Universidad de las Fuerzas Armadas “ESPE”

Departamento de ciencias de la Computación

Ingeniería en Desarrollo de Software

Tema: Algoritmos clásicos de Computación Gráfica.

Docente: Morales Caiza Darío Javier

Estudiante: Diana Carolina Guerra Coronel

Materia: Computación Gráfica

NRC: 28467

Fecha: 12/12/2025

Contenido

1. Introducción

La presente práctica tiene como objetivo implementar una aplicación de computación gráfica en C# (Windows Forms) que permita experimentar de forma interactiva con varios algoritmos clásicos de rasterización, relleno y recorte. La aplicación desarrolla, de forma funcional y demostrable, al menos tres variantes u opciones de cada uno de los siguientes algoritmos fundamentales: trazado de líneas, trazado de círculos, algoritmos de relleno, algoritmo de recorte de líneas y algoritmo de recorte de polígonos.

Cada variante puede seleccionarse desde la interfaz gráfica, visualizarse en un lienzo gráfico y probarse con diferentes parámetros de entrada. Además, se incluye una descripción técnica de los algoritmos y una comparación entre sus opciones.

2. Objetivos

2.1 Objetivo general

Desarrollar una aplicación de computación gráfica que implemente y compare múltiples variantes de los algoritmos clásicos de rasterización, relleno y recorte, permitiendo su visualización e interacción mediante una interfaz gráfica amigable.

2.2 Objetivos específicos

- Implementar al menos tres algoritmos de trazado de líneas, basados en diferentes enfoques numéricos e incrementales.
- Implementar al menos tres algoritmos de discretización de círculos, aprovechando el uso de simetría y criterios de decisión.
- Implementar algoritmos de relleno (tipo flood fill, boundary fill y scanline) para rellenar regiones y polígonos
- Implementar algoritmos de recorte de líneas, incluyendo técnicas basadas en códigos de región.
- Implementar algoritmos de recorte de polígonos, incluyendo el algoritmo de Sutherland–Hodgman y una variante incremental propia.
- Diseñar una interfaz gráfica clara e intuitiva, que permita seleccionar los algoritmos, ingresar parámetros y observar los resultados.
- Garantizar validación de entradas, manejo de errores y modularidad del código fuente.

3. Descripción general de la aplicación

La solución se implementa como un proyecto en C# Windows Forms, organizado en distintos formularios asociados a cada grupo de algoritmos (líneas, círculos, rellenos,

recorte de líneas y recorte de polígonos). A nivel general, la aplicación posee un formulario principal que actúa como menú de navegación y formularios específicos por módulo.

Cada formulario incluye un PictureBox que funciona como lienzo gráfico, donde se dibujan los resultados, y controles de entrada (MenuStrip, TextBox, Labels, RadioButton y Buttons) para ingresar parámetros y seleccionar la variante de algoritmo a utilizar.

Link de github:
https://github.com/DianyGuerra/AlgoritmosClasicosComputacionGrafica.git

Al iniciar la aplicación se presenta el formulario principal en pantalla grande para poder visualizar mejor los formularios de cada algoritmo, este contiene un MenuStrip en la parte superior que separa los formularios por categorías: “Discretización de líneas”, “Discretización de círculos”, “Algoritmos de relleno”, “Recorte de líneas”, y “Recorte de polígonos”. Este formulario principal tiene activada la propiedad “IsMdiContainer” que nos permite manejar formulario padre e hijo. De esta manera el formulario de inicio se ve de la siguiente manera:

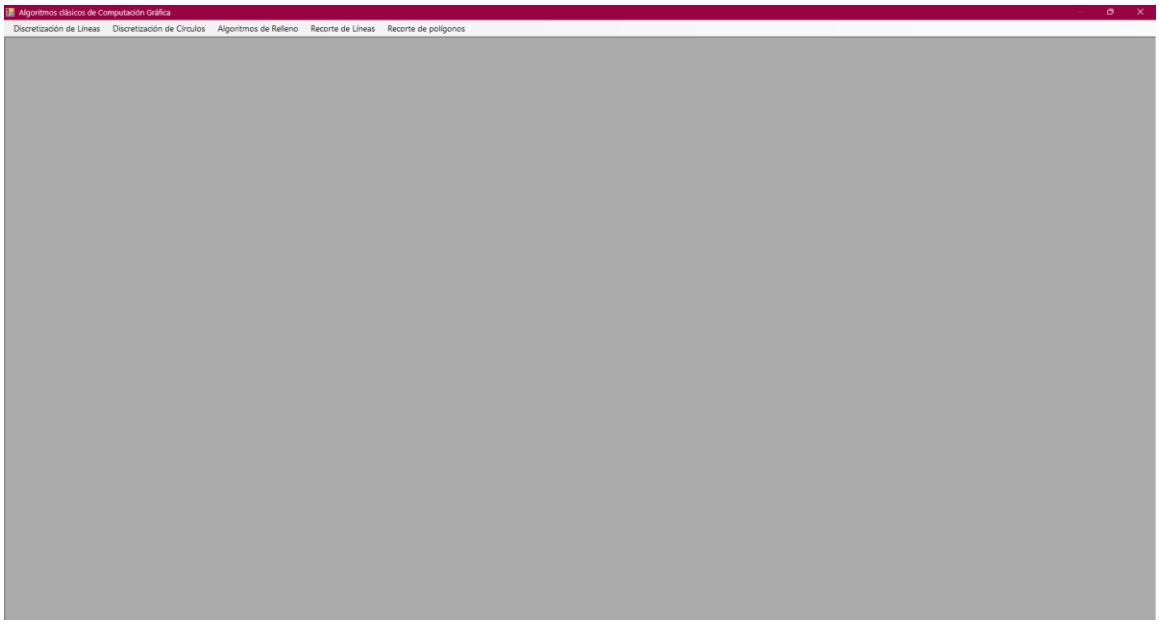


Ilustración 1. Captura de pantalla del formulario de Inicio de la aplicación

4. Algoritmos de Discretización de líneas

En el módulo de trazado de líneas se implementan tres algoritmos clásicos: el algoritmo DDA (Digital Differential Analyzer), el algoritmo de Bresenham y el algoritmo del punto medio. Estos algoritmos permiten rasterizar segmentos de línea sobre una rejilla de píxeles.

4.1. Algoritmo DDA (Digital Differential Analyzer)

El algoritmo DDA (Digital Differential Analyzer) interpola puntos de una línea calculando incrementos constantes en x o en y, según la pendiente m. Para una línea entre $(\mathrm{x}_1, \mathrm{y}_1)$ y (x_2, y_2), se calcula:

$$\Delta \mathrm{x} = x_2 - x_1$$

$$\Delta \mathrm{y} = y_2 - y_1$$

$$m = \frac{dy}{dx} = \frac{x_2 - x_1}{y_2 - y_1}$$

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

A partir de ello, se obtienen los incrementos:

$$\mathrm{x\_inc} = \Delta \mathrm{x} \,/\, \mathrm{steps}$$

$$\mathrm{y\_inc} = \Delta \mathrm{y} \,/\, \mathrm{steps}$$

y se generan los puntos sucesivos sumando estos incrementos a las coordenadas actuales y redondeando al píxel más cercano (GeeksforGeeks, 2025).

En la implementación de la aplicación, el algoritmo DDA se encapsula en un evento que recibe las coordenadas iniciales y finales de la línea por ingreso en un textbox, valida los datos de entrada y utiliza un bucle para trazar la línea en el PictureBox.

4.1.1 Justificación de la variante del algoritmo

Se eligió el algoritmo DDA (Digital Differential Analyzer) porque es uno de los métodos clásicos más sencillos para discretizar líneas: parte directamente de la pendiente del segmento y genera los puntos mediante incrementos constantes en x e y, lo que lo hace ideal para fines didácticos y para entender el paso de la recta continua a la rejilla de píxeles. Aunque es menos eficiente que algoritmos como Bresenham por usar operaciones en punto flotante, es una buena referencia para comparar precisión y rendimiento entre variantes. (*DDA Algorithm in Computer Graphics*, n.d.)

4.1.2 Descripción del Formulario

4.1.2.1 Diseño del formulario

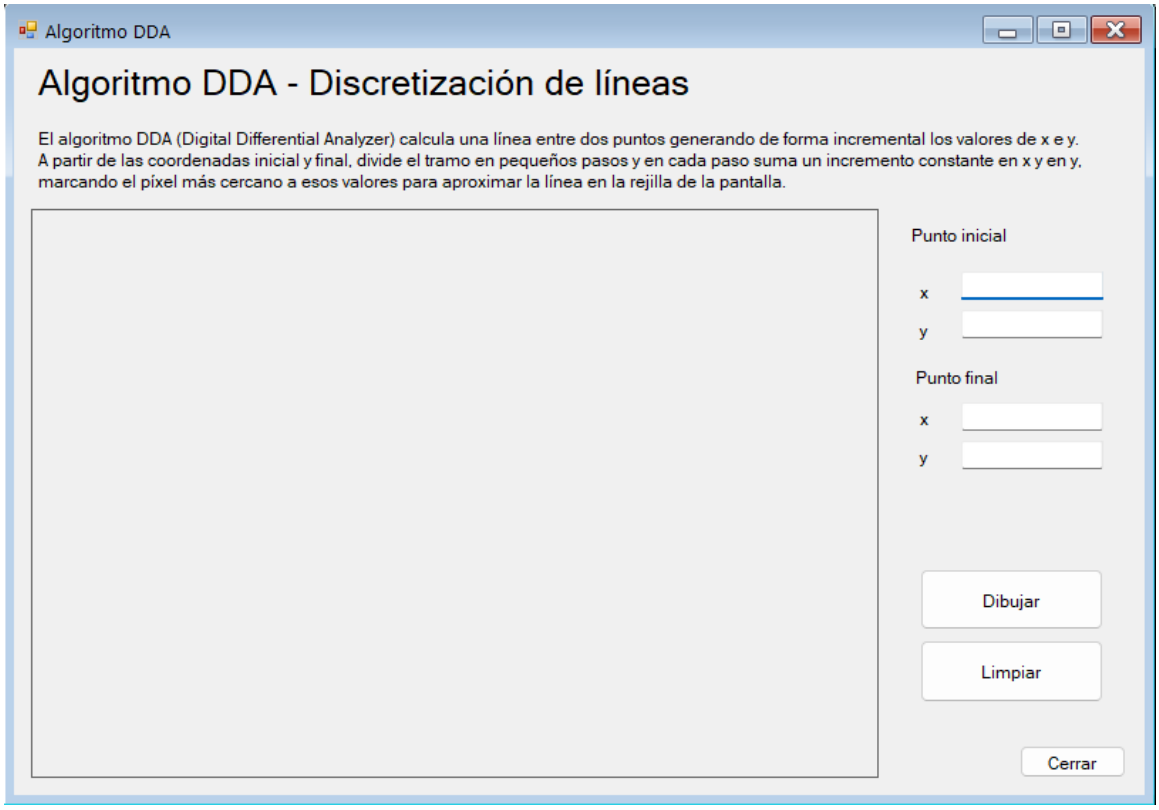


Ilustración 2. Captura del formulario para el algoritmo DDA de discretización de líneas

El formulario muestra el título *"Algoritmo DDA – Discretización de líneas"*, una breve descripción del algoritmo, un lienzo (PictureBox) grande a la izquierda donde se dibuja la línea y, a la derecha, cuatro cajas de texto para las coordenadas del punto inicial y punto final, junto con los botones Dibujar, Limpiar y Cerrar. El resultado se ve como una sucesión de cuadritos negros (píxeles de la línea) y dos puntos rojos que marcan el inicio y el final del segmento.

4.1.2.2 Parámetros utilizados

El formulario recibe como parámetros de entrada las coordenadas enteras y positivas del segmento:

- x0, y0: coordenadas del punto inicial.
- x1, y1: coordenadas del punto final.

Internamente además se usa un factor de escala SF = 10 para ampliar la visualización en el lienzo, y tamaños fijos (8×8 píxeles) para los cuadros que representan los puntos dibujados.

4.1.2.3 Explicación de las funciones

- btnDraw_Click: valida que todos los TextBox contengan enteros positivos usando int.TryParse; si alguna entrada es inválida muestra un MessageBox y no dibuja. Si todo es correcto, calcula dx, dy, el número de pasos steps y los incrementos xIncrement y yIncrement, recorre un bucle y va pintando los puntos intermedios con FillRectangle. Al final marca los puntos inicial y final en rojo con FillEllipse.
- btnLimpiar_Click: limpia el lienzo llamando a picCanvas.Refresh().
- btnCerrar_Click: cierra el formulario con this.Close().

4.1.3 Casos de prueba

1. El usuario deja uno de los campos vacío o escribe una letra (por ejemplo, x0 = "a").

   Resultado esperado: al presionar Dibujar, se muestra el mensaje *"Por favor ingresa solo números enteros positivos (sin negativos)."* y no se dibuja nada en el lienzo.

2. El usuario corrige todos los campos con valores negativos (por ejemplo, x0 = -2) o mezcla positivos y negativos.

   Resultado esperado: el mismo mensaje de error aparece y el lienzo permanece sin cambios, comprobando que la condición x < 0 y similares bloquea coordenadas negativas.

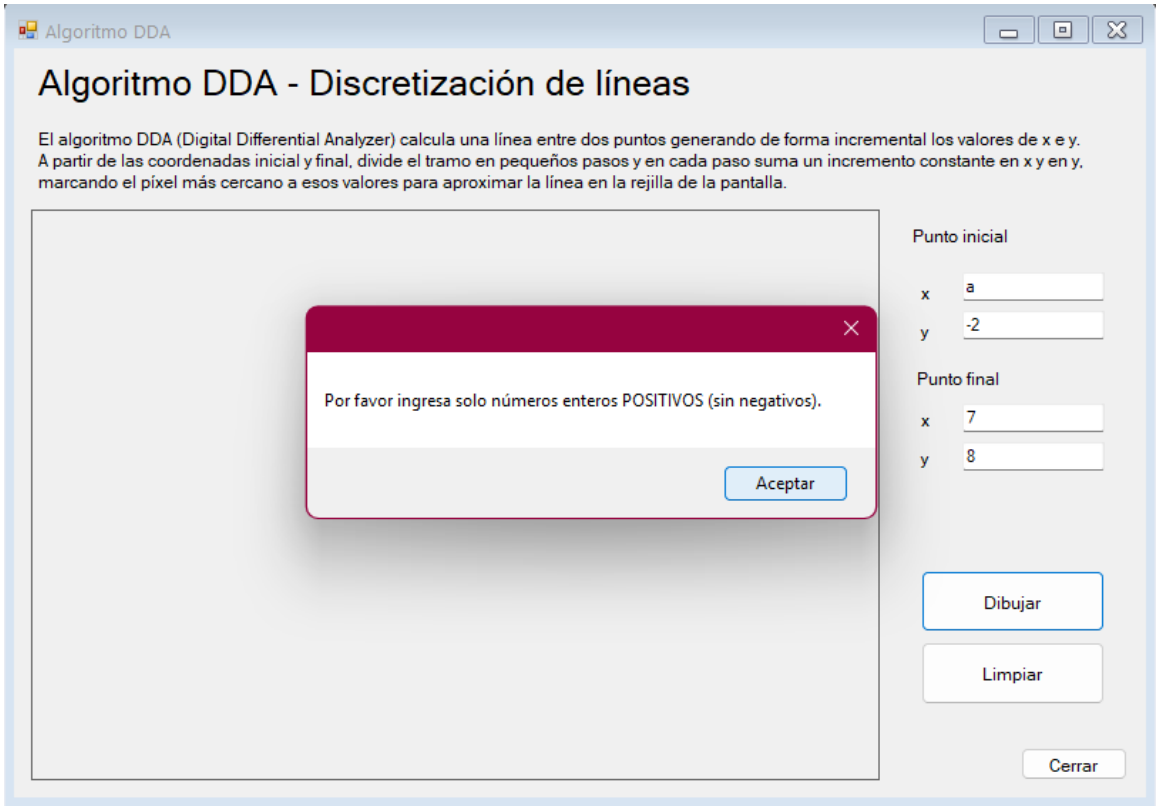


Ilustración 3. Casos de prueba 1 y 2 – Algoritmo DDA

3. Finalmente, el usuario ingresa valores válidos, por ejemplo: x0 = 2, y0 = 5, x1 = 7, y1 = 8.

   Resultado esperado: al presionar Dibujar, aparece una secuencia de cuadritos negros formando una línea discreta entre los puntos escalados (2·SF, 5·SF) y (7·SF, 8·SF), y se muestran ambos extremos marcados con puntos rojos, sin mensajes de error ni cierre inesperado del programa.

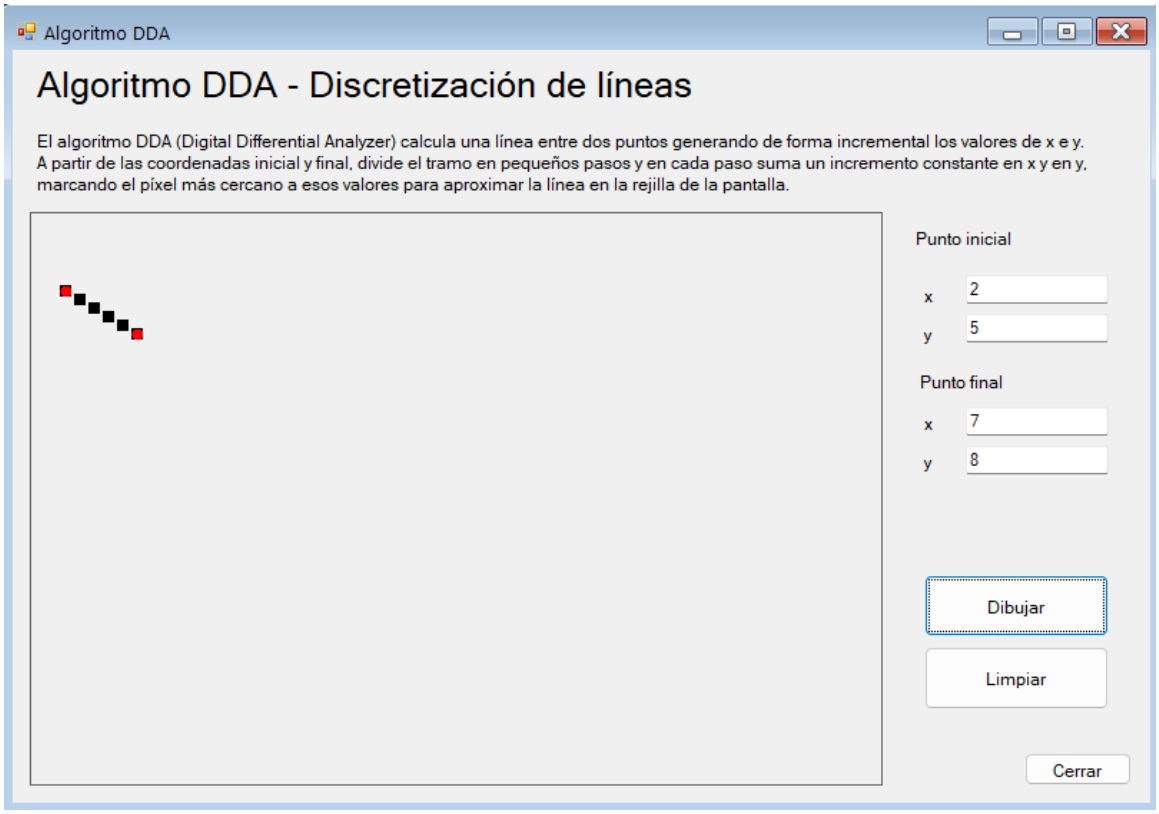


Ilustración 4. Caso de prueba 3 - Algoritmo DDA

4.2 Algoritmo de Bresenham

El algoritmo de Bresenham rasteriza una línea utilizando únicamente operaciones enteras y un parámetro de error incremental que indica qué píxel es más cercano a la línea ideal. En cada paso se decide si el siguiente píxel debe estar en la misma fila o en la fila adyacente, según el signo del error acumulado respecto a la función de línea (GeeksforGeeks, 2025).

$$f(x, y) = Ax + By + C$$

En la aplicación, el algoritmo se implementa en un método que calcula dx, dy y un criterio de decisión, y recorre la línea actualizando el error y las coordenadas. Este algoritmo es más eficiente que DDA, ya que evita el uso de números en punto flotante.

4.2.1 Justificación de la variante del algoritmo

El algoritmo de Bresenham se utiliza como segunda variante porque es el estándar clásico para el trazado de líneas en raster: determina qué píxeles activar usando únicamente aritmética entera y un error incremental, evitando multiplicaciones y divisiones en punto flotante. Esto lo hace más rápido y eficiente que el DDA, por lo que históricamente se ha

usado en plotters, firmware y librerías de gráficos como método básico para dibujar líneas precisas en pantallas discretas. (GeeksforGeeks, 2025a)

4.2.2 Descripción del Formulario

4.2.2.1 Diseño del formulario

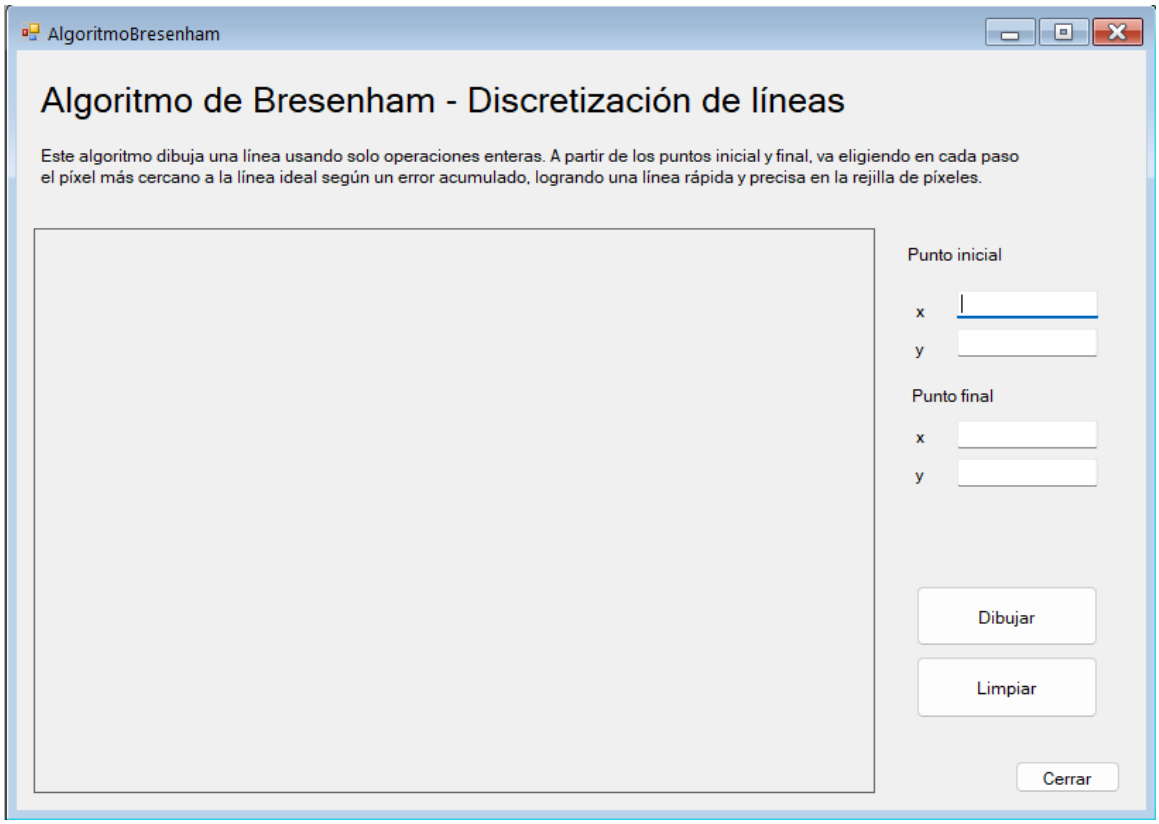


Ilustración 5. Formulario para el algoritmo de Bresenham para discretización de líneas

El formulario "Algoritmo de Bresenham – Discretización de líneas" tiene una estructura similar al de DDA: en la parte superior se muestra el título y una breve descripción del algoritmo; al centro-izquierda se ubica el lienzo (PictureBox) donde se dibuja la línea discretizada, y a la derecha se encuentran los TextBox para las coordenadas del punto inicial y punto final, junto con los botones Dibujar, Limpiar y Cerrar. El usuario ingresa los datos a la derecha y visualiza la línea formada por pequeños cuadros negros en el área de dibujo.

4.2.2.2 Parámetros utilizados

El formulario utiliza como parámetros de entrada:

- x0, y0: coordenadas enteras y positivas del punto inicial.
- x1, y1: coordenadas enteras y positivas del punto final.

Internamente se maneja un factor de escala SF = 10 para ampliar la visualización sobre el PictureBox y se dibujan cuadros de 8×8 píxeles para cada punto de la línea, más dos círculos rojos para marcar los extremos.

4.2.2.3 Explicación de las funciones

- btnDraw_Click
  - Valida las entradas con int.TryParse y verifica que todas las coordenadas sean enteros positivos; en caso contrario, muestra un MessageBox y termina la ejecución.
  - Calcula dx = |x1 − x0|, dy = |y1 − y0|, los signos sx y sy para saber en qué dirección avanza la línea y el error inicial err = dx − dy.
  - Dentro de un bucle while (true) aplica el algoritmo de Bresenham: dibuja el punto actual, calcula e2 = 2 * err y, según su valor, ajusta x0 y/o y0 y el error err hasta alcanzar el punto final.
  - Al finalizar dibuja los puntos inicial y final en rojo con FillEllipse.
- btnLimpiar_Click limpia el lienzo con picCanvas.Refresh().
- btnCerrar_Click cierra el formulario con this.Close().

4.2.3 Casos de prueba

1. El usuario deja un campo vacío o ingresa texto no numérico, por ejemplo: x0 = "a", y0 = "10", x1 = "20", y1 = "5".

   Resultado esperado: al presionar Dibujar, la condición de validación falla, se muestra el mensaje *"Por favor ingresa solo números enteros POSITIVOS (sin negativos)."* y no se dibuja nada en el PictureBox.

2. El usuario corrige los campos, pero coloca algún valor negativo, por ejemplo: x0 = -3, y0 = 4, x1 = 10, y1 = 8.

   Resultado esperado: nuevamente se muestra el mismo mensaje de validación (por la condición x < 0 o y < 0) y la línea no se dibuja, comprobando que no se aceptan coordenadas negativas.

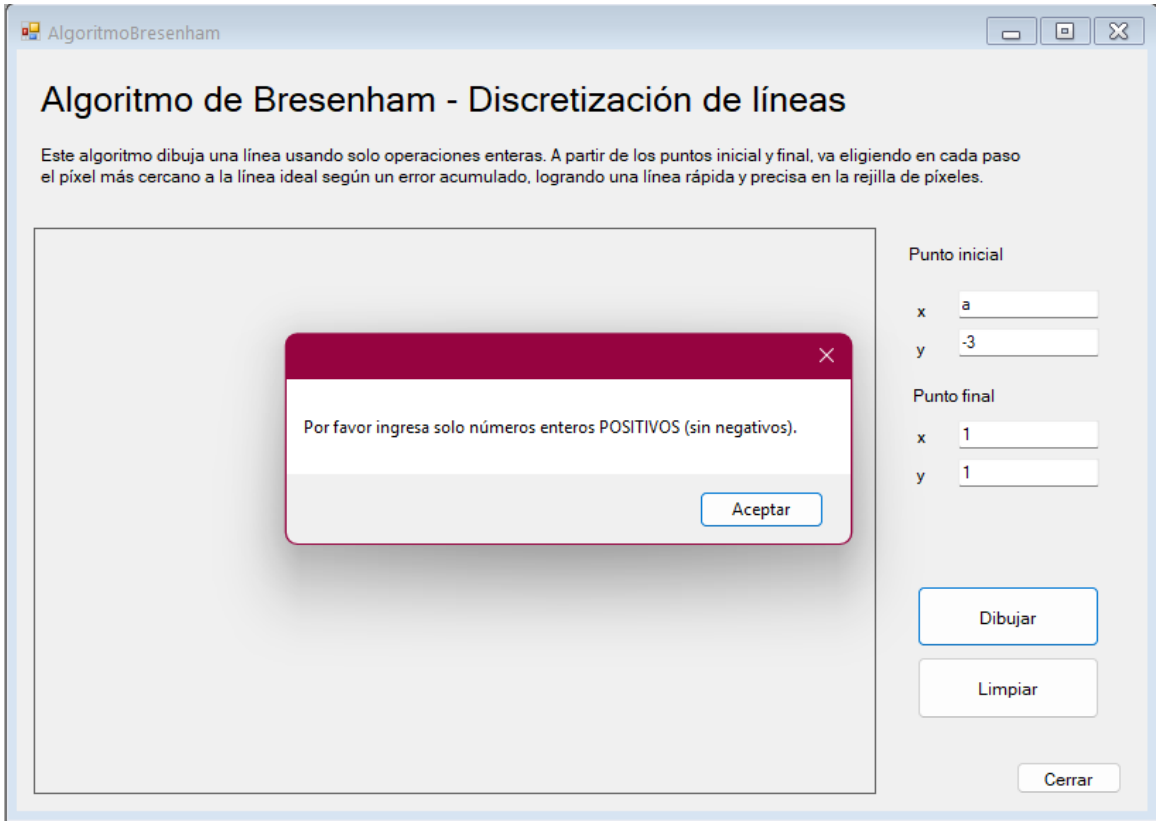


Ilustración 6. Casos de prueba 1 y 2 - Algoritmo de Bresenham

3. Finalmente ingresa un conjunto de valores válidos, por ejemplo: x0 = 2, y0 = 1, x1 = 15, y1 = 10.

   Resultado esperado: al presionar Dibujar, aparece en el lienzo una línea formada por pequeños cuadros negros que une los puntos escalados (2·SF, 1·SF) y (15·SF, 10·SF), con los extremos marcados en rojo. El trazo se muestra continuo y sin saltos, evidenciando que el algoritmo de Bresenham está calculando correctamente los píxeles de la línea.

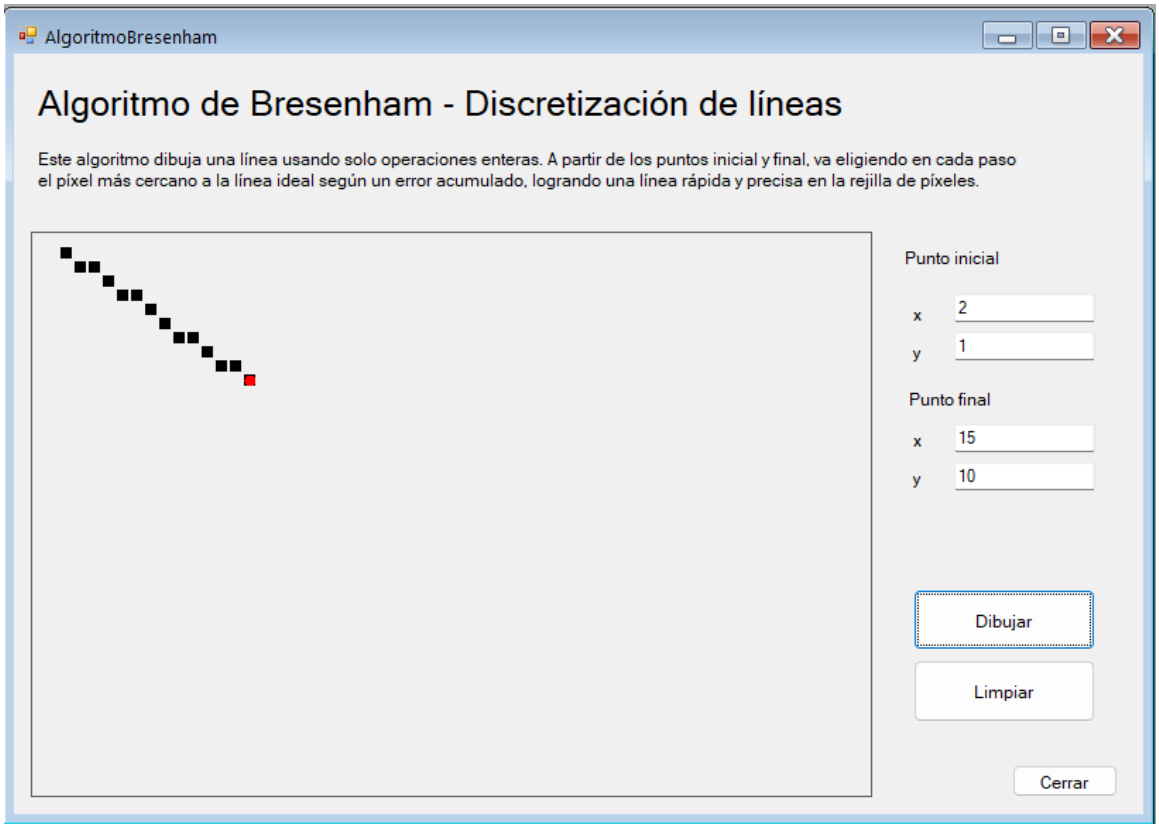


Ilustración 7. Caso de prueba 3 - Algoritmo de Bresenham

4.3 Algoritmo del punto medio para líneas

El algoritmo del punto medio para líneas es similar al de Bresenham, pero se presenta de forma más didáctica en términos de la evaluación de la función de línea en el punto medio entre dos posibles píxeles. Se utiliza la ecuación $f(x, y) = Ax + By + C$ y, en cada paso, se actualiza este valor según el movimiento seleccionado (horizontal o diagonal).

En la aplicación, este algoritmo se implementa como una tercera variante de trazado, permitiendo al usuario comparar el comportamiento con DDA y Bresenham.

4.3.1 Justificación de la variante del algoritmo

El algoritmo del punto medio para líneas se incluye como tercera variante porque representa una formulación más explícita del enfoque geométrico de Bresenham: en cada paso se evalúa la ecuación implícita de la recta en el punto medio entre dos píxeles candidatos y, según el signo de esa evaluación, se elige el siguiente píxel (horizontal o diagonal). Es un algoritmo de *scan conversion* que, igual que Bresenham, trabaja con operaciones enteras y resulta eficiente y sencillo de implementar, por lo que suele presentarse en muchos cursos como alternativa o derivación de Bresenham (GeeksforGeeks, 2025).

4.3.2 Descripción del Formulario

4.3.2.1 Diseño del formulario

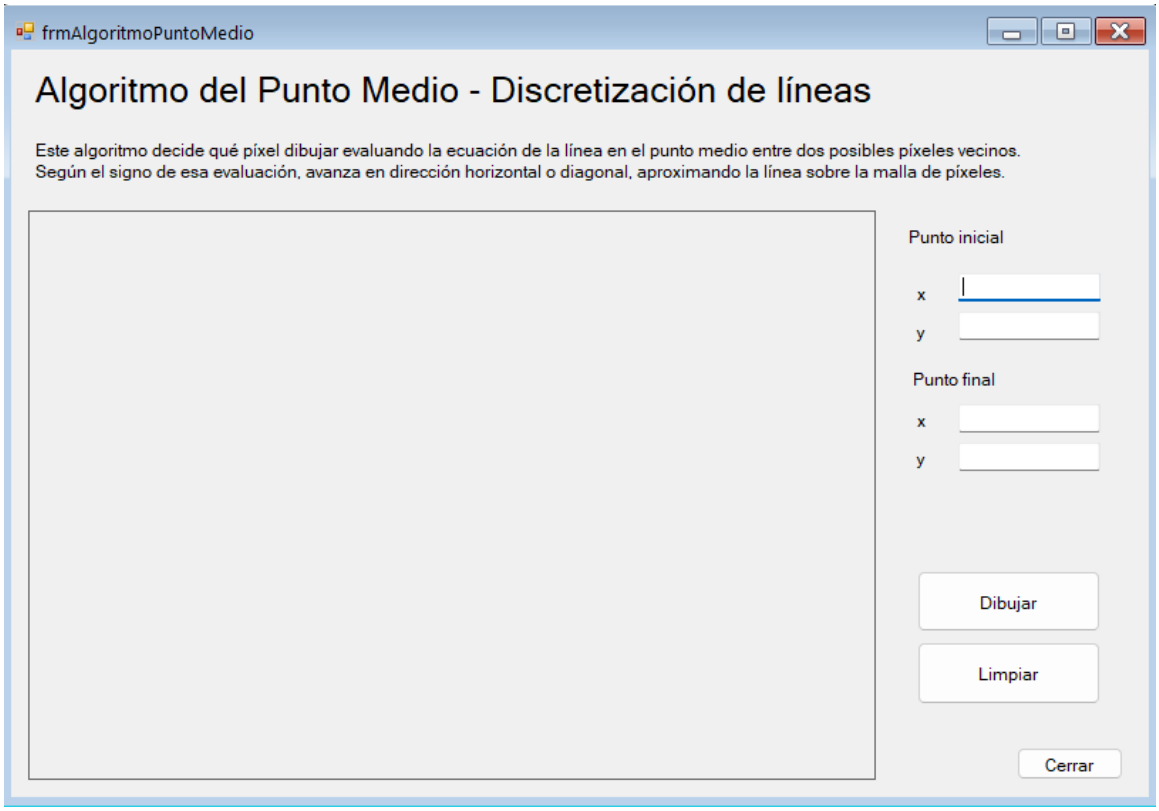


Ilustración 8. Formulario del algoritmo del Punto Medio para discretización de líneas.

El formulario "Algoritmo del Punto Medio – Discretización de líneas" mantiene la misma estructura de los formularios anteriores: título y breve descripción en la parte superior, un PictureBox grande a la izquierda como área de dibujo y, a la derecha, cuatro TextBox para las coordenadas x, y del punto inicial y del punto final, más los botones Dibujar, Limpiar y Cerrar. La línea generada se visualiza como una serie de pequeños cuadros negros en el lienzo, y los extremos se resaltan con puntos rojos.

4.3.2.2 Parámetros utilizados

Entradas que proporciona el usuario:

- x0, y0: coordenadas enteras y positivas del punto inicial.
- x1, y1: coordenadas enteras y positivas del punto final.

Parámetros internos relevantes:

- SF = 10: factor de escala para ampliar la línea en el PictureBox.
- $dx = |x1 - x0|, dy = |y1 - y0|$: diferencias absolutas en cada eje.

- sx, sy: dirección del avance en x e y (1 o −1).
- d: parámetro de decisión del algoritmo del punto medio, inicializado como $2 * dy - dx$ cuando | m |≤ 1 o $2 * dx - dy$ cuando | m |> 1.

4.3.2.3 Explicación de las funciones

- btnDraw_Click
  - Limpia el lienzo (picCanvas.Refresh()), crea el objeto Graphics y los pinceles para los pasos (negro) y puntos extremos (rojo).
  - Valida que todas las coordenadas sean enteros positivos usando int.TryParse y comparaciones x < 0/y < 0; si falla, muestra el mensaje *"Por favor ingresa solo números enteros POSITIVOS (sin negativos)."* y aborta el dibujo.
  - Calcula dx, dy, sx, sy y decide entre dos casos:
    - Si dx >= dy (|pendiente| ≤ 1), avanza en X: en cada iteración dibuja el punto, evalúa d para saber si debe incrementar y (y += sy) y actualiza d con 2*dy y −2*dx según corresponda.
    - Si dx < dy (|pendiente| > 1), avanza en Y: el esquema es similar pero intercambiando roles de dx y dy, moviéndose principalmente en y.
  - Al finalizar, dibuja los puntos inicial y final en rojo (FillEllipse).
- btnLimpiar_Click borra el contenido del PictureBox.
- btnCerrar_Click cierra el formulario.

4.3.3 Casos de prueba

1. El usuario intenta dibujar una línea dejando un campo vacío o con texto no numérico, por ejemplo: x0 = "", y0 = "5", x1 = "10", y1 = "20".

   Resultado esperado: al presionar **Dibujar**, la validación con int.TryParse falla, se muestra el mensaje de error de enteros positivos y no se dibuja nada en el lienzo.

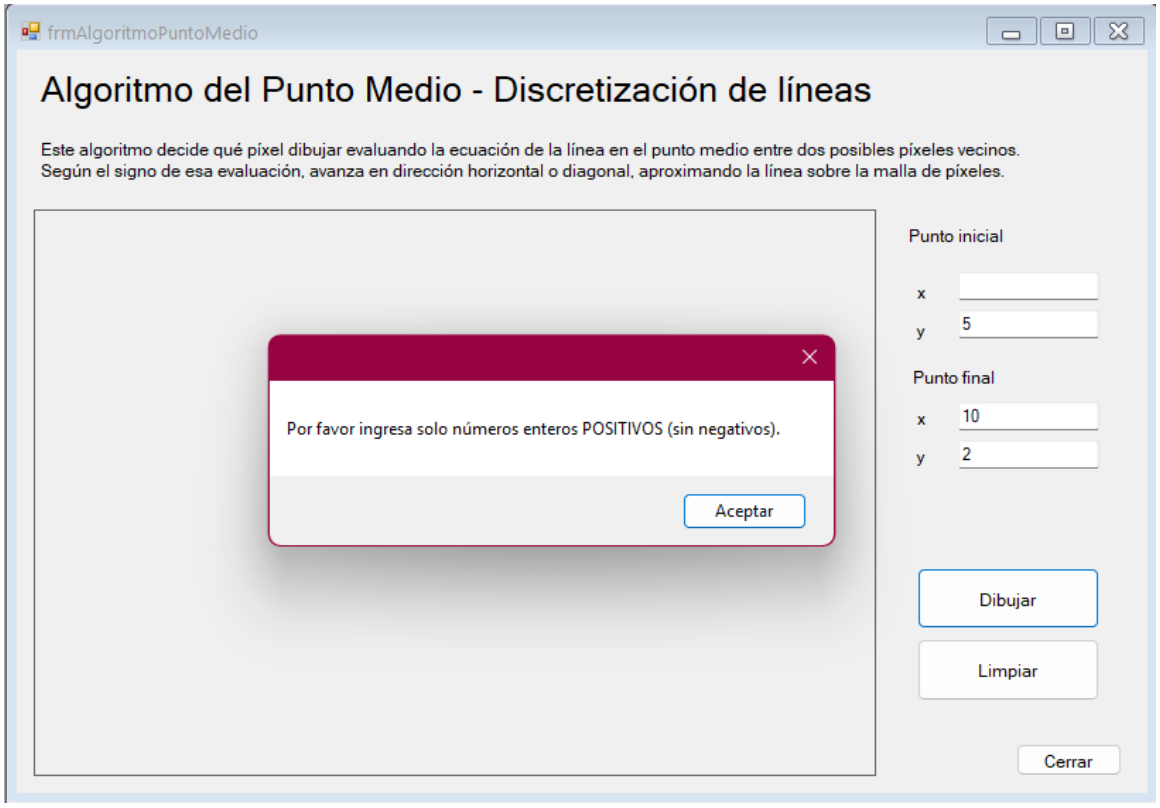


Ilustración 9. Caso de prueba 1 - Algoritmo del Punto Medio

2. A continuación, el usuario ingresa un valor negativo en alguno de los campos, por ejemplo: x0 = -2, y0 = 5, x1 = 7, y1 = 9.

   Resultado esperado: la condición x0 < 0 detecta el valor inválido, se vuelve a mostrar el mismo mensaje de validación y el PictureBox permanece sin cambios, comprobando que no se permiten coordenadas negativas.

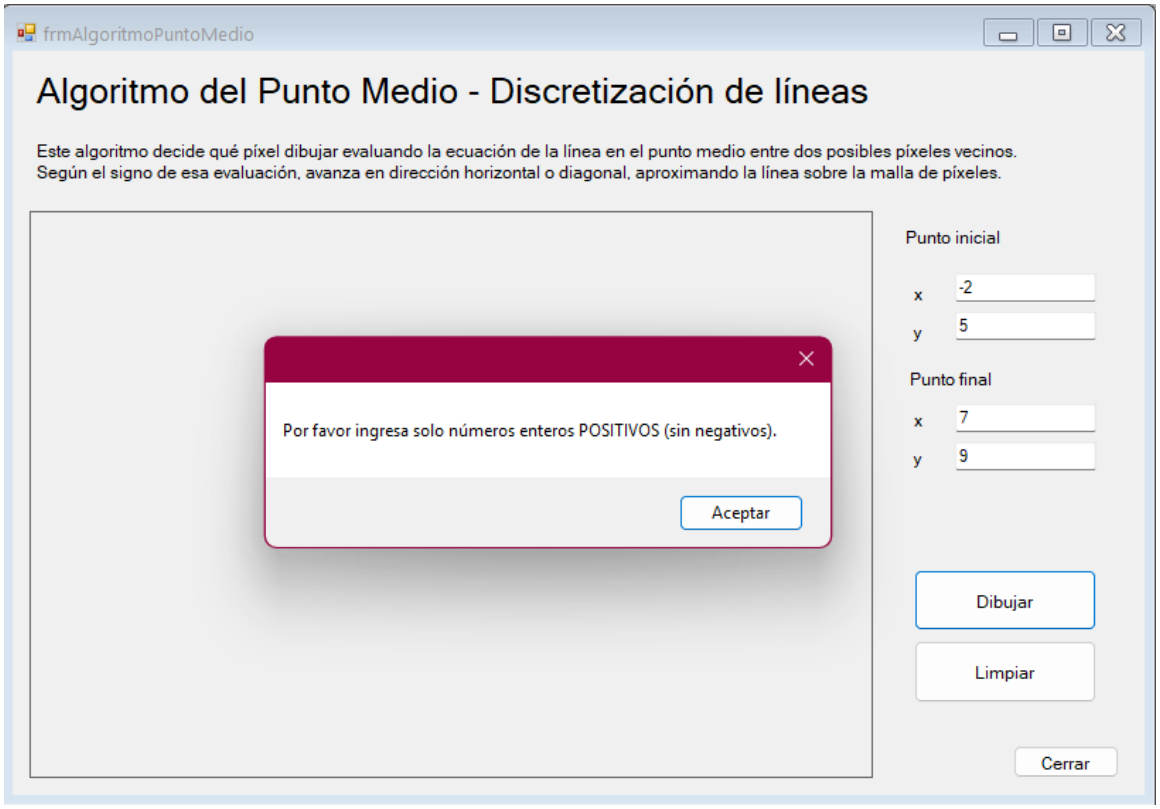


Ilustración 10. Caso de prueba 2 - Algoritmo del Punto Medio

3. Finalmente, el usuario introduce valores válidos que obliguen al algoritmo a usar ambos casos de pendiente, por ejemplo: x0 = 2, y0 = 1, x1 = 12, y1 = 4 (pendiente suave, usa el bucle que avanza en X) y luego x0 = 3, y0 = 2, x1 = 5, y1 = 12 (pendiente pronunciada, usa el bucle que avanza en Y).

   Resultado esperado: en cada ejecución de **Dibujar** se muestra una línea discreta formada por cuadros negros entre los puntos indicados, con los extremos resaltados en rojo; en ningún caso se presentan errores ni cierres inesperados, y visualmente se aprecia que el algoritmo aproxima correctamente líneas con pendientes menores y mayores que 1.

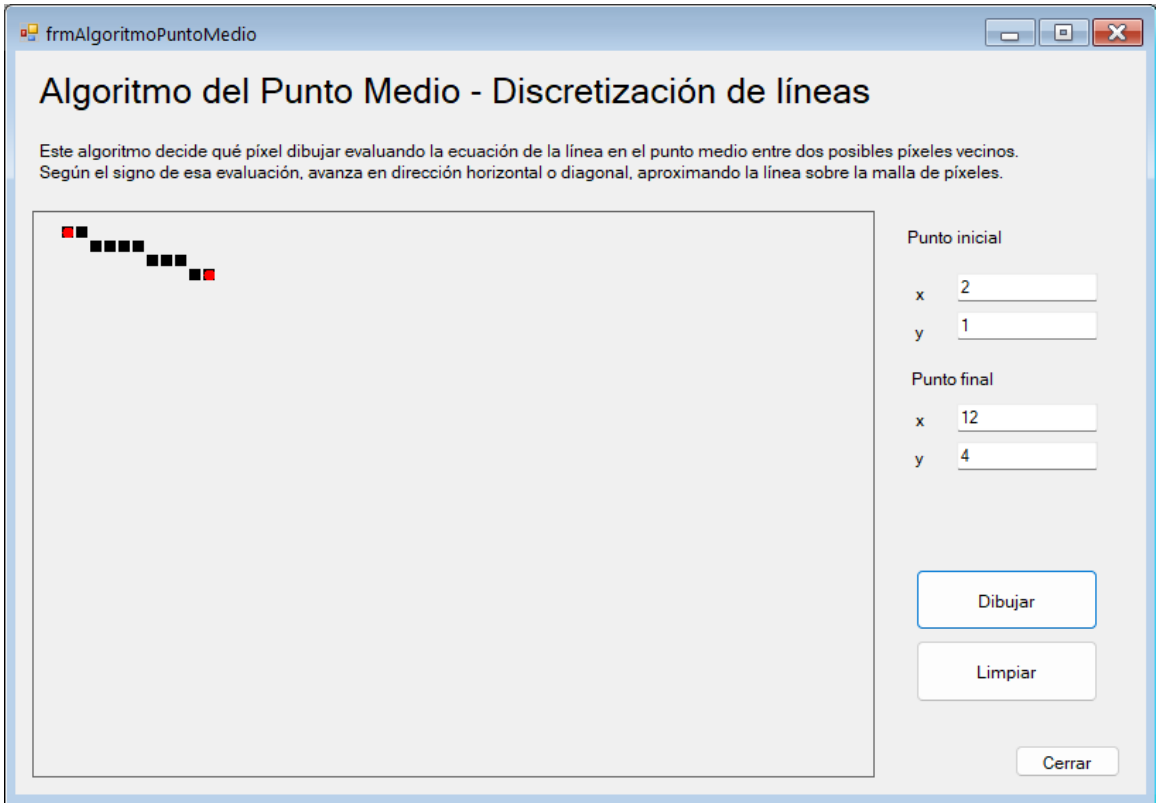


Ilustración 11. Caso de prueba 3 - Algoritmo del Punto Medio

5. Discretización de círculos

En el módulo de discretización de círculos se implementan tres variantes: un algoritmo paramétrico basado en $x = xc + r\,cos(\theta)$, $y = yc + r\,sin(\theta)$; el algoritmo de punto medio para círculos (midpoint circle algorithm); y el algoritmo de Bresenham para círculos usando puntos simétricos en los 8 octantes.

5.1 Algoritmo paramétrico polar del círculo

El algoritmo paramétrico representa el círculo de centro (xc, yc) y radio r mediante las ecuaciones $x(\theta) = xc + r\,cos(\theta)$ e $y(\theta) = yc + r\,sin(\theta)$. Para rasterizar el círculo se recorre θ desde 0 hasta 2π con un paso angular adecuado y se dibujan los píxeles correspondientes al redondear las coordenadas (TutorialsSpace- Er. Deepak Garg, 2020).

Esta variante es sencilla de comprender, pero requiere operaciones trigonométricas costosas y la elección del paso angular influye directamente en la calidad de la discretización.

5.1.1 Justificación de la variante del algoritmo

El algoritmo paramétrico polar del círculo se basa en las ecuaciones paramétricas descritas anteriormente, que describen todos los puntos de una circunferencia a partir del centro y el radio.

En computación gráfica se usa como variante básica porque:

- Permite una implementación directa del concepto matemático de círculo usando un ángulo que se incrementa desde 0 hasta 2π, generando los puntos uno a uno.
- Con un paso angular adecuado (habitualmente $d\theta = 1/r$) se obtienen puntos aproximadamente separados una unidad de píxel, logrando una circunferencia razonablemente uniforme para fines educativos.
- Es una buena base de comparación frente a algoritmos más eficientes (como el de punto medio), ya que hace visible el costo de usar funciones trigonométricas en cada iteración. (Dangi, n.d.).

5.1.2 Descripción del Formulario

5.1.2.1 Diseño del formulario

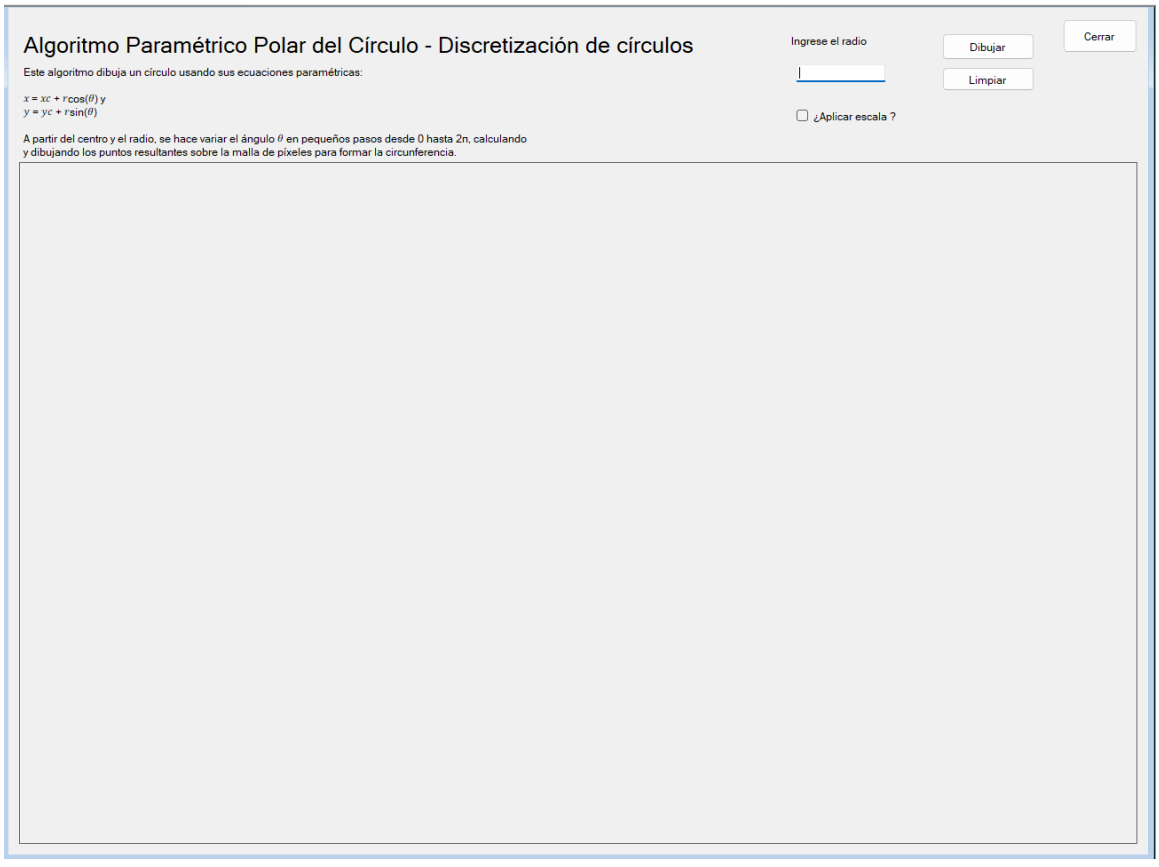


Ilustración 12. Formulario para el Algoritmo Paramétrico Polar de discretización de círculos

El formulario “Algoritmo Paramétrico Polar del Círculo – Discretización de círculos” presenta en la parte superior el título y una breve explicación con las ecuaciones paramétricas del círculo. A la derecha se ubican: un cuadro de texto para ingresar el radio, un check box que permite activar o no una escala adicional y los botones Dibujar, Limpiar y Cerrar. En la parte central inferior se dispone un PictureBox grande que actúa como lienzo, donde se van dibujando pequeños puntos que forman la circunferencia generada por el algoritmo.

5.1.2.2 Parámetros utilizados

Entradas proporcionadas por el usuario:

- radius: radio del círculo, ingresado en el cuadro de texto txtRadius (entero positivo).
- checkBox1: indica si se debe aplicar el factor de escala adicional (marcado) o dibujar en escala 1:1 (desmarcado).

Parámetros internos relevantes:

- cx, cy: coordenadas del centro del círculo, calculadas como la mitad del ancho y alto del PictureBox.
- SF: factor de escala inicial igual a 1; si checkBox1 está marcado se cambia a 5 para aumentar visualmente el tamaño del círculo.
- step: tamaño del incremento angular, definido como 1 / radius para obtener puntos aproximadamente separados una unidad de píxel.
- theta: ángulo en radianes que se incrementa desde 0 hasta 2π durante el bucle principal.

5.1.2.3 Explicación de las funciones

- btnDraw_Click
  - Limpia el lienzo (picCanvas.Refresh), obtiene el contexto Graphics y calcula el centro (cx, cy).
  - Valida que txtRadius contenga un entero positivo mediante int.TryParse y la condición radius > 0; si la validación falla, muestra un MessageBox indicando que el radio debe ser mayor que 0 y termina la ejecución.
  - Inicializa las variables del algoritmo paramétrico: radius, step = 1f / radius y theta = 0.
  - En un bucle while (theta <= 2 * Math.PI) calcula las coordenadas x = radius * cos(theta) e y = radius * sin(theta), decide el valor de SF según el estado de checkBox1 y dibuja un pequeño círculo (FillEllipse) en la posición (cx + x*SF, cy + y*SF).
  - Incrementa theta en cada iteración con theta += step hasta completar la circunferencia.
- btnLimpiar_Click

- Limpia el PictureBox, borra el contenido de txtRadius y devuelve el foco al cuadro de texto para permitir una nueva prueba rápida.

➢ btnCerrar_Click

- Cierra el formulario actual con this.Close().

5.1.3 Casos de prueba

1. El usuario intenta dibujar un círculo dejando el campo de radio vacío o ingresando texto no numérico, por ejemplo: txtRadius = "abc".

   Resultado esperado: al presionar el botón Dibujar, la validación con int.TryParse falla, se muestra el mensaje “Por favor ingresa un radio mayor a 0 (solo enteros positivos).” y no se dibuja nada en el PictureBox.

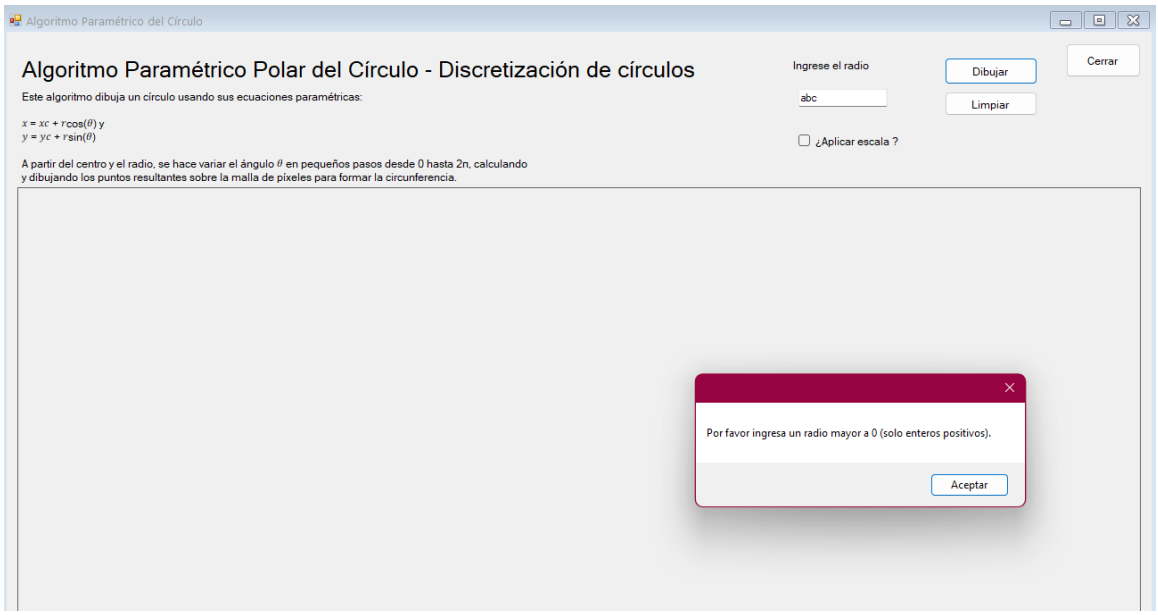


Ilustración 13. Caso de prueba 1 - Algoritmo paramétrico polar del círculo.

2. El usuario corrige el texto, pero coloca un valor no válido, por ejemplo: txtRadius = "0" o txtRadius = "-5".

   Resultado esperado: la condición radius <= 0 vuelve a activar el mismo mensaje de validación; el lienzo permanece limpio y el programa no se cierra ni produce errores, verificando que no se aceptan radios nulos o negativos.

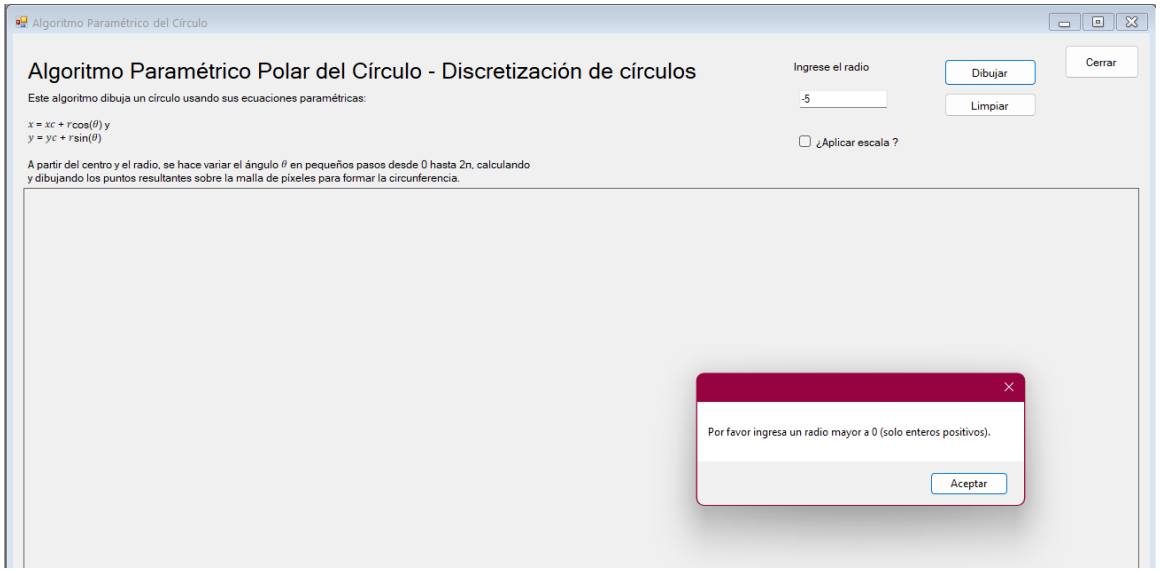


Ilustración 14. Caso de prueba 2 - Algoritmo paramétrico polar del círculo

3. El usuario ingresa un valor válido, por ejemplo: txtRadius = "30", dejando el checkBox1 desmarcado.

   Resultado esperado: al presionar Dibujar se genera una circunferencia de radio 30 centrada en el PictureBox, formada por pequeños puntos negros equiespaciados alrededor del centro; el círculo se ve completo y sin cortes apreciables.

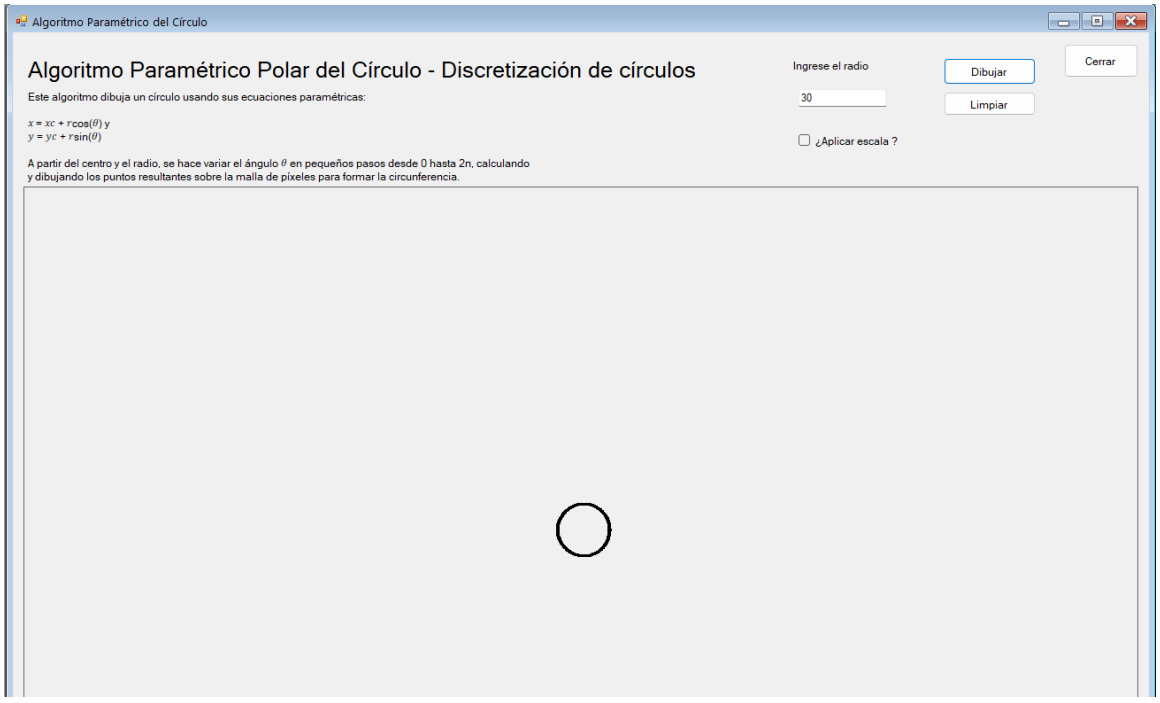


Ilustración 15. Caso de prueba 3 - Algoritmo paramétrico polar del círculo

4. Sin cerrar el formulario, el usuario marca checkBox1, limpia el lienzo con el botón Limpiar e ingresa nuevamente txtRadius = "30".

   Resultado esperado: al presionar Dibujar se obtiene otra circunferencia, ahora multiplicada por el factor SF = 5 y dibujada en color rojo; el círculo aparece más grande, pero sigue centrado en el PictureBox, confirmando que el check box modifica únicamente la escala de representación sin afectar la validez del algoritmo paramétrico.

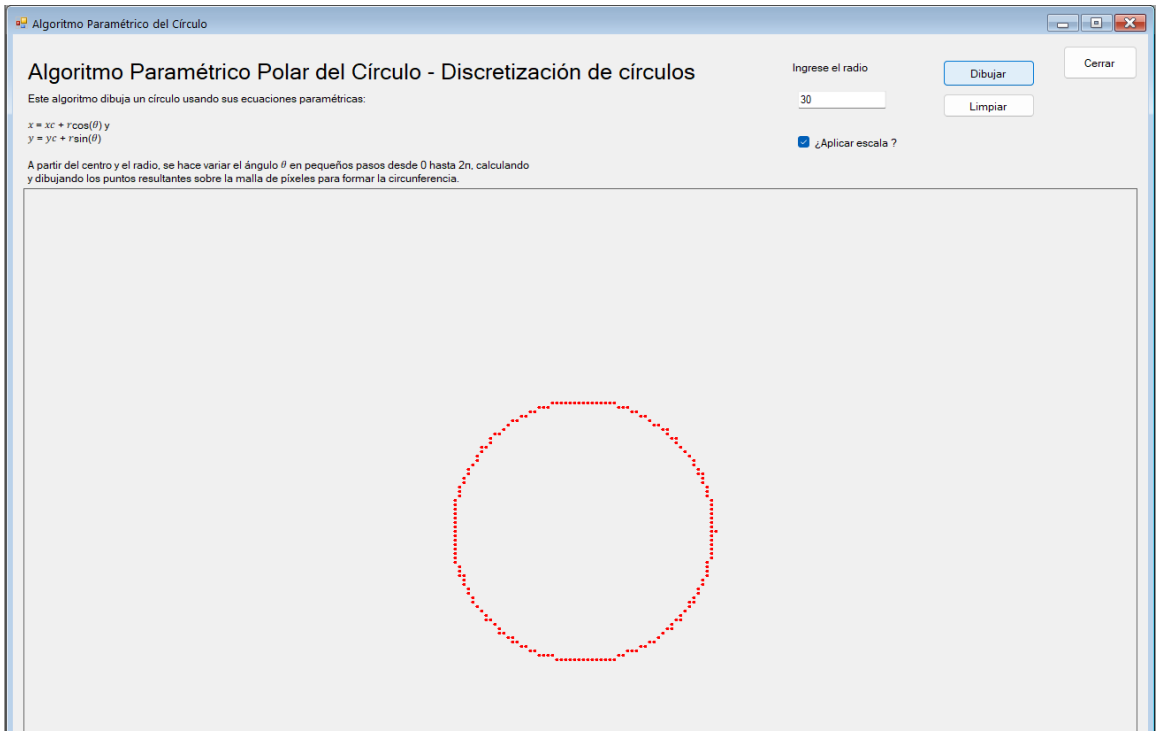


Ilustración 16. Caso de prueba 4 - Algoritmo paramétrico polar del círculo.

5.2 Algoritmo del punto medio para círculos

El algoritmo del punto medio para círculos genera los puntos del primer octante del círculo y utiliza la simetría para reflejarlos en los demás octantes. Usando la fórmula general del círculo parte del punto (x, y) = (0, r) y evalúa un criterio de decisión p que indica si el siguiente punto debe moverse hacia el Este o hacia el Sureste. Las fórmulas típicas son:

$$p_0 = 1 - r$$

Si $p < 0$ el siguiente punto es $(x + 1, y)$ y $p = p + 2x + 3$; si $p \geq 0$, el siguiente punto es $(x + 1, y - 1)$ y $p = p + 2(x - y) + 1$.

En la aplicación, este algoritmo se implementa como una variante eficiente que solo utiliza sumas y restas enteras, adecuada para gráficos en tiempo real. (GeeksforGeeks, 2022)

5.2.1 Justificación de la variante del algoritmo

El algoritmo del punto medio para círculos es una generalización de Bresenham que permite rasterizar una circunferencia usando solo operaciones enteras y la simetría de ocho octantes. Calcula primero los puntos de un octante del círculo y luego refleja esos puntos en los otros siete, reduciendo el número de cálculos necesarios. (GeeksforGeeks, 2022) Se eligió esta variante porque:

- Es un algoritmo eficiente: evita raíces cuadradas y funciones trigonométricas, usando únicamente sumas y restas enteras.
- Aprovecha de forma explícita la simetría del círculo, lo que lo hace adecuado para explicar el concepto de octantes en computación gráfica.
- Es un estándar en la literatura de gráficos para dibujar círculos discretos, por su buen equilibrio entre rendimiento y precisión.

5.2.2 Descripción del Formulario

5.2.2.1 Diseño del formulario

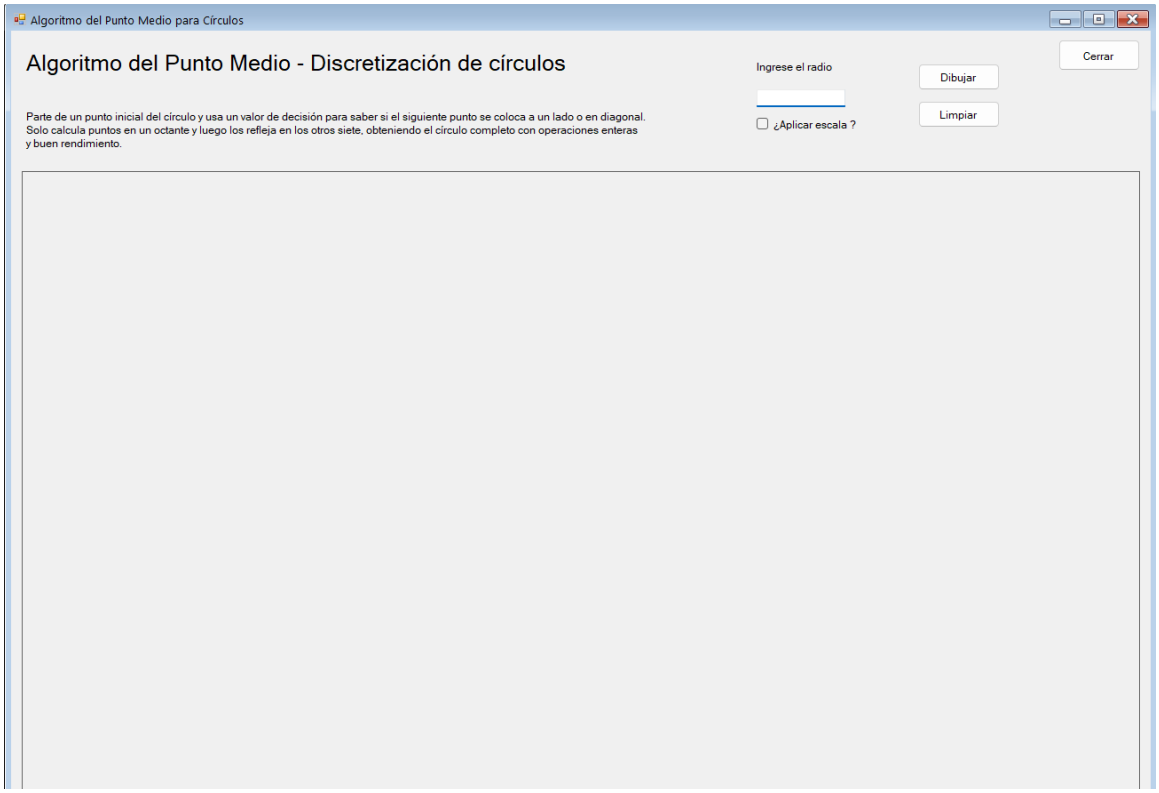


Ilustración 17. Formulario del Algoritmo del Punto Medio para discretización de círculos.

El formulario “Algoritmo del Punto Medio – Discretización de círculos” muestra en la parte superior el título y un texto breve que describe la idea del algoritmo (valor de decisión y uso de un solo octante con simetría). A la derecha se encuentra un cuadro de texto para ingresar

el radio, un CheckBox para activar o no la escala (aplicar escala) y los botones Dibujar, Limpiar y Cerrar. En la parte central inferior se dispone un PictureBox amplio que funciona como lienzo, donde se dibuja la circunferencia mediante pequeños cuadrados negros o rojos, distribuidos alrededor del centro calculado.

5.2.2.2 Parámetros utilizados

Entradas del usuario:

- txtRadius: radio del círculo (entero positivo mayor a 0).
- checkBox1: indica si se debe aplicar la escala extra (escala normal cuando está desmarcado, escala mayor y color rojo cuando está marcado).

Parámetros internos importantes:

- cx, cy: coordenadas del centro del círculo, calculadas como mitad del ancho y alto del PictureBox.
- radius: radio validado a partir del contenido de txtRadius.
- x, y: coordenadas enteras de los puntos generados en el primer octante (x inicia en 0, y inicia en radius).
- p: valor de decisión del algoritmo del punto medio, inicializado como p = 1 − radius y actualizado en cada iteración del bucle while.
- SF: factor de escala definido dentro de PlotPoint; vale 1 cuando checkBox1 está desmarcado y 5 cuando está marcado.
- size: tamaño del cuadrito dibujado; es 3 píxeles sin escala y 4 píxeles con escala activada.

5.2.2.3 Explicación de las funciones

- btnDraw_Click
    - Limpia el PictureBox con picCanvas.Refresh, crea el objeto Graphics g y fija el centro del círculo en (cx, cy) usando el tamaño del PictureBox.
    - Valida el radio con int.TryParse y la condición radius > 0; si la validación falla, muestra un MessageBox con el texto “Por favor ingresa un radio mayor a 0 (solo enteros positivos).” y termina la ejecución.
    - Inicializa las variables del algoritmo de punto medio: x = 0, y = radius, p = 1 − radius.
    - Llama una vez a PlotPoint para dibujar el primer conjunto de puntos simétricos.

- Ejecuta un bucle while (x < y) donde:
    - Incrementa x en una unidad.
    - Si p < 0, actualiza p = p + 2*x* + *3; en caso contrario decrementa y y ajusta p = p + 2*(x − y) + 1, de acuerdo con las fórmulas clásicas del algoritmo de punto medio para decidir entre moverse en horizontal o en diagonal.
    - Llama a PlotPoint para dibujar los ocho puntos simétricos correspondientes a la nueva pareja (x, y).

- PlotPoint
    - Calcula SF según el estado de checkBox1 (1 sin escala, 5 con escala) y decide el color (negro o rojo) y el tamaño del punto (3 o 4 píxeles).
    - Dibuja ocho pequeños rectángulos (FillRectangle) en las posiciones:
        - (cx ± x**SF, cy* ± *y**SF)
        - (cx ± y**SF, cy* ± *x**SF)
          cubriendo así los ocho octantes del círculo a partir del punto calculado en el primer octante.
- btnLimpiar_Click
    - Limpia el PictureBox, borra el contenido de txtRadius y devuelve el foco al cuadro de texto para facilitar una nueva prueba.
- btnCerrar_Click
    - Cierra el formulario actual con this.Close().

5.2.3 Casos de prueba

1. El usuario deja el campo de radio vacío o escribe un valor no numérico, por ejemplo: txtRadius = "" o txtRadius = "abc".
   Resultado esperado: al presionar el botón Dibujar, int.TryParse falla, se muestra el mensaje “Por favor ingresa un radio mayor a 0 (solo enteros positivos).” y no se dibuja ningún punto en el PictureBox.

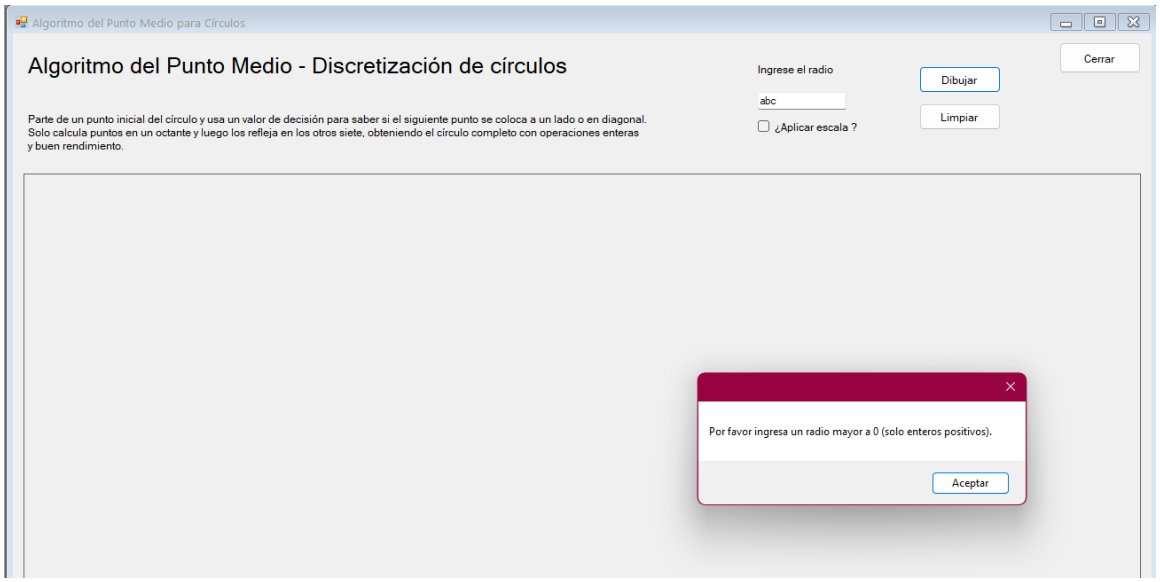


Ilustración 18. Caso de prueba 1 - Algoritmo del Punto Medio

2. El usuario corrige el texto, pero usa un valor no válido, por ejemplo: txtRadius = "0" o txtRadius = "-10".
   Resultado esperado: la condición radius <= 0 se cumple, se vuelve a mostrar el mismo mensaje de validación, el lienzo permanece vacío y el programa continúa funcionando sin cierres inesperados, confirmando que no se aceptan radios nulos o negativos.

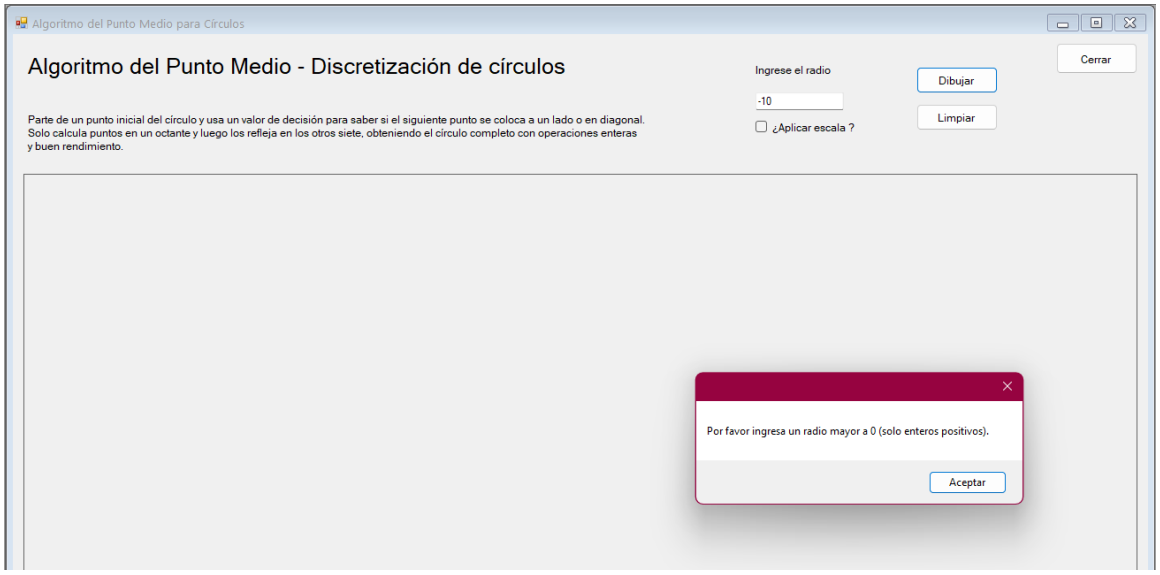


Ilustración 19. Caso de prueba 2 - Algoritmo del Punto Medio

3. El usuario ingresa un radio válido, por ejemplo: txtRadius = "20", con checkBox1 desmarcado.
   Resultado esperado: al presionar Dibujar se ejecuta el algoritmo de punto medio; se genera una circunferencia centrada en el PictureBox, formada por pequeños

cuadrados negros en ocho octantes alrededor de (cx, cy), sin huecos visibles, evidenciando el uso correcto de la simetría.

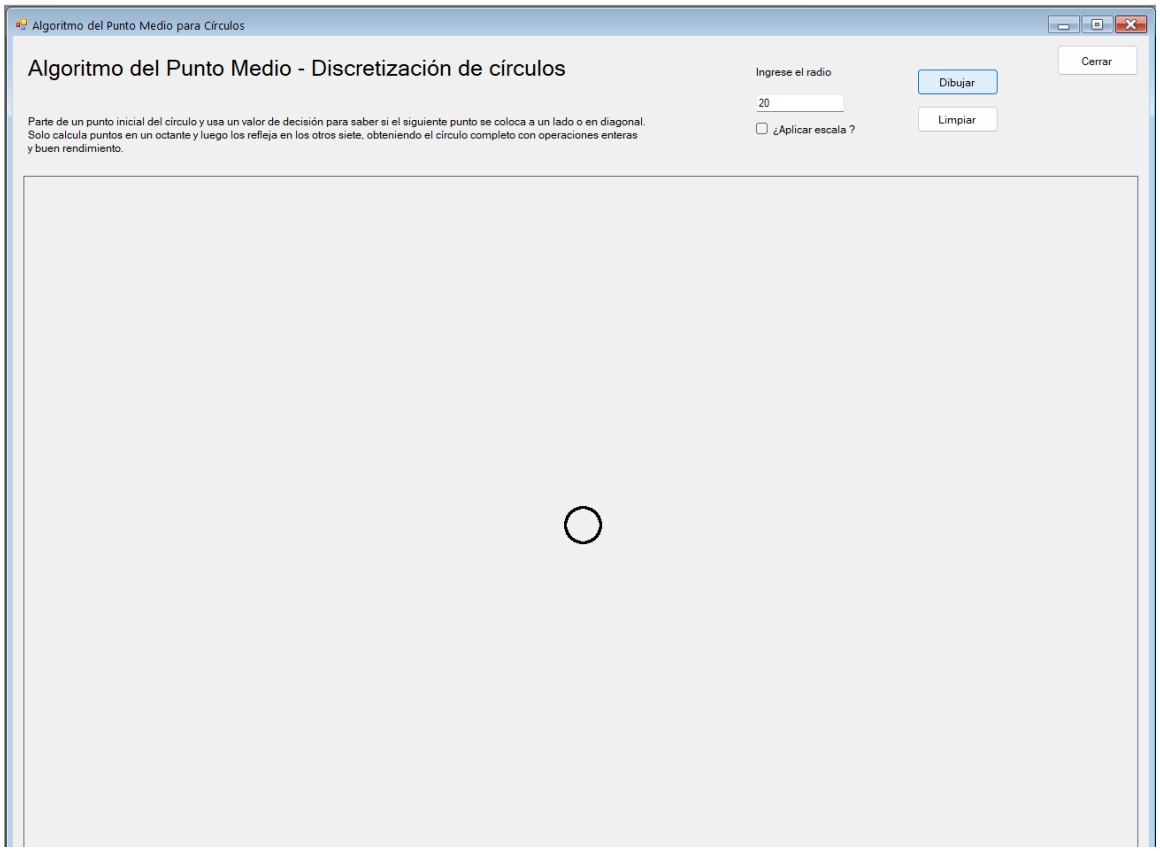


Ilustración 20. Caso de prueba 3 - Algoritmo del Punto Medio

4. El usuario marca checkBox1, limpia el lienzo con el botón Limpiar e ingresa nuevamente txtRadius = "20".
   Resultado esperado: al presionar Dibujar se dibuja nuevamente el círculo, ahora ampliado por el factor SF = 5 y en color rojo; la circunferencia sigue siendo uniforme y centrada, lo que valida que la función PlotPoint aplica correctamente la escala y el color sin alterar la lógica del algoritmo de punto medio.

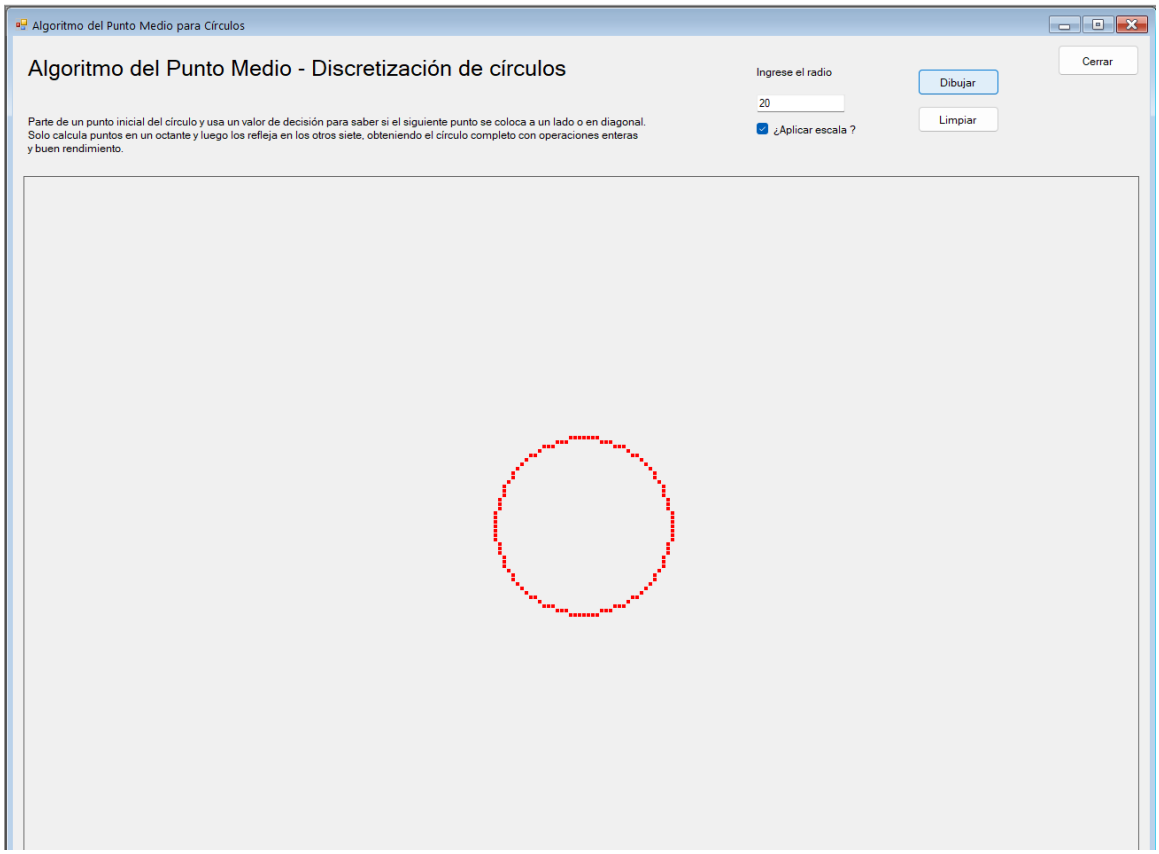


Ilustración 21. Caso de prueba 4 - Algoritmo del Punto Medio

5.3 Algoritmo de Bresenham para círculos

La aplicación explota explícitamente la simetría de 8 octantes del círculo, generando los puntos:
$(xc \pm x, yc \pm y)$ y $(xc \pm y, yc \pm x)$, a partir de los puntos calculados en el primer octante. Este enfoque es una variación práctica del algoritmo de punto medio, que garantiza un círculo completo y uniforme aprovechando la simetría (Simple Snippets, 2025).

5.3.1 Justificación de la variante del algoritmo

El algoritmo de Bresenham para círculos genera los puntos de la circunferencia usando solo aritmética entera y un criterio de decisión incremental, explotando la simetría de ocho octantes para obtener el círculo completo a partir de un solo octante. (GeeksforGeeks, 2025)
Se eligió esta variante porque:

- Es muy eficiente: evita funciones trigonométricas y raíces cuadradas, utilizando únicamente sumas y restas enteras con un parámetro de decisión.
- Usa explícitamente la simetría de 8 octantes, lo que permite explicar de forma clara cómo se reduce el trabajo de cálculo en la rasterización de círculos.

- Es uno de los algoritmos estándar de referencia para el trazado de círculos en computación gráfica y se presenta habitualmente junto al algoritmo de línea de Bresenham.

5.3.2 Descripción del Formulario

5.3.2.1 Diseño del formulario

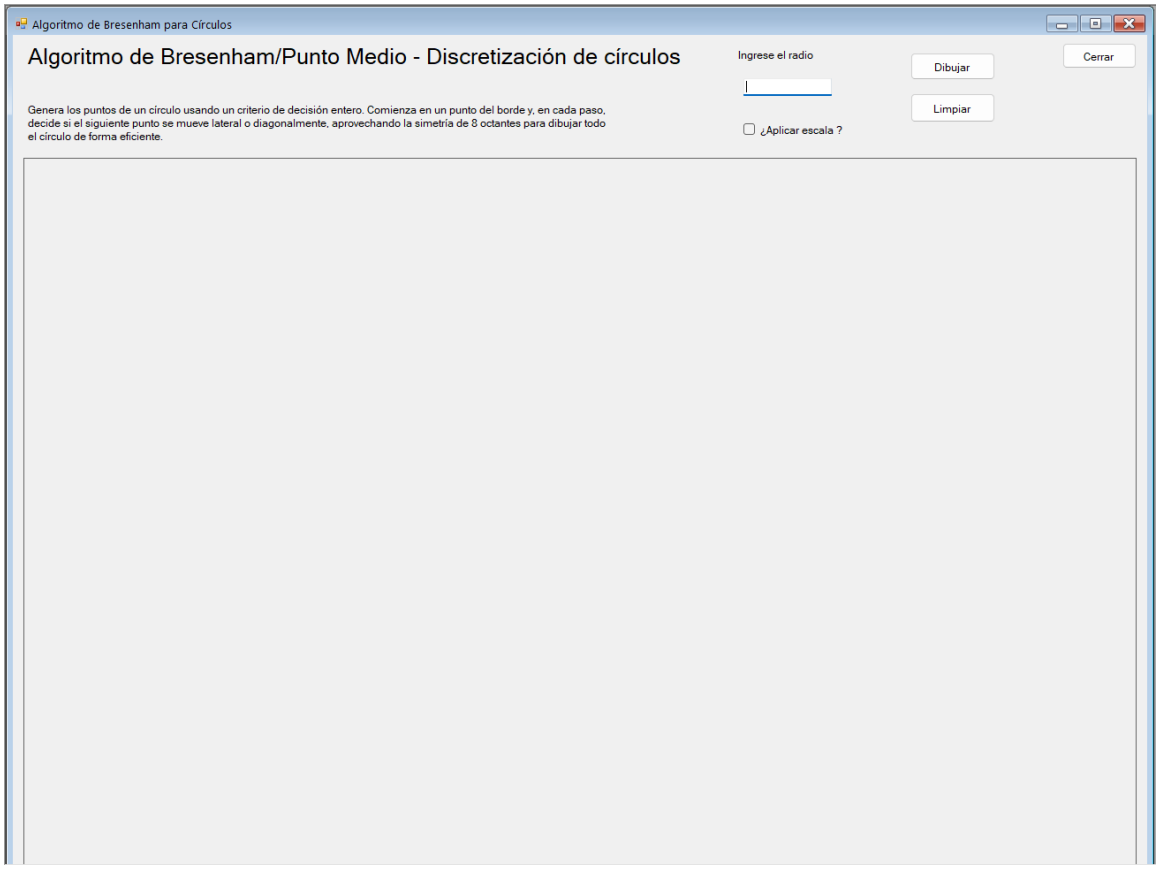


Ilustración 22. Formulario para el Algoritmo de Bresenham para discretización de círculos

El formulario "Algoritmo de Bresenham/Punto Medio – Discretización de círculos" muestra el título y una breve descripción en la parte superior. A la derecha se encuentra un cuadro de texto para ingresar el radio, un check box para activar la escala adicional (Aplicar escala), y los botones Dibujar, Limpiar y Cerrar. En la zona central inferior se dispone un PictureBox amplio que funciona como lienzo; allí se dibuja la circunferencia mediante pequeños cuadrados negros o rojos que representan los puntos generados por el algoritmo.

5.3.2.2 Parámetros utilizados

Entradas del usuario:

- txtRadius: radio del círculo (entero positivo mayor que 0).

- checkBox1: indica si se aplica o no una escala de visualización mayor (desmarcado: escala normal; marcado: escala aumentada y color rojo).

Parámetros internos principales:

- cx, cy: coordenadas del centro del círculo, calculadas como la mitad del ancho y del alto del PictureBox.
- radius: valor entero validado a partir de txtRadius.
- x, y: coordenadas enteras del punto que se recorre en el primer octante (x inicia en 0, y inicia en radius).
- d: parámetro de decisión del algoritmo de Bresenham para círculos, inicializado como d = 3 − 2*radius y actualizado en cada iteración según el resultado del criterio.
- SF: factor de escala definido dentro de Dibujar8Octantes (1 sin escala extra, 5 con escala).
- size: tamaño del cuadrito que representa cada punto (2 píxeles sin escala, 4 píxeles con escala).

5.3.2.3 Explicación de las funciones

- btnDraw_Click
    - Limpia el contenido del PictureBox con picCanvas.Refresh, crea el objeto Graphics g y calcula el centro del círculo (cx, cy) usando las dimensiones del PictureBox.
    - Valida el radio usando int.TryParse y comprobando que radius > 0; si la validación falla, muestra un MessageBox con el mensaje “Por favor ingresa un radio mayor a 0 (solo enteros positivos).” y detiene la ejecución.
    - Inicializa las variables del algoritmo: x = 0, y = radius, d = 3 − 2*radius.
    - Ejecuta un bucle while (x <= y) en el que:
        - Llama a Dibujar8Octantes para pintar los ocho puntos simétricos correspondientes a la pareja (x, y).
        - Evalúa el parámetro de decisión d: si d < 0, lo actualiza como d = d + *4x + 6; en caso contrario, decrementa y y actualiza d = d + 4*(x − y) + 10, de acuerdo con las fórmulas clásicas del algoritmo de círculo de Bresenham.
        - Incrementa x en una unidad al final de cada iteración.

- Dibujar8Octantes
    - Determina SF (1 o 5), el color del pincel (negro o rojo) y el tamaño del punto según el estado de checkBox1.
    - Dibuja ocho pequeños rectángulos (FillRectangle) en las posiciones:
        - (cx ± x*SF, cy* ± *y*SF)
        - (cx ± y*SF, cy* ± *x*SF) con lo que se obtienen simultáneamente los puntos de los ocho octantes del círculo a partir del punto del primer octante.
- btnLimpiar_Click
    - Limpia el PictureBox, borra el contenido del cuadro txtRadius y devuelve el foco a ese control para facilitar nuevas pruebas.
- btnCerrar_Click
    - Cierra el formulario con this.Close().

5.3.3 Casos de prueba

1. El usuario deja el campo de radio vacío o escribe un valor no numérico, por ejemplo: txtRadius = "" o txtRadius = "abc".
   Resultado esperado: al presionar el botón Dibujar, la validación con int.TryParse falla, se muestra el mensaje “Por favor ingresa un radio mayor a 0 (solo enteros positivos).” y no se dibuja ningún punto en el PictureBox.

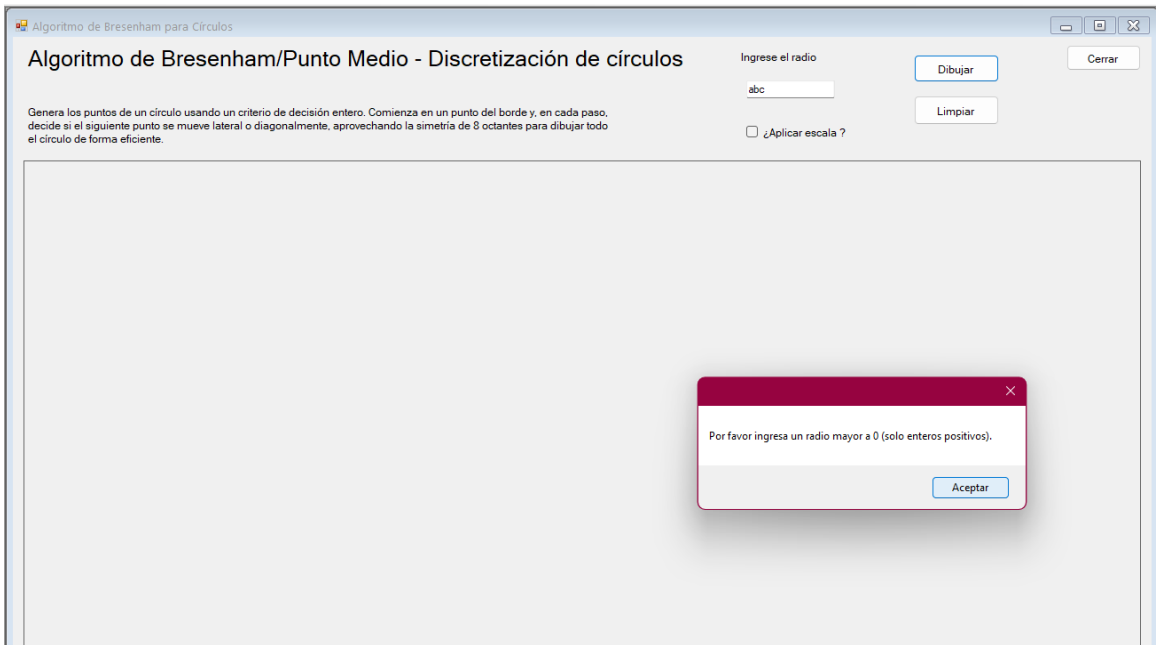


Ilustración 23. Caso de prueba 1 - Algoritmo de Bresenham para Círculos

2. El usuario corrige el texto, pero ingresa un valor no válido, por ejemplo: txtRadius = "0" o txtRadius = "-12".
   Resultado esperado: la condición radius <= 0 vuelve a cumplirse, se muestra el mismo mensaje de validación y el lienzo permanece vacío; el programa continúa funcionando sin errores, comprobando que no se aceptan radios nulos o negativos.

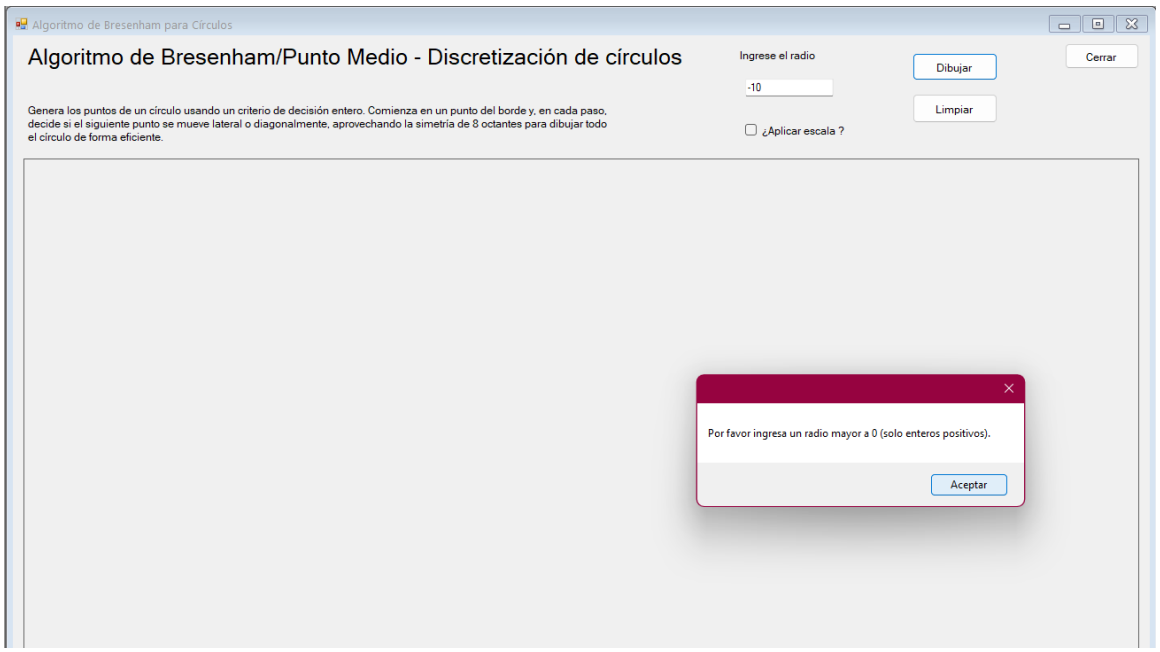


Ilustración 24. Caso de prueba 2 - Algoritmo de Bresenham para círculos.

3. El usuario introduce un radio válido, por ejemplo: txtRadius = "40", con checkBox1 desmarcado.

   Resultado esperado: al presionar Dibujar se ejecuta el algoritmo de Bresenham; en el PictureBox aparece una circunferencia centrada en (cx, cy), formada por pequeños cuadrados negros que se distribuyen en los ocho octantes del círculo, sin saltos visibles y con una forma prácticamente circular.

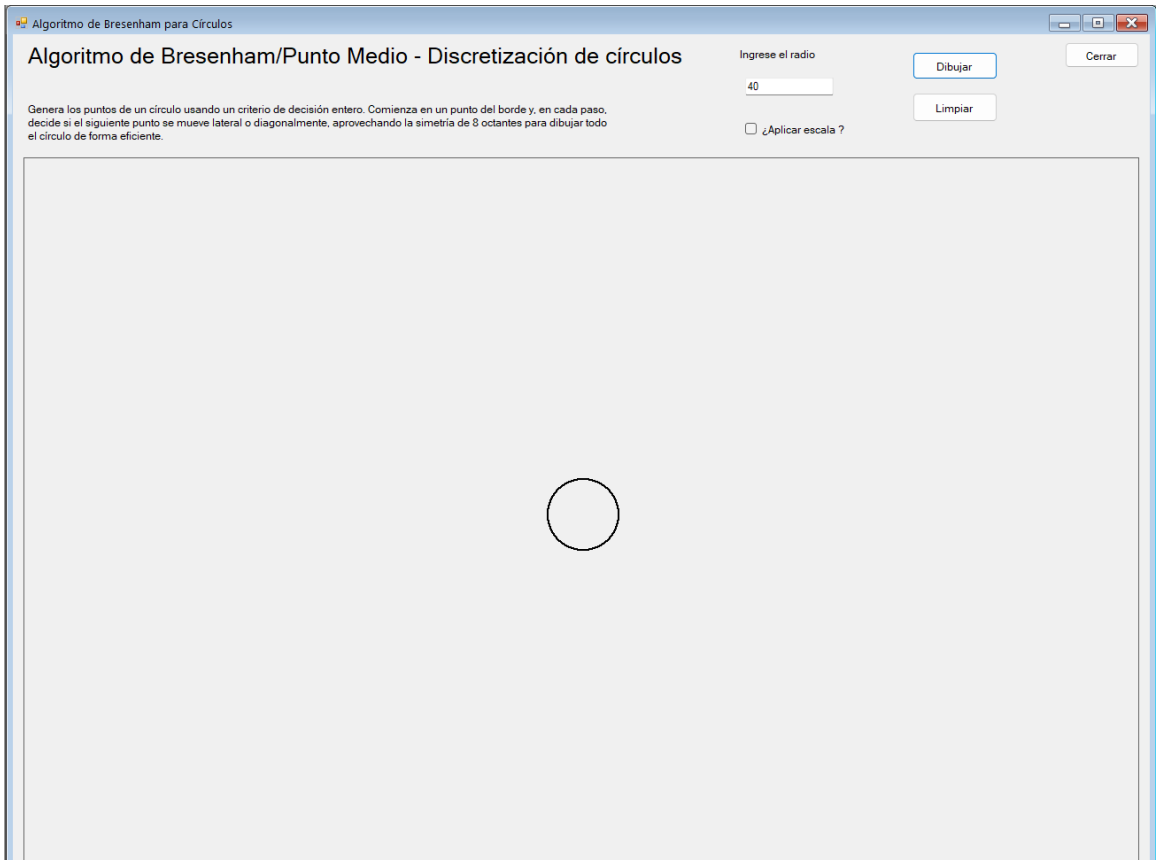


Ilustración 25. Caso de prueba 4 - Algoritmo de Bresenham para círculos.

4. Sin cerrar el formulario, el usuario marca checkBox1, limpia el lienzo con el botón Limpiar e ingresa nuevamente txtRadius = "40".

   Resultado esperado: al presionar Dibujar se dibuja otra circunferencia en el mismo centro, ahora ampliada por el factor SF = 5 y en color rojo; sigue siendo uniforme y completa, lo que confirma que la función Dibujar8Octantes aplica correctamente la escala y el color sin alterar la lógica del algoritmo de Bresenham para círculos.

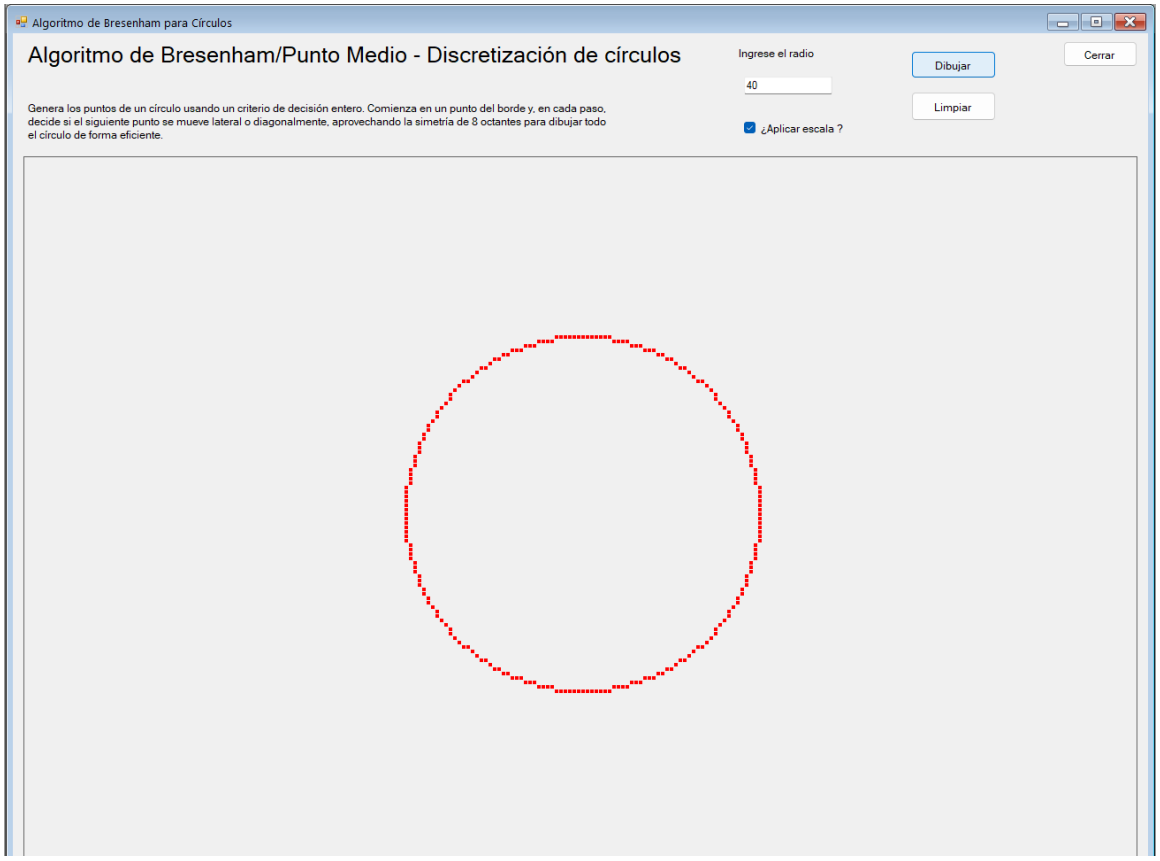


Ilustración 26. Caso de prueba 4 - Algoritmo de Bresenham para círculos

6. Algoritmos de relleno

Para el relleno de regiones y polígonos se implementan algoritmos de tipo flood fill de 4 vecinos, boundary fill de 4 vecinos y un algoritmo de relleno por líneas de exploración (scanline fill). Estos algoritmos permiten rellenar áreas delimitadas en el lienzo.

6.1 Flood fill

El algoritmo flood fill de 4 vecinos rellena una región conectada a partir de un píxel semilla, propagándose a los píxeles que comparten lado con el píxel actual (arriba, abajo, izquierda y derecha). La condición de relleno se basa en el color original o en el color de frontera, según se utilice un enfoque de seed fill o boundary fill (GeeksforGeeks, 2025)

6.1.1 Justificación de la variante del algoritmo

El algoritmo Flood Fill (o seed fill) se eligió como primera variante de relleno porque es el método clásico utilizado en herramientas de “bote de pintura” de los editores de imágenes: a partir de un punto semilla, reemplaza el color de ese píxel y de todos los píxeles conectados que tienen el mismo color, propagándose hasta encontrar una frontera diferente.

En este proyecto se implementa una versión iterativa con cola de píxeles y conectividad de

4 vecinos, lo que evita problemas de desbordamiento de pila de las versiones recursivas y permite animar el proceso de relleno de manera eficiente y visualmente clara. (GeeksforGeeks, 2025)

6.1.2 Descripción del Formulario

6.1.2.1 Diseño del formulario

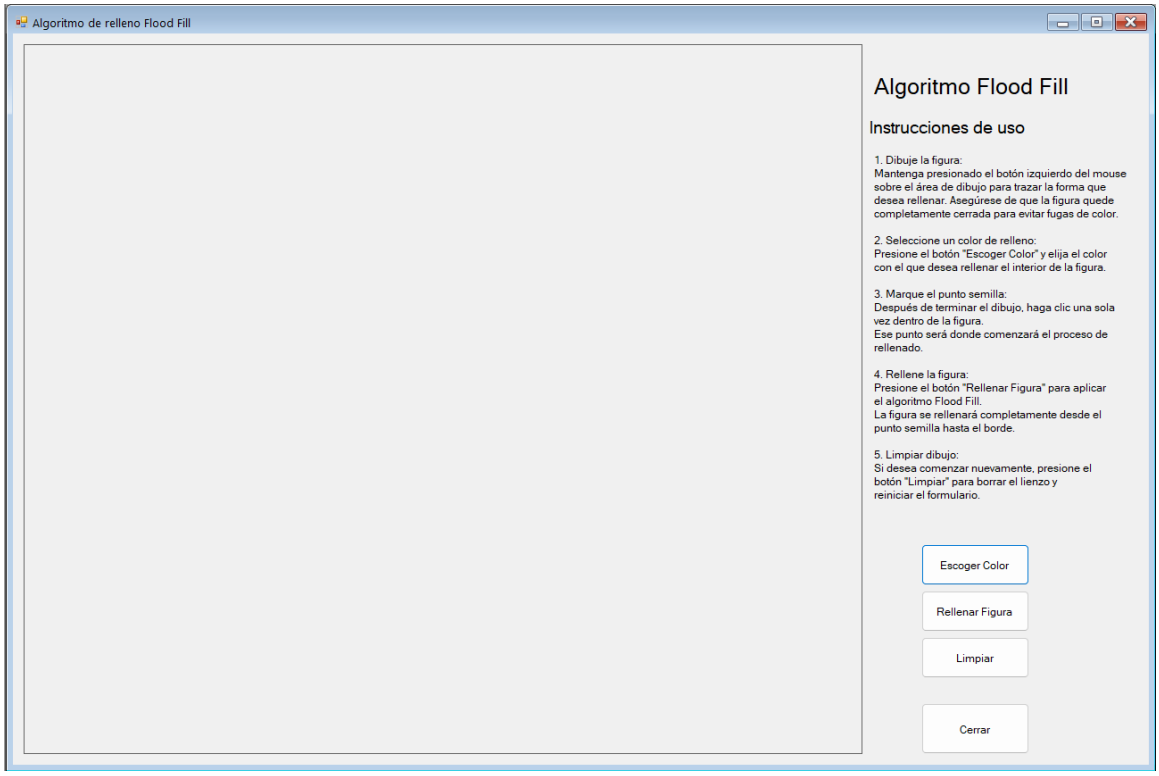


Ilustración 27. Formulario para el Algoritmo de Relleno Flood Fill.

El formulario “Algoritmo de relleno Flood Fill” está dividido en dos zonas. A la izquierda se encuentra un PictureBox grande que funciona como lienzo; el usuario dibuja la figura arrastrando el mouse sobre esta área. A la derecha se presenta un panel con el título del algoritmo y una lista de instrucciones de uso numeradas, que explican los pasos: dibujar la figura, escoger un color, seleccionar el punto semilla y rellenar. Debajo de las instrucciones se ubican los botones “Escoger Color”, “Rellenar Figura”, “Limpiar” y “Cerrar”, que controlan el flujo de la interacción.

6.1.2.2 Parámetros utilizados

Entradas y parámetros relevantes:

- colorRelleno: color elegido por el usuario mediante el cuadro de diálogo de color, usado como color de relleno.
- puntoSemilla: coordenadas del píxel dentro de la figura donde comenzará el proceso de Flood Fill.

- lienzo: bitmap asociado al PictureBox, que almacena el dibujo en memoria y sobre el cual se leen y escriben los píxeles.
- colaPixeles: cola (Queue<Point>) que contiene los píxeles pendientes por procesar durante el relleno.
- colorObjetivoAnim: color original en el área que se va a reemplazar (color de fondo o interior de la figura).
- colorRellenoAnim: color que se utiliza para pintar los píxeles durante la animación.
- timerFlood: temporizador que controla cuántos píxeles se rellenan en cada tick para animar visualmente el algoritmo.
- pixelsPorTick: cantidad de píxeles procesados por cada tick del timer (en este caso 500).

6.1.2.3 Explicación de las funciones

- Constructor frmFloodFill
  - Inicializa el bitmap lienzo con el tamaño del PictureBox y lo asigna a picCanvas.Image.
  - Configura el timer timerFlood (intervalo, evento Tick) e inicializa la colaPixeles.
- picCanvas_MouseDown, picCanvas_MouseMove, picCanvas_MouseUp
  - Permiten dibujar la figura con el mouse cuando no hay animación en curso (animando = false).
  - En MouseMove se trazan líneas negras de grosor 4 entre el punto anterior y la posición actual, utilizando Graphics.FromImage sobre lienzo.
  - En MouseUp, si el usuario solo hizo clic (sin arrastrar), se interpreta como selección del puntoSemilla y se muestra un mensaje indicando la coordenada elegida.
- btnPickColor_Click
  - Abre un ColorDialog y, si el usuario confirma, guarda el color en colorRelleno y actualiza el fondo del botón para mostrar el color seleccionado.
- btnFoodFill_Click
  - Verifica que se haya seleccionado un color de relleno (colorRelleno distinto de Color.Empty); si no, muestra un mensaje.
  - Verifica que exista un puntoSemilla; si no se ha hecho clic dentro de la figura, muestra un mensaje de advertencia.
  - Obtiene el color objetivo en la posición del punto semilla (colorObjetivoAnim = lienzo.GetPixel(x, y)).
  - Si el color objetivo es negro (borde), muestra un mensaje indicando que el punto debe estar dentro de la figura y cancela el relleno.

    - Inicializa la colaPixeles con el punto semilla, guarda colorObjetivoAnim y colorRellenoAnim, marca animando = true, deshabilita el botón de relleno y arranca el timerFlood.
- timerFlood_Tick
    - En cada tick procesa hasta pixelsPorTick píxeles de la cola:
        - Extrae un punto, comprueba que esté dentro de los límites del bitmap.
        - Revisa si el color actual coincide con colorObjetivoAnim; si no, lo descarta.
        - Si coincide, pinta el píxel con colorRellenoAnim y encola sus cuatro vecinos (derecha, izquierda, arriba, abajo), implementando Flood Fill de 4 vecinos.
        - Actualiza la pantalla con picCanvas.Invalidate.
        - Cuando la cola queda vacía, detiene el timer, marca animando = false y vuelve a habilitar el botón de relleno.
- btnClean_Click
    - Detiene el timer, limpia la colaPixeles, reinicia puntoSemilla y crea un nuevo bitmap vacío para lienzo, asignándolo al PictureBox.
- btnClose_Click
    - Cierra el formulario.

6.1.3 Casos de prueba

1. El usuario abre el formulario y, sin escoger color, dibuja una figura cerrada (por ejemplo, un rectángulo) manteniendo presionado el botón izquierdo del mouse dentro del PictureBox. Luego suelta el mouse y pulsa el botón “Rellenar Figura”.
   Resultado esperado: aparece un mensaje indicando que primero debe seleccionar un color de relleno; no se inicia la animación de Flood Fill y el dibujo permanece sin cambios.

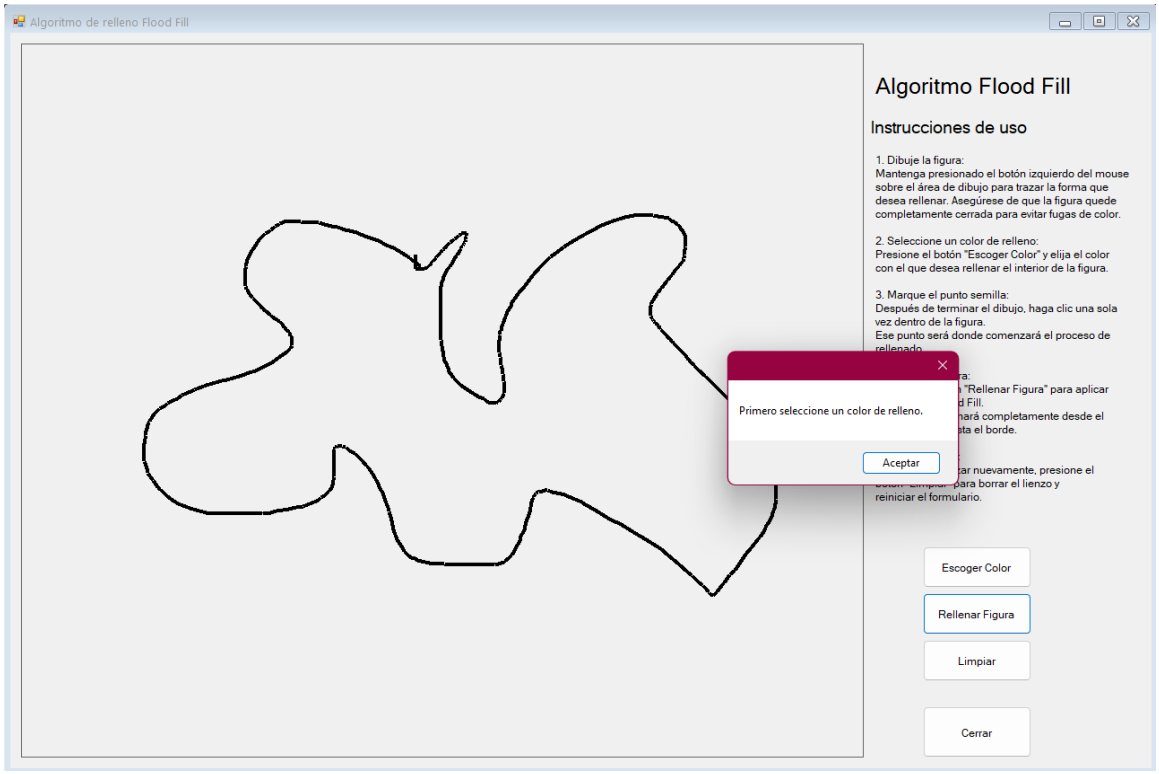


Ilustración 28. Caso de prueba 1 - Algoritmo Flood Fill

2. El usuario pulsa “Escoger Color”, elige un color en el cuadro de diálogo (por ejemplo, azul) y vuelve al formulario. A continuación, hace un clic sencillo sobre el borde negro de la figura y luego pulsa “Rellenar Figura”.
   Resultado esperado: el programa muestra un mensaje indicando que el punto debe estar dentro de la figura (no en el borde), porque el color objetivo coincide con el color del contorno; la animación no comienza y la figura continúa sin rellenarse.

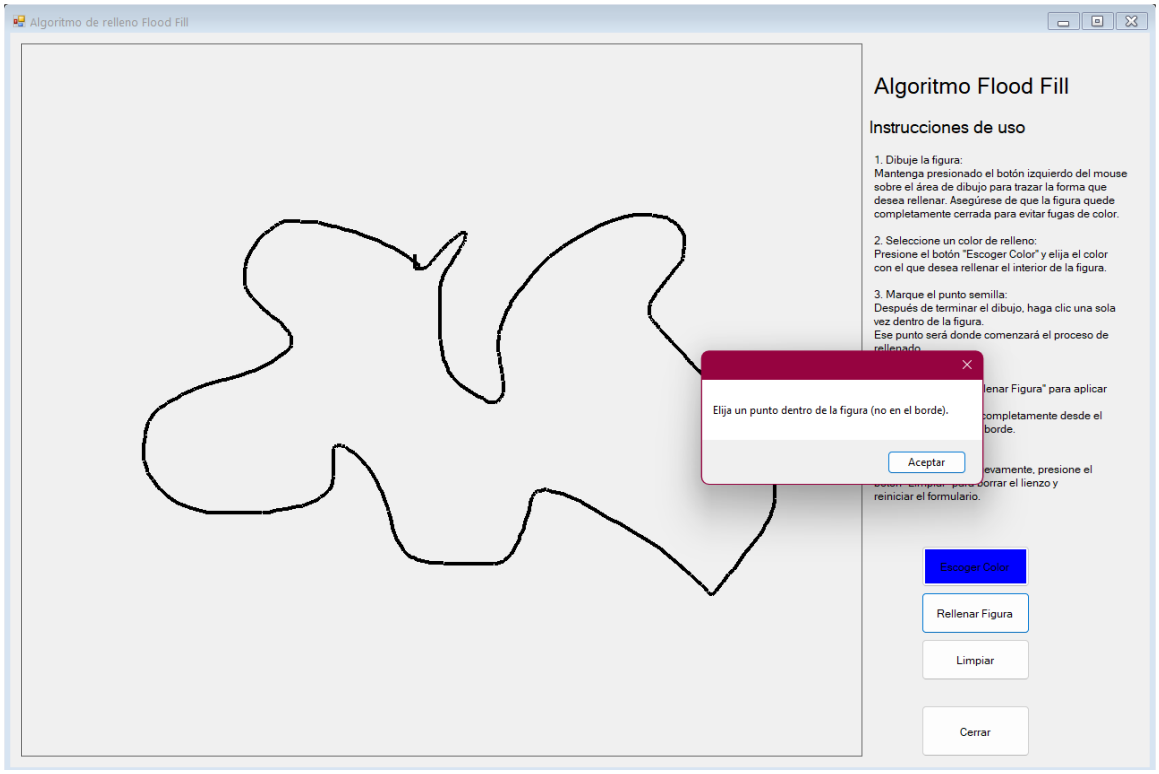


Ilustración 29. Caso de prueba 2 - Algoritmo Flood Fill

3. El usuario realiza un clic sencillo dentro del área cerrada de la figura (no en el borde), seleccionando correctamente el punto semilla, y vuelve a pulsar “Rellenar Figura”.
   Resultado esperado: se muestra un mensaje indicando la posición del punto semilla cuando se selecciona, y al pulsar “Rellenar Figura” el algoritmo inicia la animación: se observa cómo el color elegido comienza a propagarse desde el punto semilla hacia todas las direcciones, rellenando gradualmente el interior de la figura hasta llegar al contorno negro. Durante la animación el botón de relleno permanece deshabilitado y el usuario no puede dibujar nuevas líneas, evitando inconsistencias. Al finalizar, el área interior está completamente cubierta por el color de relleno, sin salirse de la figura ni producir cierres inesperados, lo que confirma el correcto funcionamiento del Flood Fill iterativo con 4 vecinos y las validaciones de entrada.

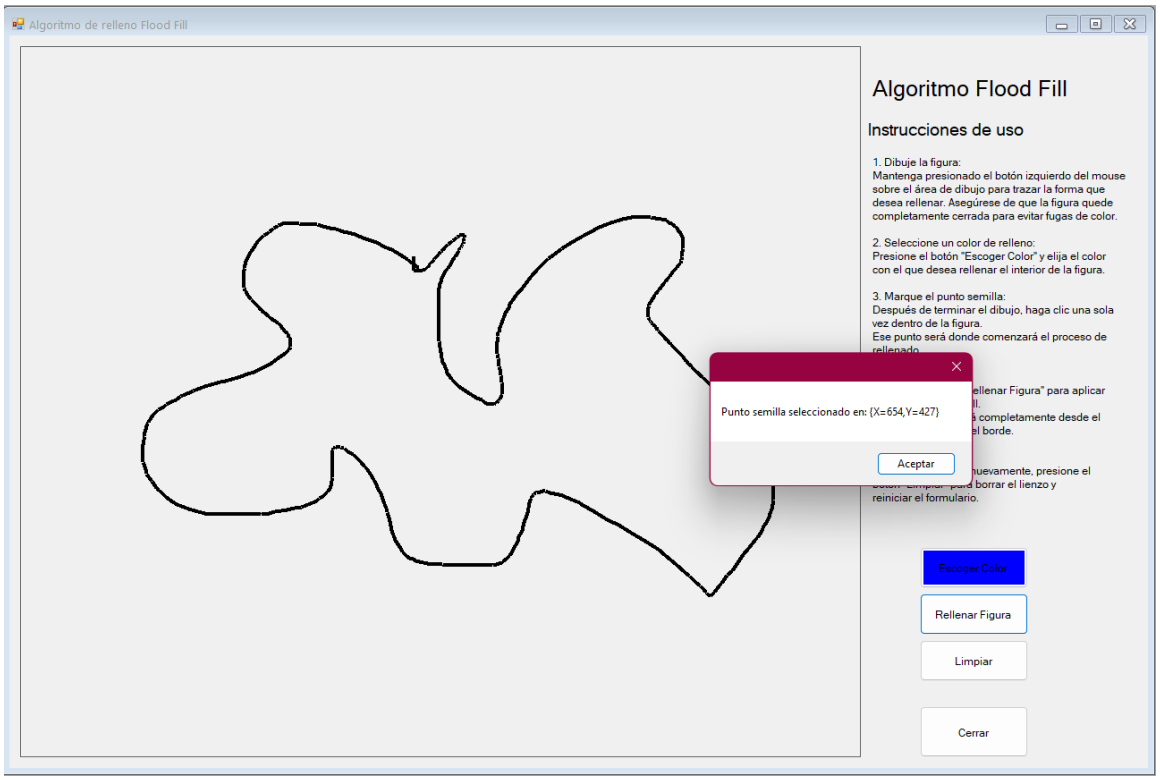


Ilustración 30. Caso de prueba 3 -1 - Algoritmo Flood Fill

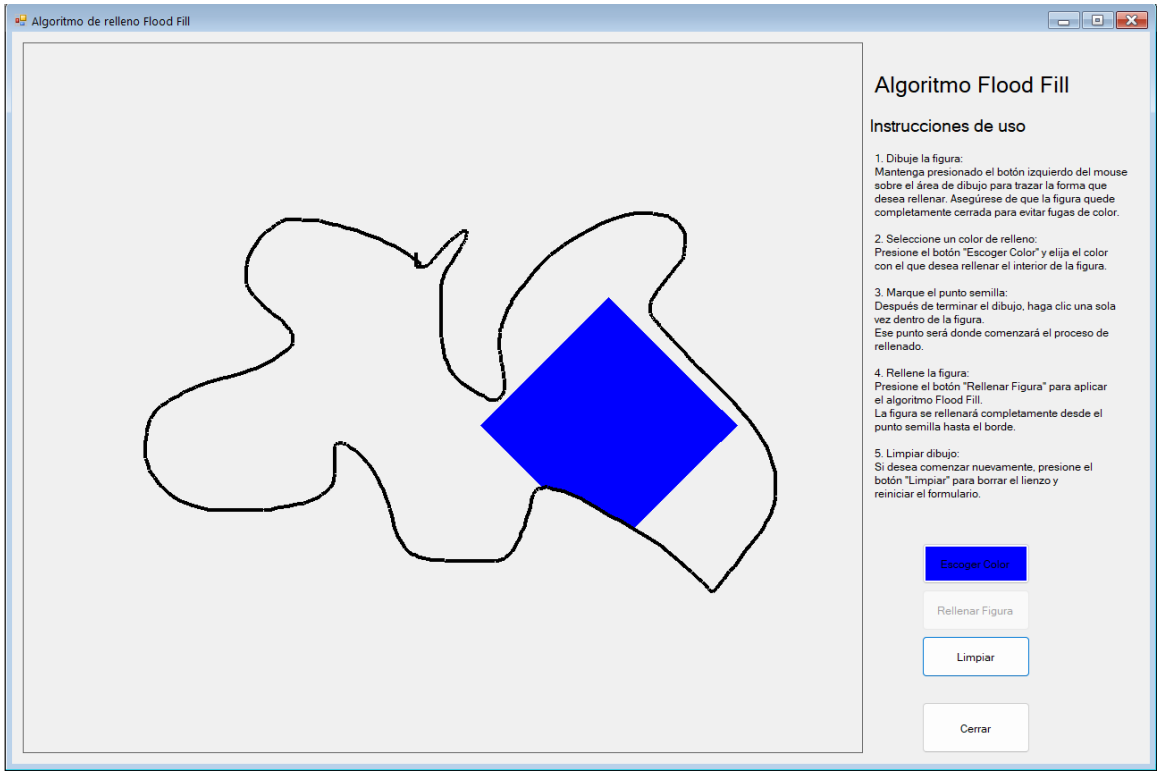


Ilustración 31. Caso de prueba 3-2 - Algoritmo Flood Fill

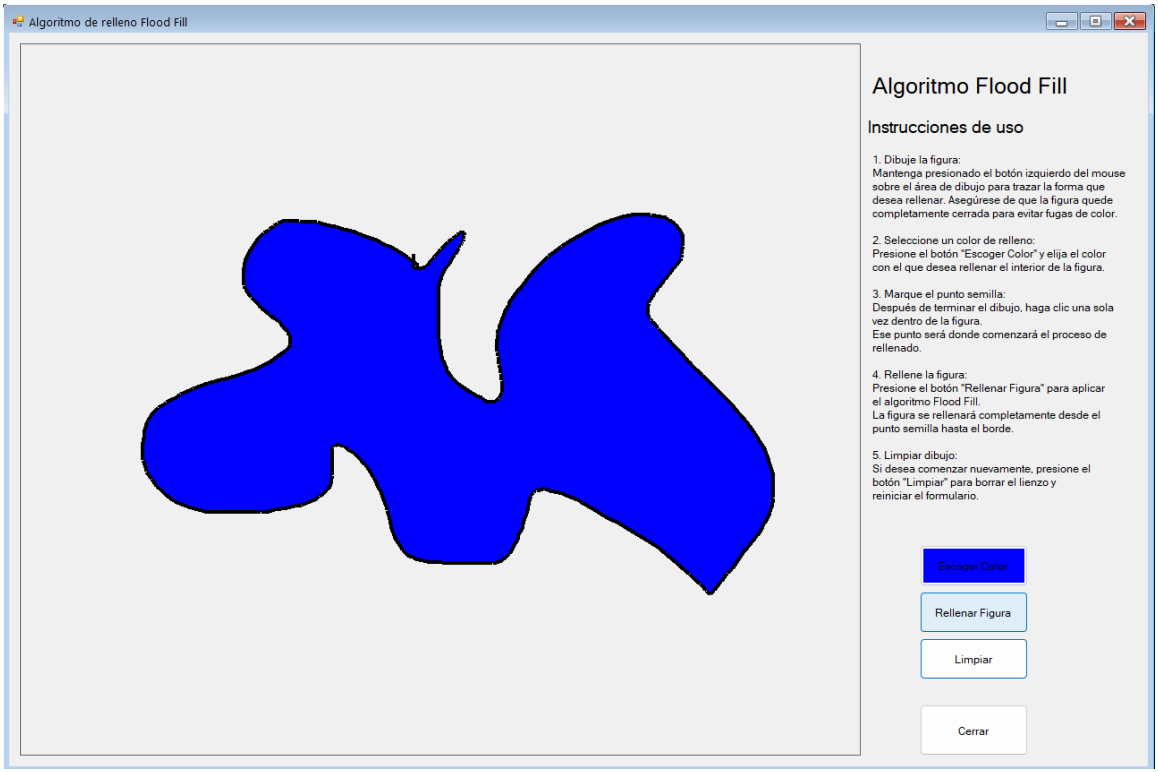


Ilustración 32. Caso de prueba 3-3 - Algoritmo Flood Fill

Ilustración 33. Caso de prueba - Algoritmo Flood Fill

6.2 Boundary Fill

En la variante de 8 vecinos, además de los píxeles adyacentes por lado, se consideran las diagonales. Con ello se consigue un relleno más completo en regiones donde los contornos o conexiones son diagonales, reduciendo la posibilidad de dejar huecos (*Algoritmo De Relleno De Límites En Gráficos Por Computadora*, n.d.)

6.2.1 Justificación de la variante del algoritmo

El algoritmo Boundary Fill es otro algoritmo clásico de relleno por semilla. A diferencia de Flood Fill, que reemplaza un color interior concreto, Boundary Fill parte de un punto interior y expande el relleno hasta encontrar un color de borde definido, deteniéndose exactamente en la frontera de la figura.

En la literatura se describe como un algoritmo adecuado para regiones delimitadas por un contorno de un solo color, pudiendo implementarse con conectividad de 4 u 8 vecinos y mediante recursión o estructuras iterativas (pila o cola).

En este proyecto se eligió una versión iterativa con pila y conectividad de 4 vecinos porque:

- Evita los problemas de desbordamiento de pila de la versión recursiva.

- Permite animar el relleno paso a paso y comparar su comportamiento frente a Flood Fill, mostrando claramente la diferencia entre algoritmos controlados por color de borde y por color interior.

6.2.2 Descripción del Formulario

6.2.2.1 Diseño del formulario

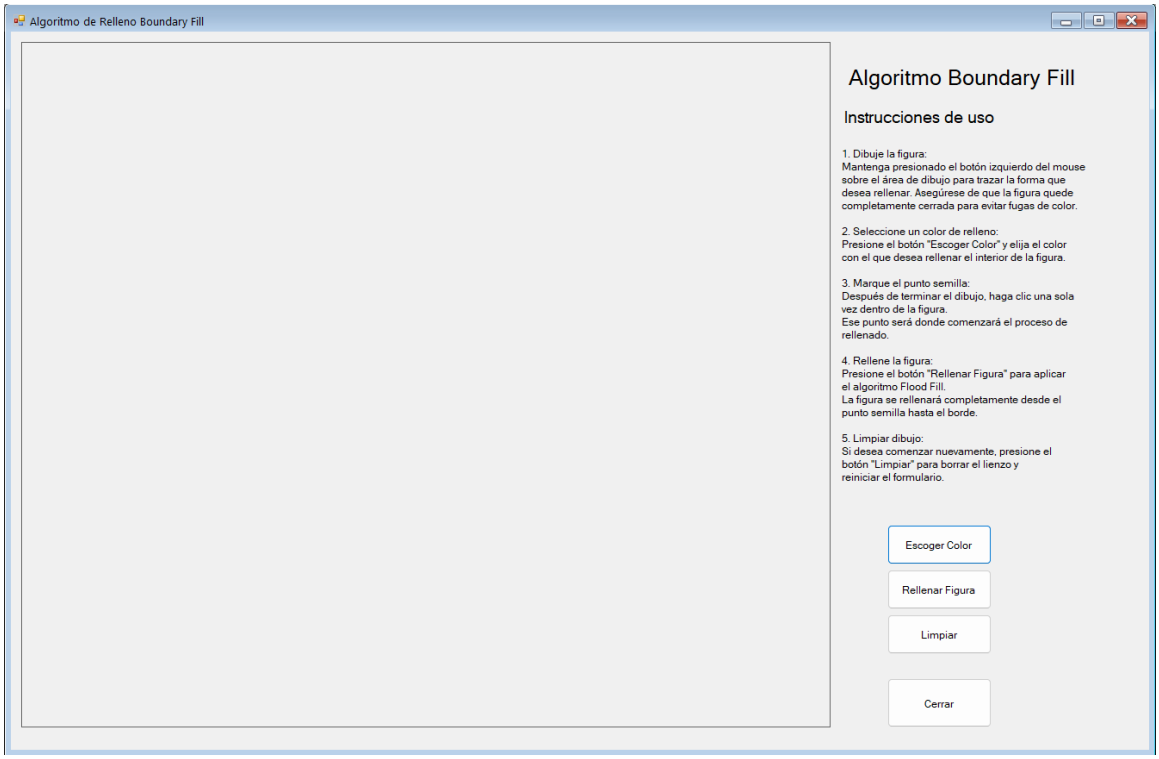


Ilustración 34. Formulario para el algoritmo de relleno Boundary Fill.

El formulario “Algoritmo Boundary Fill” tiene la misma organización general que el de Flood Fill:

- A la izquierda se encuentra un PictureBox grande que sirve como lienzo para dibujar la figura con el mouse.
- A la derecha se muestra el título del algoritmo y un bloque de “Instrucciones de uso” numeradas (dibujar la figura, escoger color, seleccionar punto semilla, aplicar relleno y limpiar).
- En la parte inferior derecha están los botones:
    - “Escoger Color” para seleccionar el color de relleno.
    - “Rellenar Figura” para iniciar el algoritmo Boundary Fill animado.
    - “Limpiar” para borrar el lienzo y reiniciar.

- “Cerrar” para salir del formulario.

6.2.2.2 Parámetros utilizados

Principales parámetros y campos usados en el código:

- lienzo: bitmap asociado al PictureBox, donde se almacena el dibujo y el relleno.
- colorRelleno: color seleccionado por el usuario para rellenar el interior de la figura.
- colorBorde: color fijo del borde, definido como negro (Color.Black), que actúa como frontera para el algoritmo Boundary Fill.
- dibujando: bandera booleana que indica si el usuario está trazando la figura con el mouse.
- puntoAnterior: último punto del mouse durante el dibujo, usado para trazar líneas continuas.
- puntoSemilla: punto interior seleccionado por el usuario para iniciar el relleno.
- pilaAnimacion: pila de puntos pendientes de procesar durante el Boundary Fill iterativo.
- timerAnim: temporizador que avanza la animación del relleno.
- animando: bandera booleana que indica si la animación está en curso (para bloquear dibujo y botones).

6.2.2.3 Explicación de las funciones

- Constructor frmBoundaryFill
  - Crea el bitmap lienzo con el tamaño del PictureBox y lo asigna a picCanvas.Image.
  - Configura el timerAnim (intervalo de 10 ms y suscripción al evento TimerAnim_Tick).
- picCanvas_MouseDown, picCanvas_MouseMove, picCanvas_MouseUp
  - Controlan el dibujo del borde a mano alzada cuando animando es false.
  - MouseDown: activa dibujando y guarda puntoAnterior.
  - MouseMove: mientras dibujando sea true, dibuja líneas negras de grosor 4 entre puntoAnterior y la nueva posición, usando Graphics.FromImage sobre lienzo, y actualiza puntoAnterior.
  - MouseUp: desactiva dibujando; si el mouse no se movió (puntoAnterior == e.Location), se interpreta como selección de puntoSemilla, se dibuja una marca gris en la posición y se muestra un mensaje con la coordenada seleccionada.
- btnPickColor_Click

    - Abre el cuadro de diálogo de color; si el usuario confirma, guarda el color en colorRelleno y cambia el fondo del botón para mostrarlo.
- btnBoundaryFill_Click
    - Impide lanzar una nueva animación si animando es true.
    - Verifica que colorRelleno no sea Color.Empty; si no hay color elegido, muestra un mensaje.
    - Verifica que puntoSemilla no sea null; si no se ha seleccionado un punto interior, muestra un mensaje.
    - Comprueba que el punto semilla esté dentro del área del bitmap; si no, muestra un mensaje de error.
    - Lee el color en el punto semilla con lienzo.GetPixel(x, y); si coincide con colorBorde, muestra un mensaje indicando que el punto debe estar dentro de la figura, no sobre el borde.
    - Inicializa la pilaAnimacion con el punto semilla, marca animando = true, desactiva los botones de relleno, color y limpiar, y arranca timerAnim.
- TimerAnim_Tick
    - Si la pilaAnimacion está vacía, detiene el timer, marca animando = false y reactiva los botones.
    - Si hay puntos pendientes, procesa hasta pixelesPorTick (400) puntos por tick:
        - Extrae un punto de la pila, verifica que esté dentro de los límites del bitmap.
        - Obtiene el color actual del píxel.
        - Si el pixel es de colorBorde o ya está pintado con colorRelleno, lo descarta.
        - Si es interior, lo pinta con colorRelleno mediante lienzo.SetPixel(x, y, colorRelleno).
        - Inserta en la pila sus vecinos 4-conectados: derecha, izquierda, arriba y abajo, implementando Boundary Fill de 4 vecinos mediante una estructura iterativa en lugar de recursiva.
    - Finalmente llama a picCanvas.Invalidate para actualizar la imagen en pantalla.
    - btnClean_Click
- Si animando es true, no hace nada (se evita limpiar mientras se rellena).
    - Crea un nuevo bitmap en blanco para lienzo, lo asigna al PictureBox, reinicia puntoSemilla y refresca la pantalla.
- btnClose_Click
    - Cierra el formulario.

6.2.3 Casos de prueba

1. El usuario abre el formulario, dibuja una figura cerrada (por ejemplo, un círculo o un polígono irregular) manteniendo presionado el botón izquierdo del mouse sobre el PictureBox. Sin escoger aún un color, hace un clic dentro de la figura para seleccionar el punto semilla y luego pulsa el botón “Rellenar Figura”.
   Resultado esperado: el programa muestra un mensaje indicando que debe seleccionar primero un color de relleno; la animación no se inicia, la figura permanece sin rellenar y el formulario sigue operativo.

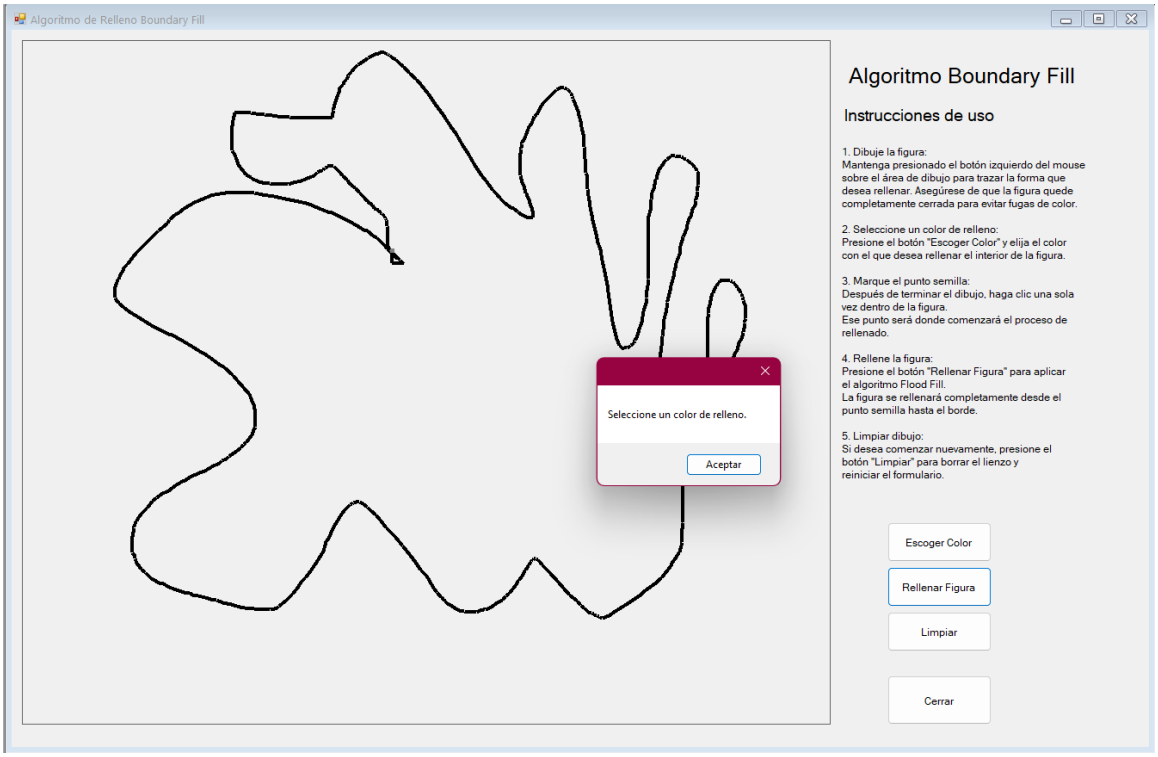


Ilustración 35. Caso de prueba 1 - Algoritmo Boundary Fill

2. El usuario pulsa “Escoger Color”, elige un color (por ejemplo, verde) y después hace clic dentro del área cerrada (píxeles de interior), y pulsa “Rellenar Figura”.
   Resultado esperado: se muestra el mensaje de confirmación del punto semilla cuando se selecciona; al pulsar “Rellenar Figura” se desactivan los botones de relleno, color y limpiar, y se inicia la animación: el color de relleno comienza a expandirse progresivamente desde el punto semilla hacia todas las direcciones, rellenando los píxeles interiores hasta encontrarse con el borde negro. La pintura se detiene exactamente en la frontera sin sobrepasarla, el PictureBox se actualiza continuamente y, al terminar, los botones se reactivan. El formulario no se cierra ni lanza excepciones, lo que valida el funcionamiento correcto del algoritmo Boundary Fill iterativo con 4 vecinos y las validaciones de color y punto semilla.

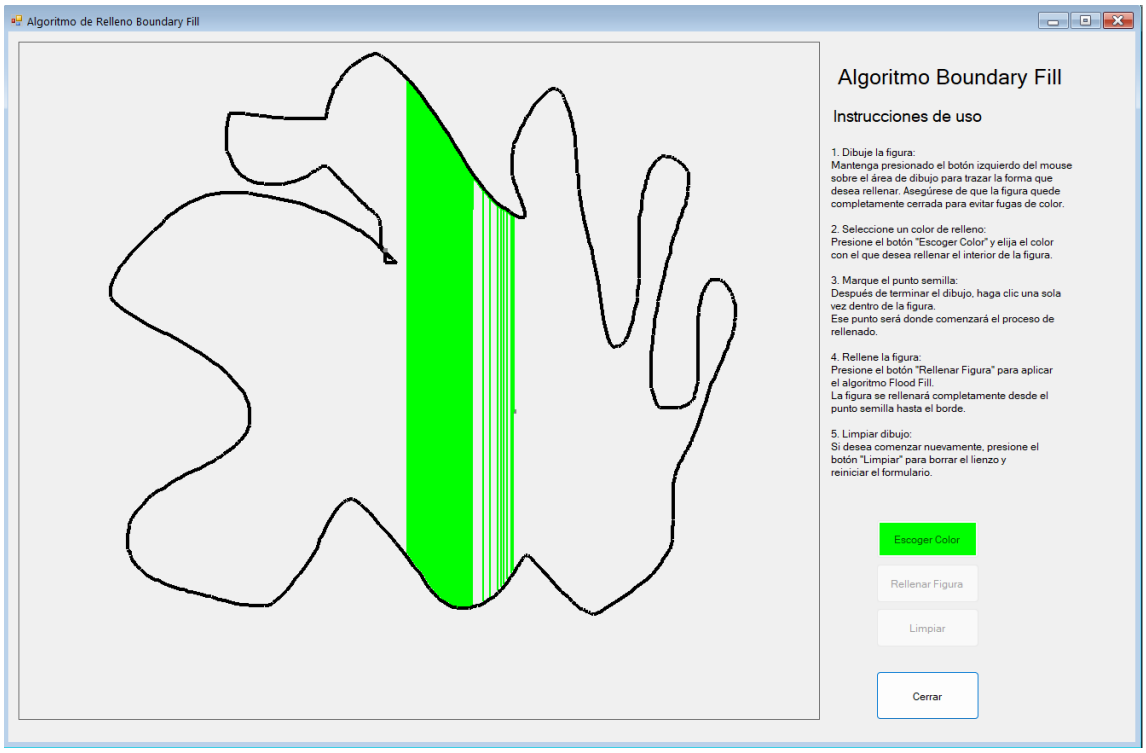


Ilustración 37. Caso de prueba 2-1 - Algoritmo Boundary Fill

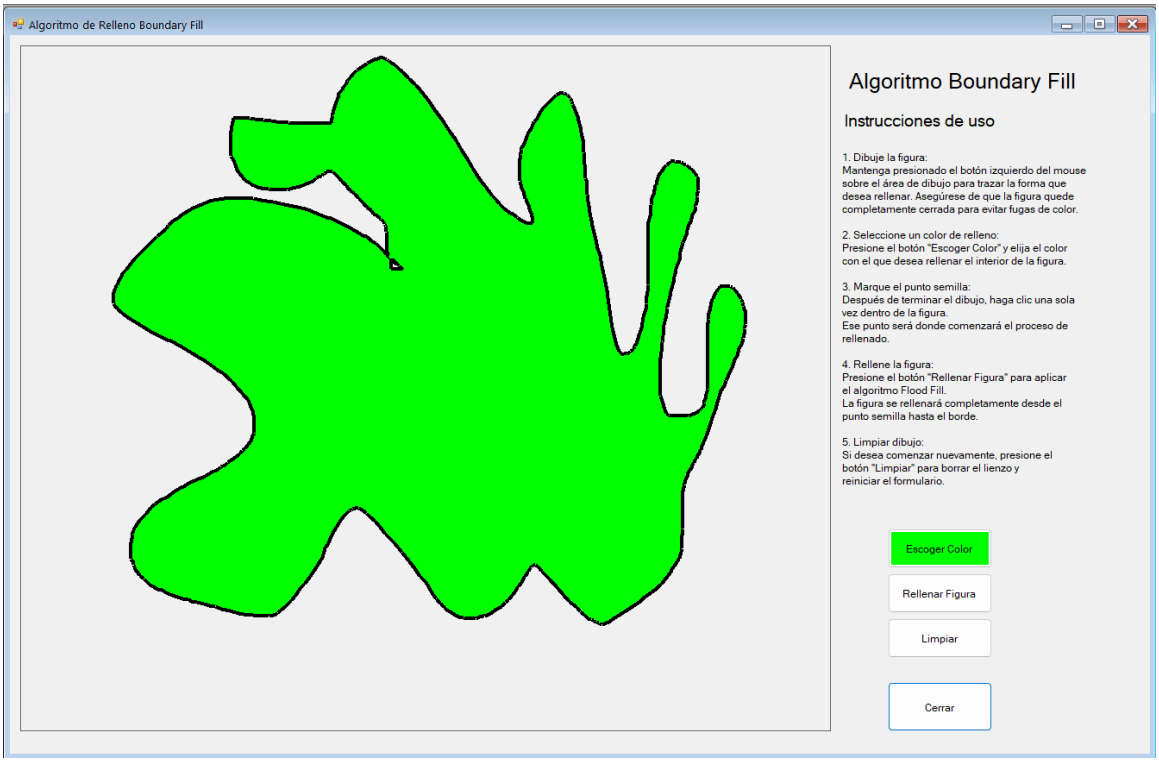


Ilustración 36. Caso de prueba 2-2 - Algoritmo Boundary Fill

6.3 ScanLine Fill

El algoritmo scanline fill rellena polígonos recorriendo el área línea por la línea de exploración horizontal (líneas de escaneo - ScanLine). Para cada scanline se calculan las intersecciones con las aristas del polígono, se ordenan por coordenada x y se rellenan los segmentos comprendidos entre pares de intersecciones (*Algoritmo De Línea De Escaneo Para El Relleno De Polígonos En Gráficos De Computadora*, n.d.).

Este enfoque es eficiente para polígonos grandes y permite un control preciso del interior del polígono.

6.3.1 Justificación de la variante del algoritmo

El algoritmo Scanline Fill se utiliza como tercera variante de relleno porque es el método clásico para rellenar polígonos a partir únicamente de la lista de vértices. La idea central es recorrer la figura línea por línea en el eje Y (scanlines), calcular las intersecciones de cada scanline con las aristas del polígono, ordenar esas intersecciones por su coordenada X y rellenar los píxeles comprendidos entre pares sucesivos de intersecciones. Este enfoque evita pruebas punto-en-polígono para cada píxel y permite rellenar de forma eficiente polígonos convexos y cóncavos, por lo que es ampliamente descrito en notas de curso y libros de computación gráfica como una de las técnicas estándar de “polygon filling”.

6.3.2 Descripción del Formulario

6.3.2.1 Diseño del formulario

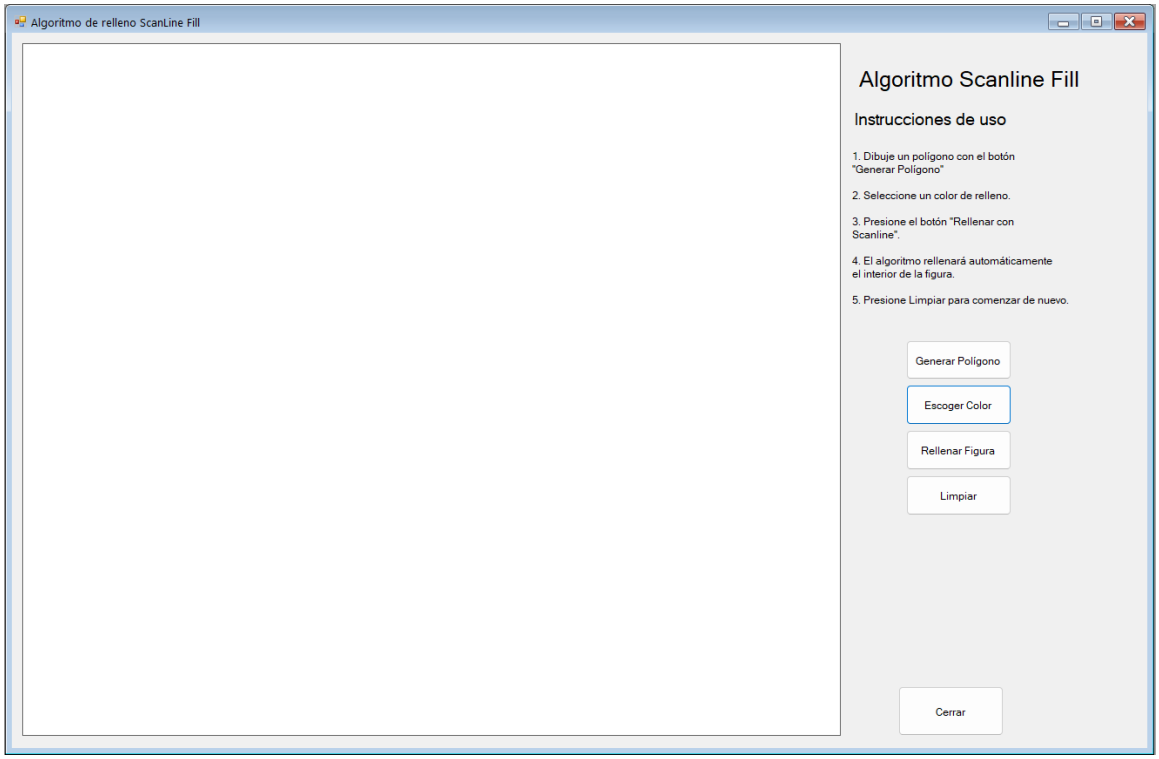


Ilustración 38. Formulario para el algoritmo de relleno ScanLine Fill

El formulario “Algoritmo Scanline Fill” está organizado en dos secciones.

- A la izquierda se encuentra un PictureBox grande, que sirve como lienzo donde se dibuja el polígono generado automáticamente.
- A la derecha se muestra el título del algoritmo y un bloque de “Instrucciones de uso” numeradas (generar polígono, escoger color, rellenar figura y limpiar).
- Debajo de las instrucciones se ubican los botones:
    - “Generar Polígono” para crear una figura aleatoria.
    - “Escoger Color” para elegir el color de relleno.
    - “Rellenar Figura” para iniciar el algoritmo Scanline.
    - “Limpiar” para borrar el lienzo y volver a empezar, y “Cerrar” en la parte inferior para salir del formulario.

6.3.2.2 Parámetros utilizados

Principales parámetros y campos del formulario y del algoritmo:

- lienzo: bitmap asociado al PictureBox, sobre el que se dibuja el contorno y luego se rellenan los píxeles interiores.
- colorRelleno: color seleccionado por el usuario mediante el cuadro de diálogo de color; se usa para pintar el interior del polígono.
- poligono: lista de puntos (List<Point>) que almacena los vértices del polígono generado aleatoriamente.
- minY, maxY: valores mínimo y máximo de la coordenada Y de los vértices, utilizados para determinar el rango vertical de scanlines a procesar.
- animando: bandera booleana que indica si el proceso de relleno animado está en ejecución.
- yInicioAnim, yFinAnim, yActualAnim: indican el primer y último valor de Y que se va a rellenar y la scanline actual que se está procesando.
- colorAnim: copia del color de relleno que se utiliza en la animación.
- timerAnim: temporizador que avanza la animación, rellenando una fila por tick.

6.3.2.3 Explicación de las funciones

- Constructor frmScanlineFill
  - Crea un bitmap en blanco para lienzo, lo asigna a picCanvas.Image y activa DoubleBuffered para un mejor refresco visual.
  - Configura el timerAnim con un intervalo de 15 ms y suscribe el evento timerAnim_Tick, que se encarga de ir rellenando cada fila.
- btnPickColor_Click
  - Abre un ColorDialog; si el usuario confirma, guarda el color en colorRelleno y lo muestra en el fondo del botón para indicar el color elegido.
- btnGeneratePolygon_Click
  - Si animando es true, detiene primero la animación.
  - Limpia el lienzo y lo rellena de blanco.
  - Vacía la lista poligono y genera un número aleatorio de vértices n entre 6 y 12.
  - Calcula un centro (cx, cy) aproximadamente en el medio del PictureBox y un radio máximo maxR.
  - Para cada vértice, genera un ángulo y un radio aleatorio (para obtener un polígono irregular pero cerrado), calcula las coordenadas (px, py) y las añade a la lista poligono, actualizando minY y maxY según la Y de cada punto.
  - Dibuja el contorno del polígono sobre el bitmap usando DrawPolygon con un lápiz negro de grosor 4 y suavizado de bordes (SmoothingMode.AntiAlias).
- ScanlineFillStep
  - Implementa un paso del algoritmo Scanline para una sola fila y.

    - Recorre todas las aristas del polígono y calcula las intersecciones de la línea horizontal y con cada arista que cruza esa fila (ignorando aristas horizontales).
    - Para cada arista válida, calcula la coordenada X de la intersección y la añade a la lista de intersecciones.
    - Si hay menos de dos intersecciones, termina sin rellenar nada.
    - Ordena las intersecciones de menor a mayor y rellena los segmentos entre pares [x0, x1], [x2, x3], etc., ajustando xIni y xFin para que estén dentro de los límites del bitmap y escribiendo el color de relleno en cada píxel entre esos extremos.
- timerAnim_Tick
    - Si animando es false, no hace nada.
    - Si yActualAnim es mayor que yFinAnim, detiene el timer, marca animando = false y reactiva los botones de rellenar, generar polígono y escoger color.
    - En caso contrario, llama a ScanlineFillStep para la fila yActualAnim con el colorAnim, incrementa yActualAnim y vuelve a invalidar el PictureBox para refrescar el dibujo, creando la animación fila por fila.
- btnScanlineFill_Click
    - Comprueba que colorRelleno no esté vacío; si no se ha escogido color, muestra un mensaje.
    - Comprueba que poligono tenga al menos 3 vértices; si no, avisa que primero debe generarse un polígono.
    - Fija yInicioAnim como el máximo entre 0 y minY, y yFinAnim como el mínimo entre la altura del lienzo menos uno y maxY; inicializa yActualAnim con yInicioAnim y colorAnim con colorRelleno.
    - Desactiva los botones de rellenar, generar polígono y escoger color, marca animando = true e inicia el timerAnim.
- btnClean_Click
    - Si animando es true, detiene el timer y marca animando = false.
    - Limpia la lista poligono, crea un nuevo bitmap en blanco, lo asigna a picCanvas.Image y vuelve a habilitar los botones de generación, relleno y color.
- btnClose_Click
    - Cierra el formulario.

6.3.3 Casos de prueba

1. El usuario entra al formulario y presiona directamente el botón “Rellenar Figura” sin escoger color ni generar polígono.
   Resultado esperado: primero se muestra un mensaje indicando que debe seleccionar un color de relleno; tras escoger un color, si vuelve a presionar “Rellenar

Figura” sin generar polígono, se muestra un mensaje indicando que debe generar un polígono antes de rellenar. En ningún caso se produce un cierre inesperado ni se modifica el lienzo.

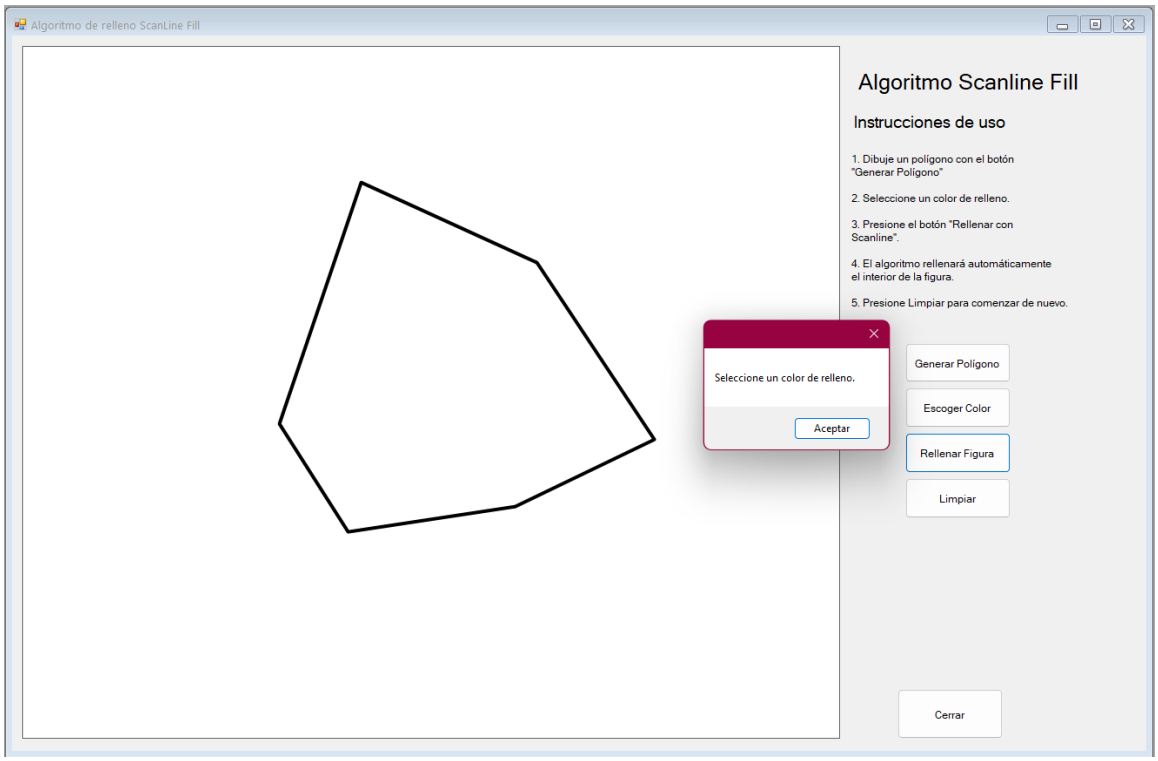


Ilustración 39. Caso de prueba 1 - Algoritmo ScanLineFill

2. El usuario pulsa “Generar Polígono”, se dibuja una figura irregular cerrada en el centro del PictureBox y luego pulsa “Escoger Color” para elegir un color (por ejemplo, naranja).
   Resultado esperado: los botones de generar polígono, escoger color y rellenar se desactivan y comienza la animación; el interior del polígono se va llenando desde la fila minY hasta la fila maxY, observándose cómo aparecen franjas horizontales sucesivas en el color elegido, respetando el contorno negro sin dejar huecos internos ni colorear píxeles fuera del polígono. Cuando yActualAnim supera yFinAnim, la animación se detiene, los botones se reactivan y el polígono queda completamente relleno.

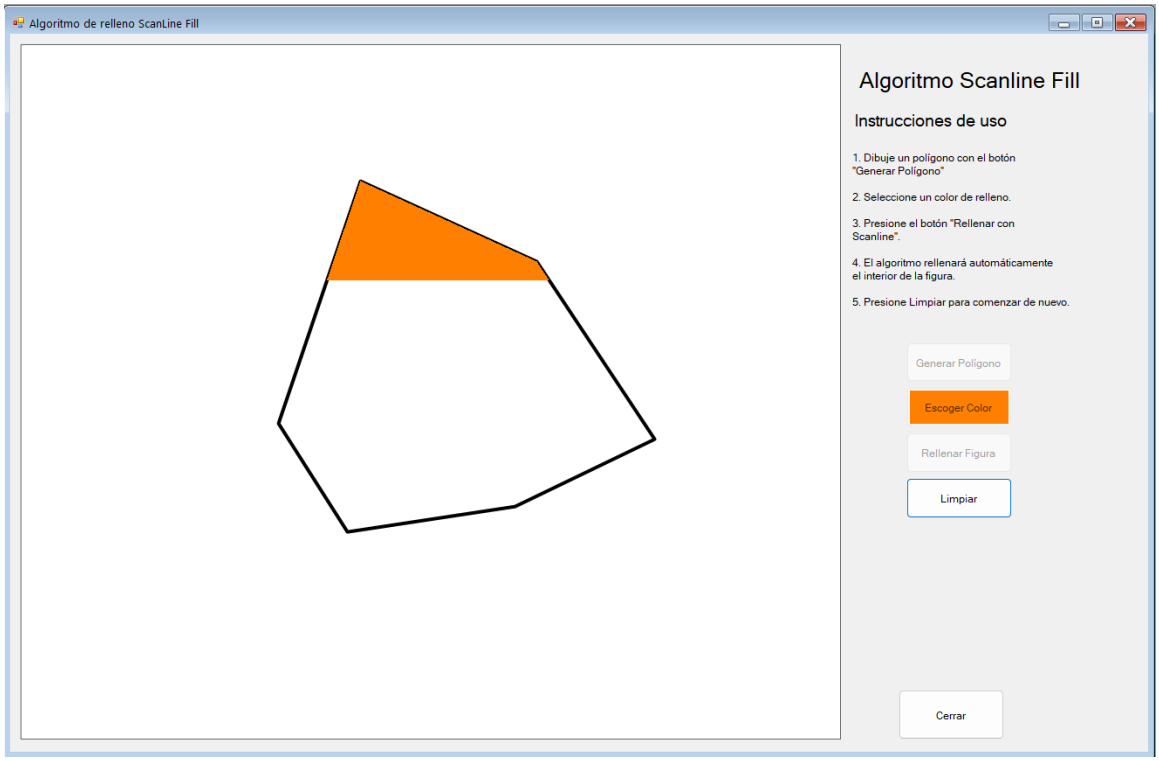


Ilustración 40. Caso de prueba 2-1 - Algoritmo ScanLineFill

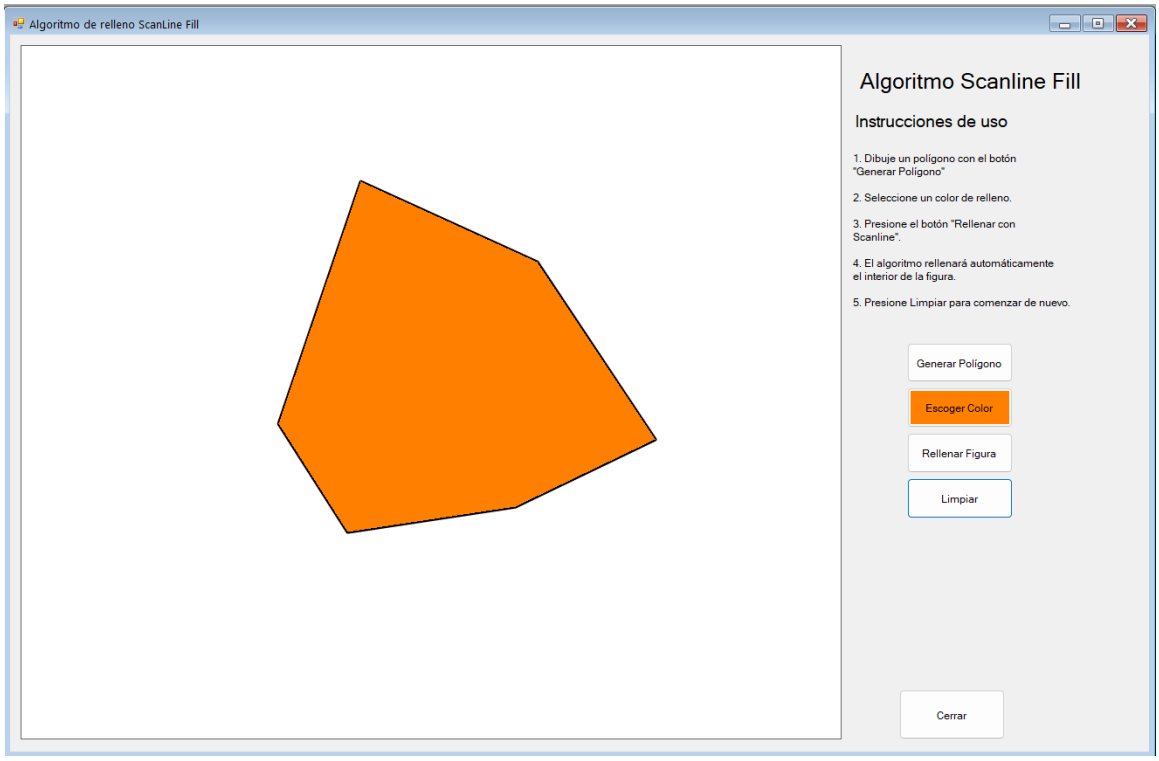


Ilustración 41. Caso de prueba 2-2 - Algoritmo ScanLine Fill

7. Algoritmos de recorte de líneas

Para el recorte de líneas respecto a una ventana rectangular se utilizan el algoritmo de Cohen–Sutherland y una variante paramétrica basada en la ecuación de la recta. Estos algoritmos permiten determinar la porción visible de un segmento con respecto a la ventana

7.1 Algoritmo de Cohen–Sutherland

El algoritmo de Cohen–Sutherland asigna a cada extremo del segmento un código de región de 4 bits que indica si el punto está a la izquierda, derecha, arriba o abajo de la ventana. Si ambos códigos son 0000, la línea se acepta directamente; si el AND bit a bit de ambos códigos es distinto de 0000, la línea se rechaza por estar completamente fuera. En otros casos, se calculan intersecciones con los bordes y se actualizan los extremos hasta aceptar o rechazar el segmento (*Cohen-Sutherland Line Clipping in Computer Graphics*, n.d.)

7.1.1 Justificación de la variante del algoritmo

El algoritmo de recorte de líneas de Cohen–Sutherland es uno de los métodos clásicos y más difundidos para recortar segmentos contra una ventana rectangular. Divide el plano en 9 regiones (la ventana central y ocho regiones exteriores) y asigna a cada extremo de la línea un código de región de 4 bits que indica si el punto está a la izquierda, derecha, arriba o abajo de la ventana.

Con estos códigos se pueden realizar pruebas de aceptación y rechazo triviales mediante operaciones OR y AND bit a bit; solo cuando una línea no se puede aceptar ni rechazar trivialmente se calculan las intersecciones con el borde de la ventana.

Se eligió esta variante porque es el algoritmo estándar de referencia en muchos libros y cursos de computación gráfica para el recorte de líneas, especialmente eficiente cuando la mayoría de los segmentos se aceptan o rechazan de forma trivial y adecuado para ventanas rectangulares como la usada en este formulario.

7.1.2 Descripción del Formulario

7.1.2.1 Diseño del formulario

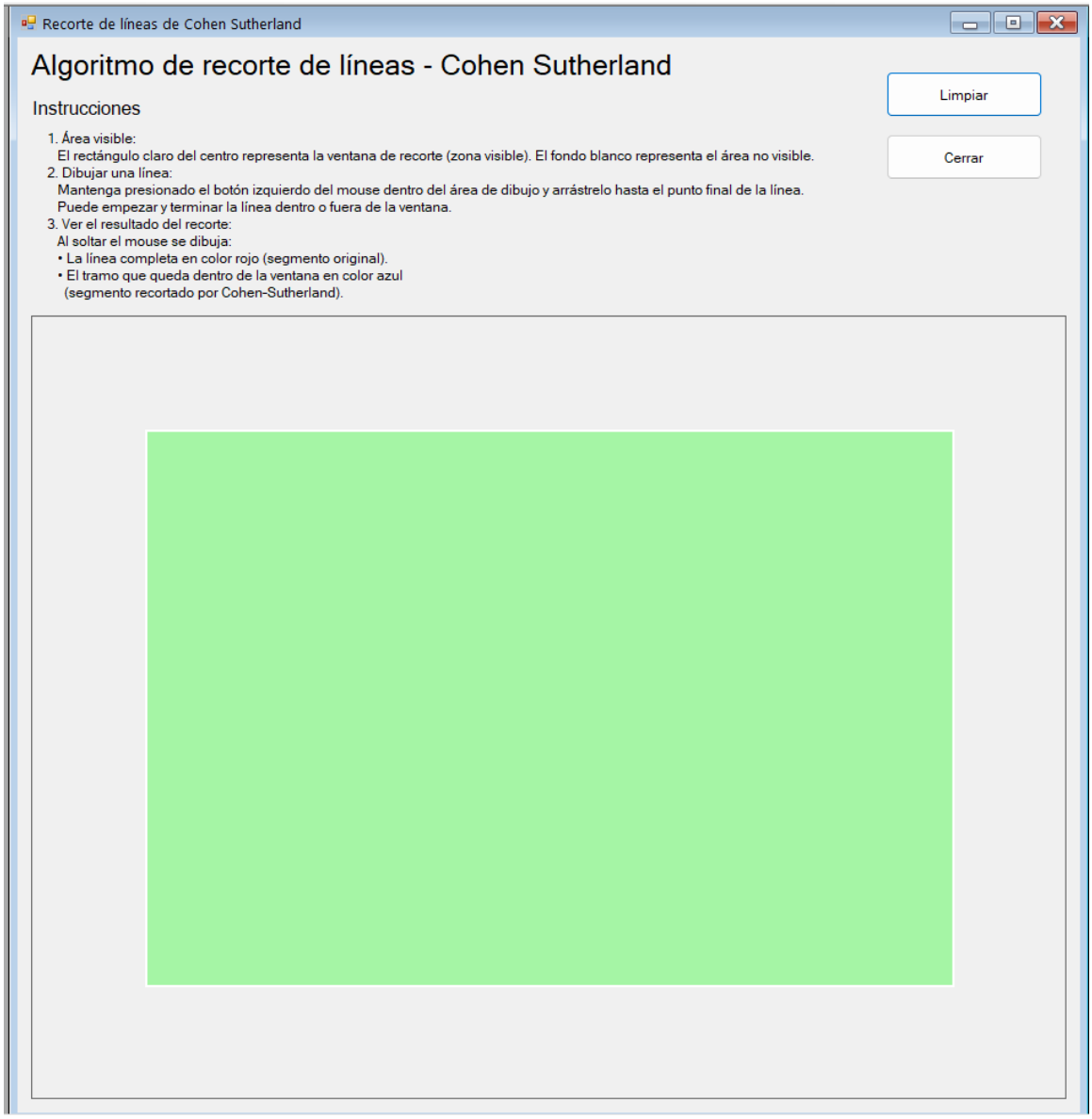


Ilustración 42. Formulario para el algoritmo de recorte de líneas Cohen – Sutherland

El formulario "Algoritmo de recorte de líneas – Cohen Sutherland" está organizado de la siguiente manera:

- En la parte superior se muestra el título y un bloque de instrucciones numeradas que explican cómo interpretar la zona visible, dibujar una línea y observar el resultado del recorte.
- A la derecha se ubican los botones "Limpiar" y "Cerrar".

- En la zona central se encuentra un PictureBox que actúa como área de dibujo. Dentro de él aparece un rectángulo central en color verde claro que representa la ventana de recorte; el resto del fondo queda blanco como zona no visible.

- Cuando el usuario dibuja una línea con el mouse, se visualiza el segmento original en color rojo y, si tiene una parte visible dentro de la ventana, se dibuja ese tramo recortado en color azul. Además, un label muestra los códigos de región de los extremos de la línea.

7.1.2.2 Parámetros utilizados

Principales parámetros y campos usados en el formulario:

- lienzo: bitmap asociado al PictureBox sobre el que se dibuja la ventana de recorte y las líneas.

- ventanaRecorte: rectángulo que define la ventana de recorte, centrado en el PictureBox con un margen fijo.

- dibujandoLinea: bandera booleana que indica si el usuario está arrastrando el mouse para definir una línea.

- pInicio, pFin: puntos flotantes que representan los extremos de la línea que el usuario dibuja.

- enumeración OutCode: conjunto de valores con banderas Inside, Left, Right, Bottom y Top que codifican la posición de un punto respecto a la ventana (esquema de 4 bits típico del algoritmo).

7.1.2.3 Explicación de las funciones

- Constructor frmCohenSutherland
- Crea el bitmap lienzo con el tamaño del PictureBox y lo asigna a picCanvas.Image.
- Define ventanaRecorte como un rectángulo centrado con un margen de 100 píxeles alrededor.
- Llama a DibujarVentana para pintar el área visible en verde y el borde blanco.
- Asocia manejadores de eventos para MouseDown y MouseUp del PictureBox.
- DibujarVentana
- Limpia el bitmap y dibuja la ventana de recorte:
- Rellena ventanaRecorte con un color semitransparente verde (zona visible).
- Dibuja el contorno blanco de la ventana.
- Refresca el PictureBox con picCanvas.Invalidate.
- ComputeOutCode
- Calcula el código de región para un punto (x, y) comparándolo con los límites de ventanaRecorte.

- Marca las banderas Left o Right si el punto está a la izquierda o derecha de la ventana, y Top o Bottom si está por encima o por debajo.
- Devuelve el código combinado mediante operaciones OR.
- OutCodeToBits
- Convierte el valor entero del código de región en una cadena de 4 bits (por ejemplo 1010), útil para mostrar en el label de información.
- RecorteCohenSutherland
- Implementa el núcleo del algoritmo de recorte.
- Obtiene los códigos de región de los extremos p0 y p1 y entra en un bucle:
- Si el OR de ambos códigos es Inside (0000), la línea se acepta trivialmente (accept = true).
- Si el AND de ambos códigos es distinto de 0, la línea se rechaza trivialmente (no hay parte visible).
- En cualquier otro caso, se elige uno de los extremos que está fuera de la ventana y se calcula su intersección con el borde correspondiente (top, bottom, left o right) usando la ecuación paramétrica de la línea; luego se reemplaza ese extremo por el punto de intersección y se recalcula su código de región.
- El bucle se repite hasta que la línea se acepta o rechaza; si se acepta, los puntos p0 y p1 quedan recortados a la ventana.
- picCanvas_MouseDown
- Se ejecuta cuando el usuario presiona el botón izquierdo del mouse.
- Activa dibujandoLinea y guarda pInicio con la posición inicial.
- picCanvas_MouseUp
- Si dibujandoLinea es false, no hace nada. En caso contrario, toma la posición final e interpreta el gesto como el trazado de una línea.
- Dibuja primero el segmento completo entre pInicio y pFin en color rojo sobre el bitmap.
- Calcula los códigos de región de los extremos originales y actualiza el label de información con los bits de P0 y P1.
- Copia los puntos a q0 y q1 y llama a RecorteCohenSutherland. Si la función devuelve true, dibuja el segmento visible resultante en color azul con un grosor mayor.
- Finalmente refresca el PictureBox con picCanvas.Invalidate.
- btnClean_Click
- Llama a DibujarVentana para borrar las líneas dibujadas y redibujar solo la ventana de recorte.
- btnClose_Click
- Cierra el formulario.

7.1.3 Casos de prueba

1. El usuario abre el formulario y observa la ventana verde claro que representa el área visible. A continuación, dibuja una línea completamente dentro de la ventana, haciendo clic y arrastrando el mouse de un punto interior a otro punto interior sin salir del área verde.

   Resultado esperado: al soltar el mouse, la línea roja coincide con la azul porque el segmento completo es visible; el algoritmo la acepta trivialmente y el label muestra códigos de región 0000 para ambos extremos.

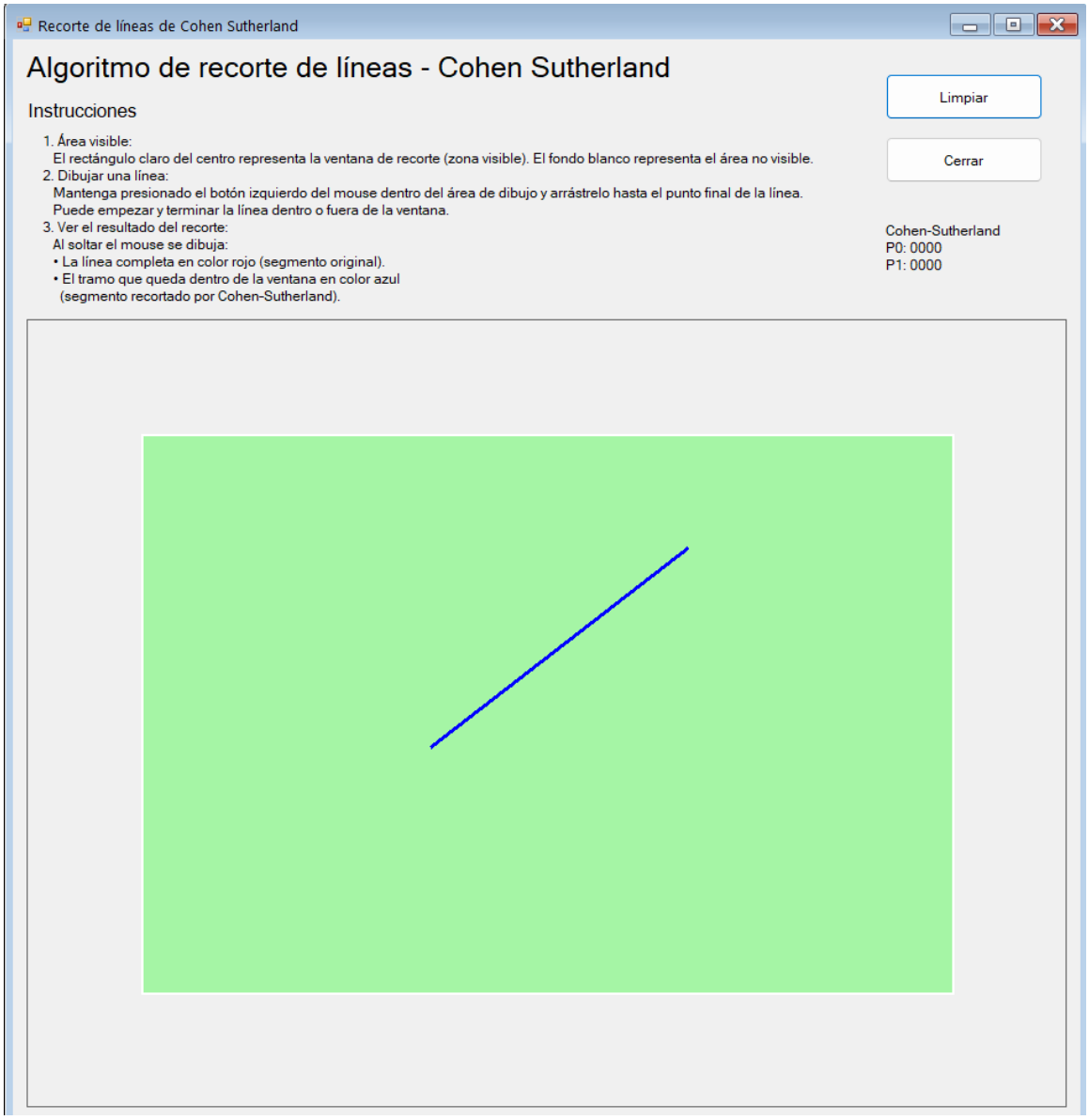


Ilustración 43. Caso de prueba 1 - Algoritmo Cohen Sutherland

2. Sin limpiar, el usuario dibuja una línea totalmente fuera de la ventana, por ejemplo, en la parte superior del PictureBox, de manera que ambos puntos queden por encima del rectángulo verde.

   Resultado esperado: se dibuja la línea original en rojo fuera de la ventana, pero no aparece ningún tramo azul, porque el algoritmo detecta mediante el AND de los códigos que los dos extremos comparten una región exterior (por ejemplo, Top) y rechaza trivialmente el segmento. El label muestra códigos con el bit superior activado (por ejemplo 1000 y 1000).

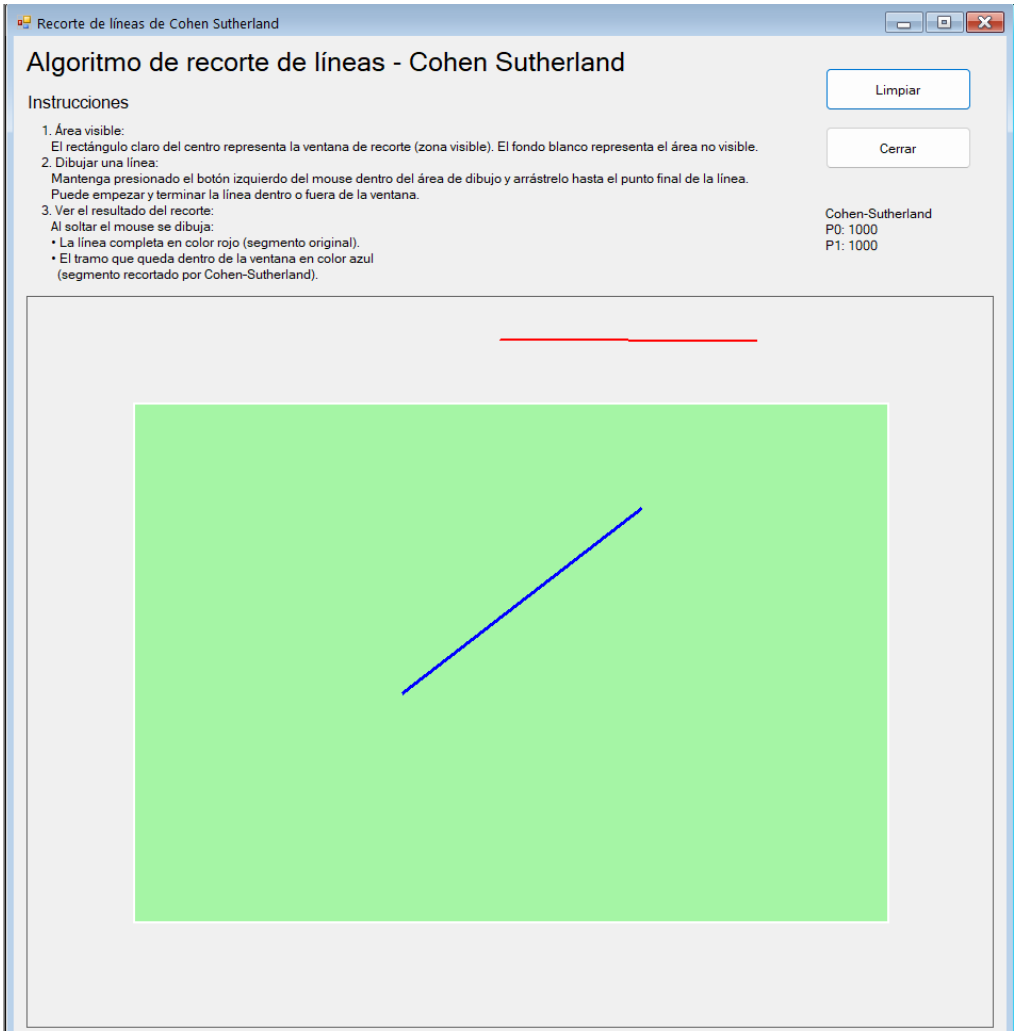


Ilustración 44. Caso de prueba 2 - Algoritmo Cohen Sutherland

3. El usuario dibuja ahora una línea que cruza la ventana, comenzando fuera del rectángulo en la parte izquierda y terminando fuera en la parte derecha, asegurándose de pasar por el interior de la zona verde.

   Resultado esperado: al soltar el mouse, se ve la línea roja completa atravesando el área verde; adicionalmente, el algoritmo calcula las intersecciones con los bordes izquierdo y derecho de la ventana y muestra en azul solo el tramo comprendido dentro del rectángulo. El label presenta códigos de región distintos de 0000 para los extremos originales (por ejemplo 0001 y 0010); la aplicación no lanza errores ni se cierra inesperadamente, confirmando que la implementación de Cohen–Sutherland recorta correctamente segmentos aceptados, rechazados y parcialmente visibles.

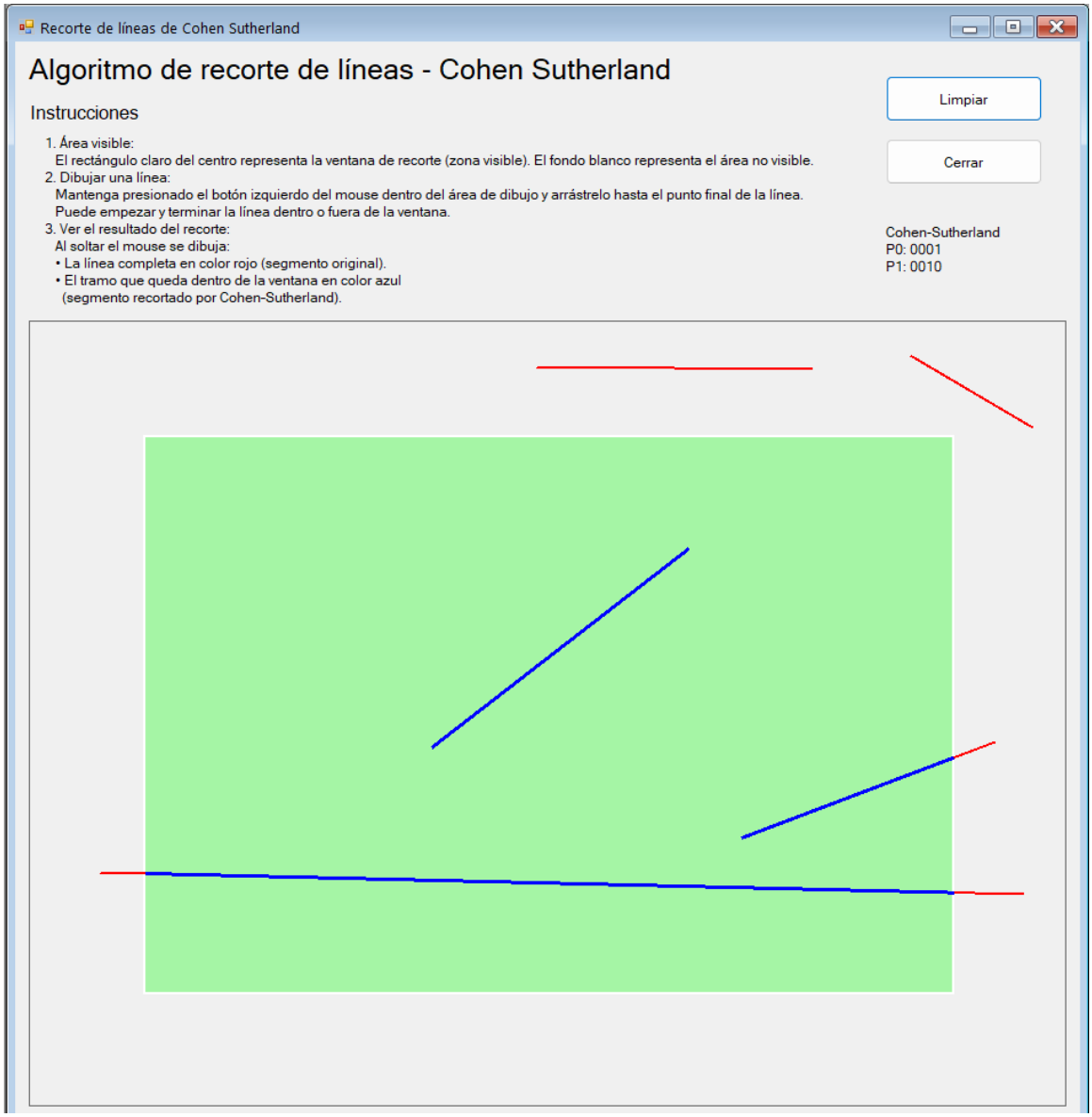


Ilustración 45. Caso de prueba 3 - Algoritmo Cohen Sutherland

7.2 Algoritmo de Liang Barsky

El algoritmo de Liang–Barsky es un método de recorte de líneas que usa la forma paramétrica del segmento $P(t) = P_0 + t(P_1 - P_0)$, $t \in [0,1]$ (Libretexts, 2025), y convierte el recorte contra una ventana rectangular en 4 desigualdades (izquierda, derecha, abajo, arriba). A partir de ellas calcula un rango válido $[t_{\text{entrada}}, t_{\text{salida}}]$: si el rango queda vacío, la línea se rechaza; si no, se dibuja solo el tramo entre esos dos valores. Es más eficiente que enfoques basados en códigos de región (p. ej., Cohen–Sutherland) porque reduce intersecciones innecesarias y hace la mayor parte del "testing" con comparaciones y mínimos/máximos. (Wikipedia contributors, 2025)

7.2.1 Justificación de la variante del algoritmo

El algoritmo de recorte de líneas de Liang–Barsky se basa en la forma paramétrica del segmento y en las desigualdades que definen la ventana de recorte. En lugar de usar códigos de región como Cohen–Sutherland, trabaja con parámetros t0 y t1 (entre 0 y 1) que representan el tramo visible de la línea dentro de la ventana. Esto permite calcular directamente los puntos de entrada y salida del segmento con respecto a cada borde del rectángulo usando unas pocas comparaciones y divisiones.
Se eligió esta variante porque:

- Es más eficiente en número de operaciones que Cohen–Sutherland, ya que solo realiza intersecciones cuando realmente hay posibilidad de visibilidad y no necesita recorrer iterativamente la línea.
- Utiliza una formulación matemática compacta basada en parámetros, lo que la hace adecuada para ilustrar el enfoque de recorte numérico frente al recorte por códigos de región.

7.2.2 Descripción del Formulario

7.2.2.1 Diseño del formulario

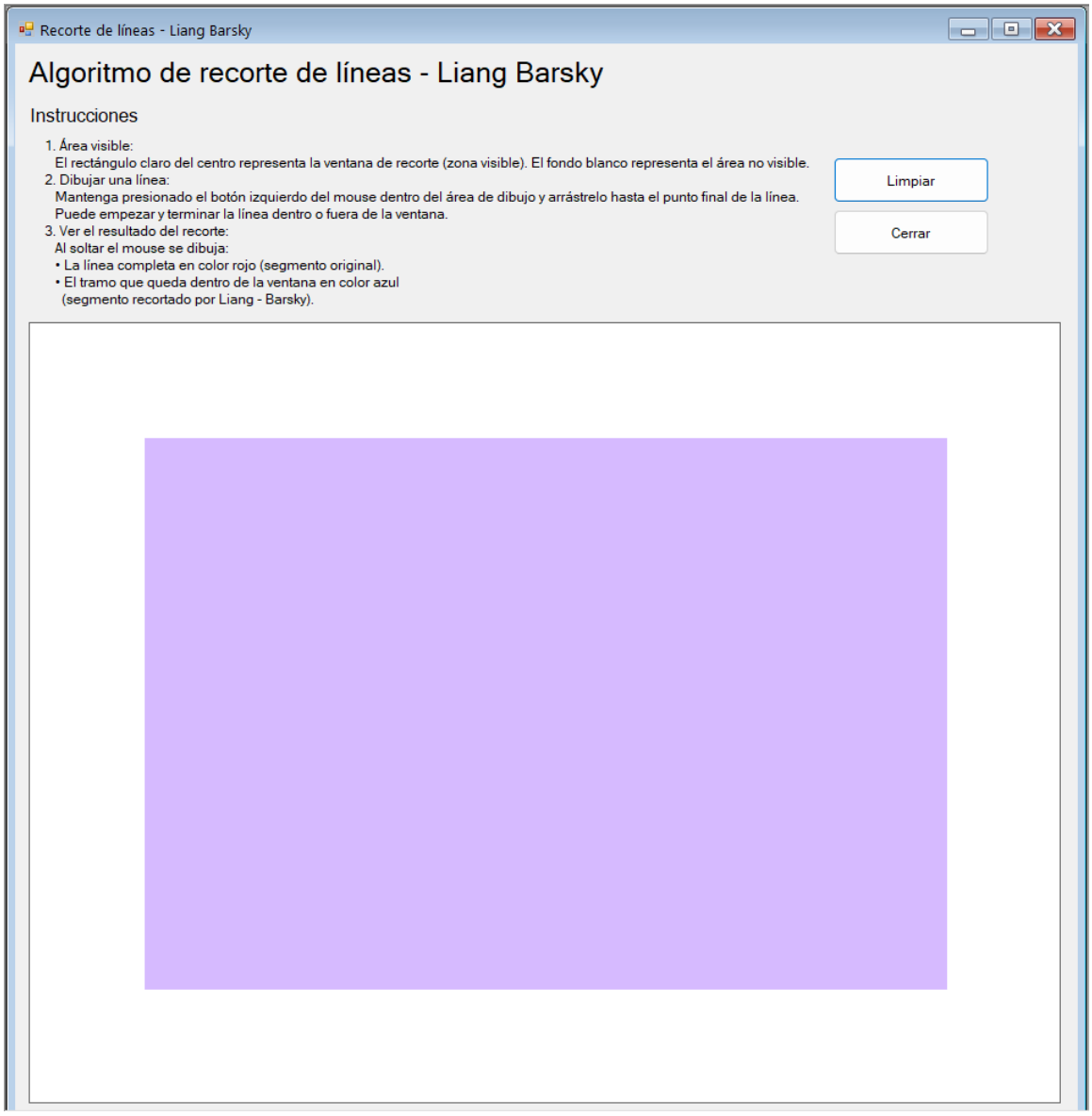


Ilustración 46. Formulario para el algoritmo de recorte de líneas Liang – Barsky

El formulario “Algoritmo de recorte de líneas – Liang Barsky” mantiene un diseño similar al de Cohen–Sutherland:

- En la parte superior se muestra el título del algoritmo y un bloque de instrucciones numeradas que explican el significado de la ventana de recorte, cómo dibujar la línea y cómo interpretar el resultado.
- A la derecha se encuentran los botones “Limpiar” (para borrar las líneas y redibujar solo la ventana) y “Cerrar”.

- En la zona central se ubica un PictureBox que funciona como área de dibujo. Dentro de este se dibuja un rectángulo centrado en color lila claro que representa la ventana; el resto del fondo se mantiene blanco.

- Cuando el usuario dibuja una línea con el mouse, el segmento original aparece en rojo y, si tiene una parte visible dentro de la ventana, se dibuja en azul únicamente el tramo recortado por el algoritmo Liang–Barsky.

7.2.2.2 Parámetros utilizados

Principales parámetros y campos empleados en el formulario y en el algoritmo:

- lienzo: bitmap del tamaño del PictureBox sobre el que se dibuja la ventana y las líneas.

- ventana: rectángulo que define la ventana de recorte, centrado con un margen fijo respecto a los bordes del PictureBox.

- dibujandoLinea: bandera booleana que indica si el usuario está arrastrando el mouse para definir un segmento.

- pInicio, pFin: puntos que representan los extremos de la línea original dibujada por el usuario.

- p0, p1 (PointF): copias de pInicio y pFin que se pasan por referencia al algoritmo Liang–Barsky y se actualizan con el tramo visible si lo hay.

- t0, t1: parámetros del segmento en forma paramétrica, inicialmente 0 y 1, que se van ajustando en las pruebas contra cada borde para determinar el subsegmento visible.

7.2.2.3 Explicación de las funciones

- Constructor frmLiangBarsky
    - Crea el bitmap lienzo con el tamaño del PictureBox y lo asigna a picCanvas.Image.
    - Llama a DibujarVentana para inicializar el fondo blanco y el rectángulo de recorte centrado.
- DibujarVentana
    - Calcula la ventana como un rectángulo con margen de 100 píxeles alrededor.
    - Limpia el lienzo con fondo blanco.
    - Rellena ventana con un color lila suave (zona visible) y dibuja su borde con un trazo blanco de grosor 2.
    - Refresca el PictureBox con picCanvas.Invalidate.
- picCanvas_MouseDown
    - Marca dibujandoLinea = true y almacena pInicio con la posición donde el usuario presiona el mouse.
    - picCanvas_MouseUp

    - Si dibujandoLinea es false, no hace nada. En caso contrario, desactiva dibujandoLinea, guarda pFin con la posición final y realiza el flujo de recorte:
    - Dibuja la línea completa en rojo entre pInicio y pFin sobre el bitmap.
    - Copia pInicio y pFin en p0 y p1 y llama a LiangBarskyClip; si la función devuelve true, dibuja en azul el subsegmento visible entre p0 y p1.
    - Actualiza el PictureBox con picCanvas.Invalidate.
- ClipTest
    - Implementa la prueba elemental de Liang–Barsky para un borde de la ventana, recibiendo:
    - p: componente de la dirección del segmento respecto a ese borde (por ejemplo −dx, dx, −dy, dy).
    - q: distancia desde el punto inicial al borde (por ejemplo x0 − xMin, xMax − x0, etc.).
    - t0, t1: parámetros globales que delimitan el tramo visible.
    - Si p es 0, el segmento es paralelo a ese borde; si q < 0, el segmento está totalmente fuera con respecto a ese borde y se rechaza devolviendo false, en caso contrario se acepta respecto a ese borde sin cambiar t0 ni t1.
    - Si p < 0, se trata de un borde de entrada: calcula r = q / p, y si r > t1 rechaza el segmento; si r > t0, actualiza t0 = r.
    - Si p > 0, se trata de un borde de salida: calcula r = q / p, y si r < t0 rechaza el segmento; si r < t1, actualiza t1 = r.
    - Devuelve true mientras el segmento siga siendo potencialmente visible.
- LiangBarskyClip
    - Obtiene las coordenadas iniciales x0, y0, x1, y1 y calcula dx = x1 − x0, dy = y1 − y0.
    - Inicializa t0 = 0 y t1 = 1.
    - Recupera los límites de la ventana (xMin, xMax, yMin, yMax).
    - Llama a ClipTest cuatro veces, una por cada borde: izquierda, derecha, arriba y abajo, pasando los valores adecuados de p y q.
    - Si todas las pruebas devuelven true, significa que existe parte visible del segmento entre los parámetros t0 y t1; actualiza entonces los puntos:
    - Si t1 < 1, recalcula el extremo final con x1 = x0 + t1dx y y1 = y0 + t1dy.
    - Si t0 > 0, recalcula el extremo inicial con x0 = x0 + t0dx y y0 = y0 + t0dy.
    - Asigna estos nuevos valores a p0 y p1 y devuelve true.
    - Si alguna prueba de ClipTest devuelve false, no hay intersección con la ventana y la función devuelve false sin modificar los puntos.
- btnClean_Click
    - Llama a DibujarVentana para borrar cualquier línea y redibujar únicamente la ventana de recorte sobre el fondo blanco.
- btnClose_Click
    - Cierra el formulario.

7.2.3 Casos de prueba

1. El usuario abre el formulario y dibuja una línea completamente horizontal y paralela a uno de los bordes de la ventana, pero situada totalmente fuera de la zona lila (por ejemplo, muy por encima del rectángulo).
   Resultado esperado: se dibuja la línea original en rojo, pero no aparece ninguna línea azul porque ClipTest detecta, con p = −dy = 0 para los bordes superior e inferior y q < 0 respecto a la región visible, que el segmento está fuera; LiangBarskyClip devuelve false y la aplicación no realiza ningún recorte adicional.

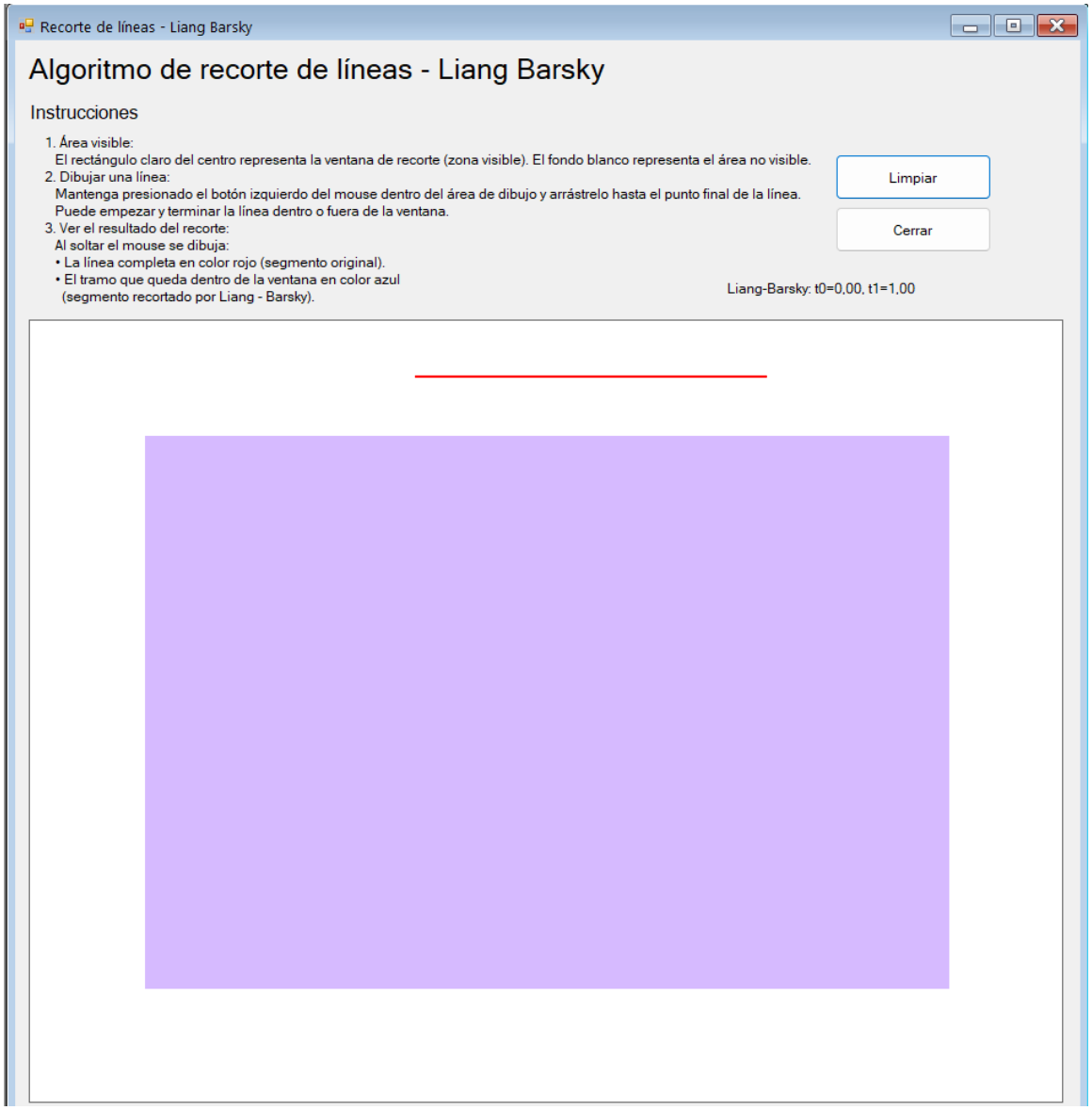
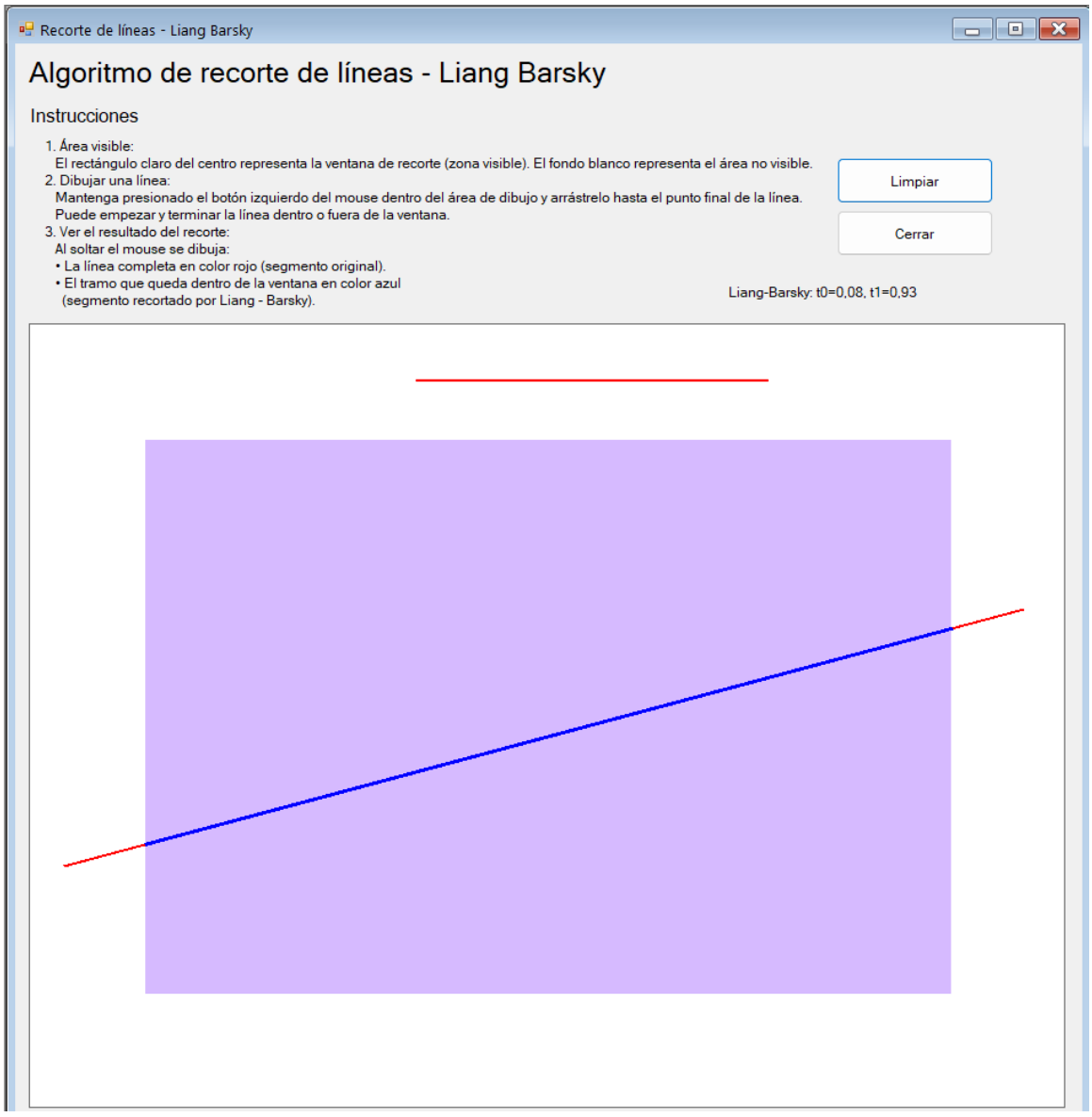


Ilustración 47. Caso de prueba 1 - Algoritmo Liang Barsky

2. El usuario dibuja otra línea horizontal paralela a los bordes superior e inferior, esta vez atravesando el rectángulo lila (por ejemplo, empezando fuera a la izquierda y terminando fuera a la derecha).

Resultado esperado: el segmento rojo se ve completo, y además se dibuja un tramo azul que corresponde exactamente a la parte de la línea comprendida dentro de la ventana. LiangBarskyClip ajusta t0 y t1 con las intersecciones en los bordes izquierdo y derecho, recalculando p0 y p1 para que coincidan con los puntos de entrada y salida de la ventana. El formulario sigue estable y no se producen errores.



3. El usuario dibuja una línea diagonal que empieza dentro de la ventana y termina fuera (o viceversa).

   Resultado esperado: se muestra la línea roja completa y, sobre ella, solo el tramo azul contenido en la ventana, desde el punto interior hasta el borde donde la recta sale del rectángulo. Los cálculos paramétricos del algoritmo ajustan uno de los extremos en función de t0 o t1, mientras el otro permanece igual; el comportamiento es coherente tanto si el punto interior es el inicio como si es el final, confirmando que la implementación de Liang–Barsky maneja correctamente

casos de segmentos parcialmente visibles, paralelos y totalmente externos sin cierres inesperados ni resultados inconsistentes.

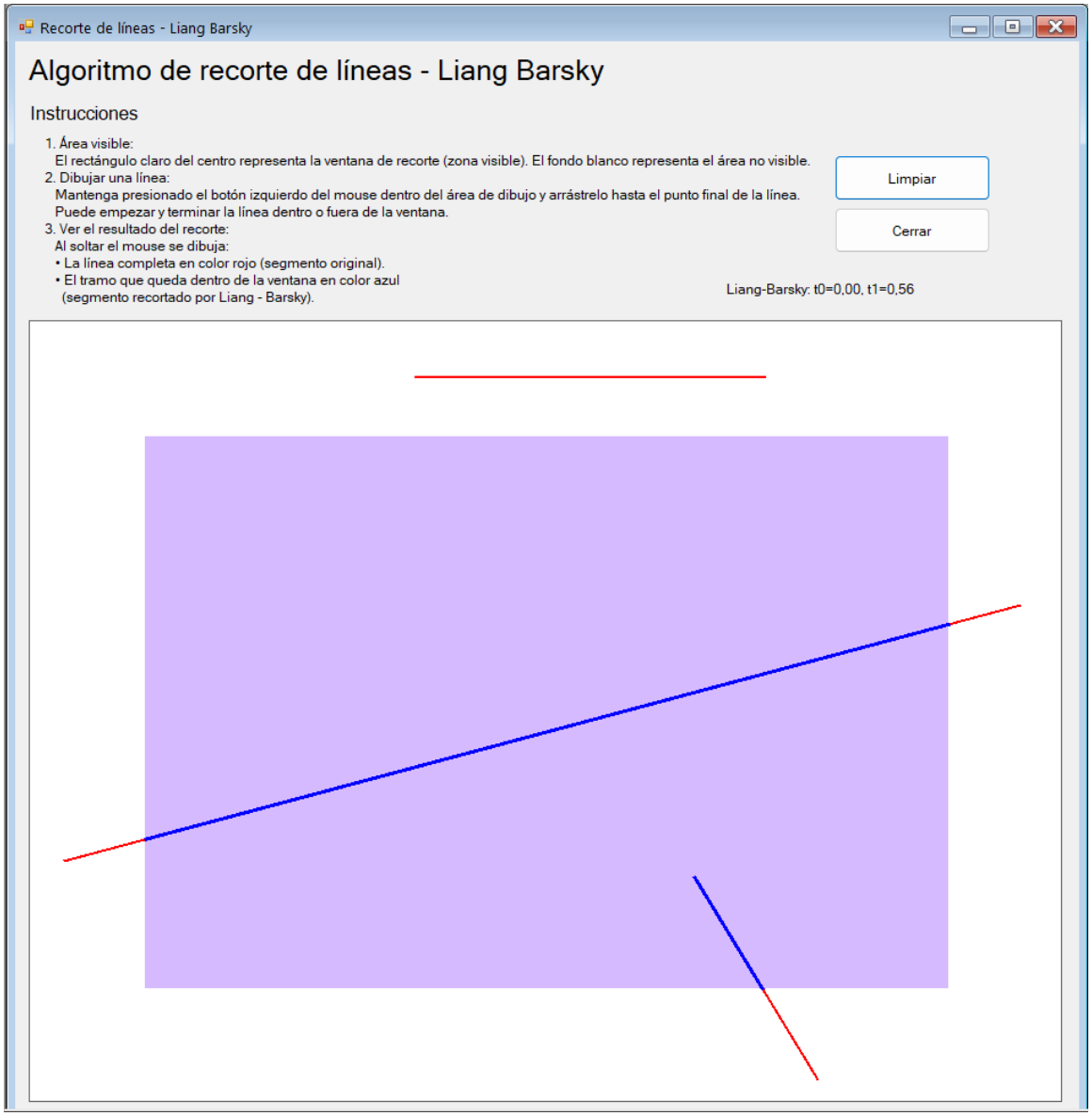


Ilustración 48. Caso de prueba 3 - Algoritmo Liang Barsky

7.3 Algoritmo de Recorte de Punto Medio

El algoritmo de recorte de líneas por punto medio (también llamado Midpoint Subdivision) recorta un segmento aplicando una estrategia de divide y vencerás: primero calcula los códigos de región (outcodes) de los dos extremos respecto a la ventana rectangular. Si ambos outcodes indican “dentro”, el segmento se acepta; si el AND de los outcodes es distinto de cero, se rechaza (está fuera por el mismo lado). Si no es trivial, el algoritmo divide el segmento por su punto medio, vuelve a evaluar outcodes y repite recursivamente sobre

los subsegmentos hasta aislar la parte visible (o hasta una tolerancia/píxel). Se considera una extensión conceptual de Cohen–Sutherland y destaca por basarse en bisecciones sucesivas (y en algunas implementaciones evita calcular intersecciones directas con la frontera). (Matthes & Drakopoulos, 2022)

7.3.1 Justificación de la variante del algoritmo

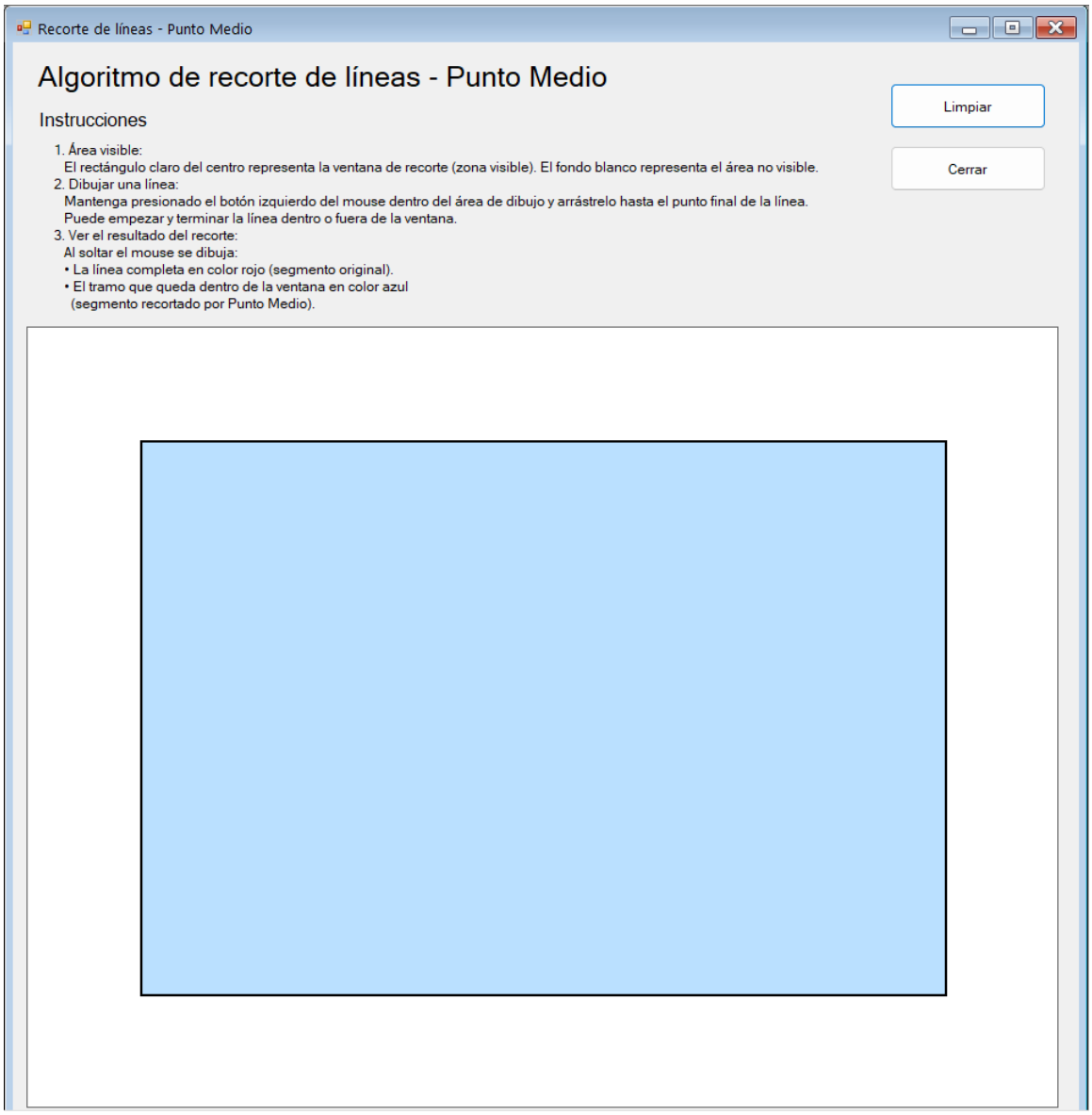
El recorte de líneas por subdivisión de punto medio (midpoint subdivision) es una variante recursiva que combina la idea de los códigos de región con la división repetida del segmento hasta separar claramente los trozos visibles de los invisibles. El algoritmo parte de una línea candidata al recorte y, si no se puede aceptar o rechazar de forma trivial, calcula su punto medio, genera dos sub-segmentos y repite el proceso hasta obtener segmentos completamente dentro o fuera de la ventana.
Esta técnica se eligió como tercera variante porque permite ilustrar un enfoque distinto a Cohen-Sutherland y Liang–Barsky: en lugar de calcular intersecciones exactas con los bordes, aproxima el recorte mediante subdivisión recursiva, lo que simplifica el cálculo y evita operaciones complejas cuando se trabaja sobre coordenadas de pantalla. Además, muchos apuntes de computación gráfica lo presentan como algoritmo educativo para comparar desempeño y precisión frente a los métodos paramétricos clásicos.

7.3.2 Descripción del Formulario

El formulario “Algoritmo de recorte de líneas – Punto Medio” conserva la misma estructura visual que las demás variantes de recorte:

- En la parte superior se muestra el título y un bloque de instrucciones numeradas que explican el significado del área visible, cómo dibujar la línea con el mouse y cómo interpretar el resultado.
- A la derecha se encuentran los botones “Limpiar” y “Cerrar”.
- En la zona central hay un PictureBox que funciona como área de dibujo. Dentro de este se dibuja un rectángulo azul claro que representa la ventana de recorte; el resto del fondo es blanco (zona no visible).
- Al dibujar una línea, el segmento original se muestra en rojo y los tramos que quedan dentro de la ventana, después del recorte por subdivisión de punto medio, se dibujan en azul. Un label inferior muestra información adicional sobre la longitud de la línea, el número de trozos visibles y la profundidad máxima de recursión utilizada.

7.3.2.1 Diseño del formulario



7.3.2.2 Parámetros utilizados

Principales parámetros y campos usados en la implementación:

- lienzo: bitmap asociado al PictureBox donde se dibuja la ventana y los segmentos (original y recortados).
- ventanaRecorte: rectángulo que define la región visible; se calcula centrado en el área de dibujo con un margen fijo de 100 píxeles.
- dibujandoLinea: bandera booleana que indica si el usuario está arrastrando el mouse para definir la línea.

- pInicio, pFin: puntos enteros que representan los extremos originales del segmento dibujado con el mouse.
- MAX_DEPTH: constante que fija la profundidad máxima de subdivisión recursiva (18 niveles) para evitar recursión excesiva.
- maxDepthUsada: contador que registra la mayor profundidad alcanzada por la recursión para mostrarla en el label de información.
- Listas visibles: (List<Tuple<PointF, PointF>>): colección de sub-segmentos que han sido clasificados como visibles dentro de la ventana después del recorte.

Funciones auxiliares clave:

- PuntoDentro: verifica si un punto está dentro de la ventana comparando sus coordenadas con los límites del rectángulo de recorte.
- SegmentoFueraBBox: calcula la caja envolvente del segmento y comprueba si esta no intersecta la ventana; si no hay intersección posible, el segmento se rechaza trivialmente.
- Distancia: obtiene la longitud del segmento para decidir cuándo es suficientemente pequeño como para detener la subdivisión.
- ClampToWindow: "recorta" un punto a los límites de la ventana, ajustando sus coordenadas para que queden dentro del rectángulo cuando se llega al límite de profundidad o a segmentos muy cortos.

7.3.2.3 Explicación de las funciones

- InicializarLienzoYVentana
  - Crea un nuevo bitmap del tamaño del PictureBox, lo asigna a picCanvas.Image y define ventanaRecorte como un rectángulo centrado con un margen de 100 píxeles.
  - Llama a DibujarVentana para dibujar el rectángulo visible en azul claro y su borde negro.
- DibujarVentana
  - Limpia el bitmap con fondo blanco.
  - Rellena ventanaRecorte con un color suave mediante FillRectangle y dibuja su contorno usando DrawRectangle con un lápiz negro de grosor 2.
  - Refresca el PictureBox con picCanvas.Invalidate.
- PuntoDentro
  - Recibe un PointF y devuelve true si sus coordenadas X e Y se encuentran entre Left y Right, y entre Top y Bottom de ventanaRecorte, respectivamente; en caso contrario devuelve false.
  - SegmentoFueraBBox

    - Calcula los valores mínimo y máximo de X e Y para los puntos extremos del segmento.
    - Si la caja del segmento queda completamente a la izquierda, derecha, arriba o abajo de la ventana (sin solapamiento), devuelve true indicando que el segmento está fuera y se puede rechazar sin subdividir.
- Distancia
    - Calcula la longitud euclidiana entre dos puntos usando la raíz cuadrada de la suma de cuadrados de las diferencias en X y Y; se usa como criterio de parada cuando el segmento es muy corto.
- ClampToWindow
    - Limita las coordenadas de un punto al rango de la ventana, utilizando funciones de máximo y mínimo sobre Left, Right, Top y Bottom.
    - Sirve para aproximar el punto de intersección con el borde cuando se ha alcanzado la profundidad máxima o un segmento muy pequeño.
- RecortePuntoMedio
    - Implementa la lógica recursiva del recorte por subdivisión de punto medio.
    - Actualiza maxDepthUsada si la profundidad actual es mayor que la previamente registrada.
    - Evalúa si ambos extremos del segmento están dentro de la ventana; en ese caso añade el segmento completo a la lista visibles y termina.
    - Aplica SegmentoFueraBBox; si el resultado es true, el segmento se rechaza porque no puede cruzar la ventana.
    - Comprueba si la profundidad ha alcanzado MAX_DEPTH o si la longitud del segmento es menor que 1 píxel; si se cumple, y al menos uno de los extremos está dentro, aproxima el recorte ajustando ambos puntos con ClampToWindow y, si los dos quedan dentro, añade ese tramo aproximado como segmento visible.
    - Si ninguna de las condiciones anteriores se cumple, calcula el punto medio m entre a y b y llama recursivamente a RecortePuntoMedio para los sub-segmentos [a, m] y [m, b], aumentando la profundidad.
- picCanvas_MouseDown
    - Se ejecuta cuando el usuario presiona el botón del mouse sobre el PictureBox.
    - Activa dibujandoLinea, almacena pInicio con la posición inicial y, de forma preventiva, inicializa pFin con el mismo valor.
- picCanvas_MouseMove
    - Mientras dibujandoLinea sea true, actualiza pFin con la posición actual del mouse; el formulario no hace previsualización, pero mantiene las coordenadas al día para usarlas al soltar el botón.
- picCanvas_MouseUp

    - Si dibujandoLinea es false, no hace nada. En caso contrario, desactiva dibujandoLinea y guarda la posición final en pFin.
    - Crea una lista vacía visibles y reinicia maxDepthUsada a 0.
    - Dibuja la línea completa entre pInicio y pFin en rojo sobre el bitmap.
    - Llama a RecortePuntoMedio con los extremos convertidos a PointF; la función va añadiendo a visibles los sub-segmentos que quedan dentro de la ventana.
    - Recorre la lista visibles y dibuja cada uno en azul con un lápiz de mayor grosor.
    - Calcula la longitud de la línea original y actualiza lblInfo con la longitud, el número de trozos visibles y la profundidad máxima de recursión alcanzada.
    - Finalmente, refresca el PictureBox con picCanvas.Invalidate.
- btnClean_Click
    - Llama a InicializarLienzoYVentana para crear un nuevo bitmap, redibujar la ventana y borrar cualquier línea existente.
- btnClose_Click
    - Cierra el formulario.

7.3.3 Casos de prueba

1. El usuario abre el formulario, observa la ventana azul claro en el centro y dibuja una línea completamente dentro de dicho rectángulo (por ejemplo de la esquina superior izquierda a la esquina inferior derecha de la ventana).
   Resultado esperado: al soltar el mouse, la línea roja coincide con una única línea azul en el mismo lugar, ya que RecortePuntoMedio detecta que ambos extremos están dentro de la ventana y añade el segmento completo como visible en un solo paso. El label muestra trozos visibles=1 y una profundidad de recursión baja.

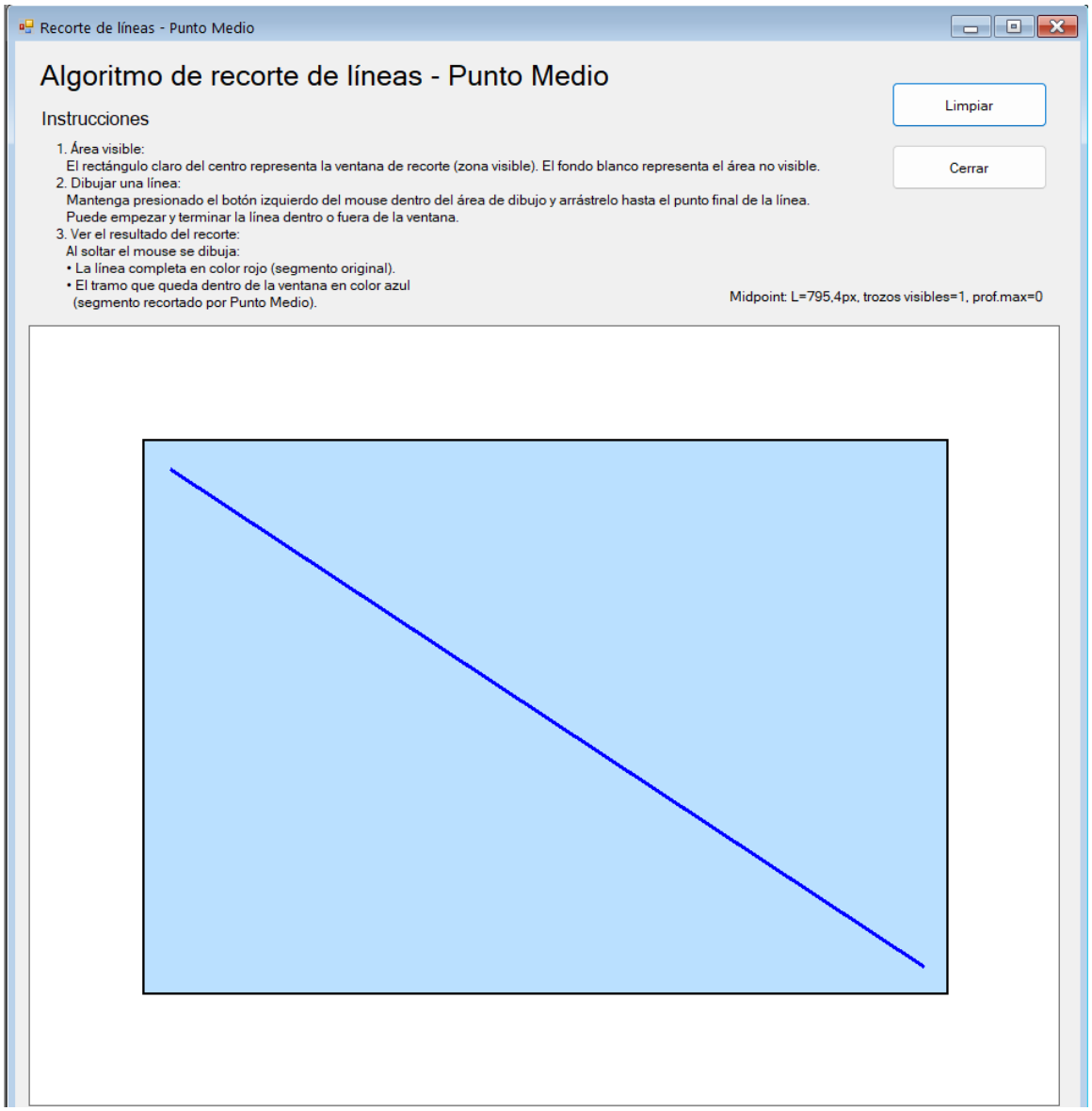


Ilustración 49. Caso de Prueba 1 - Algoritmo del Punto Medio

2. Sin limpiar, el usuario dibuja una línea completamente fuera de la ventana, por ejemplo muy por encima del rectángulo azul claro (ambos puntos exteriores en la misma zona).
   Resultado esperado: la línea roja se dibuja en la región blanca, pero no aparece ningún tramo azul; SegmentoFueraBBox detecta que la caja del segmento no intersecta la ventana y RecortePuntoMedio la rechaza sin subdividir más. El formulario sigue funcionando con normalidad y lblInfo indica trozos visibles=0.

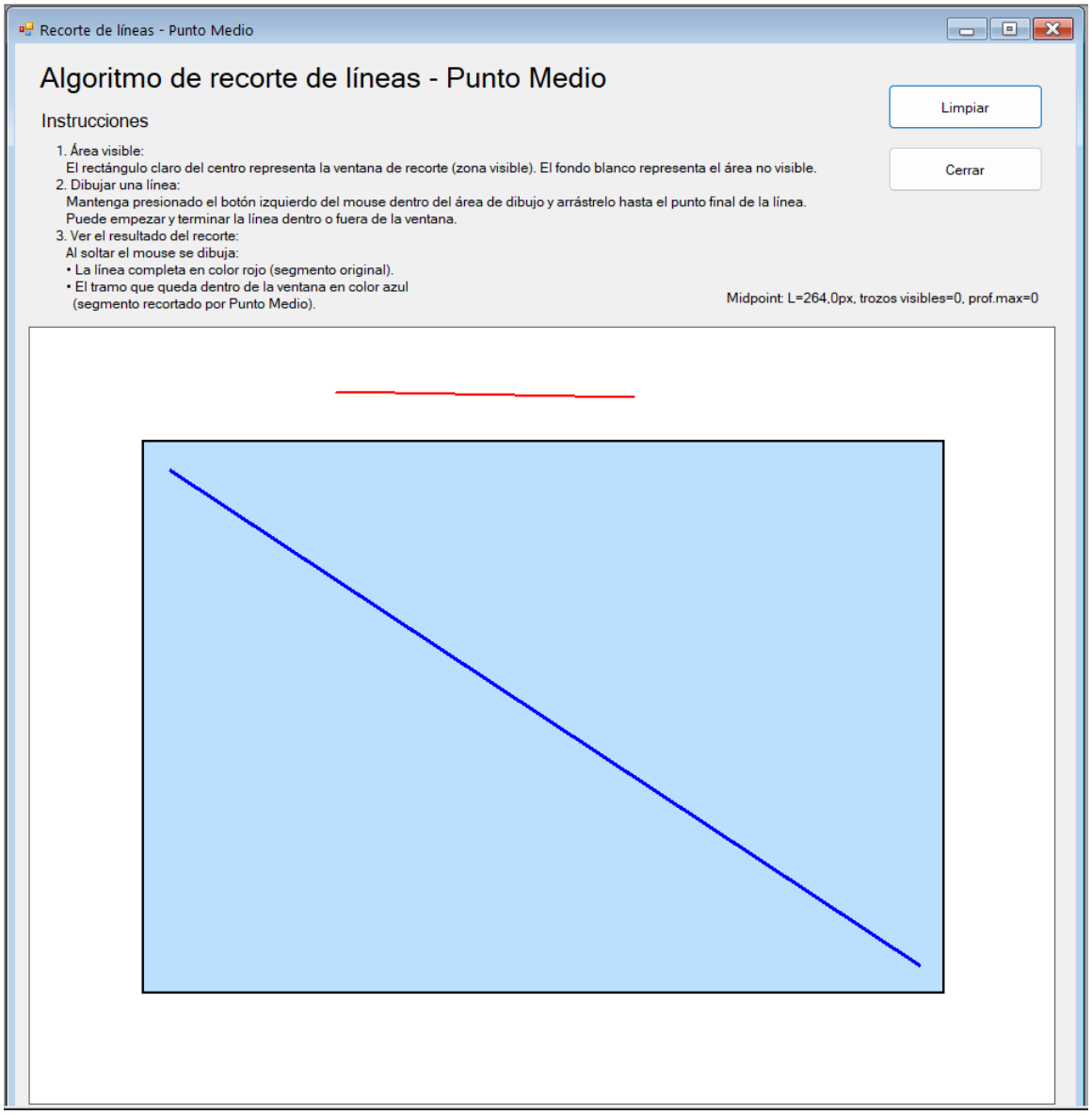


Ilustración 50. Caso de prueba 2 - Algoritmo del Punto Medio

3. Ahora el usuario dibuja una línea que cruza la ventana, comenzando fuera del rectángulo (por ejemplo a la izquierda) y terminando también fuera (por ejemplo a la derecha), de forma que el segmento atraviese el área azul clara.
   Resultado esperado: la línea roja aparece completa, atravesando el rectángulo. RecortePuntoMedio subdivide recursivamente el segmento calculando puntos medios hasta detectar las partes que quedan completamente dentro o fuera de la ventana; los tramos visibles se dibujan en azul siguiendo el borde del rectángulo con buena aproximación, sin que se recorten demasiado hacia el interior gracias al uso de MAX_DEPTH y ClampToWindow. El label muestra trozos visibles=1 (o más, si la línea entra y sale varias veces) y una profundidad máxima de recursión que no supera el valor de MAX_DEPTH.

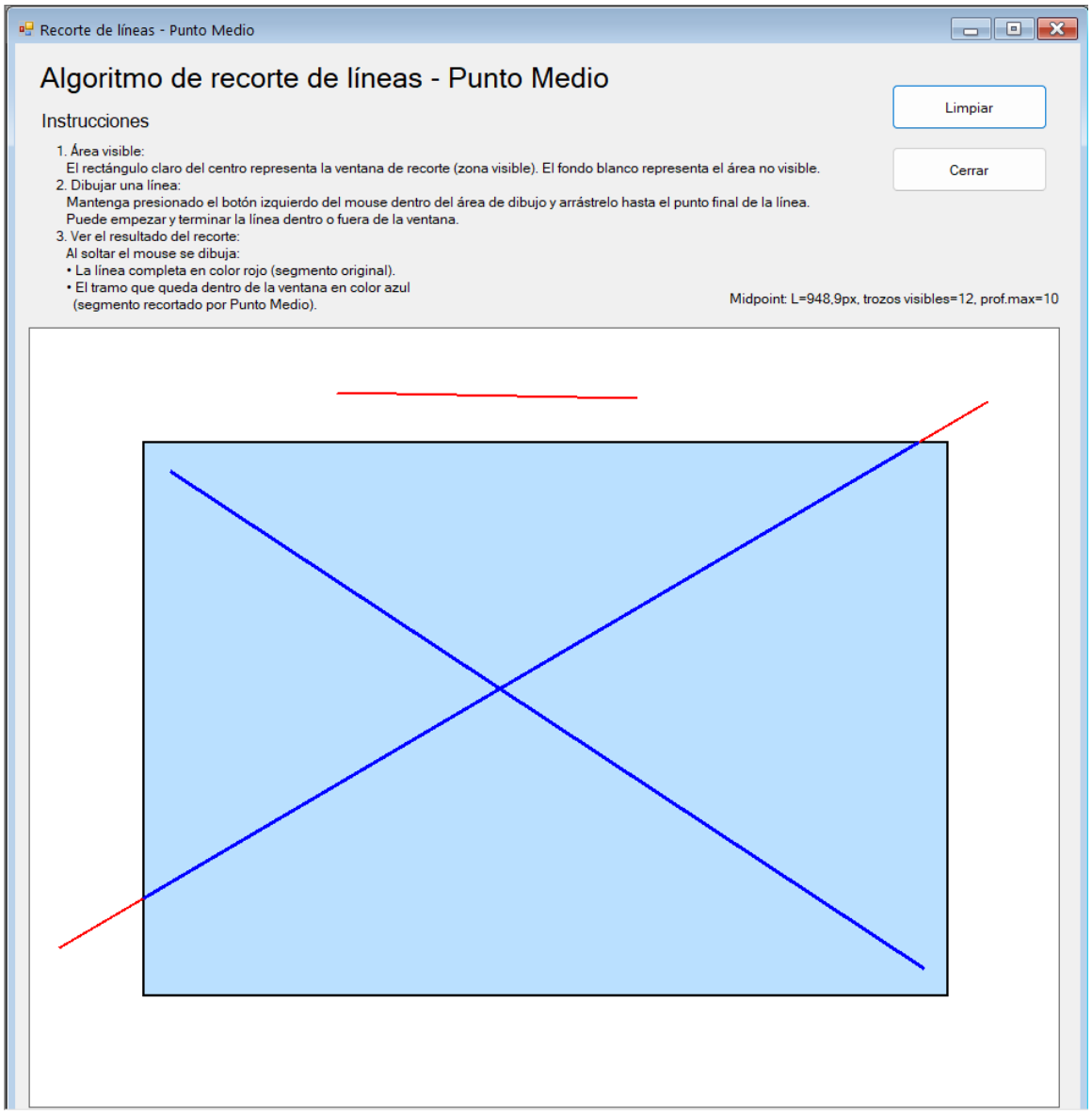


Ilustración 51. Caso de prueba 3 - Algoritmo del Punto Medio

8. Algoritmos de recorte de polígonos

Para el recorte de polígonos contra una ventana rectangular se implementan el algoritmo de Sutherland–Hodgman, algoritmo de recorte por segmentos aplicando Cohen Sutherlang y aplicando Liang Barsky.

8.1 Algoritmo de Sutherland–Hodgman

El algoritmo de Sutherland–Hodgman recorta un polígono sujeto contra un polígono de recorte convexo procesando cada borde de la ventana de manera secuencial. Para cada borde se recorre la lista de vértices del polígono de entrada y, según si los puntos están

dentro o fuera, se añaden al polígono de salida los vértices o los puntos de intersección correspondientes (GeeksforGeeks, 2024)

8.1.1 Justificación de la variante del algoritmo

El algoritmo de Sutherland–Hodgman es un método clásico para recortar polígonos contra una ventana convexa. Funciona tomando la lista de vértices del polígono de entrada y, de forma secuencial, la va filtrando contra cada borde de la ventana (izquierda, derecha, arriba y abajo), generando una nueva lista de vértices que representa solo la parte del polígono que queda dentro.

Se eligió esta variante porque es estándar en la literatura de computación gráfica, es relativamente sencilla de implementar con listas de puntos y se integra bien con una ventana rectangular fija. Además, permite comparar su enfoque basado en "in/out" de vértices con algoritmos de recorte de líneas como Cohen–Sutherland y Liang–Barsky usados en el resto del proyecto.

8.1.2 Descripción del Formulario

8.1.2.1 Diseño del formulario

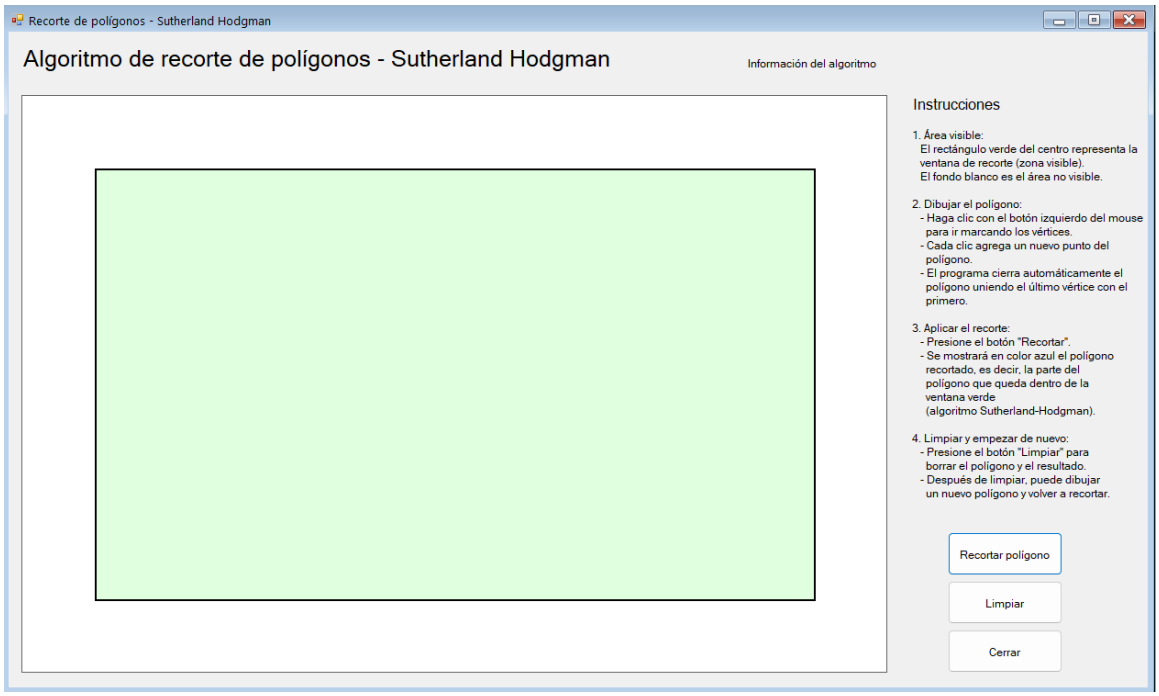


Ilustración 52. Formulario para el algoritmo de recorte de polígonos de Sutherland Hodgman

- Zona central: un PictureBox que actúa como lienzo de dibujo. Dentro se muestra un rectángulo verde claro que representa la ventana de recorte; el resto del área es fondo blanco.
- Columna derecha:
    - Título "Instrucciones" con pasos numerados para dibujar el polígono, aplicar el recorte, limpiar y empezar de nuevo.

    - Botón “Recortar polígono” para ejecutar el algoritmo de Sutherland–Hodgman.
    - Botón “Limpiar” para borrar el polígono y el resultado.
    - Botón “Cerrar” para salir del formulario.
- Encima del lienzo se muestra el título del algoritmo y un label informativo que indica el número de vértices de entrada y salida después del recorte.
- Visualmente, el polígono original se dibuja en rojo (contorno y vértices) y el polígono recortado se muestra relleno en azul translúcido dentro de la ventana verde.

8.1.2.2 Parámetros utilizados

- lienzo: bitmap asociado al PictureBox donde se dibujan la ventana de recorte, el polígono original y el polígono recortado.
- ventanaRecorte: rectángulo que define la región visible; se centra en el área de dibujo usando un margen fijo de 80 píxeles.
- poligonoOriginal: lista de vértices PointF agregados por el usuario mediante clics sobre el lienzo.
- poligonoClipped: lista de vértices PointF que contiene el resultado del algoritmo Sutherland–Hodgman después de recortar contra la ventana.
- Enumeración BordeClip: indica contra qué borde se está recortando en cada fase (Left, Right, Top, Bottom).
- Label lblInfo: muestra un mensaje del tipo “vértices entrada = n, vértices salida = m” para verificar el efecto del recorte.

8.1.2.3 Explicación de las funciones

- RedibujarTodo
    - Limpia el bitmap y dibuja la ventana de recorte rellenándola en verde claro y trazando su borde negro.
    - Dibuja el polígono original: si hay al menos dos vértices, traza las líneas en rojo y, en todo caso, marca cada vértice con un pequeño círculo rojo.
    - Si poligonoClipped tiene al menos tres vértices, dibuja el polígono recortado relleno en azul translúcido y traza su contorno azul.
    - Actualiza la imagen en pantalla mediante picCanvas.Invalidate.
- picCanvas_MouseClick
    - Cada clic izquierdo sobre el PictureBox agrega un nuevo vértice (PointF) a poligonoOriginal.
    - Al añadir un vértice, se limpia poligonoClipped para invalidar cualquier recorte previo y se actualiza lblInfo con el número de vértices de entrada.
    - Llama a RedibujarTodo para mostrar el nuevo vértice y las líneas del polígono original.
- btnClip_Click
    - Verifica que el polígono tenga al menos 3 vértices; en caso contrario, muestra un mensaje de error.

    - Construye una lista entrada a partir de poligonoOriginal y asegura que el polígono de entrada esté cerrado añadiendo el primer vértice al final si es necesario.
    - Llama a SutherlandHodgmanClip con la lista de vértices y la ventana de recorte; el resultado se asigna a poligonoClipped.
    - Actualiza lblInfo indicando el número de vértices de entrada y salida y vuelve a dibujar todo para que se vea el polígono recortado.
- btnClean_Click
    - Vacía las listas poligonoOriginal y poligonoClipped, reinicia lblInfo y llama a RedibujarTodo para dejar el lienzo únicamente con la ventana de recorte.
- btnClose_Click
    - Cierra el formulario.
- SutherlandHodgmanClip
    - Implementa el esquema del algoritmo Sutherland–Hodgman: recibe un polígono y una ventana rectangular y va llamando a ClipContraBorde en el orden izquierda, derecha, arriba y abajo.
    - El resultado final es la intersección del polígono original con la ventana, representada como una nueva lista de vértices.
- ClipContraBorde
    - Recorre la lista de vértices entrada y, para cada arista S→P, evalúa si el vértice actual y el anterior están dentro o fuera respecto del borde actual mediante EstaDentro.
    - Aplica las reglas clásicas del algoritmo:
        - in → in: agrega solo P.
        - out → in: agrega el punto de intersección I y luego P.
        - in → out: agrega solo I.
        - out → out: no agrega nada.
    - Devuelve la lista salida, que será la entrada de la siguiente fase de recorte.
- EstaDentro
    - Evalúa si un punto está dentro de la ventana en relación con un borde específico, comparando su coordenada X o Y con Left, Right, Top o Bottom.
- Interseccion
    - Calcula el punto de corte entre el segmento S–P y el borde indicado usando interpolación lineal en X o Y dependiendo del borde.
    - Devuelve un nuevo PointF con las coordenadas del punto de intersección que se insertará en la lista de vértices recortados.

8.1.3 Casos de prueba

1. El usuario hace varios clics dentro de la ventana verde para definir un polígono con al menos un “pico” que sobresalga hacia fuera (por ejemplo, algunos vértices dentro y uno o dos vértices cerca del borde derecho, cruzando ligeramente la ventana), luego presiona el botón “Recortar Polígono”.

Resultado esperado: el polígono rojo original permanece dibujado, pero aparece un polígono azul relleno que coincide con la parte del original que queda dentro de la ventana. Las aristas que cruzan el borde se sustituyen por segmentos que terminan exactamente en la frontera rectangular. lblInfo muestra que el número de vértices de salida es diferente (generalmente mayor) que el de entrada, lo que confirma que se generaron vértices de intersección y que el recorte Sutherland–Hodgman se ejecutó correctamente.

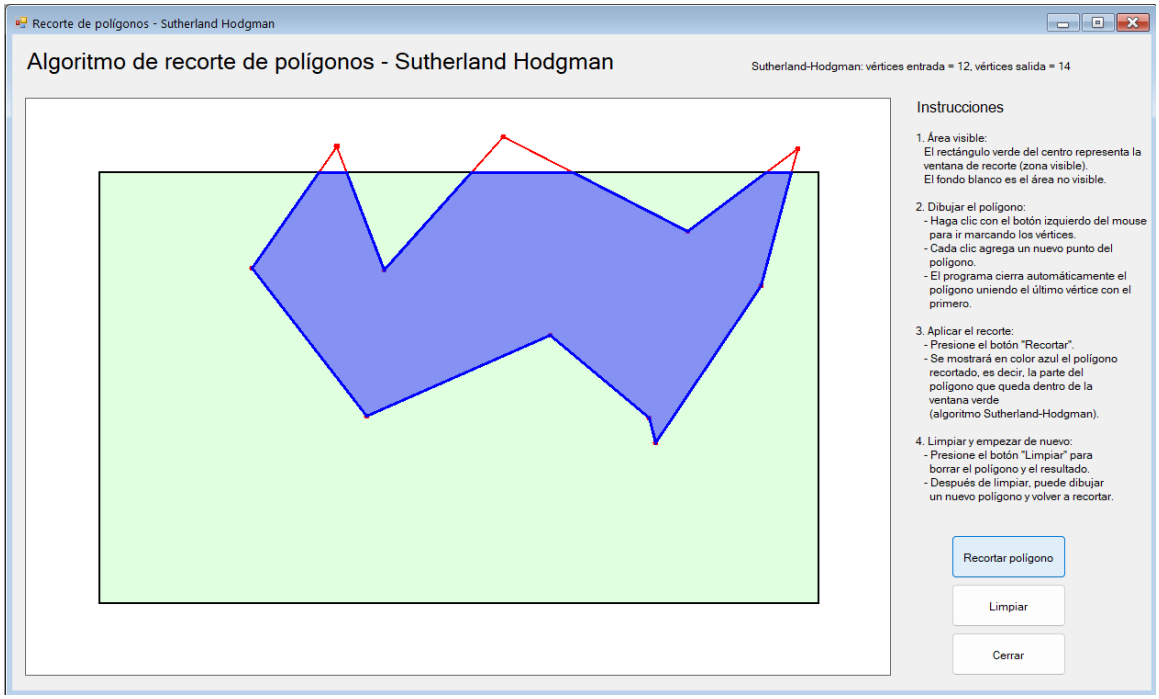


Ilustración 53. Caso de prueba - Algoritmo Sutherland Hodgman

8.2 Algoritmo de recorte por segmentos con Cohen-Sutherland

Como segunda variante, se implementa un algoritmo incremental específico para ventanas rectangulares. Este algoritmo recorre las aristas del polígono original, detecta intersecciones con los cuatro bordes de la ventana y reconstruye el polígono visible añadiendo vértices interiores e intersecciones. Aunque no corresponde a un algoritmo clásico con nombre propio, sigue los principios generales de los algoritmos de recorte de polígonos descritos en la literatura (ERA Ingeniería, 2025).

8.2.1 Justificación de la variante del algoritmo

El método de *Cohen–Sutherland* recorta segmentos contra una ventana rectangular usando códigos de región de 4 bits para cada punto. Esto permite aceptar o rechazar rápidamente segmentos completos o parciales con operaciones bit a bit, y calcular solo las intersecciones necesarias con los bordes.

8.2.2 Descripción del Formulario

8.2.2.1 Diseño del formulario

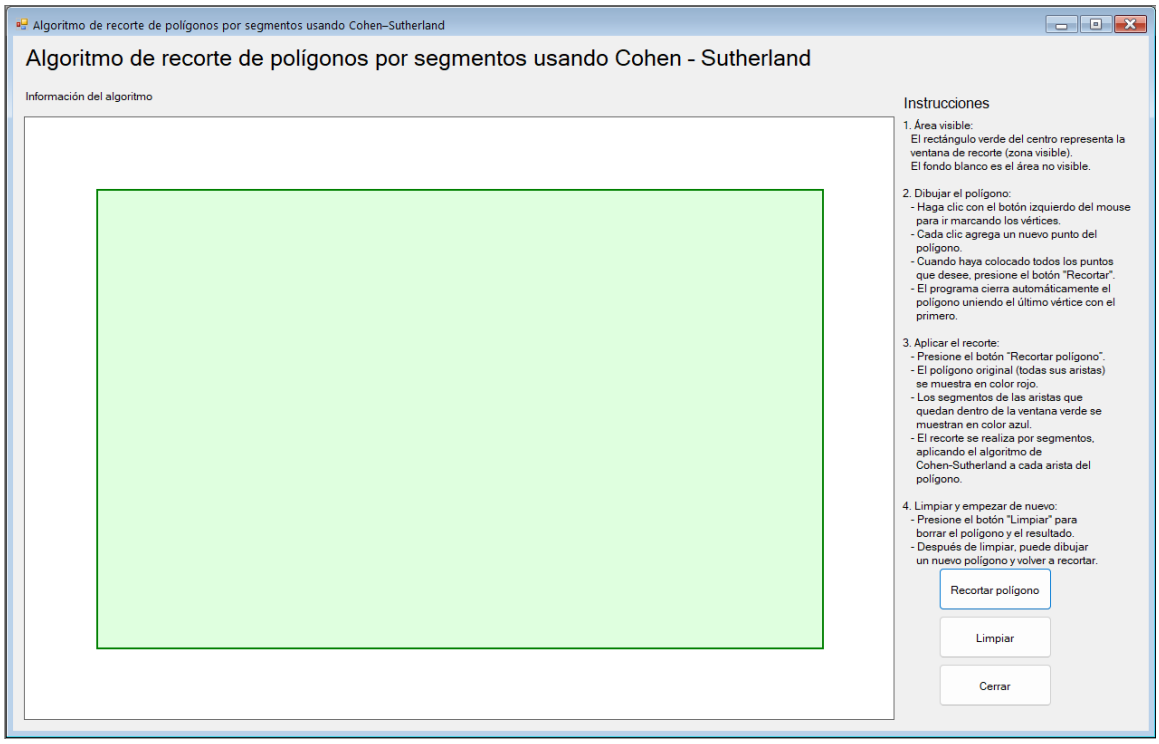


Ilustración 54. Formulario para el algoritmo de recorte de polígonos por segmentos con Cohen Sutherland

- Área de dibujo: (PictureBox) con un rectángulo verde claro representando la ventana de recorte. A la derecha están los botones: Recortar por segmentos, Limpiar y Cerrar.
- Entrada: el usuario hace clics para definir los vértices de un polígono.
- Salida: se dibuja el polígono original en rojo y, al aplicar el recorte, los tramos visibles de cada lado en azul.

8.2.2.2 Parámetros utilizados

- polígono: lista de puntos ingresados por el usuario (vértices).
- ventanaRecorte: rectángulo central que define la zona visible.
- OutCode: enum de cuatro bits para clasificar cada punto respecto a la ventana.

8.2.2.3 Explicación de las funciones

- ComputeOutCode(x, y): calcula el código de región de un punto comparando sus coordenadas con los límites de la ventana.
- CohenSutherlandClip(ref p0, ref p1): recorta un segmento y devuelve true si hay parte dentro de la ventana. Ajusta los puntos a las intersecciones con los bordes según sea necesario.
- RedibujarTodo: limpia el lienzo, dibuja la ventana de recorte, el polígono y sus vértices.
- btnClip_Click: recorre cada par de vértices consecutivos y aplica Cohen–Sutherland a cada segmento. Dibuja en azul los segmentos visibles.

- btnClean_Click: borra el polígono y redibuja solo la ventana.

8.2.3 Casos de prueba

1. El usuario dibuja un polígono con varios vértices, luego presiona el botón “Recortar Polígono”.
   Resultado esperado: por cada lado del polígono:
   - Si está totalmente dentro de la ventana, se dibuja azul completo.
   - Si está totalmente fuera, no se dibuja nada.
   - Si cruza la ventana, solo aparece la parte visible en azul.

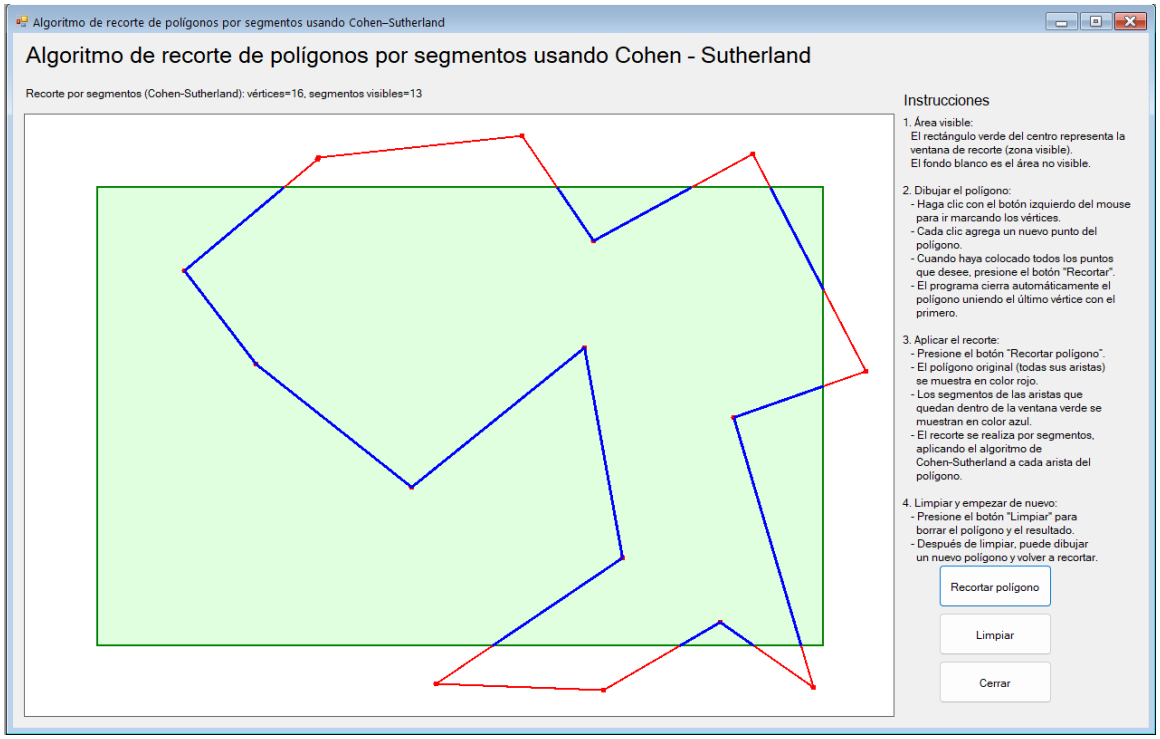


Ilustración 55. Caso de prueba 1 - Algoritmo recorte por segmentos Cohen Sutherland

8.3 Algoritmo de recorte por segmentos con Liang-Barsky

Se recorta cada arista del polígono como si fuera una línea: con la forma paramétrica y las 4 desigualdades de la ventana rectangular, se obtiene el intervalo visible $[t_E, t_L]$; si existe, se conserva $P(t_E) \rightarrow P(t_L)$, si no, de descarta. (Wikipedia contributors, 2025)

8.3.1 Justificación de la variante del algoritmo

El algoritmo de Liang–Barsky recorta segmentos usando la forma paramétrica de la línea y un rango [t0, t1] que indica los puntos de entrada y salida respecto a la ventana rectangular. Esto reduce comparaciones y cálculos de intersección frente a Cohen–Sutherland, porque cada borde de la ventana solo actualiza t0 o t1 en vez de ir cortando repetidamente el segmento.

En esta variante se aplica Liang–Barsky a cada arista del polígono por separado. Así se

reutiliza una única rutina de recorte de líneas, se simplifica el código y se facilita comparar, en el lienzo, qué tramos del polígono original sobreviven al recorte y cuáles se eliminan.

8.3.2 Descripción del Formulario

8.3.2.1 Diseño del formulario

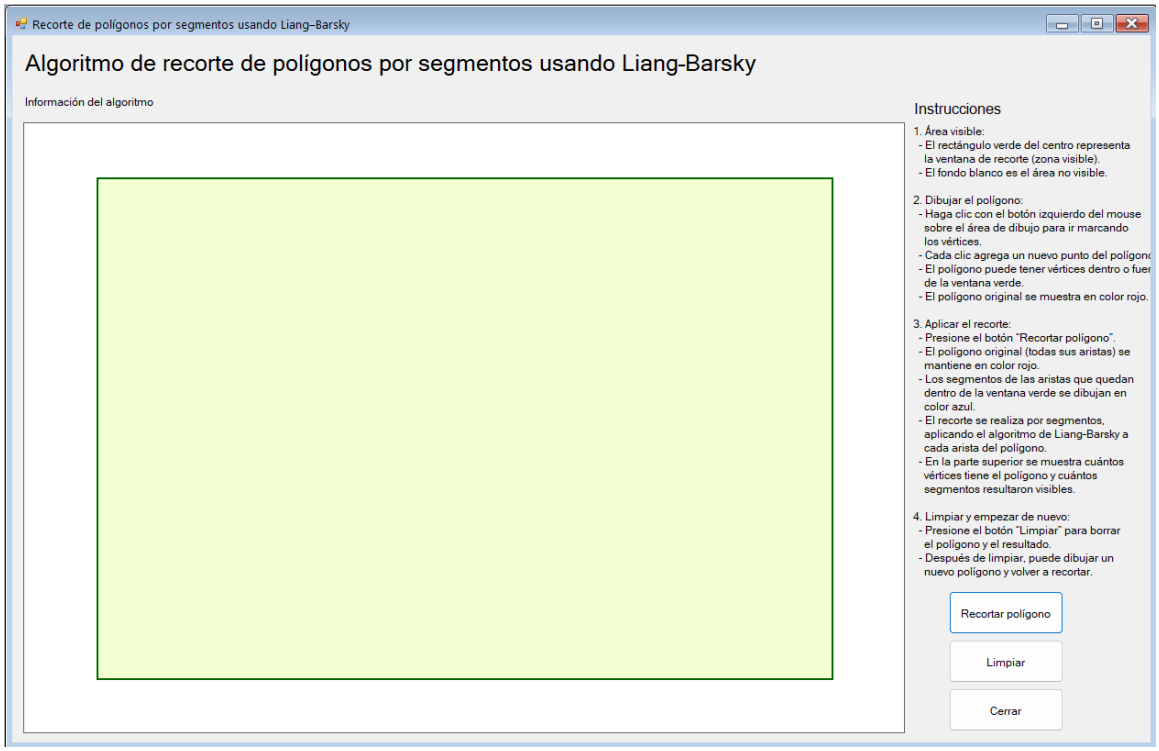


Ilustración 56. Formulario para el algoritmo de recorte de polígonos usando Liang Barsky

- Área principal de dibujo: un PictureBox donde se dibuja la ventana de recorte (rectángulo verde claro) y el polígono.
- Ventana de recorte: rectángulo centrado dentro del PictureBox con borde verde oscuro; representa la zona visible.
- Panel derecho con controles:
    - Botón “Recortar polígono”: aplica Liang–Barsky a todas las aristas.
    - Botón “Limpiar”: borra el polígono y redibuja solo la ventana.
    - Botón “Cerrar”: cierra el formulario.
- Etiqueta lblInfo en la parte superior derecha: muestra el número de vértices y cuántos segmentos quedan visibles tras el recorte.
- Interacción: cada clic izquierdo en el área de dibujo agrega un vértice nuevo al polígono, que se dibuja en rojo y se cierra visualmente uniendo último con primero.

8.3.2.2 Parámetros utilizados

- lienzo: bitmap donde se dibuja la ventana, el polígono y los segmentos recortados.

- ventanaRecorte: rectángulo que define xmin, ymin, ancho y alto de la ventana de clipping.
- vertices: lista de puntos PointF que almacena los vértices del polígono en orden de inserción.
- En LiangBarskyClip:
    - p0, p1: extremos originales del segmento a recortar.
    - dx, dy: diferencias en x e y entre p1 y p0.
    - t0, t1: parámetros iniciales 0 y 1 que se actualizan para representar el tramo visible.
    - c0, c1: extremos del segmento recortado que se devuelven si hay parte visible.
    - p, q, r dentro de ClipTest: valores auxiliares para probar cada borde de la ventana según las desigualdades del algoritmo.

8.3.2.3 Explicación de las funciones

- InicializarLienzoYVentana(): crea el bitmap, limpia el fondo y calcula ventanaRecorte con márgenes fijos; luego llama a RedibujarTodo.
- RedibujarTodo(): limpia el lienzo, dibuja la ventana verde, luego el polígono original en rojo (líneas y pequeños círculos para cada vértice).
- picCanvas_MouseClick(): cada clic izquierdo agrega un vértice a la lista vertices, actualiza la etiqueta lblInfo y vuelve a dibujar.
- LiangBarskyClip(): implementa el algoritmo de Liang–Barsky para un segmento:
    - Calcula dx y dy, inicializa t0 = 0 y t1 = 1.
    - Usa la función local ClipTest para cada borde (izquierda, derecha, arriba, abajo); cada llamada puede acotar t0 o t1 o rechazar el segmento.
    - Si tras las pruebas t0 ≤ t1, calcula los puntos recortados c0 y c1 a partir de p0 + t·(dx, dy).
- btnClip_Click(): valida que existan al menos 3 vértices, redibuja el polígono original y recorre todas las aristas (incluyendo el cierre entre último y primero). Para cada arista llama a LiangBarskyClip; si hay parte visible, dibuja ese tramo en azul y cuenta segmentos visibles. Actualiza lblInfo.
- btnClean_Click(): limpia la lista vertices, reinicia el lienzo llamando a InicializarLienzoYVentana y reinicia el mensaje de estado.
- btnClose_Click(): cierra el formulario.

8.3.3 Casos de prueba

1. El usuario dibuja un polígono con algunos vértices dentro y fuera del área de recorte, posteriormente presiona el botón “Recortar Polígono”.
   Resultado esperado: los tramos visibles del polígono se muestran en azul ajustados al borde del rectángulo, el resto permanece solo en rojo, y lblInfo indica el número de vértices totales y de segmentos visibles mayor que cero.

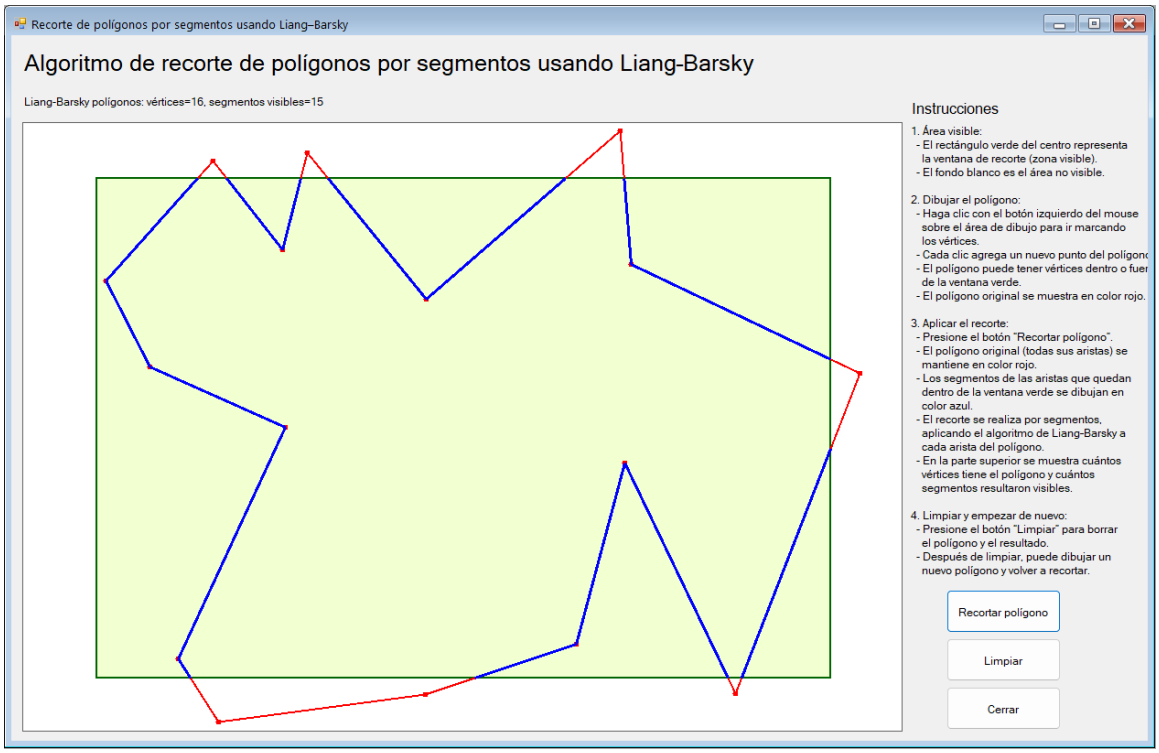


Ilustración 57. Caso de prueba 1 - Algoritmo de recorte por segmentos con Liang Barsky

9. Análisis comparativo de las variantes

Esta sección resume las principales ventajas y desventajas de las variantes implementadas, apoyándose en la literatura de computación gráfica (Hearn & Baker, 2014; Foley et al., 1996).

9.1 Trazado de líneas

El algoritmo DDA es sencillo y didáctico, pero utiliza operaciones en punto flotante. El algoritmo de Bresenham es más eficiente al usar únicamente enteros y es un estándar para el trazado de líneas en sistemas gráficos (Bresenham, 1965). El algoritmo del punto medio ofrece un razonamiento similar al de Bresenham, pero con un énfasis conceptual en la evaluación de la función de línea. (GeeksforGeeks, 2025)

9.2 Trazado de círculos

El algoritmo paramétrico polar calcula cada punto con las ecuaciones paramétricas del círculo por lo que es sencillo de entender, pero depende del tamaño del paso angular y de funciones trigonométricas costosas, lo que lo vuelve menos eficiente en hardware discreto. El algoritmo de punto medio aprovecha la simetría de 8 octantes y usa un criterio incremental entero para decidir el siguiente píxel, reduciendo operaciones y siendo una solución estándar de rasterización de círculos. El algoritmo de Bresenham para círculos sigue la misma idea incremental: trabaja solo con aritmética entera, genera los puntos de un octante y refleja por simetría, logrando una circunferencia aún más eficiente en tiempo de ejecución y adecuada para sistemas con recursos limitados.

9.3 Relleno

El algoritmo flood fill de 4 vecinos es intuitivo y adecuado para rellenar regiones a partir de una semilla, mientras que el algoritmo scanline es más eficiente y preciso para rellenar polígonos definidos por vértices. El algoritmo boundary fill, por su parte, rellena desde la semilla hasta encontrar un color de borde específico, por lo que es útil cuando se conoce claramente el contorno de la figura, aunque puede ser más sensible a pequeños huecos en el borde y al manejo de la recursión.

9.4 Recorte de líneas

El algoritmo de Cohen–Sutherland resulta útil porque permite aceptar o rechazar líneas de forma trivial a partir de sus códigos de región, recortando solo cuando realmente es necesario y funcionando muy bien con ventanas rectangulares. La variante paramétrica de Liang–Barsky es más eficiente cuando ya se trabaja con ecuaciones de recta y parámetros, ya que ajusta un rango [t0, t1] sobre el segmento y calcula directamente los puntos de entrada y salida sin repetir intersecciones innecesarias. Finalmente, el recorte por subdivisión de punto medio ofrece un enfoque más geométrico e intuitivo: divide recursivamente el segmento, descartando partes completamente fuera y conservando los tramos dentro de la ventana, lo que lo hace útil como algoritmo didáctico y para visualizar cómo se "corta" la línea paso a paso. (GeeksforGeeks, 2025).

9.5 Recorte de polígonos

El algoritmo de Sutherland–Hodgman es un estándar para el recorte de polígonos convexos y está ampliamente documentado, ya que procesa los vértices contra cada borde de la ventana y genera directamente el polígono recortado. El recorte por segmentos usando Cohen–Sutherland resulta útil cuando se quiere tratar cada lado del polígono como una línea independiente: aprovecha los códigos de región para aceptar, rechazar o recortar rápidamente cada segmento contra una ventana rectangular, aunque puede hacer más comprobaciones. Por su parte, la variante con Liang–Barsky aplica un enfoque paramétrico sobre cada lado del polígono, ajustando un rango [t0, t1] para obtener los puntos de entrada y salida en la ventana con menos cálculos de intersección, por lo que es más eficiente, pero también más matemático de implementar y menos intuitivo que Sutherland–Hodgman para entender el recorte a nivel de vértices.

10. Conclusiones

1. El desarrollo de los algoritmos clásicos de computación gráfica (trazado de líneas, círculos, relleno y recorte) permitió pasar de la teoría a una implementación completa y comprobable: se validó que cada algoritmo genera la misma solución geométrica con enfoques numéricos distintos (incrementales, paramétricos y recursivos), lo que refuerza la comprensión de cómo se rasterizan primitivas en un entorno discreto.

2. La comparación entre variantes (DDA vs. Bresenham vs. punto medio, paramétrico polar vs. punto medio vs. Bresenham en círculos, flood/boundary/scanline en relleno, y Cohen–Sutherland / Liang–Barsky / punto medio y Sutherland–Hodgman vs. recorte por segmentos en polígonos) evidencia el típico compromiso entre simplicidad, eficiencia y generalidad: algunos algoritmos son más intuitivos pero costosos, otros más eficientes pero matemáticamente más exigentes, lo que ayuda a elegir la técnica adecuada según las restricciones de la aplicación.
3. La construcción de formularios interactivos, validaciones de entrada y casos de prueba integrados demostró la importancia del diseño modular y robusto: separar cada algoritmo en su propio formulario, manejar errores de usuario y visualizar claramente los resultados (puntos, píxeles, segmentos recortados y polígonos rellenados) no solo mejora la usabilidad, sino que también facilita depurar, extender o reemplazar las implementaciones en futuros proyectos de gráficos por computadora.

Referencias bibliográficas

Algoritmo de línea de escaneo para el relleno de polígonos en gráficos de computadora. (n.d.). https://www-tutorialspoint-com.translate.goog/computer_graphics/computer_graphics_scan_line_algorithm.htm?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

Algoritmo de relleno de límites en gráficos por computadora. (n.d.). https://www-tutorialspoint-com.translate.goog/computer_graphics/computer_graphics_boundary_fill_algorithm.htm?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

Cohen-Sutherland Line Clipping in Computer Graphics. (n.d.). https://www.tutorialspoint.com/computer_graphics/computer_graphics_cohen_sutherland_line_clipping.htm

Dangi, R. (n.d.). *6 Circle Algorithm*. Scribd. https://es.scribd.com/document/619803661/6-Circle-Algorithm?utm_source=chatgpt.com

DDA Algorithm in Computer Graphics. (n.d.). https://www.tutorialspoint.com/computer_graphics/computer_graphics_dda_algorithm.htm?utm_source=chatgpt.com

ERA Ingeniería. (2025, March 21). *18 Algoritmo Cohen-Sutherland para el recorte de polígonos con la ventana* [Video]. YouTube. https://www.youtube.com/watch?v=nLcdSLGvL8c

GeeksforGeeks. (2022, March 19). *MidPoint Circle Drawing Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/mid-point-circle-drawing-algorithm/

GeeksforGeeks. (2024, April 8). *Polygon Clipping | Sutherland–Hodgman Algorithm*. GeeksforGeeks. https://www-geeksforgeeks-org.translate.goog/dsa/polygon-clipping-sutherland-hodgman-algorithm/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

GeeksforGeeks. (2025a, January 10). *Bresenham's circle drawing algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/c/bresenhams-circle-drawing-algorithm/?utm_source=chatgpt.com

GeeksforGeeks. (2025b, July 11). *Comparisons between DDA and Bresenham Line Drawing algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/competitive-programming/comparions-between-dda-and-bresenham-line-drawing-algorithm/

GeeksforGeeks. (2025c, July 15). *CohenSutherland vs. LiangBarsky line clipping algorithm*. GeeksforGeeks. https://www-geeksforgeeks-org.translate.goog/dsa/cohen-sutherland-vs-liang-barsky-line-clipping-algorithm/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

GeeksforGeeks. (2025d, July 23). *Bresenham's Line Generation Algorithm*. GeeksforGeeks. https://www-geeksforgeeks-org.translate.goog/dsa/bresenhams-line-generation-algorithm/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

GeeksforGeeks. (2025e, July 23). *DDA Line generation Algorithm in Computer Graphics*. GeeksforGeeks. https://www.geeksforgeeks.org/computer-graphics/dda-line-generation-algorithm-computer-graphics/

GeeksforGeeks. (2025f, July 23). *MidPoint Line Generation Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/mid-point-line-generation-algorithm/?utm_source=chatgpt.com

GeeksforGeeks. (2025g, November 6). *Flood Fill Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/flood-fill-algorithm/?utm_source=chatgpt.com

GeeksforGeeks. (2025h, November 6). *Flood Fill Algorithm*. GeeksforGeeks. https://www-geeksforgeeks-org.translate.goog/dsa/flood-fill-algorithm/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

Libretexts. (2025, November 21). *4.6: Parametric Lines*. Mathematics LibreTexts. https://math-libretexts-org.translate.goog/Bookshelves/Linear_Algebra/A_First_Course_in_Linear_Algebra_(Kuttler)/04%3A_R/4.06%3A_Parametric_Lines?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

Matthes, D., & Drakopoulos, V. (2022). Line Clipping in 2D: Overview, Techniques and Algorithms. *Journal of Imaging*, *8*(10), 286. https://doi.org/10.3390/jimaging8100286

Simple Snippets. (2025, October 9). *Bresenham Circle Drawing Algorithm Explained Step by Step | Computer Graphics Tutorial* [Video]. YouTube. https://www.youtube.com/watch?v=I2bQbH21XZo

TutorialsSpace- Er. Deepak Garg. (2020, July 31). *4.3- Trigonometric Or Parametric Equation of circle Drawing Algorithm In Computer Graphics In Hindi* [Video]. YouTube. https://www.youtube.com/watch?v=f6_UJHsEoSA

Wikipedia contributors. (2025, November 9). *Liang–Barsky algorithm*. Wikipedia. https://en.wikipedia.org/wiki/Liang%E2%80%93Barsky_algorithm?utm_source=chatgpt.com