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: \( u + v = w \)
    - 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

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 \( u_1 x_1 + u_2 x_2 + ... + u_n x_n \), written as \( u \cdot v \)
Fun with Dot Products

• Euclidian 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:
  \[ \sqrt{\mathbf{x} \cdot \mathbf{x}} \]

• This is also the length of vector \(\mathbf{v}\):
  \[ |\mathbf{v}| \text{ or } |\mathbf{v}| \]

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

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

Projections & Angles

• Angle between vectors, \(\theta\)
  \[ \mathbf{\hat{u}} \cdot \mathbf{\hat{v}} = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u||\mathbf{v}}|} \]
  \[ \cos(\theta) = \cos^{-1}\left(\frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u||\mathbf{v}}|}\right) \]

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

Matrices and Matrix Operators

• A \(n\)-dimensional vector:
  \[ \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \]

• Matrix Operations:
  - Addition/Subtraction
  - Identity
  - Multiplication
  - Scalar
  - Matrix Multiplication

• Implementation issue:
  Where does the index start? (0 or 1, it’s up to you…)

• Identity Matrix:
  \[ c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \]
  \[ \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \]

Matrix Multiplication

• Sum over rows & columns
• Recall: matrix multiplication is not commutative

Matrix Determinants

• A single real number
• Computed recursively
  \[ \text{det}(A) = \sum_{j=1}^{n} A_{i,j} (-1)^{i+j} M_{i,j} \]
• Example:
  \[ \text{det} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc \]
• Uses:
  - Find vector orthogonal 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 \(B\)
• \(A \times B = (a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)\)
• \(A \times B = \begin{vmatrix} x & y & z \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} \)
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
  \[ AB = BA = I \]
  \( A^{-1} \) (only defined for square matrices)

---

Line Drawing

---

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
  \[ \Delta x = 1 \]
- \( y_i = \text{Round}(mx_i + B) \)
- 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 \(|\text{slope}|\) is less than 1
  - \( \Delta x = 1 \)
  - \( \Delta y = \text{Round}(m \Delta x) \)
- Check for vertical line
  - \( m = \infty \)
- Compute corresponding \( \Delta y (\Delta x) = m / (1/m) \)
- \( x_{k+1} = x_k + \Delta x \)
- \( y_{k+1} = y_k + \Delta y \)
- Round \((x,y)\) for pixel location
- Issue: Would like to avoid floating point operations

---

Generalizing DDA

- If \(|\text{slope}|\) is less than or equal to 1
  - Ending point should be right of starting point
- If \(|\text{slope}|\) 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(int p0, int y0, int p1, int y1) {
    int dx = p1 - p0, dy = y1 - y0, d = dy - dx, x = p0, y = y0;
    if (d < 0) d = -d, x++; // start at (p0,y0)
    for (; x < p1; x++)
        if (d < 0) d++; // below midpoint - go to E
           else { d -= 2*dy; // above midpoint - go to NE
                  y++; }
}
```

Assumptions:
- \( q_x < r_x \)
- \( 0 \leq \text{slope} \leq 1 \)

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
  - \( f(x, y) < 0 \) for points above the line
  - \( f(x, y) > 0 \) for points below the line
Now…..

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:
  \[ D = f(p_x + 1, p_y + (1/2)) \]
  \[ = 2a(p_x + 1) + 2b(p_y + 1/2) + 2c \]
  \[ = 2ap_x + 2bp_y + (2a + b + 2c) \]
Bresenham’s Algorithm

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

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

Case I: When \( E \) is next

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

Case II: When \( \text{NE} \) 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:

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

The Algorithm

```java
void bresenham(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1, dy = y2 - y1; // line width and height
    int ax = y2 - y1, ay = x2 - x1; // initial decision value
    int y = y1, x = x1; // start at (x,y)
    while (true) {
        if (D <= 0) D += 2*dy; // below endpoint - go to E
            else D += 2*(dy - dx); y++; // above endpoint - go to NE
        else {
            D += 2*dx; x++; // above endpoint - 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 <= slope < 0, always increment \( x \), conditionally decrement \( y \)
- If slope <-1, always decrement \( y \), conditionally increment \( x \)
- Rework \( D \) increments

Bresenham’s Algorithm: Example

\[
\begin{align*}
F(x, y) &= 2(7-x) 
F(x, y) &= 2(5-x) - 7y = 0 
F(x, y) &= 2(5-x) - 7y = 0 
(0, 0) 
(7, 5)
\end{align*}
\]
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></td>
<td># 80 80 80</td>
<td></td>
</tr>
<tr>
<td>red</td>
<td># ff 00 00</td>
<td></td>
</tr>
<tr>
<td>green</td>
<td># 00 ff 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
Plain PBM Image files

- There is exactly one image in a file.
- The "magic number" is "P6" instead of "P4".
- Each pixel in the image is represented by a byte containing ASCII '1' or '0', representing black and white respectively. There are no BLT bits at the end of a row.
- White space in the raster section is ignored.
- You can put any junk you want after the raster, if it starts with a white space character.
- No line should be longer than 70 characters.

Here is an example of a small image in the plain PBM format.

```
P6 10 11
1 1 1 1 0 0 0 0 0 0
1 1 1 0 1 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
1 1 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 XPM
- 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 8th
- Get started now!

Questions?

Go to Assignment 1