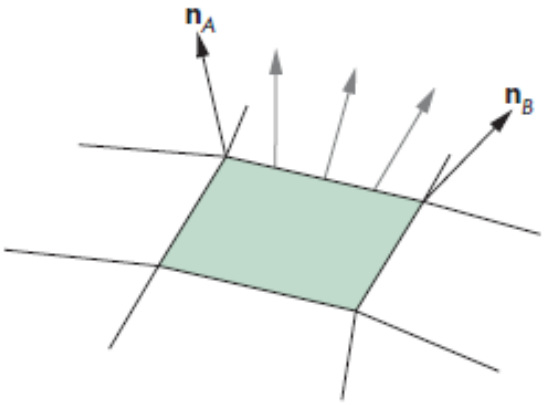
COSC 414/519I: Computer Graphics

2023W2

Shan Du

Phong Shading

- Even the smoothness introduced by Gouraud shading may not prevent the appearance of Mach bands.

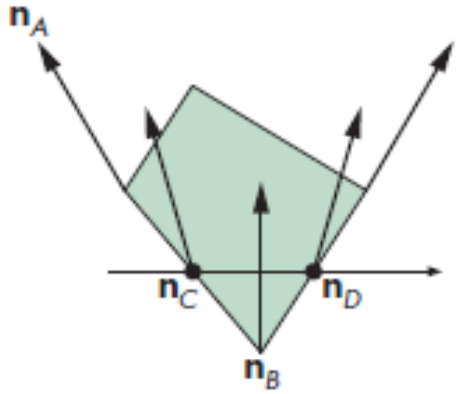


Edge normals.

- Phong proposed that instead of interpolating vertex intensities, we interpolate normals across each polygon.

Phong Shading

- We first compute vertex normals by interpolating over the normals of the polygons that share the vertex.



Interpolation of normals in Phong shading.

- Next, we use interpolation to interpolate the normals over the polygon.
  - We can use the interpolated normals at vertice *A* and *B* to interpolate normals along the edge between them:

$$n_c(\alpha) = (1 - \alpha)n_A + \alpha n_B$$

Phong Shading

- We can do a similar interpolation on all edges.
- The normal at any interior point can be obtained from points on the edges by

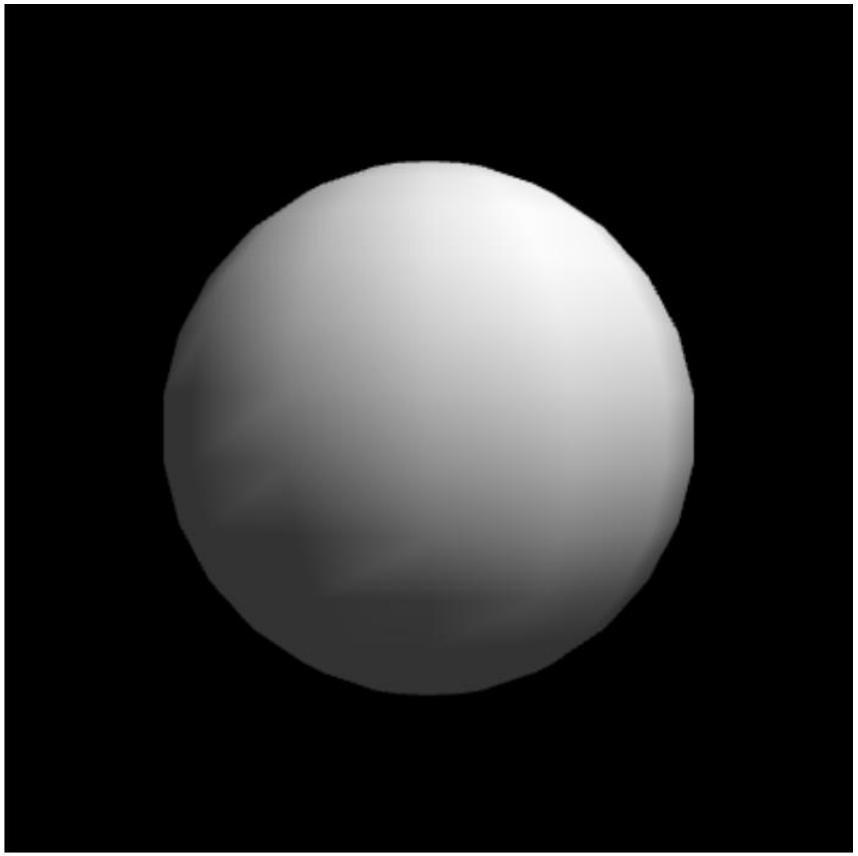
$$n(\alpha, \beta) = (1 - \beta)n_c + \beta n_D$$

- Once we have the normal at each point, we can make an independent shading calculation.
- Usually, this process can be combined with rasterization of the polygon.
- Until recently, Phong shading could only be carried out offline because it requires the interpolation of normals across each polygon.

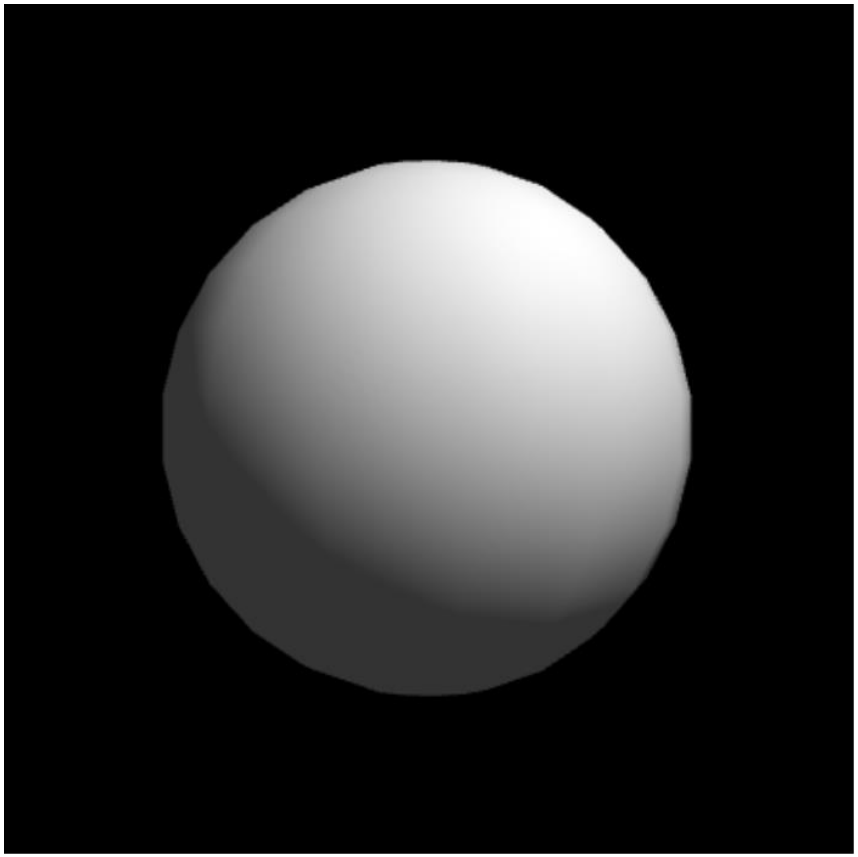
Phong Shading

- In terms of the pipeline, Phong shading requires that the lighting model be applied to each fragment, hence, the name per-fragment shading.
- We will implement Phong shading through a fragment shader.

Comparison



Per-vertex



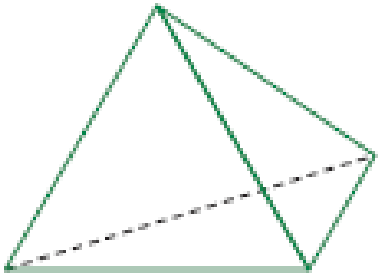
Per-fragment

Phong Shading

```
// Fragment shader program
var FSHADER_SOURCE =
 ...
  'uniform vec3 u_LightColor;\n' +     // Light color
  'uniform vec3 u_LightPosition;\n' +  // Position of the light source
  'uniform vec3 u_AmbientLight;\n' +   // Ambient light color
  'varying vec3 v_Normal;\n' +
  'varying vec3 v_Position;\n' +
  'varying vec4 v_Color;\n' +
  'void main() {\n' +
     // Normalize normal because it's interpolated and not 1.0 (length)
  '  vec3 normal = normalize(v_Normal);\n' +
     // Calculate the light direction and make it 1.0 in length
  '  vec3 lightDirection = normalize(u_LightPosition - v_Position);\n' +
     // The dot product of the light direction and the normal
  '  float nDotL = max(dot( lightDirection, normal), 0.0);\n' +
     // Calculate the final color from diffuse and ambient reflection
  '  vec3 diffuse = u_LightColor * v_Color.rgb * nDotL;\n' +
  '  vec3 ambient = u_AmbientLight * v_Color.rgb;\n' +
  '  gl_FragColor = vec4(diffuse + ambient, v_Color.a);\n' +
  '}\n';
```

Approximation of a Sphere by Recursive Subdivision

- Sphere is not an object supported by WebGL.
- We generate an approximation to a sphere using triangles through a process known as recursive subdivision.



Tetrahedron.

- Recursive subdivision is a powerful technique for generating approximations to curves and surfaces to any desired level of accuracy.
- Our start point is a tetrahedron.

Approximation of a Sphere by Recursive Subdivision

- We start with four vertices $(0,0,1)$, $\left(0, \frac{2\sqrt{2}}{3}, -\frac{1}{3}\right)$, $\left(-\frac{\sqrt{6}}{3}, -\frac{\sqrt{2}}{3}, -\frac{1}{3}\right)$, and $\left(\frac{\sqrt{6}}{3}, -\frac{\sqrt{2}}{3}, -\frac{1}{3}\right)$.
- All four lie on the unit sphere, centered at the origin.
- We get a first approximation by drawing a wireframe for the tetrahedron.

Approximation of a Sphere by Recursive Subdivision

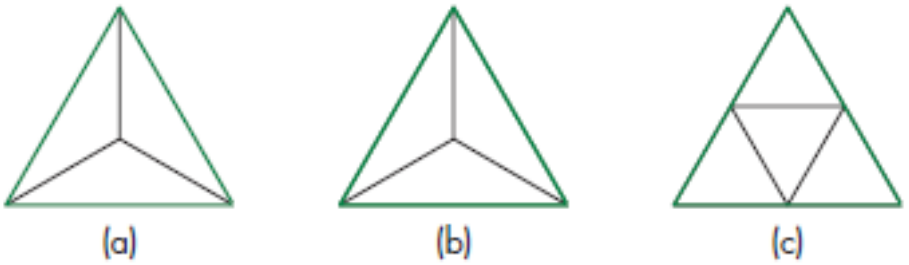
- We specify the four vertices by

```
var va = vec4(0.0, 0.0, -1.0, 1);
var vb = vec4(0.0, 0.942809, 0.333333, 1);
var vc = vec4(-0.816497, -0.471405, 0.333333, 1);
var vd = vec4(0.816497, -0.471405, 0.333333, 1);
```

- We can get a closer approximation to the sphere by subdividing each facet of the tetrahedron into smaller triangles.
- Subdividing into triangles will ensure that all the new facets will be flat.

Approximation of a Sphere by Recursive Subdivision

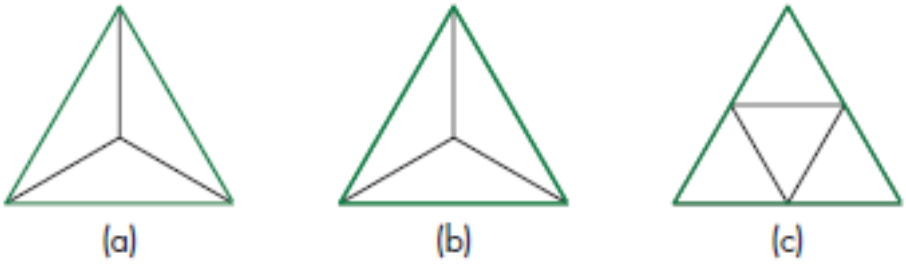
- There are at least three ways to do the subdivision.



Subdivision of a triangle by (a) bisecting angles, (b) computing the centroid, and (c) bisecting sides.

  - We can bisect each of the angles of the triangle and draw the three bisectors, which meet at a common point, thus generating three new triangles.

Approximation of a Sphere by Recursive Subdivision



Subdivision of a triangle by (a) bisecting angles, (b) computing the centroid, and (c) bisecting sides.

- We can also compute the center of mass (centroid) of the vertices by simply averaging them and then draw line from this point to the three vertices, again generating three triangles.
- We can connect the bisectors of the sides of the triangle, forming four equilateral triangles.

Approximation of a Sphere by Recursive Subdivision

- There are two main differences between the gasket program and the sphere program.
  - First, when we subdivide a face of the tetrahedron, we do not throw away the middle triangle formed from the three bisectors of the sides.
  - Second, although the vertices of a triangle lie on the circle, the bisector of a line segment joining any two of these vertices does not. We can push the bisectors to lie on the unit circle by normalizing their representations to have a unit length.

Approximation of a Sphere by Recursive Subdivision

- We initiate the recursion by

```
tetrahedron(va, vb, vc, vd, numTimesToSubdivide);
```

which divides the four sides,

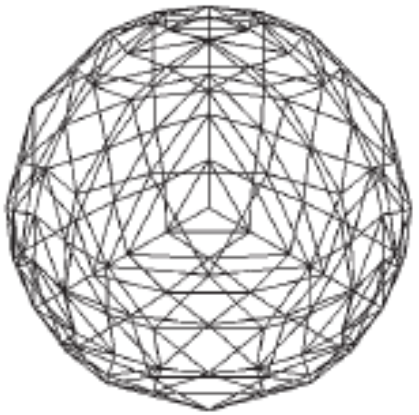
```
function tetrahedron(a, b, c, d, n)
{
  divideTriangle(a, b, c, n);
  divideTriangle(d, c, b, n);
  divideTriangle(a, d, b, n);
  divideTriangle(a, c, d, n);
}
```

with the midpoint subdivision:

Approximation of a Sphere by Recursive Subdivision

```
function divideTriangle(a, b, c, count)
{
  if (count > 0) {
    var ab = normalize(mix(a, b, 0.5), true);
    var ac = normalize(mix(a, c, 0.5), true);
    var bc = normalize(mix(b, c, 0.5), true);

    divideTriangle(a, ab, ac, count - 1);
    divideTriangle(ab, b, bc, count - 1);
    divideTriangle(bc, c, ac, count - 1);
    divideTriangle(ab, bc, ac, count - 1);
  }
  else {
    triangle(a, b, c);
  }
}
```



Sphere approximations using subdivision.

Specifying Lighting Parameters

- With programmable shaders, we are free to implement many lighting models.
- We can also choose where to apply a lighting model: in the application, in the vertex shader, or in the fragment shader.
- Consequently, we must define a group of lighting and material parameters and then either use them in the application code or send them to the shaders.

Light Sources

- For every light source, we must specify its color and either its location or its direction.
- We can specify the color for a single light as

```
var lightAmbient = vec4(0.2, 0.2, 0.2, 1.0);
var lightDiffuse = vec4(1.0, 1.0, 1.0, 1.0);
var lightSpecular = vec4(1.0, 1.0, 1.0, 1.0);
```

- We can specify the position of the light using

```
var lightPosition = vec4(1.0, 2.0, 3.0,1.0);
```

- If the fourth component is changed to zero, the source is a directional source.

```
var lightPosition = vec4(1.0, 2.0, 3.0, 0.0);
```

Light Sources

- For a positional light source, we also need to account for the attenuation of light received due to its distance from the source.
- We can use the distance attenuation model

$$f(d) = \frac{1}{a + bd + cd^2}$$

which contains constant, linear, and quadratic terms. We can use three floats for these values.

Light Sources

- We can also convert a positional source to a spotlight by setting its direction, the angle of the cone or the spotlight cutoff, and the drop-off rate or spotlight exponent. These three parameters can be specified by three floats.

Materials

- Material properties should match up directly with the supported light sources and with the chosen reflection model.
- We may also specify different material properties for the front and back faces of a surface.
- We may specify ambient, diffuse, and specular reflectivity coefficients (k_a, k_d, k_s) for each primary color through three colors.

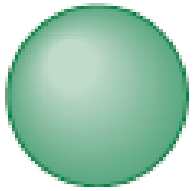
```
var materialAmbient = vec4(1.0, 0.0, 1.0, 1.0);
var materialDiffuse = vec4(1.0, 0.8, 0.0, 1.0);
var materialSpecular = vec4(1.0, 0.8, 0.0, 1.0);
```

Efficiency

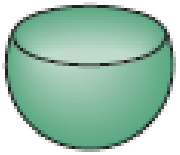
- For efficiency, we always do lighting calculation in the shaders.
- We can assume that either or both the viewer and the light source are far from the polygon. In this case, the diffuse term at each vertex would be identical, and we would need only one calculation per polygon. And we can simplify the specular term calculation.

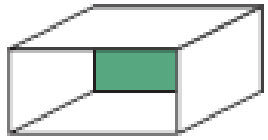
Efficiency

- In many situations, we can ignore all back faces by either culling them out or not rendering any face whose normal does not point toward the viewer.



Shading of convex objects.





Visible back surfaces.

Efficiency

- Light sources are special type of geometric objects and have geometric attributes, such as position, just like polygons and points.
- Light sources can be affected by transformations. We can specify them at the desired position or specify them in a convenient position and move them to the desired position by the model-view transformation.

Shading of the Sphere Model

- Because two adjacent triangles have different normals and thus are shaded with different colors, even if we create many triangles, we still can see the lack of smoothness.
- We can use the actual normals of the sphere for each vertex in the approximation.

Shaded sphere model.

Shading of the sphere with the true normals.

Per-Fragment Lighting

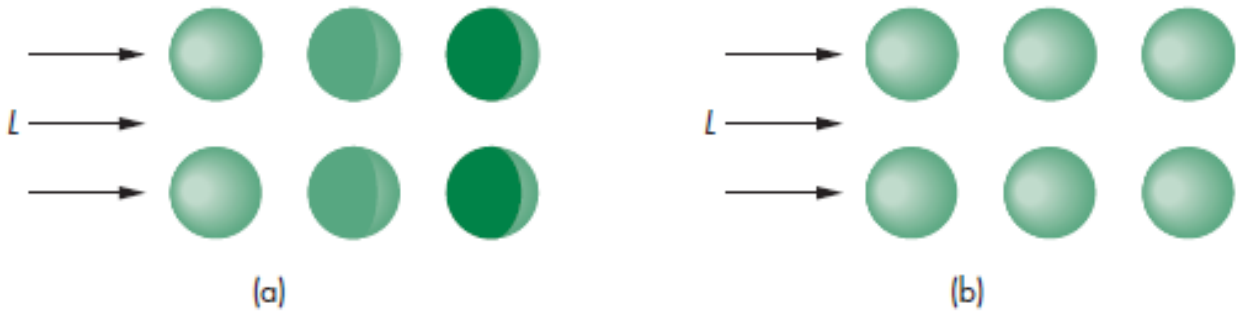
- We can do the lighting calculation on a per-fragment basis rather than on a per-vertex basis.
- With a fragment shader, we can do an independent lighting calculation for each fragment.
- The fragment shader needs to get the interpolated values of the normal vector, light source position, and eye position from the rasterizer.

Per-Fragment Lighting

- The vertex shader can compute these values and output them to the rasterizer.
- The fragment shader can apply Blinn-Phong lighting model to each fragment using the light and material parameters passed in from the application as uniform variables and the interpolated vectors from the rasterizer.

Global Illumination

- There are limitations imposed by the local lighting model that we have used.
- The objects close to the source block some of the light from the source from reaching the other objects.
- But local model shades each object independently and makes all objects appear same.



Array of shaded spheres. (a) Global lighting model. (b) Local lighting model.

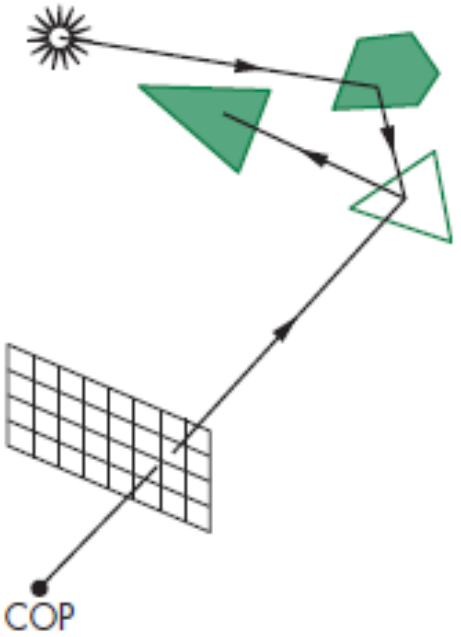
Global Illumination

- In addition, if these objects are specular, some light is scattered among objects. Thus, if the objects are very shiny, we should see the reflection of multiple objects in some of the objects and possibly even the multiple reflections of some objects in themselves.
- All these phenomena - shadows, reflections, blockage of light – are global effects and require a global lighting model.

Global Illumination

- There are two rendering strategies – ray tracing and radiosity – that can handle global effects.
  - Ray tracing starts with the synthetic-camera model but determines, for each projector that strikes a polygon, whether that point is indeed illuminated by one or more sources before computing the local shading at each point.

Global Illumination



- A local renderer might use the modified Phong model to compute the shade at the point of intersection.
- The ray tracer would find that the light source cannot strike the point of intersection directly, but that light from the source is reflected from the third polygon and this reflected light illuminates the point of intersection.

Global Illumination

- A radiosity renderer is based on energy considerations. From a physical point of view, all the light energy in a scene is conserved.
- Therefore, there is an energy balance that accounts for all the light that radiates from sources and is reflected by various surfaces in the scene.
- A radiosity calculation thus requires the solution of a large set of equations involving all the surfaces.

Global Illumination

- A ray tracer is best suited to a scene consisting of highly reflective surfaces, whereas a radiosity renderer is best suited for a scene in which all the surfaces are perfectly diffuse.