Shading

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

Complete model

• Blinn model with several light sources $i$

\[ e = \sum_i c_i (I_i \cdot n) + k_r (I_i \cdot n)^r + k_a c_a \]

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

Today

Shading

• Light sources
• Shader programming

Light sources

• At each point on surfaces need to know
  - Direction of incoming light (the $\mathbf{L}$ 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
### 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

![Directional light diagram](image)

### 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?

![Point lights diagram](image)

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

![Attenuation diagram](image)

### 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 he beam (cosine exponent)

![Spotlights diagram](image)
### 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
<table>
<thead>
<tr>
<th>Gouraud vs. per-pixel shading</th>
<th>Demo</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Gouraud has problems with highlights</td>
<td>• Cg browser surface reflectance</td>
</tr>
<tr>
<td>• Could use more triangles...</td>
<td></td>
</tr>
</tbody>
</table>

![Gouraud vs. Per-pixel](image)

<table>
<thead>
<tr>
<th>Shading in OpenGL</th>
<th>Transforming normals</th>
<th>Questions?</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Somewhere in the initialization part of your program.</td>
<td>• If object-to-camera transformation $M$ includes shearing, transforming normals using $M$ does not work</td>
<td></td>
</tr>
<tr>
<td>glEnable(GL_LIGHTING);</td>
<td>- Transformed normals are not perpendicular to surface any more</td>
<td></td>
</tr>
<tr>
<td>glEnable(GL_LIGHT0);</td>
<td>• To avoid problem, need to transform normals by $M^{-T}$</td>
<td></td>
</tr>
<tr>
<td>// Make sure vertex colors are used as material properties</td>
<td>• No derivation here, but remember for rotations $R^{-T} = R$</td>
<td></td>
</tr>
<tr>
<td>glEnable(GL_COLOR_MATERIAL);</td>
<td>• OpenGL does this automatically for us</td>
<td></td>
</tr>
<tr>
<td>glEnableMaterial(GL_FRONT, GL_DIFFUSE);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>glEnableMaterial(GL_FRONT, GL_SPECULAR);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>// Create light components</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLfloat ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLfloat diffuseLight[] = { 0.8f, 0.8f, 0.8f, 1.0f };</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLfloat specularLight[] = { 0.5f, 0.5f, 0.5f, 1.0f };</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLfloat position[] = { -1.5f, 1.0f, -4.0f, 1.0f };</td>
<td></td>
<td></td>
</tr>
<tr>
<td>// Assign created components to GL_LIGHT0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>glLightfv(GL_LIGHT0, GL_POSITION, position);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>// Need to provide per vertex normals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>// Shading is performed in camera space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Position, direction of light sources is transformed by GL_MODELVIEW matrix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• If light sources should be fixed relative to objects</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Set GL_MODELVIEW to desired object-to-camera transform</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Choose object space coordinates for light position</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Will be transformed using current GL_MODELVIEW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Lots of details, highly recommend OpenGL programming guide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- <a href="http://glprogramming.com/red/chapter05.html">http://glprogramming.com/red/chapter05.html</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- <a href="http://www.falloutsoftware.com/tutorials/gl/g8.htm">http://www.falloutsoftware.com/tutorials/gl/g8.htm</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shading in OpenGL</th>
<th>Transforming normals</th>
<th>Questions?</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Need to provide per vertex normals</td>
<td>• If object-to-camera transformation $M$ includes shearing, transforming normals using $M$ does not work</td>
<td></td>
</tr>
<tr>
<td>• Shading is performed in camera space</td>
<td>- Transformed normals are not perpendicular to surface any more</td>
<td></td>
</tr>
<tr>
<td>- Position, direction of light sources is transformed by GL_MODELVIEW matrix</td>
<td>• To avoid problem, need to transform normals by $M^{-T}$</td>
<td></td>
</tr>
<tr>
<td>• If light sources should be fixed relative to objects</td>
<td>• No derivation here, but remember for rotations $R^{-T} = R$</td>
<td></td>
</tr>
<tr>
<td>- Set GL_MODELVIEW to desired object-to-camera transform</td>
<td>• OpenGL does this automatically for us</td>
<td></td>
</tr>
<tr>
<td>- Choose object space coordinates for light position</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Will be transformed using current GL_MODELVIEW</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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’ 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)
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
  - Transformed vertices, processed vertex attributes

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

**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[ ];
    ```

  - **User defined varying outputs**

**Output variables**

**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

  ```
  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

- Uniform parameters
  - OpenGL state, application specified parameters

**Fragment data**

- E.g., interpolated fragment color, texture coordinates
- Standard OpenGL fragment data accessible through **predefined** variables
  - `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

```cpp
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**

[Link to GLSL quick reference]
### Questions?

- OpenGL and GLSL specifications
- GLSL tutorials
- 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)

### Next time

- Texture mapping