Rasterization & Scan Conversion
3D Rendering Pipeline

3D Geometric Primitives

- **Modeling Transformation**: Transform into 3D world coordinate system
- **Lighting**: Illuminate according to lighting and reflectance
- **Viewing Transformation**: Transform into 3D camera coordinate system
- **Projection Transformation**: Transform into 2D camera coordinate system
- **Clipping**: Clip primitives outside camera’s view
- **Scan Conversion**: Draw pixels (includes texturing, hidden surface, etc.)
- **Image**
Overview

- Rasterization:
  - Figure out which pixels to draw/fill?
- Later...
- Shading
  - Determine a color for each filled pixel
- Texture Mapping
  - Describe shading variation within polygon interiors
Rasterization

- Render an image of a geometric primitive by setting pixel colors

```c
void SetPixel(int x, int y, Color rgba)
```

- Example: Filling the inside of a triangle
2D Line

- **Implicit representation:**
  \[ \alpha x + \beta y + \gamma = 0 \]

- **Explicit representation:**
  \[ y = mx + B \]
  \[ m = \frac{y_1 - y_0}{x_1 - x_0} \]

- **Parametric representation:**
  \[ P = \begin{pmatrix} x \\ y \end{pmatrix} \]
  \[ P = P_0 + (P_1 - P_0) t \]
  \[ t \in [0..1] \]
2D Line - Scan Conversion

\[(x_0, y_0) \rightarrow (x_1, y_1)\]
Scan Conversion - Lines

Basic Algorithm

For \( x = x_0 \) to \( x_1 \)
\[
\begin{align*}
y &= mx + B \\
\text{PlotPixel}(x, \text{round}(y))
\end{align*}
\]
end;

For each iteration: 1 float multiplication, 1 addition, 1 Round

- slope = \( m = \frac{y_1 - y_0}{x_1 - x_0} \)
- offset = \( B = y_1 - mx_1 \)
- Assume \( |m| < 1 \)
- Assume \( x_0 < x_1 \)
Scanning the X-axis v.s.
Scanning the Y-axis
Slopes

\begin{align*}
-1 > m > -\infty & \quad 1 < m < \infty \\
y_1 < y_2 & \quad y_1 < y_2 \\
x_2 < x_1 & \\
0 \geq m \geq -1 & \quad x_1 < x_2 \\
0 < m \leq 1 & \\
x_2 < x_1 & \\
y_2 < y_1 & \quad y_2 < y_1 \\
1 < m < \infty & \quad -1 > m > -\infty \\
\end{align*}
Incremental Algorithm

\[ y_{i+1} = mx_{i+1} + B = m(x_i + \Delta x) + B = y_i + m \Delta x \]

if \( \Delta x = 1 \) then  \( y_{i+1} = y_i + m \)

Algorithm

\[ y = y_0 \]
\[ \text{For } x = x_0 \text{ to } x_1 \]
\[ \text{PlotPixel}(x, \text{Round}(y)) \]
\[ y = y + m \]
\[ \text{end;} \]

For each iteration: 1 addition, 1 Round
Full Code

Assume $x_1 > x_0$ and $|m| \leq 1$

```
Line(x_0, y_0, x_1, y_1)
begin
  float dx, dy, x, y, slope;
  dx := x_1 - x_0;
  dy := y_1 - y_0;
  slope := dy/dx;
  y := y_0;
  for x := x_0 to x_1 do
    begin
      PlotPixel( x, Round( y ) );
      y := y + slope;
    end;
end;
```
Symmetric Cases

If $|m| \geq 1$

$x = x_0$
For $y = y_0$ to $y_1$
PlotPixel(round(x),y)
$x = x + 1/m$
end;

if $x_0 > x_1$ for $|m| \leq 1$ or $y_0 > y_1$ for $|m| \geq 1$
swap(((x_0,y_0),(x_1,y_1))

Special Cases:

$m = \pm 1$ (diagonals)
m = 0, \infty (horizontal, vertical)
Still Drawbacks…

- Accumulated error
- Float arithmetic
- Round operations
Accumulating Error

- The error after rounding is always smaller than 0.5
- We begin from integer value and accumulate error at a fixed rate (+m)
- We change the y when accumulated error is larger than 1/2!
Accumulating Error

Line(x0, x1, y0, y1)
    int deltax := x1 - x0
    int deltay := y1 - y0
    real error := 0
    // Assume deltax != 0 (line is not vertical)
    real m := deltay / deltax
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error + m
        if error ≥ 0.5 then
            y := y + 1
            error := error - 1.0

For each iteration: 1 float addition
Moving to Integers

- Multiply everything by $dx$ we get
  - $m \rightarrow dy$
  - $error \rightarrow error \cdot dx$
  - $error = error + m \rightarrow error = error + dy$
  - $error > \frac{1}{2} \rightarrow error > \frac{dx}{2}$
  - $error = error - 1 \rightarrow error = error - dx$
Bresenham Line Drawing

Line(x₀, x₁, y₀, y₁)

int deltax := x₁ - x₀
int deltay := y₁ - y₀
int error := 0
int over := deltax/2

// Assume deltax != 0 (line is not vertical)
int y := y₀
for x from x₀ to x₁
    plot(x, y)
    error := error + deltay
    if error > over then
        y := y + 1
        error := error - deltax

For each iteration: 1 int addition
Drawing Circles

- Implicit representation (centered at the origin with radius R):
  \[ x^2 + y^2 - R^2 = 0 \]

- Explicit representation:
  \[ y = \pm \sqrt{R^2 - x^2} \]

- Parametric representation:
  \[
  \begin{pmatrix}
  x \\
  y
  \end{pmatrix}
  =
  \begin{pmatrix}
  R \cos (t) \\
  R \sin (t)
  \end{pmatrix}
  \]

  \[ t \in [0..2\pi] \]
Scan Conversion - Circles

**Basic Algorithm**

```
For x = -R to R
    y = sqrt(R^2 - x^2)
    PlotPixel(x, round(y))
    PlotPixel(x, -round(y))
end;
```

**Comments:**

- Square-root operations are expensive.
- Float arithmetic.
- Large gap for x values close to R.
For a circle centered at the origin:
If \((x,y)\) is on the circle then
\[-(y,x) \ (y,-x) \ (x,-y) \ (-x,-y) \ (-y,-x) \ (-y,x) \ (-x,y)\]
are on the circle as well.
Therefore we need to compute only one octant \((45^\circ)\) segment.
Circle Midpoint Algorithm

- The circle is located at (0,0) with radius R.
- We start from \((x_0, y_0) = (0, R)\).
- One can move either East or South-East.
- \(d(x, y)\) will be a threshold criteria at the midpoint.
Threshold Criteria

- Key Idea: check the sign of the function at the midpoint if the circle is above or below the midpoint

\[ d(x,y) = f(x,y) = x^2 + y^2 - R^2 \]
Criterion

• At the beginning \((X_0 = 0, Y_0 = R)\).

\[
d_{\text{start}} = d(x_0+1, y_0-1/2) = d(1, R-1/2) = 1 + R^2 - R + 1/4 - R^2 = 5/4 - R
\]

• If \(d > 0\) we move \textit{South-East}:

\[
\Delta_{SE} = d(x_0+2, y_0-3/2) - d(x_0+1, y_0-1/2) = 2(x_0 - y_0) + 5
\]

• If \(d < 0\) we move \textit{East}:

\[
\Delta_{E} = d(x_0+2, y_0-1/2) - d(x_0+1, y_0-1/2) = 2x_0 + 3
\]
Comments

• $\Delta_E$ and $\Delta_{SE}$ are not constants.
• Since d is incremented by integer values, we can use $d_{\text{start}} = 1-R$, yielding an integer algorithm. This has no affect on the threshold criteria.
Midpoint Circle Algorithm

Circle Octant2 (R)
begin
int x, y, d;
x := 0;
y := R;
d := 1-R;
PlotPixel(x,y);
while ( y>x ) do
  if ( d<0 ) then /* East */
    begin
      d := d+2x+3;
x := x+1;
    end;
  else begin /* South East */
    d := d+2(x-y)+5;
x := x+1;
y := y-1;
PlotPixel( x,y );
end;
Can draw all 8 octants
Triangle Rasterization

- Properties of a good algorithm
  - Symmetric
  - Straight edges
  - Antialiased edges
  - No cracks between adjacent primitives
  - MUST BE FAST!
Simple Algorithm

- Color all pixels inside triangle

```c
void ScanTriangle(Triangle T, Color rgba){
    for each pixel P at (x,y){
        if (Inside(T, P))
            SetPixel(x, y, rgba);
    }
}
```
Inside Triangle Test

- A point is inside a triangle if it is in the positive halfspace of all three boundary lines.
  - Triangle vertices are ordered counter-clockwise.
  - Point must be on the left side of every boundary line.
Inside Triangle Test

```c
Boolean Inside(Triangle T, Point P) {
    for each boundary line L of T {
        d = L(Px, Py);
        if (d < 0.0) return FALSE;
    }
    return TRUE;
}
```

$L(x,y) = ax + by + c$
Triangle Sweep-Line

- Take advantage of spatial coherence
  - Compute which pixels are inside using horizontal spans
  - Process horizontal spans in scan-line order

- Take advantage of edge linearity
  - Use edge slopes to update coordinates incrementally
Sweep-Line Algorithm

```c
void ScanTriangle(Triangle T, Color rgba) {
    for each edge pair {
        initialize $x_L$, $x_R$;
        compute $dx_L/dy_L$ and $dx_R/dy_R$;
        for each scanline at $y$
            for (int $x = x_L$; $x <= x_R$; $x++$)
                SetPixel($x$, $y$, rgba);
        $x_L += dx_L/dy_L$;
        $x_R += dx_R/dy_R$;
    }
}
```

Use Bresenham Algorithm
Rasterizing Triangles

- There are two questions:
  1. Which pixels to draw?
  2. What color to give them?
Color: Use Interpolation!

- Assume we know the color at vertices (later…)
- We want to determine color values of points (pixels) inside the triangle based on the vertices
- How?

Convex combination $\rightarrow$ Barycentric Coordinates:

$$\alpha_1 + \alpha_2 + \alpha_3 = 1$$  \quad \Rightarrow \quad F(p) = \alpha_1 F_1 + \alpha_2 F_2 + \alpha_3 F_3$$

$$p = \alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3$$
Simpler:
(Bi-)Linear Interpolation

- Assumption: Data varies linearly between adjacent data points (vertices).
- On triangle edges:
  \[(1-t)f_0 + tf_1\] when \(0 \leq t \leq 1\)
- Linear interpolation (a mapping from \([0,1]\)):
- What happens inside the triangles?
Bi-Linear Interpolation

- We can use two linear interpolations:
f varies linearly in the triangle

- This means we can find a plane
  \[ f(x,y) = Ax + By + C \]
- If we find A, B, C we can set \( f(x_0, y_0) = Ax_0 + By_0 + C \)
- But the plane passes through \((x_1, y_1, f_1)\), \((x_2, y_2, f_2)\) and \((x_3, y_3, f_3)\) so:

\[
\begin{align*}
  f_1 &= Ax_1 + By_1 + C \\
  f_2 &= Ax_2 + By_2 + C \\
  f_3 &= Ax_3 + By_3 + C
\end{align*}
\]

\[
\begin{pmatrix}
  x_1 & y_1 & 1 \\
  x_2 & y_2 & 1 \\
  x_3 & y_3 & 1
\end{pmatrix}
\begin{pmatrix}
  A \\
  B \\
  C
\end{pmatrix}
= 
\begin{pmatrix}
  f_1 \\
  f_2 \\
  f_3
\end{pmatrix}
\]

\[
\begin{pmatrix}
  A \\
  B \\
  C
\end{pmatrix}
= 
\begin{pmatrix}
  x_1 & y_1 & 1 \\
  x_2 & y_2 & 1 \\
  x_3 & y_3 & 1
\end{pmatrix}^{-1}
\begin{pmatrix}
  f_1 \\
  f_2 \\
  f_3
\end{pmatrix}
\]
Polygon Rasterization (non-triangular)

- Fill pixels inside a polygon
  - Triangle
  - Quadrilateral
  - Convex
  - Star-shaped
  - Concave
  - Self-intersecting
  - Holes

What problems do we encounter with arbitrary polygons?
Polygon Scan Conversion

- Need better test for points inside polygon
  - Triangle method works only for convex polygons

Convex Polygon

Concave Polygon
Inside Polygon Rule

What is a good rule for which pixels are inside?

- Concave
- Self-Intersecting
- With Holes
Inside Polygon Rule

- Odd-parity rule
  - Any ray from P to infinity crosses odd number of edges

Concave

Self-Intersecting

With Holes
Flood Fill

- Assume a polygon is simple (no self intersections, no holes)
- Let \( P \) be a polygon with \( n \) vertices, \( v_0 \) to \( v_{n-1} \). Denote \( v_n = v_0 \).
- Let \( c \) be the color to paint this polygon.
- Let \( p = (x, y) \) be a point in \( P \).
Flood Fill Algorithm

\[
\text{FloodFill}(P, x, y, c) \\
\text{if } (\text{OnBoundary}(x, y, P) \text{ or Colored } (x, y, c)) \\
\quad \text{then return;} \\
\text{else begin} \\
\quad \text{PlotPixel}(x, y, c); \\
\quad \text{FloodFill}(P, x+1, y, c); \\
\quad \text{FloodFill}(P, x, y+1, c); \\
\quad \text{FloodFill}(P, x, y-1, c); \\
\quad \text{FloodFill}(P, x-1, y, c); \\
\text{end;}
\]

Comment: Slow algorithm due to recursion.
Polygon Sweep-Line Algorithm

- Incremental algorithm to find spans
  - Determine insideness with odd parity rule
  - Takes advantage of scanline coherence
Scan Conversion - Basic Algorithm

ScanConvert (P,c)
For j:=0 to ScreenYMax do
  I := points of intersection of edges from P with line y=j;
  Sort I in increasing x order and fill with color c alternating segments;
end;

How do we find the intersecting edges?
What happens in such cases?
Polygon Sweep-Line Algorithm

One can maintain an active list of edges active_edge_list, that contains the edges that currently intersect with the scan line:

```c
void ScanPolygon(Triangle T, Color rgba){
    sort edges by maxy
    make empty "active_edge_list"
    for each scanline (top-to-bottom) {
        insert/remove edges from "active_edge_list"
        update x coordinate of every active edge
        sort intersections by x coordinate
        for each pair of intersections (left-to-right)
            SetPixels(x_i, x_{i+1}, y, rgba);
    }
}
```
Implementation with Linked List

**Edge List**

(Ordered by $y_{\text{min}}$) $\rightarrow$ easy to insert

(Ordered by $y_{\text{max}}$) $\rightarrow$ easy to insert

**Active Edges**

(Ordered by $y_{\text{max}}$) $\rightarrow$ easy to remove

(Ordered by $y_{\text{min}}$) $\rightarrow$ easy to remove
Why do we need x-sorting?

- Edge (a,b) will be placed before edge (c,d) in the active list because it has smaller yMax.
- Therefore the right intersection point will be found before the left intersection point.
<table>
<thead>
<tr>
<th>Flood Fill</th>
<th>Scan Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very simple.</td>
<td>More complex.</td>
</tr>
<tr>
<td>Requires a seed point.</td>
<td>No seed point is required.</td>
</tr>
<tr>
<td>Requires very large stack size.</td>
<td>Requires small stack size.</td>
</tr>
<tr>
<td>Common in paint packages.</td>
<td>Used in image rendering.</td>
</tr>
<tr>
<td>Unsuitable for line based Z-buffer.</td>
<td>Suitable for line based Z-buffer.</td>
</tr>
</tbody>
</table>
Hardware Scan Conversion

- Convert everything into triangles
  - Scan convert the triangles
Aliasing

Jaggies
Anti-Aliasing
Anti-Aliasing
Hardware Antialiasing

- Supersample pixels
  - Multiple samples per pixel
  - Average subpixel intensities (box filter)
  - Trades intensity resolution for spatial resolution
Super Sampling

Original 100x100

Original 50x50

Averaged and reduced to 50x50

(+ simple & general
(-) x4 memory size
(-) x4 scan conversion + reduction