Announcements

- Wed-Sat on travel
  - Limited email access
  - Guest lecture Thurs by Wes Griffin on OpenGL
- Project 2
  - Status/issues
Graphics Pipeline

- Object-order approach to rendering
- Sequence of operations
  - Vertex processing
    - Transforms
    - Viewing
    - Vertex components of shading/texture
  - Rasterization
    - Break primitives into fragments/pixels
    - Clipping
  - Fragment processing
    - Fragment components of shading/texture
  - Blending

Line Drawing

- Given endpoints of line, which pixels to draw?
Line Drawing

- Given endpoints of line, which pixels to draw?

- Assume one pixel per column (x index), which row (y index)?
- Choose based on relation of line to midpoint between candidate pixels
**Line Drawing**

- **Implicit representation**
  - \( f(x,y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0 = 0 \)
  - Slope \( m = \frac{y_1 - y_0}{x_1 - x_0} \) (assume \( 0 \leq m \leq 1 \))

- **Midpoint algorithm**
  
  \[
  \begin{align*}
  y &= y_0 \\
  d &= f(x_0 + 1, y_0 + 0.5) \\
  \text{for } x &= x_0 \text{ to } x_1 \text{ do} \\
  &\quad \text{draw } (x,y) \\
  &\quad \text{if } (d < 0) \text{ then} \\
  &\quad \quad y = y+1 \\
  &\quad \quad d = d + (x_1 - x_0) + (y_0 - y_1) \\
  &\quad \text{else} \\
  &\quad \quad d = d + (y_0 - y_1)
  \end{align*}
  \]

**Scan conversion**

- **Problem**
  - How to generate filled polygons (by determining which pixel positions are inside the polygon)
  - Conversion from continuous to discrete domain

- **Concepts**
  - Spatial coherence
  - Span coherence
  - Edge coherence
Scanning Rectangles

for ( y from y0 to yn )
  for ( x from x0 to xn )
    Write Pixel (x, y, val)

Scanning Rectangles (2)

for ( y from y0 to yn )
  for ( x from x0 to xn )
    Write Pixel (x, y, val)
Scanning Rectangles (3)

for ( y from y0 to yn )
  for ( x from x0 to xn )
    Write Pixel (x, y, val)

Scanning Arbitrary Polygons

• vertices:
  (4, 1), (7, 13), (11, 2)
Scanning Arbitrary Polygons (2)

- vertices: (4, 1), (7, 13), (11, 2)
- Intersect scanline w/pgon edges => span extrema

Scanning Arbitrary Polygons (3)

- vertices: (4, 1), (7, 13), (11, 2)
- Intersect scanline w/pgon edges => span extrema
- Fill between pairs of span extrema
Scanning Arbitrary Polygons (4)

- vertices: 
  (4, 1), (7, 13), (11, 2)

For each nonempty scanline
  Intersect scanline w/pgon edges => span extrema
  Fill between pairs of span extrema

Example Cases (2)

4 intersections w/ scanline 6 at x = 1, 6, 6, 12 1/7
Example Cases (3)

- 3 intersections w/sanline 5 at x = 1, 1, 11 5/7

Example Cases (4)

3 intersections w/scanline 5 at x = 1, 1, 11 5/7

=>

Count continuing edges once (shorten lower edge) now x=1, 11 5/7
Example Cases (5)

4 intersections w/ scanline 1 at x = 5, 5, 10, 10

Example Cases (6)

4 intersections w/ scanline 1 at x = 5, 5, 10, 10

=>

Don't count vertices of horizontal edges.
Now x = 5, 10
**Scanline Data Structures**

**Sorted edge table:**
- all edges
- sorted by min y
- holds:
  - max y
  - init x
  - inverse slope

**Active edge table:**
- edges intersecting current scanline
- holds:
  - max y
  - current x
  - inverse slope

**Scanline Algorithm**

1. Bucket sort edges into sorted edge table
2. Initialize y & active edge table
   - \( y = \text{first non-empty scanline} \)
   - \( \text{AET} = \text{SET}[y] \)
3. Repeat until AET and SET are empty
   - Fill pixels between pairs of x intercepts in AET
   - Remove exhausted edges
   - \( Y++ \)
   - Update x intercepts
   - Resort table (AET)
   - Add entering edges
Example: vertices (4,1), (1,11), (9,5), (12,8), (12,1)

bucket sort edges into sorted edge table

sort on minY: 1
store:
  max Y: 11
  min X: 4
1/m : (Xmax - Xmin) / (Ymax - Ymin) = (1 - 4) / (11 - 1) = -3 / 10
Example: vertices (4,1), (1,11), (9,5), (12,8), (12,1)

bucket sort edges into sorted edge table
initialize active edge list to first non empty scanline
**Example:** vertices (4,1), (1,11), (9,5), (12,8), (12,1)

- Bucket sort edges into sorted edge table
- Initialize active edge list to first non-empty scanline
- For each non-empty scanline:
  - Fill between pairs \((x=4,12)\)
  - Remove exhausted edges
  - Update intersection points
  - Resort table
  - Add entering edges
**Example:** vertices (4,1), (1,11), (9,5), (12,8), (12,1)

bucket sort edges into sorted edge table
initialize active edge list to first non empty scanline
for each non empty scanline
  fill between pairs (x=3 1/10, 12)
  remove exhausted edges
  update intersection points

**Example:** vertices (4,1), (1,11), (9,5), (12,8), (12,1)

bucket sort edges into sorted edge table
initialize active edge list to first non empty scanline
for each non empty scanline
  fill between pairs (x=3 1/10, 12)
  remove exhausted edges
  update intersection points
  resort table
  add entering edges
Example: \( \text{vertices (4,1), (1,11), (9,5), (12,8), (12,1)} \)

bucket sort edges into sorted edge table
initialize active edge list to first non empty scanline
for each non empty scanline
  fill between pairs \((x = 2 \frac{8}{10}, 9; 9,12)\)
  remove exhausted edges
  update intersection points
  resort table
  add entering edges

Example: \( \text{vertices (4,1), (1,11), (9,5), (12,8), (12,1)} \)

bucket sort edges into sorted edge table
initialize active edge list to first non empty scanline
for each non empty scanline
  fill between pairs \((x = 2 \frac{5}{10}, 7 \frac{2}{3}; 10,12)\)
  remove exhausted edges
  update intersection points
  resort table
  add entering edges
Fill between pairs:

```c
for ( x = x1; x < x2; x++ )
    framebuffer [ x, y ] = c
```

### Fill Variants (2)

- **Pattern Fill**

Fill between pairs:

```c
for ( x = x1; x < x2; x++ )
    if ( ( x + y ) % 2 )
        framebuffer [ x, y ] = c1
    else
        framebuffer [ x, y ] = c2
```
Fill Variants (3)

- Colorwash
  Red to blue

Fill between pairs:

```c
for ( x = x1; x < x2; x++ )
  framebuffer[ x, y ] = C0 + dC * ( x1 - x )
```

For efficiency carry C and dC in AET and calculate color incrementally.

---

Fill Variants (4)

- Vertex colors
  Red, green, blue

Fill between pairs:

```c
for ( x = x1; x < x2; x++ )
  framebuffer[ x, y ] = Cy1x1 + [(x - x1)/(x2 - x1)*(Cy1x2 - Cy1x1)]/dCx
```

For efficiency carry Cy and dCy in AET calculate dCx at beginning of scanline.
Barycentric Coordinates

- Use non-orthogonal coordinates to describe position relative to vertices

\[ p = a + \beta (b - a) + \gamma (c - a) \]

\[ p(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c \]

- Coordinates correspond to scaled signed distance from lines through pairs of vertices.
Barycentric Coordinates

- Computing coordinates

\[
\gamma = \frac{(y_a - y_b)x + (x_b - x_a)y + x_a y_b - x_b y_a}{(y_a - y_b)x_c + (x_b - x_a)y_c + x_a y_b - x_b y_a}
\]

\[
\beta = \frac{(y_a - y_c)x + (x_c - x_a)y + x_a y_c - x_c y_a}{(y_a - y_c)x_b + (x_c - x_a)y_b + x_a y_c - x_c y_a}
\]

\[\alpha = 1 - \beta - \gamma\]

Alternative Computation

\[
b_i = \frac{a_i}{a_i + a_j + a_k}
\]

\[
b_j = \frac{a_i}{a_i + a_j + a_k}
\]

\[
b_k = \frac{a_k}{a_i + a_j + a_k}
\]
Barycentric Rasterization

For all x do
   For all y do
      Compute \((\alpha, \beta, \gamma)\) for \((x,y)\)
      If \((\alpha \in [0,1] \text{ and } \beta \in [0,1] \text{ and } \gamma \in [0,1])\) then
         \[ c = \alpha c_0 + \beta c_1 + \gamma c_2 \]
         Draw pixel \((x,y)\) with color \(c\)

Barycentric Rasterization

\[ x_{\text{min}} = \text{floor}(x) \]
\[ x_{\text{max}} = \text{ceiling}(x) \]
\[ y_{\text{min}} = \text{floor}(y) \]
\[ y_{\text{max}} = \text{ceiling}(y) \]
for \(y = y_{\text{min}} \text{ to } y_{\text{max}}\) do
   for \(x = x_{\text{min}} \text{ to } x_{\text{max}}\) do
      \[ \alpha = f_{12}(x,y)/f_{12}(x_0,y_0) \]
      \[ \beta = f_{20}(x,y)/f_{20}(x_1,y_1) \]
      \[ \gamma = f_{01}(x,y)/f_{01}(x_2,y_2) \]
      If \((\alpha \in [0,1] \text{ and } \beta \in [0,1] \text{ and } \gamma \in [0,1])\) then
         \[ c = \alpha c_0 + \beta c_1 + \gamma c_2 \]
         Draw pixel \((x,y)\) with color \(c\)
Barycentric Rasterization

• Computing coordinates

\[ \gamma = \frac{f_{01}(x, y)}{f_{01}(x_2, y_2)} = \frac{(y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0}{(y_0 - y_1)x_2 + (x_1 - x_0)y_2 + x_0y_1 - x_1y_0} \]

\[ \beta = \frac{f_{20}(x, y)}{f_{20}(x_1, y_1)} = \frac{(y_2 - y_0)x + (x_0 - x_2)y + x_2y_0 - x_0y_2}{(y_2 - y_0)x_1 + (x_0 - x_2)y_1 + x_2y_0 - x_0y_2} \]

\[ \alpha = \frac{f_{12}(x, y)}{f_{12}(x_0, y_0)} = \frac{(y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1}{(y_1 - y_2)x_0 + (x_2 - x_1)y_0 + x_1y_2 - x_2y_1} \]

Visibility

• We can convert simple primitives to pixels/fragments
• How do we know which primitives (or which parts of primitives) should be visible?
Back-face Culling

- Polygon is back-facing if
  - $V \cdot N > 0$
- Assuming view is along Z ($V=0,0,1$)
  - $V \cdot N + (0 + 0 + z_n)$
- Simplifying further
  - If $z_n > 0$, then cull
- Works for non-overlapping convex polyhedra
- With concave polyhedra, some hidden surfaces will not be culled

Painter’s Algorithm

- First polygon:
  - (6,3,10), (11, 5,10), (2,2,10)
- Second polygon:
  - (1,2,8), (12,2,8), (12,6,8), (1,6,8)
- Third polygon:
  - (6,5,5), (14,5,5), (14,10,5), (6,10,5)
**Painter’s Algorithm**

- **Given**
  - List of polygons \( \{P_1, P_2, \ldots, P_n\} \)
  - An array of Intensity \([x, y]\)

- **Begin**
  - Sort polygon list on minimum \(z\) (largest \(z\)-value comes first in sorted list)
  - For each polygon \(P\) in selected list do
    - For each pixel \((x, y)\) that intersects \(P\) do
      - Intensity\([x, y]\) = intensity of \(P\) at \((x, y)\)
  - Display Intensity array

**Painter’s Algorithm: Cycles**

- Which order to scan?
- Split along line, then scan 1, 2, 3
Painter’s Algorithm: Cycles

• Which to scan first?
• Split along line, then scan 1,2,3,4 (or split another polygon and scan accordingly)
• Moral: Painter’s algorithm is fast and easy, except for detecting and splitting cycles and other ambiguities

Depth-sort: Overlapping Surfaces

• Assume you have sorted by maximum Z
  – Then if $Z_{\text{min}} > Z_{\text{max}}$, the surfaces do not overlap each other (minimax test)
• Correct order of overlapping surfaces may be ambiguous. Check it.
**Depth-sort: Overlapping Surfaces**

- No problem: paint $S$, then $S'$

- Problem: painting in either order gives incorrect result

- Problem? Naïve order $S$ $S'$ $S''$; correct order $S'$ $S''$ $S$

**Depth-sort: Order Ambiguity**

1. Bounding rectangles in $xy$ plane do not overlap
   - Check overlap in $x$
     \[ x_{\text{min}}' > x_{\text{max}} \text{ or } x_{\text{min}} > x_{\text{max}}' \Rightarrow \text{no overlap} \]
   - Check overlap in $y$
     \[ y_{\text{min}}' > y_{\text{max}} \text{ or } y_{\text{min}} > y_{\text{max}}' \Rightarrow \text{no overlap} \]

2. Surface $S$ is completely behind $S'$ relative to viewing direction.
   - Substitute all vertices of $S$ into plane equation for $S'$, if all are “inside” ($< 0$), then there is no ambiguity
3. Surface $S'$ is completely in front $S$ relative to viewing direction.
   - Substitute all vertices of $S'$ into plane equation for $S$, if all are “outside” ($>0$),
     then there is no ambiguity

4. Projection of the two surfaces onto the viewing plane do not overlap
   - Test edges for intersection
   - Rule out some pairs with minimax tests (can eliminate 3-4 intersection, but not 1-2)
   - Check slopes -- parallel lines do not intersect
   - Compute intersection points:
     - $s = [(x'_1 - x'_2)(y_1 - y'_1) - (x_1 - x'_1)(y'_1 - y'_2)]/D$
     - $t = [(x_1 - x_2)(y_1 - y'_1) - (x_1 - x'_1)(y_1 - y_2)]/D$
     - $D = (x'_1 - x'_2)(y_1 - y_2) - (x_1 - x_2)(y'_1 - y'_2)$
Z-Buffer

- First polygon
  - (1, 1, 5), (7, 7, 5), (1, 7, 5)
  - scan it in with depth
- Second polygon
  - (3, 5, 9), (10, 5, 9), (10, 9, 9), (3, 9, 9)
- Third polygon
  - (2, 6, 3), (2, 3, 8), (7, 3, 3)

Z-Buffer Algorithm

- Originally Cook, Carpenter, Catmull
- Given
  - List of polygons \( \{P_1, P_2, \ldots, P_n\} \)
  - An array \( x\text{-buffer}[x,y] \) initialized to +infinity
  - An array \( \text{Intensity}[x,y] \)
- Begin
  - For each polygon \( P \) in selected list do
    - For each pixel \((x,y)\) that intersects \( P \) do
      - Calculate z-depth of \( P \) at \((x,y)\)
      - If z-depth < \( z\text{-buffer}[x,y] \) then
        - \( \text{Intensity}[x,y] = \) intensity of \( P \) at \((x,y)\)
        - \( z\text{-buffer}[x,y] = \) z-depth
    - Display \( \text{Intensity} \) array
Z-Buffer: Calculating Z-depth

• From plane equation, depth at position (x,y):
  \[ z = \frac{-Ax - By - D}{C} \]

• Incrementally across scanline (x+1, y)
  \[ z' = \frac{-A(x+1) - By - D}{C} \]
  \[ = \frac{-Ax - By - D}{C} - \frac{A}{C} \]
  \[ = z - \frac{A}{C} \]

• Incrementally between scanlines (x’, y+1)
  \[ z' = \frac{-A(x') - B(y+1) - D}{C} \]
  \[ = z - \frac{A}{m} + \frac{B}{C} \]

Z-Buffer Characteristics

• Good
  – Easy to implement
  – Requires no sorting of surfaces
  – Easy to put in hardware

• Bad
  – Requires lots of memory (about 9MB for 1280x1024 display)
  – Can alias badly (only one sample per pixel)
  – Cannot handle transparent surfaces
A-Buffer Method

- Basically z-buffer with additional memory to consider contribution of multiple surfaces to a pixel
- Need to store
  - Color (rgb triple)
  - Opacity
  - Depth
  - Percent area covered
  - Surface ID
  - Misc rendering parameters
  - Pointer to next

Taxonomy of Visibility Algorithms

- Ivan Sutherland -- A Characterization of Ten Hidden Surface Algorithms
- Basic design choices
  - Space for operations
    - Object
    - Image
  - Object space
    - Loop over objects
    - Decide the visibility of each
  - Timing of object sort
    - Sort-first
    - Sort-last
**Taxonomy of Visibility Algorithms**

- **Image space**
  - Loop over pixels
  - Decide what’s visible at each
- **Timing of sort at pixel**
  - Sort first
  - Sort last
  - Subdivide to simplify

---

**Scanline Algorithm**

- Simply problem by considering only one scanline at a time
- intersection of 3D scene with plane through scanline
Scanline Algorithm

- Consider xz slice

- Calculate where visibility can change

- Decide visibility in each span

Scanline Algorithm

1. Sort polygons into sorted surface table (SST) based on $Y$
2. Initialize $Y$ and active surface table (AST)
   $Y = \text{first nonempty scanline}$
   $\text{AST} = \text{SST}[y]$
3. Repeat until AST and SST are empty
   Identify spans for this scanline (sorted on $x$)
   For each span
      determine visible element (based on $z$)
      fill pixel intensities with values from element
   Update AST
      remove exhausted polygons
      $y++$
      update x intercepts
      resort AST on $x$
      add entering polygons
4. Display Intensity array
Scanline Visibility Algorithm

- Scanline $\alpha$
  - AST: $ABC$
  - Spans
    - $0 \rightarrow x_1$ background
    - $x_1 \rightarrow x_2$ $ABC$
    - $x_2 \rightarrow \text{max}$ background

- Scanline $\beta$
  - AST: $ABC, DEF$
  - Spans
    - $0 \rightarrow x_1$ background
    - $x_1 \rightarrow x_2$ $ABC$
    - $x_2 \rightarrow x_3$ background
    - $x_3 \rightarrow x_4$ $DEF$
    - $x_4 \rightarrow \text{max}$ background
Scanline Visibility Algorithm

- Scanline $\gamma$
  - AST: $ABC\, DEF$
  - Spans
    - $0 \rightarrow x_1$ background
    - $x_1 \rightarrow x_2$ ABC
    - $x_2 \rightarrow x_3$ DEF
    - $x_3 \rightarrow x_4$ DEF
    - $x_4 \rightarrow \text{max}$ background

Scanline Visibility Algorithm

- Scanline $\gamma + 1$
  - Spans
    - $0 \rightarrow x_1$ background
    - $x_1 \rightarrow x_2$ ABC
    - $x_2 \rightarrow x_3$ DEF
    - $x_3 \rightarrow x_4$ DEF
    - $x_4 \rightarrow \text{max}$ background

- Scanline $\gamma + 2$
  - Spans
    - $0 \rightarrow x_1$ background
    - $x_1 \rightarrow x_2$ ABC
    - $x_2 \rightarrow x_3$ background
    - $x_3 \rightarrow x_4$ DEF
    - $x_4 \rightarrow \text{max}$ background
Characteristics of Scanline Algorithm

• **Good**
  – Little memory required
  – Can generate scanlines as required
  – Can antialias within scanline
  – Fast
    • Simplification of problem simplifies geometry
    • Can exploit coherence

• **Bad**
  – Fairly complicated to implement
  – Difficult to antialias between scanlines

Taxonomy Revisted

• Another dimension
  – Point-sampling
  – continuous
BSP Tree: Building the Tree

BSPTree MakeBSP ( Polygon list ) {
    if ( list is empty ) return null
    else {
        root = some polygon ; remove it from the list
        backlist = frontlist = null
        for ( each remaining polygon in the list ) {
            if ( p in front of root )
                addToList ( p, frontlist )
            else if ( p in back of root )
                addToList ( p, backlist )
            else {
                splitPolygon (p,root,frontpart,backpart)
                addToList ( frontpart, frontlist )
                addToList ( backpart, backlist )
            }
        }
        return (combineTree(MakeBSP(frontlist),root,
                              MakeBSP(backlist)))
    }
}
Building a BSP Tree

• Use pgon 3 as root, split on its plane
• Pgon 5 split into 5a and 5b

Building a BSP Tree

• Split left subtree at pgon 2
Building a BSP Tree

• Split right subtree at pgon 4

Building a BSP Tree

• Alternate tree if splits are made at 5, 4, 3, 1
BSP Tree: Displaying the Tree

DisplayBSP ( tree )
{
    if ( tree not empty ) {
        if ( viewer in front of root ) {
            DisplayBSP ( tree -> back )
            DisplayPolygon ( tree -> root )
            DisplayBSP ( tree -> front )
        }
        else {
            DisplayBSP ( tree -> front )
            DisplayPolygon ( tree -> root )
            DisplayBSP ( tree -> back )
        }
    }
}
BSP Tree Display

For view point at C
   at 3: viewpoint on front -> display back first
   at 4: viewpoint on back -> display front first

BSP Tree Display

For view point at C
   at 3: viewpoint on front -> display back first
   at 4: viewpoint on back -> display front first (none)
       display self
       display back
BSP Tree
Display

For view point at C
at 3 : viewpoint on front -> display back first
   at 4 : viewpoint on back -> display front first (none)
      display self
      display back
at 5b : viewpoint on back -> display front (none)
      display self
      display back (none)

BSP Tree
Display

For view point at C
at 3 : viewpoint on front -> display back first
   at 4 : viewpoint on back -> display front first (none)
      display self
      display back
at 5b : viewpoint on back -> display front
      display self
      display back (none)
BSP Tree Display

For viewpoint at C
at 3 : viewpoint on front -> display back first
  at 4 : viewpoint on back -> display front first (none)
    display self
    display back
  at 5b : viewpoint on back -> display front
    display self
    display back (none)
  display self
  display front

at 2 : viewpoint on back -> display front first
BSP Tree Display

For viewpoint at C
at 3: viewpoint on front -> display back first
  at 4: viewpoint on back -> display front first (none)
    display self
    display back
      at 5b: viewpoint on back -> display front
        display self
        display back (none)

  display self
  display front

at 2: viewpoint on back -> display front first
at 5a: viewpoint on back -> display front (none)
  display self
  display back (none)
Shading Revisited

- Illumination models compute appearance at a location
- How do you efficiently fill areas?

Diffuse Shading Models

- Flat shading
- Gouraud shading
Flat Shading Algorithm

For each visible polygon
  Evaluate illumination with polygon normal
For each scanline
  For each pixel on scanline
    Fill with calculated intensity

Interpolated Shading Algorithm

For each visible polygon
  For each vertex
    Evaluate illumination with vertex normals
For each scanline
  Interpolate intensity along edges (for span extrema)
  For each pixel on scanline
    Interpolate intensity from extrema
• The normal vector at vertex V is calculated as the average of the surface normals for each polygon sharing that vertex.

Gouraud Calculations

1. Calculate intensity at vertices ($I_1, I_2, I_3$)
2. Interpolate vertex intensities along edges ($I_a, I_b$)
3. Interpolate intensities at span extrema to pixels ($I_p$)
Barycentric Rasterization

\[
x_{\min} = \text{floor}(x_i)
\]
\[
x_{\max} = \text{ceiling}(x_i)
\]
\[
y_{\min} = \text{floor}(y_i)
\]
\[
y_{\max} = \text{ceiling}(y_i)
\]

for \( y = y_{\min} \) to \( y_{\max} \) do
  for \( x = x_{\min} \) to \( x_{\max} \) do
    \[
    \alpha = \frac{f_{12}(x,y)}{f_{12}(x_0,y_0)}
    \]
    \[
    \beta = \frac{f_{20}(x,y)}{f_{20}(x_1,y_1)}
    \]
    \[
    \gamma = \frac{f_{01}(x,y)}{f_{01}(x_2,y_2)}
    \]
    If \( \alpha \in [0,1] \) and \( \beta \in [0,1] \) and \( \gamma \in [0,1] \) then
      \[
      c_0 = \text{evaluate_illumination}(x_0,y_0,z_0)
      \]
      \[
      c_1 = \text{evaluate_illumination}(x_1,y_1,z_1)
      \]
      \[
      c_2 = \text{evaluate_illumination}(x_2,y_2,z_2)
      \]
      \[
      c = \alpha c_0 + \beta c_1 + \gamma c_2
      \]
    Draw pixel \((x,y)\) with color \(c\)

Problems with Interpolated Shading

- Polygon silhouette
- Perspective distortion
- Orientation dependence
- Problems at shared vertices
- Unrepresentative vertex normals
Phong Shading

• Ideally: shade from normals of curved surface
• Approximate with normals interpolated between vertex normals

\[ N_a = \frac{|P_a - P_0|}{|P_1 - P_0|} N_1 + \frac{|P_1 - P_a|}{|P_1 - P_0|} N_0 \]

Phong Algorithm

• For each visible polygon
  – For each scanline
    • Calculate normals at edge intersections (span extrema) by linear interpolation
    • For each pixel on scanline
      — Calculate normal by interpolation of normals at span extrema
      — Evaluate illumination model with that normal
Barycentric Rasterization

\[ x_{\text{min}} = \text{floor}(x_i) \]
\[ x_{\text{max}} = \text{ceiling}(x_i) \]
\[ y_{\text{min}} = \text{floor}(y_i) \]
\[ y_{\text{max}} = \text{ceiling}(y_i) \]

for \( y = y_{\text{min}} \) to \( y_{\text{max}} \) do
  for \( x = x_{\text{min}} \) to \( x_{\text{max}} \) do
    \[ \alpha = \frac{f_{12}(x,y)}{f_{12}(x_0,y_0)} \]
    \[ \beta = \frac{f_{20}(x,y)}{f_{20}(x_1,y_1)} \]
    \[ \gamma = \frac{f_{01}(x,y)}{f_{01}(x_2,y_2)} \]
    If \( \alpha \in [0,1] \) and \( \beta \in [0,1] \) and \( \gamma \in [0,1] \) then
      \( n = \alpha n_0 + \beta n_1 + \gamma n_2 \)
      Normalize \( n \)
      \( c = \text{evaluate_illumination}(x,y,n) \)
      Draw pixel \((x,y)\) with color \( c\)

Artistic Illumination

- Concept: intentionally mimic artistic effects which may not match photorealism (NPR)
- Examples
  - Line drawing
  - Shading effects
    - Cool-warm (tone shading)
    - Toon
  - Media Emulation
Silhouette Drawing

- Want to draw silhouette edge to emphasize shape
- Silhouette defined by points where surface normal is orthogonal to view vector
  \[ V \cdot N = 0 \]
- Implementation for polygonal meshes: draw edge when polygons change from forward to back
  \[ (V \cdot N_0)(V \cdot N_1) \leq 0 \]
  Draw silhouette (edge between polygons)
- Add sharp creases
  \[ (N_0 \cdot N_1) \leq \text{threshold} \]
  Draw silhouette (edge between polygons)

Diffuse Only

\[ K_d = 1, \, k_a = 0 \]

Gooch ‘98
Highlights and Edges

Phong Shading and Edges

Kd=0.5
Ka=0.1

Gooch 98
Tone Shading Model

\[ I = \left( \frac{1 + \hat{i} \cdot \hat{n}}{2} \right) k_{\text{cool}} + \left( 1 - \frac{1 + \hat{i} \cdot \hat{n}}{2} \right) k_{\text{warm}} \]

with

\[ k_{\text{cool}} = k_{\text{blue}} + \alpha k_{d} \]
\[ k_{\text{warm}} = k_{\text{yellow}} + \beta k_{d} \]

Mixing Tone and Color

- Pure blue to yellow: darken
- Pure black to object color: select

\[ + \]

\[ = \]

Final tone
Constant Luminance Tone

Luminance/Tone Rendering

B=0.4, y=0.4
α=.2, β = .6
Luminance/Tone Rendering

\[ B=0.55, \ g=0.8 \]
\[ \alpha = .25, \ \beta = .5 \]

Hue/Tone Interactions

- Gooch 98