Universidad de Las Fuerzas Armadas - ESPE

Departamento de Ciencias de la Computación (DCCO)

Carrera de Ingeniería de Software

Computación Gráfica - NRC23463

Tema: Implementación de Algoritmos Gráficos Básicos

Estudiante:

Luis Sagnay

Docente:

Ing. Dario Morales

Fecha:
2025-06-15

Índice

Trazado y Relleno de Figuras con Algoritmos Gráficos en C#

1. Introducción

El presente documento explica el desarrollo del programa en C# .NET Framework, el cual permite al usuario ingresar coordenadas para graficar líneas rectas utilizando los algoritmos DDA (Digital Differential Analyzer) y Bresenham. También se incluye la discretización de circunferencias con el algoritmo de Bresenham mediante la técnica de los 8 puntos simétricos, y un algoritmo de relleno por inundación que actúa sobre polígonos regulares generados automáticamente. Todos los algoritmos muestran visualmente el encendido progresivo de los píxeles, acompañado de una tabla con las coordenadas de cada punto encendido.

2. Objetivos

2.1. Objetivo General

El presente programa tiene como finalidad la implementación y visualización de algoritmos fundamentales de computación gráfica utilizando el lenguaje de programación C# y la plataforma .NET Framework. Se busca proporcionar una herramienta didáctica que permita comprender el funcionamiento de los algoritmos de discretización de líneas y circunferencias, así como técnicas de relleno por inundación, mediante una interfaz gráfica interactiva.

2.2. Objetivos Específicos

- Implementar el algoritmo DDA para el trazado de líneas rectas entre dos puntos
- Implementar el algoritmo de Bresenham para la discretización de líneas.
- Visualizar paso a paso el cálculo de las coordenadas intermedias para el trazado de líneas.
- Implementar el algoritmo de Bresenham para el trazado de circunferencias, visualizar la simetría octante para optimizar el cálculo de puntos.
- Implementar el relleno por inundación basado en recursión.
- Permitir la selección del color de relleno por parte del usuario.
- Permitir la introducción manual de puntos de inicio/fin mediante inputs en un formulario.
- Mostrar en un DataGridView los valores calculados en cada paso

3. Arquitectura del Sistema

El sistema está estructurado siguiendo los principios de programación orientada a objetos y el patrón de diseño por capas, lo que facilita la separación de responsabilidades, el mantenimiento del código y la escalabilidad del proyecto.

3.1. Capas del Programa

3.1.1. Capa de Dominio (Domain)

Contiene los modelos y las clases abstractas base que definen las estructuras fundamentales del sistema.

- Point2D: modelo para representar puntos en 2D.
- AlgorithmCalculator: Clase abstracta base que define la estructura general para cualquier algoritmo gráfico (con método Draw, herramientas de dibujo y animación).

3.1.2. Capa de Algoritmos (Algorithms)

En el sistema se han implementado cuatro algoritmos gráficos principales, cada uno encapsulado en su propia clase especializada

- LineAlgorithm (abstracta): Hereda de AlgorithmCalculator. Está diseñada específicamente para algoritmos de dibujo de líneas y proporciona herramientas adicionales para trabajar con puntos iniciales y finales.
- DDAAlgorithm: Contiene la lógica del algoritmo Analizador Diferencial Digital para dibujar líneas
- BresenhamLineAlgorithm: Contiene la lógica del algoritmo Bresenham para dibujar líneas
- BresenhamCircleAlgorithm: Contiene la lógica del algoritmo Bresenham para dibujar circunferencias
- FloodFillAlgorithm: Contiene la lógica para dibujar un polígono y la lógica del algoritmo FloodFill para pintar el polígono.

3.1.3. Capa de Utilidades (Utils)

Contiene clases auxiliares para generación de polígonos, gestión de colores, y manejo de datos en la interfaz.

- PolygonGenerator: clase con el método estático para calcular los puntos del polígono a dibujar, en donde solo recibe el número de lados del polígono.
- ColorHelper: clase con método estático para verificar si un pixel ya fué pintado o no
- Table: clase con métodos estáticos para la creación de la tabla donde recibe el objeto de DataGridView.

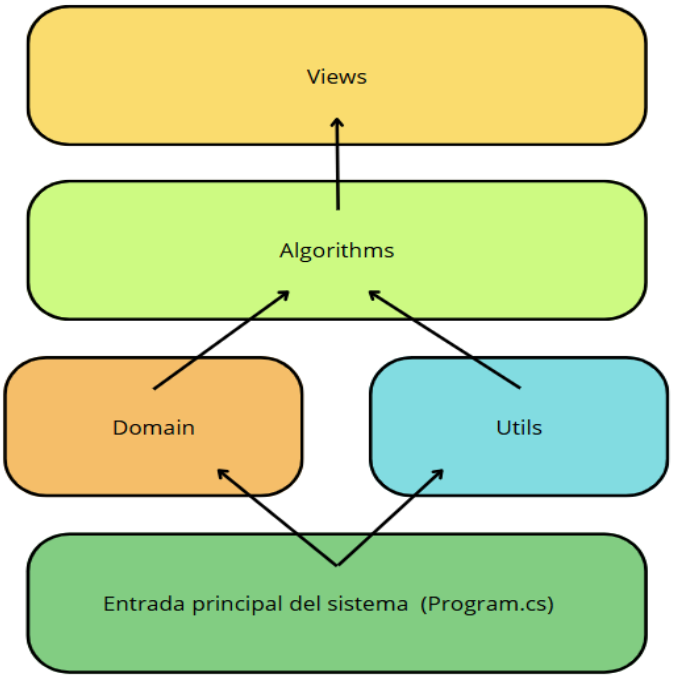
3.1.4. Capa de Presentación (Forms)

Conformada por formularios Windows Forms (MainForm, FrmDDA, FrmBrasenham, etc.), que gestionan la interacción con el usuario. Estos formularios utilizan los algoritmos y muestran los resultados en un PictureBox y una DataGridView.

3.1.5. Archivo de Entrada (Program.cs)

Punto de entrada del programa, que inicializa el formulario principal.

3.2 Representación Gráfica de la Arquitectura



4. Detalles de la Implementación

Esta sección describe la lógica y el comportamiento de los principales algoritmos gráficos implementados en el sistema.

4.1 Algoritmo DDA (Digital Differential Analyzer)

El algoritmo DDA genera puntos intermedios de una línea utilizando incrementos fraccionales en x o y, dependiendo de la pendiente. Se basa en la ecuación de la línea y se utiliza la aritmética de punto flotante. Cómo utiliza operaciones en punto flotante al redondear puede haber acumulación de errores pequeños, lo que en líneas largas se traduce en una menor precisión del trazo.

Lógica implementada:

Se determina la diferencia entre los puntos finales de la línea en los ejes x y y, es decir, dx: la diferencia horizontal y dy: la diferencia vertical

A partir de dx y dy, se calcula la pendiente de la línea: $m = dy / dx$ (si $dx \neq 0$)
Para asegurarse de cubrir todos los puntos entre el inicio y el fin, el algoritmo define el

número de pasos como el valor máximo entre |dx| y |dy|. Esto garantiza una transición suave, independientemente de si la línea es más horizontal o vertical.

Según el número de pasos, se calculan los valores fraccionarios que se deben sumar en cada iteración para avanzar en el eje x y y.

- Si la pendiente $|m| \leq 1$, se incrementa x en 1 unidad por paso y y en m unidades.
- Si $|m| > 1$, se incrementa y en 1 unidad por paso y x en 1/m.

Se parte desde el punto inicial y en cada iteración se actualizan las coordenadas x e y utilizando los incrementos. Aunque las coordenadas calculadas pueden tener valores decimales, se redondean al entero más cercano para determinar qué píxel se debe encender en la pantalla.

En cada paso se dibuja un píxel en el canvas en la ubicación calculada. También se actualiza una la donde se muestra el paso a paso del valor de x e y.

Fragmento de código:

```
if (Math.Abs(m) <= 1)
{
    xk += (dx >= 0) ? 1 : -1;
    yk += (dx >= 0) ? m : -m;
}
else
{
    yk += (dy >= 0) ? 1 : -1;
    xk += (dy >= 0) ? 1/m : -1/m;
}
```

4.2 Algoritmo de Bresenham para Líneas

El algoritmo de Bresenham es uno de los más eficientes para el trazado de líneas rectas en pantallas digitales. Fue diseñado específicamente para trabajar exclusivamente con operaciones enteras, lo que lo hace más rápido y preciso que algoritmos como el DDA. Su principal ventaja es que evita el uso de decimales o flotantes, eliminando así errores de redondeo acumulativos.

Lógica implementada:

Primero se calcula la diferencia horizontal (dx) y vertical (dy) entre los dos extremos de la línea. Además, se determina si el desplazamiento debe ser hacia la derecha o izquierda (según

el signo de dx), y hacia arriba o abajo (según el signo de dy). Esto permite que el algoritmo pueda dibujar líneas en cualquier dirección o cuadrante del plano.

Se evalúa si la línea es más horizontal cuando dx es mayor que dy o más vertical cuando dy es menor que dx. Esta decisión es importante porque define cuál será el eje principal de iteración, es decir, si se recorrerán los píxeles avanzando en x o en y.

El algoritmo se maneja con el parámetro de decisión (p).
Si ($|dx| > |dy|$): $p_0 = 2 * dy - dx$, pero si ($|dy| \geq |dx|$): $p_0 = 2 * dx - dy$

Por ejemplo, si se está avanzando en x, el valor de p determinará si y debe cambiar en ese paso o no. Este parámetro se actualiza en cada iteración con operaciones enteras, y gracias a su diseño eficiente, no requiere multiplicaciones complejas ni divisiones.

El algoritmo comienza desde el punto inicial y avanza paso a paso en el eje dominante. En cada paso:

- Se dibuja un píxel en la ubicación actual.
- Se evalúa el valor del parámetro p para decidir si se actualiza también la otra coordenada (x o y).
- Se actualiza el valor de p para su próxima evaluación hasta completar con los pasos.

En cada paso, además de dibujar el píxel correspondiente en el canvas, se actualiza la tabla que muestra los valores actuales de x, y, p y el número de paso. Esta tabla permite seguir paso a paso cómo evoluciona el algoritmo, ideal para propósitos didácticos y de depuración.

Fragmento del código:

```
for (int i = 0; i <= steps; i++)
{
    DrawPixel(centerX + px, centerY - py);
    dgv?.Rows.Add(step, px, py, p);
    AnimationPause();

    if (isXMajor)
    {
        px += sx;
        if (p < 0) p += 2 * absDy;
        else { py += sy; p += 2 * (absDy - absDx); }
    }
    else
    {
        py += sy;
        if (p < 0) p += 2 * absDx;
        else { px += sx; p += 2 * (absDx - absDy); }
    }
    step++;
}
```

4.3 Algoritmo de Bresenham para Círculos

Este algoritmo permite dibujar círculos aprovechando su simetría en 8 octantes. A diferencia de otros métodos que requieren funciones trigonométricas o raíces cuadradas, el algoritmo de Bresenham utiliza solo operaciones con enteros (sumas, restas y multiplicaciones por 2) y aprovecha la simetría del círculo para reducir el trabajo.

Lógica implementada:

Se comienza desde el punto más alto del círculo, con coordenadas: x = 0, y = r (el radio)

Se define un parámetro de decisión p que permite decidir en cada paso si el siguiente punto debe mantener la coordenada y o si esta debe disminuir, este parámetro se basa en una función de error respecto al círculo ideal definido por la ecuación: $x^2 + y^2 = r^2$

A partir del punto (0, r), se avanza hacia la derecha en x (incrementando x uno a uno), y en cada paso se evalúa el parámetro de decisión p para decidir si se debe disminuir y o mantenerla:

- Si $p < 0$, eso significa que el punto actual está dentro del círculo ideal, por lo que no es necesario ajustar y, entonces se incrementa x y se actualiza p sumando una cantidad proporcional al nuevo x.

- Si $p \geq 0$, eso significa que el punto actual está fuera del círculo, por lo que se necesita ajustar la vertical, entonces se incrementa x, se decrementa y para acercarse al borde ideal del círculo y se actualiza p con un nuevo valor que refleja este doble cambio.

Este proceso se repite hasta que x = y, es decir, hasta que se alcanza la diagonal del primer octante

Por cada punto (x, y) calculado en ese primer octante, se generan los otros 7 puntos simétricos

Simetría	Coordenadas generadas
Original	(xc + x, yc + y)
Reflejo en eje X	(xc + x, yc - y)
Reflejo en eje Y	(xc - x, yc + y)
Reflejo en origen	(xc - x, yc - y)

Reflejo en la diagonal y = x	(xc + y, yc + x)
Reflejo en y = -x	(xc + y, yc - x)
Reflejo en -y = x	(xc - y, yc + x)
Reflejo en -y = -x	(xc - y, yc - x)

Teniendo en cuenta al punto (xc, yc) como el centro del círculo, cada uno de esos puntos está a la misma distancia del centro.

El algoritmo continúa generando nuevos puntos con la lógica del parámetro p, y por cada punto calcula sus reflejos para completar el contorno circular sin dejar huecos ni irregularidades.

Fragmento del código:

```
while (x < y) {
    x++;
    p += (p < 0) ? 2 * x + 1 : 2 * (x - y--) + 1;
    AnimationPause();
    DrawSymmetricPointsAnimated(x, y, centerX, centerY, dgv, step++, p);
}
```

```
var points = new List<Point> {
    new Point(cx + x, cy + y),
    new Point(cx - x, cy + y),
    new Point(cx + x, cy - y),
    new Point(cx - x, cy - y),
    new Point(cx + y, cy + x),
    new Point(cx - y, cy + x),
    new Point(cx + y, cy - x),
    new Point(cx - y, cy - x)
};
```

4.4 Algoritmo Flood Fill (Relleno por difusión)

El algoritmo Flood Fill o algoritmo de relleno por difusión consiste en rellenar toda una región conectada de píxeles que comparten un mismo color, a partir de un punto inicial, con un nuevo color que es el color de relleno.

Lógica implementada:

Se selecciona un punto inicial dentro de la figura al capturar el evento clic del usuario sobre el polígono. Este punto tiene un color, llamado color objetivo.

El algoritmo verifica si el píxel actual tiene el color objetivo (es decir, que aún no ha sido pintado ni es parte del borde), si el color coincide, se cambia por el nuevo color de relleno.

Desde ese punto, se hace una llamada recursiva a sus cuatro vecinos:

- arriba (x, y - 1)
- abajo (x, y + 1)
- izquierda (x - 1, y)
- derecha (x + 1, y)

Este proceso se repite desde cada nuevo punto que también tenga el color objetivo, lo que provoca una "difusión" del color hacia toda la región conectada.

Se usa una función que se llama a sí misma para procesar los vecinos. Esto hace que el algoritmo se expanda como una mancha, llenando todos los espacios conectados.

Para evitar errores como desbordamiento de pila o que se pinte fuera del área válida, se incluye una validación que detiene la recursión si:

- El punto está fuera de los límites de la imagen.
- El color actual no coincide con el color objetivo.
- El punto ya fue pintado.

Como la operación de pintar es ejecutada en un hilo paralelo usando Task.Run, se debe usar Invoke para actualizar el control PictureBox de forma segura desde ese otro hilo. Esto permite mostrar visualmente el avance del relleno sin bloquear la interfaz.

Fragmento del código:

```
private void FloodFillRecursive(int x, int y, Color targetColor, PictureBox picCanvas, DataGridView dgv)
{
    if (x < 0 || y < 0 || x >= _canvas.Width || y >= _canvas.Height)
        return;

    if (!ColorHelper.ColorsMatch(_canvas.GetPixel(x, y), targetColor) ||
        ColorHelper.ColorsMatch(_canvas.GetPixel(x, y), _fillColor))
        return;

    _canvas.SetPixel(x, y, _fillColor);

    picCanvas.Invoke((MethodInvoker)(() =>
    {
        picCanvas.Image = _canvas;
        dgv?.Rows.Add(_ordinal++, x, y);
    }));

    Thread.Sleep(AnimationDelay);

    FloodFillRecursive(x, y - 1, targetColor, picCanvas, dgv);
    FloodFillRecursive(x + 1, y, targetColor, picCanvas, dgv);
    FloodFillRecursive(x, y + 1, targetColor, picCanvas, dgv);
    FloodFillRecursive(x - 1, y, targetColor, picCanvas, dgv);
}
```

4.4. Generación del polígono

El método genera los vértices de un polígono regular centrado en un punto específico. Recibe como parámetros el centro del polígono, el radio de la circunferencia circunscrita y el número de lados.

Calcula los puntos usando trigonometría básica: para cada vértice, determina su posición mediante funciones seno y coseno. Cada ángulo se incrementa en pasos de 2π dividido por el número de lados, luego se almacenan en una lista que finalmente se devuelve.

Fragmento de Código:

```
private static List<PointF> GenerateRegularPolygon(Point center, int radius, int sides)
{
    var points = new List<PointF>();
    double angleStep = 2 * Math.PI / sides;

    for (int i = 0; i < sides; i++)
    {
        double angle = i * angleStep;
        float x = center.X + (float)(radius * Math.Cos(angle));
        float y = center.Y + (float)(radius * Math.Sin(angle));
        points.Add(new PointF(x, y));
    }

    return points;
}
```

4.5. Generación de la tabla

El método prepara un control DataGridView para mostrar datos de puntos. Primero limpia cualquier columna existente y luego agrega las columnas nuevas: "Pasos", "X" e "Y", configurando sus encabezados y anchos específicos. Esta función sirve como inicialización básica para mostrar datos tabulares relacionados con coordenadas y pasos de ejecución de algoritmos.

Fragmento de Código:

```
public static void InicializarDataGridView(DataGridView dataGridViewPuntos)
{
    dataGridViewPuntos.Columns.Clear();
    dataGridViewPuntos.Columns.Add("Pasos", "Paso");
    dataGridViewPuntos.Columns.Add("X", "X");
    dataGridViewPuntos.Columns.Add("Y", "Y");
    dataGridViewPuntos.Columns["Pasos"].Width = 85;
    dataGridViewPuntos.Columns["X"].Width = 70;
    dataGridViewPuntos.Columns["Y"].Width = 70;

    ConfigurarPropiedadesBasicas(dataGridViewPuntos);
}
```

5. Interfaz de Usuario

Cada formulario está dividida en las siguientes secciones:

Área de dibujo (PictureBox): Es el lienzo donde se visualizan los resultados de los algoritmos. Su centro representa el origen del sistema de coordenadas cartesianas (0, 0), con ejes X y Y dibujados para referencia.

Controles de entrada: TextBox para ingresar la o las entradas numéricas

Botones de acción:

- **Dibujar**: Ejecuta el algoritmo seleccionado y muestra la animación del trazado.
- **Limpiar**: Borra el PictureBox y limpia los controles y la tabla de pasos.
- **Rellenar**: Aplica el algoritmo de Flood Fill una vez se ha dibujado una figura cerrada.

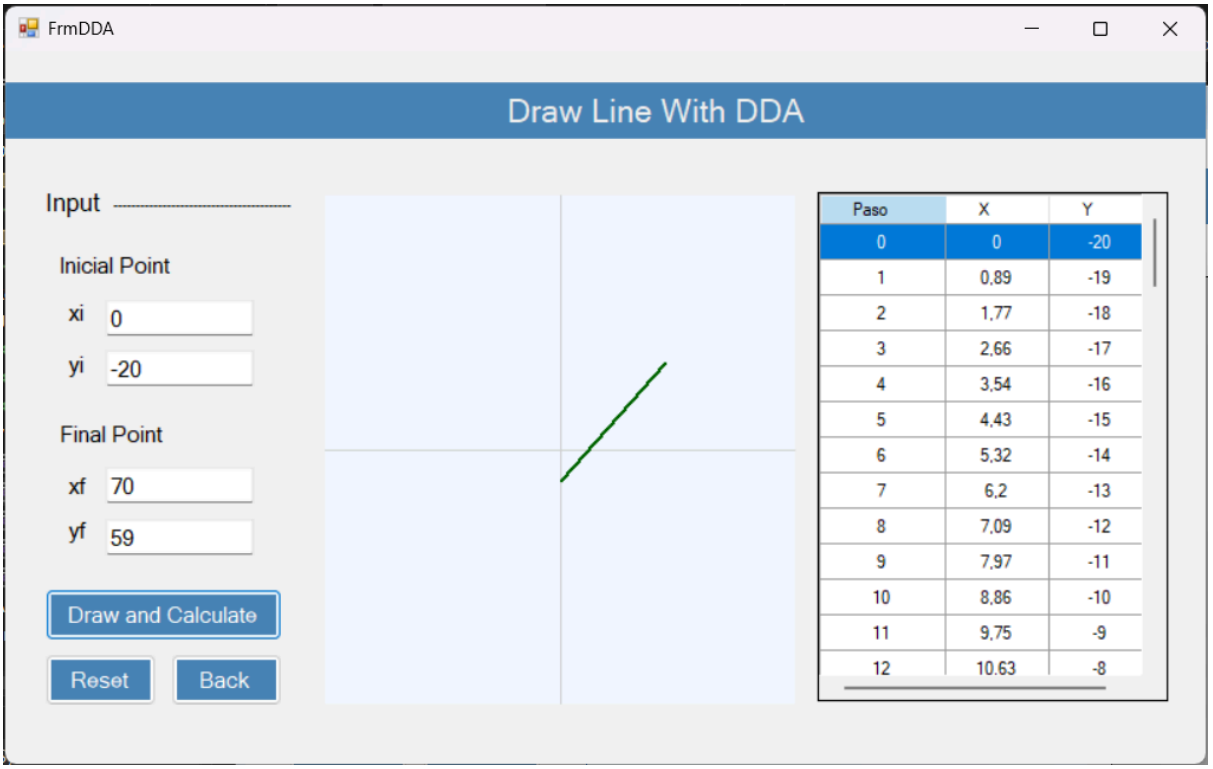
Tabla de pasos (DataGridView): Muestra los valores calculados en cada iteración del algoritmo.

- Para la creación de líneas con DDA y Bresenham: paso, x, y.
- Para el círculo usando Bresenham: paso, x, y, parámetro de decisión p.
- Para Flood Fill: paso, x, y de cada píxel rellenado.

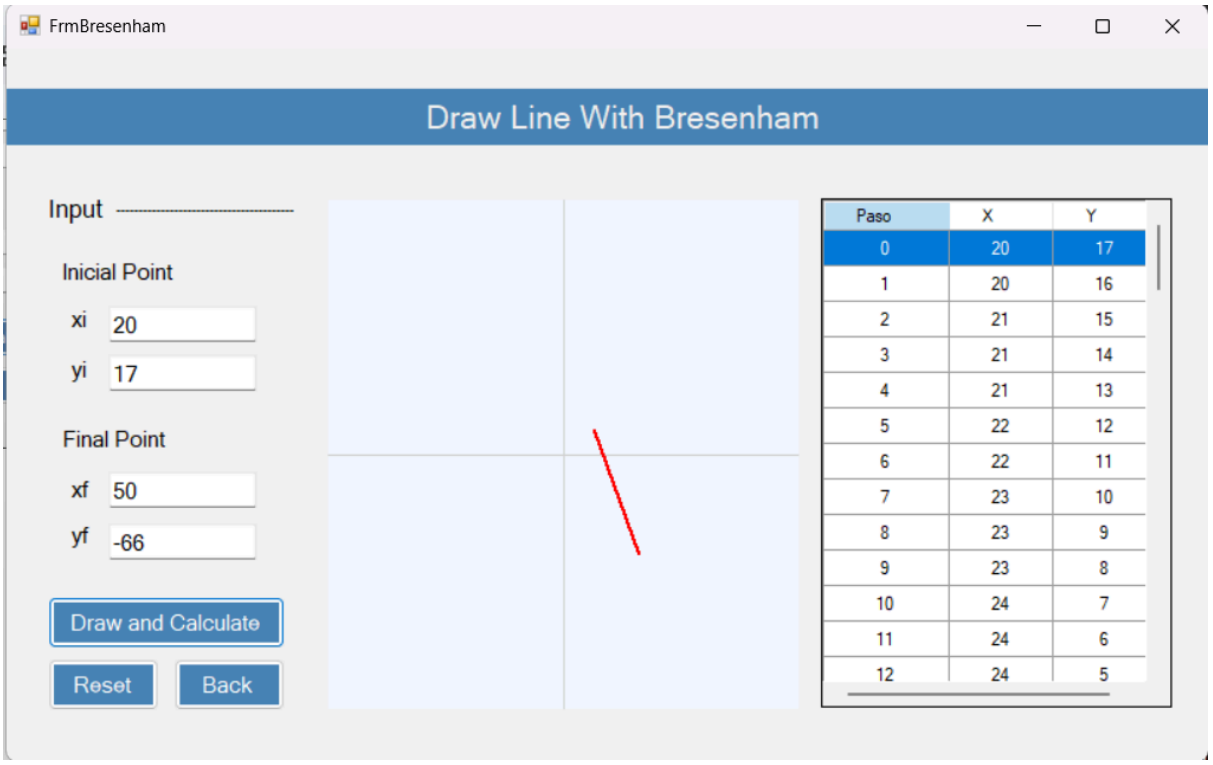
5.1. Formularios funcionales:

FrmMain:

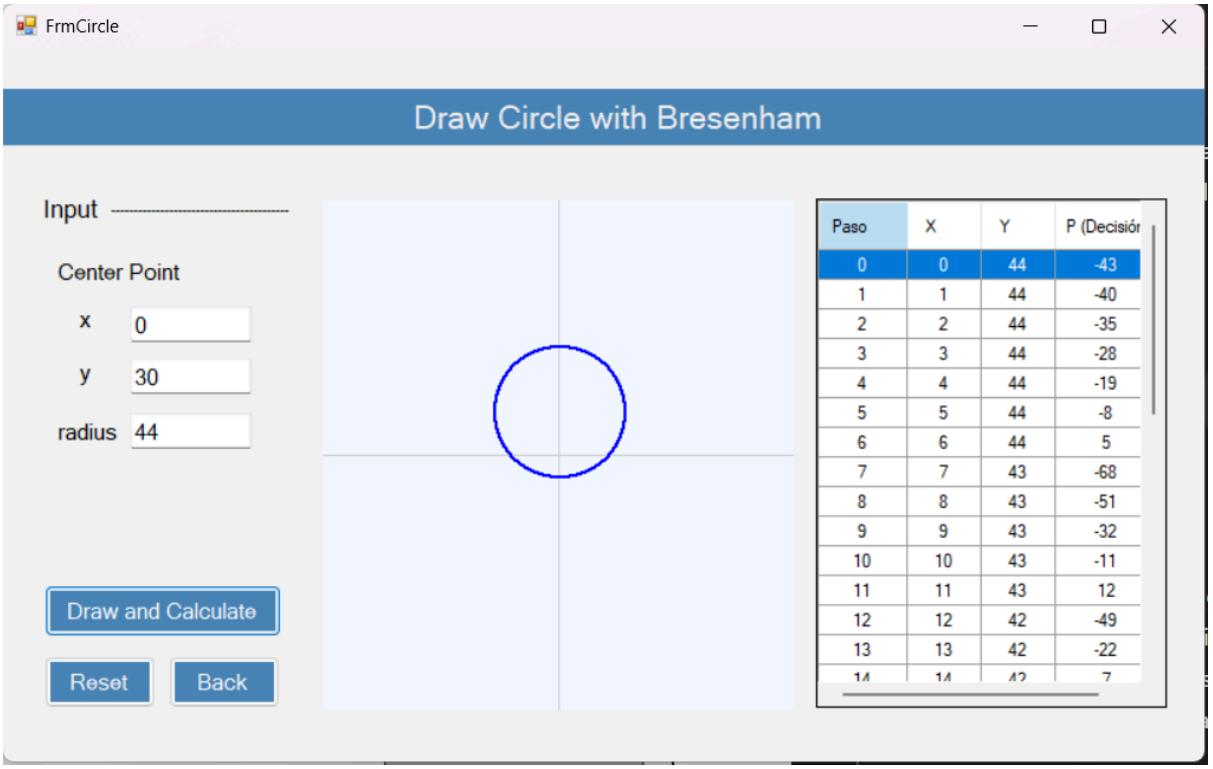
FrmDDA:



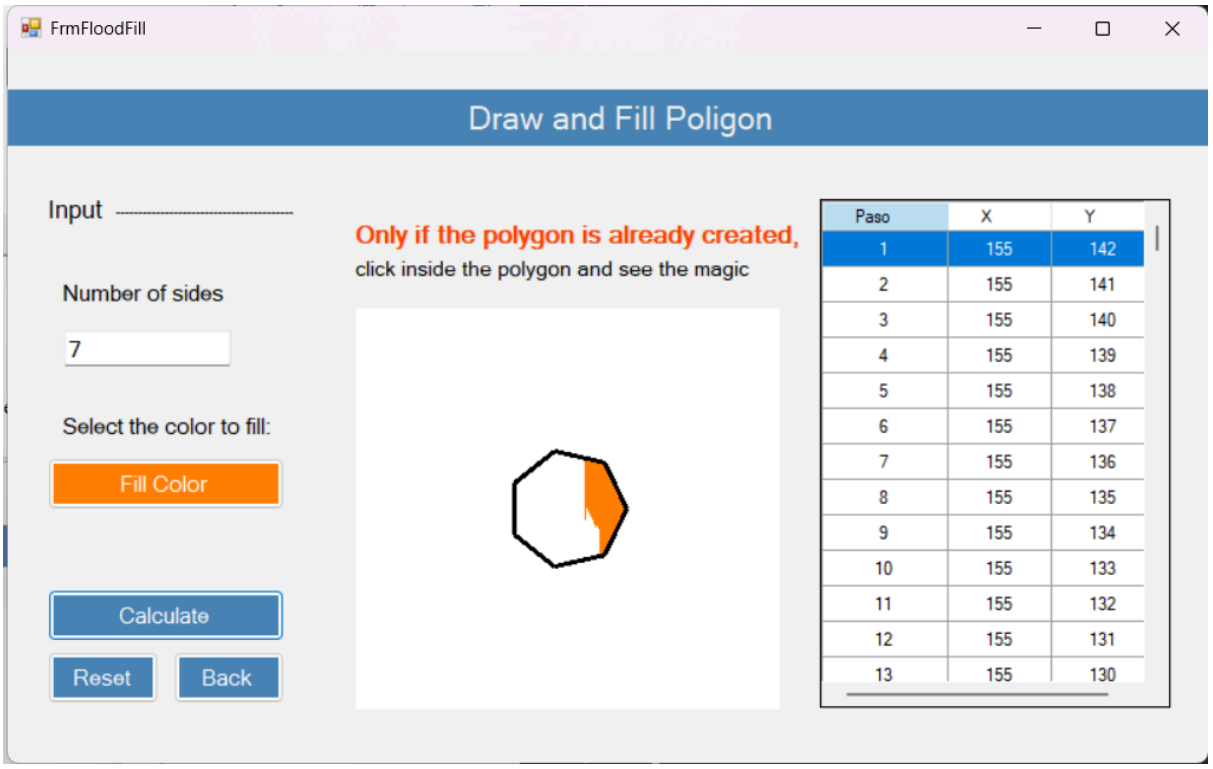
FrmBresenham:



FrmCicle:

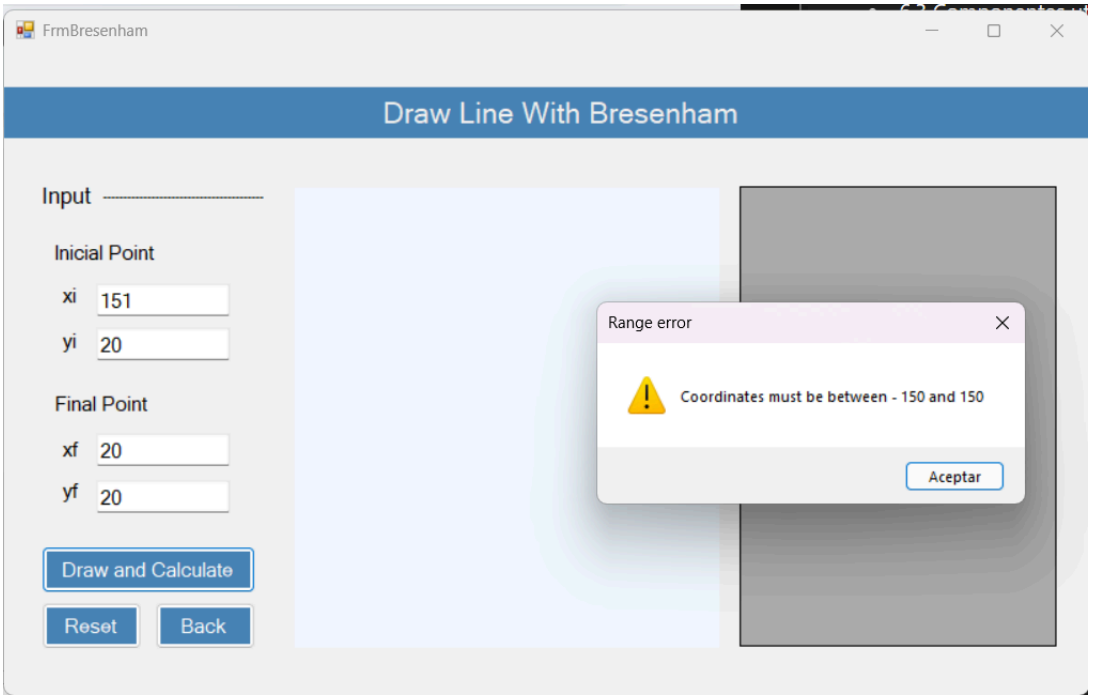


FrmFloodFill:

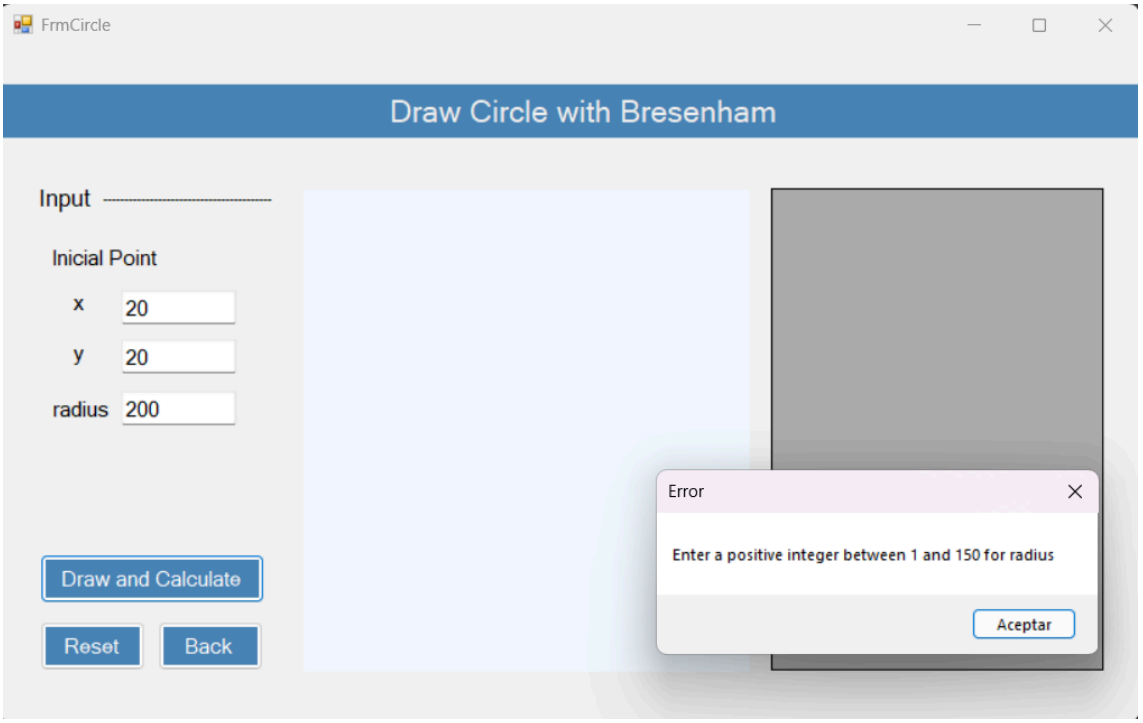


6. Pruebas y Validaciones

Se validó que el ingreso de los puntos para los formularios de DDA y Bresenham para la creación de líneas sea entre [150, 150], con el fin de que no salgan de los ejes.



Dentro del formulario del Dibujo del círculo se validó que el radio no sea negativo y tampoco sea mayor de 150, además que el punto central debe estar entre [100, 100]



7. Conclusiones

La implementación de los algoritmos DDA, Bresenham (línea y círculo) y Flood Fill permitió profundizar en la lógica detrás del trazado de figuras primitivas en un entorno gráfico bidimensional, así como en la eficiencia de los métodos basados en aritmética entera versus flotante.

El uso de animación paso a paso y una tabla de valores por iteración facilita la comprensión del comportamiento interno de cada algoritmo, cumpliendo con un enfoque pedagógico orientado a estudiantes o desarrolladores que deseen aprender computación gráfica de bajo nivel.

Se evidenció que algoritmos como el de Bresenham son más eficientes en términos computacionales frente a DDA, al evitar cálculos en punto flotante. Además, el uso de simetrías en el algoritmo del círculo reduce significativamente el número de operaciones requeridas.

8. Recomendaciones

1. El algoritmo Flood Fill recursivo puede llevar al desbordamiento de pila en regiones grandes. Se sugiere implementar una versión iterativa con estructura de pila o cola

(Stack/Queue) para mejorar la eficiencia y evitar errores.

2. Incluir funciones para guardar las imágenes generadas o exportar las tablas de pasos a archivos CSV o PDF, permitiría documentar y compartir resultados fácilmente.

3. Implementar botones de control (Siguiente, Anterior, Pausar) para permitir un control manual del paso a paso ayudaría al usuario a enfocarse en puntos clave de la ejecución de los algoritmos.

9. Anexos

Link de GitHub: https://github.com/luis-sagx/ComputacionGrafica/tree/main/Ejercicios2P