Bresenham's Circle Drawing Algorithm

A Structured Study Note with Code Examples

Quick Notes

- An efficient algorithm for drawing circles on raster displays (pixel grids).
- Uses only **integer arithmetic** (addition, subtraction, bit-shifting), avoiding costly floating-point operations and square roots.
- Exploits **8-way symmetry**: calculates points for one octant (45-degree segment) and reflects them to get the full circle.
- Incremental calculation: determines the next pixel based on a decision parameter derived from the circle equation.

1. Initial Setup

We start by defining the circle's properties and the initial parameters for the algorithm. We focus on the second octant (from 90 to 45 degrees) and start at the top point of the circle.

- Circle Center $(x_c, y_c) = (0, 0)$ (For simplicity; can be translated later)
- Radius R
- Starting Point: $(x_0, y_0) = (0, R)$
- Initial Decision Parameter p_0: Derived from evaluating the circle equation $f(x, y) = x^2 + y^2 - R^2$ at the midpoint between the two potential next pixels $(1, R)$ and $(1, R-1)$. The midpoint is $(1, R-0.5)$.

  $$p_0 = f(1, R-0.5) = 1^2 + (R-0.5)^2 - R^2 = 1 + R^2 - R + 0.25 - R^2 = 1.25 - R$$

  To avoid fractions, we can use an integer-only version: $p_0 = 4 \times (1.25 - R) = 5 - 4R$. *(Note: Sometimes $p_0 = 3 - 2R$ is used, derived differently but leading to equivalent update rules with adjustments.)* Let's use $p_0 = 3 - 2R$ for consistency with the original example.
- **Example:** For $R = 5$:

  $$p_0 = 3 - 2 \times 5 = 3 - 10 = -7$$

2. Decision Parameter and Update Rules

At each step k, starting with (x_k, y_k), we decide whether the next point (x_{k+1}, y_{k+1}) is $(x_k + 1, y_k)$ (move East) or $(x_k + 1, y_k - 1)$ (move South-East). This decision is based on the sign of the decision parameter p_k.

Table 1: Update Rules Based on Decision Parameter p_k

Condition	Next Pixel Choice	Next Point (x_{k+1}, y_{k+1})	Update Rule for p_{k+1}
$p_k < 0$	Pixel $(x_k + 1, y_k)$ is closer	$(x_k + 1, y_k)$	$p_{k+1} = p_k + 4x_k + 6$
$p_k \geq 0$	Pixel $(x_k + 1, y_k - 1)$ is closer	$(x_k + 1, y_k - 1)$	$p_{k+1} = p_k + 4(x_k - y_k) + 10$

The calculation stops when $x \geq y$, as this marks the end of the second octant.

3. Step-by-Step Calculation (Example: R=5)

Given: $R = 5$, Center $(0, 0)$, Start Point $(0, 5)$, $p_0 = -7$.

Table 2: Calculation steps for R=5 using the specified update rules.

Step (k)	Current Point (x_k, y_k)	p_k	$p_k < 0$?	Next Point (x_{k+1}, y_{k+1})	Calculation for p_{k+1}
0	(0, 5)	-7	Yes	(1, 5)	$p_1 = p_0 + 4x_0 + 6$ $= -7 + 4(0) + 6 = -1$
1	(1, 5)	-1	Yes	(2, 5)	$p_2 = p_1 + 4x_1 + 6$ $= -1 + 4(1) + 6 = 9$
2	(2, 5)	9	No	(3, 4)	$p_3 = p_2 + 4(x_2 - y_2) + 10$ $= 9 + 4(2 - 5) + 10$ $= 9 - 12 + 10 = 7$
3	(3, 4)	7	No	(4, 3)	$p_4 = p_3 + 4(x_3 - y_3) + 10$ $= 7 + 4(3 - 4) + 10$ $= 7 - 4 + 10 = 13$
Stop: $x = 4, y = 3 \implies x > y$					

The points generated in the second octant are: $(0, 5), (1, 5), (2, 5), (3, 4), (4, 3)$. *(Note: The previous calculation in the prompt stopped early and had slightly different intermediate values, likely due to using $p_k + 4(x + 1) + 6$ instead of $p_k + 4x_k + 6$. The rules in Table 1 are standard.)*

4. Exploiting 8-Way Symmetry

The core strength of Bresenham's circle algorithm is calculating points for only one 45-degree segment (an octant) and then reflecting these points across the axes and diagonals ($y = x, y = -x$) to obtain the full circle. If (x, y) is a point calculated in the primary octant (e.g., the one from $x = 0$ to $x = y$), the following points are also on the circle:

Table 3: Mapping a Point (x, y) to All 8 Octants

Octant	Symmetric Point	Transformation from (x, y)
1 ($0° - 45°$)	(y, x)	Swap x and y
2 ($45° - 90°$)	(x, y)	Original point (calculated)
3 ($90° - 135°$)	$(-x, y)$	Negate x
4 ($135° - 180°$)	$(-y, x)$	Swap x and y, negate new x
5 ($180° - 225°$)	$(-y, -x)$	Swap x and y, negate both
6 ($225° - 270°$)	$(-x, -y)$	Negate both
7 ($270° - 315°$)	$(x, -y)$	Negate y
8 ($315° - 360°$)	$(y, -x)$	Swap x and y, negate new y

Applying Symmetry to R=5 Example Points:
The calculated points were $(0, 5), (1, 5), (2, 5), (3, 4), (4, 3)$. Using the symmetry rules:

- From (0, 5): (5, 0), (0, 5), (0, 5), (-5, 0), (-5, 0), (0, -5), (0, -5), (5, 0) → Unique: $(\pm 5, 0), (0, \pm 5)$
- From (1, 5): (5, 1), (1, 5), (-1, 5), (-5, 1), (-5, -1), (-1, -5), (1, -5), (5, -1)
- From (2, 5): (5, 2), (2, 5), (-2, 5), (-5, 2), (-5, -2), (-2, -5), (2, -5), (5, -2)
- From (3, 4): (4, 3), (3, 4), (-3, 4), (-4, 3), (-4, -3), (-3, -4), (3, -4), (4, -3)
- From (4, 3): (3, 4), (4, 3), (-4, 3), (-3, 4), (-3, -4), (-4, -3), (4, -3), (3, -4) *(Note: (3,4) and (4,3) generate the same set due to symmetry around y=x)*

These points, when plotted relative to the center (x_c, y_c), form the complete circle.

5. Visualization (TikZ)

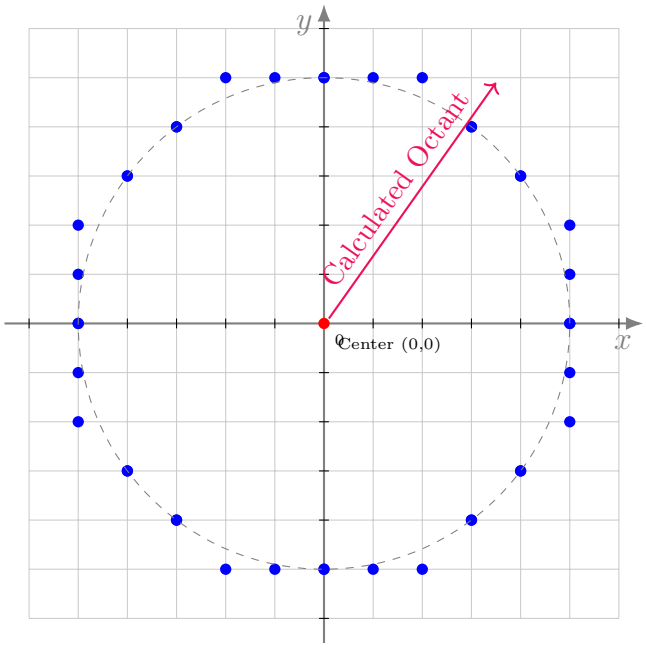

Figure 1: Plot of points for R=5 generated by Bresenham's algorithm, showing 8-way symmetry.

6. Code Implementations

6.1. C++ / OpenGL Snippet

This snippet shows the core Bresenham logic. It assumes a function 'drawPixel(x, y)' exists to plot a pixel relative to the circle center (cx, cy) in an OpenGL context.

C++ / OpenGL Implementation Snippet

```cpp
#include <GL/glut.h> // Or your preferred GL header

// Placeholder for actual pixel drawing function
void drawPixel(int x, int y) {
    // In a real scenario, you'd translate by (cx, cy)
    // and use appropriate OpenGL drawing commands.
    // Example: glVertex2i(x, y);
    glVertex2i(x, y);
}

// Function to draw all 8 symmetric points
void drawCirclePoints(int cx, int cy, int x, int y) {
    drawPixel(cx + x, cy + y);
    drawPixel(cx + y, cy + x);
    drawPixel(cx - x, cy + y);
    drawPixel(cx - y, cy + x);
    drawPixel(cx + x, cy - y);
    drawPixel(cx + y, cy - x);
    drawPixel(cx - x, cy - y);
    drawPixel(cx - y, cy - x);
}

// Bresenham's Circle Drawing Algorithm
void drawCircleBresenham(int cx, int cy, int r) {
    int x = 0;
    int y = r;
    int p = 3 - 2 * r; // Initial decision parameter

    // Start drawing points (typically within glBegin/glEnd)
    // glBegin(GL_POINTS); // Example

    // Draw the first set of points
    drawCirclePoints(cx, cy, x, y);

    // Calculate points for the second octant (0 to
       r/sqrt(2))
    while (x < y) {
        x++; // Always increment x
```

```
38          if (p < 0) {
39              // Move East
40              p = p + 4 * x + 6;
41          } else {
42              // Move South-East
43              y--;
44              p = p + 4 * (x - y) + 10;
45          }
46          // Draw the calculated point and its symmetric
                counterparts
47          drawCirclePoints(cx, cy, x, y);
48      }
49
50      // glEnd(); // Example
51  }
52
53  /*
54  // Example Usage in a GLUT display function:
55  void display() {
56      glClear(GL_COLOR_BUFFER_BIT);
57      glColor3f(0.0, 0.0, 1.0); // Set color to blue
58
59      glBegin(GL_POINTS); // Begin drawing points
60      drawCircleBresenham(0, 0, 50); // Draw circle at center
            (0,0) with radius 50
61      glEnd(); // End drawing points
62
63      glFlush();
64  }
65  */
```

6.2. Python Snippet

This Python code calculates the points in the second octant and provides a function to generate all symmetric points.

Python Implementation Snippet

```
1  import matplotlib.pyplot as plt
2
3  def bresenham_circle_octant(r):
4      """
5      Calculates points for the second octant (90 to 45
           degrees)
6      using Bresenham's circle algorithm.
7      Starts at (0, r).
8      """
9      x = 0
```

```
    y = r
    p = 3 - 2 * r  # Initial decision parameter
    points = []

    # Add the starting point
    points.append((x, y))

    # Iterate until x >= y (end of the octant)
    while x < y:
        x += 1
        if p < 0:
            # Select point (x, y) - move East
            p = p + 4 * x + 6
        else:
            # Select point (x, y-1) - move South-East
            y -= 1
            p = p + 4 * (x - y) + 10

        points.append((x, y))

    return points

def get_symmetric_points(x, y):
    """Generates all 8 symmetric points for a given point (x, y)."""
    return [
        (x, y), (y, x), (-x, y), (-y, x),
        (x, -y), (y, -x), (-x, -y), (-y, -x)
    ]

def get_all_circle_points(cx, cy, r):
    """
    Generates all points for a circle centered at (cx, cy) with radius r.
    """
    octant_points = bresenham_circle_octant(r)
    all_points = set() # Use a set to avoid duplicates

    for x, y in octant_points:
        symmetric = get_symmetric_points(x, y)
        for sx, sy in symmetric:
            # Translate points relative to the center (cx, cy)
            all_points.add((cx + sx, cy + sy))

    return list(all_points)

# --- Example Usage ---
radius = 5
center_x = 0
```

```
center_y = 0

# Calculate points for the second octant
octant2_points = bresenham_circle_octant(radius)
print(f"Points in the second octant (R={radius}):
    {octant2_points}")

# Calculate all points for the circle
circle_points = get_all_circle_points(center_x, center_y,
    radius)
print(f"\nTotal unique points for the circle:
    {len(circle_points)}")
# print(f"All points: {sorted(list(circle_points))}") #
    Uncomment to see all points

# --- Optional: Plotting with Matplotlib ---
def plot_circle(points):
    if not points:
        print("No points to plot.")
        return

    x_coords, y_coords = zip(*points) # Unzip points into x
        and y lists

    plt.figure(figsize=(6, 6))
    plt.scatter(x_coords, y_coords, color='blue', s=10) #
        Plot points
    plt.axhline(0, color='grey', lw=0.5)
    plt.axvline(0, color='grey', lw=0.5)
    plt.grid(True, linestyle='--', alpha=0.6)
    plt.gca().set_aspect('equal', adjustable='box') # Ensure
        aspect ratio is equal
    plt.title(f"Bresenham's Circle Points (R={radius},
        Center=({center_x},{center_y}))")
    plt.xlabel("X-axis")
    plt.ylabel("Y-axis")
    # Set limits slightly larger than radius
    lim = radius + 1
    plt.xlim(-lim, lim)
    plt.ylim(-lim, lim)
    plt.show()

# Plot the calculated points
# plot_circle(circle_points) # Requires matplotlib installed
```

7. Summary

Key Takeaways

- Bresenham's Circle Algorithm provides an extremely efficient method for rasterizing circles using only integer operations.
- Its efficiency stems from incremental calculations based on a decision parameter and exploiting the inherent 8-way symmetry of circles.
- The algorithm calculates points for one octant, and symmetry rules generate the remaining points.
- It remains a fundamental algorithm in computer graphics for drawing circles and ellipses.