Introduction to Line Drawing

Graphics 1 CMP-5010B

Dr. David Greenwood

david.greenwood@uea.ac.uk

SCI 2.16a University of East Anglia

Spring 2022

Contents

Lines

A **line** is an *infinitely* thin, infinitely long collection of points extending in two opposite directions.

Lines

A line **segment** has two *endpoints* and all the points of the line between them.

Lines

A **ray** is part of a line with one endpoint and extends infinitely in *one* direction.

Representing Lines

We will consider two line representations:

- Parametric, or vector form.
- Cartesian form.

Parametric Line Equation

A line can be defined as the set of all points in space that satisfy two criteria:

1. Contains a point, which we identify by a position vector $\mathbf{r}_0$.
2. The vector between $\mathbf{r}_0$ and *any* position vector $\mathbf{r}$ *on* the line, is **parallel** to a given vector $\mathbf{v}$.

Parametric Line Equation

The vector with initial point $\mathbf{r}_0$ and terminal point $\mathbf{r}$ is given by:

$$\mathbf{s} = \mathbf{r} - \mathbf{r}_0$$

This vector must be parallel to $\mathbf{v}$ hence, for some scalar λ:

$$\mathbf{s} = \lambda \mathbf{v}$$

Parametric Line Equation

Any position vector $\mathbf{r}$, corresponding to a point P on the line has the form:

$$\mathbf{r} = \mathbf{r}_0 + \lambda \mathbf{v}$$

where λ is a scalar called a *parameter*, and this is the **vector** equation.

Parametric Line Equation

$$\mathbf{r} = \mathbf{r}_0 + \lambda\mathbf{v}$$

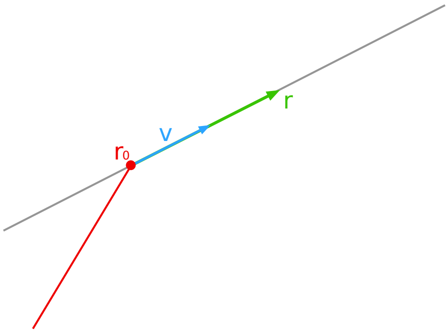

Figure 1: Parametric Line

Cartesian Line Equation

Algebraically, we can define a line with an **implicit** *linear* equation:

$$ax + by + c = 0$$

Cartesian Line Equation

We can derive the implicit form of the line equation from the vector equation.

- Consider coordinates of points on the line as vectors projected to the x-axis and y-axis.
- Apply the vector equation to the x-axis and y-axis projection to obtain the implicit form.

Cartesian Line Equation

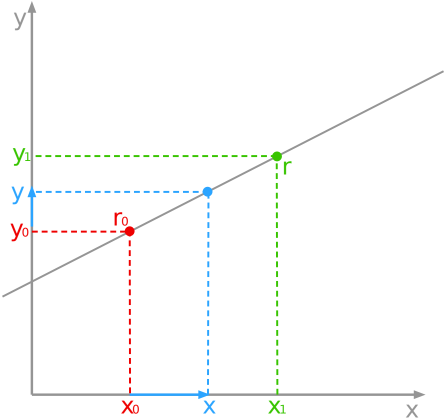

Figure 2: Parametric Line

Projecting to the x-axis and y-axis.

$$\mathbf{x} = \mathbf{x}_0 + \lambda(\mathbf{x}_1 - \mathbf{x}_0)$$
$$\mathbf{y} = \mathbf{y}_0 + \lambda(\mathbf{y}_1 - \mathbf{y}_0)$$

Cartesian Line Equation

We can remove the scalar λ using simultaneous equations:

$$\begin{aligned} \mathbf{x} &= \mathbf{x}_0 + \lambda(\mathbf{x}_1 - \mathbf{x}_0) \qquad && \times (\mathbf{y}_1 - \mathbf{y}_0) \\ -\mathbf{y} &= \mathbf{y}_0 + \lambda(\mathbf{y}_1 - \mathbf{y}_0) \qquad && \times (\mathbf{x}_1 - \mathbf{x}_0) \end{aligned}$$

Cartesian Line Equation

Giving:

$$\mathbf{x}(\mathbf{y}_1 - \mathbf{y}_0) - \mathbf{y}(\mathbf{x}_1 - \mathbf{x}_0) = \mathbf{x}_0(\mathbf{y}_1 - \mathbf{y}_0) - \mathbf{y}_0(\mathbf{x}_1 - \mathbf{x}_0)$$
$$a\mathbf{x} + b\mathbf{y} = -c$$

with:

$$\begin{aligned} a &= \mathbf{y}_1 - \mathbf{y}_0 \\ b &= \mathbf{x}_0 - \mathbf{x}_1 \\ c &= -b\mathbf{y}_0 - a\mathbf{x}_0 \end{aligned}$$

Cartesian Line Equation

The vectors $\mathbf{x}$ and $\mathbf{y}$ can be replaced with scalar values x and y, yielding:

$$ax + by + c = 0 \qquad \square$$

Cartesian Line Equation

The implicit equation has the form:

$$f(x, y) = C$$

where C is a constant.

Cartesian Line Equation

There is also an **explicit** algebraic equation of the form:

$$y = f(x)$$

Cartesian Line Equation

From:

$$
\begin{aligned}
ax + by + c = 0 & \\
\Rightarrow y =& -\frac{a}{b}x - \frac{c}{b} \\
\Rightarrow y =& mx + d
\end{aligned}
$$

where:

$$m = -\frac{a}{b}\ ,\ \ d = -\frac{c}{b}$$

Cartesian Line Equation

Although the explicit equation $y = mx + c$ may be familiar, for computer graphics it is inconvenient, since for vertical lines $m = \infty$.

Scan Conversion

Lines in mathematics are continuous and have *infinite* resolution.

A computer screen has finite resolution using discrete picture elements, or **pixels**.

Scan Conversion

For rendering, we will discretise the line equation using finite deltas.

$$y = mx + c \Rightarrow \delta y = \delta mx + c$$

Scan Conversion

– We always render line **segments**.
– Line segments have a defined start and end point.
– Hence, we can derive the slope and intercept of our line.

$$m = \frac{y_{end} - y_0}{x_{end} - x_0}$$
$$c = y_0 - mx_0$$

Scan Conversion

NB: We will ignore the intercept c for the following derivations.

- it should be added to the right-hand side of the equation for lines with $c \neq 0$

Digital Differential Analyser (DDA)

The digital differential analyser (DDA) is a scan-conversion line algorithm based on calculating either δy or δx

Digital Differential Analyser (DDA)

Calculating δx. Since the distance between points is measured in pixels; if we move pixel by pixel along the positive x axis, we have:

$$\delta x = x_{i+1} - x_i = 1$$

Digital Differential Analyser (DDA)

Calculating δy.

$$\delta y = m\delta x$$
$$y_{i+1} - y_i = m(x_{i+1} - x_i)$$

Where i is a grid position of a discreet point on the line, and $i + 1$ is an immediate neighbour on the grid.

Digital Differential Analyser (DDA)

Given $\delta x = x_{i+1} - x_i = 1$:

$$y_{i+1} = y_i + m$$

Specifically for:

$$0 \leq |m| \leq 1$$

Digital Differential Analyser (DDA)

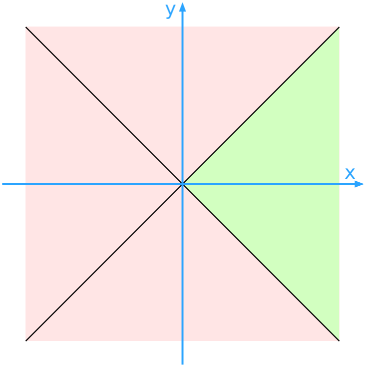

Figure 3: render octants

So far, our algorithm will draw lines when:

$$0 \leq |m| \leq 1 \text{ and } x \geq 0$$

Digital Differential Analyser (DDA)

```
#include <stdlib.h>
#include <math.h>

inline int round (const float a) {
    return int (a + 0.5);
    }

// Assume a function setPixel exists.
...
```

Digital Differential Analyser (DDA)

```
void naiveDDA (int x0, int y0, int xEnd, int yEnd){
    int x = x0;
    float y = float (y0);
    float m = float (yEnd - y0) / float (xEnd - x0);

   for (x = x0; x <= xEnd; x++) {
        setPixel (x, round (y));
        y += m;
} }
```

Digital Differential Analyser (DDA)

How do we draw in the other octants?

Digital Differential Analyser (DDA)

For lines with an absolute positive slope greater than 1.0, we reverse the roles of x and y.

That is, we sample at unit y intervals, $\delta y = 1$, and calculate consecutive x values as:

$$x_{i+1} = x_i + \frac{1}{m}$$

Digital Differential Analyser (DDA)

To cover the remaining octants, we *decrement* x and y.

Hence, for top, right, left and bottom octants, we have:

$$\begin{aligned}
|m| > 1\ , &\quad \delta y = 1, \quad x_{i+1} = x_i + \frac{1}{m} \\
0 \le |m| \le 1\ , &\quad \delta x = 1, \quad y_{i+1} = y_i + m \\
0 \le |m| \le 1\ , &\quad \delta x = -1, \quad y_{i+1} = y_i + m \\
|m| > 1\ , &\quad \delta y = -1, \quad x_{i+1} = x_i + \frac{1}{m}
\end{aligned}$$

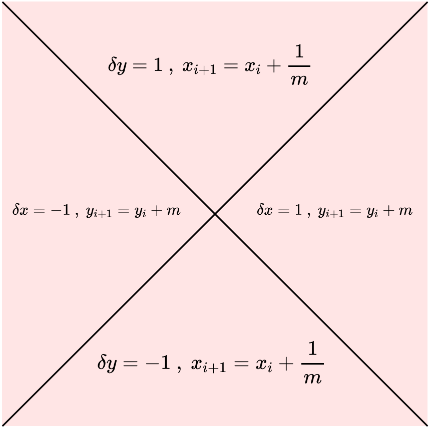

Figure 4: all octants

```
void lineDDA (int x0, int y0, int xEnd, int yEnd){
     int dx=xEnd-x0, dy=yEnd-y0, steps, k;
     float xIncrement, yIncrement, x = x0, y = y0;

    if (fabs (dx) > fabs (dy))
       steps = fabs (dx);
    else
       steps = fabs (dy);

    xIncrement = float (dx) / float (steps);
    yIncrement = float (dy) / float (steps);

    setPixel (round (x), round (y));
    for (k = 0; k < steps; k++) {
         x += xIncrement;
         y += yIncrement;
         setPixel (round (x), round (y));
} }
```

Digital Differential Analyser (DDA)

DDA has a few problems:

- fails to take advantage of the integral nature of pixels
- floating point variables to store the slope.
- costly division operations to calculate the slope.

Digital Differential Analyser (DDA)

The algorithm has a few problems:

- fails to take advantage of the integral nature of pixels
- floating point variables to store the slope.
- costly division operations to calculate the slope.

We will address these short comings in the next lecture.

Summary

- Theory and Concepts
- Scan Conversion
- Digital Differential Analyser (DDA)

Reading:

- Hearn & Baker, *Computer Graphics with OpenGL*, 4th Edition, Chapter 5