Scan Conversion & Shading
Tom Funkhouser
Princeton University
COS 426, Spring 2006

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

Scan Conversion

- 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

```cpp
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

- 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

```cpp
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;
}
```

Inside Triangle Test

- What is bad about this algorithm?
Inside Polygon Rule

- What is a good rule for which pixels are inside?
  - Odd-parity rule
    - Any ray from P to infinity crosses odd number of edges

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

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?

Convex Polygon

Concave Polygon
**Polygon Sweep-Line Algorithm**

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

**Hardware Scan Conversion**

- Convert everything into triangles
  - Scan convert the triangles

**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

**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?

**Ray Casting**

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

\[ I = I_L + K_d I_d + \sum (K_g (N \cdot L) I_L + K_s (V \cdot R)^k I_L) \]
**Polygon Shading**

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

\[ I = I_d + K_s I_{sh} + \sum (K_p (N \cdot L_i) I_s + K_s (V \cdot R_i)^n I_d) \]

**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

\[ I = I_d + K_s I_{sh} + \sum (K_p (N \cdot L_i) I_s + K_s (V \cdot R_i)^n I_d) \]

**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
**Polygon Shading Algorithms**

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

**Gouraud Shading**

- Method 1: One lighting calculation per vertex
  - Assign pixels inside polygon by interpolating colors computed at vertices

**Gouraud Shading**

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

$$I = I_e + K_d I_d + \sum (K_p (N \cdot L) I_d + K_v (V \cdot R) I_d)$$

**Gouraud Shading**

- Bilinearly interpolate colors at vertices down and across scan lines

$$A = \alpha l_1 + (1-\alpha)l_3$$

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

**Gouraud Shading**

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

Mesh with shared normals at vertices

**Gouraud Shading**

- Flat Shading
- Gouraud Shading
**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?

\[ I = I_e + K_d I_d + \sum (K_g (N \cdot L) I_i + K_r (V \cdot R) I_i) \]

**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 computing shared vertex normals
  - Problems at T-vertices
Shading with Antialiasing

• How avoid aliasing at edges of polygons?

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

Textures

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

Angel Figure 9.3
Surface Textures

- Add visual detail to surfaces of 3D objects

3D Rendering Pipeline (for direct illumination)

Texture Mapping Overview

- Texture mapping methods
  - Parameterization
  - Mapping
  - Filtering

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

Parameterization

- Add visual detail to surfaces of 3D objects

Option: Varieties of projections

- Q: How do we decide where on the geometry each color from the image should go?

Texture mapping

- A: ____

geometry + image = texture map

- Q: How do we decide where on the geometry each color from the image should go?
Texture Mapping

- Texture mapping methods
  - Parameterization
  - Mapping
  - Filtering

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

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

- 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 Diagram](image)

---

**Overview**

- Texture mapping methods
  - Parameterization
  - Mapping
  - Filtering

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

![Overview Diagram](image)

---

**Texture Filtering**

- Aliasing is a problem

![Texture Filtering Diagram](image)

---

**Texture Filtering**

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

![Texture Filtering Diagram](image)

---

**Texture Filtering**

- Ideally, use elliptically shaped convolution filters

![Texture Filtering Diagram](image)
**Texture Filtering**

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

![Magnification and Minification Diagram](image)

*Angel Figure 9.14*

**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

**Modulation textures**

Map texture values to scale factor

- Wood texture

![Texture Mapping Diagram](image)

**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

**Overview**

- Texture mapping methods
  - Parameterization
  - Mapping
  - Filtering

- Texture mapping applications
  - Modulation textures
  - Illumination mapping
  - Bump mapping
  - Environment mapping
  - Image-based rendering

**Illumination Mapping**

Map texture values to surface material parameter

- $K_a$
- $K_d$
- $K_s$
- $K_f$
- $n$

$$I = T(x, y) I_d + K_a I_a + \sum (K_m (N \cdot L) + K_s (V \cdot R)) S_l I_l + K_f I_f + K_a I_a$$
**Bump Mapping**

Texture values perturb surface normals

**Environment Mapping**

Texture values are reflected off surface patch

**Solid textures**

Texture values indexed by 3D location (x,y,z)
- Expensive storage, or
- Compute on the fly, e.g. Perlin noise

**Image-Based Rendering**

Map photographic textures to provide details for coarsely detailed polygonal model

**Texture Mapping Summary**

- Texture mapping methods
  - Parameterization
  - Mapping
  - Filtering

- Texture mapping applications
  - Modulation textures
  - Illumination mapping
  - Bump mapping
  - Environment mapping
  - Image-based rendering
  - Volume textures
**Shading Languages**

- Execute arbitrary function for every pixel

**Example: Phong lighting**

```c
void main()
{
    vec3 N = normalize(normal); // Surface normal
    float NdotL = dot(N, light); // Dot product between normal and light vector
    if (NdotL > 0.0)
    {
        float intensity, at, af;
        intensity = max(0.0, 0.5 * NdotL);
        at = intensity * 0.5;
        af = intensity * 0.95;
        gl_FragColor = color; // Set fragment color
    }
    else if (intensity > 0.5)
    {
        gl_FragColor = vec4(0.4, 0.2, 0.2, 1.0); // Use another color
    }
    else if (intensity > 0.95)
    {
        gl_FragColor = vec4(1.0, 0.5, 0.5, 1.0); // Use another color
    }
}
```

**Shading Languages**

- Example: modulating texture on diffuse surface

```c
uniform sample2D tex;

void main()
{
    vec4 color;
    float NdotL, NdotHV;
    vec3 n, halfV;
    vec4 ambient = vec4(0.1, 0.6, 0.2, 1.0); // Ambient color
    vec4 diffuse = vec4(0.5, 0.5, 0.5, 1.0); // Diffuse color
    vec4 specular = vec4(0.9, 0.9, 0.9, 1.0); // Specular color
    vec3 texel = texel2D(tex, texCoord[0].st); // Sample texture
    n = normalize(normal); // Surface normal
    halfV = normalize(n); // Half-vector
    NdotL = dot(n, light); // Dot product between normal and light vector
    NdotHV = dot(halfV, light); // Dot product between half-vector and light vector
    color += diffuse * NdotL; // Add diffuse component
    color += specular * pow(NdotHV, 2); // Add specular component
    gl_FragColor = color; // Set fragment color
}
```

**Shading Languages**

- Example: toon shading

```c
uniform sample2D tex;
uniform vec3 light;

void main()
{
    vec4 color;
    float NdotL, NdotHV;
    vec3 n, halfV;
    vec4 ambient = vec4(0.1, 0.6, 0.2, 1.0); // Ambient color
    vec4 diffuse = vec4(0.5, 0.5, 0.5, 1.0); // Diffuse color
    vec4 specular = vec4(0.9, 0.9, 0.9, 1.0); // Specular color
    vec3 texel = texel2D(tex, texCoord[0].st); // Sample texture
    n = normalize(normal); // Surface normal
    halfV = normalize(n); // Half-vector
    NdotL = dot(n, light); // Dot product between normal and light vector
    NdotHV = dot(halfV, light); // Dot product between half-vector and light vector
    color += diffuse * NdotL; // Add diffuse component
    color += specular * pow(NdotHV, 2); // Add specular component
    gl_FragColor = color; // Set fragment color
}
```

**Summary**

- 2D polygon scan conversion
  - Paint pixels inside primitive
  - Sweep-line algorithm for polygons

- Polygon Shading Algorithms
  - Flat
  - Gouraud
  - Texture mapping
  - Shading languages

- Key ideas:
  - Sampling and reconstruction
  - Spatial coherence

- Less expensive

- More expensive