Pixel-Based Graphical Displays

Most computer graphical displays consist of a rectangular grid (eg, 600×900) of square *pixels.*

Each pixel displays a single colour. Displays come in several varieties:

Black-and-white: Each pixel has only two possible values — black or white.

Gray-scale: Each pixel can be a shade of gray (eg, 256 shades from black to white).

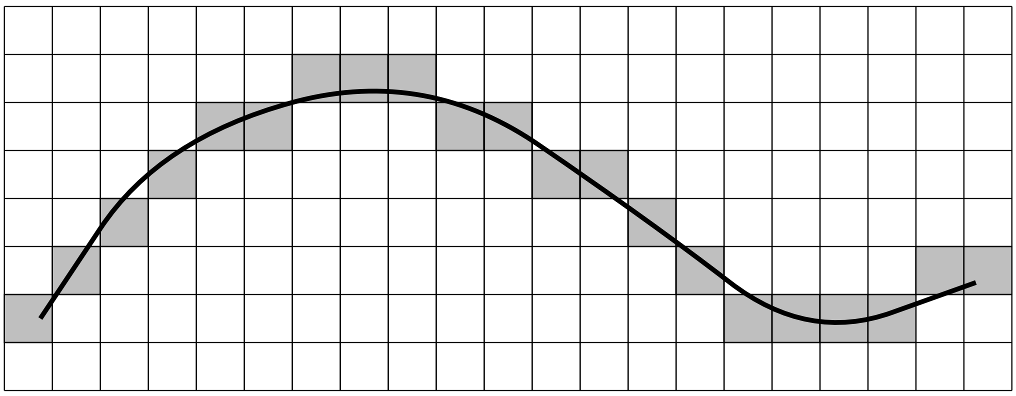
Colour-mapped: Each pixel can be a colour from some limited set (eg, out of 256 possible).

Full-colour: Each pixel can be any of a wide range of colours (eg, specified using 8 bits for red, green, and blue).

Rendering Curves as Pixels

When we display a curve on a pixel-based device, we need to colour a set of pixels that will approximate the curve.

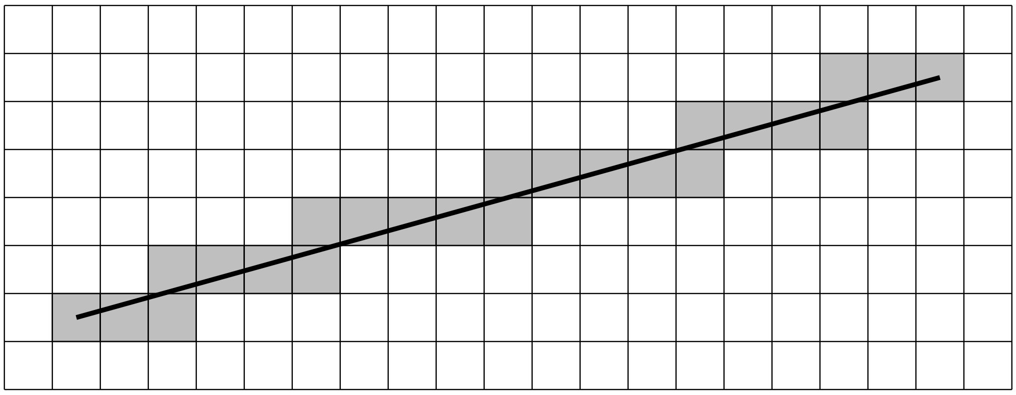
For example:



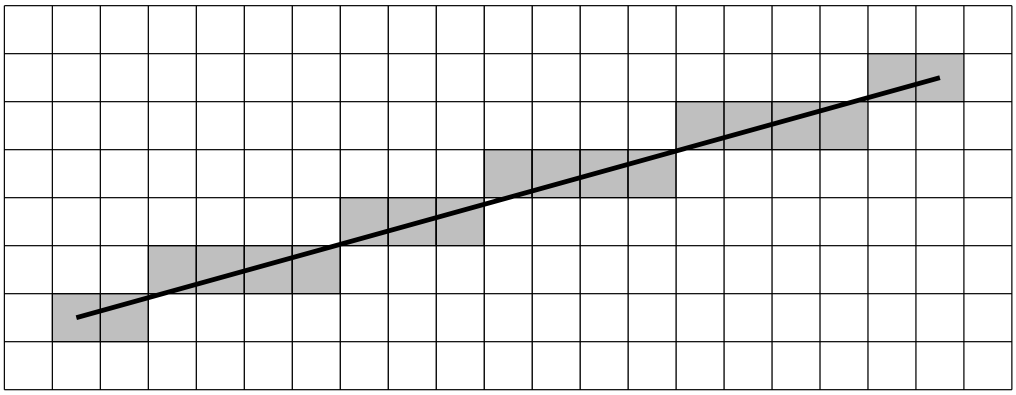
This rendering assumes that pixels can take on only two colours. One can do better by using intermediate shades, if the device can display them, but we won't talk about how to do that.

Two Renderings of a Line Segment

One that sets every pixel that the line crosses:



One that sets exactly one pixel in each column:



Is one of these better than the other?

General Strategies for Rendering Parametric Curves

Suppose we have a parametric representation of a curve, as $(x(t), y(t))$, for t over some range. How can we render it?

First strategy: For a sufficiently large number of values for t over its range

> Evaluate $x(t)$ and $y(t)$.
> Round these values to the nearest pixel.
> Set the pixel to the desired colour

Second strategy: For values of t over some range, determine the slope of the curve, and step one pixel along the axis that matches this most closely, setting a single pixel to the desired colour.

Rendering A Line Segment (1)

Following Fiume's book (section 1.5), we look at how to construct an efficient program for rendering a line segment.

We follow the second strategy, and assume that curve starts at $(0, 0)$ and has slope between 0 and 1. We can transform other cases to this one.

We also assume that the end-points are integers. Could this cause problems?

```
proc DrawLine (x, y : int; c : colour)
    var yt, m : real
    var yi : int
    m := y / x
    for xi : 0..x
        yt := m * xi
        yi := floor (yt + 1/2)
        PutPixel (xi, yi, c)
    end for
end DrawLine
```

This brute-force method is not very efficient.

Differential Computation of Curves

Suppose we need to evaluate $x(t)$ and $y(t)$ for many values of t: $t = t_0,\ t_0 + \delta,\ t_0 + 2\delta,\ \ldots$

Rather than evaluating $x(t)$ separately for each t, we can evaluate $x(t_0) = x_0$, and then compute

$$x_{i+1} \;=\; x(t_{i+1}) \;=\; x_i + \delta\,\frac{dx}{dt}(t_i)$$

This will be faster if dx/dt is easier to evaluate that $x(t)$ itself. For a line segment, dx/dt is a constant.

On the other hand, we might wonder about accumulated round-off error, and error from dx/dt varying as t varies over the interval δ.

Rendering A Line Segment (2)

We can transform program #1 to use differential computation. This replaces a multiplication by an addition, which is faster.

```
proc DrawLine (x, y : int; c : colour)
    var yt, m : real
    var yi : int
    yt := 0
    m := y / x
    for xi : 0..x
        yi := floor (yt + 1/2)
        PutPixel (xi, yi, c)
        yt += m
    end for
end DrawLine
```

Does this program do exactly the same thing as program #1?

Rendering A Line Segment (3)

We can eliminate the addition of 1/2 each time around the loop as follows:

```
proc DrawLine (x, y : int; c : colour)
    var ys, m : real
    var yi : int
    ys := 1/2
    m := y / x
    for xi : 0..x
        yi := floor(ys)
        PutPixel (xi, yi, c)
        ys += m
    end for
end DrawLine
```

Rendering A Line Segment (4)

Next, we split `ys` into two variables, `ysi` and `ysf`, giving the integer and fractional parts. The integer part directly specifies the y coordinate of the pixel to set:

```
proc DrawLine (x, y : int; c : colour)
    var ysf, m, mm : real
    var ysi : int
    m := y / x
    mm := m - 1
    ysi := 0
    ysf := 1/2
    for xi : 0..x
        PutPixel (xi, ysi, c)
        if (ysf+m>=1) then
            ysi += 1
            ysf += mm
        else
            ysf += m
        end if
    end for
end DrawLine
```

Rendering A Line Segment (5)

Now we get rid of the floating-point quantities `ysf`, `m`, and `mm`, replacing them by a sort of fixed-point representation scaled by `2*x`:

```
proc DrawLine (x, y : int; c : colour)
    var Ysf, M, MM : int
    var ysi : int
    M := 2*y
    MM := M - 2*x
    ysi := 0
    Ysf := x
    for xi : 0..x
        PutPixel (xi, ysi, c)
        if (Ysf+M>=2*x) then
            ysi += 1
            Ysf += MM
        else
            Ysf += M
        end if
    end for
end DrawLine
```

Does the change from floating-point to fixed-point representations change what the program does?

Rendering A Line Segment (6)

Finally, we define

```
r = Ysf + M - 2*x = 2*y + 2*(ysf-1)*x
```

This lets us replace the test for `Ysf + M >= 2*x` by a test for `r >= 0`.

We thus obtain *Bresenham's Algorithm*:

```
proc DrawLine (x, y : int; c : colour)
    var r, M, MM : int
    var ysi : int
    M := 2*y
    MM := M - 2*x
    ysi := 0
    r := 2*y - x
    for xi : 0..x
        PutPixel (xi, ysi, c)
        if (r>=0) then
            ysi += 1
            r += MM
        else
            r += M
        end if
    end for
end DrawLine
```

Rendering a Circle

We can render a circle using the first strategy mention before, using differential evaluation.

A circle of radius r centred at the origin can be represented parametrically as follows:

$$x(u) \; = \; r \cos(u), \qquad y(u) \; = \; r \sin(u)$$

The corresponding differential form is:

$$\frac{dx}{du}(u) \; = \; -r \sin(u), \qquad \frac{dy}{du}(u) \; = \; r \cos(u)$$

This may seem just as costly to use as the orginal, until you notice that it's equivalent to

$$\frac{dx}{du}(u) \; = \; -y(u), \qquad \frac{dy}{du}(u) \; = \; x(u)$$

Our strategy: Start with $x(0) = r$, $y(0) = 0$, and proceed from there by small steps in u, setting pixels as we go. We choose our step size to be small enough to not leave gaps.

First Implementation

To implement this, we need to settle some details.

After setting $x(0) = r$ and $y(0) = 0$, we can iterate the following, for some step size δ:

$$\begin{aligned} x(u+\delta) &= x(u) - y(u)\,\delta \\ y(u+\delta) &= y(u) + x(u)\,\delta \end{aligned}$$

We choose the step size, δ, to $1/r$. The total distance travelled each step should then be equal to 1, so we won't skip any pixels.

We need to continue for $2\pi/\delta$ steps in order to go all the way around.

The Result for $r = 12$

```
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + # # # # # # + + + + + + + + + + + + + + +
+ + + + + + + + + + + + # # + + + + + + # # # + + + + + + + + + + + +
+ + + + + + + + + + # # + + + + + + + + + + + # # + + + + + + + + + +
+ + + + + + + + # # + + + + + + + + + + + + + + + # + + + + + + + + +
+ + + + + + + # + + + + + + + + + + + + + + + + + # + + + + + + + + +
+ + + + + + # + + + + + + + + + + + + + + + + + + + # # + + + + + + +
+ + + + + + # + + + + + + + + + + + + + + + + + + + + # + + + + + + +
+ + + + + # + + + + + + + + + + + + + + + + + + + + + + # + + + + + +
+ + + + + # + + + + + + + + + + + + + + + + + + + + + + # + + + + + +
+ + + + # + + + + + + + + + + + + + + + + + + + + + + + + # + + + + +
+ + + + # + + + + + + + + + + + + + + + + + + + + + + + + # + + + + +
+ + + + # + + + + + + + + + + + + + + + + + + + + + + + + # + + + + +
+ + + # + + + + + + + + + + + + + + + + + + + + + + + + + # + + + + +
+ + + # + + + + + + + + + + + + + 0 + + + + + + + + + + + # + + + + +
+ + + # + + + + + + + + + + + + + + + + + + + + + + + + + + + + + # +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + # + +
+ + + # + + + + + + + + + + + + + + + + + + + + + + + + + + + + # + +
+ + + + # + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + # + + + + + + + + + + + + + + + + + + + + + + + + + + + # + +
+ + + + # + + + + + + + + + + + + + + + + + + + + + + + + + + # + + +
+ + + + + # + + + + + + + + + + + + + + + + + + + + + + + + + # + + +
+ + + + + # + + + + + + + + + + + + + + + + + + + + + + + + # + + + +
+ + + + + + # + + + + + + + + + + + + + + + + + + + + + + # + + + + +
+ + + + + + + # + + + + + + + + + + + + + + + + + + + + # + + + + + +
+ + + + + + + + # # + + + + + + + + + + + + + + + + + + # + + + + + +
+ + + + + + + + + # + + + + + + + + + + + + + + + + # # + + + + + + +
+ + + + + + + + + + # + # + + + + + + + + + + # # + + + + + + + + + +
+ + + + + + + + + + + + + # # # + + + + # # # + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + # # + # + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
```

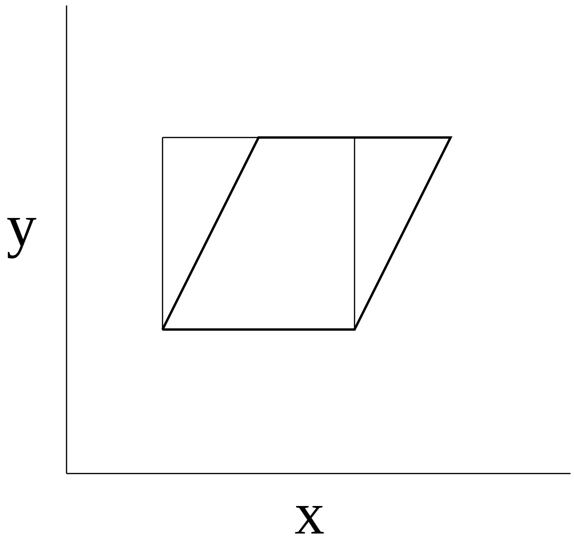
Second Implementation

We can stop the circle from spiralling outwards by changing the iteration to the following:

$$\begin{aligned} x(u+\delta) &= x(u) - y(u)\,\delta \\ y(u+\delta) &= y(u) + x(u+\delta)\,\delta \end{aligned}$$

Why does this work?

It stops regions of points from "expanding", because each step is a "shear" transformation that doesn't change the area of a region:



The Result for $r = 12$, New Version

```
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + # # # # # # # + + + + + + + + + + + + + +
+ + + + + + + + + + + + # # + + + + + + + # # # + + + + + + + + + + +
+ + + + + + + + + + + # # + + + + + + + + + + + # # + + + + + + + + +
+ + + + + + + + + # # + + + + + + + + + + + + + + # + + + + + + + + +
+ + + + + + + + # + + + + + + + + + + + + + + + + + # + + + + + + + +
+ + + + + + + + # + + + + + + + + + + + + + + + + + + # + + + + + + +
+ + + + + + + # + + + + + + + + + + + + + + + + + + + + # + + + + + +
+ + + + + + # # + + + + + + + + + + + + + + + + + + + + # + + + + + +
+ + + + + + # + + + + + + + + + + + + + + + + + + + + + + # + + + + +
+ + + + + # + + + + + + + + + + + + + + + + + + + + + + + # + + + + +
+ + + + + # + + + + + + + + + + + + + + + + + + + + + + + # + + + + +
+ + + + + # + + + + + + + + + + + + + + + + + + + + + + + # + + + + +
+ + + + + # + + + + + + + + + + + 0 + + + + + + + + + + + # + + + + +
+ + + + + # + + + + + + + + + + + + + + + + + + + + + + + # + + + + +
+ + + + + # + + + + + + + + + + + + + + + + + + + + + + + # + + + + +
+ + + + + # + + + + + + + + + + + + + + + + + + + + + + # + + + + + +
+ + + + + + # + + + + + + + + + + + + + + + + + + + + + # + + + + + +
+ + + + + + # + + + + + + + + + + + + + + + + + + + + + # + + + + + +
+ + + + + + # + + + + + + + + + + + + + + + + + + + + # + + + + + + +
+ + + + + + + # + + + + + + + + + + + + + + + + + + + # + + + + + + +
+ + + + + + + # # + + + + + + + + + + + + + + + + # # + + + + + + + +
+ + + + + + + + + # + + + + + + + + + + + + + + # + + + + + + + + + +
+ + + + + + + + + + # + + + + + + + + + + + + # # + + + + + + + + + +
+ + + + + + + + + + + # # # + + + + + + + # # + + + + + + + + + + + +
+ + + + + + + + + + + + + + # # # # # # # + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
```