Shading & Texturing
Overview

- **Scan conversion**
  - Figure out which pixels to draw/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?

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

- 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
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
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

- **Step 1:** One lighting calculation per **vertex**
  - Assign pixels inside polygon by interpolating colors computed at vertices
Gouraud Shading

- Step 2: Bilinearly interpolate colors at vertices down and across scan lines

\[ A = \alpha l_1 + (1-\alpha)l_3 \]

\[ B = \beta l_2 + (1-\beta)l_3 \]

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

Flat Shading

Gouraud Shading
Gouraud Shading

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

- Poor behavior of specular light
- Same sphere – high polygon count
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.

\[ A = \alpha N_1 + (1-\alpha) N_3 \]

\[ B = \beta N_2 + (1-\beta) N_3 \]

\[ I = \phi A + (1-\phi) B \]
Guarard Vs. Phong Shading

- Guarard: calculate color in vertices and interpolate pixel colors
- Phong: interpolate normals and calculate colors at each pixel
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 draw/fill
- Shading
  - Determine a color for each filled pixel
- Texture Mapping
  - Describe shading variation within polygon interiors
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**
Surface Textures

Polygonal model + surface texture
Surface Textures

- Add visual detail to surfaces of 3D objects

Example: Google Earth
Textures

- Describe color variation in interior of 3D polygon
  - When scan converting a polygon, vary pixel colors according to values fetched from a texture, (not the vertexes color)
Overview

- Texture mapping methods
  - Mapping
  - Filtering
  - Parameterization
- Texture mapping applications
  - Modulation textures
  - Illumination mapping
  - Bump mapping
  - Environment mapping
  - Image-based rendering
Texture Mapping

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

![Texture Mapping Diagram]

Texture Coordinate System

Modeling Coordinate System
Texture Mapping

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

\[
\begin{align*}
(0,0) & \quad (1,0) \\
(0,1) & \quad (1,1)
\end{align*}
\]
Texture Mapping

- Texture mapping is in fact a 2D projective transformation
  - texture coordinate system: \((u,v)\) to
  - screen coordinate system \((t,s)\)
A simple case...

image coordinates
$u$ and $v$ range from 0 to 1

$P_{ij} = \left( r \cos \frac{2\pi i}{10}, h \frac{j}{5}, r \sin \frac{2\pi i}{10} \right)$

$u$-coordinate of vertex $P_{ij}$: $i/10$
$v$-coordinate of vertex $P_{ij}$: $j/5$
Textures

We know how to go from this… to this

J. Birn
Textures

But what about this…

?to this

J. Birn
Textures

Properties:

- Alter shading of individual pixels
- Implemented as part of shading process
- Rely on maps being stored as 1D, 2D, or 3D images
- Subject to aliasing errors
Textures

General Implementation Approach:

Associate a collection of coordinates \((s_1,\ldots,s_n)\) to every vertex \((0 \leq s_i \leq 1)\)

Use the color of the image at position \((s_1,\ldots,s_n)\) to define the color of a vertex
Another Example: Brick Wall
Another Example: Brick Wall
2D Texture

- Coordinates described by variables \( s \) and \( t \) and range over interval \((0,1)\)
- Texture elements are called texels
- Often 4 bytes (rgba) per texel
2D Texture

```c
glBegin(GL_TRIANGLES);
glTexCoord2f(0.0, 0.0);
glVertex3f(0.0, 0.0, 0.0);

glTexCoord2f(1.0, 0.0);
glVertex3f(1.0, 0.0, 0.0);

glTexCoord2f(1.0, 1.0);
glVertex3f(1.0, 1.0, 0.0);
glEnd();
```
Parameterization

Q: How do we decide *where* on the geometry each color from the image should go?
Option: Unfold/Map Entire Surface

[Piponi2000]
Option: Unfold/Map Entire Surface

- Tricky, because mapped surface may have severe distortions

- However, because texture is continuous, may be easier to think about

Gu et al. 2003
Option: Unfold/Map Entire Surface

- Tricky, because mapped surface may have severe distortions

- However, because texture is continuous, may be easier to think about

parameterize a it is impossible togeneral,In that soa simple base domainshape tocomplex areas are preservedangles andboth
Option: Atlas

charts  atlas  surface

[Sander2001]
Option: Atlas

- Less distortion on each little piece of atlas
- Need to pack patches to reduce wasted space in texture image
- May be more difficult to think about the relationships between the different pieces
Texture Mapping

- During scan conversion, we interpolate texture coordinates down/across scan lines.

- **Affine texture mapping.** Noticeable discontinuity between adjacent triangles when these triangles are at an angle to the plane of the screen. Why?
- closer to viewer, difference from pixel to pixel between texture coordinates is smaller (stretching the texture wider)
- parts farther away, difference is larger (compressing the texture)
Perspective correct: account for vertices' positions in 3D space, rather than simply interpolating a 2D triangle

Linear interpolation of texture coordinates

\[ u_\alpha = (1 - \alpha)u_0 + \alpha u_1 \]

Correct interpolation with perspective divide

\[ u_\alpha = \frac{(1 - \alpha)\frac{u_0}{z_0} + \alpha\frac{u_1}{z_1}}{(1 - \alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}} \]

Dividing each texture coordinate is like projecting it
Aliasing?

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

- Image mapped onto polygon
- **Occur when screen resolution differs from texture resolution**
- Magnification aliasing
  - Screen resolution finer than texture resolution
  - Multiple pixels per texel
- Minification aliasing
  - Screen resolution coarser than texture resolution
  - Multiple texels per pixel
Minification & Magnification

**Minification:**
More than one texel per pixel

**Magnification:**
Less than one texel per pixel
Magnification Filtering (simpler?)

- Nearest neighbor
- Linear interpolation
Aliasing Problem

Point sampling

After area filtering
Minification Filtering

- Multiple texels per pixel
- Potential for aliasing since texture signal bandwidth greater than framebuffer
- Box filtering requires averaging of texels
- Precomputation of pre-filtered images
  - MIP Mapping
  - Summed Area Tables
MIP Mapping (Lance Williams, 1983)

- “Multum In Parvo” (much in little)
- Create a resolution pyramid of textures
  - Repeatedly subsample texture at half resolution
  - Until single pixel
  - Need extra storage space
MIP Mapping

- Accessing
  - Use texture resolution closest to screen resolution
  - Or interpolate between two closest resolutions
  - Fast, easy for hardware
Summed-area Tables (Frank Crow, 1984)

- A 2D table the size of the texture.
- At entry (i,j), store the sum of all texels in the rectangle defined by (0,0) and (i,j).
- Sum of all texels is easily obtained from:
  \[
  \text{area} = A - B - C + D
  \]
Summed-area Tables

- At each entry keep sum of all values from top left to bottom right
  \[ T(i, j) = \sum_{i'<i, j'<j} I(i', j') \]
- Can be computed in one pass:
  \[ T(i, j) = I(i, j) + T(i-1, j) + T(i, j-1) - T(i-1, j-1) \]
Breakdown

\[
\begin{align*}
T_{i,j} & = I_{i,j} + T_{(i,j)} + T_{(i,j)} - T_{(i,j)} \\
\end{align*}
\]
Computing Any Rectangle Values

- Sum of all values within any rectangle?

\[
\sum I(i', j') = T(i_1, j_1) - T(i_0, j_1) - T(i_1, j_0) + T(i_0, j_0)
\]

- Better ability to capture very oblique projections
- But, cannot store values in a single byte
Breakdown

\[
T_{i_0 i_1 j_0 j_1} = T_{i_0 i_1 j_0 j_1} - \]

\[
T_{i_0 i_1 j_0 j_1} - T_{i_0 i_1 j_0 j_1} + T_{i_0 i_1 j_0 j_1}
\]
Summed Area Table (SAT)

- Determine the rectangle with the same aspect ratio as the bounding box and the same area as the pixel mapping.
Mip-maps

- Find level of the mip-map where the area of each mip-map pixel is closest to the area of the mapped pixel.
Elliptical Weighted Average (EWA) Filter

- Treat each pixel as circular, rather than square.
- Mapping of a circle is elliptical in texel space.
Mip Maps

- Keep textures prefiltred at multiple resolutions. For each pixel, use the mip-map closest level. Fast, easy for hardware.

Again: we’re trading aliasing for blurring!
Mip Maps

- Keep textures prefiltered at multiple resolutions
  - For each pixel, use the mip-map closest level
  - Fast, easy for hardware

- This type of filtering is isotropic:
  - It doesn’t take into account that there is more compression in the vertical direction than in the horizontal one

Again: we’re trading aliasing for blurring!
Summed-area tables

Key Idea:

• Approximate the summation/integration over an arbitrary region by a summation/integration over an axis-aligned rectangle.

\[
\text{Sum}(\left[ a, b \right] \times \left[ c, d \right]) = \int_a^b \int_c^d f(x, y) \, dy \, dx
\]
Summed-area tables

- Precompute the values of the integral:

\[ S(a, b) = \int_0^a \int_0^b f(x, y) \, dy \, dx \]

- Each texel is the sum of all texels below and to the left of it

<table>
<thead>
<tr>
<th>Original image</th>
<th>Summed area table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1</td>
<td>4  8  12 16</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td>3  6  9  12</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td>2  4  6  8</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td>1  2  3  4</td>
</tr>
</tbody>
</table>

Courtesy Simon Green
Summed-area tables

- Now, suppose I have some pixel *on screen* that maps to these pixels in my texture. What to do?
  - Explicitly computing the average (applying a box filter) is too slow!
Summed-area tables

- Now, suppose I have some pixel *on screen* that maps to these pixels in my texture. What to do?
  - Explicitly computing the average (applying a box filter) is too slow!
  - Use summed-area table formula

\[
\text{Sum}([0,1] \times [3,3]) = S(3,3) - S(0,3) - S(3,1) + S(0,1) \\
= 16 - 8 - 4 + 2 = 6
\]

Original image

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Summed-area table

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
Summed-area tables

- Now, suppose I have some pixel on screen that maps to these pixels in my texture. What to do?
  - Explicitly computing the average (applying a box filter) is too slow!
  - Use summed-area table formula

\[
\text{Sum}([0,1] \times [3,3]) = S(3,3) - S(0,3) - S(3,1) + S(0,1) = 16 - 8 - 4 + 2 = 6
\]

\[
\text{Average}([0,1] \times [3,3]) = \text{Sum}([0,1] \times [3,3]) / \text{Area}([0,1] \times [3,3]) = 6/6 = 1
\]
Textures as Lookup Tables

- Key Idea: anything can be parametrized using “texture”
- Instead of using direct or “calculated” value, one uses the value as an index into a lookup table

\[ f(i, j) = val(i, j) \]

\[ f(i, j) = T(val(i, j)) \]
Modulation Textures

- Map texture values to scale factor

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

- Replace model of light to one based on a texture lookup:

- Treat eye vector as a point of the unit sphere, and use it to index into a texture-mapped sphere to look up light

- Write eye vector in polar coordinates, \((\theta, \varphi)\): \(u = \theta/2\pi\), \(v = \varphi/\pi + \frac{1}{2}\)

- Environment map: A fisheye-lens photograph taken from the center of a scene
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 any material parameter

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

Map texture values to any material parameter

Specular

Diffuse

Modulation

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

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

$$K_T = T(s,t)$$

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

- An image-based lighting technique for approximating the appearance of a reflective surface by means of a precomputed texture image. The texture is used to store the image of the distant environment surrounding the rendered object.
Environment Mapping

- Generate a spherical/cubic map of the environment around the model.
Environment Mapping

- Generate a spherical/cubic map of the environment around the model.

- Texture values are reflected off surface patch
Environment Mapping

Texture values are reflected off surface patch
Environment Maps / Light Probes
Cube Maps
Bump Mapping

- Map texture values to perturbations of surface normals
Bump Mapping

- assume that at each surface point \( P \) we have a pair of unit vectors \( t_1 \) and \( t_2 \)
- Map \( R,G \) from \([-128, 127]\) to \([-1, 1]\)
- Adjust normal from bump map at \( P \): \( \tilde{n} = S(n + rt_1 + gt_2) \).
Bump Mapping

Note the silhouette is smooth ➔ no real geometry!
Bump Mapping

- Recall that many parts of our lighting calculation depend on surface normals

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

Phong shading approximates smoothly curved surface
Bump Mapping

Phong shading approximates smoothly curved surface

We can store perturbations to normals in a texture map

P. Rheingans
Bump Mapping

Phong shading approximates smoothly curved surface

Now Phong shading gives the appearance of a bumpy surface

P. Rheingans
Bump Mapping

Note that bump mapping does not change object silhouette

Siggraph.org
Interactive Shadows

- A surface area that is not "visible" by the light source but is visible by the viewer appears as a shadowed area.
Z-Buffer

A simple three dimensional scene

Z-buffer representation
Z-Buffer Algorithm

- Any pixel rendered holds a color and a distance from the viewer (z).
- Pixels are accumulated in a buffer using the z-buffer algorithm:
  
  Clear Z buffer (set z to infinity and store bg-color)
  For each new pixel P(i,j) rendered
    If the distance $z_P$ of P is smaller than $Z(i,j)$
      store the $z_P$ & color of P in $Z(i,j)$
  Send the buffer to display
Shadow Maps: Z-Buffer Shadow Generation

- Uses multi-pass rendering
  - Render scene from the light source "view-point" and save the Z depth instead of the color. This is called the shadow map and indicates the closest pixel to the light.
  - Render the scene as usual from the observer and save Z depth values (z-buffer).
  - Transform every pixel \((x,y,z)\) in the z-buffer to the light source coordinate space \((x',y',z')\).
  - Project \((x',y',z')\) onto the shadow map (the “light” z-buffer) to get \((xs,ys)\) and compare the z value at that pixel to \(z'\).
  - If they are identical, that pixel is illuminated by the light source, otherwise \((z'\) must be larger) the original \((x,y,z)\) pixel is shadowed!
Shadow Map Geometry
Comparing the “Z” Values

Camera Viewpoint

Light “Viewpoint”
Light's Z-buffer
Observer's Z-buffer
Observer's Image
With shadows.
Environment Stored Textures
3D Rendering Pipeline Done!

**3D Geometric Primitives**

- Transform into 3D world coordinate system
- Illuminate according to lighting and reflectance
- Transform into 3D camera coordinate system
- Transform into 2D camera coordinate system
- Clip primitives outside camera’s view
- Draw pixels (includes texturing, hidden surface, etc.)