Lecture 5:
The Rasterization Pipeline

Computer Graphics and Imaging
UC Berkeley CS184/284A, Spring 2016
What We’ve Covered So Far

Position objects and the camera in the world

Compute position of objects relative to the camera

Project objects onto the screen

Sample triangle coverage

Interpolate triangle attributes

Sample texture maps
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
- Intro
- Rasterization
- Transforms & Projection
- Texture Mapping
- Today: Visibility, Shading, Overall Pipeline

Geometric Modeling

Lighting & Materials

Cameras & Imaging

CS184/284A, Lecture 5
Ren Ng, Spring 2016
Visibility
Painter’s Algorithm

Inspired by how painters paint
Paint from back to front, overwrite in the framebuffer

[Wikipedia]
Painter’s Algorithm

Requires sorting in depth ($O(n \log n)$ for $n$ triangles)

Can have unresolvable depth order
Z-Buffer (Visibility Solution That’s Won)

Store current min. z-value for each sample position

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

Image credit: Dominic Alves, flickr.
Z-Buffer Algorithm

Initialize depth buffer to $\infty$

During rasterization:

for (each triangle $T$)
  for (each sample $(x,y,z)$ in $T$)
    if ($z < \text{zbuffer}[x,y]$) // closest so far
      framebuffer[$x,y$] = rgb; // update color
      zbuffer[$x,y$] = z; // update z
Z-Buffer Algorithm

```
R R R R R R R R
R R R R R R R R
R R R R R R R R
R R R R R R R R
R R R R R R R R
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5
```
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
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, ...)

\[
\begin{align*}
v & \quad l & \quad n & \quad v \\
\end{align*}
\]
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 \( \cos \theta = l \cdot n \)
Light Falloff

intensity here: $I / r^2$

intensity here: $I$

$r$
Lambertian (Diffuse) Shading

Shading independent of view direction

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

illumination from source

diffuse coefficient

diffusely reflected light
Lambertian (Diffuse) Shading

Produces matte appearance

$k_d$
Specular Shading (Blinn-Phong)

Intensity depends on view direction

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

Close to mirror ⇔ half vector near normal

- Measure “near” by dot product of unit vectors

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

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

Increasing $p$ narrows the reflection lobe

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

\[ L_s = k_s \left( \frac{I}{r^2} \right) \max(0, n \cdot h)^p \]
Ambient Shading

Shading that does not depend on anything

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

\[ L_a = k_a I_a \]
Blinn-Phong Reflection Model

\[ L = L_a + L_d + L_s \]

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

- **Application**
  - **Vertex Processing**
    - **Vertex Stream**
  - **Triangle Processing**
    - **Triangle Stream**
  - **Rasterization**
    - **Fragment Stream**
  - **Fragment Processing**
    - **Shaded Fragments**
  - **Framebuffer Operations**
    - **Display**

Input: vertices in 3D space

Vertices positioned in screen space

Triangles positioned in screen space

Fragments (one per covered sample)

Shaded fragments

Output: image (pixels)
Rasterization Pipeline

Vertex Processing

Triangle Processing

Rasterization

Fragment Processing

Framebuffer Operations

Application

Modeling & viewing transforms

Vertex Stream

Triangle Stream

Fragment Stream

Shaded Fragments

Display

CS184/284A, Lecture 5

Ren Ng, Spring 2016
Rasterization Pipeline

- Application
  - Vertex Processing
    - Vertex Stream
  - Triangle Processing
    - Triangle Stream
  - Rasterization
    - Fragment Stream
  - Fragment Processing
    - Shaded Fragments
  - Framebuffer Operations
    - Display

Sampling triangle coverage
Rasterization Pipeline

- **Vertex Processing**
  - Vertex Stream

- **Triangle Processing**
  - Triangle Stream

- **Rasterization**
  - Fragment Stream
  - Fragment Processing
    - Shaded Fragments

- **Framebuffer Operations**

  Display

Evaluating shading functions:

- Ambient + Diffuse
- + Specular = Phong Reflection
Rasterization Pipeline

Application

- Vertex Processing
  - Vertex Stream
- Triangle Processing
  - Triangle Stream
- Rasterization
  - Fragment Stream
- Fragment Processing
  - Shaded Fragments
- Framebuffer Operations
  - Display

Texture mapping
Rasterization Pipeline

Application

Vertex Processing
Vertex Stream

Triangle Processing
Triangle Stream

Rasterization
Fragment Stream

Fragment Processing
Shaded Fragments

Framebuffer Operations

Display

Z-Buffer Visibility Tests

CS184/284A, Lecture 5
Ren Ng, Spring 2016
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

```
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
}
```
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 (2-4 megapixel + supersampling)
- 30-60 frames per second
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)
Modern GPUs offer ~2-4 Tera-FLOPs of performance for executing vertex and fragment shader programs. Tera-Op’s of fixed-function compute capability over here.
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

Thanks to Steve Marschner, Mark Pauly and Kayvon Fatahalian for presentation resources.