3D Polygon Rendering Pipeline

Scan Conversion & Shading

Thomas Funkhouser
Princeton University
COS 426, Fall 1999
3D Rendering Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

3D Modeling Coordinates

Lighting

3D World Coordinates

Viewing Transformation

3D World Coordinates

Projection Transformation

3D Camera Coordinates

Clipping

2D Screen Coordinates

Viewport Transformation

2D Screen Coordinates

Scan Conversion

2D Image Coordinates

Scan Conversion & Shading
Overview

• Scan conversion
  ◦ Figure out which pixels to fill

• Shading
  ◦ Determine a color for each filled pixel

• Texture Mapping
  ◦ Describe shading variation within polygon interiors

• Visible Surface Determination
  ◦ Figure out which surface is front-most at every pixel
Scan Conversion

- 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
Scan Conversion

- 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

![Diagram of triangle with vertices P1, P2, and P3]
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

```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

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

• What is bad about this algorithm?

```cpp
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
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 \( x = x_L \); \( x \leq x_R \); \( x++ \))
                SetPixel(\( x \), \( y \), rgba);
            \( x_L \) += \( dx_L/dy_L \);
            \( x_R \) += \( dx_R/dy_R \);
        }
    }
}
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
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
Polygon Sweep-Line Algorithm

- Incremental algorithm to find spans, and determine insideness with odd parity rule
  - Takes advantage of scanline coherence
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);
    }
}
Hardware Scan Conversion

- Convert everything into triangles
  - Scan convert the triangles
Hardware Antialiasing

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

• Scan conversion
  ◦ Figure out which pixels to fill

• Shading
  ◦ Determine a color for each filled pixel

• Texture Mapping
  ◦ Describe shading variation within polygon interiors

• Visible Surface Determination
  ◦ Figure out which surface is front-most at every pixel
Shading

• How do we choose a color for each filled pixel?
  ◦ Each illumination calculation for a ray from the eyepoint through the view 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 Casting

- Simplest shading approach is to perform independent lighting calculation for every pixel
  - When is this unnecessary?

\[
I = I_E + K_A I_{AL} + \sum_i (K_D (N \cdot L_i) I_i + K_S (V \cdot R_i)^n I_i)
\]
Polygon Shading

- Can take advantage of spatial coherence
  - Illumination calculations for pixels covered by same primitive are related to each other

\[ I = I_E + K_A I_{AL} + \sum_i (K_D (N \cdot L_i) I_i + K_S (V \cdot R_i)^n I_i) \]
Polygon Shading Algorithms

- Flat Shading
- Gouraud Shading
- Phong Shading
Flat Shading

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

\[ I = I_E + K_A I_{AL} + \sum_i (K_D (N \cdot L_i) I_i + K_S (V \cdot R_i)^n I_i) \]
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 ones with smooth surfaces
Polygon Shading Algorithms

- Flat Shading
- **Gouraud Shading**
- Phong Shading
Gouraud Shading

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

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

\[ A = \alpha I_1 + (1-\alpha)I_3 \]

\[ B = \beta I_2 + (1-\beta)I_3 \]

\[ I = \phi A + (1-\phi)B \]
Gouraud Shading

- Smooth shading over adjacent polygons
  - Curved surfaces
  - Illumination highlights
  - Soft shadows

Mesh with shared normals at vertices
Gouraud Shading

- Produces smoothly shaded polygonal mesh
  - Piecewise linear approximation
  - Need fine mesh to capture subtle lighting effects
Gouraud Shading

- Produces smoothly shaded polygonal mesh
  - Piecewise linear approximation
  - Need fine mesh to capture subtle lighting effects

![Illustration: Poor behavior of specular light](image1)

![Illustration: Same sphere – high polygon count](image2)
Polygon Shading Algorithms

- Flat Shading
- Gouraud Shading
- Phong Shading
Phong Shading

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

\[ I = I_E + K_A I_{AL} + \sum_i (K_D (N \cdot L_i) I_i + K_S (V \cdot R_i)^n I_i) \]
Phong Shading

- 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

\[ A = \alpha N_1 + (1-\alpha)N_3 \]
\[ B = \beta N_2 + (1-\beta)N_3 \]
\[ I = \varphi A + (1-\varphi)B \]
Polygon Shading Algorithms

Wireframe

Flat

Gouraud

Phong
Shading Issues

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

• Scan conversion
  ◦ Figure out which pixels to fill

• Shading
  ◦ Determine a color for each filled pixel

• Texture Mapping
  ◦ Describe shading variation within polygon interiors

• Visible Surface Determination
  ◦ Figure out which surface is front-most at every pixel
Textures

- Describe color variation in interior of 3D polygon
  - When scan converting a polygon, vary pixel colors according to values fetched from a texture
Surface Textures

- Add visual detail to surfaces of 3D objects

Polygonal model

With surface texture
Surface Textures
3D Rendering Pipeline (for direct illumination)

3D Primitives

- 3D Modeling Coordinates

Modeling Transformation

- 3D World Coordinates

Lighting

- 3D World Coordinates

Viewing Transformation

- 3D Camera Coordinates

Projection Transformation

- 2D Screen Coordinates

Clipping

- 2D Screen Coordinates

Viewport Transformation

- 2D Image Coordinates

Scan Conversion

- 2D Image Coordinates

Image

Texture mapping
Overview

• Texture mapping methods
  ◦ Mapping
  ◦ Filtering
  ◦ Parameterization

• Texture mapping applications
  ◦ Modulation textures
  ◦ Illumination mapping
  ◦ Bump mapping
  ◦ Environment mapping
  ◦ Image-based rendering
  ◦ Non-photorealistic rendering
Texture Mapping

- **Steps:**
  - Define texture
  - Specify mapping from texture to surface
  - Lookup texture values during scan conversion

---

**Coordinate Systems**
- **Texture Coordinate System**
- **Modeling Coordinate System**
- **Image Coordinate System**
Texture Mapping

• When scan convert, map from …
  - image coordinate system \((x,y)\) to
  - modeling coordinate system \((u,v)\) to
  - texture image \((t,s)\)

![Diagram of Texture Mapping](https://via.placeholder.com/150x150)

- Texture Coordinate System
- Modeling Coordinate System
- Image Coordinate System
Texture Mapping

• Texture mapping is a 2D projective transformation
  ○ texture coordinate system: (t,s) to
  ○ image coordinate system (x,y)
Texture Mapping

- Scan conversion
  - Interpolate texture coordinates down/across scan lines
  - Distortion due to bilinear interpolation approximation
    - Cut polygons into smaller ones, or
    - Perspective divide at each pixel
Texture Mapping

Linear interpolation of texture coordinates

Correct interpolation with perspective divide

Hill Figure 8.42
Overview

• Texture mapping methods
  ◦ Mapping
  ◦ Filtering
  ◦ Parameterization

• Texture mapping applications
  ◦ Modulation textures
  ◦ Illumination mapping
  ◦ Bump mapping
  ◦ Environment mapping
  ◦ Image-based rendering
  ◦ Non-photorealistic rendering
Texture Filtering

- Must sample texture to determine color at each pixel in image
Texture Filtering

• Aliasing is a problem

Point sampling

Area filtering

Angel Figure 9.5
Texture Filtering

• Ideally, use elliptically shaped convolution filters

In practice, use rectangles
Texture Filtering

- Size of filter depends on projective warp
  - Can prefiltering images
    - Mip maps
    - Summed area tables

Angel Figure 9.14
Mip Maps

• Keep textures prefiltered at multiple resolutions
  ◦ For each pixel, linearly interpolate between two closest levels (e.g., trilinear filtering)
  ◦ Fast, easy for hardware
Summed-area tables

- At each texel keep sum of all values down & right
  - To compute sum of all values within a rectangle simply subtract two entries
  - Better ability to capture very oblique projections
  - But, cannot store values in a single byte
Overview

• Texture mapping methods
  ◦ Mapping
  ◦ Filtering
  ◦ Parameterization

• Texture mapping applications
  ◦ Modulation textures
  ◦ Illumination mapping
  ◦ Bump mapping
  ◦ Environment mapping
  ◦ Image-based rendering
  ◦ Non-photorealistic rendering
Modulation textures

- Map texture values to scale factor

\[
I = T(s,t)(I_E + K_A I_A + \sum_L (K_D (N \cdot L) + K_S (V \cdot R)^n) S_L I_L + K_T I_T + K_S I_S)
\]
Illumination Mapping

- Map texture values to surface material parameter
  - $K_A$
  - $K_D$
  - $K_S$
  - $K_T$
  - $n$

$I = I_E + K_A I_A + \sum_L (K_D (N \cdot L) + K_S (V \cdot R)^n) S_L I_L + K_T I_T + K_S I_S$
Bump Mapping

• Map texture values to perturbations of surface normals
Bump Mapping
Environment Mapping

- Map texture values to perturbations of surface normals
Image-Based Rendering

• Map photographic textures to provide details for coarsely detailed polygonal model
Solid Textures

- Texture values indexed by 3D location
  - Expensive storage, or
  - Compute on the fly, E.g Perlin noise
Solid Textures

- Texture values indexed by 3D location
  - Expensive storage, or
  - Compute on the fly, E.g Perlin noise