Experiment Date: 26-05-2025

Experiment Name: Draw Square using DDA Line Drawing Algorithm

Description:

The Digital Differential Analyzer (DDA) algorithm is a simple and accurate method to draw a line between two points using floating-point arithmetic.

To draw a square, we connect its four corners using four DDA-generated lines.

Algorithm:

1. Starting point (x_1, y_1) and ending point (x_2, y_2)
2. Let (x_i, y_i) be any point on the line
3. Slope, m = ${}^{dy}/_{dx}$ where, $dy = y_{i+1} - y_i$ and $dx = x_{i+1} - x_i$
4. From above equation, we get

   $m = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$

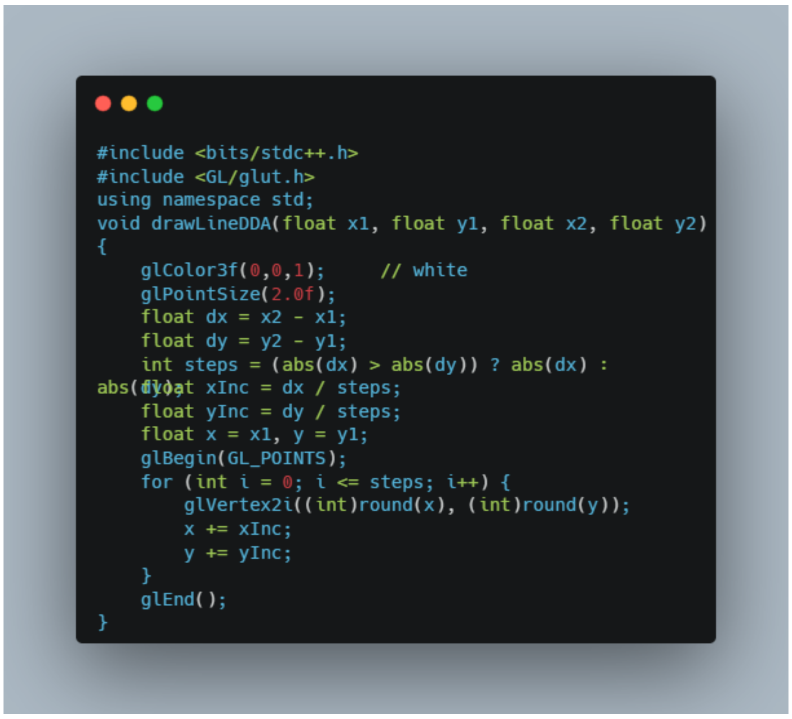
   ⇨ $y_{i+1} = y_i + mdx$ or, $x_{i+1} = x_i + {}^{dy}/_{m}$
5. If |m| <= 1, $x = x_1, y = y_1$ and set dx = 1

   That is, $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i + m$

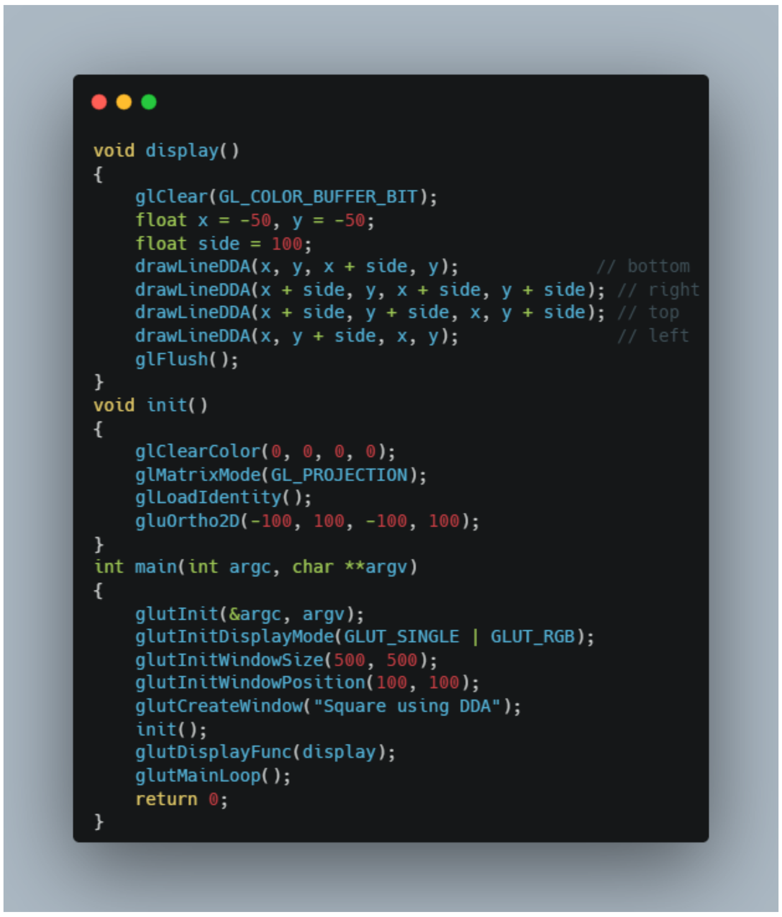
   Else, |m|>1, $x = x_1, y = y_1$ and set dy = 1

   That is, $x_{i+1} = x_i + {}^{1}/_{m}$ and $y_{i+1} = y_i + 1$
6. Continue until x reaches x_2 for |m| <= 1 case or, y reaches y_2 for |m|>1 case

Implementation:

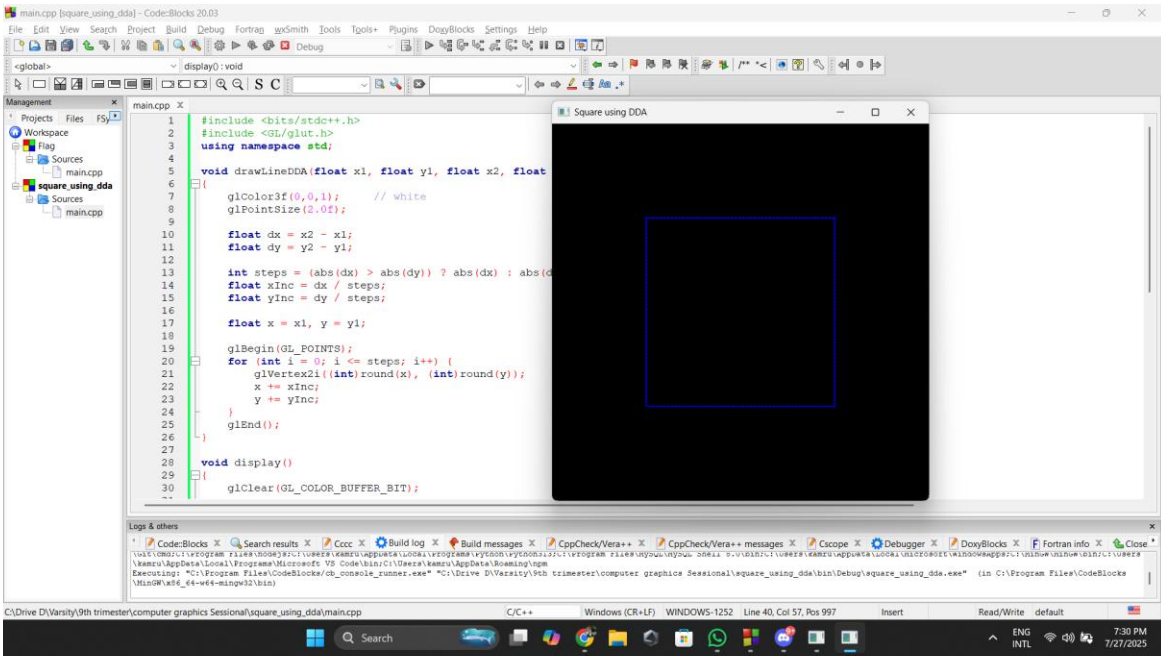


```
#include <bits/stdc++.h>
#include <GL/glut.h>
using namespace std;
void drawLineDDA(float x1, float y1, float x2, float y2)
{
    glColor3f(0,0,1);     // white
    glPointSize(2.0f);
    float dx = x2 - x1;
    float dy = y2 - y1;
    int steps = (abs(dx) > abs(dy)) ? abs(dx) :
abs(dy);
    float xInc = dx / steps;
    float yInc = dy / steps;
    float x = x1, y = y1;
    glBegin(GL_POINTS);
    for (int i = 0; i <= steps; i++) {
        glVertex2i((int)round(x), (int)round(y));
        x += xInc;
        y += yInc;
    }
    glEnd();
}
```



```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    float x = -50, y = -50;
    float side = 100;
    drawLineDDA(x, y, x + side, y);                   // bottom
    drawLineDDA(x + side, y, x + side, y + side); // right
    drawLineDDA(x + side, y + side, x, y + side); // top
    drawLineDDA(x, y + side, x, y);                   // left
    glFlush();
}
void init()
{
    glClearColor(0, 0, 0, 0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-100, 100, -100, 100);
}
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Square using DDA");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Output:



```
#include <bits/stdc++.h>
#include <GL/glut.h>
using namespace std;

void drawLineDDA(float x1, float y1, float x2, float
{
    glColor3f(0,0,1);      // white
    glPointSize(2.0f);

    float dx = x2 - x1;
    float dy = y2 - y1;

    int steps = (abs(dx) > abs(dy)) ? abs(dx) : abs(d
    float xInc = dx / steps;
    float yInc = dy / steps;

    float x = x1, y = y1;

    glBegin(GL_POINTS);
    for (int i = 0; i <= steps; i++) {
        glVertex2i((int)round(x), (int)round(y));
        x += xInc;
        y += yInc;
    }
    glEnd();
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
```

```
\kamru\AppData\Local\Programs\Microsoft VS Code\bin;C:\Users\kamru\AppData\Roaming\npm
Executing: "C:\Program Files\CodeBlocks/cb_console_runner.exe" "C:\Drive D\Varsity\9th trimester\computer graphics Sessional\square_using_dda\bin\Debug\square_using_dda.exe"  (in C:\Program Files\CodeBlocks
\MinGW\x86_64-w64-mingw32\bin)
```

main.cpp [square_using_dda] - Code::Blocks 20.03
Square using DDA
C:\Drive D\Varsity\9th trimester\computer graphics Sessional\square_using_dda\main.cpp
Line 40, Col 57, Pos 997
7:30 PM
7/27/2025

Discussion: In this experiment, we implemented the DDA line drawing algorithm to draw a square. The algorithm divides the line into small steps and calculates intermediate points using floating-point arithmetic, ensuring smooth lines. This method is simple and works effectively for shapes made of straight lines, like squares or rectangles. However, it can be computationally more expensive due to floating-point operations compared to Bresenham's algorithm.

Experiment Date: 16-06-2025

Experiment Name: Draw rectangle Bresenham line drawing algorithm

Description: Bresenham's line algorithm is an *integer* incremental scan-conversion method. It decides the next pixel using only additions/subtractions and comparisons—no floating points—so it's faster than DDA.

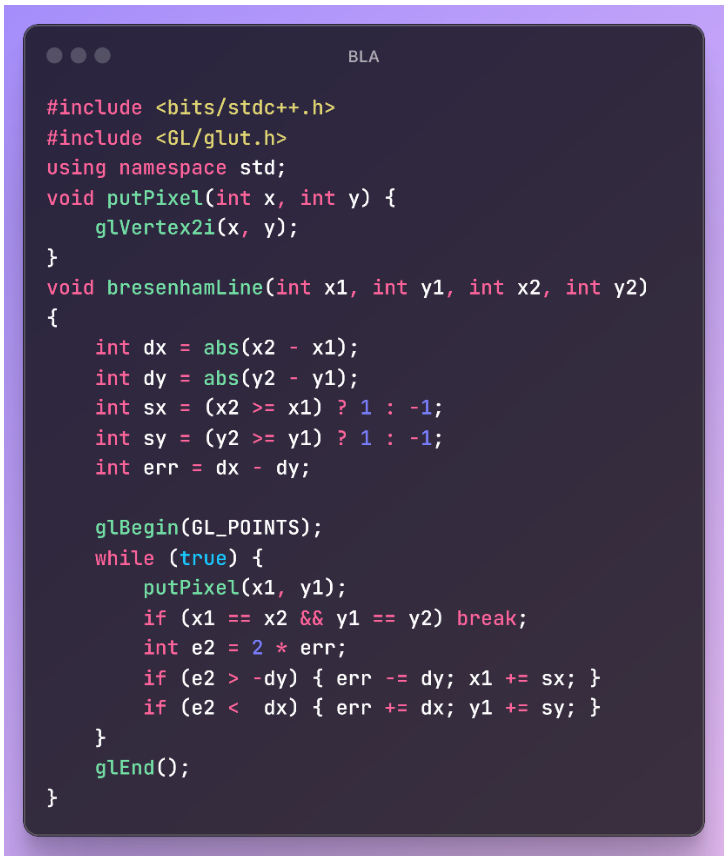
To draw a rectangle, we just draw 4 Bresenham lines: bottom, right, top, and left.

Algorithm:

1. Compute the initial values:

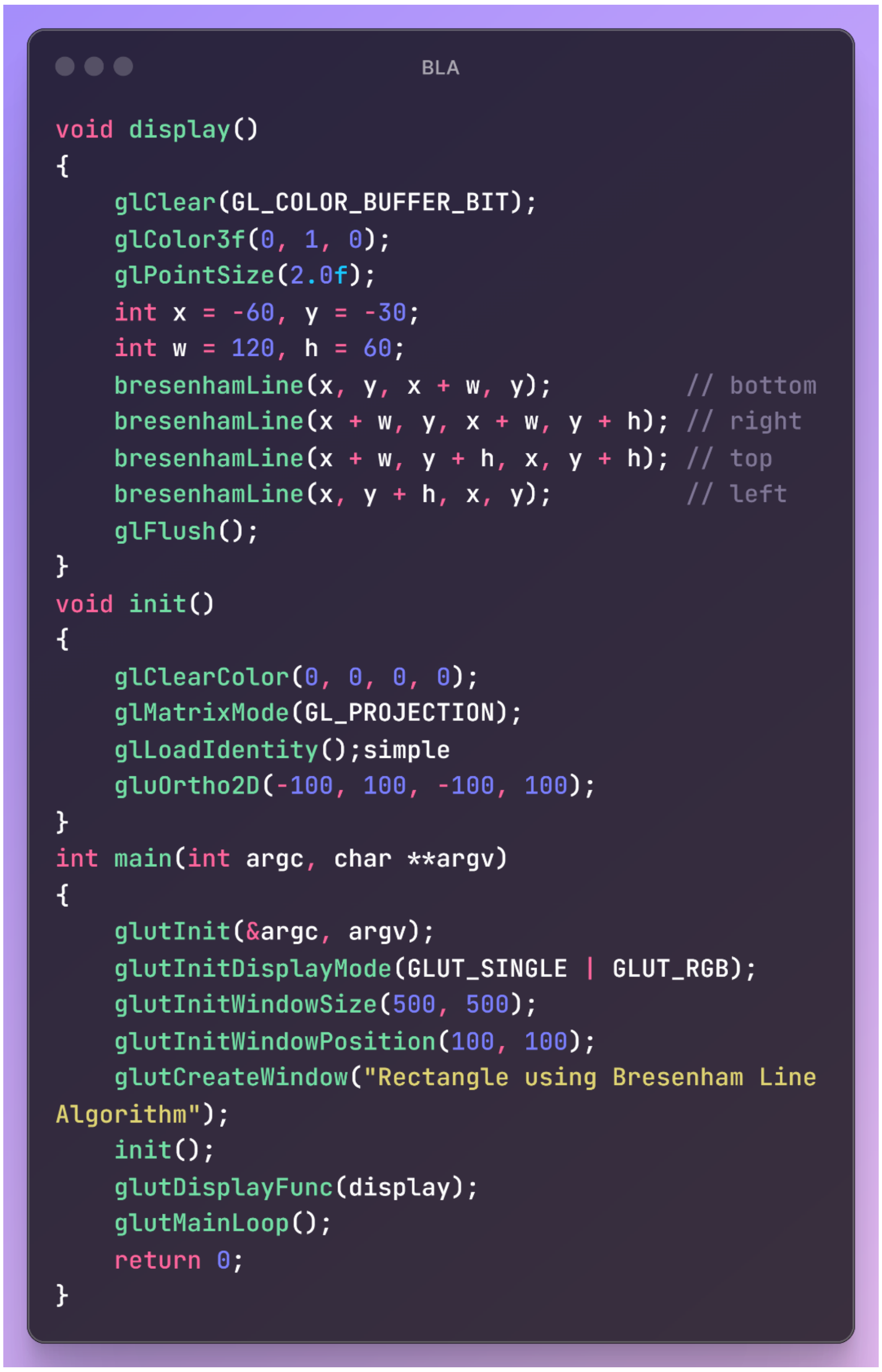
   dx = x_2 - x_1, dy = y_2 - y_1, Inc_1= 2dy, d = $Inc_1 - dx$, Inc_2= 2(dy – dx)
2. Set (x, y) equal to the lower left-hand endpoint and x_{end} equal to the largest value of x. If dx < 0, then x = x_2, y = y_2, x_{end} = x_1. If dx > 0, then x = x_1, y = y_1, x_{end} = x_2.
3. Plot a point at the current (x, y) coordinates.
4. Test to see whether the entire line has been drawn. If x = x_{end}, stop.
5. Compute the location of the next pixel. If d < 0, then d = d + Inc_1. If d ≥ 0, then d = d + Inc_2 and then y = y + 1.
6. Increment x: x = x +1.
7. Plot a point at the current (x, y) coordinates.
8. Go to step 4.

Implementation:



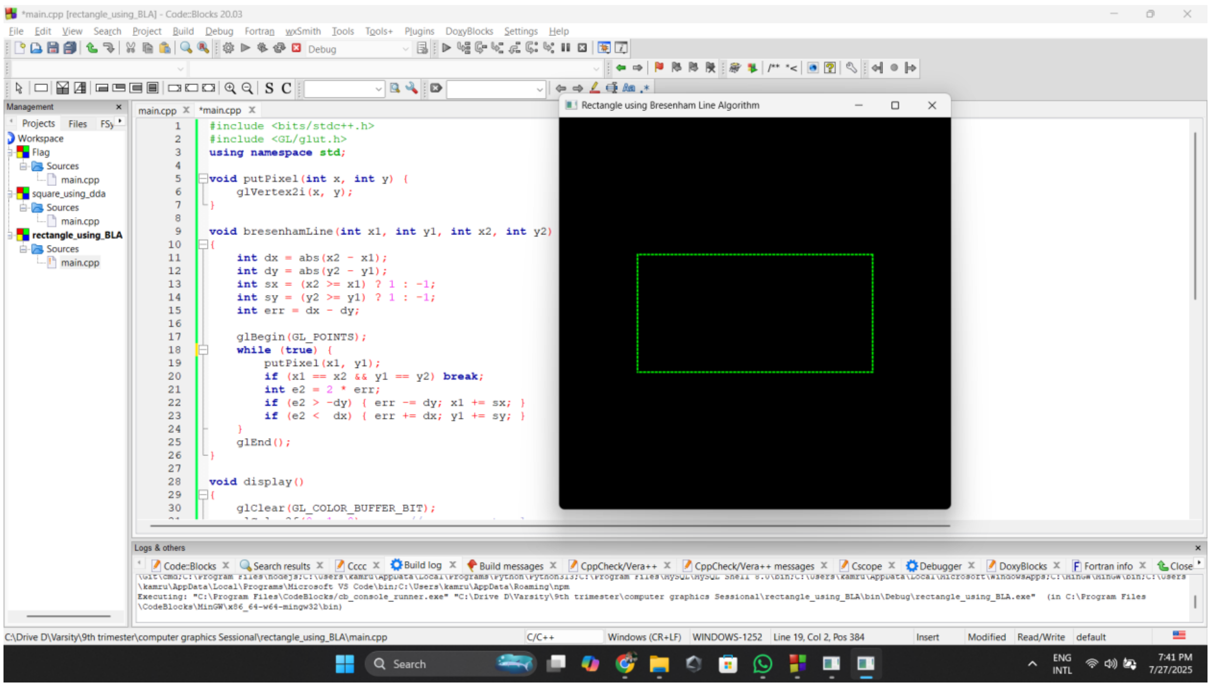
```
#include <bits/stdc++.h>
#include <GL/glut.h>
using namespace std;
void putPixel(int x, int y) {
    glVertex2i(x, y);
}
void bresenhamLine(int x1, int y1, int x2, int y2)
{
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);
    int sx = (x2 >= x1) ? 1 : -1;
    int sy = (y2 >= y1) ? 1 : -1;
    int err = dx - dy;

    glBegin(GL_POINTS);
    while (true) {
        putPixel(x1, y1);
        if (x1 == x2 && y1 == y2) break;
        int e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x1 += sx; }
        if (e2 <  dx) { err += dx; y1 += sy; }
    }
    glEnd();
}
```



```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0, 1, 0);
    glPointSize(2.0f);
    int x = -60, y = -30;
    int w = 120, h = 60;
    bresenhamLine(x, y, x + w, y);         // bottom
    bresenhamLine(x + w, y, x + w, y + h); // right
    bresenhamLine(x + w, y + h, x, y + h); // top
    bresenhamLine(x, y + h, x, y);         // left
    glFlush();
}
void init()
{
    glClearColor(0, 0, 0, 0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();simple
    gluOrtho2D(-100, 100, -100, 100);
}
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Rectangle using Bresenham Line
Algorithm");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Output:



```
#include <bits/stdc++.h>
#include <GL/glut.h>
using namespace std;

void putPixel(int x, int y) {
    glVertex2i(x, y);
}

void bresenhamLine(int x1, int y1, int x2, int y2)
{
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);
    int sx = (x2 >= x1) ? 1 : -1;
    int sy = (y2 >= y1) ? 1 : -1;
    int err = dx - dy;

    glBegin(GL_POINTS);
    while (true) {
        putPixel(x1, y1);
        if (x1 == x2 && y1 == y2) break;
        int e2 = 2 * err;
        if (e2 > -dy) { err -= dy; x1 += sx; }
        if (e2 <  dx) { err += dx; y1 += sy; }
    }
    glEnd();
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
```

```
\kamru\AppData\Local\Programs\Microsoft VS Code\bin;C:\Users\kamru\AppData\Roaming\npm
Executing: "C:\Program Files\CodeBlocks/cb_console_runner.exe" "C:\Drive D\Varsity\9th trimester\computer graphics Sessional\rectangle_using_BLA\bin\Debug\rectangle_using_BLA.exe"  (in C:\Program Files\CodeBlocks\MinGW\x86_64-w64-mingw32\bin)
```

*main.cpp [rectangle_using_BLA] - Code::Blocks 20.03
Rectangle using Bresenham Line Algorithm
C:\Drive D\Varsity\9th trimester\computer graphics Sessional\rectangle_using_BLA\main.cpp
Line 19, Col 2, Pos 384
7:41 PM
7/27/2025

Discussion:

In this experiment, we used Bresenham's line drawing algorithm to draw a rectangle. The algorithm uses only integer arithmetic to determine the nearest pixel to approximate the line, which makes it faster and more efficient than DDA. It is highly suitable for systems with limited processing power and works accurately for shapes like rectangles composed of straight edges.

Experiment Date: 23-06-2025

Experiment Name: Draw national flag using Bresenham line drawing and Bresenham circle drawing algorithm

Description:

To draw the national flag of Bangladesh, we combine two efficient rasterization techniques — Bresenham's Line Drawing Algorithm for the flag's rectangular body and Bresenham's Circle Drawing Algorithm for the red circle at the center-left.

These algorithms use only integer operations, which makes them optimal for digital displays and embedded systems where floating-point calculations are costly.

Algorithm:

Bresenham Line Drawing:

1. Compute the initial values:

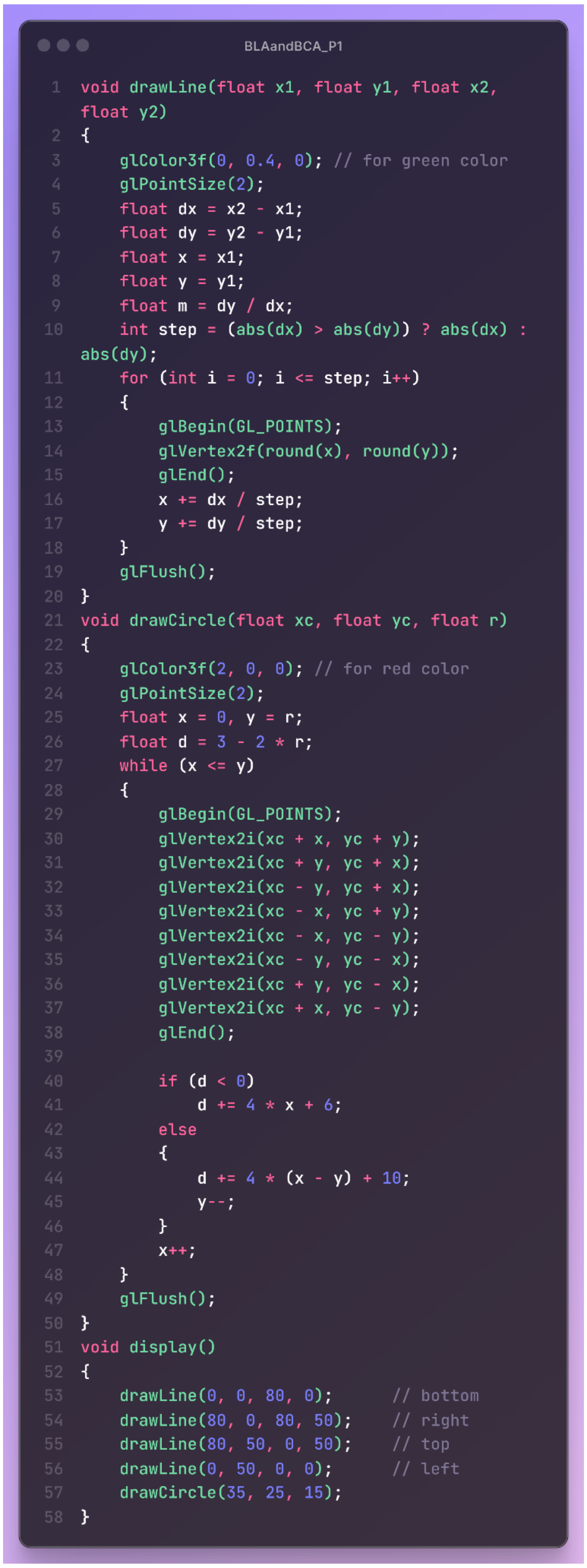
   dx = x_2 - x_1,   dy = y_2 - y_1, Inc_1= 2dy,   d = $Inc_1 - dx$, Inc_2= 2(dy – dx)
2. Set (x, y) equal to the lower left-hand endpoint and x_{end} equal to the largest value of x. If dx < 0, then x = x_2, y = y_2, x_{end} = x_1. If dx > 0, then x = x_1, y = y_1, x_{end} = x_2.
3. Plot a point at the current (x, y) coordinates.
4. Test to see whether the entire line has been drawn. If x = x_{end}, stop.
5. Compute the location of the next pixel. If d < 0, then d = d + Inc_1. If d ≥ 0, then d = d + Inc_2 and then y = y + 1.
6. Increment x: x = x +1.
7. Plot a point at the current (x, y) coordinates.
8. Go to step 4.

Bresenham Circle Drawing:

1. Take center and take radius of the circle
2. Set initial point (x, y) at (0, r) where r being the radius
3. Compute d: d = 3 – 2r
4. Test to determine whether the entire circle is drawn. If x > y, then stop
5. Plot (x, y)
6. If d < 0, then d = d+4x+6, x = x+1

   Else d = d+4(x-y)+10, x = x+1, y = y-1
7. Go to step 4

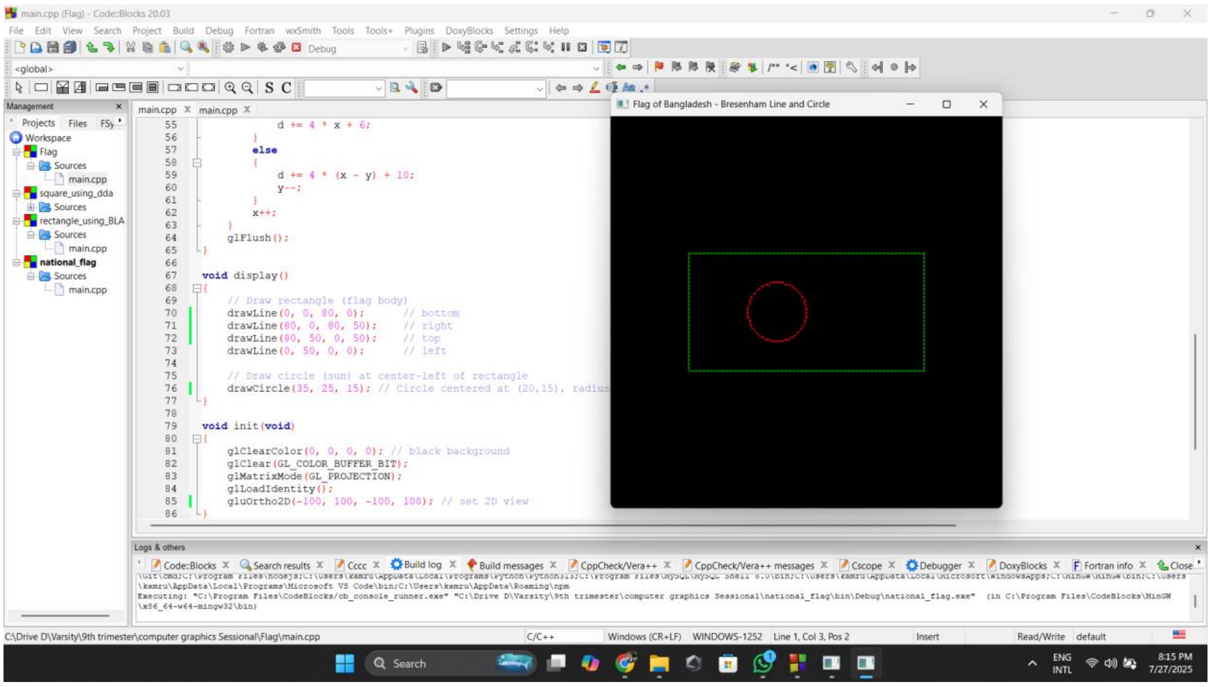
Implementation:



```
void drawLine(float x1, float y1, float x2, float y2)
{
    glColor3f(0, 0.4, 0); // for green color
    glPointSize(2);
    float dx = x2 - x1;
    float dy = y2 - y1;
    float x = x1;
    float y = y1;
    float m = dy / dx;
    int step = (abs(dx) > abs(dy)) ? abs(dx) : abs(dy);
    for (int i = 0; i <= step; i++)
    {
        glBegin(GL_POINTS);
        glVertex2f(round(x), round(y));
        glEnd();
        x += dx / step;
        y += dy / step;
    }
    glFlush();
}
void drawCircle(float xc, float yc, float r)
{
    glColor3f(2, 0, 0); // for red color
    glPointSize(2);
    float x = 0, y = r;
    float d = 3 - 2 * r;
    while (x <= y)
    {
        glBegin(GL_POINTS);
        glVertex2i(xc + x, yc + y);
        glVertex2i(xc + y, yc + x);
        glVertex2i(xc - y, yc + x);
        glVertex2i(xc - x, yc + y);
        glVertex2i(xc - x, yc - y);
        glVertex2i(xc - y, yc - x);
        glVertex2i(xc + y, yc - x);
        glVertex2i(xc + x, yc - y);
        glEnd();

        if (d < 0)
            d += 4 * x + 6;
        else
        {
            d += 4 * (x - y) + 10;
            y--;
        }
        x++;
    }
    glFlush();
}
void display()
{
    drawLine(0, 0, 80, 0);      // bottom
    drawLine(80, 0, 80, 50);    // right
    drawLine(80, 50, 0, 50);    // top
    drawLine(0, 50, 0, 0);      // left
    drawCircle(35, 25, 15);
}
```

Output:


main.cpp (Flag) - Code::Blocks 20.03
Flag of Bangladesh - Bresenham Line and Circle

```
                d += 4 * x + 6;
            }
            else
            {
                d += 4 * (x - y) + 10;
                y--;
            }
            x++;
        }
        glFlush();
}

void display()
{
    // Draw rectangle (flag body)
    drawLine(0, 0, 80, 0);       // bottom
    drawLine(80, 0, 80, 50);     // right
    drawLine(80, 50, 0, 50);     // top
    drawLine(0, 50, 0, 0);       // left

    // Draw circle (sun) at center-left of rectangle
    drawCircle(35, 25, 15); // Circle centered at (20,15), radius
}

void init(void)
{
    glClearColor(0, 0, 0, 0); // black background
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-100, 100, -100, 100); // set 2D view
}
```

```
\kamru\AppData\Local\Programs\Microsoft VS Code\bin;C:\Users\kamru\AppData\Roaming\npm
Executing: "C:\Program Files\CodeBlocks/cb_console_runner.exe" "C:\Drive D\Varsity\9th trimester\computer graphics Sessional\national_flag\bin\Debug\national_flag.exe"  (in C:\Program Files\CodeBlocks\MinGW
\x86_64-w64-mingw32\bin)
```

C:\Drive D\Varsity\9th trimester\computer graphics Sessional\Flag\main.cpp

Discussion:
In this experiment, we drew the national flag of Bangladesh by combining two Bresenham algorithms — one for lines and one for circles. The rectangle part of the flag was drawn using Bresenham's line algorithm, while the red circle was rendered using the circle drawing algorithm. This showed how multiple algorithms can be used together to construct complex figures with high precision and minimal computational cost.

Experiment Date: 23-06-2025

Experiment Name: Mid-point circle drawing algorithm

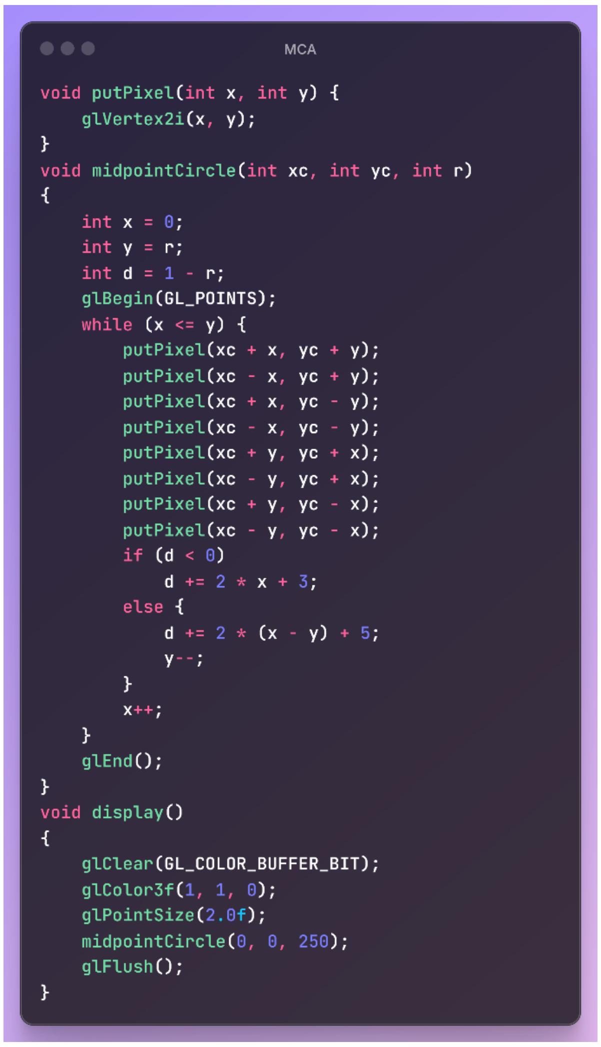
Description:

The Midpoint Circle Algorithm is a rasterization technique that uses decision parameters and symmetry to efficiently plot points on a circle using only integer arithmetic. This method avoids floating-point operations and trigonometric functions, making it suitable for low-resource graphics systems.

Algorithm:

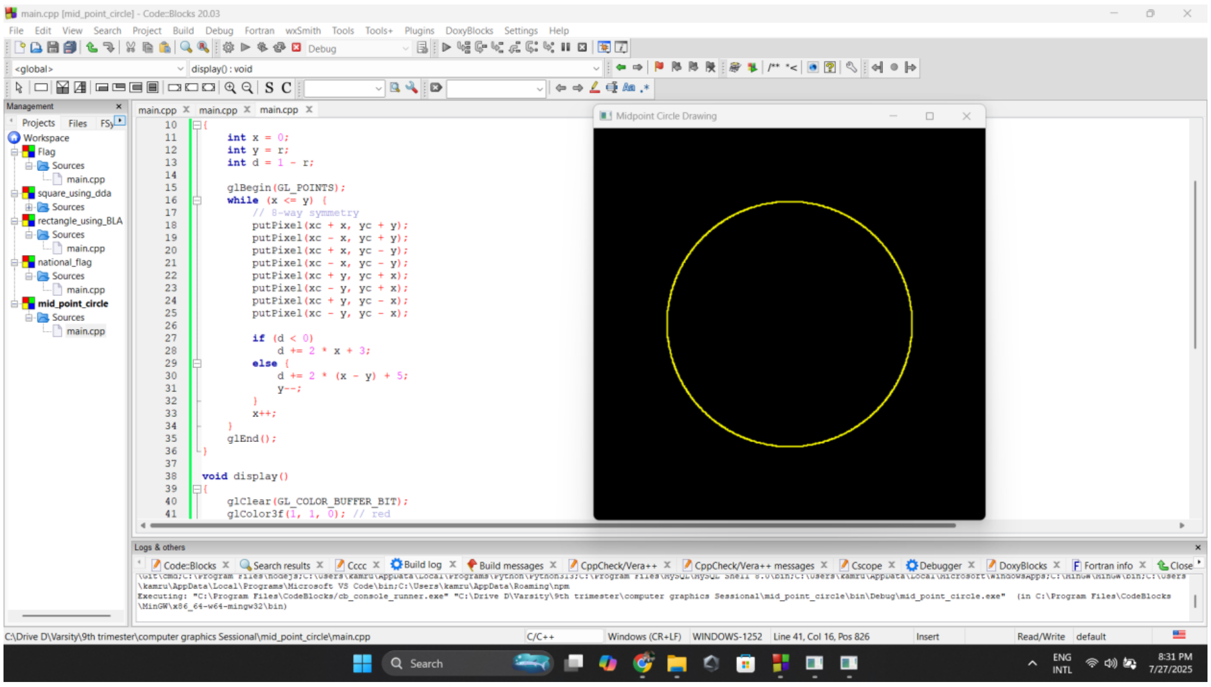
```
1. Take center and radius of the circle
2. Set initial point (x, y) at (0, r) where r being the radius
3. Compute p: p = 1 – r
4. Test to determine whether the entire circle is drawn. If x > y, then stop
5. Plot (x, y)
6. If p < 0, then p = p+2x+3, x = x+1
   Else p = p+2(x-y)+5, x = x+1, y = y-1
7. Go to step 4
```

Implementation:



```
void putPixel(int x, int y) {
    glVertex2i(x, y);
}
void midpointCircle(int xc, int yc, int r)
{
    int x = 0;
    int y = r;
    int d = 1 - r;
    glBegin(GL_POINTS);
    while (x <= y) {
        putPixel(xc + x, yc + y);
        putPixel(xc - x, yc + y);
        putPixel(xc + x, yc - y);
        putPixel(xc - x, yc - y);
        putPixel(xc + y, yc + x);
        putPixel(xc - y, yc + x);
        putPixel(xc + y, yc - x);
        putPixel(xc - y, yc - x);
        if (d < 0)
            d += 2 * x + 3;
        else {
            d += 2 * (x - y) + 5;
            y--;
        }
        x++;
    }
    glEnd();
}
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1, 1, 0);
    glPointSize(2.0f);
    midpointCircle(0, 0, 250);
    glFlush();
}
```

Output:



```
{
    int x = 0;
    int y = r;
    int d = 1 - r;

    glBegin(GL_POINTS);
    while (x <= y) {
        // 8-way symmetry
        putPixel(xc + x, yc + y);
        putPixel(xc - x, yc + y);
        putPixel(xc + x, yc - y);
        putPixel(xc - x, yc - y);
        putPixel(xc + y, yc + x);
        putPixel(xc - y, yc + x);
        putPixel(xc + y, yc - x);
        putPixel(xc - y, yc - x);

        if (d < 0)
            d += 2 * x + 3;
        else {
            d += 2 * (x - y) + 5;
            y--;
        }
        x++;
    }
    glEnd();
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1, 1, 0); // red
```

```
\Git\cmd;C:\Program Files\nodejs;C:\Users\kamru\AppData\Local\Programs\Python\Python313;C:\Program Files\MySQL\MySQL Shell 8.0\bin;C:\Users\kamru\AppData\Local\Microsoft\WindowsApps;C:\MinGW\MinGW\bin;C:\Users
\kamru\AppData\Local\Programs\Microsoft VS Code\bin;C:\Users\kamru\AppData\Roaming\npm
Executing: "C:\Program Files\CodeBlocks/cb_console_runner.exe" "C:\Drive D\Varsity\9th trimester\computer graphics Sessional\mid_point_circle\bin\Debug\mid_point_circle.exe"  (in C:\Program Files\CodeBlocks
\MinGW\x86_64-w64-mingw32\bin)
```

main.cpp [mid_point_circle] - Code::Blocks 20.03
Midpoint Circle Drawing
C:\Drive D\Varsity\9th trimester\computer graphics Sessional\mid_point_circle\main.cpp
Line 41, Col 16, Pos 826

Discussion:

In this experiment, we implemented the Midpoint Circle Drawing Algorithm to draw a circle. The algorithm uses a decision parameter to select the next pixel, leveraging symmetry to reduce computations. Unlike floating-point-based methods, it only uses integer arithmetic, making it faster and well-suited for real-time applications where circles are frequently used, such as in GUIs and game graphics.