CS-184: Computer Graphics

Lecture 12: Scan Conversion

Maneesh Agrawala
University of California, Berkeley

Slides based on those of James O'Brien and Greg Humphreys

Announcements

~~Assignment 4: due Fri Oct 8 by 11pm~~

Midterm: Wed Oct 13

Assignment 5: due Fri Nov 5 by 11pm

Today

2D Scan Conversion

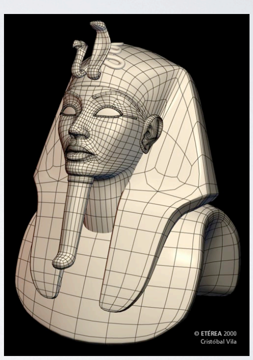
- Drawing Lines
- Filling Polygons
- Shading Polygons

Line Drawing

Drawing a Line

Basically, its easy... but for the details

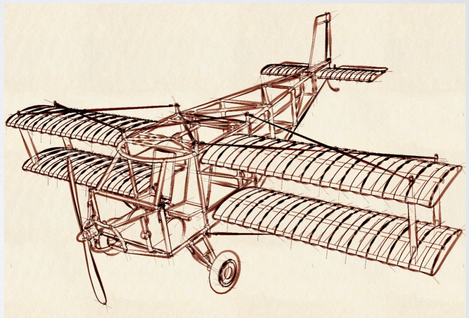
Lines are a basic primitive that needs to be done well...



Drawing a Line

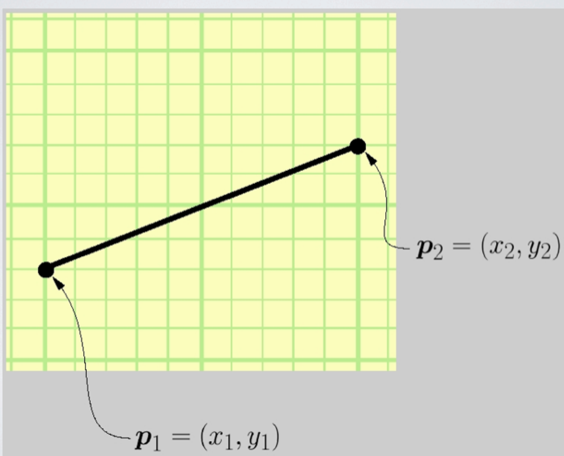
Basically, its easy... but for the details

Lines are a basic primitive that needs to be done well...

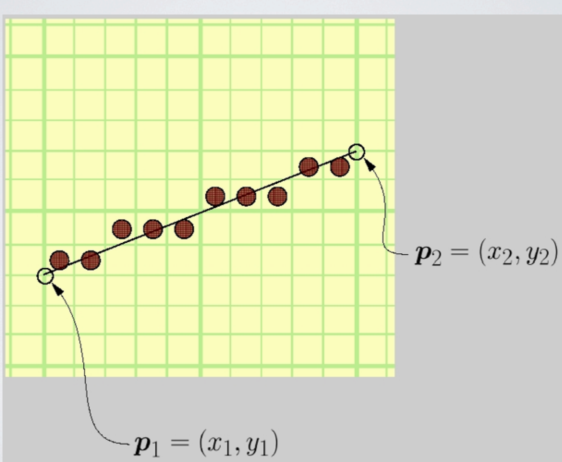


From "A Procedural Approach to Style for NPR Line Drawing from 3D models," by Grabli, Durand, Turquin, Sillion

Drawing a Line



Drawing a Line

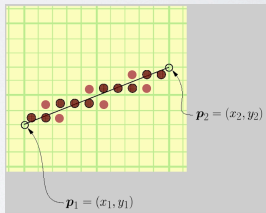


Drawing a Line

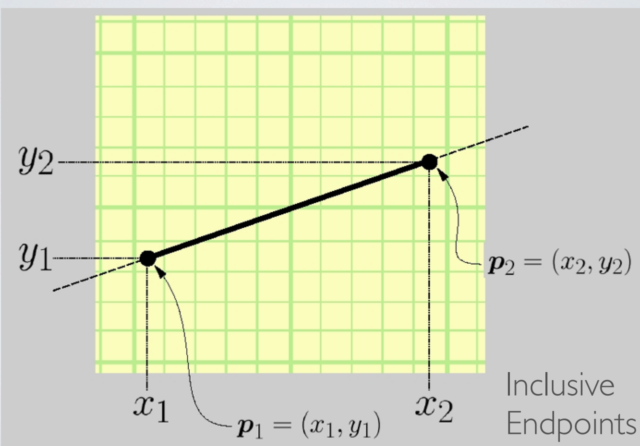
Some things to consider

- How thick are lines?
- How should they join up?
- Which pixels are the right ones?

For example:



Drawing a Line

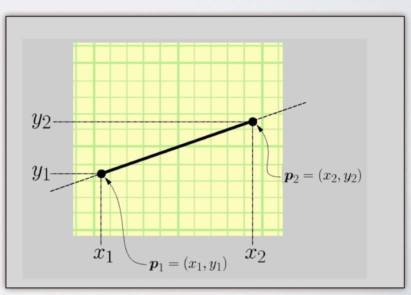


Drawing a Line

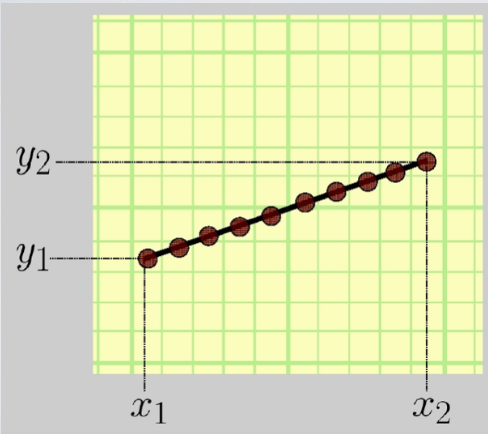
$$y = m \cdot x + b, x \in [x_1, x_2]$$

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

$$b = y1 - m \cdot x_1$$



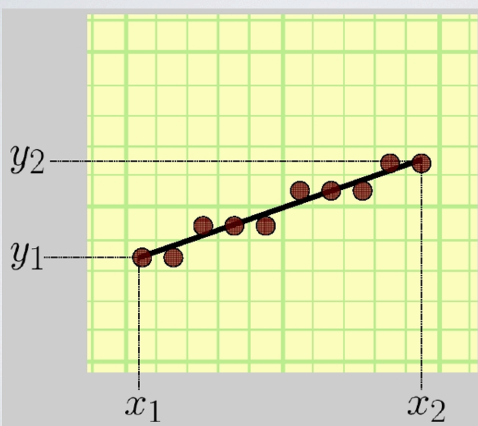
Drawing a Line



$\Delta x = 1$
$\Delta y = m \cdot \Delta x$

```
x=x1
y=y1
while(x<=x2)
   plot(x,y)
   x++
   y+=Dy
```

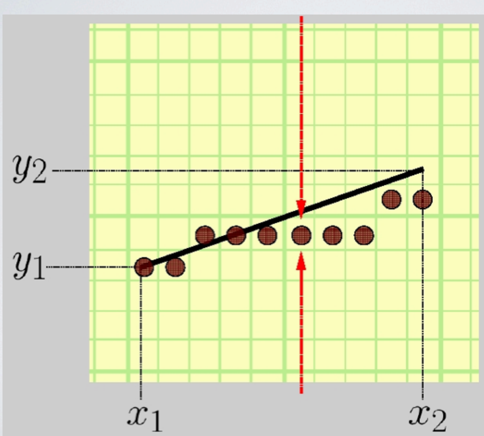

Drawing a Line



$\Delta x = 1$
$\Delta y = m \cdot \Delta x$

After rounding

Drawing a Line

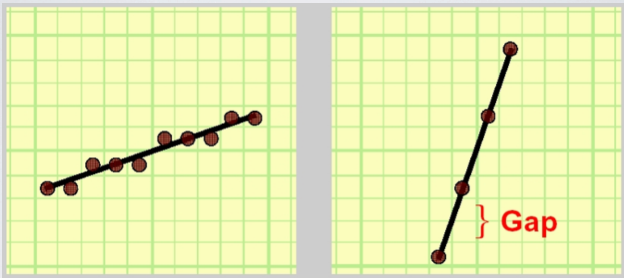


$\Delta x = 1$
$\Delta y = m \cdot \Delta x$
$y {+}{=}\ \Delta y$

Accumulation of roundoff errors

How slow is float-to-int conversion?

Drawing a Line



$|m| \leq 1$ $\qquad$ $|m| > 1$

Drawing a Line

```
void drawLine-Error1(int x1,x2, int y1,y2)

  float m = float(y2-y1)/(x2-x1)
  int x = x1
  float y = y1

  while (x <= x2)

    setPixel(x,round(y),PIXEL_ON)

    x += 1
    y += m
```

Not exact math

Accumulates errors

Drawing a Line

```
void drawLine-Error2(int x1,x2, int y1,y2)

  float m = float(y2-y1)/(x2-x1)
  int x = x1
  int y = y1
  float e = 0.0

  while (x <= x2)

    setPixel(x,y,PIXEL_ON)

    x += 1
    e += m
    if (e >= 0.5)
      y+=1
      e-=1.0
```

No more rounding

Drawing a Line

```
void drawLine-Error3(int x1,x2, int y1,y2)

  int x = x1
  int y = y1
  float e = -0.5

  while (x <= x2)

    setPixel(x,y,PIXEL_ON)

    x += 1
    e += float(y2-y1)/(x2-x1)
    if (e >= 0.0)
      y+=1
      e-=1.0
```


Drawing a Line

```
void drawLine-Error4(int x1,x2, int y1,y2)

  int x = x1
  int y = y1
  float e = -0.5*(x2-x1)          // was -0.5

  while (x <= x2)

    setPixel(x,y,PIXEL_ON)

    x += 1
    e += y2-y1                    // was /(x2-x1)
    if (e >= 0.0)                 // no change
      y+=1
      e-=(x2-x1)                  // was 1.0
```


Drawing a Line

```
void drawLine-Error5(int x1,x2, int y1,y2)

  int x = x1
  int y = y1
  int e = -(x2-x1)                // removed *0.5

  while (x <= x2)

    setPixel(x,y,PIXEL_ON)

    x += 1
    e += 2*(y2-y1)                // added 2*
    if (e >= 0.0)                 // no change
      y+=1
      e-=2*(x2-x1)                // added 2*
```

Drawing a Line

```
void drawLine-Bresenham(int x1,x2, int y1,y2)

  int x = x1
  int y = y1
  int e = -(x2-x1)

  while (x <= x2)

    setPixel(x,y,PIXEL_ON)

    x += 1
    e += 2*(y2-y1)
    if (e >= 0.0)
      y+=1
      e-=2*(x2-x1)
```

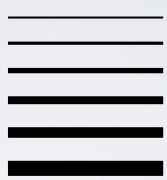
Faster
Not wrong

$|m| \leq 1$

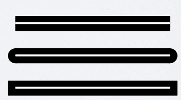
$x_1 \leq x_2$

Drawing a Line

How thick?

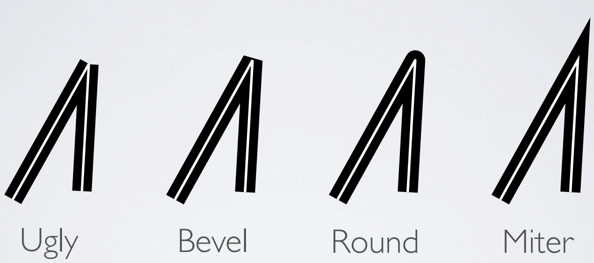


Ends?



Drawing a Line

Joining?



Filling Polygons

Flood Fill

The idea: fill a "connected region" with a solid color

Term definitions:

8	4	8
4	1	4
8	4	8

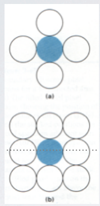
The center "1" pixel is ***4-connected*** to the pixels marked "4", and ***8-connected*** to the pixels marked "8"

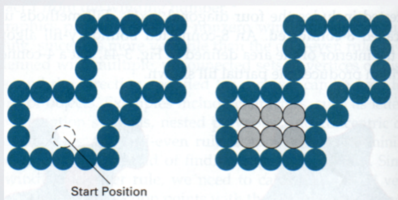
Simple 4-Connected Fill

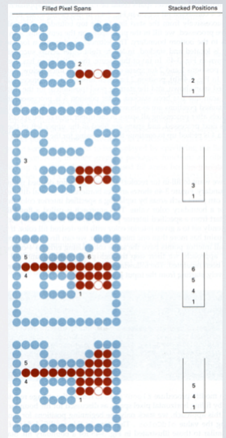
The simplest algorithm to fill a 4-connected region is a recursive one:

```
FloodFill( int x, int y, int inside_color, int new_color )
{
  if (GetColor( x, y ) == inside_color)
  {
     SetColor( x, y, new_color );
     FloodFill( x+1, y  , inside_color, new_color );
     FloodFill( x-1, y  , inside_color, new_color );
     FloodFill( x,   y+1, inside_color, new_color );
     FloodFill( x,   y-1, inside_color, new_color );
  }
}
```

Flood Fill

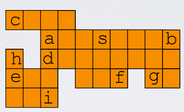






Span-Based Algorithm

Definition: a ***run*** is a horizontal span of identically colored pixels



1. Start at pixel "s", the seed.
2. Find the run containing "s" ("b" to "a").
3. Fill that run with the new color.
4. Search every pixel above run, looking for pixels of interior color
5. For each one found,
6. Find left side of that run ("c"), and push that on a stack.
7. Repeat lines 4-7 for the pixels below ("d").
8. Pop stack and repeat procedure with the new seed

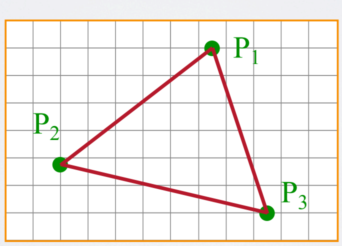
The algorithm finds runs ending at "e", "f", "g", "h", and "i"

Filling Triangles

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

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

- Example: Filling the inside of a triangle

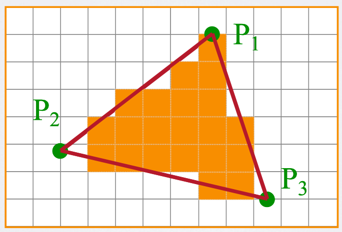


Filling Triangles

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

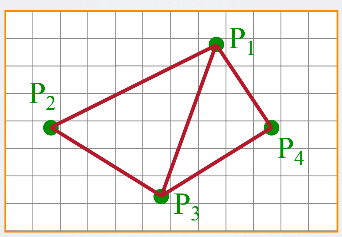
```
void SetPixel(int x, int y, Color rgba)
```

- Example: Filling the inside of a triangle



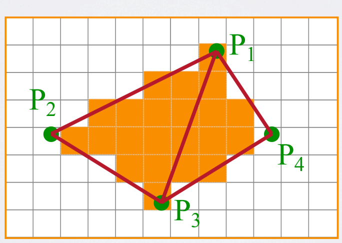
Triangle Scan Conversion

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



Triangle Scan Conversion

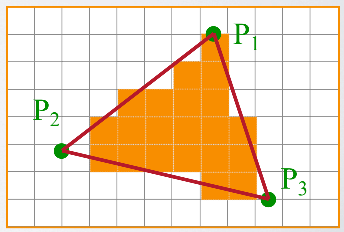
- Properties of a good algorithm
  - Symmetric
  - Straight edges
  - Antialiased edges
  - No cracks between adjacent primitives
  - MUST BE FAST!



Simple Algorithm

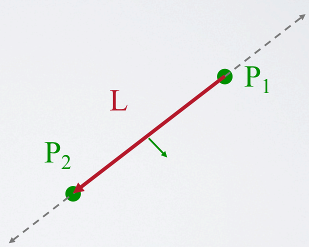
- Color all pixels inside triangle

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



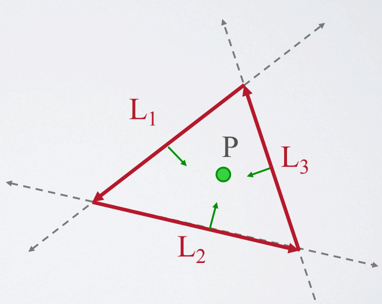
Line Defines Two Halfspaces

- Implicit equation for a line
  - On line: $ax + by + c = 0$
  - On right: $ax + by + c < 0$
  - On left: $ax + by + c > 0$



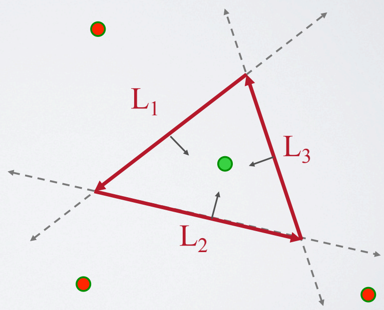
Inside Triangle Test

- Point is inside triangle if it is in 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

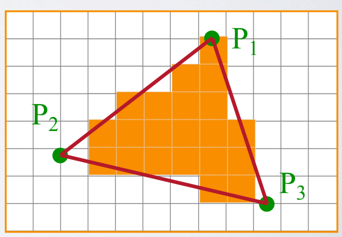
```
Boolean Inside(Triangle T, Point P)
{
   for each boundary line L of T {
      Scalar d = L.a*P.x + L.b*P.y + L.c;
      if (d < 0.0) return FALSE;
   }
   return TRUE;
}
```



Simple Algorithm

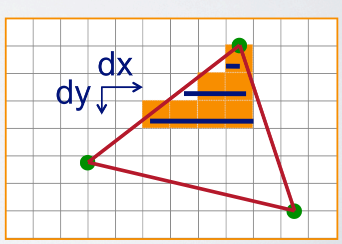
- What is bad about this algorithm?

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



Triangle Sweep-Line Algorithm

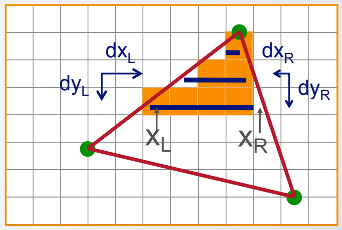
- Take advantage of spatial coherence
  - Compute which pixels are inside using horizontal spans
  - Process horizontal spans in scan-line order
- Take advantage of edge linearity
  - Use edge slopes to update coordinates incrementally



Triangle Sweep-Line Algorithm

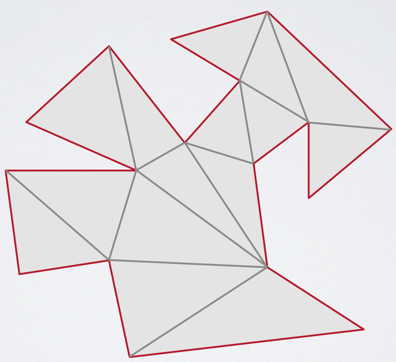
```
void ScanTriangle(Triangle T, Color rgba){
    for each edge pair {
        initialize xL, xR;
        compute dxL/dyL and dxR/dyR;
        for each scanline at y
            for (int x = ceil(xL); x <= xR; x++)
                SetPixel(x, y, rgba);
        xL += dxL/dyL;
        xR += dxR/dyR;
    }
}
```

Bresenham's algorithm works the same way, but uses only integer operations!



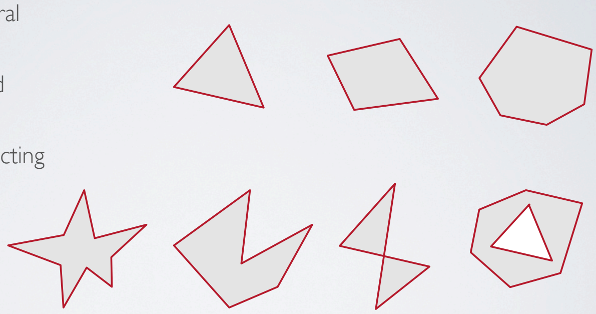
Hardware Scan Conversion

- Convert everything into triangles
  - Scan convert the triangles



Polygon Scan Conversion

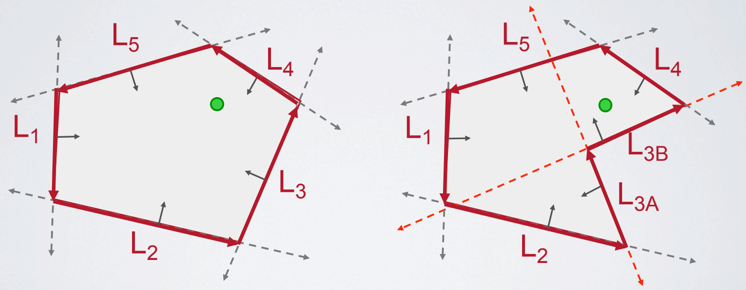
- Fill pixels inside a polygon
  - Triangle
  - Quadrilateral
  - Convex
  - Star-shaped
  - Concave
  - Self-intersecting
  - Holes



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

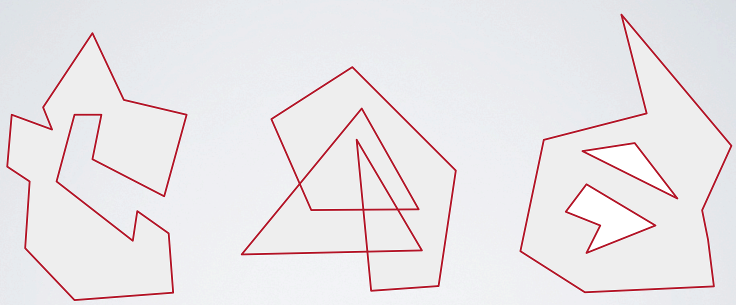


Convex Polygon

Concave Polygon

Inside Polygon Rule

- What is a good rule for which pixels are inside?



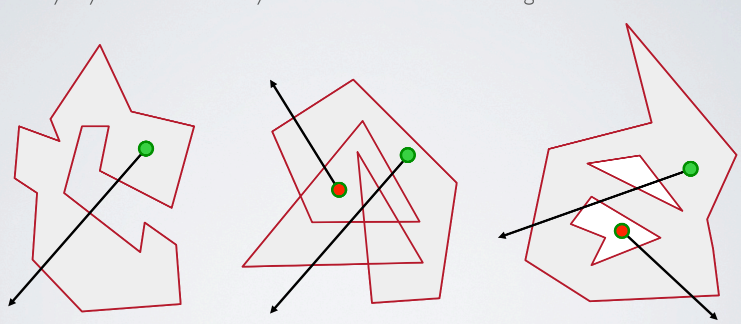
Concave

Self-Intersecting

With Holes

Inside Polygon Rule

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



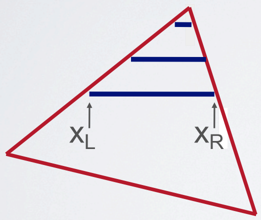
Concave

Self-Intersecting

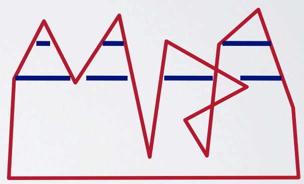
With Holes

Polygon Sweep-Line Algorithm

- Incremental algorithm to find spans, and determine insideness with odd parity rule
  - Takes advantage of scanline coherence



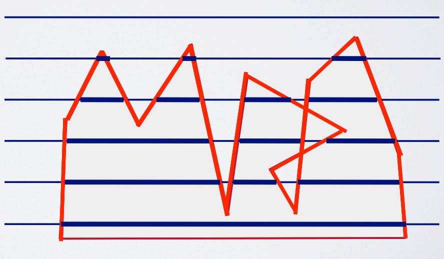
Triangle



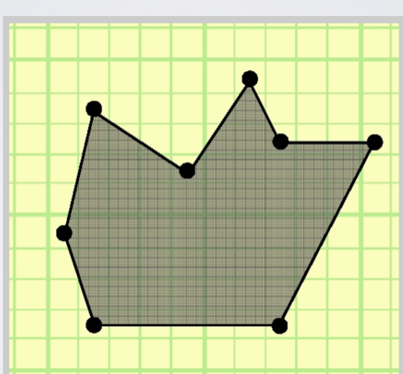
Polygon

Polygon Sweep-Line Algorithm

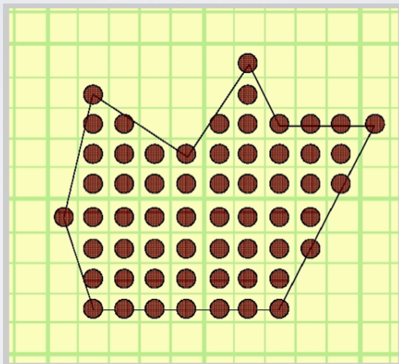
```
void ScanPolygon(Triangle T, Color rgba){
   sort edges by maxy
   make empty "active edge list"
   for each scanline (top-to-bottom) {
   insert/remove edges from "active edge list"
       update x coordinate of every active edge
       sort active edges by x coordinate
       for each pair of active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



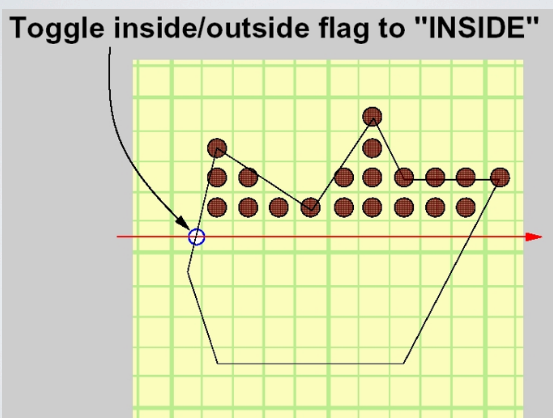
Filled Polygons



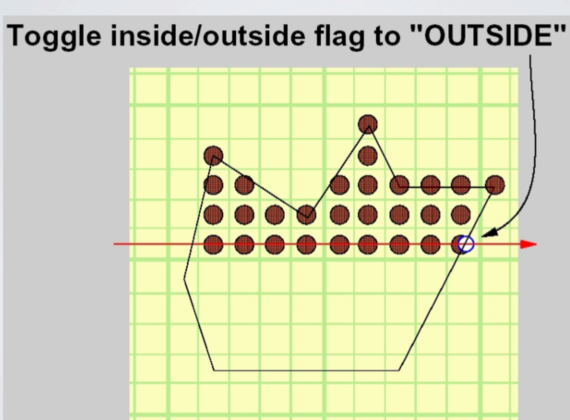
Filled Polygons



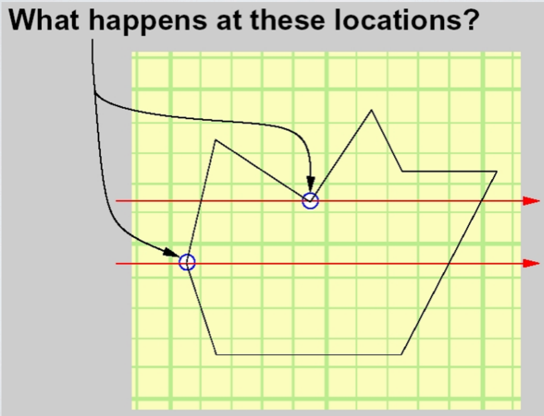
Filled Polygons



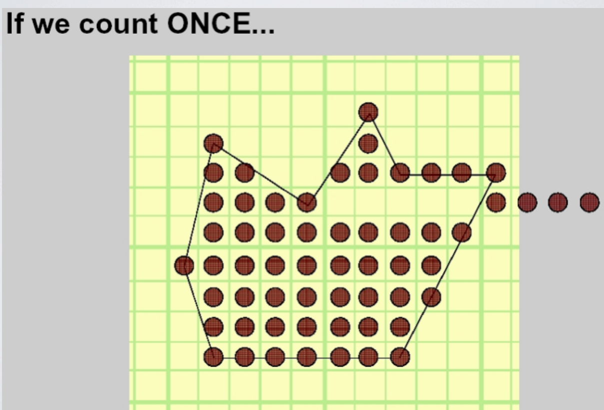
Filled Polygons



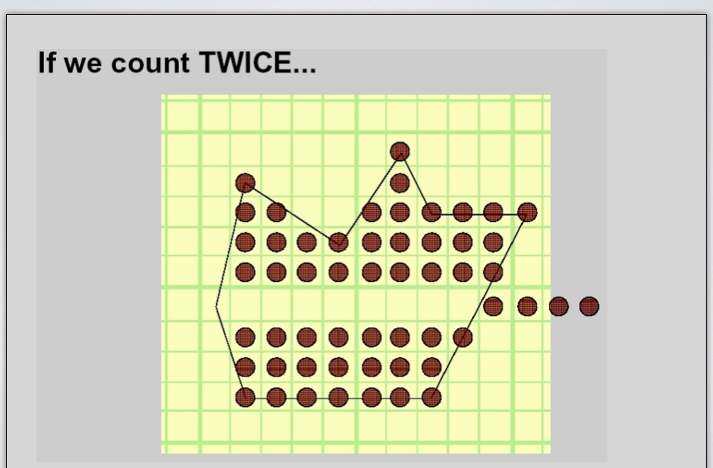
Filled Polygons



Filled Polygons

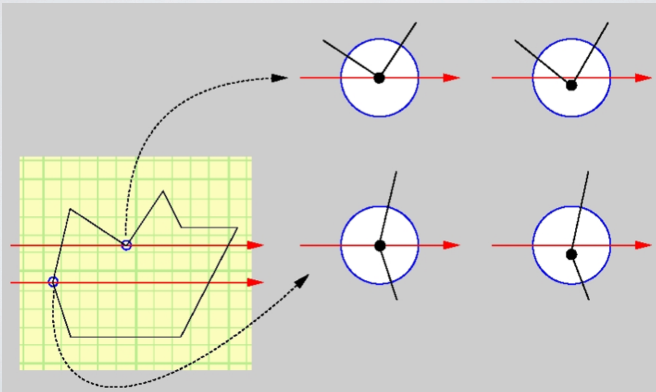


Filled Polygons



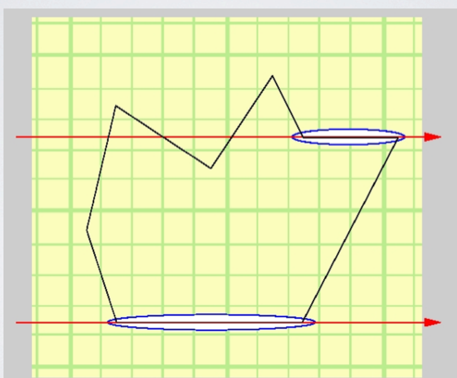
Filled Polygons

Treat (scan y = vertex y) as (scan y > vertex y)



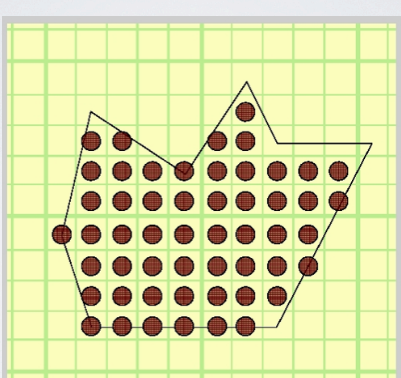
Filled Polygons

Horizontal edges



Filled Polygons

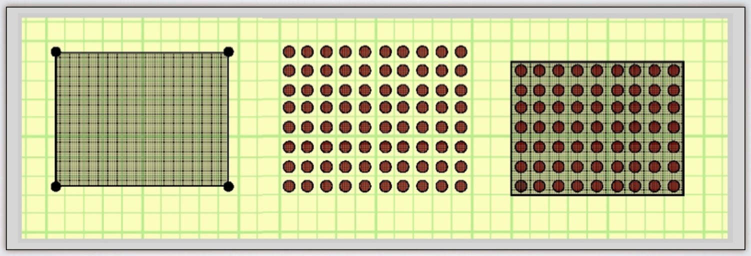
Final result:



Filled Polygons

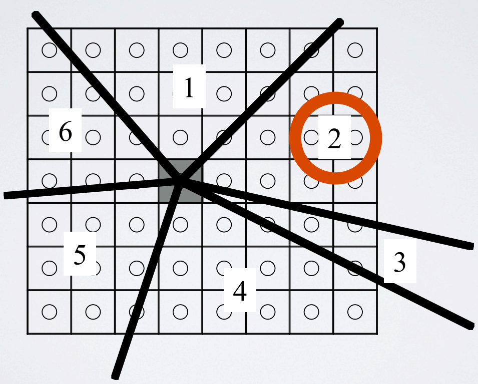
"Equality Removal" applies to all vertices

Both x and y coordinates



Filled Polygons

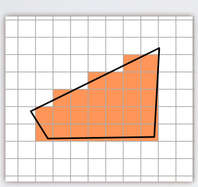
Who does this pixel belong to?

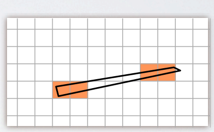


Antialiasing

Boolean on/off for pixels causes problems

- Consider scan conversion algorithm:





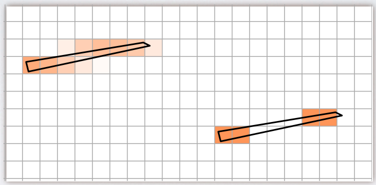
- Compare to casting a ray through each pixel center

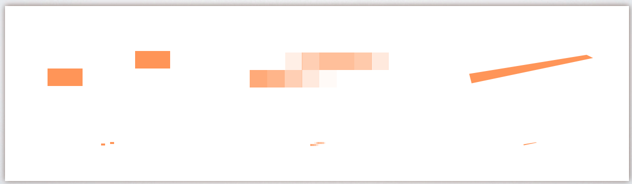
Recall Nyquist Theorem

- *Sampling rate ≥ twice highest frequency*

Antialiasing

Desired solution of an integral over pixel

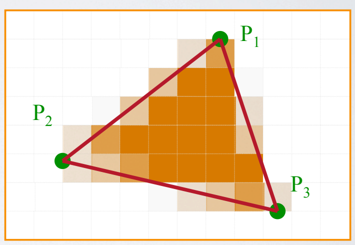




Hardware Antialiasing

Supersample pixels

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



Shading Triangles

Shading

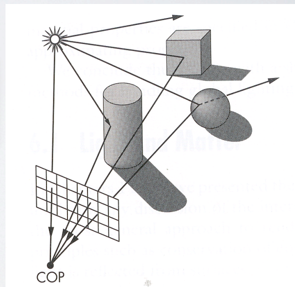
How do we choose a color for each filled pixel?

Each illumination calculation for a ray from the viewpoint through the image plane provides a radiance sample

How do we choose where to place samples?

How do we filter samples to reconstruct image?

Emphasis on methods that can be implemented in hardware

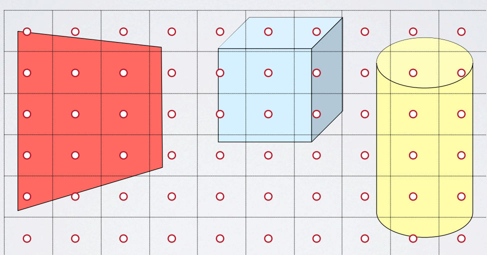


Ray Tracing

Simple approach

Perform independent lighting calculation for every pixel

When is this unnecessary?

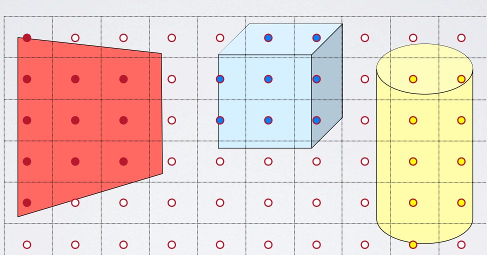


$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

Polygon Shading

Can take advantage of spatial coherence

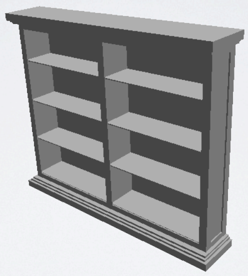
Illumination calculations for pixels covered by same primitive are related



$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

Flat Shading

What if a faceted object is illuminated only by directional light sources and is either diffuse or viewed from infinitely far away

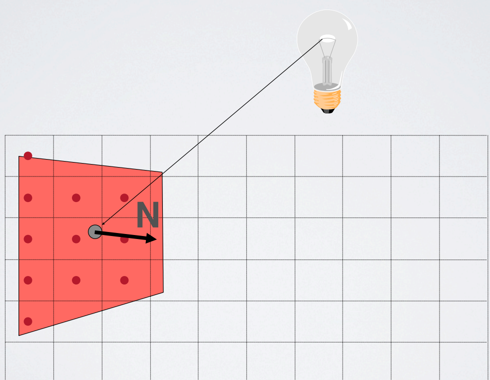


$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

Flat Shading

One illumination calculation per polygon

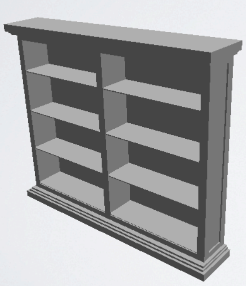
- Assign all pixels inside each polygon the same color

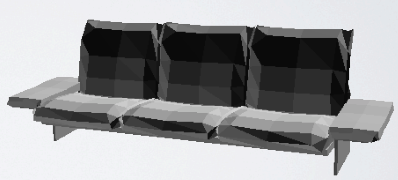


Flat Shading

Objects look like they are composed of polygons

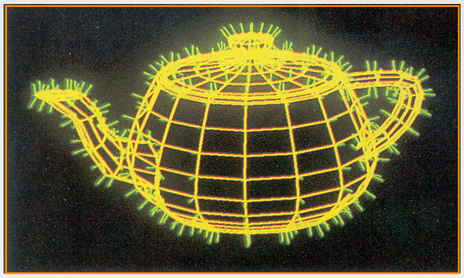
- OK for polyhedral objects
- Not so good for smooth surfaces





Gouraud Shading

What if smooth surface is represented by polygonal mesh with a normal at each vertex?



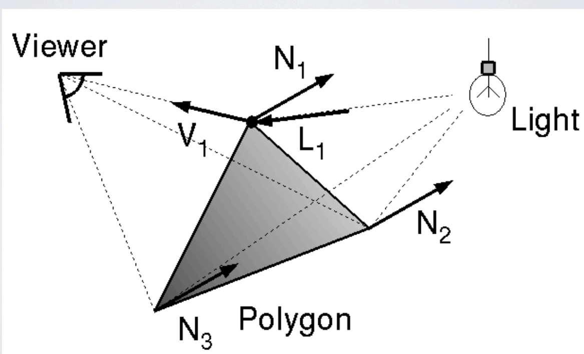
Watt Plate 7

$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

Gouraud Shading

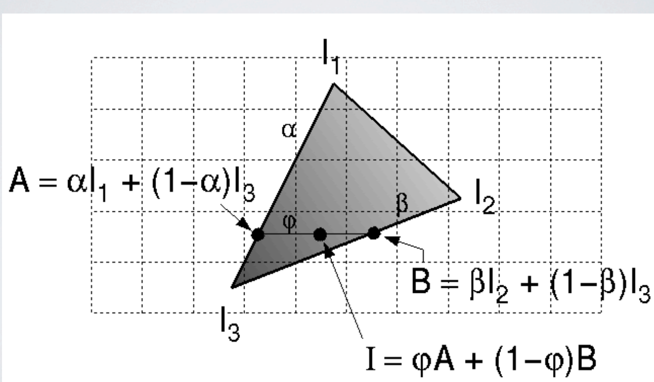
Method 1: One lighting calculation per vertex

- Assign pixels inside polygon by interpolating colors computed at vertices



Gouraud Shading

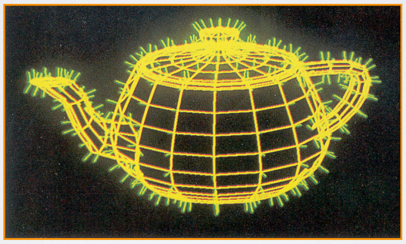
Bilinearly interpolate colors at vertices down and across scan lines



Gouraud Shading

Smooth shading over adjacent polygons

- Curved surfaces
- Illumination highlights
- Soft shadows



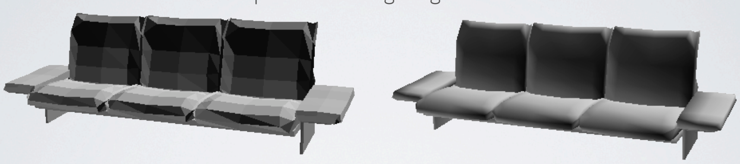
Mesh with shared normals at vertices

Watt Plate 7

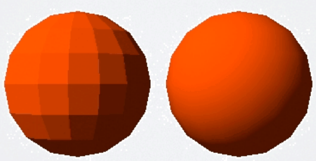
Gouraud Shading

Produces smoothly shaded polygonal mesh

- Piecewise linear approximation
- Need fine mesh to capture subtle lighting effects



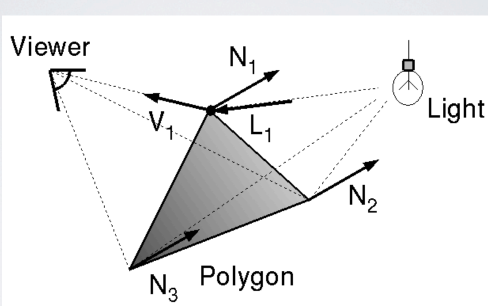
Flat Shading Gouraud Shading



Flat **Gouraud**

Phong Shading

What if polygonal mesh is too coarse to capture illumination effects in polygon interiors?

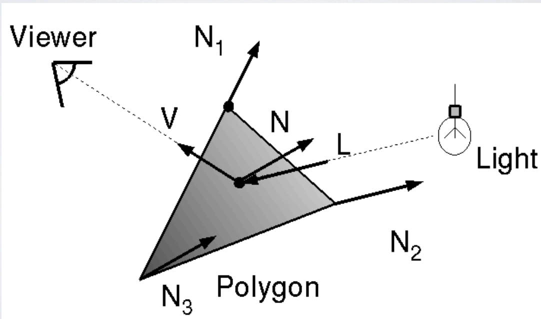


$$I = I_E + K_A I_{AL} + \sum_i (K_D (N \bullet L_i) I_i + K_S (V \bullet R_i)^n I_i)$$

Phong Shading

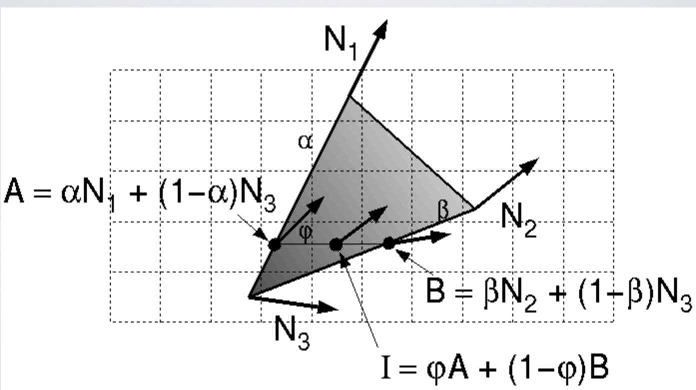
Method 2: One lighting calculation per pixel

- Approximate surface normals for points inside polygons by bilinear interpolation of normals from vertices

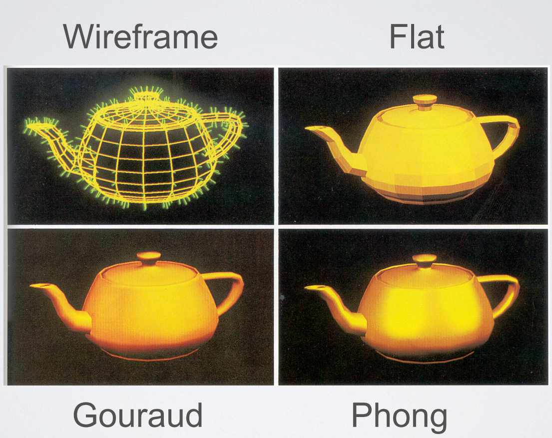


Phong Shading

Bilinearly interpolate surface normals at vertices down and across scan lines



Polygon Shading Algorithms

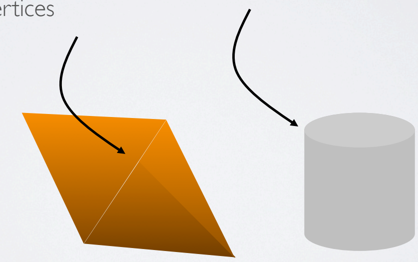


Watt Plate 7

Shading Issues

Problems with interpolated shading:

- Polygonal silhouettes
- Perspective distortion
- Orientation dependence (due to bilinear interpolation)
- Problems computing shared vertex normals
- Problems at T-vertices



Summary

2D polygon scan conversion

- Paint pixels inside primitive
- Sweep-line algorithm for polygons

Polygon Shading Algorithms

- Flat
- Gouraud
- Phong
- Ray casting

Less expensive

More accurate

Key ideas:

- Sampling and reconstruction
- Spatial coherence