Abstract:

**Geometric Preliminaries**

- **Affine Geometry**
  - Scalars + Points + Vectors and their ops
- **Euclidean 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: $u + v = w$
    - Commutative, associative, identity element (0)
  - Scalar multiplication: $cv$

Note: Vectors and Points are different

- Can not add points
- Can find the vector between two points

**Outline**

- Math refresher
- Line drawing
- Digital differential analyzer
- Bresenham's algorithm
- XPM file format

**Affine Geometry**

- **Affine Operations**
  - Affine Combinations
- **Dot Products**
  - A linear combination of the vectors $v_1, v_2, ..., v_n$
  - is any vector of the form $\alpha_1 v_1 + \alpha_2 v_2 + ... + \alpha_n v_n$
  - where $\alpha_i$ is a real number (i.e. a scalar)

**Linear Combinations & Dot Products**

- A linear combination of the vectors $v_1, v_2, ..., v_n$
  - is any vector of the form $\alpha_1 v_1 + \alpha_2 v_2 + ... + \alpha_n v_n$
  - where $\alpha_i$ is a real number (i.e. a scalar)

- Dot Product:
  - $u \cdot v = \sum_{k=1}^{n} u_k v_k$
  - a real value $u_1 \alpha_1 + u_2 \alpha_2 + ... + u_n \alpha_n$ written as $u \cdot v$
Fun with Dot Products

• **Euclidean 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:
\[
\sqrt{\text{length}^2}
\]

This is also the length of vector \(v\):
\[
|v|
\]
or
\[
|v|
\]

• **Normalization of a vector:**
\[
\hat{v} = \frac{v}{|v|}
\]

• **Orthogonal vectors:**
\[
\hat{u} \cdot \hat{v} = 0
\]

Projections & Angles

• **Angle between vectors,** \(\theta\)
\[
\hat{u} \cdot \hat{v} = |\hat{u}| \times |\hat{v}| \cos(\theta)
\]

\[
\theta = \cos^{-1}\left(\frac{\hat{u} \cdot \hat{v}}{|\hat{u}| \times |\hat{v}|}\right)
\]

• **Projection of vectors**
\[
\hat{u}_1 = \left(\frac{\hat{u} \cdot \hat{v}}{|\hat{v}|} \right) \hat{v}
\]
\[
\hat{u}_2 = \hat{u} - \hat{u}_1
\]

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...)

Matrix Multiplication

• \([C] = [A][B]\)
• Sum over rows & columns
• **Recall:** matrix multiplication is **not** commutative
• **Identity Matrix:**
\[
1s \text{ on diagonal} \quad c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}
\]
\[
0s \text{ everywhere else}
\]

Matrix Determinants

• A single real number
• Computed recursively
\[
\det(A) = \sum_{j=1}^{n} A_{i,j} (-1)^{i+j} M_{i,j}
\]
• Example:
\[
\det\begin{bmatrix}
a & c \\
0 & d
\end{bmatrix} = ad - bc
\]
• **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 \(B\)
\[
A \times B = (a_xb_y - a_yb_x, a_xb_z - a_zb_x, a_yb_z - a_zb_y)
\]
\[
A \times B = \begin{bmatrix}
a_x & a_y & a_z \\
b_x & b_y & b_z
\end{bmatrix}
\]

\[
\hat{x} \times \hat{y} \times \hat{z}
\]
Matrix Transpose & Inverse

- **Matrix Transpose**: Swap rows and cols:
  \[ A = \begin{bmatrix} 2 \\ 8 \end{bmatrix} \quad A^T = \begin{bmatrix} 2 & 8 \end{bmatrix} \]
- **Facts about the transpose**:
  \[ (A + B)^T = A^T + B^T \]
  \[ (cA)^T = c(A^T) \]
  \[ (AB)^T = B^T A^T \]
- **Matrix Inverse**: Given \( A \), find \( B \) such that \( AB = BA = I \) \( \Rightarrow A^{-1} \)
  (only defined for square matrices)

Scan-Conversion 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
  \( \Rightarrow \Delta x = 1 \)
- \( y_i = \text{Round}(mx_i + B) \)
- 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 than 1
  - \( \Delta x = \) f
  - else \( \Delta y = \) f
- Check for vertical line
  - \( m = \) m
- Compute corresponding \( \Delta y (\Delta x) = m (1/m) \)
- \( x_{i+1} = x_i + \Delta x \)
- \( y_{i+1} = y_i + \Delta y \)
- 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
- 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

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

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

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
- \( a = d_y, \ b = -d_x, \ c = -(q_xr_y - r_xq_y) \)
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 \( 0 \leq \text{slope} \leq 1 \))
- 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 \( NE: M = p_y + 1/2 \)
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:
  \[ D = f(p_x + 1, p_y + (1/2)) \]

Create \( 2a \) for \( x \) and \( 2b \) for \( y \):
- \( 2a(p_x + 1) + 2b(p_y + (1/2)) + 2c \)
- \( 2ap_x + 2bp_y + (2a + b + 2c) \)
**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

\[
\begin{align*}
D_{\text{new}} &= f(p_x + 2, p_y + (1/2)) \\
&= 2a(p_x + 2) + 2b(p_y + \frac{1}{2}) + 2c \\
&= 2ap_x + 2bp_y + (4a + b + 2c) \\
&= D + 2a = D + 2d_y \\
\end{align*}
\]

- Hence, increment by: $2d_y$

**Case I: When E is next**

- What increment for computing a new $D$?
- Next midpoint is: $(p_x + 2, p_y + (1/2))$

**Case II: When NE is next**

- What increment for computing a new $D$?
- Next midpoint is: $(p_x + 2, p_y + 1 + (1/2))$

**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:

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

**The Algorithm**

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

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 $-1 \leq$ slope < 0, always increment $x$, conditionally decrement $y$
- If slope < -1, always decrement $y$, conditionally increment $x$
- Rework D increments

**Generalize Algorithm**

- Reflect line into first case
- Calculate pixels
- Reflect pixels back into original orientation

---

**Bresenham’s Algorithm: Example**

- Reflect line into first case
- Calculate pixels
- Reflect pixels back into original orientation

---

**Bresenham’s Algorithm: Example**

- Reflect line into first case
- Calculate pixels
- Reflect pixels back into original orientation

---

**Bresenham’s Algorithm: Example**

- Reflect line into first case
- Calculate pixels
- Reflect pixels back into original orientation

---

**Bresenham’s Algorithm: Example**

- Reflect line into first case
- Calculate pixels
- Reflect pixels back into original orientation
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
XPM Format

- Encoded pixels
- C code

XPM Basics

- X PixelMap (XPM)
- Native file format in X Windows
- Color cursor and icon bitmaps
- Files are actually C source code
- Read by compiler instead of viewer
- Successor of X BitMap (XBM) B-W format

XPM Supports Color

XPM: Defining Grayscales and Colors

- Each pixel specified by an ASCII char
- key describes the context this color should be used within. You can always use "c" for "color".
- Colors can be specified:
  - color name
  - "#" followed by the RGB code in hexadecimal
- RGB – 24 bits (2 characters '0' - 'f') for each color.

XPM: Specifying Color

<table>
<thead>
<tr>
<th>Color Name</th>
<th>RGB</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>black</td>
<td># 00 00 00</td>
<td></td>
</tr>
<tr>
<td>white</td>
<td># ff ff ff</td>
<td></td>
</tr>
<tr>
<td>red</td>
<td># 80 80 80</td>
<td></td>
</tr>
<tr>
<td>green</td>
<td># ff 00 00</td>
<td></td>
</tr>
<tr>
<td>blue</td>
<td># 00 00 ff</td>
<td></td>
</tr>
</tbody>
</table>

XPM Example

- Array of C strings
- The XPM format assumes the origin (0,0) is in the upper left-hand corner.
- First string is "width height ncolors cpp"
- Then you have "ncolors" strings associating characters with colors.
- And last you have "height" strings of "width * chars_per_pixel" characters
Programming assignment 1

- Input PostScript-like file
- Output B/W XPM
- Primary I/O formats for the course
- Create data structure to hold points and lines in memory \textit{(the world model)}
- Implement 2D translation, rotation and scaling of the world model
- Implement line drawing and clipping
- January 20th
- Get started now!

Questions?

Go to Assignment 1