What We’ve Covered So Far

1. Position objects and the camera in the world
2. Compute position of objects relative to the camera
3. Project objects onto the screen
4. Sample triangle coverage
5. Interpolate triangle attributes
6. Sample texture maps
Rotating Cubes in Perspective
Rotating Cubes in Perspective
Rotating Cubes in Perspective
What Else Are We Missing?

Credit: Bertrand Benoit. “Sweet Feast,” 2009. [Blender /VRay]
What Else Are We Missing?

Credit: Giuseppe Albergo. “Colibri” [Blender]
What Else Are We Missing?

Surface representations

• Objects in the real world exhibit highly complex geometric details

Lighting and materials

• Appearance is a result of how light sources reflect off complex materials

Camera models

• Real lenses create images with focusing and other optical effects
Course Roadmap

Rasterization Pipeline
  Core Concepts
    • Sampling
    • Antialiasing
    • Transforms

Geometric Modeling

Lighting & Materials

Cameras & Imaging

Intro
Rasterization
Transforms & Projection
Texture Mapping
Today: Visibility, Shading, Overall Pipeline

CS184/284A
Ren Ng, James O’Brien
Visibility
Painter's Algorithm

Inspired by how painters paint

Paint from back to front, overwrite in the framebuffer
Painter’s Algorithm

Requires sorting in depth (O(n log n) for n triangles)
Can have unresolvable depth order

(BSP Trees will provide a way of dealing with this problem.)
Z-Buffer

This is the hidden-surface-removal algorithm that eventually won.

Idea:

• Store current min. z-value for each sample position
• Needs an additional buffer for depth values
  • framebuffer stores RBG color values
  • depth buffer (z-buffer) stores depth (16 to 32 bits)
Z-Buffer Example

Rendering

Depth buffer
Z-Buffer Algorithm

Initialize depth buffer to $\infty$

During rasterization:

for (each triangle $T$)
  for (each sample $(x,y,z)$ in $T$)
    if ($z < zbuffer[x,y]$) // closest sample so far
      framebuffer[x,y] = rgb; // update color
      zbuffer[x,y] = z; // update z
    else
      ; // do nothing, this sample is not closest

Ren Ng, James O’Brien
Z-Buffer Algorithm

(Pretend these numbers are negative, i.e. distance from near plane.)
Z-Buffer Complexity

Complexity

• $O(n)$ for $n$ triangles

• How can we sort $n$ triangles in linear time?

Most important visibility algorithm

• Implemented in hardware for all GPUs

• Used by OpenGL
Z-Buffer and Transparency

Transparency requires partial sorting

Common solution:

- Draw opaque polygons first
- Then draw transparent polygons (Ideally in sorted order)
Z-Buffer and Transparency

Transparency requires partial sorting

- Linked list of RGB-Z-\(\alpha\) at each pixel (Alpha Buffer)
Shadow Maps

- Pre-render scene from perspective of light source
  - Only render Z-Buffer (the shadow buffer)
- Render scene from camera perspective
  - Compare with shadow buffer
  - If nearer light, if further shadow
Shadow Maps

Shadow Buffer

Image w/ Shadows

From Stamminger and Drettakis
SIGGRAPH 2002

Note: These images don’t really go together; see the paper...

Ren Ng, James O’Brien
Deep Shadow Maps

- Some objects only partially occlude light
  - A single shadow value will not work
  - Similar to transparency in Z-Buffer

From Lokovic and Veach
SIGGRAPH 2000

Ren Ng, James O’Brien
Simple Shading
(Blinn-Phong Reflection Model)
Simple Shading vs Realistic Lighting & Materials

What we will cover today

• A local shading model: simple, per-pixel, fast
• Based on perceptual observations, not physics

What we will cover later in the course

• Physics-based lighting and material representations
• Global light transport simulation
Perceptual Observations

Specular highlights

Diffuse reflection

Ambient lighting

Photo credit: Jessica Andrews, flickr
Local Shading

Compute light reflected toward camera

Inputs:

- Viewer direction, \( v \)
- Surface normal, \( n \)
- Light direction, \( l \)  
  (for each of many lights)
- Surface parameters 
  (color, shininess, ...)

No “global” effects.
Falloff

Physically correct: $\frac{1}{r^2}$ light intensify falloff

• Tends to look bad with local shading (why?)

Sometimes compromise of $\frac{1}{r}$ used.

Very important to use $\frac{1}{r^2}$ for correct global illumination methods.
Diffuse Reflection

Light is scattered uniformly in all directions

- Surface color is the same for all viewing directions

Lambert’s cosine law

Top face of cube receives a certain amount of light

Top face of 60° rotated cube intercepts half the light

In general, light per unit area is proportional to

$$\cos \theta = \mathbf{l} \cdot \mathbf{n}$$
Lambertian (Diffuse) Shading

Shading independent of view direction

\[ L_d = k_d \left( \frac{I}{r^2} \right) \max(0, n \cdot l) \]
Lambertian (Diffuse) Shading

Produces matte appearance

\[ k_d \]
Perceptual Observations

- Specular highlights
- Diffuse reflection
- Ambient lighting

Photo credit: Jessica Andrews, flickr
Specular Shading (Blinn-Phong)

Intensity depends on view direction

• Bright near mirror reflection direction
Specular Shading (Blinn-Phong)

Close to mirror direction $\Leftrightarrow$ half vector near normal

- Measure “near” by dot product of unit vectors

\[
\mathbf{h} = \text{bisector}(\mathbf{v}, \mathbf{l}) = \frac{\mathbf{v} + \mathbf{l}}{||\mathbf{v} + \mathbf{l}||}
\]

\[
L_s = k_s \left( \frac{I}{r^2} \right) \max(0, \cos \alpha)^p
\]

\[
= k_s \left( \frac{I}{r^2} \right) \max(0, \mathbf{n} \cdot \mathbf{h})^p
\]
Cosine Power Plots

Increasing $p$ narrows the reflection lobe

[Foley et al.]
Specular Shading (Blinn-Phong)

\[ L_s = k_s \left( \frac{I}{r^2} \right) \max(0, \mathbf{n} \cdot \mathbf{h})^p \]
Specular Shading (Blinn-Phong)
Direction -vs- Point Lights

For a point light, the light direction changes over the surface.

For “distant” light, the direction is constant.

Similar for orthographic/perspective viewer.
Spot and Other Lights

Other calculations for useful effects
- Spot light
- Only light certain objects
- Negative lights
- etc.
Ugly....
Ugly....
Perceptual Observations

Specular highlights

Diffuse reflection

Ambient lighting

Photo credit: Jessica Andrews, flickr
Ambient Shading

Shading that does not depend on anything

- Add constant color to account for disregarded illumination and fill in black shadows

\[ L_a = k_a I_a \]

- ambient coefficient
- reflected ambient light

Ren Ng, James O’Brien
Blinn-Phong Reflection Model

\[ L = L_a + L_d + L_s \]

\[ = k_a I_a + k_d \left( \frac{I}{r^2} \right) \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s \left( \frac{I}{r^2} \right) \max(0, \mathbf{n} \cdot \mathbf{h})^p \]
Blinn-Phong Reflection Model

\[
L = L_a + L_d + L_s \\
= k_a I_a + k_d \left( \frac{I}{r^2} \right) \max(0, n \cdot l) + k_s \left( \frac{I}{r^2} \right) \max(0, n \cdot h)^p
\]
Ashikhmin-Shirley BRDF

- More realistic specular term (for some materials)
- Anisotropic specularities
- Fresnel behavior (grazing angle highlights)
- Energy preserving diffuse term
- Sum of diffuse and specular terms (as before)

\[ \rho(\hat{l}, \hat{v}) = \rho_d(\hat{l}, \hat{v}) + \rho_s(\hat{l}, \hat{v}) \]

Ashikhmin-Shirley BRDF

\[
\rho_s(\hat{l}, \hat{e}) = \frac{\sqrt{(p_u + 1)(p_v + 1)}}{8\pi} \frac{(\hat{n} \cdot \hat{h})p_u \cos^2 \phi + p_v \sin^2 \phi}{(\hat{h} \cdot \hat{e}) \max((\hat{n} \cdot \hat{e}), (\hat{n} \cdot \hat{l}))} F(\hat{h} \cdot \hat{e})
\]

\[
F(\hat{h} \cdot \hat{e}) = K_s + (1 - K_s)(1 - (\hat{h} \cdot \hat{e}))^5
\]

- \(\hat{l}\) Light direction
- \(\hat{e}\) Viewer (eye) direction
- \(p_u, p_v\) Specular powers
- \(\hat{n}\) Normal
- \(\hat{h}\) Half angle
- \(K_s\) Specular coefficient (color)
- \(\hat{u}, \hat{v}\) Parametric directions
Ashikhmin-Shirley BRDF

\[
\rho_s(\hat{l}, \hat{e}) = \frac{\sqrt{(p_u + 1)(p_v + 1)}}{8\pi} \frac{(\hat{n} \cdot \hat{h}) \frac{p_u(\hat{h} \cdot \hat{n})^2 + p_u(\hat{h} \cdot \hat{v})^2}{1 - (\hat{h} \cdot \hat{n})^2}}{(\hat{h} \cdot \hat{e}) \max((\hat{n} \cdot \hat{e}), (\hat{n} \cdot \hat{l}))} F(\hat{h} \cdot \hat{e})
\]

\[
F(\hat{h} \cdot \hat{e}) = K_s + (1 - K_s)(1 - (\hat{h} \cdot \hat{e}))^5
\]

Approximate Fresnel function

\hat{l} \quad \text{Light direction}
\hat{e} \quad \text{Viewer (eye) direction}
p_u, p_v \quad \text{Specular powers}
\hat{n} \quad \text{Normal}
\hat{h} \quad \text{Half angle}
K_s \quad \text{Specular coefficient (color)}
\hat{u}, \hat{v} \quad \text{Parametric directions}
Ashikhmin-Shirley BRDF

\[
\rho_d(\hat{l}, \hat{e}) = \frac{28K_d}{23\pi} (1 - K_s) \left( 1 - \left( 1 - \frac{\hat{n} \cdot \hat{e}}{2} \right)^5 \right) \left( 1 - \left( 1 - \frac{\hat{n} \cdot \hat{l}}{2} \right)^5 \right)
\]

Note: The Phong diffuse term (Lambertian) is independent of view. But this term accounts for unavailable light due to specular/Fresnel reflection.

\begin{itemize}
  \item \(\hat{l}\) Light direction
  \item \(\hat{e}\) Viewer (eye) direction
  \item \(p_u, p_v\) Specular powers
  \item \(\hat{n}\) Normal
  \item \(\hat{h}\) Half angle
  \item \(K_s\) Specular coefficient (color)
  \item \(\hat{u}, \hat{v}\) Parametric directions
\end{itemize}
Ashikhmin-Shirley BRDF
Ashikhmin-Shirley BRDF
Ashikhmin-Shirley BRDF

Figure 1: Metallic spheres for various exponents.

- $n_v = 10000$
- $n_v = 1000$
- $n_v = 100$
- $n_v = 10$
- $n_u = 10$
- $n_u = 100$
- $n_u = 1000$
- $n_u = 10000$

We now illustrate the use of the model in several figures. After that, the remainder of the paper deals with specifying and implementing the model. Figure 1 shows spheres with $R_d = 0$ and varying $n_u$ and $n_v$. The spheres along the diagonal have $n_u = n_v$ so have a look similar to the traditional Phong model. Figure 2 shows another metallic object. This appearance is achieved by using the "right" mapping of tangent vectors on the surface. Figure 3 shows a "polished" surface with $R_s = 0$. This means the diffuse component will dominate for near normal viewing angles. However, as the viewing angle becomes oblique the specular component dominates despite its low near-normal value. Figure 4 shows the model for a diffuse surface. Note how the ball on the right has highlights near the edge of the model, and how the constant-BRDF ball on the left is more "flat". The highlights produced by the new model are present in the measured BRDFs of some paints, so are desirable for some applications [4].
Beyond BRDFs

The BRDF model does not capture everything
• e.g. Subsurface scattering (BSSRDF)

Images from Jensen et. al, SIGGRAPH 2001
Beyond BRDFs

The BRDF model does not capture everything
• e.g. Inter-frequency interactions

\[ \rho = \rho(\theta_V, \theta_L, \lambda_{in}, \lambda_{out}) \quad \text{This version would work...} \]
Measured BRDFs

BRDFs for automotive paint
Measured BRDFs

BRDFs for aerosol spray paint

Images from Cornell University Program of Computer Graphics
Measured BRDFs

BRDFs for house paint
Measured BRDFs

BRDFs for lucite sheet
Measuring BRDF

Images from Marc Levoy
Other Color Effects

Images from Gooch et al, 1998

Ren Ng, James O’Brien
Shading Triangle Meshes
Shading Frequency: Triangle, Vertex or Pixel

Shade each triangle (flat shading)
- Triangle face is flat — one normal vector
- Not good for smooth surfaces

Shade each vertex ("Gouraud" shading)
- Interpolate colors from vertices across triangle
- Each vertex has a normal vector

Shade each pixel ("Phong" shading)
- Interpolate normal vectors across each triangle
- Compute full shading model at each pixel
Shading Frequency: Face, Vertex or Pixel

Shading freq.: Face, Vertex, Pixel
Shading type: Flat, Gouraud, Phong (*)

Num Vertices
Defining Per-Vertex Normal Vectors

Best to get vertex normals from the underlying geometry
  • e.g. consider a sphere

Otherwise have to infer vertex normals from triangle faces
  • Simple scheme: average surrounding face normals

\[ \mathbf{N}_v = \frac{\sum_i \mathbf{N}_i}{\| \sum_i \mathbf{N}_i \|} \]
Defining Per-Pixel Normal Vectors

Barycentric interpolation of vertex normals

Problem: length of vectors?
Smooth Shading

From blender.stackexchange.com
Rasterization Pipeline
Rasterization Pipeline

- **Vertex Processing**
  - **Input**: vertices in 3D space
  - **Output**: vertices positioned in screen space

- **Triangle Processing**
  - **Input**: vertices in 3D space
  - **Output**: triangles positioned in screen space

- **Rasterization**
  - **Input**: vertices in 3D space
  - **Output**: triangles positioned in screen space

- **Fragment Processing**
  - **Input**: vertices in 3D space
  - **Output**: fragments (one per covered sample)

- **Framebuffer Operations**
  - **Input**: vertices in 3D space
  - **Output**: shaded fragments

- **Display**
  - **Input**: vertices in 3D space
  - **Output**: image (pixels)
Shader Programs

• Program vertex and fragment processing stages
• Describe operation on a single vertex (or fragment)

Example GLSL fragment shader program

```glsl
uniform sampler2D myTexture;
uniform vec3 lightDir;
varying vec2 uv;
varying vec3 norm;

void diffuseShader()
{
  vec3 kd;
  kd = texture2d(myTexture, uv);
  kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);
  gl_FragColor = vec4(kd, 1.0);
}
```

• Shader function executes once per fragment.
• Outputs color of surface at the current fragment’s screen sample position.
• This shader performs a texture lookup to obtain the surface’s material color at this point, then performs a diffuse lighting calculation.
Shader Programs

• Program vertex and fragment processing stages
• Describe operation on a single vertex (or fragment)

Example GLSL fragment shader program

```glsl
uniform sampler2D myTexture;    // program parameter
uniform vec3 lightDir;          // program parameter
varying vec2 uv;                // per fragment value (interp. by rasterizer)
varying vec3 norm;              // per fragment value (interp. by rasterizer)

void diffuseShader()
{
    vec3 kd;
    kd = texture2d(myTexture, uv);  // material color from texture
    kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);  // Lambertian shading model
    gl_FragColor = vec4(kd, 1.0);  // output fragment color
}
```
Shader Programs

- Program vertex and fragment processing stages
- Describe operation on a single vertex (or fragment)

Example GLSL fragment shader program

```cpp
uniform sampler2D myTexture; // program parameter
uniform vec3 lightDir; // program parameter
varying vec2 uv; // per fragment value (interp. by rasterizer)
varying vec3 norm; // per fragment value (interp. by rasterizer)

void diffuseShader()
{
    vec3 kd;
    kd = texture2d(myTexture, uv); // material color from texture
    kd *= clamp(dot(-lightDir, norm), 0.0, 1.0); // Lambertian shading model
    gl_FragColor = vec4(kd, 1.0); // output fragment color
}
```
Shader Programs

Measuring and Modeling the Appearance of Finished Wood


Code on GitHub: https://github.com/mckennapsean/wood-shader
Goal: Highly Complex 3D Scenes in Realtime

- 100’s of thousands to millions of triangles in a scene
- Complex vertex and fragment shader computations
- High resolution (3-5+ megapixel + supersampling)
- 30-60 frames per second (even higher for VR)
Graphics Pipeline Implementation: GPUs

Specialized processors for executing graphics pipeline computations

Discrete GPU Card
(NVIDIA GeForce Titan X)

Integrated GPU:
(Part of Intel CPU die)
CPU vs GPU

https://www.youtube.com/watch?v=ZrJeYFxpUyQ

CS184/284A

Ren Ng, James O’Brien
CPU vs GPU

https://www.youtube.com/watch?v=ZrJeYFxpUyQ
Modern GPUs offer ~2-4 Tera-FLOPs of performance for executing vertex and fragment shader programs.
Things to Remember

Visibility

• Painter’s algorithm and Z-Buffer algorithm

Simple Shading Model

• Key geometry: lighting, viewing & normal vectors
• Ambient, diffuse & specular reflection functions
• Shading frequency: triangle, vertex or fragment

Graphics Rasterization Pipeline

• Where do transforms, rasterization, shading, texturing and visibility computations occur?
• GPU = parallel processor implementing graphics pipeline
Acknowledgments

This slide set contain contributions from:

• Kayvon Fatahalian
• David Forsyth
• Pat Hanrahan
• Angjoo Kanazawa
• Ren Ng
• James O’Brien
• Mark Pauly