**UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE**

**DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN**

**COMPUTACIÓN GRÁFICA**



**Hecho por:**

Mateo Medranda

**NRC:**

23299

**Fecha:**

2025-05-09

# **Introducción**

El presente informe tiene como objetivo explicar e implementar algunos algoritmos de dibujo y relleno, entre los cuales se encuentran:

* Algoritmo de discretización de líneas DDA (Digital Differential Analyzer)
* Algoritmo de Bresenham para líneas
* Algoritmo de Bresenham para circunferencia (simetría de 8 lados)
* Algoritmo FloodFill para relleno por inundación (4 direcciones)

Estos algoritmos son de gran importancia en el mundo de la programación, ya que son la base principal del funcionamiento de las interfaces gráficas y el análisis computacional sobre decidir que pixeles pintar de la pantalla.

Los algoritmos son implementados en el lenguaje de programación C# mediante la herramienta de Visual Studio para su implementación.

Link al repositorio: <https://github.com/MateoMedranda/Medranda_Deber_C-lculo-de-rea-y-Per-metro-de-Figuras-Geom-tricas-con-Interfaz-Gr-fica-en-C-.git>

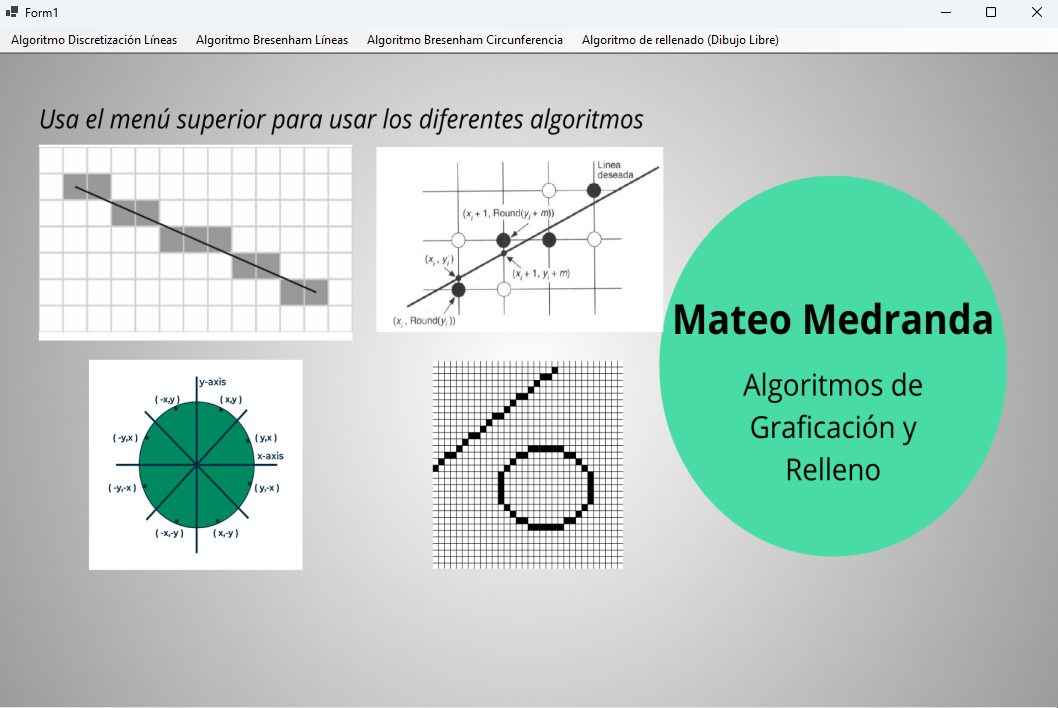


Ilustración . Formulario padre, con presentación a los diferentes algoritmos

# **Lógica De Programación**

Dentro de la lógica de la programación, se uso el paradigma orientado a objetos, para poder crear clases reutilizables con el fin de tener un código limpio y bien organizado.

Siguiendo lo anterior, el código se encuentra bien definido por funciones que siguen el principio de responsabilidad única basado en Clean Code de Robert C. Martin.

A continuación, se detalla la forma matemática de los algoritmos.

* Algoritmo DDA

El algoritmo DDA se basa en el uso de la pendiente para definir que fórmula usar e incrementar el valor en 1 de la variable que no esta involucrada en la fórmula usada.

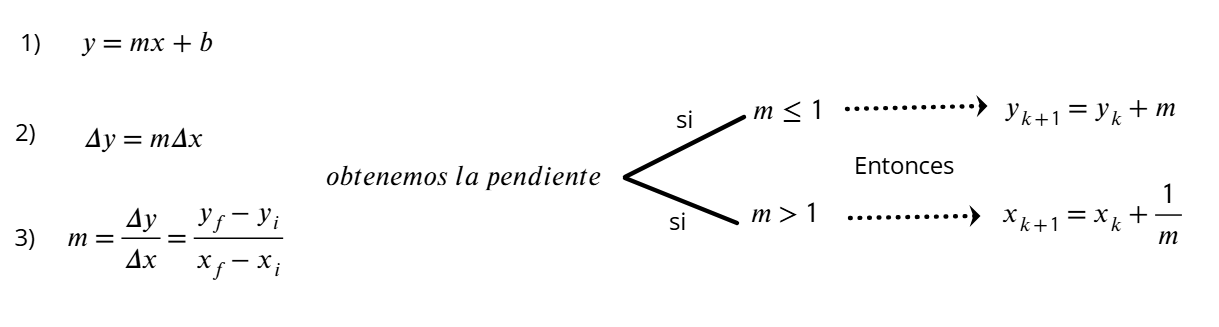


Ilustración . Pasos para el desarrollo DDA y cálculo de puntos

En la ilustración 2 se aprecia los pasos necesarios para realizar los cálculos pertinentes, estos son:

1. Calcular con el punto inicial y el punto final
2. Calcular la pendiente m usando los deltas calculados
3. Verificar que condición se cumple
4. Aplicar la fórmula según la condición e incrementar la variable opuesta

En el caso de que la pendiente sea menor o igual que uno, entonces la variable a incrementar en una unidad será x, mientras que cuando la pendiente es mayor que uno, entonces se incrementará y en una unidad.

De esta forma se consigue una serie de puntos que forman una recta

*Nota: El algoritmo DDA fue diseñado para usarse en un plano simulando el primer cuadrante, es decir con x y positivos, por lo que el algoritmo no contempla pendientes negativas, lo que puede causar problemas.*

* Algoritmo Bresenham Líneas

El algoritmo de Bresenham para líneas se basa en el algoritmo DDA, pero incrementando un parámetro para mayor precisión, siendo este el valor de (p).

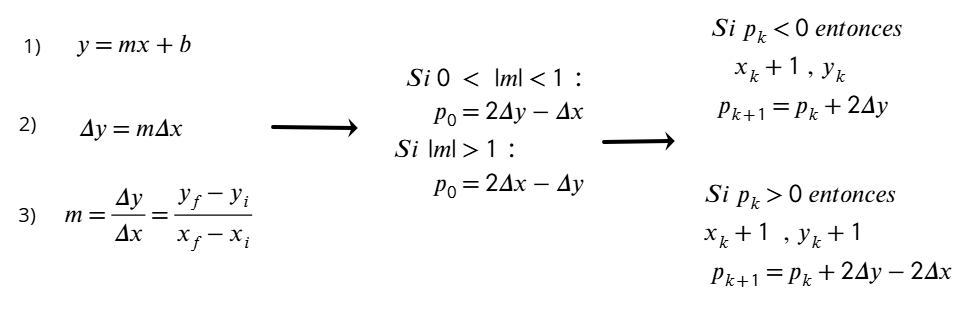


Ilustración . Pasos para el desarrollo con Bresenham y el cálculo de puntos

En la ilustración 3 se reconoce los diferentes pasos y validaciones a seguir, los cuales son:

1. Calcular con el punto inicial y el punto final
2. Calcular la pendiente m usando los deltas calculados
3. Verificar que condición se cumple con la pendiente y aplicar la fórmula para p0 correspondiente
4. Al encontrar p0 verificar la condición que se cumple
5. Aplicar la fórmula de forma iterativa y validar la condición para cada valor de p calculado

Dentro de este algoritmo se contempla un parámetro p, que permitirá intercambiar entre ecuaciones y variables dependiendo del valor calculado, de este modo es más preciso al momento de graficar.

*Nota: El algoritmo de Bresenham fue diseñado para valores positivos de la pendiente, por lo que un valor negativo no esta contemplado dentro del algoritmo, además para poder funcionar en pendientes mayores a 1 hay que hacer un ajuste e intercambiar las variables que incrementan.*

* Algoritmo Bresenham Circunferencia

El algoritmo de Bresenham para la circunferencia suele tener un problema por si solo, y es la aparición de huecos en la figura, por lo que es necesario aplicar el principio de simetría de 8 lados que permitirá cubrir los huecos de forma correcta y graficar una circunferencia de forma acertada.

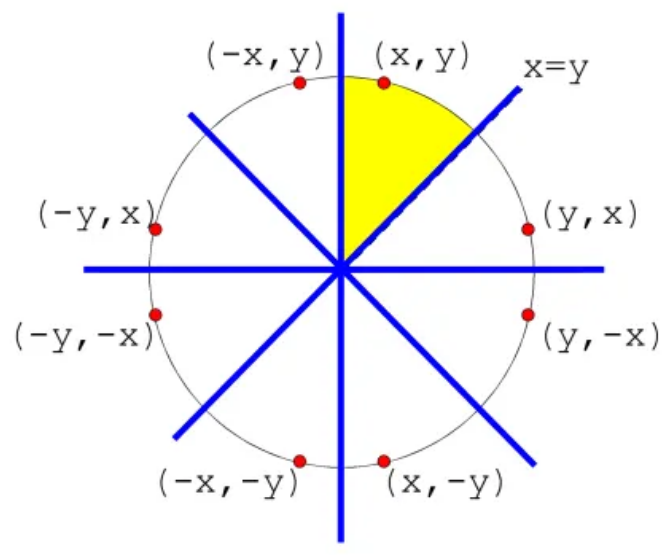


Ilustración . Fígura representativa de la Simetría de los 8 lados

Lo que se busca es principalmente aplicar una función que encuentre los 8 puntos correspondientes e ir dibujando la circunferencia en estos 8 segmentos de curva de modo que se vayan llenando los huecos, mediante la ilustración 4 se observa las coordenadas necesarias para realizar el çalculo.

* Algoritmo de rellenado FloodFill

El algoritmo para rellenar figuras, se basa en el coloreo de cada pixel dentro de la misma, y para conseguir este efecto es necesario usar un proceso recursivo que permita explorar las 4 direcciones de cada pixel y de esa forma encontrar un camino sin llenar y completarlo con puntos del color deseado.

Ahora es de suma importancia destacar que este algoritmo funciona de forma adecuada para figuras pequeñas, ya que, al ser un algoritmo recursivo, el hecho de tener un espacio de gran tamaño para rellenar, causa que las recursivas apiladas se desborden y el programa de un error de desbordamiento, pero no se trata de un error de código, sino de un desbordamiento en la pila recursiva al tener tantos pixeles.

Para poder solucionar este problema, pero también para poder observar el comportamiento y el patrón de rellenado, se optó por el uso de 2 algoritmos, el uno es por inundación recursivo en 4 direcciones, y el otro es con un enfoque iterativo.

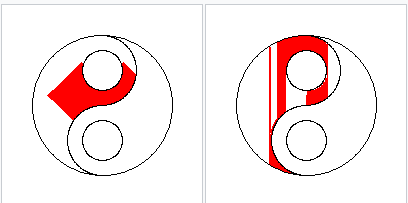


Ilustración . Diferencia en el patrón de rellenado por inundación

Gracias a la ilustración 5 se observa la diferencia entre un patrón siguiendo un enfoque iterativo (imagen de la izquierda) y un patrón siguiendo un enfoque recursivo (imagen de la derecha).

Al observar de forma animada, en el iterativo se trata de una expansión de píxeles siguiendo una cola o pila, mientras que con el recursivo se usa una lista y suelen quedar huecos en blanco que cuando la recursividad retorne, estos se llenarán.

# **Interfaces**

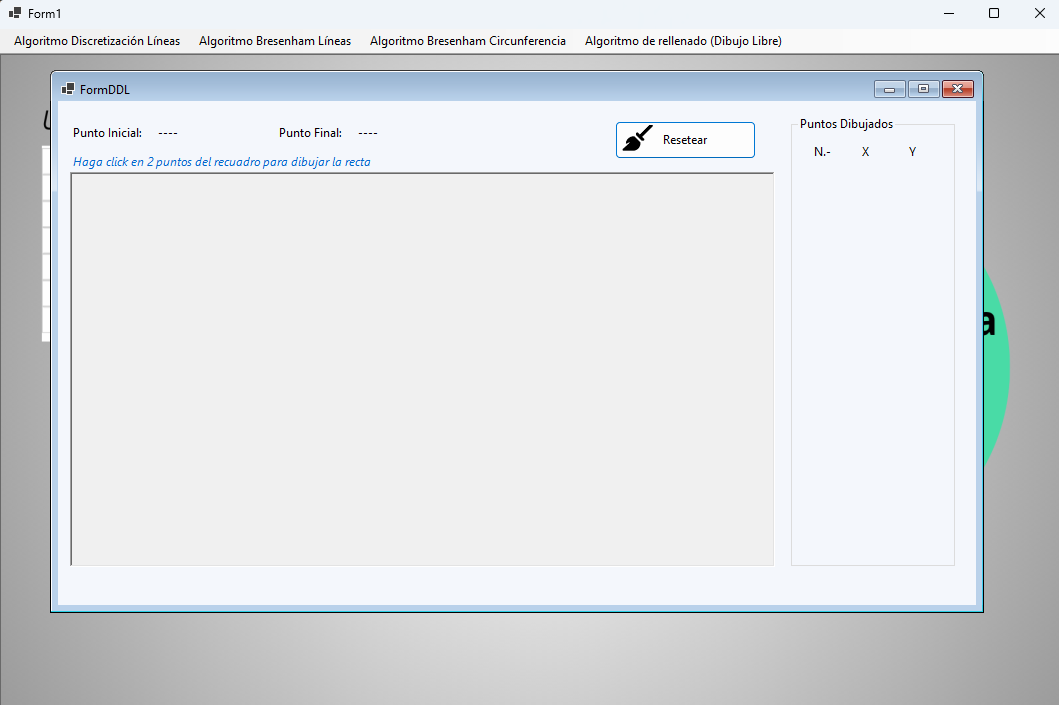


Ilustración . Interfaz para la implementación del algoritmo DDA

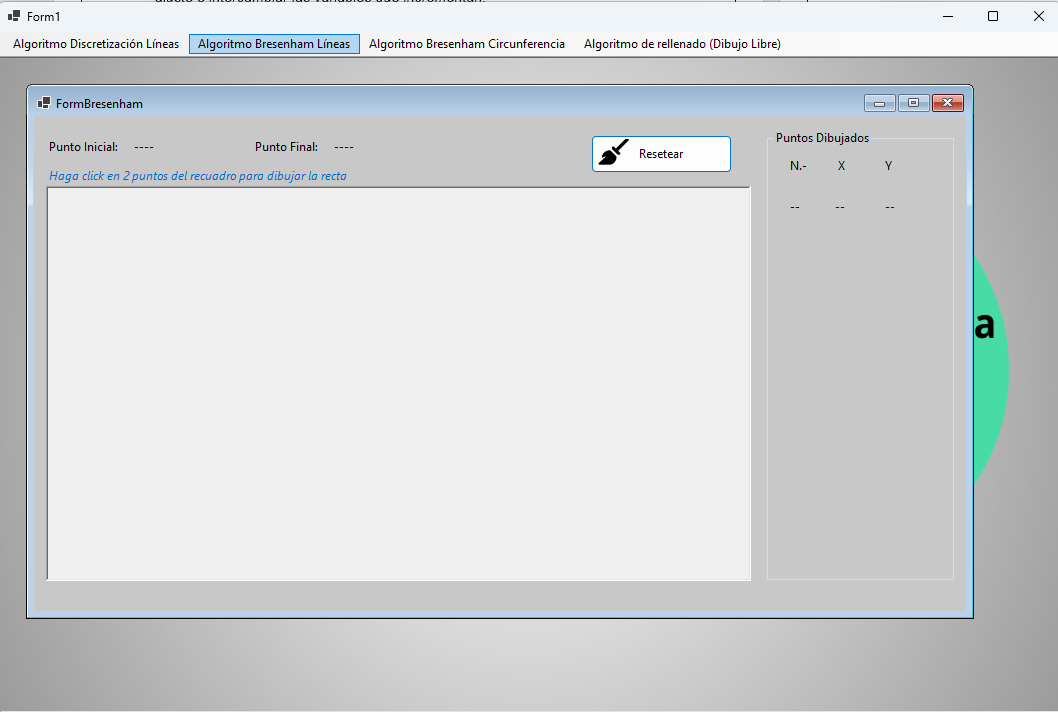


Ilustración . Interfaz para la implementación del algoritmo de Bresenham en líneas

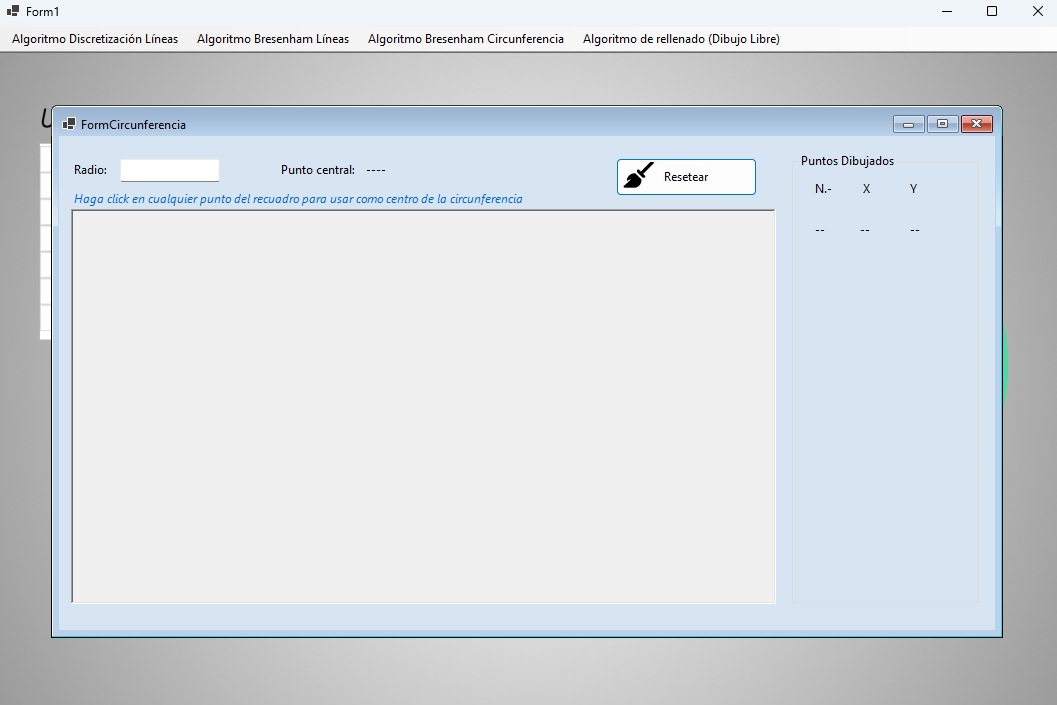


Ilustración . Interfaz para la implementación del algoritmo de Bresenham con simetría de 8 lados

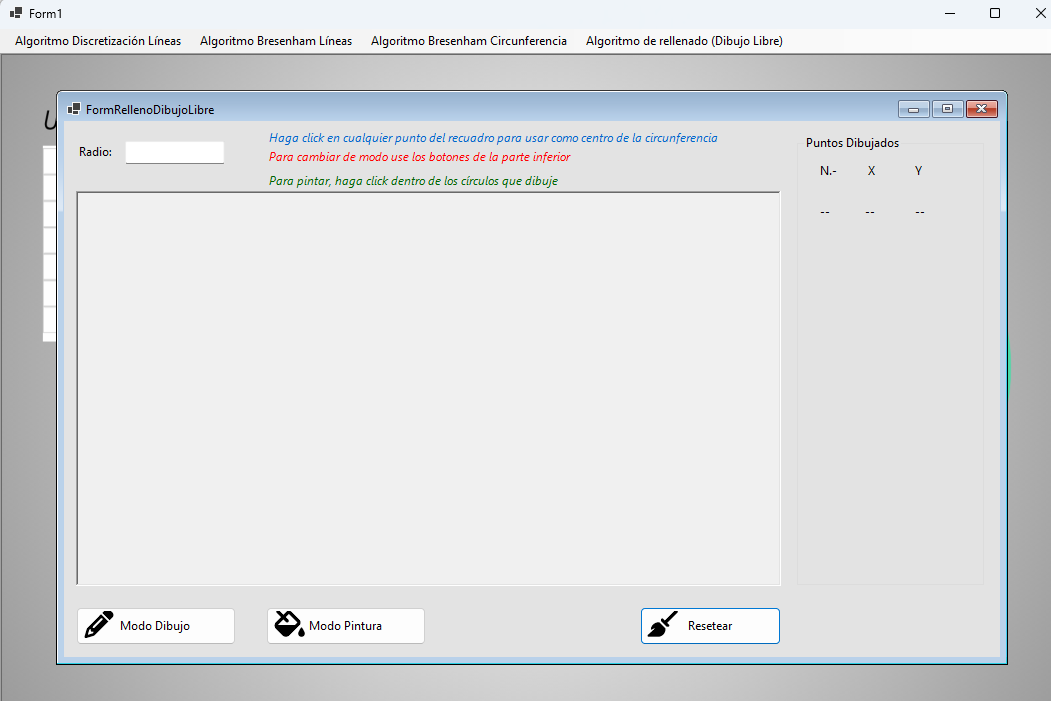


Ilustración . Interfaz para la implementación del algoritmo de relleno y dibujo libre

# **Clases**

Dentro del sistema se han enumerado las clases para poder crear objetos y usar los métodos correspondientes de forma útil y que sean reusables.

* Algoritmo DDA

class DDA

{

private Point Pi;

private Point Pf;

private int deltaX;

private int deltaY;

private int k;

private float m;

private float xIncrement;

private float yIncrement;

private Graphics mGraph;

private Pen mPen;

private int cont;

private string contString = "";

private string XString = "";

private string YString = "";

private Label NLabel;

private Label XLabel;

private Label YLabel;

public DDA()

{

Pi = new Point(0, 0);

Pf = new Point(0, 0);

deltaX = 0;

deltaY = 0;

cont = 0;

}

public void ReadData(Point point1, Point point2, PictureBox picCanvas, Label N, Label X, Label Y)

{

Pi = point1;

Pf = point2;

mGraph = picCanvas.CreateGraphics();

cont = 0;

NLabel = N;

XLabel = X;

YLabel = Y;

}

public void CalculateConstant()

{

deltaX = Pf.X - Pi.X;

deltaY = Pf.Y - Pi.Y;

}

public void CalculatePendient()

{

m = (float)deltaY / (float)deltaX;

}

public void CalculateKValor()

{

if (Math.Abs(deltaX) > Math.Abs(deltaY))

{

k = Math.Abs(deltaX);

}

else

{

k = Math.Abs(deltaY);

}

}

public void CalculatePoints()

{

float xk = Pi.X;

float yk = Pi.Y;

plotInitialPoints();

for (int i = 0; i < k; i++)

{

plotPoint(new Point((int)Math.Round(xk), (int)Math.Round(yk)));

CreateStringPoints((int)Math.Round(xk), (int)Math.Round(yk));

if (m <= 1)

{

yk =( yk + m);

xk++;

}else if (m > 1)

{

xk = (xk + 1/m);

yk++;

}

}

}

public void plotPoint(Point point)

{

mPen = new Pen(Color.Green, 2);

mGraph.DrawRectangle(mPen, point.X, -point.Y, 1, 1);

}

public void plotInitialPoints()

{

mPen = new Pen(Color.Red, 2);

mGraph.DrawRectangle(mPen, Pf.X, -Pf.Y, 6, 6);

mGraph.DrawRectangle(mPen, Pi.X, -Pi.Y, 6, 6);

}

private void CreateStringPoints(int x, int y)

{

cont++;

contString += cont + "\n";

XString += x + "\n";

YString += y + "\n";

NLabel.Text = contString;

XLabel.Text = XString;

YLabel.Text = YString;

}

}

* Algoritmo de Bresenham

class Bresenham

{

private Point Pi;

private Point Pf;

private int deltaX;

private int deltaY;

private int doubleDeltaX;

private int doubleDeltaY;

private float m;

private int k;

private int p;

private int cont;

private Graphics mGraph;

private Pen mPen;

private string contString = "";

private string XString = "";

private string YString = "";

private Label NLabel;

private Label XLabel;

private Label YLabel;

public Bresenham()

{

Pi = new Point(0, 0);

Pf = new Point(0, 0);

deltaX = 0;

deltaY = 0;

doubleDeltaX = 0;

doubleDeltaY = 0;

cont = 0;

}

public void ReadData(Point point1, Point point2, PictureBox picCanvas, Label N, Label X, Label Y)

{

Pi = point1;

Pf = point2;

mGraph = picCanvas.CreateGraphics();

cont = 0;

NLabel = N;

XLabel = X;

YLabel = Y;

}

public void CalculateConstant()

{

deltaX = Pf.X - Pi.X;

deltaY = Pf.Y - Pi.Y;

doubleDeltaX = 2 \* deltaX;

doubleDeltaY = 2 \* deltaY;

}

public void CalculatePendient()

{

m = (float)deltaY / (float)deltaX;

}

public void CalculateKValor()

{

if (Math.Abs(deltaX) > Math.Abs(deltaY))

{

k = Math.Abs(deltaX);

}

else

{

k = Math.Abs(deltaY);

}

}

public void CalculatePValor()

{

if (Math.Abs(m) > 0 && Math.Abs(m) < 1)

{

p = doubleDeltaY - deltaX;

}

else if (Math.Abs(m) > 1)

{

p = doubleDeltaX - deltaY;

}

}

public void CalculatePoints()

{

plotInitialPoints();

int xk = Pi.X;

int yk = Pi.Y;

int pk = p;

for (int i = 0; i <= k; i++)

{

plotPoint(new Point(xk, yk));

CreateStringPoints(xk, yk);

if (Math.Abs(m) > 0 && Math.Abs(m) < 1)

{

if (pk < 0)

{

pk = pk + 2 \* deltaY;

}

else if (pk > 0)

{

pk = pk + doubleDeltaY - doubleDeltaX;

yk++;

}

xk++;

}

else if (Math.Abs(m) > 1)

{

if (pk < 0)

{

pk = pk + 2 \* deltaX;

}

else if (pk > 0)

{

pk = pk + doubleDeltaX - doubleDeltaY;

xk++;

}

yk++;

}

}

}

public void plotPoint(Point point)

{

mPen = new Pen(Color.Blue, 2);

mGraph.DrawRectangle(mPen, point.X, -point.Y, 1, 1);

}

public void plotInitialPoints()

{

mPen = new Pen(Color.Red, 2);

mGraph.DrawRectangle(mPen, Pf.X, -Pf.Y, 6, 6);

mGraph.DrawRectangle(mPen, Pi.X, -Pi.Y, 6, 6);

}

private void CreateStringPoints(int x, int y)

{

cont++;

contString += cont + "\n";

XString += x + "\n";

YString += y + "\n";

NLabel.Text = contString;

XLabel.Text = XString;

YLabel.Text = YString;

}

}

* Algoritmo Bresenham Circunferencia

class Circunferencia

{

private PictureBox picCanvas;

private int xc;

private int yc;

private int r;

private int cont = 0;

private string contString = "";

private string XString = "";

private string YString = "";

private HashSet<Point> visited = new HashSet<Point>();

private Label NLabel;

private Label XLabel;

private Label YLabel;

public Circunferencia()

{

xc = 0;

yc = 0;

r = 0;

}

public void readData(Point center, int radio, PictureBox Canvas, Label N, Label X, Label Y )

{

xc = center.X; yc = center.Y; r = radio;

picCanvas = Canvas;

NLabel = N;

XLabel = X;

YLabel = Y;

cont = 0;

contString = "";

XString = "";

YString = "";

}

public void DrawCircleMidPoint()

{

int x = 0;

int y = r;

int p = 1 - r;

PlotCirclePoints(x, y);

while (x < y)

{

x++;

if (p < 0)

{

p += 2 \* x + 1;

}

else

{

y--;

p += 2 \* (x - y) + 1;

}

PlotCirclePoints(x, y);

}

}

private void PlotCirclePoints(int x, int y)

{

plotPoint(xc + x, yc + y);

plotPoint(xc - x, yc + y);

plotPoint(xc + x, yc - y);

plotPoint(xc - x, yc - y);

plotPoint(xc + y, yc + x);

plotPoint(xc - y, yc + x);

plotPoint(xc + y, yc - x);

plotPoint(xc - y, yc - x);

}

private void plotPoint(int x, int y)

{

visited.Add(new Point(x, y));

picCanvas.Invoke(new Action(() =>

{

CreateStringPoints(x, y);

picCanvas.CreateGraphics().FillRectangle(Brushes.Black, x, y, 1, 1);

}));

}

private void CreateStringPoints(int x, int y)

{

cont++;

contString += cont + "\n";

XString += x + "\n";

YString += y + "\n";

NLabel.Text = contString;

XLabel.Text = XString;

YLabel.Text = YString;

}

public HashSet<Point> getBorderPoints()

{

return visited;

}

}

* Algoritmo de rellenado FloodFill

class FloodFill

{

private HashSet<Point> visited = new HashSet<Point>();

private PictureBox Canvas;

public FloodFill() {

}

public void ReadData(PictureBox picCanvas, HashSet<Point> borderC)

{

Canvas = picCanvas;

visited = visited.Union(borderC).ToHashSet();

}

public void miFloodFill(int x, int y, int depth = 0)

{

if (depth > 10000)

{

IterativeFloodFill(x, y);

return;

}

if (!visited.Contains(new Point(x, y)) && (x > 0 && y > 0 && x < Canvas.Width && y < Canvas.Height))

{

drawPixel(x, y);

miFloodFill(x, y + 1, depth + 1);

miFloodFill(x + 1, y, depth + 1);

miFloodFill(x, y - 1, depth + 1);

miFloodFill(x - 1, y, depth + 1);

}

}

public void IterativeFloodFill(int x, int y)

{

Queue<Point> queue = new Queue<Point>();

queue.Enqueue(new Point(x, y));

while (queue.Count > 0)

{

Point p = queue.Dequeue();

if (visited.Contains(p) || p.X <= 0 || p.Y <= 0 || p.X >= Canvas.Width || p.Y >= Canvas.Height)

continue;

drawPixel(p.X, p.Y);

queue.Enqueue(new Point(p.X + 1, p.Y));

queue.Enqueue(new Point(p.X, p.Y+1));

queue.Enqueue(new Point(p.X-1, p.Y));

queue.Enqueue(new Point(p.X, p.Y - 1));

}

}

public void drawPixel(int x, int y)

{

visited.Add(new Point(x,y));

Canvas.Invoke(new Action(() =>

{

Canvas.CreateGraphics().FillRectangle(Brushes.Black, x, y, 1, 1);

}));

}

}

# **Ejecución**

Por último, la ejecución del código en todas sus facetas para reconocer como es el funcionamiento y de esa forma observar los asombrosos resultados de estos algoritmos.

A continuación, se muestran los resultados obtenidos:

