Line Drawing

Richard (Hao) Zhang

Introduction to Computer Graphics
CMPT 361 – Lecture 3
Today

- **The graphics pipeline:** a first look
- Line representations: explicit, implicit, parametric
- Naïve algorithm and DDA
- **Bresenham’s algorithm**
- OpenGL diamond exit rule
- Circle drawing

Readings: 1.7, 8.8 - 8.9
The graphics pipeline

- From the modeling stage to image formation
- Pipelined approach helps increase system throughput
  - Throughput: rate at which data flows through the system
  - Data processing in different blocks can be done in parallel
  - Effective when same sequence of operations are performed on large quantity of data (vertices and pixels) – true in graphics
Vertex processor

- **Per-vertex** operations (vertices define objects/primitives)

- Two main functionalities:
  - Coordinate transformations
  - Color computation at each vertex (illumination and shading)

- Objects/geometry transformed from their own coordinate systems into world space — **modeling transformation**

- Transform from world space into camera coordinate system — **viewing and projection transformation**
Clipping and primitive assembly

- Model a finite field of vision
- Remove objects outside a finite clipping volume
- Done on primitives (e.g., polygons), not vertices
- Output: primitives whose projections appear in image
Rasterizer

- Convert each primitive into a set of **fragments**

- Fragment: **stores per pixel info for its primitive**
  - Raster/pixel location
  - Depth, e.g., to determine whether this fragment “survives”
  - Texture coordinates
  - Alpha value (for blending)
Fragment processing

- Performs per-fragment operations

- Main function is to **compute the color at a pixel**, using information stored in the fragments

- A (programmable) **fragment shader** is a program that performs the per-fragment operations

- Similar to a **vertex shader** (per-vertex operations)

- Shader programs typically have limited instruction set
Where are we at now?

Drawing or rasterization of primitives first. Consider lines and polygons.
Line drawing in OpenGL

```c
void myDraw()
{
    glDrawArrays(GL_LINES, 0, numOfPoints);

    FYI: Old OpenGL
    glBegin(GL_LINES);
    glVertex2f(-0.5, -0.5);
    glVertex2f(-0.5, 0.5);
    glVertex2f(0.5, 0.5);
    glVertex2f(0.5, -0.5);
    glEnd();
}
```

- `GL_LINE_STRIP`
- `GL_LINE_LOOP`
- There is no `GL_CIRCLE`
- Draws two line segments
Assumptions

- Transformation, clipping, projection already done
- Primitives to rasterize are actually on the screen
- Work with 2D screen coordinates with square pixels

(n, m)

(0, 0)
Representation of line segment

- Explicit: $y = f(x)$
  
  \[ y = \frac{dy}{dx} (x - x_0) + y_0 \]

- Parametric: $x = f(t), y = g(t)$
  
  \[
  \begin{align*}
  x(t) &= x_0 + t(x_1 - x_0) \\
  y(t) &= y_0 + t(y_1 - y_0), \quad t \in [0, 1] \\
  P(t) &= P_0 + t(P_1 - P_0), \quad \text{or} \\
  P(t) &= (1 - t)P_0 + tP_1
  \end{align*}
  \]
Representation of line segment

- Implicit: $F(x, y) = 0$

$$F(x, y) = (x - x_0)dy - (y - y_0)dx$$

if

$F(x, y) = 0$ then $(x, y)$ is on the line

$F(x, y) > 0$ then $(x, y)$ is below the line

$F(x, y) < 0$ then $(x, y)$ is above the line

Assumption: $dx > 0$
Line drawing problem

- Given an explicit line equation
  \[ y = mx + n, \quad m = \frac{dy}{dx} \] is the slope
  and end points \((x_0, y_0)\) and \((x_1, y_1)\), which pixels to set?
- This is an approximation problem using grid points
- Why not use the parametric form?

In OpenGL

Integer coordinates

Pixel center

Pixels
Desirable properties

The line rasterization algorithm should …

- Passes through both end points (pixels)
- Returns a straight line
- Independent of order of endpoints
- Gives uniform brightness
- Brightness independent of slope
- Efficient
Naive algorithm

- Basic idea: Increase \( x \) one pixel at a time and update \( y \)
- Brute force
  - for each pixel \( x \) compute
    
    \[
    y = \text{round}(mx + n)
    \]
  - (costly) floating point multiplication per pixel
- Try to avoid floating-point multiplications as they are much more expensive than additions
The DDA algorithm

- **DDA** – *digital differential analyzer*

Line drawing resembling solution of simple differential equation for a line

```plaintext
endpoints: from \((x_0, y_0)\) to \((x_1, y_1)\)

\[ m = \frac{(y_1 - y_0)}{(x_1 - x_0)}: \text{a floating-point number} \]

write_pixel\((x_0, y_0)\);

\(y = y_0;\)

for \((x = x_0 + 1; x \leq x_1; x++)\) {
    \(y += m;\) /* Note that \(y = mx + n\)*/
    write_pixel\((x, \text{round}(y))\);
}
```
Two drawbacks

- Floating-point addition still needed
- Large slope leads to discontinuity – fix using symmetry

Do $0 \leq m \leq 1$ and the rest use symmetry
Bresenham’s Algorithm

- Uses only **integer arithmetic** & no multiplications
- A standard line-rasterization algorithm today
- Also known as the **midpoint algorithm**
- Idea: provide best-fit approximation to a true line by minimizing the error, i.e., a **vertical distance**, to true line
- OpenGL uses different implementation: **diamond exit rule**
Bresenham: basic idea

- Assume slope: \(0 \leq m \leq 1\)
- In our presentation, **pixel centers are integers**
- Let us suppose that pixel \((x_p, y_p)\) is just drawn
- Next pixel is

\[
\text{E: } (x_p + 1, y_p) \text{ or NE: } (x_p + 1, y_p + 1)
\]

- **Rule:** If midpoint is above the line, then E, otherwise NE – error or vertical distance will be \(\leq 1/2\)
A few minor points

- What if, e.g., starting point is (1, 1) and endpoint (0, 0)?
  
  if \( x_0 > x_1 \) then switch start and end points

- What if slope > 1, etc.?  
  
  Use symmetry

- What if midpoint lies on the true line?
  
  Make an arbitrary choice: E or NE
Efficiency

- Determine whether midpoint is above/below line efficiently
- If $dx > 0$, i.e., $x_1 > x_0$ (an assumption), then

\[
F(x, y) = (x - x_0)dy - (y - y_0)dx
\]

if

- $F(x, y) = 0$ then $(x, y)$ is on the line
- $F(x, y) > 0$ then $(x, y)$ is below the line
- $F(x, y) < 0$ then $(x, y)$ is above the line
Incremental updates

Let \( d = 2F(x, y) \) – why multiply by 2?

Initially, \( d = 2F(x_0+1, y_0+1/2) = 2dy – dx \)

Note that \((x_0+1, y_0+1/2)\) is the first midpoint to test …

If \( d_{old} = 2F(x, y) \) is computed for a midpoint \((x, y)\) at a step

If E were chosen at this step,

\[
d_{new} = 2F(x+1, y) = 2dy + 2F(x, y) = d_{old} + 2dy
\]

If NE were chosen at this step,

\[
d_{new} = 2F(x+1, y+1) = 2(dy – dx) + 2F(x, y)
\]

\[
= d_{old} + 2(dy – dx)
\]

Keep in mind: \( F(x, y) = (x - x_0)dy - (y - y_0)dx \)
Bresenham’s algorithm:

The following algorithm produces the desired results for lines having $x_0$ less than $x_1$ and a slope between 0 and 1.

```c
drawline(x0, y0, x1, y1) {
    int dx, dy, d, incE, incNE, x, y;
    dx = x1 - x0;
    dy = y1 - y0;
    d = 2 * dy - dx; // initial d = 2F(x_0 + 1, y_0 + 1/2)
    incE = 2 * dy;   // if go E, only x += 1
    incNE = 2 * (dy - dx); // if go NE, x += 1 and y += 1
    y = y0;
    for (x = x0; x ≤ x1; x++) {
        write_pixel(x, y);
        if (d > 0) { d = d + incNE; y = y + 1; }
        else { d = d + incE; }
    }
}
```
OpenGL diamond exit rule

- Given a line from $p$ to $q$, all pixels whose diamonds are exited by the line are set.
- If line only enters the diamond of the last pixel, it is not set.
- When drawing a polygon, pixels at vertices of an edge are only drawn once.
- Except for end pixels, will it lead to same results as Bresenham?
Circle drawing

- Use the explicit circle equation
- Divide into 8 octants and work with the 2nd
- The rest is by symmetry
Representation of a circle

- **Explicit:**
  \[ y = \pm \sqrt{r^2 - x^2}, \quad |x| \leq r \]

- **Parametric:**
  \[ x(\theta) = r \cos(\theta) \]
  \[ y(\theta) = r \sin(\theta), \quad \theta \in [0, 2\pi] \]

- **Implicit:**
  \[ x^2 + y^2 = r^2 \]
  \[ F(x, y) = x^2 + y^2 - r^2 \]
  if
  \[ F(x, y) = 0 \quad \text{then} \quad (x, y) \text{ is on the circle} \]
  \[ F(x, y) > 0 \quad \text{then} \quad (x, y) \text{ is outside the circle} \]
  \[ F(x, y) < 0 \quad \text{then} \quad (x, y) \text{ is inside the circle} \]
Midpoint circle

- Choose between E and SE pixel at each stage

- Possible: integer-only with +, -, and bit shifts only