M30242 – Graphics and Computer Vision

Lecture 08: Lighting & Materials

Overview

- The shading problem.
- Light sources and their properties.
- Global lighting vs. local lighting.
- Phong lighting/reflection model and surface materials.
- Use lighting in WebGL.

The Shading Problem

- Shading is the method/process of determining the shade/colour of a pixel/fragment.
- Shading is not a trivial problem, at least in practice. Realistic shades/colours demand for accurate simulation of the interactions between light rays and surfaces of objects.
- Any solution for the shading problem must provide visually acceptable and computationally feasible approximation to the following properties or processes:
  - the properties of various light sources, e.g., sun light, light bulbs, etc.
  - the properties of surfaces of different material types, i.e., how different materials interact with lights – the lighting models
  - Methods for evaluating the shades of each pixels (fragments) – shading algorithms (next lecture).

Cont’d

- A few decades’ research and hardware development have produced some acceptable solutions to the shading problem.
- The solutions normally consist of a lighting model (this lecture) and a shading model (next lecture):
- The lighting model describes how light rays interact with a tiny **flat** surface patch and determines the amount of reflection of light.
  - The reflected light is what we see (therefore needs to be drawn/rendered). If a surface does not reflect light, it is simply invisible.
  - The Phong lighting model is used in various CG packages and applications

Light Sources

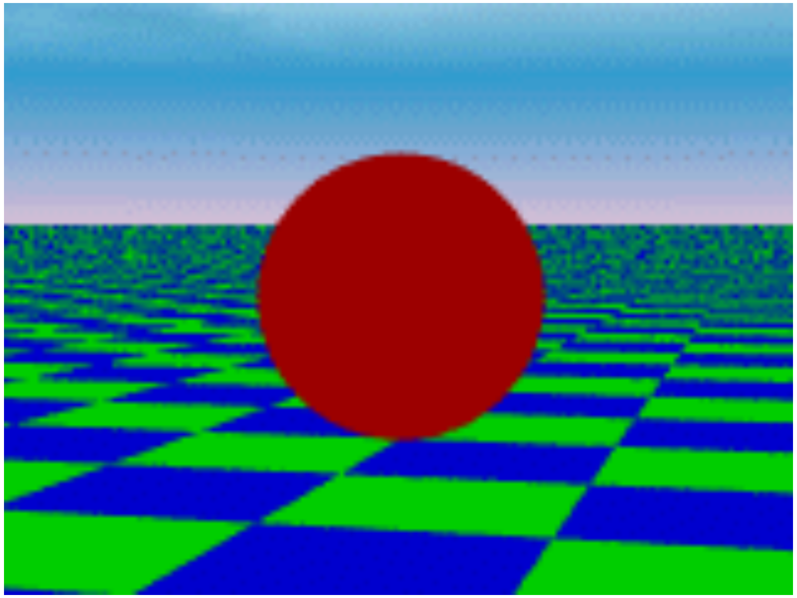
- Various light sources are used in CG to produce different visual effects, e.g., sun light produce daylight effect, point light produces effect a lump, etc
- Most of the lights in CG have their physical counterparts, e.g., point light and a light bulb, but some do not, e.g., the ambient light.
- A light source has a number of properties
  - Colour – spectral frequencies,
  - Intensity – radiance,
  - Geometry (position and direction), and
  - Directional attenuation – intensity distribution.

Ambient Light Source

- Ambient light exists in almost every scene:
  - Objects not directly lit are typically still visible, e.g., the ceiling in this room, undersides of desks.
  - This is the result of indirect illumination – the lights bouncing off intermediate surfaces.
- But ambient light is computationally too expensive to simulate.
- Therefore, a **fictional** ambient light source is invented to account for the effect of scattered light in a scene:
  - No spatial or directional characteristics (therefore, no highlight)
  - Illuminates all surfaces with equal intensity, $I_{ambient}$

An Example

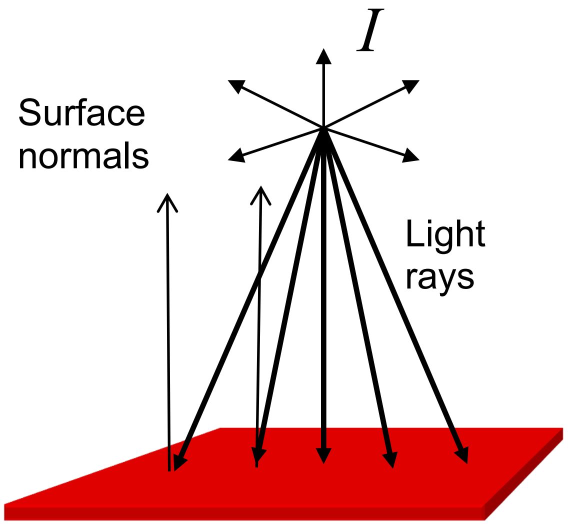
- A scene lit only with an ambient light source
  - Make the scene visible, but
  - Chalky scene, no highlight



The default light in most CG software and APIs

Point Light Sources

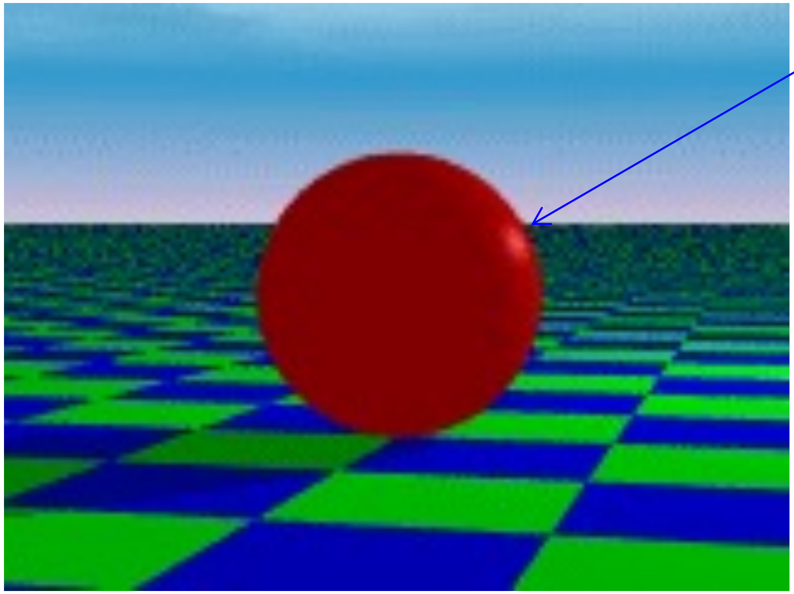
- A point light source emits light equally (i.e., the same intensity) in all directions from a single point.
- It is characterised by:
  - A location/position, and
  - An intensity (radiance).
- The directions (angles) of the light rays with respect to the surface normals vary.



- Light bulbs at some distance away are typical point lights.

Point Light Sources

- Using an ambient and a point light source:

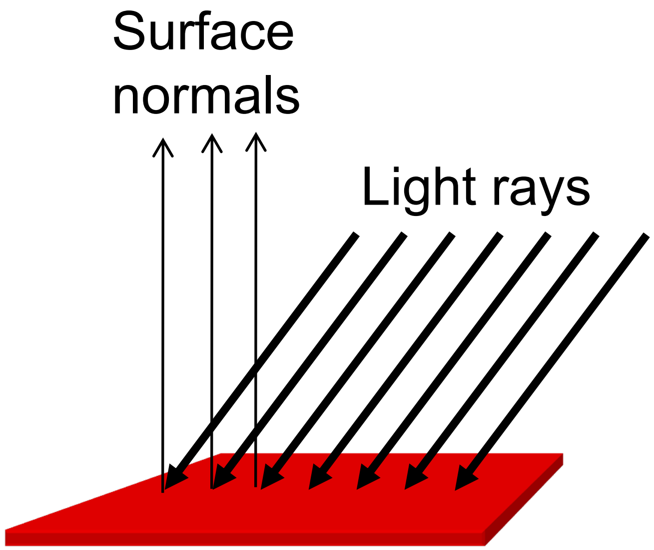


Because the light rays of a point light have directions, a point light will cause highlight.

Directional Light Sources

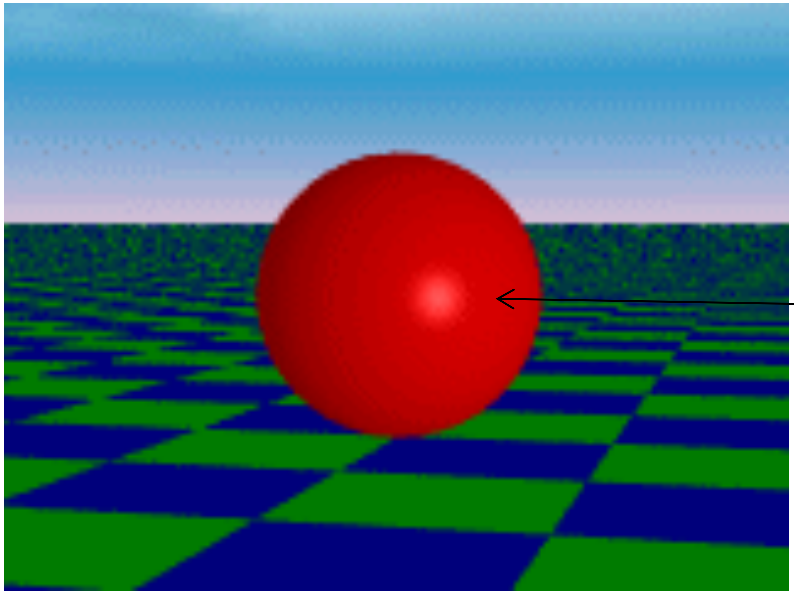
- For a directional light source, all light rays are parallel.
- It is characterised by
  - an intensity, and
  - a direction (with respect to the surface normal).

- A point light **at infinity** (or far enough) could be regarded as a directional light source, e.g., sunlight



Example

- The same scene lit with a directional AND an ambient light source.

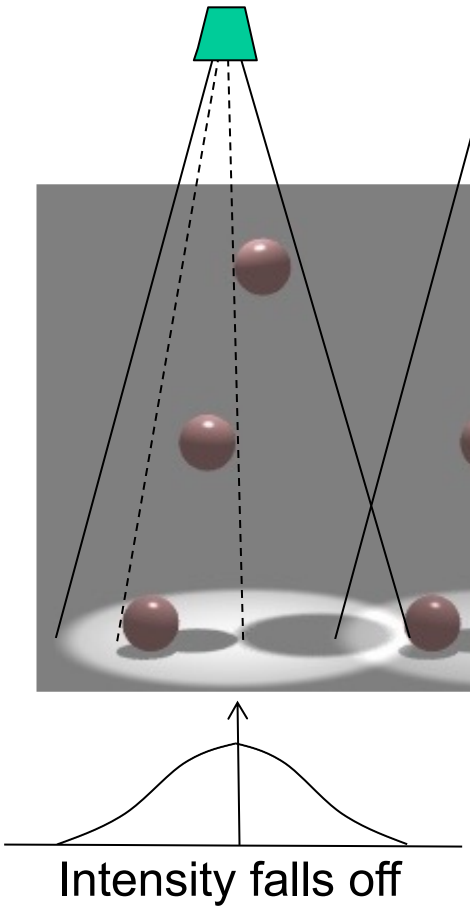


It induces highlight

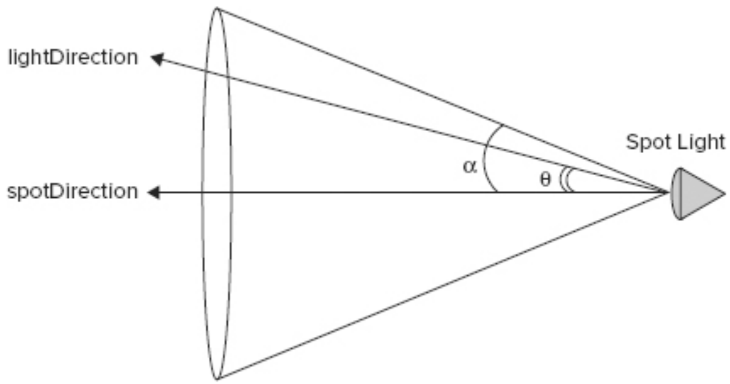
Spotlight

- Spotlights are point lights whose intensity falls off directionally.
- It is characterised by a light cone that has
  - a position (of the apex of the cone),
  - a direction of the main axis,
  - a cutoff angle,
  - a falloff function.

A scene lit by two spot lights



Falloff Function

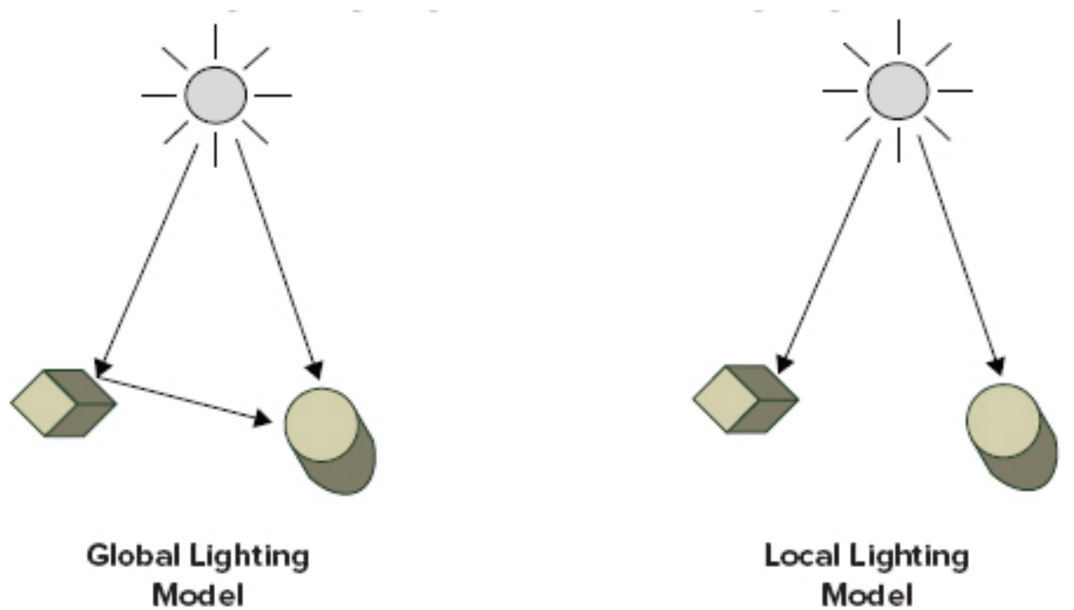


- A point outside the light cone, which is represented by the cutoff angle α, is not lit by the spotlight.
- Inside the light cone, the intensity of light falls off gradually as the angle between the main axis of the cone and the light ray, θ, increases.
- A *falloff function* is used to describe this change. A commonly used function is to let the falloff follow
- $$I(\theta) = \cos^a(\theta)$$
- where a is the falloff exponent. A large a makes the intensity decrease faster as the angle θ increases. If we use two unit-vectors $spotDirection$ and $lightDirection$ to represent the directions of main axis and the light ray in question, we have

$$I(\theta) = (spotDirection \bullet lightDirection)^a$$

Lighting

- Lighting is about how lights illuminate the scene objects.
- When simulating lighting in 3D graphics, one can choose to use one of two different types of lighting: local lighting and global lighting.



Global Lighting

- Global lighting considers the lights from the light sources **AND** the lights reflected from the objects in scene. E.g., a light source is reflected from an object; the reflected light will illuminate a second object; the light reflected from the second object then illuminates the 3rd object, and so on.
- Example of rendering (i.e., shade calculation) using global lighting are ray tracing and radiosity rendering.
- These techniques try to mimic the complicated behavior of light.
- They can produce a highly realistic scene, but it requires a lot of computing resource to track down the paths of light rays, and therefore not very efficient. Ray-tracing is now the default rendering method in some graphics software, e.g., 3DS Max (it also has the algorithm for radiosity calculation)

Local Lighting

- In local lighting, only the lights that come directly from the light sources are accounted for in shade calculation.
- A property (a defect, actually) of a strict local lighting model is that objects will not block light that hits them. This means that shadows are not automatically created in a (strict) local lighting model.

Lighting/Reflection Models

- When considering lighting and shading issues, we usually focus on a **small and flat** patch of surface. The overall shape of the surface is not very relevant (at this stage).
- The small patch is normally characterised by
  - a location,
  - a direction (represented by its normal), and
  - a reflectance spectrum (i.e., colour of the surface)
  - reflective ability (e.g., matte or gloss)
- The reflectance spectrum and the reflective ability are determined by:
  - the atomic properties of the material (decides the colour), and
  - the micro-structures of surface (the smoothness).
  - The true mechanism underlying light reflection is hard to model/describe.

Cont'd

- In practice, these properties are usually approximated by much simpler reflection functions.
  - the spectral properties of a material are often simplified by assigning a colour to the material, and
  - the reflective property is replaced by a (simpler) relationship between incident intensity (irradiance) and reflective intensity (radiance).
- Different lighting models have been developed/used in computer graphics.

Phong Lighting

- Phong Lighting model (published in 1973) is, arguably, the most popular for its simplicity and the acceptable visual effect it produces.
- Other lighting models are similar and can be seen as the variations of Phong lighting model.
- Phong lighting model breaks the reflection from a facet into three parts :

  Total reflection =

  ***ambient** reflection + **diffuse** reflection + **specular** reflection.*
- By this model, different materials have different combinations of these three parts.
- There is not much theoretical basis for this breakdown, but it produces acceptable results!

- To calculate each of the three reflection components, the model further assumes that the light sources in a scene have three corresponding components:
  - ambient light, I_a
  - diffuse light, I_d and
  - specular light, I_s
- It also assumes that a surface has the following properties that measure its capabilities of reflecting the three types of lights:
  - the ambient reflectivity, k_a ,
  - the diffuse reflectivity, k_d,
  - the specular reflectivity, k_s, and the shininess of a surface, α, which is larger for smooth, mirror-like surfaces.
- With these assumptions, calculation of reflection could be easily done.

Ambient Reflection

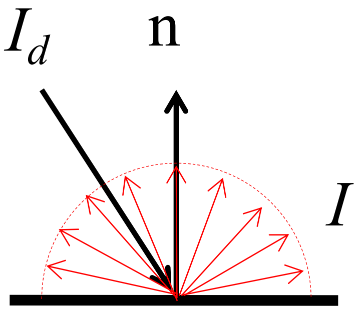
- Assume a global ambient illumination in the scene, I_a
- The ambient light reflected from a surface depends on
  - The surface properties, k_a
  - The intensity of the ambient light source (constant for all points on all surfaces).
  - The reflection function is linear (and simple):

$$I = I_a . k_a$$

- Empirical, no theoretical basis whatsoever.

Diffuse Reflection

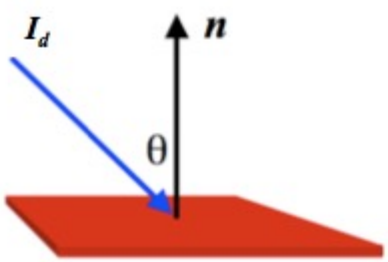
- Rough surfaces (at the microscopic level) reflect light in ALL directions. E.g., chalk and matte surfaces.
- Because of the microscopic variations, an incoming ray of light would be reflected with equal intensity in every direction over the hemisphere.



Lambertian Surfaces

- Ideal diffuse surfaces are called Lambertian surfaces. Chalk and most matte surfaces are very close to ideal diffuse surfaces.
- The reflection from a diffuse surface is calculated according to the *Lambert's cosine law*:

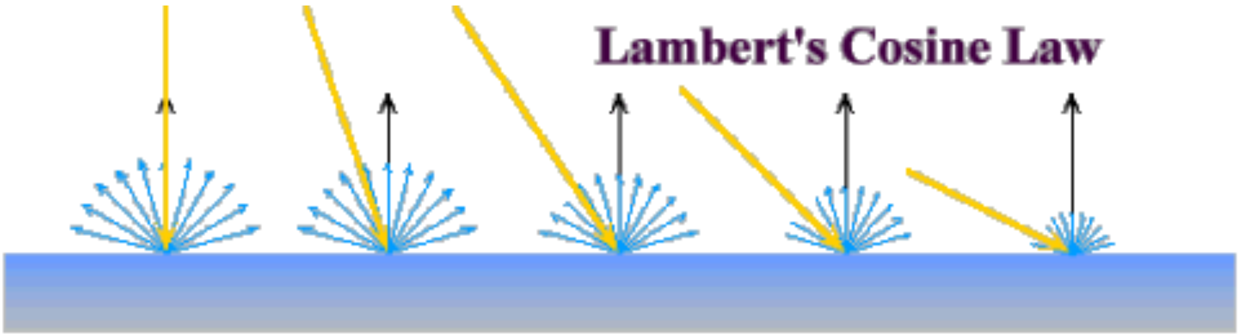
$$I = I_d k_d \cos\theta$$



where

- k_d is the diffuse reflectivity of the material
- I_d is the intensity of the incident light
- θ is the angle between the surface normal $\mathbf{n}$ and the direction of incident of the light ray.

Cont'd

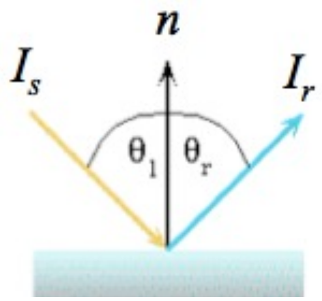


- Given the constant intensity of the incident light, the intensity of reflection varies as the incident angle changes. Also the intensity of reflection is **independent** of the viewing direction (because equal amount is reflected in all directions)

Specular Reflection

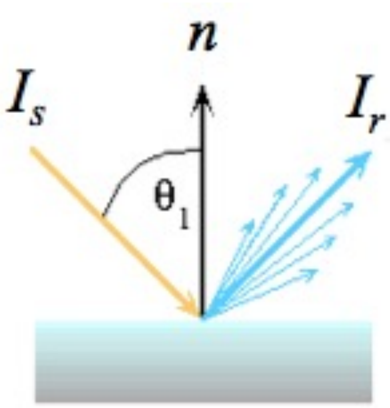
- Shiny surfaces, e.g.,, mirrors, polished metals, glossy car finish, etc., exhibit specular reflection.
- Light rays cast on a specular surface cause a bright spot known as a specular highlight.
- For ideal specular surfaces, specular reflection has these properties:
  - highlight appears as the colour of the light, NOT the colour of the surface,
  - highlight appears in the direction of ideal reflection, which is decided by the incident direction (For ideal specular surface, the direction of ideal reflection, θ_r, equals the incident angle, θ_i, as described by Snell's law),
  - highlight intensity equals the incident intensity, i.e., there is no energy loss in the process of reflection.

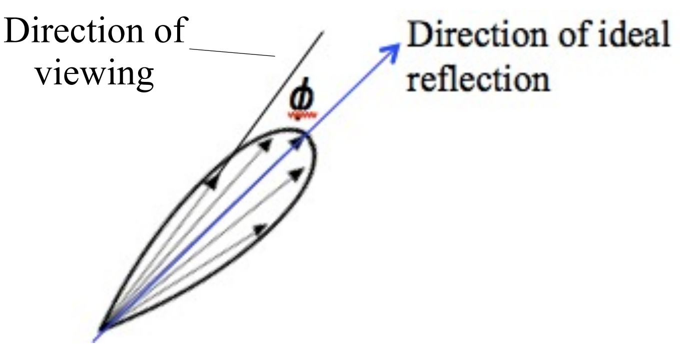
$$I_r = I_s$$



Non-Ideal Specular Surfaces

- Except for mirror-like surfaces, most real-world surfaces are non-ideal, so the highlight appears softer and less defined (i.e., its appearance/visibility is not restricted to the direction of ideal reflection).





- Experiments had shown that most of the reflected light will travel in direction of ideal reflection, but some will go in the directions that are slightly ***off*** the direction of ideal direction. The bigger the off-angle ϕ is, the lower the intensity of reflected light.

Cont’d

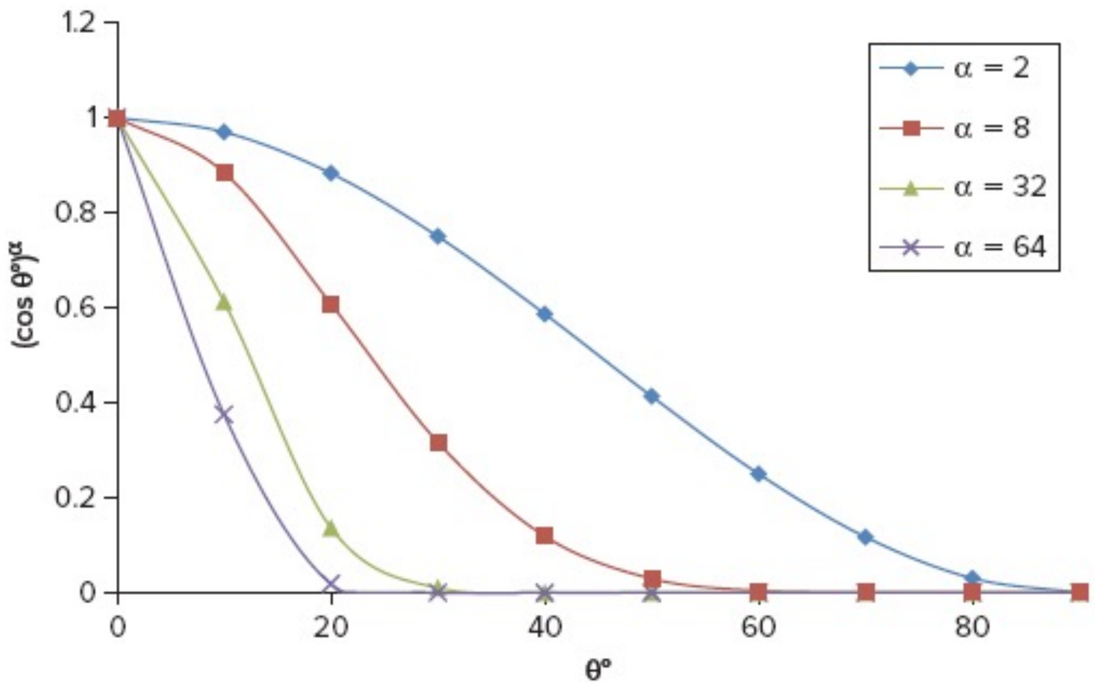
- Such properties of non-ideal specular surfaces can be captured by the formula:

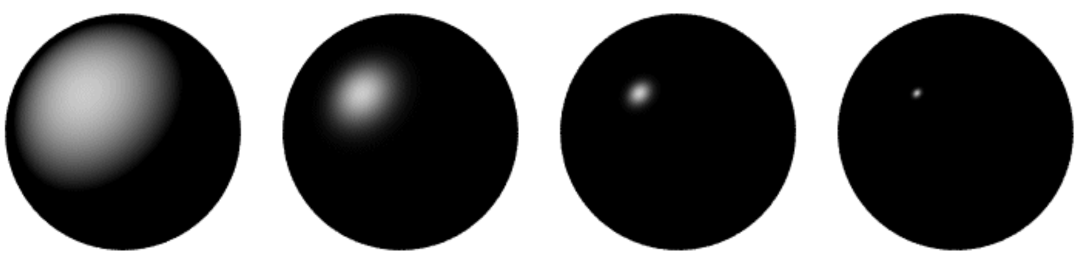
$$I_r = k_s I_s (\cos\phi)^{\alpha}$$

where

- k_s is the specular reflectivity of the material;
- I_s is the intensity of the incident light;
- ϕ is the angle between the direction of ideal reflection and the direction of viewing;
- α is the shininess of the surface.

Shininess





α increases

Put All Together

- Put all three components of reflection together, we have the formula for the Phong Lighting Model

$$I_{total} = \underbrace{k_a I_a}_{\text{Ambient term}} + \sum_{i=1}^{all\_lights} \left(\underbrace{I_{d_i} k_d \cos\theta}_{\text{Diffuse term}} + \underbrace{I_{s_i} k_s (\cos\phi)^{\alpha}}_{\text{Specular term}} \right)$$

Put All Together

$$I_{total} = \underbrace{k_a I_a}_{\text{Ambient term}} + \sum_{i=1}^{all\_lights} \left(I_{d_i} \underbrace{k_d \cos\theta}_{\text{Diffuse term}} + I_{s_i} \underbrace{k_s (\cos\phi)^{\alpha}}_{\text{Specular term}} \right)$$

- In practice, when used in WebGL the model can be further simplified by replacing $k_a I_a$, $k_d I_d$ and $k_s I_s$ by a single colour c_a , c_d and c_s, respectively. That is,

$$I_{total} = c_a + \sum_{i=1}^{all\_lights} \left(c_d \cos\theta + c_s (\cos\phi)^{\alpha} \right)$$

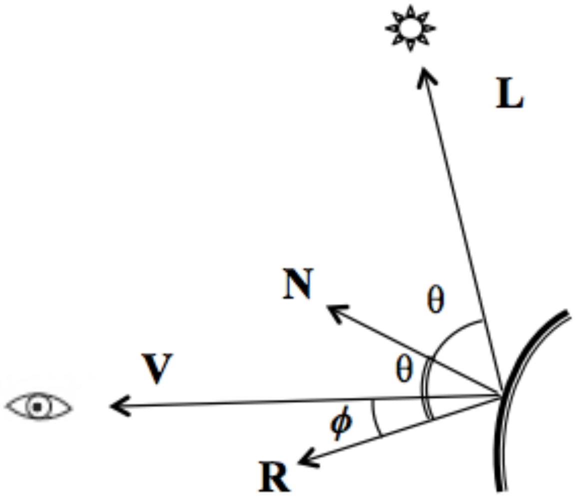
Equation in Vector Form

- Suppose we use **unit vectors** $\mathbf{N}$, $\mathbf{L_i}$, $\mathbf{V}$ and $\mathbf{R}$ to represent, respectively,
  - the surface normal at a point,
  - the direction to the light sources from the point,
  - the direction to the camera/viewer from the point, and
  - the direction of ideal reflection of light rays,

  the formula can be re-written vector form:

$$I_{total} = c_a + \sum_{i=1}^{all\_lights} \left(c_d \cos\theta + c_s (\cos\phi)^{\alpha} \right)$$

$$= c_a + \sum_{i=1}^{all\_lights} \left(c_d \mathbf{L_i} \bullet \mathbf{N} + c_s (\mathbf{V} \bullet \mathbf{R})^{\alpha} \right)$$

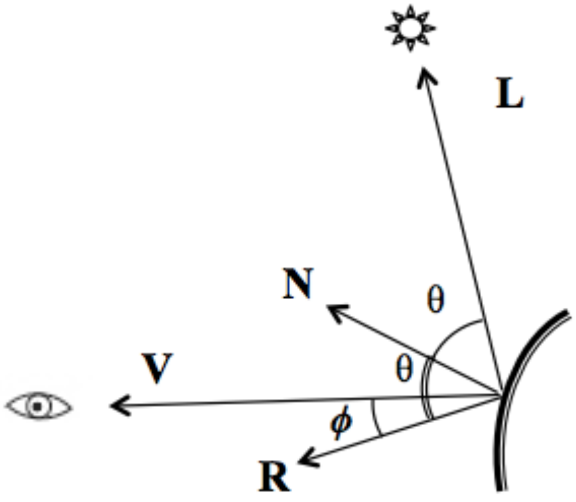


Cont'd

- The unit vector for reflection $\mathbf{R}$ is calculated using

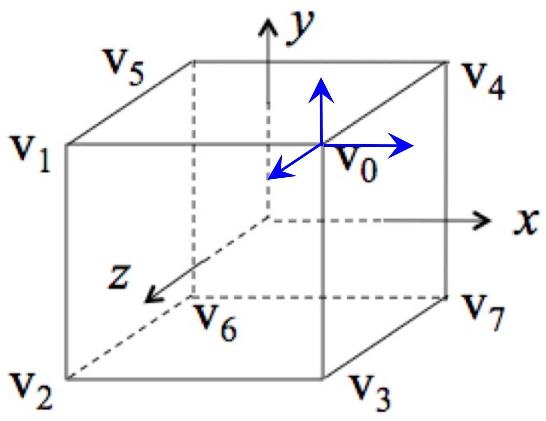
$$\mathbf{R} = 2(\mathbf{L_i} \bullet \mathbf{N})\mathbf{N} - \mathbf{L_i}$$

The OpenGL ES Shading Language has the built-in function `reflect()` for this calculation



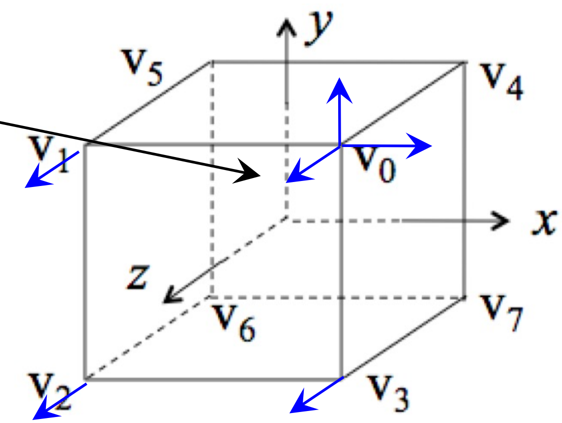
Lighting in WebGL

- Normals at vertices are used.
- Vertices may have multiple normal vectors when they are shared by polygons not in the same plane (e.g., vertex v_0 has three normal vectors).
- Use the normals associated with a polygon for lighting calculation.



Normal Buffer

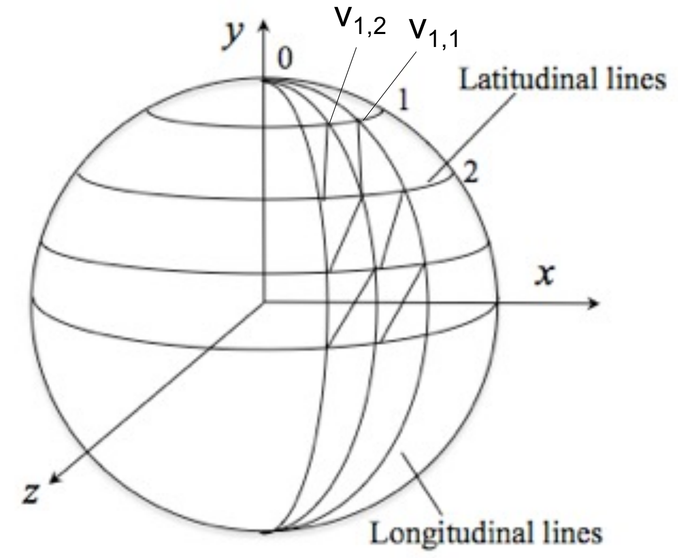
```
pwgl.cubeVertexNormalBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.cubeVertexNormalBuffer);
var cubeVertexNormals = [
     // Front face
     0.0,  0.0,  1.0, //v0
     0.0,  0.0,  1.0, //v1
     0.0,  0.0,  1.0, //v2
     0.0,  0.0,  1.0, //v3

     // Back face
     0.0,  0.0, -1.0, //v4
     0.0,  0.0, -1.0, //v5
     0.0,  0.0, -1.0, //v6
     0.0,  0.0, -1.0, //v7

     . . .
    ];

gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(cubeVertexNormals),
  gl.STATIC_DRAW);
pwgl.CUBE_VERTEX_NORMAL_BUF_ITEM_SIZE = 3;
pwgl.CUBE_VERTEX_NORMAL_BUF_NUM_ITEMS = 24;
```



Coursework: Vertex Normal

- Tessellation (See lecture 4 for detail): the values of θ and φ for a vertex at ***i*th row and *j*th column, $v_{i,j}$**

$$\theta_{i,j}=i\pi/m \quad (i=0,1,\ldots,m)$$
$$\varphi_{i,j}=2j\pi/n \quad (j=0,1,\ldots,n)$$



Coursework: Vertex Normal

- The formula for vertex coordinates of the vertex $v_{i,j}$ ($\theta = i\pi/m$ and $\varphi = 2j\pi/n$):

  $x = r \sin\theta \cos\varphi = r \sin(i\pi/m) \cos(2j\pi/n)$

  $y = r \cos\theta = r \cos(i\pi/m)$

  $z = r \sin\theta \sin\varphi = r \sin(i\pi/m) \sin(2j\pi/n)$

- The same formula gives the normal $\mathbf{n}_{i,j}$, at vertex $v_{i,j}$, when the radius is set to r=1:

  $n_x = \sin\theta \cos\varphi = \sin(i\pi/m) \cos(2j\pi/n)$

  $n_y = \cos\theta = \cos(i\pi/m)$

  $n_z = \sin\theta \sin\varphi = \sin(i\pi/m) \sin(2j\pi/n)$

Coursework: Vertex Normal

```
var normalData = [];
for (var i=0; i <= m; i++) {
    for (var j=0; j <= n; j++) {
        //Calculate nx,ny,nz
        normalData.push(nx);
        normalData.push(ny);
        normalData.push(nz);
    }
}
```

Variables in Vertex Shader

```
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat3 uNMatrix; // Pay attention to uNMatrix
                      // it is for transforming normal vectors

uniform vec3 uLightPosition;  // position of point light

uniform vec3 uAmbientLightColor;
uniform vec3 uDiffuseLightColor;
uniform vec3 uSpecularLightColor;

const float shininess = 32.0;

varying vec3 vLightWeighting;
```

Transform Vertex Normals

- Lighting calculation needs to be done in the shaders because it is per-vertex or per-fragment operation.
- Lighting is done after the the position and normal of a vertex are transformed into camera coordinate system (because it decide what the viewer/camera sees).
- For vertex coordinates, we use the `modelview` transformation to transform them into the camera coordinates (as we did in the program).
- But transforming a vector is different from transforming a vertex.
- We cannot get correct result if we apply the `modelview` transformation to a normal vector.

Transform Vertex Normals

- This is because a modelview matrix normally contain non-uniform scale factors (i.e., the diagonal elements have values other than 1s), which will result in the transformed normal vector is no long orthogonal to the underlying surface (i.e., it is no longer a normal vector).
- We can remove the effect of non-uniform scale in the modelview matrix by using the *transpose of the inverse of the modelview transformation* to transform the normals.

$$uNMatrix = (uMVMatrix^{-1})^{T}$$

- The proof of this formula involve simple linear algebra manipulations (next slide for reference)

For Info Only

- Suppose **n** and **p** are the normal and tangent vectors at a point on a surface.
- By definition, the dot product of **n** and **p** equals 0: **n.p**=0
- Written in the form of matrix multiplication: $\mathbf{n}^T\mathbf{p}=0$.
- In vertex shader, we transform the vertex and other coordinate by the modelview matrix **M**. After being transformed, all coordinates are in camera coordinate system.
- Suppose we transform the normal and tangent vectors use the same modelview matrix **M**: **Mn** and **Mp** would be the transformed normal and tangent vectors.
- We want to make sure that after transformation being done, the two vector are still orthogonal.

For Info Only

- By definition, if **Mn** and **Mp** still orthogonal (ie., they still the normal and the tangent of the of the surface), $(M\mathbf{n}).(M\mathbf{p})=0$, i.e., $(M\mathbf{n})^T(M\mathbf{p})=0$, $\mathbf{n}^T M^T M\mathbf{p}=0$.
- To get $\mathbf{n}^T\mathbf{p}=0$, M^TM must equal $\mathbf{I}$, which implies M is an orthogonal matrix (the rows are orthogonal to the columns). A pure rotation transformation (without non-uniform scale) is indeed orthogonal. However, when non-uniform scale are involved, $M^TM \neq \mathbf{I}$. i.e., we cannot transform normal vector by the same modelview transformation.
- But which transformation should be used to transform a normal? Suppose **A** is an unknown transformation. Therefore, by definition, if **n** is still the normal vector, we must have $(A\mathbf{n})^T(M\mathbf{p})=0$, i.e., $\mathbf{n}^T A^T M\mathbf{p}=0$, then $A^TM=\mathbf{I}$. Then $A^TMM^{-1}=\mathbf{I}M^{-1}$, $A^T=M^{-1}$, $A=(M^{-1})^T$

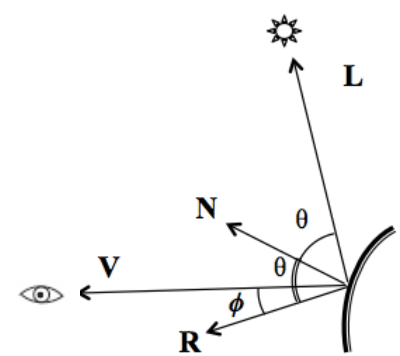
Transform Vertex Normals

- The calculation of $(M^{-1})^T$ is done in the main WebGL using JavaScript and sent to the uniform variable declared in vertex shader, `uNMatrix`:

```
function uploadNormalMatrixToShader() {
  var normalMatrix = mat3.create();
  mat4.toInverseMat3(pwgl.modelViewMatrix,
    normalMatrix);
 mat3.transpose(normalMatrix);
gl.uniformMatrix3fv(pwgl.uniformNormalMatrixLoc,
    false, normalMatrix);
}
```

Vertex Shader: `main()`

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

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

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

  // Calculate (N dot L) for diffuse lighting
  float diffuseLightWeighting = max(dot(normalEye,
                                    vectorToLightSource), 0.0);

  // Calculate the reflection vector (R) for specular light
  vec3 reflectionVector = normalize(reflect(-vectorToLightSource,
                                    normalEye));
```



```
  // The with respect to the camera coordinate system, the camera is at
   (0.0, 0.0, 0.0) pointing along the negative z-axis.
  //Calculate viewVector (v) in camera(eye) coordinates as:
  //      (0.0, 0.0, 0.0) - vertexPositionEye3
  vec3 viewVectorEye = -normalize(vertexPositionEye3);
  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
  vLightWeighting = uAmbientLightColor +
                    uDiffuseLightColor * diffuseLightWeighting +
                    uSpecularLightColor * specularLightWeighting;

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

Fragment Shader

- If there is no texture involved, the calculated colours are used as the fragment colour.
- If texture is used, the calculated light colours are used to modulate the RGB components of texel colour:

```
precision mediump float;
varying vec2 vTextureCoordinates;
varying vec3 vLightWeighting;
uniform sampler2D uSampler;

void main() {
  vec4 texelColor = texture2D(uSampler,
                     vTextureCoordinates);
  gl_FragColor = vec4(vLightWeighting.rgb * texelColor.rgb,
                                 texelColor.a);
}
```

Upload Normal to Shader

- Vertex normal and light attributes are uploaded to the shaders in the usual way:

```
pwgl.vertexNormalAttributeLoc = gl.getAttribLocation(shaderProgram,
                                "aVertexNormal");
gl.enableVertexAttribArray(pwgl.vertexNormalAttributeLoc);

pwgl.uniformLightPositionLoc =gl.getUniformLocation(shaderProgram,
                                "uLightPosition");
pwgl.uniformAmbientLightColorLoc = gl.getUniformLocation
                      (shaderProgram, "uAmbientLightColor");
pwgl.uniformDiffuseLightColorLoc = gl.getUniformLocation
                      (shaderProgram, "uDiffuseLightColor");
pwgl.uniformSpecularLightColorLoc = gl.getUniformLocation
                      (shaderProgram, "uSpecularLightColor");
```

- Lighting attributes, such as light colours, can be set directly in the shader if these properties do not change on an object to object basis.

Further Reading

- Anyuru, A., WebGL Programming – Develop 3D Graphics for the Web
  - Chapter 7