CS130 : Computer Graphics
Lecture 8: Lighting and Shading (cont.)

Tamar Shinar
Computer Science & Engineering
UC Riverside
Shading Polygonal Geometry
Smooth surfaces are often approximated by polygons.

Shading approaches:

1. Flat
2. Smooth (Gouraud)
3. Phong

Each polygon is flat and has a well-defined normal.
Flat Shading

do the shading calculation once per polygon

valid for light at $\infty$ and viewer at $\infty$ and faceted surfaces

In general, $l$, $n$, and $v$ vary from point to point on a surface. If we assume a distant viewer, $v$ can be thought of as constant. If we assume a distant light source, $l$ can be thought of as constant. For a flat polygon, $n$ is constant.

If the light source or viewer is not at inf, we need heuristic for picking color – e.g., first vertex, or polygon center.
Flat shading doesn’t usually look too good. The **lateral inhibition** effect makes flat shading seem even worse.
Smooth Shading

We assign the vertex normals based on the surrounding polygon normals.

\[ n = \frac{n_1 + n_2 + n_3 + n_4}{\left| n_1 + n_2 + n_3 + n_4 \right|} \]

Do the shading calculation once per vertex.
Interpolating Normals

- Must renormalize
Interpolating Normals

- Must renormalize
Interpolating Normals

• Must renormalize
We can interpolate attributes using barycentric coordinates

\[ c = \alpha c_0 + \beta c_1 + \gamma c_2 \]

Gouraud shading
(Gouraud, 1971)

Using barycentric coordinates also has the advantage that we can easily interpolate colors or other attributes from triangle vertices
Phong Shading

do the shading calculation once per fragment

Phong shading requires normals to be interpolated across each polygon -- this wasn’t part of the fixed function pipeline. This can now be done in the pipeline in the fragment shader.
Comparison

- Phong interpolation looks smoother -- can see edges on the Gouraud model
- but Phong is a lot more work
- both Phong and Gouraud require vertex normals
- both Phong and Gouraud leave silhouettes
Problems with Interpolated Shading

- Polygonal silhouette
- Perspective distortion
- Orientation dependence
- Unrepresentative surface normals
Programmable Shading
Fixed-Function Pipeline

User Program \[\rightarrow\] Geometry Processing \[\rightarrow\] Pixel Processing

CPU \[\rightarrow\] GPU

Control pipeline through GL state variables

- The application supplies geometric primitives through a graphics API such as **OpenGL** or **DirectX**
- control of pipeline operation through state variables only
Programmable Pipeline

- Supply shader programs to be executed on GPU as part of pipeline

- can supply shader programs to carry out vertex processing, geometry processing, and pixel processing
Phong reflectance in vertex and pixel shaders using GLSL

Phong reflectance as a vertex shader
- vertex shaders can be used to move/animate verts
- linear interpolation of vertex lighting

as a fragment shader
- each fragment is calculated individually – don’t know about neighboring pixels

void main(void)
{
    vec4 v = gl_modelView_Matrix * gl_Vertex;
    vec3 n = normalize(gl_NormalMatrix * gl_Normal);
    vec3 l = normalize(gl_lightSource[0].position - v);
    vec3 h = normalize(l - normalize(v));

    float p = 16;
    vec4 cr = gl_FrontMaterial.diffuse;
    vec4 cl = fl_LightSource[0].diffuse;
    vec4 ca = vec4(0.2, 0.2, 0.2, 1.0);

    vec4 color;
    if (dot(h,n) > 0)
        color = cr * (ca + cl * max(0,dot(l,n,l)))
            + cl* pow(dot(h,n), p);
    else
        color = cr * (ca + cl * max(0,dot(l,n,l)));

    gl_FrontColor = color;
    gl_Position = ftransform();
}

varying vec4 v;
varying vec3 n;

void main(void)
{
    vec3 l = normalize(gl_lightSource[0].position - v);
    vec3 h = normalize(l - normalize(v));

    float p = 16;
    vec4 cr = gl_FrontMaterial.diffuse;
    vec4 cl = fl_LightSource[0].diffuse;
    vec4 ca = vec4(0.2, 0.2, 0.2, 1.0);

    vec4 color;
    if (dot(h,n) > 0)
        color = cr * (ca + cl * max(0,dot(l,n,l)))
            + cl* pow(dot(h,n), p);
    else
        color = cr * (ca + cl * max(0,dot(l,n,l)));

    gl_FragColor = color;

}
Programmable shader examples from NVIDIA and ATI
Computing Normal Vectors
Plane Normals

\[ \mathbf{v} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0) \]

\[ \mathbf{n} = \frac{\mathbf{v}}{||\mathbf{v}||} \]
Implicit function normals

\[ f(p) = 0 \]

\[ \nabla f(p) \]

\[ \nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{pmatrix} \]

sphere

\[ p \cdot p - r^2 = 0 \]

plane

\[ n \cdot (p - p_0) = 0 \]
Parametric form

\[ p(u, v) = \begin{pmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{pmatrix} \]

**tangent vectors**

\[ \frac{\partial p}{\partial u}, \frac{\partial p}{\partial v} \]

**normal**

\[ \frac{\frac{\partial p}{\partial u} \times \frac{\partial p}{\partial v}}{|| \frac{\partial p}{\partial u} \times \frac{\partial p}{\partial v} ||} \]
Texture Mapping
There are limits to geometric modeling

Although modern GPUs can render millions of triangles/sec, that’s not enough sometimes...
Use texture mapping to increase realism through detail

This image is just 8 polygons!

Add visual complexity.

http://www.siggraph.org/education/materials/HyperGraph/mapping/r_wolfe/r_wolfe_mapping_1.htm
No texture

With texture
Pixar - Toy Story
Textures can be anything that you can lookup values in -- photo, procedurally generated, or even a function that computes a value on the fly.
3D solid textures
Other uses of textures...

- Light maps
- Shadow maps
- Environment maps
- Bump maps
- Opacity maps
- Animation

[Angel and Shreiner]

[Stam 99]
Texture mapping in the OpenGL pipeline

- Geometry and pixels have separate paths through pipeline
- meet in **fragment processing** - where textures are applied
- texture mapping applied at end of pipeline - efficient since relatively few polygons get past clipper
**uv Mapping**

- Texture is parameterized by \((u,v)\)
- Assign polygon vertices texture coordinates
- Interpolate within polygon

Texture coordinates are per-vertex data – a position in the \((u,v)\) space can interpolate tex coordinates with barycentric coordinates.
Texture Calibration