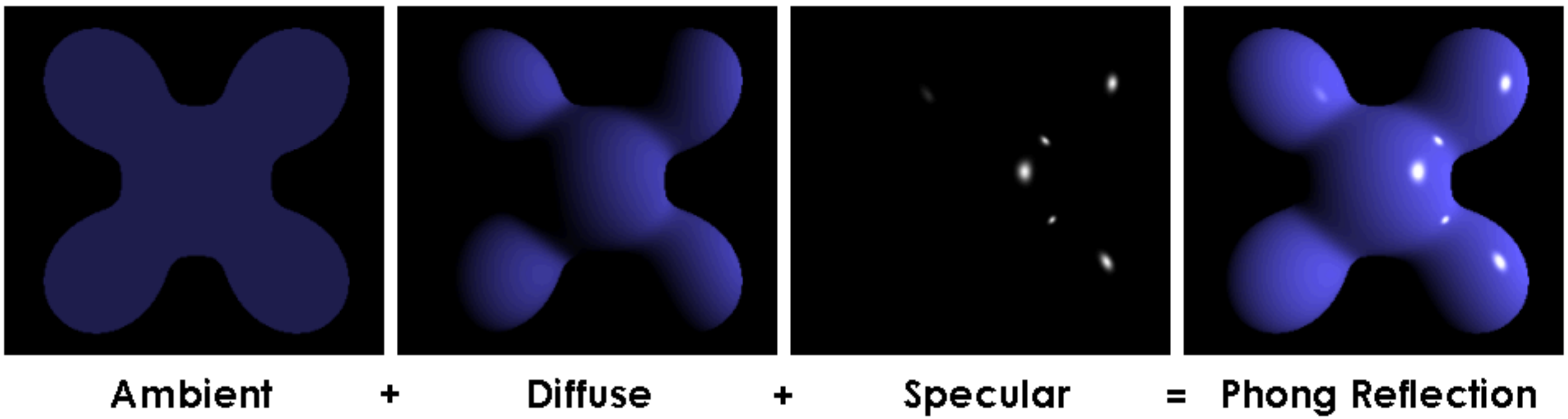
CS 418: Interactive Computer Graphics

Basic Shading in WebGL

Eric Shaffer

Phong Reflectance Model



Modified Phong Model

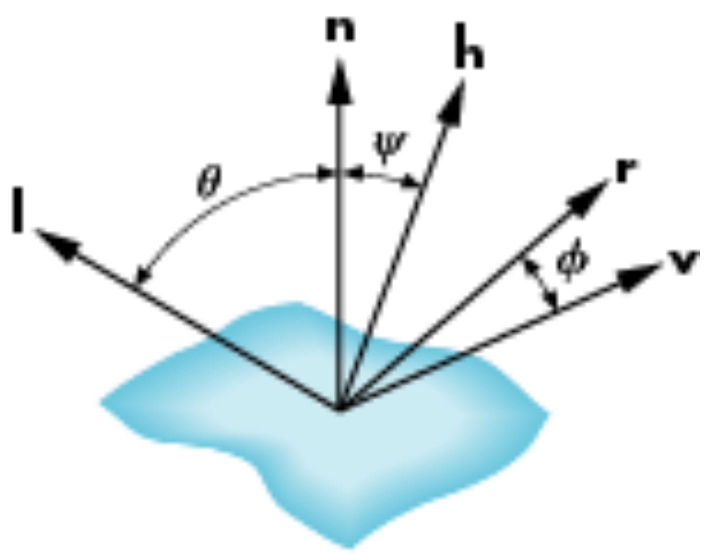
- The specular term in the Phong model is problematic
  - requires calculation of new reflection vector and view vector at each vertex

- Blinn suggested an approximation using the halfway vector
  - More efficient in terms of the operations used
  - If light and view don't change, computation is the same for all vertices
    - Uncommon situation IMO
  - Closer to physically correct lighting

The Halfway Vector

- **h** is normalized vector halfway between **l** and **v**

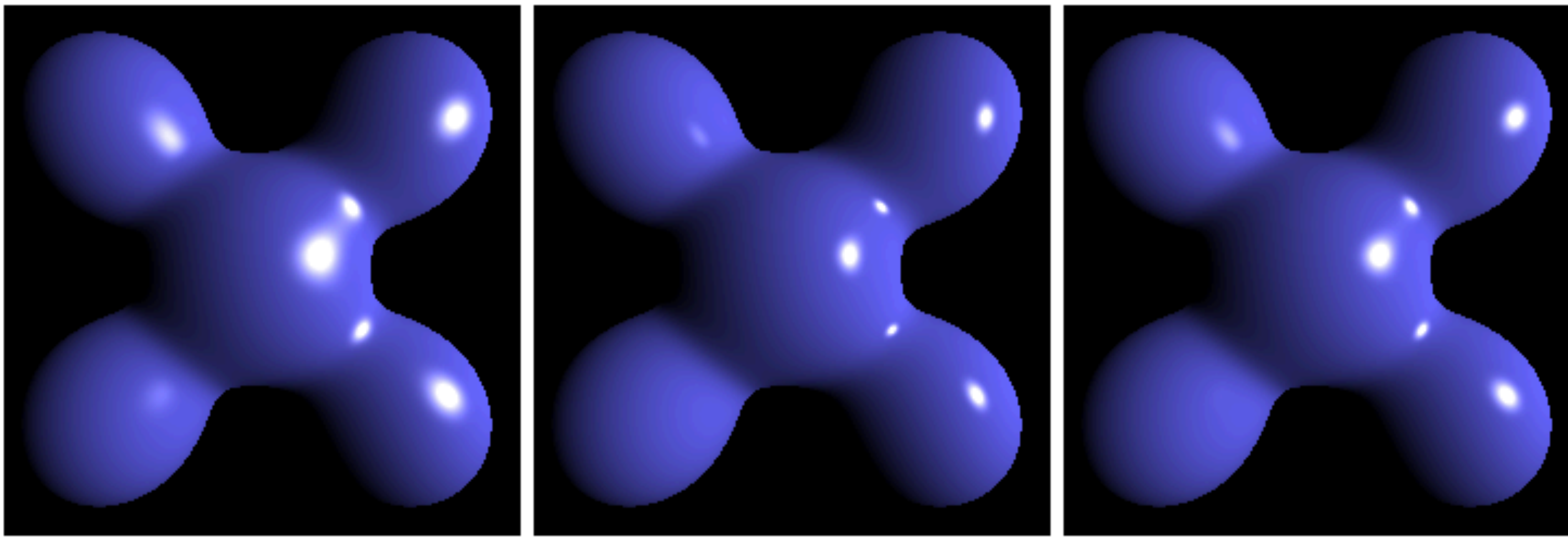
$$\mathbf{h} = (\mathbf{l} + \mathbf{v}) / |\mathbf{l} + \mathbf{v}|$$



Using the halfway vector

- Replace $(\mathbf{v} \cdot \mathbf{r})^{\alpha}$ by $(\mathbf{n} \cdot \mathbf{h})^{\beta}$
- β is chosen to match shininess
- Halfway angle is half of angle between **r** and **v**
  - if vectors are coplanar
- Model is known as the Phong-Blinn lighting model

Phong versus Blinn-Phong



Blinn–Phong | Phong | Blinn–Phong (higher exponent)

Normalization

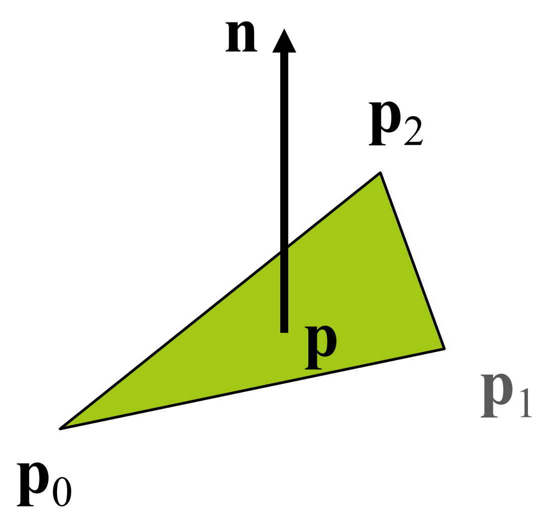
- Cosine terms can be computed using dot product
- Unit length vectors simplify calculation
- Usually we want to set the magnitudes to have unit length
- Be careful
  - Length can be affected by transformations
  - Note that scaling does not preserved length
- GLSL and glMatrix have a normalization function

Computing a Normal for a Triangle

plane $\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0$

$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)$

normalize $\mathbf{n} \leftarrow \mathbf{n} / |\mathbf{n}|$



Note that right-hand rule determines outward face

You can use the glMatrix library to compute normals

Specifying a Point Light Source

- For each light source define an RGBA color for
  - diffuse component
  - specular component
  - ambient component
  - the position

```
var diffuse0 = vec4.fromValues(1.0, 0.0, 0.0, 1.0);
var ambient0 = vec4.fromValues (1.0, 0.0, 0.0, 1.0);
var specular0 = vec4.fromValues (1.0, 0.0, 0.0, 1.0);
var light0_pos = vec4.fromValues (1.0, 2.0, 3,0, 1.0);
```

Distance and Direction

- The source colors are specified in RGBA
- The position is given in homogeneous coordinates
  - If w =1.0, we are specifying a finite location
  - If w =0.0, we are parallel source with the given direction vector
- The coefficients in distance terms are usually quadratic
  - (1/(a+b*d+c*d*d))
  - d is the distance from the point being rendered to the light source
  - a,b,c are constants of your choice

Moving Light Sources

- Light sources are geometric objects
- Positions and directions can be affected by the model-view matrix
  - If you want them to be
- Depending on where we place the position (direction) setting function, we can
  - Move the light source(s) with the object(s)
  - Fix the object(s) and move the light source(s)
  - Fix the light source(s) and move the object(s)
  - Move the light source(s) and object(s) independently

Material Properties

- Material properties
  - should match the terms in the light model
- Reflectivities
- last component gives opacity

```
var materialAmbient = vec4.fromValues( 1.0, 0.0, 1.0, 1.0 );
var materialDiffuse = vec4.fromValues( 1.0, 0.8, 0.0, 1.0);
var materialSpecular = vec4.fromValues( 1.0, 0.8, 0.0, 1.0 );
var materialShininess = 100.0;
```

Transparency

- Material properties are specified as RGBA values
- The A (alpha) value can be used to make the surface translucent
- The default is that all surfaces are opaque
- Later we will enable blending and use this feature

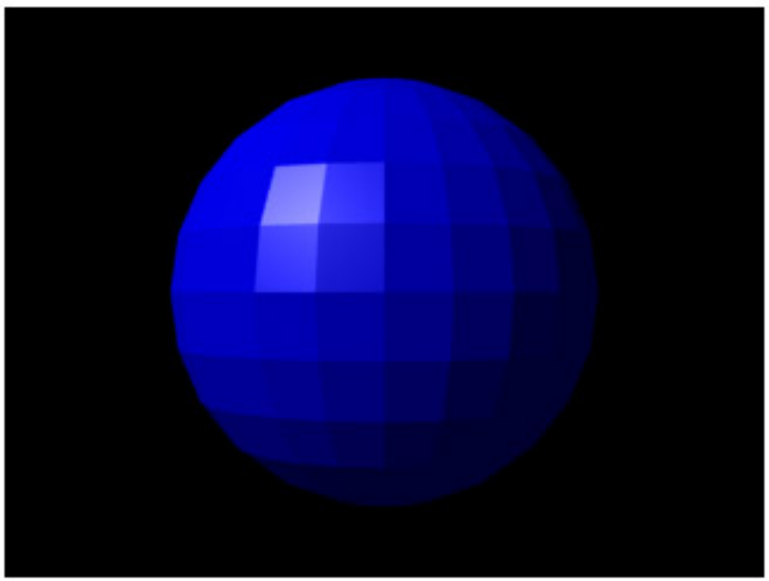
Polygonal Shading

- **Flat shading**
  - Each polygon is rendered with a color generated by the lighting model
    - Use the normal of the polygon in the shading calculation
  - Color is constant across the polygon
  - In WebGL, compute a color (shade for the polygon)
    - Use gl.drawArrays and gl.TRIANGLES with the same normal for each vertex
    - Why wouldn't this work with gl.TRIANGLE_FAN
- **Smooth Shading**
  - In per vertex shading we compute averaged normals for each vertex
    - Shading calculations are done for each vertex
    - Use the lighting model to compute a color (shade) at each vertex
    - Can be done at the application level or in the vertex shader
      - Shader is better…use the cores of the GPU to work in parallel

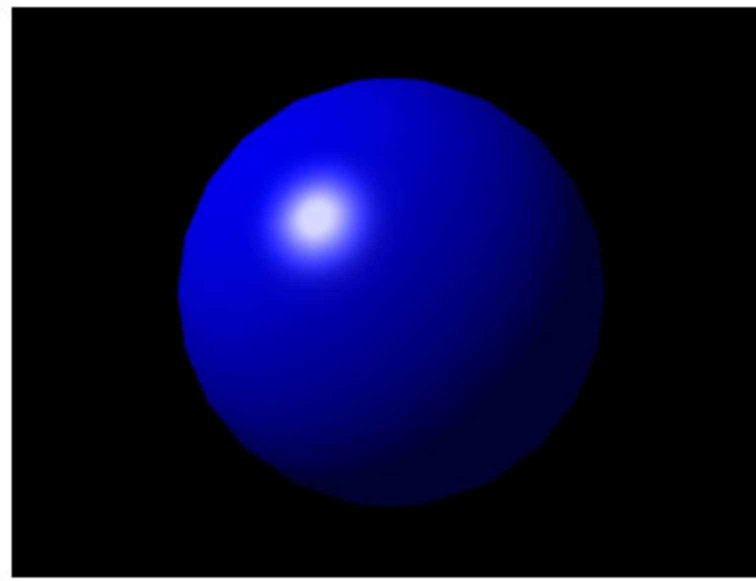
Smooth Shading: Two Types

- Gouraud Shading
  - Find average normal at each vertex (vertex normals)
  - Apply modified Phong model at each vertex
  - Interpolate vertex shades across each polygon
- Phong shading
  - Find vertex normals
  - Interpolate vertex normals across edges
  - Interpolate edge normals across polygon
  - Apply modified Phong model at each fragment

Smooth Shading and Flat Shading



FLAT SHADING



PHONG SHADING

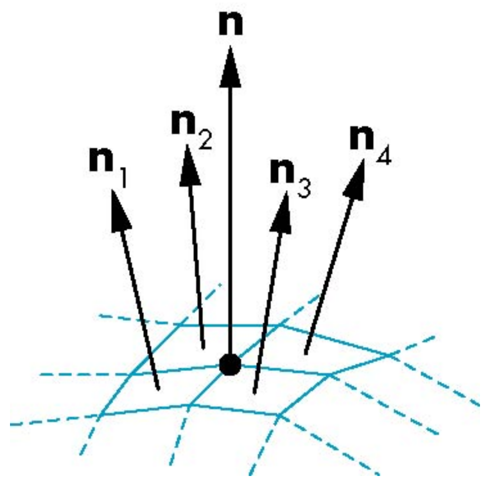
Computing Normals

- Easy for a sphere model
  - If centered at origin **n** = **p**

Mesh Shading

- The previous example is not general because we knew the normal at each vertex analytically
- For polygonal models, Gouraud proposed we use the average of the normals around a mesh vertex

$$\mathbf{n} = (\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4) / |\mathbf{n}_1+\mathbf{n}_2+\mathbf{n}_3+\mathbf{n}_4|$$



Comparison

- If polygon mesh approximates surfaces with high curvature
  - Phong shading may look smooth
  - Gouraud shading may show edges
- Phong shading requires more work than Gouraud
  - Until recently not available in real time systems
  - Now can be done using fragment shaders

Shading in the Vertex Shader

- Use uniforms for lighting parameters constant over all vertices
- Need to send in a normal as an attribute
- Send the color of the vertex to fragment shader

```
attribute vec3 aVertexNormal;
attribute vec3 aVertexPosition;
uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix;
uniform vec3 uLightPosition; // Already in Eye coordinates
uniform vec3 uAmbientLightColor;
uniform vec3 uDiffuseLightColor;
uniform vec3 uSpecularLightColor;
uniform vec3 uAmbientMatColor;
uniform vec3 uDiffuseMatColor;
uniform vec3 uSpecularMatColor;
const float shininess = 32.0;
varying vec4 vColor;
```

Shading in the Vertex Shader

- Compute necessary dot products and vectors

```
void main(void) {
// Get the vertex position in eye coordinates
  vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
  vec3 vertexPositionEye3 = vertexPositionEye4.xyz;

// Calculate the vector (l) to the light source
  vec3 vectorToLightSource = normalize(uLightPosition - vertexPositionEye3);

// Transform the normal (n) to eye coordinates
  vec3 normalEye = normalize(uNMatrix * aVertexNormal);

// Calculate the reflection vector (r) that is needed for specular light
 vec3 reflectionVector=normalize(reflect(-vectorToLightSource,
                                         normalEye));

// The camera in eye coordinates is located at the origin and is pointing
// along the negative z-axis. Calculate viewVector (v)
// in eye coordinates as: (0.0, 0.0, 0.0) - vertexPositionEye3
  vec3 viewVectorEye = -normalize(vertexPositionEye3);
```

Shading in the Vertex Shader

- Perform the shading calculation

```
// Calculate n dot l for diffuse lighting
 float diffuseLightWeighting = max(dot(normalEye, vectorToLightSource), 0.0);

// Calculate r dot v for specular lighting
 float rdotv = max(dot(reflectionVector, viewVectorEye), 0.0);
 float specularLightWeighting = pow(rdotv, shininess);

// Sum up all three reflection components and send to the fragment shader
 vColor = vec4((uAmbientLightColor * uAmbientMatColor)
              + ((uDiffuseLightColor * uDiffuseMatColor) * diffuseLightWeighting)
              + ((uSpecularLightColor * uSpecularMatColor)*specularLightWeighting),
              1.0);

 gl_Position = uPMatrix*uMVMatrix*vec4(aVertexPosition, 1.0);
 }
```