**Today**

**Shading**
- Light sources
- Shader programming

**Shading**
- Compute interaction of light with surfaces
- Interactive applications
  - Local illumination
  - Simplified models reflection at surfaces

**Diffuse reflection**
- Ideal diffuse material reflects light equally in all directions
- View-independent
- Matte, not shiny materials
  - Paper
  - Unfinished wood
  - Unpolished stone

**Diffuse reflection**
- Lambert's cosine law
- Lambertian surfaces
- What about the moon?
Specular reflection
- Mirror like reflection
  - Glossy materials

Glossy teapot

Phong model
- Specular reflectance coefficient $k_s$
- Phong exponent $p$
  - Higher $p$, smaller (sharper) highlight

Blinn model (Jim Blinn, 1977)
- Define unit halfway vector
  \[ h = \frac{L + e}{||L + e||} \]
- Halfway vector represents normal of microfacet that would lead to mirror reflection to the eye

Blinn model
- The larger the angle between microfacet orientation and normal, the less likely
- Use cosine of angle between them
- Shininess parameter $s$
- Very similar to Phong

Local illumination
Simplified model
- Sum of 3 components
- Covers a large class of real surfaces

Ambient light
- In real world, light is bounced all around scene
- Could use global illumination techniques to simulate
- Simple approximation
  - Add constant ambient light at each point $k_a c_a$
  - Ambient light $c_a$
  - Ambient reflection coefficient $k_a$
- Areas with no direct illumination are not completely dark
Questions?

Light sources
- Light sources can have complex properties
  - Geometric area over which light is produced
  - Anisotropy in direction
  - Variation in color
  - Reflective surfaces act as light sources

- Interactive rendering is based on simple, standard light sources

Light sources
- At each point on surfaces need to know
  - Direction of incoming light (the $L$ vector)
  - Strength of incoming light (the $c_i$ values)

- Standard light sources in OpenGL
  - Directional: from a specific direction
  - Point light source: from a specific point
  - Spotlight: from a specific point with intensity that depends on the direction

Point lights
- Simple model for light bulbs
- Point that radiates light in all directions equally
  - Light vector varies across the surface
  - Intensity drops off proportionally to the inverse square of the distance from the light
  - Intuition for inverse square falloff?

Directional light
- Light from a distant source
  - Light rays are parallel
  - Direction and strength the same everywhere
  - As if the source were infinitely far away
  - Good approximation to sunlight
- Specified by a unit length direction vector, and a color

Light sources
- Light sources can have complex properties
  - Geometric area over which light is produced
  - Anisotropy in direction
  - Variation in color
  - Reflective surfaces act as light sources

- Interactive rendering is based on simple, standard light sources

Complete model
- Blinn model with several light sources $i$

$$ e = \sum_i c_i (k_d (L \cdot n) + k_s (L \cdot n)^2) + k_a c_i $$

- Directional: from a specific direction
- Point light source: from a specific point
- Spotlight: from a specific point with intensity that depends on the direction

Directional light
- Light from a distant source
  - Light rays are parallel
  - Direction and strength the same everywhere
  - As if the source were infinitely far away
  - Good approximation to sunlight
- Specified by a unit length direction vector, and a color

$ c_{src} \cdot d $

Light source

$L = -d$

$ c_i = c_{src} $
Point lights

- Light source
- Receiving surface

\[
L = \frac{p - v}{\|p - v\|}
\]
\[
c_l = \frac{c_{src}}{\|p - v\|^2}
\]

Attenuation

- Sometimes, it is desirable to modify the inverse square falloff behavior of point lights
- Common (OpenGL) model for distance attenuation
  \[
c_l = \frac{c_{src}}{k_v + k_i\|p - v\| + k_q\|p - v\|^2}
\]
- Not physically accurate

Spotlights

- Like point source, but intensity depends on direction

Parameters

- Position, the location of the source
- Spot direction, the center axis of the light
- Falloff parameters
  - how broad the beam is (cone angle)
  - how light tapers off at edges of the beam (cosine exponent)

\[
L = \frac{p - v}{\|p - v\|}
\]
\[
c_l = \begin{cases} 
0 & \text{if } -L \cdot d \leq \cos(\theta_{max}) \\
\frac{c_{src}(-L \cdot d)^f}{c_l} & \text{otherwise}
\end{cases}
\]

Questions?
Per-triangle, -vertex, -pixel shading

• May compute shading operations
  - Once per triangle
  - Once per vertex
  - Once per pixel

Per-triangle shading

• Known as flat shading
• Evaluate shading once per triangle
• Advantages
  - Fast
• Disadvantages
  - Faceted appearance

Per-vertex shading

• Known as Gouraud shading (Henri Gouraud 1971)
• Interpolate vertex colors across triangles
• OpenGL default
• Advantages
  - Fast
  - Smoother than flat shading
• Disadvantages
  - Problems with small highlights

Per-pixel shading

• Also known as Phong interpolation (not to be confused with Phong illumination model)
  - Rasterizer interpolates normals across triangles
  - Illumination model evaluated at each pixel
  - Implemented using fragment shaders (later today)
• Advantages
  - Higher quality than Gouraud shading
• Disadvantages
  - Much slower

Gouraud vs. per-pixel shading

• Gouraud has problems with highlights
• Could use more triangles...

Demo

• Cg browser surface reflectance
Shading in OpenGL

// Somewhere in the initialization part of your
// program:
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);

// Make sure vertex colors are used as material properties
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glColorMaterial(GL_FRONT, GL_SPECULAR);

// Create light components
GLfloat ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
GLfloat diffuseLight[] = { 0.8f, 0.8f, 0.8, 1.0f };
GLfloat specularLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat position[] = { -1.5f, 1.0f, -4.0f, 1.0f };

// Assign created components to GL_LIGHT0
glightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
glightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
glightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
glightfv(GL_LIGHT0, GL_POSITION, position);

Shading computations
(diffuse, specular, ambient)
performed automatically
(unless you use shader programs)

Transforming normals
• If object-to-camera transformation $M$ includes shearing, transforming normals using $M$ does not work
  - Transformed normals are not perpendicular to surface any more
• To avoid problem, need to transform normals by $M^{-T}$
• No derivation here, but remember for rotations $R^{-T} = R$
• OpenGL does this automatically for us

Questions?

Today

Shading
• Light sources
• Shader programming
Configurable pipeline

Before 2002

- APIs (OpenGL, Direct3D) to configure the rendering pipeline
- Enable/disable functionality
  - E.g., lighting, texturing
- Set parameters for given functionality
  - E.g., light direction, texture blending mode

Disadvantages

- Restricted to preset functionality
  - Limited types of light sources (directional, point, spot)
  - Limited set of reflectance models (ambient, diffuse, Phong)
  - Limited use of texture maps
- Want more flexibility, fancier effects

Programmable pipeline

- Replace functionality in parts of the pipeline by user specified programs
- Called shaders, or shader programs
- Not all functionality in the pipeline is programmable

Shader programs

- Written in a shading language
  - Cg, early shading language by NVidia
  - OpenGL’s shading language is GLSL
  - DirectX’s shading language is HLSL (high level shading language)
  - Similar to C, with specialties
- Recent, quickly changing technology
- Driven by more and more flexible GPUs

Programmable pipeline (2006)

Two types of shader programs
1. Vertex program
2. Fragment program (fragment: pixel location inside a triangle and interpolated data)

GPU architecture

NVidia NV80 (GeForce 8800 Series)
128 stream processors

Programmable pipeline (2006)

Still fixed functionality
• Projective division
• Rasterization
  - I.e., determine which pixels lie inside triangle
  - Vertex attribute interpolation (color, texture coords.)
• Access to framebuffer
  - Texture filtering
  - Z-buffering
  - Framebuffer blending

Shader programming

• Application can provide
  - No shaders, standard OpenGL functions are executed
  - Vertex shader only
  - Fragment shader only
  - Vertex and fragment shader
• Each shader is a separate piece of code
• Output of vertex shader is interpolated at each fragment and accessible as input to fragment shader

Vertex programs

• Executed once for every vertex
• Replaces functionality for
  - Modelview, projection transformation
  - Per-vertex shading
• If you use a vertex program, need to implement this functionality in the program
• Vertex shader often used for animation
  - Characters
  - Particle systems

Fragment programs

• Executed once for every fragment
• Implements functionality for
  - Texturing
  - Fancy per pixel effects
  - Per-pixel shading
  - Bump mapping
  - Shadows
  - Etc.

Creating shaders in OpenGL

Questions?
**Vertex programs**

- **Vertex attributes**
  - Coordinates in object space, additional vertex attributes
  - From application
  - To rasterizer
  - Transformed vertices, processed vertex attributes

- **Uniform parameters**
  - OpenGL state, application specified parameters

**Types of input data**

- **Vertex attributes**
  - Change for each execution of the vertex program
  - Predefined OpenGL attributes (color, position, etc.)
  - User defined attributes

- **Uniform parameters**
  - Do usually not change from vertex to vertex
  - OpenGL state variables
  - Application defined parameters

**Vertex attributes**

- “Data that flows down the pipeline with each vertex”
- Per-vertex data that your application specifies
- E.g., vertex position, color, normal, texture coordinates
- Declared using `attribute` storage classifier in your shader code
  - Read-only

**OpenGL state variables**

- Provide access to state of rendering pipeline, which you set through OpenGL calls in application
- Predefined variables
  - `uniform mat4 gl_ModelViewMatrix;`
  - `uniform mat4 gl_ModelViewProjectionMatrix;`
  - `uniform mat4 gl_ProjectionMatrix;`
  - `uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];`
  - etc.
- Declared using `uniform` storage classifier
  - Read-only

**Uniform parameters**

- Parameters that are set by the application
- Should not change frequently
  - Not on a per-vertex basis!
- Will be the same for each vertex until application changes it again
- Declared using `uniform` storage classifier
  - Read-only
Uniform parameters

- **To access**, use `glGetUniformLocation`, `glUniform*` in application

**Example**
- In shader declare
  ```
  uniform float a;
  ```
- In application, set a using
  ```
  GLuint p;
  //... initialize program p
  int i=glGetUniformLocation(p,"a");
  glUniform1f(i, 1.f);
  ```

Output variables

- **Required output**: homogeneous vertex coordinates
  ```
  vec4 gl_Position
  ```
- **varying outputs**
  - Mechanism to send data to the fragment shader
  - Will be interpolated during rasterization
  - Interpolated values accessible in fragment shader (using same variable name)
- **Predefined varying outputs**
  ```
  varying vec4 gl_FrontColor;
  varying vec4 gl_TexCoord[ ];
  ```
  etc.
- **User defined varying outputs**

**Note**

- Any predefined output variable that you do not write will assume the value of the current OpenGL state
- E.g., your vertex shader does not write varying vec4 gl_TexCoord[ ]
  - Your fragment shader may still read it
  - The value will be the current OpenGL state

“Hello world” vertex program

- **main()** function is executed for every vertex
- Use predefined variables

```
void main()
{
  gl_Position = // required output
  gl_ProjectionMatrix * // predefined uniform
  gl_ModelViewMatrix * // predefined uniform
  gl_Vertex; // predefined attribute
}
```
- Alternatively, use
  ```
  gl_ModelViewProjectionMatrix OR ftransform()
  ```

Vertex programs

**Current limitations (2006)**

- Cannot write data to any memory accessible by application
- Cannot pass data between vertices
  - Each vertex is independent
- One vertex in, one vertex out
  - Cannot generate new geometry

Examples

- Animation
  - Character skinning
  - Particle systems
  - Water
### Fragment programs

- **Fragment data**
  - Interpolated vertex attributes, additional fragment attributes
  - From rasterizer
  - To fixed framebuffer access functionality (z-buffering, etc.)

- **Uniform parameters**
  - OpenGL state, application specified parameters
  - From rasterizer

### Types of input data

- **Fragment data**
  - Change for each execution of the fragment program
  - Interpolated from vertex data during rasterization, varying variables
  - User defined varying variables
  - Predefined OpenGL varying variables

- **Uniform parameters**
  - Do not change from fragment to fragment
  - OpenGL state variables
  - Application defined parameters

### Fragment data

- E.g., interpolated fragment color, texture coordinates
- Standard OpenGL fragment data accessible through **predefined** variables
  ```glsl```
  ```
  varying vec4 gl_Color;
  varying vec4 gl_TexCoord[ ];
  ```
  ```
  etc.
  ```

- **Note** varying storage classifier, read-only
- User defined data possible, too

### Uniform parameters

- Same as in vertex shader
- OpenGL state
- Application defined parameters
  - Use `glGetUniformLocation`, `glUniform*` in application

### Output variables

- **Predefined outputs**
  - `gl_FragColor`
  - `gl_FragDepth`
- Result is undefined if you don't write these variables

### “Hello world” fragment program

- **main()** function is executed for every fragment
- Use predefined variables
- Draws everything in bluish color

```glsl```
```c
void main()
{
    gl_FragColor = vec4(0.4, 0.4, 0.8, 1.0);
}
```
Examples

- Fancy per pixel shading
  - Bump mapping
  - Displacement mapping
  - Realistic reflection models
  - Cartoon shading
  - Shadows
  - Etc.
- Most often, vertex and fragment shader work together to achieve desired effect

Fragment programs

Current limitations (2006)

- Cannot read framebuffer
- Can only write to framebuffer pixel that corresponds to fragment being processed
  - No random write access to framebuffer
- Number of varying variables passed from vertex to fragment shader is limited
- Number of application defined uniform parameters is limited

Summary

- Shader programs replace part of the rendering pipeline
- Written in special shading language (GLSL in OpenGL)
- Sequence of OpenGL calls to compile/activate shaders
- Two types of shaders (2006)
  - Vertex shaders
  - Fragment shaders

GLSL main features

- Similar to C, with specialties
  - attribute, uniform, varying storage classifiers
- Set of predefined variables to access OpenGL state
- Built in vector data types, vector operations
- No pointers

GLSL quick reference

http://www.opengl.org/sdk/libs/OpenSceneGraph/glsl_quickref.pdf
Questions?

Tutorials and documentation
- OpenGL and GLSL specifications
  http://www.opengl.org/documentation/specs/
- GLSL tutorials
  http://www.lighthouse3d.com/opengl/glsll/
  http://www.clockworkcoders.com/oglsl/tutorials.html
- OpenGL Programming Guide (red book)
- OpenGL Shading Language (orange book)

Shader development tools
- ATI/AMD render monkey (OpenGL and DirectX shaders)
- NVidia FX composer (DirectX shaders only)
  http://developer.nvidia.com/object/fx_composer_home.html

Next time
- Texture mapping