Level 0 - Full Resolution Texture
Level 2 - Downsampling 4x4

Aliasing

Blurring
Level 4 - Downsampling 16x16
Mipmap (L. Williams 83)

“Mip” comes from the Latin “multum in parvo”, meaning a multitude in a small space
Mipmap (L. Williams 83)

“Mip hierarchy”
level = D

What is the storage overhead of a mipmap?
Computing Mipmap Level D

Estimate texture footprint using texture coordinates of neighboring screen samples
Computing Mipmap Level D

\[ D = \log_2 L \]

\[ L = \max \left( \sqrt{\left( \frac{du}{dx} \right)^2 + \left( \frac{dv}{dx} \right)^2}, \sqrt{\left( \frac{du}{dy} \right)^2 + \left( \frac{dv}{dy} \right)^2} \right) \]
Computing Mipmap Level D

\[ D = \log_2 L \]

\[ L = \max \left( \sqrt{\left( \frac{du}{dx} \right)^2 + \left( \frac{dv}{dx} \right)^2}, \sqrt{\left( \frac{du}{dy} \right)^2 + \left( \frac{dv}{dy} \right)^2} \right) \]
Visualization of Mipmap Level

D clamped to nearest level
Trilinear Filtering

Linear interpolation based on continuous D

Mipmap Level \( \lfloor D \rfloor \)

Mipmap Level \( \lfloor D \rfloor + 1 \)

Bilinear result

Bilinear result
Visualization of Mipmap Level

Trilinear filtering: visualization of continuous D
Bilinear vs Trilinear Filtering Cost

Bilinear resampling:
• 4 texel reads
• 3 lerps (3 mul + 6 add)

Trilinear resampling:
• 8 texel reads
• 7 lerps (7 mul + 14 add)
Mipmap Limitations

Point sampling
Mipmap Limitations

Supersampling 512x
Mipmap Limitations

Overblur
Why?

Mipmap trilinear sampling
Recall: Screen Pixel Footprint in Texture

NB: texture sampling pattern not rectilinear or isotropic
Mipmap Limitations

Mipmap trilinear sampling
Anisotropic Filtering

Elliptical weighted average (EWA) filtering
Anisotropic Filtering

Ripmaps and summed area tables
- Can look up axis-aligned rectangular zones
- Diagonal footprints still a problem

EWA filtering
- Use multiple lookups
- Weighted average
- Mipmap hierarchy still helps
Advanced Texturing Methods
Environment Map

A function from the sphere to colors, stored as a texture.

Lat / long texture map

Reflection vector indexes into texture map

[Blinn & Newell 1976]
Spherical Environment Map

Hand with Reflecting Sphere. M. C. Escher, 1935. lithograph

Light Probes, Paul Debevec
Environmental Lighting

Environment map (left) used to render realistic lighting
Cube Map

A vector maps to cube point along that direction. The cube is textured with 6 square texture maps.
Bump Mapping

Texture stores perturbation to surface normal

[Blinn 1978]

CS184/284A

Ren Ng
Displacement Mapping

Texture stores perturbation to surface position
Bump Mapping vs. Displacement Mapping

What is different?

Bump mapping
Perturbs normals

Displacement mapping
Perturbs positions
3D Textures and Volume Rendering
3D Procedural Noise + Solid Modeling

Perlin noise, Ken Perlin
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
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

[Foley et al.]
Z-Buffer (Visibility Solution That’s Won)

Store current min. z-value for each sample position

Additional buffer for depth values

- framebuffer stores color values (RGB, RGBA, ...)
- 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
Z-Buffer Complexity

Complexity

• $O(n)$ for $n$ triangles

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, …)
Diffuse Reflection

Light is scattered uniformly in all directions

- Surface color is the same for all viewing directions

Lambert’s cosine law

In general, light per unit area is proportional to $\cos \theta = l \cdot n$
Light Falloff

intensity here: $I$

intensity here: $I/r^2$
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
Specular Shading (Blinn-Phong)

Intensity depends on view direction

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

Close to mirror $\Leftrightarrow$ 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
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

\[ L_a = k_a I_a \]

ambient coefficient

reflected ambient light

\[ L_a = k_a I_a \]
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

\[
N_v = \frac{\sum_i N_i}{\| \sum_i N_i \|}
\]
Defining Per-Pixel Normal Vectors

Barycentric interpolation of vertex normals
Rasterization Pipeline
Rasterization Pipeline

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

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

Modeling & viewing transforms
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

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

Z-Buffer Visibility Tests

Display
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
}
```
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.
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.