Computer Graphics and Visualisation

Lecture 18: Scan Conversion
Dr William Smith
wsmith@cs.york.ac.uk
No matter how complex the rendering process (2D/3D, orthographic/perspective, flat/smooth shading etc), ultimately all graphics comes down to:

\texttt{write\_pixel(x,y,colour)}

We are likely to have to perform this operation many times for every pixel (think back to the z-buffer)

If we wish to draw frames fast enough for smooth 3D graphics, we clearly need to be able to turn our graphical primitives into \texttt{write\_pixel} operations extremely quickly

It turns out even division operations need to be avoided

This process is called \textit{scan conversion} or \textit{rasterisation}
We will look in some detail at two scan conversion algorithms for lines:

- DDA Algorithm
- Bresenham’s Algorithm
- Plus issues related to line intensity

We will then look at methods for polygons:

- Filled polygons
- Edge tables
In order to scan convert lines, we must tackle the following problem:

**Input**

- We have a line defined on the plane of real numbers \((x, y)\).
- We have a discrete model of the \((x, y)\) plane consisting of a regular array of rectangles called pixels, which can be coloured.
Scan Conversion for Lines

In order to scan convert lines, we must tackle the following problem:

**Input**

- We have a line defined on the plane of real numbers \((x, y)\).
- We have a discrete model of the \((x, y)\) plane consisting of a regular array of rectangles called pixels, which can be coloured.

**What we wish to do**

Map the line onto the pixel array, while satisfying or optimising the following constraints:

- Maintain constant brightness
- Differing pen styles or thicknesses may be required
- Shape of endpoints if line thicker than one pixel
- Minimise ‘jaggies’
The pixel space is a rectangle on the x y plane, bounded by:

$$0 \leq x \leq x_{\text{max}} \text{ and } 0 \leq y \leq y_{\text{max}}$$

Each axis is divided into an integer number of pixels, $N_x$ and $N_y$. The pixels there for have width and height:

$$W = \frac{x_{\text{max}}}{N_x} \quad H = \frac{y_{\text{max}}}{N_y}$$

Pixels are referred to using integer coordinates, that either refer to the location of their lower left hand corners or their centres.

Knowing the $W$ and $H$ values allows the pixel aspect ratio to be defined.

Assume that $W$ and $H$ are equal so pixels are square.
Pixel Space

The diagram shows a grid with axes labeled x and y. The point (1, 2) is marked in the grid. The region labeled W is shaded.
Both of the algorithms we will look at make the assumption that the gradient of the line satisfies $0 \leq m \leq 1$

However, through symmetry we can handle all other cases

- $1 \leq m \leq \infty$: Switch x’s and y’s
- $0 \leq m \leq 1$
- $m < 0$: Switch sign of y
A line segment is to be drawn from \((x_1, y_1)\) to \((x_2, y_2)\)

Generating a line segment is equivalent to solving a simple differential equation:

\[
m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}
\]

Starting from \((x_1, y_1)\), for any change in \(x\) of size \(\Delta x\), the corresponding change in \(y\) is:

\[
\Delta y = m \Delta x
\]
The DDA algorithm uses a function \( \text{round}(x) \) to convert floating point, real valued coordinates to integer pixel coordinates:

\[
\begin{align*}
\text{line}_\text{start} &= \text{round}(x_1); \\
\text{line}_\text{end} &= \text{round}(x_2); \\
\text{colour}_\text{pixel}\left(\text{round}(x_1), \text{round}(y_1)\right); \\
\text{for}(i = \text{line}_\text{start} + 1; i \leq \text{line}_\text{end}; i++){
    & y_1 = y_1 + m; \quad \text{Floating point addition} \\
    & \text{colour}_\text{pixel}(i, \text{round}(y_1)); \\
}\end{align*}
\]

There is no scaling involved in moving from projection to viewport coordinates, so we can use these directly.
If we use this algorithm when \( m > 1 \), the line is drawn incorrectly. The algorithm says: “for each \( x \), find best \( y \)” but for large \( m \), gap between pixels is large.

Solution, when \( m > 1 \), switch roles of \( x \) and \( y \). Say “for each \( y \), find best \( x \)”.

Using this modified approach we get…
Correctly drawn lines using modified DDA algorithm that checks gradient and exploits symmetry as appropriate.
Bresenham’s Algorithm

The DDA algorithm requires a floating point addition and rounding operation for every iteration.

Bresenham’s algorithm operates only on integers requiring only that the start and end points of a line segment are rounded.

From the current pixel \((i,j)\), the value of \(y\) at \(x = i+1\) must lie between \(j\) and \(j+1\) (since we restrict \(0 \leq m \leq 1\)).

Hence, the next pixel we colour will be either E, \((i+1,j)\) or NE, \((i+1,j+1)\).
Bresenham’s Algorithm

The key idea of Bresenham’s algorithm is to reduce line drawing to the decision problem of choosing between N and NE for the next pixel.

It turns out that we can design a decision variable on which to base this choice which we can compute without division.

Rewrite equation of a straight line:

\[ y = mx + c \]

\[ mx - y + c = 0 \]

\[ \frac{\Delta y}{\Delta x} x - y + c = 0 \]

Multiplying through by \( \Delta x \) gives us our decision variable:

\[ F(x, y) = x\Delta y - y\Delta x + c\Delta x = x\Delta y - y\Delta x + B = 0 \]
Bresenham’s Algorithm

The useful property of this function is that:

• $F(x,y) < 0$ for points above the line

• $F(x,y) > 0$ for points below the line

We can now use the decision variable to derive our line drawing algorithm

This involves some rational numbers in the derivation, but we cancel for these at the end
Consider an arbitrary pixel \((x_p, y_p)\) that is on the line segment

We need to design a test to decide which of the next 2 pixels to colour

The point to test is \((x_p+1, y_p+1/2)\):

This point will be on the boundary between pixels E and NE and at the horizontal midpoint

So:

If \(F(x_p+1, y_p+1/2) \leq 0\) then next pixel is E

Otherwise next pixel is NE
Bresenham’s Algorithm

The decision for our next step is dependent on the decision we have just made:

If E was chosen then the next test will be \( F(x_p+2, y_p+1/2) \)

If NE was chosen the next test will be \( F(x_p+2, y_p+3/2) \)

We can obtain an iterative form of this decision variable which accounts for these two cases

Let \( d_n = F(x_p+1, y_p+1/2) \)

If we substitute \((x+2, y+1/2)\) and \((x+2, y+3/2)\) back into the definition of \( F \) we end up with:

If E was chosen, the next test will be \( d_{n+1} = d_n + \Delta y \)

If NE was chosen, the next test will be \( d_{n+1} = d_n + (\Delta y - \Delta x) \)
Bresenham’s Algorithm

To initialise the process we need a formula to compute $d_1$:

$$
\begin{align*}
    d_1 &= F(x_0 + 1, y_0 + 1/2) = \Delta y(x_0 + 1) - \Delta x(y_0 + 1/2) + B \\
    d_1 &= (x_0 \Delta y - y_0 \Delta x + B) + \Delta y - \Delta x/2 \\
    d_1 &= F(x_0, y_0) + \Delta y - \Delta x/2 \\
    d_1 &= d_0 + \Delta y - \Delta x/2
\end{align*}
$$

Since we know that the starting point is on the line, $d_0 = 0$ and hence:

$$
    d_1 = \Delta y - \Delta x/2
$$

This still has a factor of 1/2 in it and we wanted only integers

Solution: simply multiply everything by 2
void bresenham(int x0, int y0, int x1, int y1) {
  int dx = x1 - x0;
  int dy = y1 - y0;
  int d = (2*dy) - dx;
  int incr_E = 2*dy;
  int incr_NE = 2*(dy - dx);
  int x = x0, y = y0;
  colour_pixel(x0, y0);
  while (x < x1) {
    if (d <= 0) {
      d += incr_E;
      x++;
    } else {
      d += incr_NE;
      x++;
      y++;
    }
    colour_pixel(x, y);
  }
}

Set up constants
Initialise d
Repeatedly apply decision test based on outcome of decision at previous iteration and colour pixel accordingly
Bresenham’s Algorithm: Worked Example

We would like to draw a line from (1,1) to (4,2):

```
1 2 3 4
1 X 2 X 3
2 3 4 X
```

Lecture 18
We would like to draw a line from (1,1) to (4,2):

We can begin by colouring pixel (1,1) and computing some values that we will need:

\[ \Delta x = 4 - 1 = 3 \quad 2\Delta y = 2 \]
\[ \Delta y = 2 - 1 = 1 \quad 2(\Delta y - \Delta x) = 2(1 - 3) = -4 \]
Bresenham’s Algorithm: Worked Example

We initialise \( d_0 = 0 \) and hence:

\[
d_1 = 2\Delta y - \Delta x = 2 - 3 = -1
\]

When \( d \leq 0 \) we move East, so we can colour the next pixel:

\[
\Delta x = 4 - 1 = 3 \quad 2\Delta y = 2
\]

\[
\Delta y = 2 - 1 = 1 \quad 2(\Delta y - \Delta x) = 2*(1 - 3) = -4
\]
Now we simply need to iteratively apply the rule:

\[ d_{n+1} = d_n + \begin{cases} 
2\Delta y & \text{if } d_n \leq 0 \\
2(\Delta y - \Delta x) & \text{otherwise}
\end{cases} \]

\( \Delta x = 4 - 1 = 3 \quad 2\Delta y = 2 \)
\( \Delta y = 2 - 1 = 1 \quad 2(\Delta y - \Delta x) = 2(1 - 3) = -4 \)
Bresenham’s Algorithm: Worked Example

Again, apply the iterative rule:

\[ d_{n+1} = d_n + \begin{cases} 
2\Delta y & \text{if } d_n \leq 0 \\
2(\Delta y - \Delta x) & \text{otherwise}
\end{cases} \]

\[ \Delta x = 4-1 = 3 \quad 2\Delta y = 2 \]
\[ \Delta y = 2-1 = 1 \quad 2(\Delta y-\Delta x) = 2*(1-3) = -4 \]

\[ d_2 \text{ was } >0, \text{ so we use:} \]
\[ d_3 = d_2 + 2(\Delta y - \Delta x) \]
\[ = 1 - 4 = -3 \]

This time, \( d \) is less than 0, so we finish by moving East and colouring the last pixel.

\[ \Delta x = 4-1 = 3 \quad 2\Delta y = 2 \]
\[ \Delta y = 2-1 = 1 \quad 2(\Delta y-\Delta x) = 2*(1-3) = -4 \]
Simple line drawing algorithms such as DDA and Bresenham’s algorithm do not give lines that appear with the correct intensity:

- Apparent line intensity is a function of the gradient
- Lines with different slopes have a different number of pixels per unit length
- To draw two such lines with the same intensity must make pixel intensity a function of the gradient (or use antialiasing)
Line Intensity

Pixels per unit line length is a function of gradient (densest for horizontal and vertical lines, sparsest for 45 degree lines). To make lines appear the same brightness, must vary intensity according to gradient.
The best solution to the problem of maintaining consistent line intensity is to use antialiasing:

If lines are assigned a thickness, we can calculate what proportion of each pixel is covered by the line and colour accordingly.

Pixels only partially covered should be blended with whatever is behind.
If a line segment has been clipped, we need to be careful how the clipped end is treated:

- At the clipping boundary, one coordinate will be integer, one real
- Pixel at boundary will be correctly drawn
- Subsequent pixels may not be as gradient of clipped line is different

The solution is:
- Draw edge pixel
- Initialise \( F(\ldots) \) for next column over
- Use original (not clipped gradient)
Filled Polygons

The basic idea:

for each scan line
calculate list of edge intersections
sort list in increasing x
for each pair (p1, p2) in list
fill span from p1 to p2

Line through a polygon will cross an even number of edges
Edge Tables

This is the same scan line algorithm used in for visible surface determination.

There is a global list of edges maintained each with whatever interpolation information is required.

At any point on a scan within a polygon we interpolate the colour intensity from the 2 current edges.

Active edge table:

```
AB → EF → ED → CD
```

Odd number of crossing = inside, even = outside
We need to handle some special cases:

**Singularities**

- Count as 2 edge crossings
- Count as 1 edge crossing

**Shared vertices** – draw it as belonging to lowest/leftmost vertex

**Horizontal edges** – don’t count vertices at all

**Integer edge intersections** – consistently round up/down
Conclusions

You should know:

- All graphics eventually comes down to plotting pixels
- DDA – conceptually simple line drawing algorithm
- Bresenham – highly efficient, avoids floating point operations
- Polygon rasterisation

Next time:

- 2 lectures on texturing surfaces: realism from images

Recommended reading for this lecture:

- Angel Chapter 7.8-7.10
Take Home Lessons: Scan Conversion  
(ref: Angel chap. 7.8-7.10)

• Displaying graphical content of any kind, whether 2D/3D, orthographic/perspective, flat/smooth shading etc, eventually it comes down to drawing individual pixels with individual colours to the screen. This is the fundamental operation in graphics.

• Basic problem: primitives defined over plane of real numbers (x,y). Pixels are discrete elements of a grid defined over the same plane. For a given primitive, which pixels should be coloured?

• Line drawing: 2 algorithms

• **DDA Algorithm**: Conceptually simple, but requires floating point arithmetic at every iteration

• **Bresenham’s Algorithm**: Draws the same lines, but avoids all floating point operation. Based on evaluating a decision variable at every iteration which

• Points to note for both algorithms: we didn’t consider aliasing, so line brightness will depend on orientation and there will be ‘jaggies’. For basic algorithm, gradient must be between 0 and 1, but easy to cover all possibilities using symmetry.

• Rasterizing polygons is in some ways easier. We just need to keep track of when we enter and exit a polygon as we move along a scanline, either colouring pixels or not. Need to be careful with abutting polygons etc.