button() == Qt::LeftButton) {
        emit cellClicked(cell);
        emit leftClick(cell);
    }
}

bool GridScene::isCellFilled(const QPoint& cell) const
{
    if (!cellItems.contains(cell))
    return false;
    QColor color = cellItems[cell]->brush().color();
    return color.alpha() > 0;
}

void GridScene::drawBackground(QPainter* painter, const QRectF& rect) {
    painter->setRenderHint(QPainter::Antialiasing, false);

    int left = std::floor(rect.left() / cellSize);
    int right = std::ceil(rect.right() / cellSize);
    int top = std::floor(rect.top() / cellSize);
    int bottom = std::ceil(rect.bottom() / cellSize);

```

```
52              // grid lines
53              QPen gridPen(Qt::lightGray);
54              gridPen.setWidth(0);
55              gridPen.setCosmetic(true);
56              painter->setPen(gridPen);
57              for (int x = left; x <= right; ++x)
58              painter->drawLine(x * cellSize, rect.top(), x * cellSize, rect.bottom
                    ());
59              for (int y = top; y <= bottom; ++y)
60              painter->drawLine(rect.left(), y * cellSize, rect.right(), y *
                    cellSize);
61
62              // axes (filled cell-wise, like drawline app)
63              painter->setPen(Qt::NoPen);
64              painter->setBrush(Qt::black);
65
66              // X-axis row (y = 0)
67              for (int x = left; x <= right; ++x) {
68                  QRectF r(x * cellSize, 0, cellSize, cellSize);
69                  if (rect.intersects(r))
70                  painter->drawRect(r);
71              }
72
73              // Y-axis column (x = 0)
74              for (int y = top; y <= bottom; ++y) {
75                  QRectF r(0, y * cellSize, cellSize, cellSize);
76                  if (rect.intersects(r))
77                  painter->drawRect(r);
78              }
79
80              // painted cells
81              painter->setPen(Qt::NoPen);
82              for (auto it = cellItems.constBegin(); it != cellItems.constEnd(); ++
                    it) {
83                  QRectF r(it.key().x() * cellSize, it.key().y() * cellSize,
                        cellSize, cellSize);
84                  painter->setBrush(it.value()->brush());
85                  if (rect.intersects(r))
86                  painter->drawRect(r);
87              }
88          }
```

gridview.cpp

```
1           #include "gridview.h"
2           #include "gridscene.h"
3           #include <QPainter>
4           #include <QWheelEvent>
5           #include <QApplication>
6           #include <QDebug>
7
8           GridView::GridView(QWidget *parent)
9           : QGraphicsView(parent) {
```

```
10              setRenderHint(QPainter::Antialiasing, false);
11              setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
12              setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
13              setDragMode(QGraphicsView::NoDrag);
14              setMouseTracking(true);
15          }
16
17          GridView::GridView(GridScene *scene, QWidget *parent)
18          : QGraphicsView(scene, parent) {
19              setRenderHint(QPainter::Antialiasing, false);
20              setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
21              setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
22              setDragMode(QGraphicsView::NoDrag);
23              setMouseTracking(true);
24          }
25
26          void GridView::wheelEvent(QWheelEvent *event) {
27              if (QApplication::keyboardModifiers() & Qt::ControlModifier) {
28                  if (event->angleDelta().y() > 0) zoomIn();
29                  else zoomOut();
30                  event->accept();
31              } else {
32                  QGraphicsView::wheelEvent(event);
33              }
34          }
35
36          void GridView::zoomIn() {
37              if (zoomFactor < maxZoom) {
38                  double newZoom = qMin(zoomFactor + zoomIncrement, maxZoom);
39                  setZoom(newZoom);
40              }
41          }
42
43          void GridView::zoomOut() {
44              if (zoomFactor > minZoom) {
45                  double newZoom = qMax(zoomFactor - zoomIncrement, minZoom);
46                  setZoom(newZoom);
47              }
48          }
49
50          void GridView::resetZoom() {
51              setZoom(1.0);
52          }
53
54          double GridView::getZoomFactor() const {
55              return zoomFactor;
56          }
57
58          void GridView::setZoom(double factor) {
59              if (factor >= minZoom && factor <= maxZoom && factor != zoomFactor) {
60                  QPointF oldPos = mapToScene(viewport()->rect().center());
61                  resetTransform();
```

```
62                  scale(factor, factor);
63                  zoomFactor = factor;
64                  centerOn(oldPos);
65                  qDebug() << "Zoom level:" << zoomFactor * 100 << "%";
66              }
67          }
```

mainwindow.cpp

```
1           #include "mainwindow.h"
2           #include "ui_mainwindow.h"
3           #include "gridscene.h"
4           #include "gridview.h"
5           #include <QMessageBox>
6           #include <cmath>
7           #include <algorithm>
8
9           MainWindow::MainWindow(QWidget *parent)
10          : QMainWindow(parent)
11          , ui(new Ui::MainWindow)
12          , scene(new GridScene(this))
13          , isDrawingLine(false)
14          , lineClickCount(0)
15          , hasClippingWindow(false)
16          , isDrawingWindow(false)
17          , windowClickCount(0)
18          {
19              ui->setupUi(this);
20              ui->grid->setScene(scene);
21              ui->grid->setFixedSize(ui->grid->width(), ui->grid->height());
22              ui->grid->centerOn(0, 0);
23              scene->setCellSize(10);
24
25              connect(ui->drawLine, &QPushButton::clicked, this, &MainWindow::
                    onDrawLine);
26              connect(ui->restoreLine, &QPushButton::clicked, this, &MainWindow::
                    onRestoreLine);
27              connect(ui->eraseLine, &QPushButton::clicked, this, &MainWindow::
                    onEraseLine);
28              connect(ui->drawWindow, &QPushButton::clicked, this, &MainWindow::
                    onDrawClippingWindow);
29              connect(ui->eraseWindow, &QPushButton::clicked, this, &MainWindow::
                    onEraseClippingWindow);
30              connect(ui->clipLineCohenSuther, &QPushButton::clicked, this, &
                    MainWindow::onClipLineCohenSutherland);
31              connect(ui->clipLineLiangBarsky, &QPushButton::clicked, this, &
                    MainWindow::onClipLineLiangBarsky);
32              connect(ui->clearAll, &QPushButton::clicked, this, &MainWindow::
                    onClearAll);
33              connect(scene, &GridScene::leftClick, this, &MainWindow::onCellClicked
                    );
34          }
35
```

```
36          MainWindow::~MainWindow()
37          {
38              delete scene;
39              delete ui;
40          }
41
42          void MainWindow::onCellClicked(const QPoint& cell)
43          {
44              if (isDrawingLine) {
45                  linePoints.append(cell);
46                  lineClickCount++;
47                  scene->paintCell(cell, QBrush(Qt::blue));
48
49                  if (lineClickCount == 2) {
50                      bresenhamLine(linePoints[0], linePoints[1], QBrush(Qt::blue));
51                      originalLinePoints = linePoints;
52                      isDrawingLine = false;
53                      lineClickCount = 0;
54                      scene->update();
55                  }
56              }
57              else if (isDrawingWindow) {
58                  if (windowClickCount == 0) {
59                      windowStart = cell;
60                      scene->paintCell(cell, QBrush(Qt::red));
61                      windowClickCount++;
62                  }
63                  else if (windowClickCount == 1) {
64                      int xMin = std::min(windowStart.x(), cell.x());
65                      int yMin = std::min(windowStart.y(), cell.y());
66                      int xMax = std::max(windowStart.x(), cell.x());
67                      int yMax = std::max(windowStart.y(), cell.y());
68
69                      clippingWindow = QRect(QPoint(xMin, yMin), QPoint(xMax, yMax))
                            ;
70
71                      fillWindow(clippingWindow, QColor(173, 216, 230), 120);
72                      drawRectangle(clippingWindow, QBrush(Qt::red), 1);
73
74                      hasClippingWindow = true;
75                      isDrawingWindow = false;
76                      windowClickCount = 0;
77                      scene->update();
78                  }
79              }
80          }
81
82          void MainWindow::fillWindow(const QRect& rect, const QColor& color, int
                alpha)
83          {
84              QColor translucent = color;
85              translucent.setAlpha(alpha);
```

```
86              for (int x = rect.left(); x <= rect.right(); ++x)
87              for (int y = rect.top(); y <= rect.bottom(); ++y) {
88                  if (!scene->isCellFilled(QPoint(x, y)))  // skip if already
                        painted
89                  scene->paintCell(QPoint(x, y), QBrush(translucent));
90              }
91          }
92
93          void MainWindow::onDrawLine()
94          {
95              if (isDrawingWindow) {
96                  QMessageBox::warning(this, "Warning", "Please finish drawing the
                        clipping window first!");
97                  return;
98              }
99
100             clearLine();
101             linePoints.clear();
102             originalLinePoints.clear();
103             isDrawingLine = true;
104             lineClickCount = 0;
105             QMessageBox::information(this, "Draw Line", "Click two points to draw
                    a line.");
106         }
107
108         void MainWindow::onRestoreLine()
109         {
110             if (originalLinePoints.size() != 2) {
111                 QMessageBox::warning(this, "Warning", "No original line to restore
                        !");
112                 return;
113             }
114
115             clearLine();
116             linePoints = originalLinePoints;
117
118             if (hasClippingWindow) {
119                 fillWindow(clippingWindow, QColor(173, 216, 230), 120);
120                 drawRectangle(clippingWindow, QBrush(Qt::red), 1);
121             }
122
123             bresenhamLine(linePoints[0], linePoints[1], QBrush(Qt::blue));
124             scene->update();
125         }
126
127         void MainWindow::onEraseLine()
128         {
129             clearLine();
130             linePoints.clear();
131             originalLinePoints.clear();
132             scene->update();
133         }
```

```
134
135         void MainWindow::onDrawClippingWindow()
136         {
137             if (isDrawingLine) {
138                 QMessageBox::warning(this, "Warning", "Please finish drawing the
                        line first!");
139                 return;
140             }
141
142             clearWindow();
143             isDrawingWindow = true;
144             windowClickCount = 0;
145             hasClippingWindow = false;
146             QMessageBox::information(this, "Draw Clipping Window", "Click two
                    diagonal corners to define the clipping window.");
147         }
148
149         void MainWindow::onEraseClippingWindow()
150         {
151             clearWindow();
152             hasClippingWindow = false;
153             scene->update();
154         }
155
156         void MainWindow::onClipLineCohenSutherland()
157         {
158             if (linePoints.size() != 2) {
159                 QMessageBox::warning(this, "Warning", "Please draw a line first!")
                        ;
160                 return;
161             }
162
163             if (!hasClippingWindow) {
164                 QMessageBox::warning(this, "Warning", "Please draw a clipping
                        window first!");
165                 return;
166             }
167
168             clearLine();
169             fillWindow(clippingWindow, QColor(173, 216, 230), 120);
170             drawRectangle(clippingWindow, QBrush(Qt::red), 1);
171
172             drawPartialLine(linePoints[0], linePoints[1], clippingWindow, QBrush(
                    Qt::green), QBrush(Qt::gray), true);
173
174             scene->update();
175         }
176
177         void MainWindow::onClipLineLiangBarsky()
178         {
179             if (linePoints.size() != 2) {
180                 QMessageBox::warning(this, "Warning", "Please draw a line first!")
```

```
                    ;
181                 return;
182             }
183
184             if (!hasClippingWindow) {
185                 QMessageBox::warning(this, "Warning", "Please draw a clipping
                        window first!");
186                 return;
187             }
188
189             clearLine();
190             fillWindow(clippingWindow, QColor(173, 216, 230), 120);
191             drawRectangle(clippingWindow, QBrush(Qt::red), 1);
192
193             drawPartialLine(linePoints[0], linePoints[1], clippingWindow, QBrush(
                    Qt::green), QBrush(Qt::gray), false);
194
195             scene->update();
196         }
197
198         void MainWindow::onClearAll()
199         {
200             scene->clearCells();
201             linePoints.clear();
202             originalLinePoints.clear();
203             isDrawingLine = false;
204             lineClickCount = 0;
205             hasClippingWindow = false;
206             isDrawingWindow = false;
207             windowClickCount = 0;
208             scene->update();
209         }
210
211         void MainWindow::bresenhamLine(const QPoint& p1, const QPoint& p2, const
                QBrush& brush)
212         {
213             int x1 = p1.x(), y1 = p1.y(), x2 = p2.x(), y2 = p2.y();
214             int dx = abs(x2 - x1), dy = abs(y2 - y1);
215             int sx = (x1 < x2) ? 1 : -1;
216             int sy = (y1 < y2) ? 1 : -1;
217             int err = dx - dy;
218
219             while (true) {
220                 scene->paintCell(QPoint(x1, y1), brush);
221                 if (x1 == x2 && y1 == y2) break;
222                 int e2 = 2 * err;
223                 if (e2 > -dy) { err -= dy; x1 += sx; }
224                 if (e2 < dx) { err += dx; y1 += sy; }
225             }
226         }
227
228         void MainWindow::drawRectangle(const QRect& rect, const QBrush& brush, int
```

```
                thickness)
229         {
230             for (int i = 0; i < thickness; ++i) {
231                 bresenhamLine(QPoint(rect.left() + i, rect.top() + i), QPoint(rect
                        .right() - i, rect.top() + i), brush);
232                 bresenhamLine(QPoint(rect.left() + i, rect.bottom() - i), QPoint(
                        rect.right() - i, rect.bottom() - i), brush);
233                 bresenhamLine(QPoint(rect.left() + i, rect.top() + i), QPoint(rect
                        .left() + i, rect.bottom() - i), brush);
234                 bresenhamLine(QPoint(rect.right() - i, rect.top() + i), QPoint(
                        rect.right() - i, rect.bottom() - i), brush);
235             }
236         }
237
238         bool MainWindow::isPointInsideWindow(const QPoint& p, const QRect& window)
239         {
240             return p.x() >= window.left() && p.x() <= window.right() && p.y() >=
                    window.top() && p.y() <= window.bottom();
241         }
242
243         void MainWindow::drawPartialLine(const QPoint& p1, const QPoint& p2, const
                QRect& window, const QBrush& insideBrush, const QBrush& outsideBrush,
                bool useCohenSutherland)
244         {
245             int x1 = p1.x(), y1 = p1.y(), x2 = p2.x(), y2 = p2.y();
246             int dx = abs(x2 - x1), dy = abs(y2 - y1);
247             int sx = (x1 < x2) ? 1 : -1;
248             int sy = (y1 < y2) ? 1 : -1;
249             int err = dx - dy;
250
251             while (true) {
252                 QPoint cell(x1, y1);
253                 if (isPointInsideWindow(cell, window))
254                 scene->paintCell(cell, insideBrush);
255                 else
256                 scene->paintCell(cell, outsideBrush);
257
258                 if (x1 == x2 && y1 == y2) break;
259                 int e2 = 2 * err;
260                 if (e2 > -dy) { err -= dy; x1 += sx; }
261                 if (e2 < dx)  { err += dx; y1 += sy; }
262             }
263         }
264
265
266         void MainWindow::clearLine()
267         {
268             if (linePoints.size() != 2) return;
269
270             int x1 = linePoints[0].x(), y1 = linePoints[0].y();
271             int x2 = linePoints[1].x(), y2 = linePoints[1].y();
272             int dx = abs(x2 - x1), dy = abs(y2 - y1);
```

```
273             int sx = (x1 < x2) ? 1 : -1;
274             int sy = (y1 < y2) ? 1 : -1;
275             int err = dx - dy;
276 
277             while (true) {
278                 QPoint cell(x1, y1);
279                 bool isOnWindow = hasClippingWindow && isPointInsideWindow(cell,
                        clippingWindow);
280                 if (!isOnWindow)
281                 scene->paintCell(cell, QBrush(Qt::transparent));
282 
283                 if (x1 == x2 && y1 == y2) break;
284                 int e2 = 2 * err;
285                 if (e2 > -dy) { err -= dy; x1 += sx; }
286                 if (e2 < dx) { err += dx; y1 += sy; }
287             }
288         }
289 
290         void MainWindow::clearWindow()
291         {
292             if (hasClippingWindow) {
293                 for (int x = clippingWindow.left(); x <= clippingWindow.right();
                        ++x)
294                 for (int y = clippingWindow.top(); y <= clippingWindow.bottom();
                        ++y)
295                 scene->paintCell(QPoint(x, y), QBrush(Qt::transparent));
296             }
297         }
298 
299         int MainWindow::computeOutCode(double x, double y, const QRect& rect)
300         {
301             int code = INSIDE;
302             if (x < rect.left()) code |= LEFT;
303             else if (x > rect.right()) code |= RIGHT;
304             if (y < rect.top()) code |= BOTTOM;
305             else if (y > rect.bottom()) code |= TOP;
306             return code;
307         }
308 
309         bool MainWindow::cohenSutherlandClip(double& x1, double& y1, double& x2,
                double& y2, const QRect& rect)
310         {
311             int outcode1 = computeOutCode(x1, y1, rect);
312             int outcode2 = computeOutCode(x2, y2, rect);
313             bool accept = false;
314 
315             while (true) {
316                 if (!(outcode1 | outcode2)) { accept = true; break; }
317                 else if (outcode1 & outcode2) break;
318                 else {
319                     int outcodeOut = outcode1 ? outcode1 : outcode2;
320                     double x, y;
```

```
321
322                     if (outcodeOut & TOP) {
323                         x = x1 + (x2 - x1) * (rect.bottom() - y1) / (y2 - y1);
324                         y = rect.bottom();
325                     }
326                     else if (outcodeOut & BOTTOM) {
327                         x = x1 + (x2 - x1) * (rect.top() - y1) / (y2 - y1);
328                         y = rect.top();
329                     }
330                     else if (outcodeOut & RIGHT) {
331                         y = y1 + (y2 - y1) * (rect.right() - x1) / (x2 - x1);
332                         x = rect.right();
333                     }
334                     else if (outcodeOut & LEFT) {
335                         y = y1 + (y2 - y1) * (rect.left() - x1) / (x2 - x1);
336                         x = rect.left();
337                     }
338
339                     if (outcodeOut == outcode1) {
340                         x1 = x; y1 = y;
341                         outcode1 = computeOutCode(x1, y1, rect);
342                     }
343                     else {
344                         x2 = x; y2 = y;
345                         outcode2 = computeOutCode(x2, y2, rect);
346                     }
347                 }
348             }
349             return accept;
350         }
351
352         bool MainWindow::liangBarskyClip(double& x1, double& y1, double& x2,
                double& y2, const QRect& rect)
353         {
354             double xmin = rect.left(), xmax = rect.right(), ymin = rect.top(),
                    ymax = rect.bottom();
355             double dx = x2 - x1, dy = y2 - y1, t0 = 0.0, t1 = 1.0;
356             double p[4] = {-dx, dx, -dy, dy};
357             double q[4] = {x1 - xmin, xmax - x1, y1 - ymin, ymax - y1};
358
359             for (int i = 0; i < 4; i++) {
360                 if (p[i] == 0 && q[i] < 0) return false;
361                 double t = q[i] / p[i];
362                 if (p[i] < 0) t0 = std::max(t0, t);
363                 else if (p[i] > 0) t1 = std::min(t1, t);
364                 if (t0 > t1) return false;
365             }
366
367             if (t0 > 0) { x1 += t0 * dx; y1 += t0 * dy; }
368             if (t1 < 1) { x2 -= (1 - t1) * dx; y2 -= (1 - t1) * dy; }
369
370             return true;
```

```
371         }
```

mainwindow.cpp

```
1     #include "mainwindow.h"
2
3     #include <QApplication>
4
5     int main(int argc, char *argv[])
6     {
7         QApplication a(argc, argv);
8         MainWindow w;
9         w.show();
10        return a.exec();
11    }
```

polygon_clipping.pro

```
1         QT       += core gui
2
3         greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
4
5         CONFIG += c++17
6
7         # You can make your code fail to compile if it uses deprecated APIs.
8         # In order to do so, uncomment the following line.
9         #DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the
              APIs deprecated before Qt 6.0.0
10
11        SOURCES += \
12        gridscene.cpp \
13        gridview.cpp \
14        main.cpp \
15        mainwindow.cpp
16
17        HEADERS += \
18        gridscene.h \
19        gridview.h \
20        mainwindow.h
21
22        FORMS += \
23        mainwindow.ui
24
25        # Default rules for deployment.
26        qnx: target.path = /tmp/$${TARGET}/bin
27        else: unix:!android: target.path = /opt/$${TARGET}/bin
28        !isEmpty(target.path): INSTALLS += target
```

gridscene.h

```
1         #ifndef GRIDSCENE_H
2         #define GRIDSCENE_H
```

```
3
4           #include <QGraphicsScene>
5           #include <QMap>
6           #include <QPoint>
7           #include <QBrush>
8           #include <QGraphicsRectItem>
9
10          class GridScene : public QGraphicsScene {
11              Q_OBJECT
12
13              public:
14              explicit GridScene(QObject *parent = nullptr);
15
16              void paintCell(const QPoint& cell, const QBrush& brush);
17              void clearCells();
18              void setCellSize(int size) { cellSize = size; }
19              int getCellSize() const { return cellSize; }
20
21              signals:
22              void cellClicked(const QPoint& cell);
23              void leftClick(const QPoint& cell);
24              void rightClick(const QPoint& cell);
25
26              protected:
27              void mousePressEvent(QGraphicsSceneMouseEvent *event) override;
28              void drawBackground(QPainter* painter, const QRectF& rect) override;
29
30              private:
31              int cellSize = 10;
32              QHash<QPoint, QGraphicsRectItem*> cellItems;
33          };
34
35          #endif // GRIDSCENE_H
```

gridview.h

```
1           #ifndef GRIDVIEW_H
2           #define GRIDVIEW_H
3
4           #include <QGraphicsView>
5
6           class GridScene;
7
8           class GridView : public QGraphicsView {
9               Q_OBJECT
10
11              public:
12              explicit GridView(QWidget *parent = nullptr);
13              explicit GridView(GridScene *scene, QWidget *parent = nullptr);
14
15              void zoomIn();
16              void zoomOut();
17              void resetZoom();
```

```
18              double getZoomFactor() const;
19
20              protected:
21              void wheelEvent(QWheelEvent *event) override;
22
23              private:
24              void setZoom(double factor);
25
26              double zoomFactor = 1.0;
27              const double zoomIncrement = 0.1;
28              const double minZoom = 0.2;
29              const double maxZoom = 5.0;
30          };
31
32          #endif // GRIDVIEW_H
```

mainwindow.h

```
1           #ifndef MAINWINDOW_H
2           #define MAINWINDOW_H
3
4           #include <QMainWindow>
5           #include <QList>
6           #include <QPoint>
7           #include <QPointF>
8           #include <QRect>
9           #include <QSet>
10          #include <QMap>
11          #include <cmath>
12          #include "gridscene.h"
13
14          QT_BEGIN_NAMESPACE
15          namespace Ui {
16              class MainWindow;
17          }
18          QT_END_NAMESPACE
19
20          struct VertexData {
21              QPointF pos;
22              bool isIntersection = false;
23              bool isStartNode = false;
24              int link = -1;
25              bool visited = false;
26          };
27
28          class MainWindow : public QMainWindow
29          {
30              Q_OBJECT
31
32              public:
33              MainWindow(QWidget *parent = nullptr);
34              ~MainWindow();
35
```

```
36              private slots:
37              void onDrawPolygon();
38              void onRestorePolygon();
39              void onErasePolygon();
40              void onDrawClippingWindow();
41              void onEraseClippingWindow();
42              void onClipPolygonSutherHodge();
43              void onClipPolygonWeilerAther();
44              void onClearAll();
45              void onCellClicked(const QPoint& cell);
46
47              private:
48              Ui::MainWindow *ui;
49              GridScene *scene;
50
51              QList<QPoint> polygonVertices;
52              QList<QPoint> originalPolygonVertices;
53              bool hasPolygon;
54              bool isSelectingVertices;
55
56              QRect clippingWindow;
57              bool hasClippingWindow;
58              bool isDrawingWindow;
59              int windowClickCount;
60              QPoint windowStart;
61
62              void bresenhamLine(const QPoint& p1, const QPoint& p2, const QBrush&
                    brush, bool collect = false);
63              void drawPolygonOutline(const QList<QPoint>& vertices, const QBrush&
                    brush, bool collect = false);
64              void drawRectangle(const QRect& rect, const QBrush& brush);
65              void drawRectangle(const QRect& rect, const QBrush& brush, int
                    thickness);
66              void clearPolygon();
67              void clearWindow();
68              void fillWindow(const QRect& rect, const QColor& color, int alpha);
69
70              enum EdgePosition {
71                  LEFT_EDGE = 0,
72                  RIGHT_EDGE = 1,
73                  BOTTOM_EDGE = 2,
74                  TOP_EDGE = 3
75              };
76
77              QList<QPointF> clipAgainstEdge(const QList<QPointF>& inputVertices,
                    EdgePosition edge, const QRect& clipRect);
78              bool isInside(const QPointF& point, EdgePosition edge, const QRect&
                    clipRect);
79              QPointF getIntersection(const QPointF& p1, const QPointF& p2,
                    EdgePosition edge, const QRect& clipRect);
80              QList<QPointF> sutherlandHodgemanClip(const QList<QPointF>& polygon,
                    const QRect& clipRect);
```

```
81              QPointF computeLineIntersection(const QPointF& p1, const QPointF& p2,
                    const QPointF& p3, const QPointF& p4);
82              double distance(const QPointF& p1, const QPointF& p2);
83              bool isPointInsidePolygon(const QPointF& point, const QList<QPointF>&
                    polygon);
84              void drawPolygonInsideRect(const QList<QPoint>& vertices, const QRect&
                     rect, const QBrush& brush);
85
86
87              QPointF getIntersectionWA(const QPointF& p1, const QPointF& p2, const
                    QPointF& w1, const QPointF& w2);
88              void weilerAthertonPolygonClip();
89              bool isInsideClipWindow(const QPointF& p) const;
90              bool pointInPolygon(const QVector<QPointF>& poly, const QPointF& test)
                     const;
91              QVector<QPoint> preClipPolygon;
92              QVector<QPair<QPoint, QPoint>> preClipLines;
93              bool hasPreClip = false;
94          };
95
96          #endif // MAINWINDOW_H
```

gridscene.cpp

```
1           #include "gridscene.h"
2           #include <QGraphicsSceneMouseEvent>
3           #include <QPainter>
4           #include <cmath>
5           #include <QDebug>
6
7           GridScene::GridScene(QObject *parent)
8           : QGraphicsScene(parent) {
9               setSceneRect(-5000, -5000, 10000, 10000);
10          }
11
12          void GridScene::paintCell(const QPoint& cell, const QBrush& brush) {
13              if (!cellItems.contains(cell)) {
14                  auto *rect = addRect(cell.x() * cellSize, cell.y() * cellSize,
                        cellSize, cellSize, Qt::NoPen, brush);
15                  cellItems[cell] = rect;
16              } else {
17                  cellItems[cell]->setBrush(brush);
18              }
19          }
20
21          void GridScene::clearCells() {
22              for (auto *item : cellItems) removeItem(item);
23              qDeleteAll(cellItems);
24              cellItems.clear();
25          }
26
27          void GridScene::mousePressEvent(QGraphicsSceneMouseEvent *event) {
28              QPoint cell(qFloor(event->scenePos().x() / cellSize),
```

```
29              qFloor(event->scenePos().y() / cellSize));
30              if (event->button() == Qt::LeftButton) {
31                  emit cellClicked(cell);
32                  emit leftClick(cell);
33              }
34          }
35
36          void GridScene::drawBackground(QPainter* painter, const QRectF& rect) {
37              painter->setRenderHint(QPainter::Antialiasing, false);
38
39              int left = std::floor(rect.left() / cellSize);
40              int right = std::ceil(rect.right() / cellSize);
41              int top = std::floor(rect.top() / cellSize);
42              int bottom = std::ceil(rect.bottom() / cellSize);
43
44              // grid lines
45              QPen gridPen(Qt::lightGray);
46              gridPen.setWidth(0);
47              gridPen.setCosmetic(true);
48              painter->setPen(gridPen);
49              for (int x = left; x <= right; ++x)
50              painter->drawLine(x * cellSize, rect.top(), x * cellSize, rect.bottom
                    ());
51              for (int y = top; y <= bottom; ++y)
52              painter->drawLine(rect.left(), y * cellSize, rect.right(), y *
                    cellSize);
53
54              // axes (filled cell-wise, like drawline app)
55              painter->setPen(Qt::NoPen);
56              painter->setBrush(Qt::black);
57
58              // X-axis row (y = 0)
59              for (int x = left; x <= right; ++x) {
60                  QRectF r(x * cellSize, 0, cellSize, cellSize);
61                  if (rect.intersects(r))
62                  painter->drawRect(r);
63              }
64
65              // Y-axis column (x = 0)
66              for (int y = top; y <= bottom; ++y) {
67                  QRectF r(0, y * cellSize, cellSize, cellSize);
68                  if (rect.intersects(r))
69                  painter->drawRect(r);
70              }
71
72              // painted cells
73              painter->setPen(Qt::NoPen);
74              for (auto it = cellItems.constBegin(); it != cellItems.constEnd(); ++
                    it) {
75                  QRectF r(it.key().x() * cellSize, it.key().y() * cellSize,
                        cellSize, cellSize);
76                  painter->setBrush(it.value()->brush());
```

```
77                  if (rect.intersects(r))
78                  painter->drawRect(r);
79              }
80          }
```

gridview.cpp

```
1           #include "gridview.h"
2           #include "gridscene.h"
3           #include <QPainter>
4           #include <QWheelEvent>
5           #include <QApplication>
6           #include <QDebug>
7
8           GridView::GridView(QWidget *parent)
9           : QGraphicsView(parent) {
10              setRenderHint(QPainter::Antialiasing, false);
11              setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
12              setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
13              setDragMode(QGraphicsView::NoDrag);
14              setMouseTracking(true);
15          }
16
17          GridView::GridView(GridScene *scene, QWidget *parent)
18          : QGraphicsView(scene, parent) {
19              setRenderHint(QPainter::Antialiasing, false);
20              setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
21              setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
22              setDragMode(QGraphicsView::NoDrag);
23              setMouseTracking(true);
24          }
25
26          void GridView::wheelEvent(QWheelEvent *event) {
27              if (QApplication::keyboardModifiers() & Qt::ControlModifier) {
28                  if (event->angleDelta().y() > 0) zoomIn();
29                  else zoomOut();
30                  event->accept();
31              } else {
32                  QGraphicsView::wheelEvent(event);
33              }
34          }
35
36          void GridView::zoomIn() {
37              if (zoomFactor < maxZoom) {
38                  double newZoom = qMin(zoomFactor + zoomIncrement, maxZoom);
39                  setZoom(newZoom);
40              }
41          }
42
43          void GridView::zoomOut() {
44              if (zoomFactor > minZoom) {
45                  double newZoom = qMax(zoomFactor - zoomIncrement, minZoom);
46                  setZoom(newZoom);
```

```
            }
        }

        void GridView::resetZoom() {
            setZoom(1.0);
        }

        double GridView::getZoomFactor() const {
            return zoomFactor;
        }

        void GridView::setZoom(double factor) {
            if (factor >= minZoom && factor <= maxZoom && factor != zoomFactor) {
                QPointF oldPos = mapToScene(viewport()->rect().center());
                resetTransform();
                scale(factor, factor);
                zoomFactor = factor;
                centerOn(oldPos);
                qDebug() << "Zoom level:" << zoomFactor * 100 << "%";
            }
        }
```

mainwindow.cpp

```
        #include "mainwindow.h"
#include "ui_mainwindow.h"
#include "gridscene.h"
#include "gridview.h"
#include <QMessageBox>
#include <cmath>
#include <algorithm>

QSet<QPoint> polygonPixels;
static inline bool feq(double a, double b, double eps=1e-6){
    return std::abs(a-b)<=eps; }

static inline bool peq(const QPointF& a, const QPointF& b){
    return feq(a.x(),b.x()) && feq(a.y(),b.y());
}


MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
    , scene(new GridScene(this))
    , hasPolygon(false)
    , isSelectingVertices(false)
    , hasClippingWindow(false)
    , isDrawingWindow(false)
    , windowClickCount(0)
{
    ui->setupUi(this);

```

```
30      ui->grid->setScene(scene);
31      ui->grid->setFixedSize(ui->grid->width(), ui->grid->height());
32      ui->grid->centerOn(0, 0);
33      scene->setCellSize(10);
34
35      connect(ui->drawPolygon, &QPushButton::clicked, this, &MainWindow::
            onDrawPolygon);
36      connect(ui->restorePolygon, &QPushButton::clicked, this, &MainWindow::
            onRestorePolygon);
37      connect(ui->erasePolygon, &QPushButton::clicked, this, &MainWindow::
            onErasePolygon);
38      connect(ui->drawWindow, &QPushButton::clicked, this, &MainWindow::
            onDrawClippingWindow);
39      connect(ui->eraseWindow, &QPushButton::clicked, this, &MainWindow::
            onEraseClippingWindow);
40      connect(ui->clipPolygonSutherHodge, &QPushButton::clicked, this, &MainWindow::
            onClipPolygonSutherHodge);
41      connect(ui->clipPolygonWeilerAther, &QPushButton::clicked, this, &MainWindow::
            onClipPolygonWeilerAther);
42      connect(ui->clearAll, &QPushButton::clicked, this, &MainWindow::onClearAll);
43      connect(scene, &GridScene::leftClick, this, &MainWindow::onCellClicked);
44
45      polygonVertices.clear();
46      originalPolygonVertices.clear();
47  }
48
49  MainWindow::~MainWindow()
50  {
51      delete scene;
52      delete ui;
53  }
54
55  void MainWindow::onCellClicked(const QPoint& cell)
56  {
57      if (isSelectingVertices) {
58          polygonVertices.append(cell);
59          scene->paintCell(cell, QBrush(Qt::blue));
60          scene->update();
61      }
62      else if (isDrawingWindow) {
63          if (windowClickCount == 0) {
64              windowStart = cell;
65              scene->paintCell(cell, QBrush(Qt::red));
66              windowClickCount++;
67          }
68          else if (windowClickCount == 1) {
69              int xMin = std::min(windowStart.x(), cell.x());
70              int yMin = std::min(windowStart.y(), cell.y());
71              int xMax = std::max(windowStart.x(), cell.x());
72              int yMax = std::max(windowStart.y(), cell.y());
73              clippingWindow = QRect(QPoint(xMin, yMin), QPoint(xMax, yMax));
74              fillWindow(clippingWindow, QColor(173, 216, 230), 120);
```

```
75              drawRectangle(clippingWindow, QBrush(Qt::red), 1);
76              if (hasPolygon) drawPolygonOutline(polygonVertices, QBrush(Qt::blue));
77              hasClippingWindow = true;
78              isDrawingWindow = false;
79              windowClickCount = 0;
80              scene->update();
81          }
82      }
83  }
84
85
86  void MainWindow::onDrawPolygon()
87  {
88      if (isDrawingWindow) {
89          QMessageBox::warning(this, "Warning", "Please finish drawing the clipping
                window first!");
90          return;
91      }
92
93      if (isSelectingVertices) {
94          if (polygonVertices.size() < 3) {
95              QMessageBox::warning(this, "Warning", "Please select at least 3
                    vertices for a polygon!");
96              return;
97          }
98
99          originalPolygonVertices = polygonVertices;
100         drawPolygonOutline(polygonVertices, QBrush(Qt::blue), true);
101         hasPolygon = true;
102         isSelectingVertices = false;
103         scene->update();
104     }
105     else {
106         clearPolygon();
107         polygonVertices.clear();
108         originalPolygonVertices.clear();
109         isSelectingVertices = true;
110         hasPolygon = false;
111         QMessageBox::information(this, "Draw Polygon", "Click on the grid to
                select vertices. Click 'Draw Polygon' again to finish.");
112     }
113 }
114
115 void MainWindow::onRestorePolygon()
116 {
117     if (originalPolygonVertices.size() < 3) {
118         QMessageBox::warning(this, "Warning", "No original polygon to restore!");
119         return;
120     }
121
122     clearPolygon();
123     polygonVertices = originalPolygonVertices;
```

```
124     drawPolygonOutline(polygonVertices, QBrush(Qt::blue), true);
125     hasPolygon = true;
126     scene->update();
127 }
128 
129 
130 void MainWindow::onErasePolygon()
131 {
132     clearPolygon();
133     polygonVertices.clear();
134     originalPolygonVertices.clear();
135     hasPolygon = false;
136     isSelectingVertices = false;
137     scene->update();
138 }
139 
140 
141 void MainWindow::onDrawClippingWindow()
142 {
143     if (isSelectingVertices) {
144         QMessageBox::warning(this, "Warning", "Please finish drawing the polygon
                first!");
145         return;
146     }
147 
148     clearWindow();
149     isDrawingWindow = true;
150     windowClickCount = 0;
151     hasClippingWindow = false;
152     QMessageBox::information(this, "Draw Clipping Window", "Click two diagonal
            corners to define the clipping window.");
153 }
154 
155 
156 void MainWindow::onEraseClippingWindow()
157 {
158     clearWindow();
159     hasClippingWindow = false;
160     scene->update();
161 }
162 
163 
164 void MainWindow::drawPolygonInsideRect(const QList<QPoint>& /*unused*/, const
        QRect& rect, const QBrush& brush)
165 {
166     for (const QPoint& p : polygonPixels) {
167         if (rect.contains(p, true)) scene->paintCell(p, brush);
168     }
169 }
170 
171 
172 void MainWindow::onClipPolygonSutherHodge()
```

```
{
    if (!hasPolygon) { QMessageBox::warning(this, "Warning", "Please draw a polygon first!"); return; }
    if (!hasClippingWindow) { QMessageBox::warning(this, "Warning", "Please draw a clipping window first!"); return; }
    if (polygonVertices.size() < 3) { QMessageBox::warning(this, "Warning", "Invalid polygon!"); return; }

    QList<QPointF> inputPolygon;
    for (const QPoint& p : polygonVertices) inputPolygon.append(QPointF(p));
    QList<QPointF> clippedPolygon = sutherlandHodgemanClip(inputPolygon, clippingWindow);

    QList<QPoint> original = polygonVertices;
    clearPolygon();
    drawPolygonOutline(original, QBrush(Qt::gray), true);
    fillWindow(clippingWindow, QColor(173, 216, 230), 120);
    drawRectangle(clippingWindow, QBrush(Qt::red), 1);

    if (clippedPolygon.size() >= 2) {
        QList<QPoint> clippedInt;
        clippedInt.reserve(clippedPolygon.size());
        for (const QPointF& p : clippedPolygon)
            clippedInt.append(QPoint(qRound(p.x()), qRound(p.y())));

        drawPolygonOutline(clippedInt, QBrush(Qt::green));

        polygonVertices = clippedInt;
        hasPolygon = true;
    } else {
        polygonVertices.clear();
        hasPolygon = false;
        QMessageBox::information(this, "Clipping Result", "Polygon is completely outside the clipping window or too small after clipping!");
    }

    scene->update();
}


void MainWindow::onClipPolygonWeilerAther()
{
    if (!hasPolygon) { QMessageBox::warning(this, "Warning", "Please draw a polygon first!"); return; }
    if (!hasClippingWindow) { QMessageBox::warning(this, "Warning", "Please draw a clipping window first!"); return; }
    if (polygonVertices.size() < 3) { QMessageBox::warning(this, "Warning", "Invalid polygon!"); return; }

    weilerAthertonPolygonClip();
}

```

```
217
218 QPointF MainWindow::getIntersectionWA(const QPointF& p1, const QPointF& p2, const
        QPointF& q1, const QPointF& q2)
219 {
220     double A1 = p2.y() - p1.y();
221     double B1 = p1.x() - p2.x();
222     double C1 = A1 * p1.x() + B1 * p1.y();
223
224     double A2 = q2.y() - q1.y();
225     double B2 = q1.x() - q2.x();
226     double C2 = A2 * q1.x() + B2 * q1.y();
227
228     double det = A1 * B2 - A2 * B1;
229     if (std::abs(det) < 1e-9) return QPointF();
230
231     double x = (B2 * C1 - B1 * C2) / det;
232     double y = (A1 * C2 - A2 * C1) / det;
233
234     const double eps = 1e-6;
235     if (x >= std::min(p1.x(), p2.x()) - eps && x <= std::max(p1.x(), p2.x()) + eps
            &&
236         y >= std::min(p1.y(), p2.y()) - eps && y <= std::max(p1.y(), p2.y()) + eps
                &&
237         x >= std::min(q1.x(), q2.x()) - eps && x <= std::max(q1.x(), q2.x()) + eps
                &&
238         y >= std::min(q1.y(), q2.y()) - eps && y <= std::max(q1.y(), q2.y()) + eps
               )
239     {
240         return QPointF(x, y);
241     }
242     return QPointF();
243 }
244
245
246 bool MainWindow::isInsideClipWindow(const QPointF& p) const
247 {
248     return p.x() >= clippingWindow.left()  && p.x() <= clippingWindow.right() && p
            .y() >= clippingWindow.top()   && p.y() <= clippingWindow.bottom();
249 }
250
251 bool MainWindow::pointInPolygon(const QVector<QPointF>& poly, const QPointF& test)
        const
252 {
253     bool inside = false;
254     int n = poly.size();
255     for (int i = 0, j = n - 1; i < n; j = i++) {
256         const QPointF& pi = poly[i];
257         const QPointF& pj = poly[j];
258         bool intersect = ((pi.y() > test.y()) != (pj.y() > test.y())) &&
259                          (test.x() < (pj.x() - pi.x()) * (test.y() - pi.y()) /
260                                                  (pj.y() - pi.y() + 1e-12) + pi.x());
261         if (intersect) inside = !inside;
```

```
262     }
263     return inside;
264 }
265
266 void MainWindow::weilerAthertonPolygonClip()
267 {
268     QVector<QPoint> origPolyInt = QVector<QPoint>::fromList(polygonVertices);
269     QVector<QPointF> origPoly; origPoly.reserve(origPolyInt.size());
270     for (const auto& q : origPolyInt) origPoly.append(QPointF(q));
271
272
273     QList<VertexData> subjectList;
274     for (const auto& p : polygonVertices) subjectList.append({QPointF(p)});
275
276     QList<VertexData> clipList;
277     QVector<QPoint> windowVerts = {
278         clippingWindow.topLeft(),
279         QPoint(clippingWindow.right(),  clippingWindow.top()),
280         clippingWindow.bottomRight(),
281         QPoint(clippingWindow.left(),   clippingWindow.bottom())
282     };
283     for (const auto& p : windowVerts) clipList.append({QPointF(p)});
284
285     bool allInside = true;
286     for (const auto& v : subjectList) {
287         if (!isInsideClipWindow(v.pos)) {
288             allInside = false; break;
289         }
290     }
291
292     QList<QPair<int, VertexData>> subjectInserts, clipInserts;
293     for (int i = 0; i < subjectList.size(); ++i) {
294         QPointF p1 = subjectList[i].pos;
295         QPointF p2 = subjectList[(i + 1) % subjectList.size()].pos;
296
297         for (int j = 0; j < clipList.size(); ++j) {
298             QPointF w1 = clipList[j].pos;
299             QPointF w2 = clipList[(j + 1) % clipList.size()].pos;
300
301             QPointF I = getIntersectionWA(p1, p2, w1, w2);
302             if (!I.isNull()) {
303                 bool isStartNode = isInsideClipWindow(p1) && !isInsideClipWindow(
                        p2);
304                 subjectInserts.append({ i + 1, { I, true, isStartNode, -1, false }
                        });
305                 clipInserts.append   ({ j + 1, { I, true, isStartNode, -1, false }
                        });
306             }
307         }
308     }
309
310     QList<QPoint> original = polygonVertices;
```

```
311     clearPolygon();
312     if (!original.isEmpty()) drawPolygonOutline(original, QBrush(Qt::lightGray),
            /*collect=*/true);
313     fillWindow(clippingWindow, QColor(173, 216, 230), 120);
314     drawRectangle(clippingWindow, QBrush(Qt::red), 1);
315
316
317     if (subjectInserts.isEmpty()) {
318         if (allInside) {
319             if (!original.isEmpty()) drawPolygonOutline(original, QBrush(Qt::green
                    ));
320         } else {
321             QPointF center((clippingWindow.left()+clippingWindow.right())/2.0,
322                            (clippingWindow.top() +clippingWindow.bottom())/2.0);
323             if (!origPoly.isEmpty() && pointInPolygon(origPoly, center)) {
324                 drawRectangle(clippingWindow, QBrush(Qt::green));
325             }
326         }
327         scene->update();
328         return;
329     }
330
331     std::sort(subjectInserts.begin(), subjectInserts.end(),
332               [](auto& a, auto& b){ return a.first > b.first; });
333     for (auto& ins : subjectInserts) subjectList.insert(ins.first, ins.second);
334
335     std::sort(clipInserts.begin(), clipInserts.end(),
336               [](auto& a, auto& b){ return a.first > b.first; });
337     for (auto& ins : clipInserts) clipList.insert(ins.first, ins.second);
338
339
340     for (int i = 0; i < subjectList.size(); ++i) {
341         if (!subjectList[i].isIntersection) continue;
342         for (int j = 0; j < clipList.size(); ++j) {
343             if (clipList[j].isIntersection && subjectList[i].pos == clipList[j].
                    pos) {
344                 subjectList[i].link = j;
345                 clipList[j].link = i;
346                 break;
347             }
348         }
349     }
350
351     QList<QVector<QPointF>> resultPolygons;
352     for (int i = 0; i < subjectList.size(); ++i) {
353         if (subjectList[i].isIntersection && subjectList[i].isStartNode && !
                subjectList[i].visited) {
354             QVector<QPointF> cur;
355             int idx = i;
356             bool onSubject = true;
357             do {
358                 VertexData* v = onSubject ? &subjectList[idx] : &clipList[idx];
```

```
359                 v->visited = true;
360                 cur.append(v->pos);
361                 if (v->isIntersection && v->link != -1) { onSubject = !onSubject;
                        idx = v->link; }
362                 idx = (idx + 1) % (onSubject ? subjectList.size() : clipList.size
                        ());
363             } while (cur.first() != cur.back() || cur.size() < 2);
364             cur.pop_back();
365             if (cur.size() >= 3) resultPolygons.append(cur);
366         }
367     }
368 
369 
370     if (!resultPolygons.isEmpty()) {
371         for (const auto& poly : resultPolygons) {
372             QList<QPoint> edge;
373             edge.reserve(poly.size());
374             for (const QPointF& p : poly) edge.append(QPoint(qRound(p.x()), qRound
                    (p.y())));
375             drawPolygonOutline(edge, QBrush(Qt::green));
376         }
377     }
378     scene->update();
379 }
380 
381 
382 
383 void MainWindow::onClearAll()
384 {
385     scene->clearCells();
386     polygonVertices.clear();
387     originalPolygonVertices.clear();
388     hasPolygon = false;
389     isSelectingVertices = false;
390     hasClippingWindow = false;
391     isDrawingWindow = false;
392     windowClickCount = 0;
393     scene->update();
394 }
395 
396 
397 void MainWindow::bresenhamLine(const QPoint& p1, const QPoint& p2, const QBrush&
        brush, bool collect)
398 {
399     int x1 = p1.x(), y1 = p1.y(), x2 = p2.x(), y2 = p2.y();
400     int dx = abs(x2 - x1), dy = abs(y2 - y1);
401     int sx = (x1 < x2) ? 1 : -1;
402     int sy = (y1 < y2) ? 1 : -1;
403     int err = dx - dy;
404 
405     while (true) {
406         QPoint q(x1, y1);
```

```
407             scene->paintCell(q, brush);
408             if (collect) polygonPixels.insert(q);
409 
410             if (x1 == x2 && y1 == y2) break;
411             int e2 = 2 * err;
412             if (e2 > -dy) { err -= dy; x1 += sx; }
413             if (e2 < dx) { err += dx; y1 += sy; }
414         }
415     }
416 
417 
418     void MainWindow::drawPolygonOutline(const QList<QPoint>& vertices, const QBrush&
        brush, bool collect)
419     {
420         if (vertices.size() < 2) return;
421 
422         for (const QPoint& v : vertices) {
423             scene->paintCell(v, brush);
424             if (collect) polygonPixels.insert(v);
425         }
426 
427         for (int i = 0; i < vertices.size(); i++) {
428             int nextIndex = (i + 1) % vertices.size();
429             bresenhamLine(vertices[i], vertices[nextIndex], brush, collect);
430         }
431     }
432 
433 
434     void MainWindow::drawRectangle(const QRect& rect, const QBrush& brush)
435     {
436         bresenhamLine(rect.topLeft(), rect.topRight(), brush);
437         bresenhamLine(rect.topRight(), rect.bottomRight(), brush);
438         bresenhamLine(rect.bottomRight(), rect.bottomLeft(), brush);
439         bresenhamLine(rect.bottomLeft(), rect.topLeft(), brush);
440     }
441 
442     void MainWindow::drawRectangle(const QRect& rect, const QBrush& brush, int
        thickness)
443     {
444         for (int i = 0; i < thickness; ++i) {
445             bresenhamLine(QPoint(rect.left() + i, rect.top() + i), QPoint(rect.right()
                 - i, rect.top() + i), brush);
446             bresenhamLine(QPoint(rect.left() + i, rect.bottom() - i), QPoint(rect.
                right() - i, rect.bottom() - i), brush);
447             bresenhamLine(QPoint(rect.left() + i, rect.top() + i), QPoint(rect.left()
                + i, rect.bottom() - i), brush);
448             bresenhamLine(QPoint(rect.right() - i, rect.top() + i), QPoint(rect.right
                () - i, rect.bottom() - i), brush);
449         }
450     }
451 
452 
```

```
453 void MainWindow::fillWindow(const QRect& rect, const QColor& color, int alpha)
454 {
455     for (int x = rect.left(); x <= rect.right(); ++x)
456         for (int y = rect.top(); y <= rect.bottom(); ++y)
457             scene->paintCell(QPoint(x, y), QBrush(Qt::transparent));
458     QColor translucent = color;
459     translucent.setAlpha(alpha);
460     for (int x = rect.left(); x <= rect.right(); ++x)
461         for (int y = rect.top(); y <= rect.bottom(); ++y)
462             scene->paintCell(QPoint(x, y), QBrush(translucent));
463 }
464
465
466 void MainWindow::clearPolygon()
467 {
468     if (polygonVertices.size() < 2) return;
469     polygonPixels.clear();
470
471     for (const QPoint& v : polygonVertices) {
472         bool skip = false;
473         if (hasClippingWindow) {
474             if (clippingWindow.contains(v)) skip = true;
475             bool onLeft = (v.x() == clippingWindow.left() && v.y() >=
                    clippingWindow.top() && v.y() <= clippingWindow.bottom());
476             bool onRight = (v.x() == clippingWindow.right() && v.y() >=
                    clippingWindow.top() && v.y() <= clippingWindow.bottom());
477             bool onTop = (v.y() == clippingWindow.top() && v.x() >= clippingWindow
                    .left() && v.x() <= clippingWindow.right());
478             bool onBottom = (v.y() == clippingWindow.bottom() && v.x() >=
                    clippingWindow.left() && v.x() <= clippingWindow.right());
479             if (onLeft || onRight || onTop || onBottom) skip = true;
480         }
481         if (!skip) scene->paintCell(v, QBrush(Qt::transparent));
482     }
483
484     for (int i = 0; i < polygonVertices.size(); i++) {
485         int nextIndex = (i + 1) % polygonVertices.size();
486         QPoint p1 = polygonVertices[i];
487         QPoint p2 = polygonVertices[nextIndex];
488         int x1 = p1.x(), y1 = p1.y(), x2 = p2.x(), y2 = p2.y();
489         int dx = std::abs(x2 - x1), dy = std::abs(y2 - y1);
490         int sx = (x1 < x2) ? 1 : -1;
491         int sy = (y1 < y2) ? 1 : -1;
492         int err = dx - dy;
493         while (true) {
494             bool skip = false;
495             if (hasClippingWindow) {
496                 if (x1 >= clippingWindow.left() && x1 <= clippingWindow.right() &&
497                     y1 >= clippingWindow.top()  && y1 <= clippingWindow.bottom())
                            skip = true;
498                 bool onLeft = (x1 == clippingWindow.left() && y1 >= clippingWindow
                        .top() && y1 <= clippingWindow.bottom());
```

```
499                 bool onRight = (x1 == clippingWindow.right() && y1 >=
                        clippingWindow.top() && y1 <= clippingWindow.bottom());
500                 bool onTop = (y1 == clippingWindow.top() && x1 >= clippingWindow.
                        left() && x1 <= clippingWindow.right());
501                 bool onBottom = (y1 == clippingWindow.bottom() && x1 >=
                        clippingWindow.left() && x1 <= clippingWindow.right());
502                 if (onLeft || onRight || onTop || onBottom) skip = true;
503             }
504             if (!skip) scene->paintCell(QPoint(x1, y1), QBrush(Qt::transparent));
505             if (x1 == x2 && y1 == y2) break;
506             int e2 = 2 * err;
507             if (e2 > -dy) { err -= dy; x1 += sx; }
508             if (e2 < dx) { err += dx; y1 += sy; }
509         }
510     }
511 }
512
513
514 void MainWindow::clearWindow()
515 {
516     if (hasClippingWindow) {
517         for (int x = clippingWindow.left(); x <= clippingWindow.right(); ++x)
518             for (int y = clippingWindow.top(); y <= clippingWindow.bottom(); ++y)
519                 scene->paintCell(QPoint(x, y), QBrush(Qt::transparent));
520         drawRectangle(clippingWindow, QBrush(Qt::transparent));
521     }
522 }
523
524 bool MainWindow::isInside(const QPointF& point, EdgePosition edge, const QRect&
        clipRect)
525 {
526     switch (edge) {
527     case LEFT_EDGE:
528         return point.x() >= clipRect.left();
529     case RIGHT_EDGE:
530         return point.x() <= clipRect.right();
531     case BOTTOM_EDGE:
532         return point.y() >= clipRect.top();
533     case TOP_EDGE:
534         return point.y() <= clipRect.bottom();
535     }
536     return false;
537 }
538
539 QPointF MainWindow::getIntersection(const QPointF& p1, const QPointF& p2,
        EdgePosition edge, const QRect& clipRect)
540 {
541     double x1 = p1.x(), y1 = p1.y(), x2 = p2.x(), y2 = p2.y();
542     double x = 0, y = 0;
543
544     switch (edge) {
545
```

```
546     case LEFT_EDGE:
547         x = clipRect.left();
548         if (x2 != x1) y = y1 + (y2 - y1) * (x - x1) / (x2 - x1); else y = y1;
549         y = std::max<double>(clipRect.top(), std::min<double>(clipRect.bottom(),
                std::round(y)));
550         return QPointF(x, y);
551 
552     case RIGHT_EDGE:
553         x = clipRect.right();
554         if (x2 != x1) y = y1 + (y2 - y1) * (x - x1) / (x2 - x1); else y = y1;
555         y = std::max<double>(clipRect.top(), std::min<double>(clipRect.bottom(),
                std::round(y)));
556         return QPointF(x, y);
557 
558     case BOTTOM_EDGE:
559         y = clipRect.top();
560         if (y2 != y1) x = x1 + (x2 - x1) * (y - y1) / (y2 - y1); else x = x1;
561         x = std::max<double>(clipRect.left(), std::min<double>(clipRect.right(),
                std::round(x)));
562         return QPointF(x, y);
563 
564     case TOP_EDGE:
565         y = clipRect.bottom();
566         if (y2 != y1) x = x1 + (x2 - x1) * (y - y1) / (y2 - y1); else x = x1;
567         x = std::max<double>(clipRect.left(), std::min<double>(clipRect.right(),
                std::round(x)));
568         return QPointF(x, y);
569     }
570     return QPointF();
571 }
572 
573 
574 QList<QPointF> MainWindow::clipAgainstEdge(const QList<QPointF>& inputVertices,
        EdgePosition edge, const QRect& clipRect)
575 {
576     QList<QPointF> outputVertices;
577 
578     if (inputVertices.isEmpty()) {
579         return outputVertices;
580     }
581 
582     QPointF previousVertex = inputVertices.last();
583 
584     for (const QPointF& currentVertex : inputVertices) {
585         bool currentInside = isInside(currentVertex, edge, clipRect);
586         bool previousInside = isInside(previousVertex, edge, clipRect);
587 
588         if (currentInside) {
589             if (!previousInside) {
590                 QPointF intersection = getIntersection(previousVertex,
                        currentVertex, edge, clipRect);
591                 outputVertices.append(intersection);
```

```
592             }
593             outputVertices.append(currentVertex);
594         }
595         else if (previousInside) {
596             QPointF intersection = getIntersection(previousVertex, currentVertex,
                    edge, clipRect);
597             outputVertices.append(intersection);
598         }
599 
600         previousVertex = currentVertex;
601     }
602 
603     return outputVertices;
604 }
605 
606 QList<QPointF> MainWindow::sutherlandHodgemanClip(const QList<QPointF>& polygon,
        const QRect& clipRect)
607 {
608     QList<QPointF> outputPolygon = polygon;
609     outputPolygon = clipAgainstEdge(outputPolygon, LEFT_EDGE, clipRect);
610 
611     if (outputPolygon.isEmpty()) return outputPolygon;
612     outputPolygon = clipAgainstEdge(outputPolygon, RIGHT_EDGE, clipRect);
613 
614     if (outputPolygon.isEmpty()) return outputPolygon;
615     outputPolygon = clipAgainstEdge(outputPolygon, BOTTOM_EDGE, clipRect);
616 
617     if (outputPolygon.isEmpty()) return outputPolygon;
618     outputPolygon = clipAgainstEdge(outputPolygon, TOP_EDGE, clipRect);
619 
620     if (outputPolygon.isEmpty()) return outputPolygon;
621 
622     QList<QPointF> snapped;
623     snapped.reserve(outputPolygon.size());
624     for (const QPointF& p : outputPolygon) {
625         QPointF q(qRound(p.x()), qRound(p.y()));
626         if (snapped.isEmpty() || q != snapped.last()) snapped.append(q);
627     }
628     if (snapped.size() >= 2 && snapped.first() == snapped.last()) snapped.
            removeLast();
629     return snapped;
630 }
631 
632 
633 QPointF MainWindow::computeLineIntersection(const QPointF& p1, const QPointF& p2,
        const QPointF& p3, const QPointF& p4)
634 {
635     double denom = (p1.x() - p2.x()) * (p3.y() - p4.y()) - (p1.y() - p2.y()) * (p3
            .x() - p4.x());
636 
637     if (qFuzzyIsNull(denom)) {
638         return QPointF();
```

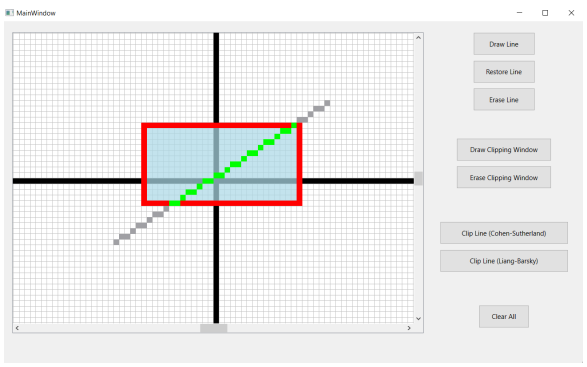
```
639     }
640 
641     double t = ((p1.x() - p3.x()) * (p3.y() - p4.y()) - (p1.y() - p3.y()) * (p3.x
            () - p4.x())) / denom;
642     double u = -((p1.x() - p2.x()) * (p1.y() - p3.y()) - (p1.y() - p2.y()) * (p1.x
            () - p3.x())) / denom;
643 
644     if (t >= 0 && t <= 1 && u >= 0 && u <= 1) {
645         return QPointF(p1.x() + t * (p2.x() - p1.x()), p1.y() + t * (p2.y() - p1.y
                ()));
646     }
647 
648     return QPointF();
649 }
650 
651 
652 double MainWindow::distance(const QPointF& p1, const QPointF& p2)
653 {
654     return std::sqrt(std::pow(p2.x() - p1.x(), 2) + std::pow(p2.y() - p1.y(), 2));
655 }
656 
657 
658 bool MainWindow::isPointInsidePolygon(const QPointF& point, const QList<QPointF>&
        polygon)
659 {
660     int crossings = 0;
661     int n = polygon.size();
662 
663     for (int i = 0; i < n; i++) {
664         const QPointF& p1 = polygon[i];
665         const QPointF& p2 = polygon[(i + 1) % n];
666 
667         if (((p1.y() <= point.y() && point.y() < p2.y()) ||
668              (p2.y() <= point.y() && point.y() < p1.y())) &&
669             point.x() < (p2.x() - p1.x()) * (point.y() - p1.y()) / (p2.y() - p1.y
                    ()) + p1.x()) {
670             crossings++;
671         }
672     }
673 
674     return (crossings % 2 == 1);
675 }
```

main.cpp

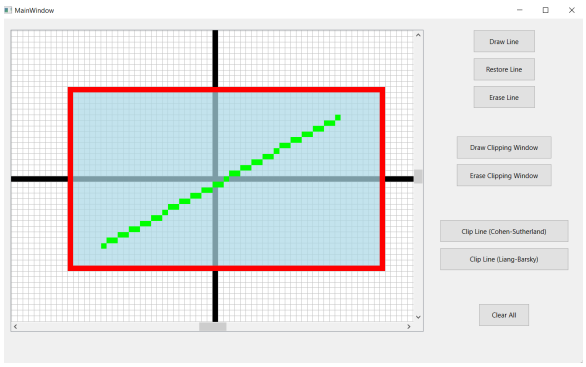
```
1         #include "mainwindow.h"
2 
3         #include <QApplication>
4 
5         int main(int argc, char *argv[])
6         {
7             QApplication a(argc, argv);
8             MainWindow w;
```

```
9               w.show();
10              return a.exec();
11          }
```

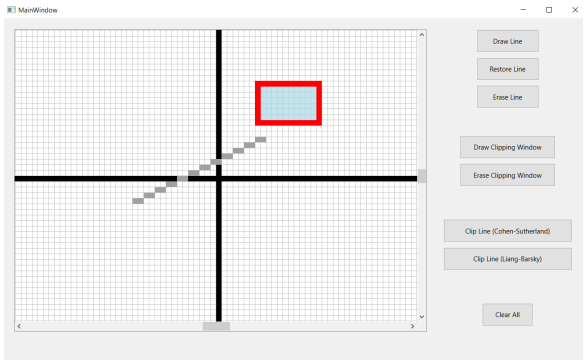
Outputs



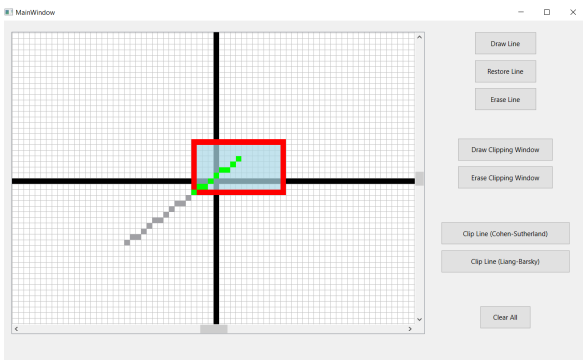
(a) line_clipping1



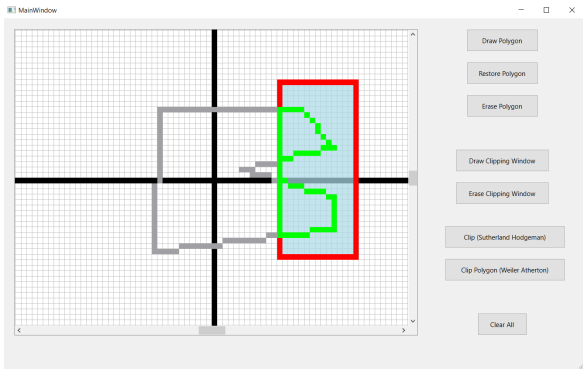
(b) line_clipping2



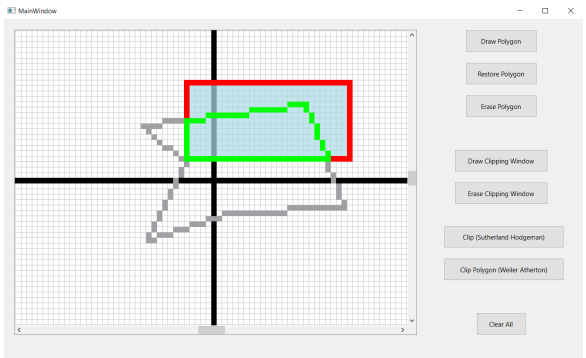
(a) line_clipping3



(b) line_clipping4



(a) cohen_sutherland_fail



(b) polygon_clipping

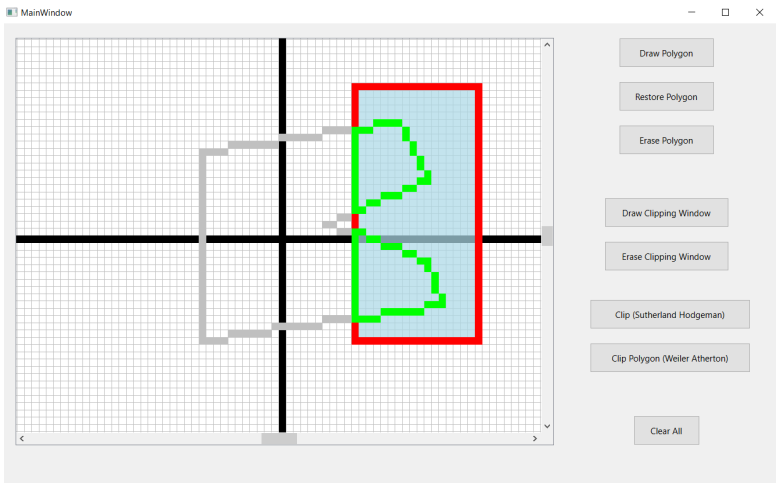


Figure 26: weiler_atherton_pass

Output Explanation

For line clipping, both Cohen–Sutherland and Liang–Barsky correctly produce the visible portion of the segment inside the rectangular window. In my implementation, Cohen–Sutherland iteratively moves endpoints towards the window using region codes, while Liang–Barsky directly computes the entering and leaving parameters along the line. Since Liang–Barsky works purely with a single parametric interval $[t_0, t_1]$ and does not repeatedly subdivide the line into segments, it is computationally more efficient for the same clipping task.

For polygon clipping, Sutherland–Hodgeman works correctly for convex polygons, but for concave inputs it may generate artefacts such as an extra connecting strip that incorrectly links what should be two separate clipped polygons. In contrast, the Weiler–Atherton implementation explicitly tracks intersection nodes and alternates traversal between the subject and clip polygons, so it correctly produces multiple disjoint clipped regions for concave polygons without any spurious connecting edges.

Assignment 7: Cubic Bézier Curve

Problem Statement (Optional)

Implement a cubic Bézier curve drawing algorithm for a given set of control points.

Code and Theoretical Explanation

In this assignment, I implemented a cubic Bézier curve drawing routine starting from the general spline formulation. A Bézier curve of degree n is defined for $n+1$ control points

$$p_k = (x_k, y_k), \quad k = 0, 1, \ldots, n$$

by the parametric position vector

$$P(u) = \sum_{k=0}^{n} p_k \, BEZ_{k,n}(u), \qquad 0 \le u \le 1,$$

where the blending (basis) functions $BEZ_{k,n}(u)$ are the Bernstein polynomials

$$BEZ_{k,n}(u) = C(n,k)\, u^k (1-u)^{n-k}, \qquad C(n,k) = \frac{n!}{k!(n-k)!}.$$

Writing $p_k = (x_k, y_k)$, the curve components are

$$x(u) = \sum_{k=0}^{n} x_k \, BEZ_{k,n}(u), \qquad y(u) = \sum_{k=0}^{n} y_k \, BEZ_{k,n}(u).$$

For a cubic Bézier curve we use $n = 3$ and four control points p_0, p_1, p_2, p_3. The blending functions simplify to

$$\begin{aligned} BEZ_{0,3}(u) &= (1-u)^3, \\ BEZ_{1,3}(u) &= 3u(1-u)^2, \\ BEZ_{2,3}(u) &= 3u^2(1-u), \\ BEZ_{3,3}(u) &= u^3, \end{aligned}$$

so the curve becomes

$$P(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u) p_2 + u^3 p_3.$$

This construction has several important properties: the curve starts and ends at the first and last control points ($P(0) = p_0$, $P(1) = p_3$); it is contained inside the convex hull of the control points (all $BEZ_{k,3}(u) \ge 0$ and $\sum_k BEZ_{k,3}(u) = 1$); and its end tangents are aligned with the first and last edges of the control polygon:

$$P'(0) = 3(p_1 - p_0), \qquad P'(1) = 3(p_3 - p_2).$$

The implementation follows this mathematical formulation directly. The helper function

```
long long choose(int n, int k) { ... }
```

computes the binomial coefficient $C(n,k)$, which appears in the Bernstein basis. In the drawing routine

```
void MainWindow::drawBezierCurve(const QVector<QPoint>& control_points) {
    int n = control_points.size() - 1;
    double u = 0, step = 1.0 / 730.0;
    ...
    while (u <= 1) {
        double xf = 0, yf = 0;
        for (int k = 0; k <= n; k++) {
            xf += pow(u, k) * pow(1 - u, n - k)
                  * control_points[k].x() * choose(n, k);
            yf += pow(u, k) * pow(1 - u, n - k)
                  * control_points[k].y() * choose(n, k);
        }
        int x = static_cast<int>(0.5 + xf);
        int y = static_cast<int>(0.5 + yf);
        ...
        to_fill.append({x, y});
        u += step;
    }
    addPoints(to_fill, QColor(200, 150, 20));
}
```

I first set $n = \texttt{control\_points.size()} - 1$, so with four control points we obtain a cubic curve ($n = 3$). For each parameter value $u \in [0, 1]$ (sampled in small increments of `step`), I evaluate the Bézier equations

$$x(u) = \sum_{k=0}^{n} C(n,k)\, u^k (1-u)^{n-k} x_k, \qquad y(u) = \sum_{k=0}^{n} C(n,k)\, u^k (1-u)^{n-k} y_k$$

by accumulating `xf` and `yf` in the inner loop. The term

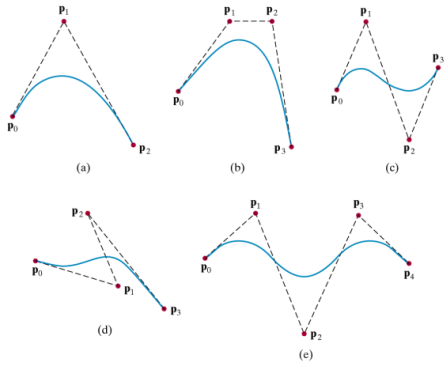
```
pow(u, k) * pow(1 - u, n - k) * choose(n, k)
```

is exactly $BEZ_{k,n}(u)$, and multiplying by the control point coordinates `control_points[k].x()` and `control_points[k].y()` implements the weighted sum that defines $P(u)$.

The floating-point positions (x_f, y_f) are rounded to the nearest integer grid cells using

```
int x = static_cast<int>(0.5 + xf);
int y = static_cast<int>(0.5 + yf);
```

to map the continuous curve to discrete pixels. To avoid redundant overplotting of the same grid cell when successive u samples quantise to identical integer coordinates, the code skips appending points whose (x, y) coincide with the previous sample. Finally, all sampled points are rendered at once via `addPoints`, producing a smooth visual approximation of the continuous cubic Bézier curve defined by the control polygon.

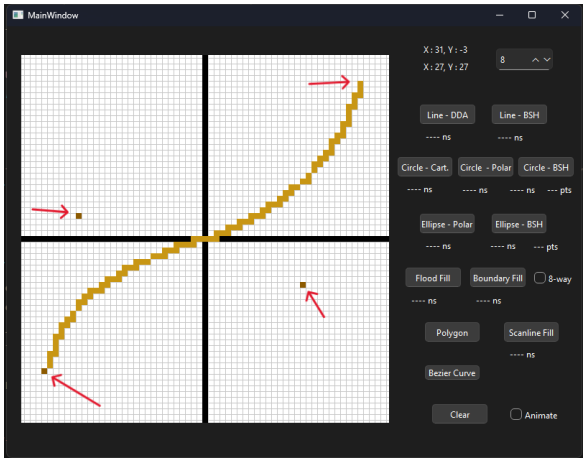


Code

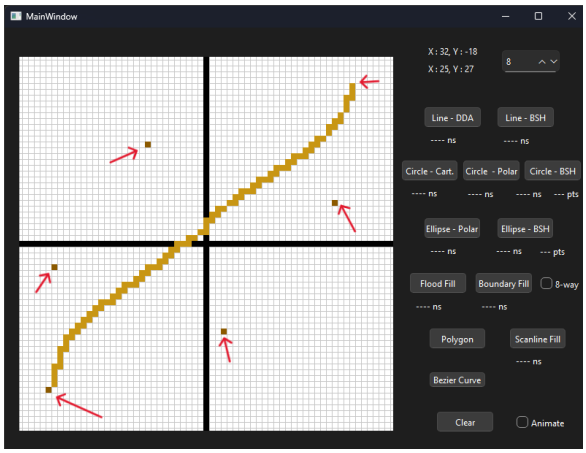
Cubic Bezier Curve:

```
long long choose(int n, int k) {
    if (k < 0 || k > n) return 0;
    if (k == 0 || k == n) return 1;

    if (k > n / 2) k = n - k;

    long long res = 1;
    for (int i = 1; i <= k; ++i)
        res = res * (n - i + 1) / i;
    return res;
}
void MainWindow::drawBezierCurve(const QVector<QPoint>& control_points) {
    int n = control_points.size() - 1, prev_x, prev_y;
    bool first_it = true;
    double u = 0, step = 1.0 / 730.0;
    QVector<QPoint> to_fill;
    to_fill.reserve(static_cast<int>(0.5 + (1.0 / step)));
    while (u <= 1) {
        double xf = 0, yf = 0;
        for (int k = 0; k <= n; k++) {
            xf += pow(u, k) * pow(1-u, n-k) * control_points[k].x() * choose(n,k);
            yf += pow(u, k) * pow(1-u, n-k) * control_points[k].y() * choose(n,k);
        }
        int x = static_cast<int>(0.5 + xf), y = static_cast<int>(0.5 + yf);
        if ((!first_it) && (x == prev_x) && (y == prev_y)) {
            u += step;
            continue;
        }
        if (first_it) first_it = false;
        prev_x = x;
        prev_y = y;
        to_fill.append({x, y});
        u += step;
    }
    addPoints(to_fill, QColor(200, 150, 20));
}
```

Outputs



(a) Bezier Curve with 4 control points



(b) Bezier Curve with 6 control points

Output Explanation

For a given set of control points, the implementation correctly evaluates the cubic Bézier formulation and plots a smooth curve that starts at the first control point, ends at the last control point, and follows the general shape of the control polygon. With four control points, the curve shows good visual smoothness and intuitive control: moving the inner control points changes the local shape while preserving endpoint interpolation and C^1 continuity at the ends.

When I increase the number of control points (e.g. to six, yielding a higher-degree Bézier), the curve is still drawn correctly, but each control point influences a comparatively large portion of the curve. As a result, the local contributions of individual control points are low: small changes to any one control point tend to disturb a wide region of the curve, illustrating the loss of strong local control for higher-degree Bézier representations.