CS 430
Computer Graphics

Line Drawing
Week 1, Lecture 2
David Breen, William Regli and Maxim Peysakhov
Department of Computer Science
Drexel University

Outline
• Math refresher
• Line drawing
• Digital differential analyzer
• Bresenham’s algorithm
• PBM file format

Geometric Preliminaries
• Affine Geometry
  – Scalars + Points + Vectors and their ops
• Euclidian Geometry
  – Affine Geometry lacks angles, distance
  – New op: Inner/Dot product, which gives
    • Length, distance, normalization
    • Angle, Orthogonality, Orthogonal projection
• Projective Geometry

Mathematical Preliminaries
• Vector: an n-tuple of real numbers
• Vector Operations
  – Vector addition: \( \mathbf{u} + \mathbf{v} = \mathbf{w} \)
    • Commutative, associative, identity element (0)
  – Scalar multiplication: \( c \mathbf{v} \)
• Note: Vectors and Points are different
  – Can not add points
  – Can find the vector between two points

Affine Geometry
• Affine Operations:
  • Affine Combinations:
    \[ a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2 + \ldots + a_n \mathbf{v}_n \]
    where \( \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n \) are vectors and \( \sum a_i = 1 \)
    Example: \( \mathbf{R} = (1 - \alpha) \mathbf{P} + \alpha \mathbf{G} \)

Linear Combinations & Dot Products
• A linear combination of the vectors
  \( \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n \)
  is any vector of the form
  \[ \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2 + \ldots + \alpha_n \mathbf{v}_n \]
  where \( \alpha \) is a real number (i.e. a scalar)
• Dot Product:
  \[ \mathbf{u} \cdot \mathbf{v} = \sum_{k=1}^{n} u_k v_k \]
  a real value \( u_1 v_1 + u_2 v_2 + \ldots + u_n v_n \) written as \( \mathbf{u} \cdot\mathbf{v} \)
Fun with Dot Products

- **Eucldian Distance** from \((x,y)\) to \((0,0)\)
  \[
  \sqrt{x^2 + y^2}
  \]
  in general:
  \[
  \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}
  \]
  which is just:
  \[
  \sqrt{\vec{x} \cdot \vec{x}}
  \]
- This is also the length of vector \(\vec{v}\):
  \[
  ||\vec{v}||
  \]
  or
  \[
  |\vec{v}|
  \]
- **Normalization** of a vector:
  \[
  \frac{\vec{v}}{||\vec{v}||}
  \]
- **Orthogonal vectors**:
  \[
  \vec{u} \cdot \vec{v} = 0
  \]

Projections & Angles

- **Angle between vectors**, \(\vec{u} \cdot \vec{v} = ||\vec{u}|| ||\vec{v}|| \cos(\theta)\)
  \[
  \theta = \arccos\left(\frac{\vec{u} \cdot \vec{v}}{||\vec{u}|| ||\vec{v}||}\right)
  \]
- **Projection of vectors**
  \[
  \vec{u}_1 = \frac{(\vec{u} \cdot \vec{v})}{(\vec{v} \cdot \vec{v})} \vec{v}
  \]
  \[
  \vec{u}_2 = \vec{u} - \vec{u}_1
  \]

Matrices and Matrix Operators

- A n-dimensional vector:
  \[
  \begin{bmatrix}
  x_1 \\
  \vdots \\
  x_n
  \end{bmatrix}
  \]
- **Matrix Operations**:
  - Addition/Subtraction
  - Identity
  - Multiplication
  - **Scalar**
  - **Matrix Multiplication**
- **Implementation issue**:
  Where does the index start?
  (0 or 1, it’s up to you…)
  \[
  1A = A
  \]
  \[
  c(A + B) = cA + cB
  \]
  \[
  (c + d)A = cA + dA
  \]

Matrix Multiplication

- \([C] = [A][B]\)
- Sum over rows & columns
- Recall: matrix multiplication is **not** commutative
- **Identity Matrix**:
  \[
  \begin{bmatrix}
  1 & 0 & \cdots & 0 \\
  0 & 1 & \cdots & 0 \\
  \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & \cdots & 1
  \end{bmatrix}
  \]
  where \(0\)s everywhere else

Matrix Determinants

- A single real number
- Computed recursively
  \[
  \det(A) = \sum_{i=1}^{n} A_{ij} (-1)^{i+j}M_{ij}
  \]
- Example:
  \[
  \det\begin{bmatrix}
  a & c \\
  b & d
  \end{bmatrix} = ad - bc
  \]
- Uses:
  - Find vector ortho to two other vectors
  - Determine the plane of a polygon

Cross Product

- Given two non-parallel vectors, A and B
- A \times B calculates third vector C that is orthogonal to A and B
- \[
  \begin{align*}
  A \times B &= (a_x b_2 - a_2 b_x, a_2 b_y - a_y b_2, a_y b_x - a_x b_y) \\
  \end{align*}
  \]
  \[
  A \times B = \begin{bmatrix}
  \hat{x} & \hat{y} & \hat{z} \\
  a_x & a_y & a_z \\
  b_x & b_y & b_z
  \end{bmatrix}
  \]
Matrix Transpose & Inverse

- **Matrix Transpose**: Swap rows and cols:
  \[ A = \begin{bmatrix} 2 & 8 \\ 8 & 2 \end{bmatrix}, \quad A^T = \begin{bmatrix} 2 & 8 \\ 8 & 2 \end{bmatrix} \]

  - Facts about the transpose:
    - \((A^T)^T = A\)
    - \((A + B)^T = A^T + B^T\)
    - \((cA)^T = c(A^T)\)
    - \((AB)^T = B^T A^T\)

- **Matrix Inverse**: Given \(A\), find \(B\) such that
  \[ AB = BA = I \]
  \[ B = A^{-1} \]
  (only defined for square matrices)

---

Line Drawing

---

Scan-Conversation Algorithms

- **Scan-Conversion**: Computing pixel coordinates for ideal line on 2D raster grid
  - Pixels best visualized as circles/dots
    - Why? Monitor hardware

---

Drawing a Line

- \(y = mx + B\)
- \(m = \Delta y / \Delta x\)
- Start at leftmost \(x\) and increment by 1
  \[ \Delta x = 1 \]
- \(y_i = \text{Round}(mx_i + B)\)
- This is expensive and inefficient
  - Since \(\Delta x = 1\), \(y_{i+1} = y_i + \Delta y = y_i + m\)
  - No more multiplication!
- This leads to an incremental algorithm

---

Digital Differential Analyzer (DDA)

- If |slope| is less then 1
  - \(\Delta x = 1\)
  - \(\Delta y = f\)
- Check for vertical line
  - \(m = \infty\)
- Compute corresponding \(\Delta y (\Delta x) = m (\Delta x)\)
- \(x_{i+1} = x_i + \Delta x\)
- \(y_{i+1} = y_i + \Delta y\)
- Round (x,y) for pixel location
- Issue: Would like to avoid floating point operations

---

Generalizing DDA

- If |slope| is less than or equal to 1
  - Ending point should be right of starting point
- If |slope| is greater than 1
  - Ending point should be above starting point
- Keep \(x\) and \(y\) as floating point values
- Vertical line is a special case
  \[ \Delta x = 0 \]
Bresenham’s Algorithm

• 1965 @ IBM
• Basic Idea:
  – Only integer arithmetic
  – Incremental

• Consider the implicit equation for a line:
  \[ f(x, y) = ax + by + c = 0 \]

The Algorithm

void bresenham(int Point p1, int Point p2) {
  int dx = p2.x - p1.x, dy = p2.y - p1.y;
  if (dy > 0) dy = -dy; // initial decision value
  if (dx > dy) { // steep enough
    dx = 2*dy - dx; // switch to x is dominant
    dy = dx;
    dx = 2*dy;
  }
  int M = dy, E, NE;
  int delM = M + 1; int delE = delM + 1;
  E = p1.x; NE = p1.x + 1;
  for (; E <= NE; E++) {
    if (E = p1.x) "line width and height"
    dy = dy - dx;
    if (dy = 0) D = -2*dy; // below midpoint - go to E
  } else {
    D = 2*(dy - delE); // above midpoint - go to NE
  }
}

Assumptions: \( \frac{q_x}{q_y} \leq 1 \)
0 ≤ slope ≤ 1

Bresenham’s Algorithm

Given:

implicit line equation: \( f(x, y) = ax + by + c = 0 \)

Let:

- \( d_x = r_x - q_x,\) \( d_y = r_y - q_y \)
  where \( r \) and \( q \) are points on the line and
- \( d_x, d_y \) are positive
  \[ a = d_y,\ b = -d_x,\ c = -(q_x r_y - r_x q_y) \]

Then:

Observe that all of these are integers
and: \( f(x, y) < 0 \) for points above the line
\( f(x, y) > 0 \) for points below the line

Now.....

Bresenham’s Algorithm

• Suppose we just finished \((p_x, p_y)\)
  – (assume 0 ≤ slope ≤ 1)
  other cases symmetric

• Which pixel next?
  – E or NE

East \( (E = (p_x + 1, p_y)) \)
NorthEast \( (NE = (p_x + 1, p_y + 1)) \)

Bresenham’s Algorithm

Assume:

• \( Q = \) exact \( y \) value at \( x = p_x + 1 \)
• \( y \) midway between \( E \) and \( NE: M = p_x + 1/2 \)

Observe:

If \( Q < M \), then pick \( E \)
Else pick \( NE \)
If \( Q = M \), it doesn’t matter

Bresenham’s Algorithm

Create “modified” implicit function (2x)
\[ f(x, y) = 2ax + 2by + 2c = 0 \]
Create a decision variable \( D \) to select,
where \( D \) is the value of \( f \) at the midpoint:
\[ D = f(p_x + 1, p_y + (1/2)) \]
\[ = 2ap_x + 2b(p_y + 1) + (2a + b + 2c) \]
Bresenham’s Algorithm

- If $D > 0$ then $M$ is below the line $f(x, y)$;
  - $NE$ is the closest pixel
- If $D \leq 0$ then $M$ is above the line $f(x, y)$;
  - $E$ is the closest pixel

Note: because we multiplied by $2x$, $D$ is now an integer—which is very good news

How do we make this incremental??

Case I: When $E$ is next

- What increment for computing a new $D$?
- Next midpoint is: $(p_x + 2, p_y + (1/2))$
  
  \[
  D_{\text{new}} = f(p_x + 2, p_y + (1/2))
  = 2a(p_x + 2) + 2b \left( p_y + \frac{1}{2} \right) + 2c
  = 2ap_x + 2bp_y + (4a + b + 2c)
  = 2ap_x + 2bp_y + (2a + b + 2c) + 2a
  = D + 2a = D + 2d_y
  \]
- Hence, increment by: $2d_y$

Case II: When $NE$ is next

- What increment for computing a new $D$?
- Next midpoint is: $(p_x + 2, p_y + (1/2))$
  
  \[
  D_{\text{new}} = f(p_x + 2, p_y + (1/2))
  = 2a(p_x + 2) + 2b \left( p_y + \frac{3}{2} \right) + 2c
  = 2ap_x + 2bp_y + (4a + 3b + 2c)
  = 2ap_x + 2bp_y + (2a + b + 2c) + (2a + 2b)
  = D + 2(a + b) = D + 2(d_x - d_y)
  \]
- Hence, increment by: $2(d_y - d_x)$

How to get an initial value for $D$?

- Suppose we start at: $(q_x, q_y)$
- Initial midpoint is: $(q_x + 1, q_y + 1/2)$

Then:

\[
D_{\text{init}} = f(q_x + 1, q_y + 1/2)
= 2a(q_x + 1) + 2b \left( q_y + \frac{1}{2} \right) + 2c
= (2aq_x + 2bq_y + 2c) + (2a + b)
= 0 + 2a + b
= 2d_y - d_x
\]

The Algorithm

```c
void bresenham(intPoint q, intPoint r) {
    int dx = r.x - q.x; // line width and height
    dy = r.y - q.y;
    D = 2*dy - dx; // initial decision value
    y = q.y; // start at (q.x,q.y)
    for (x = q.x; x <= r.x; x++) {
        writePixel(x, y); // below midpoint - go to E
        if (D <= 0) D += 2*dy; // above midpoint - go to NE
        else D += 2*(dy - dx); y++; // above midpoint - go to NE
    }
}
```

Assumptions: $q_x < r_x$
$0 \leq \text{slope} \leq 1$
Pre-computed: $2d_y, 2(d_y - d_x)$
**Generalize Algorithm**

- If \( q_x > r_x \), swap points
- If slope > 1, always increment \( y \), conditionally increment \( x \)
- If \(-1 \leq \text{slope} < 0\), always increment \( x \), conditionally decrement \( y \)
- If \( \text{slope} < -1 \), always decrement \( y \), conditionally increment \( x \)
- Rework \( D \) increments

**Bresenham’s Algorithm: Example**

\[
F(x,y) = 2D_x - D_y = 0
\]

\[
F(x,y) = 2D_x - 2D_y = 0
\]

(7,5)

**Bresenham’s Algorithm: Example**

(7,5)
Bresenham’s Algorithm: Example

Some issues with Bresenham’s Algorithms

- Pixel ‘density’ varies based on slope
  - straight lines look darker, more pixels per unit length
- Endpoint order
- Line from P1 to P2 should match P2 to P1
- Always choose E when hitting M, regardless of direction
Plain PBM Image File

- There is exactly one image in a file
- File begins with "magic number" "P1"
- Next line specifies pixel resolution
- Each pixel is represented by a byte containing ASCII ‘1’ (black) or ‘0’ (white)
- All fields/values separated by whitespace characters
- No line longer than 70 characters?

Plain PBM Image Example

```
P1
# foop.pbm
24 7
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0
0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 1 1 1 0
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

There is a newline character at the end of each of these lines.

Programming assignment 1

- Input PostScript-like file
- Output B/W PBM
- Primary I/O formats for the course
- Create data structure to hold points and lines in memory (the world model)
- Implement 2D translation, rotation and scaling of the world model
- Implement line drawing and clipping
- Due October 4th
- Get started now!

Questions?

Go to Assignment 1