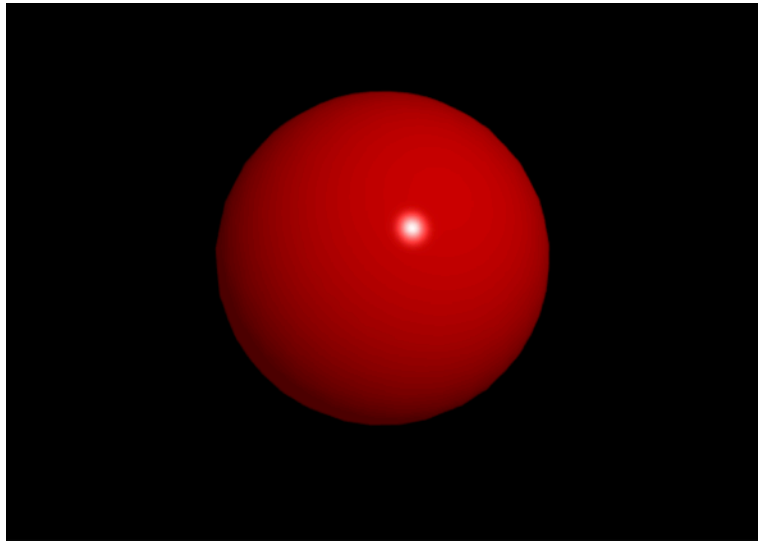
Visual Computing I:

Interactive Computer Graphics and Vision

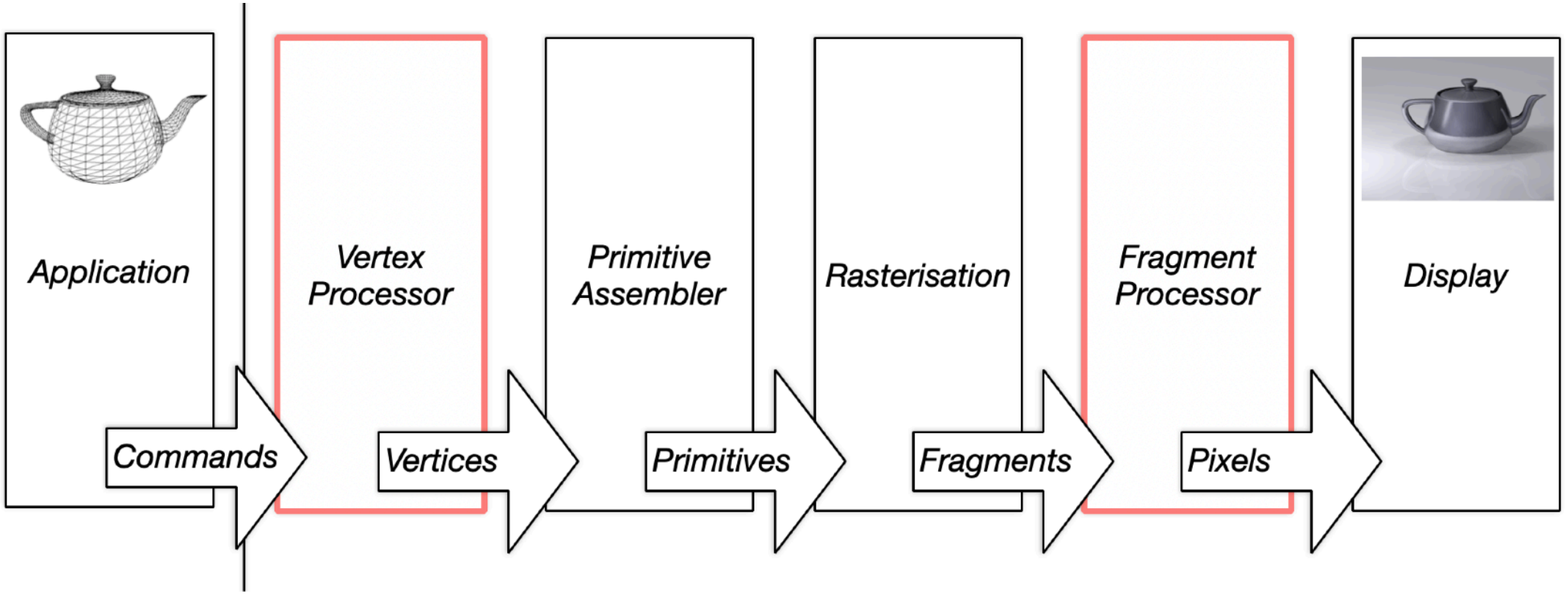




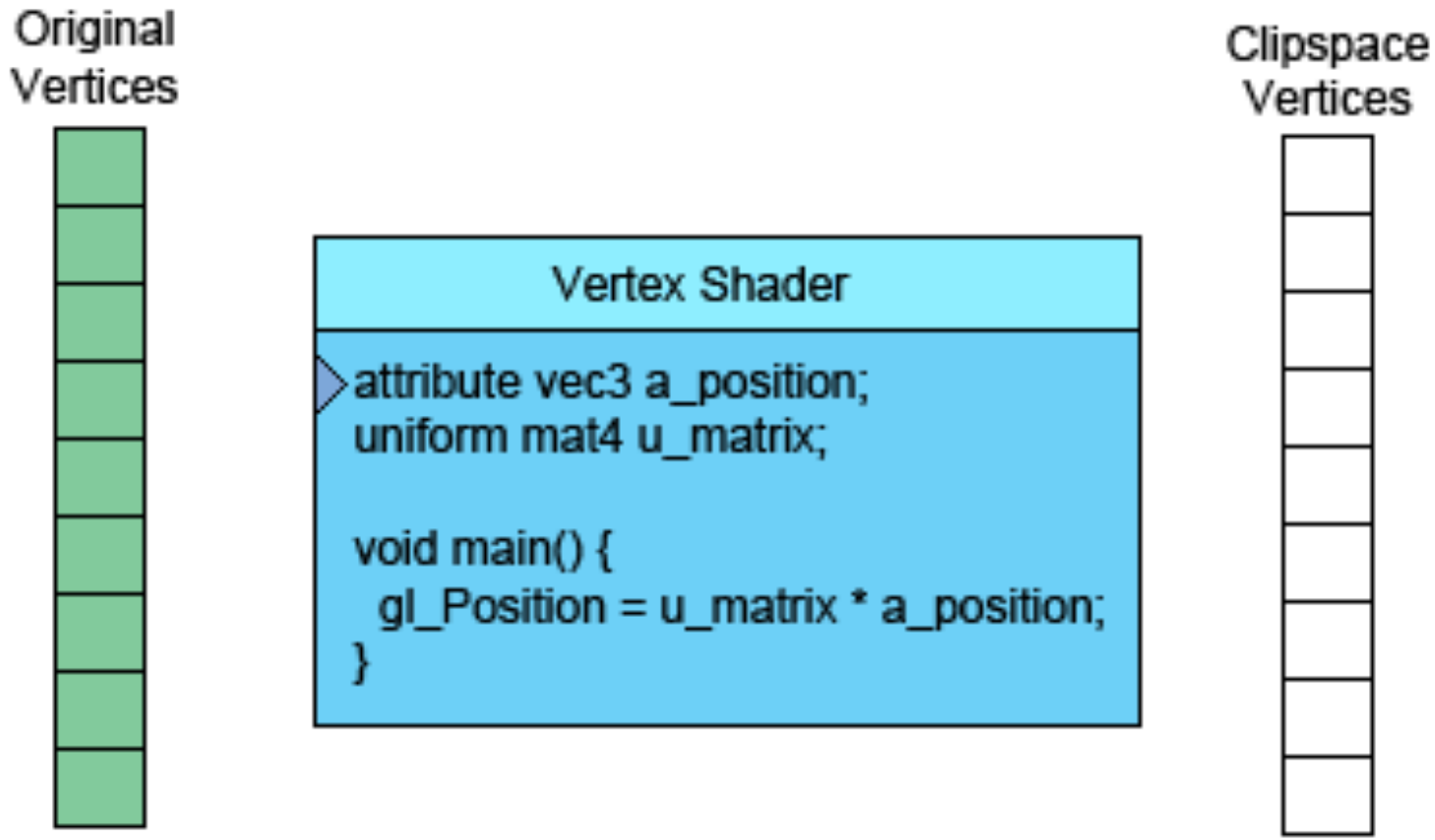
Illumination

Stefanie Zollmann and Tobias Langlotz

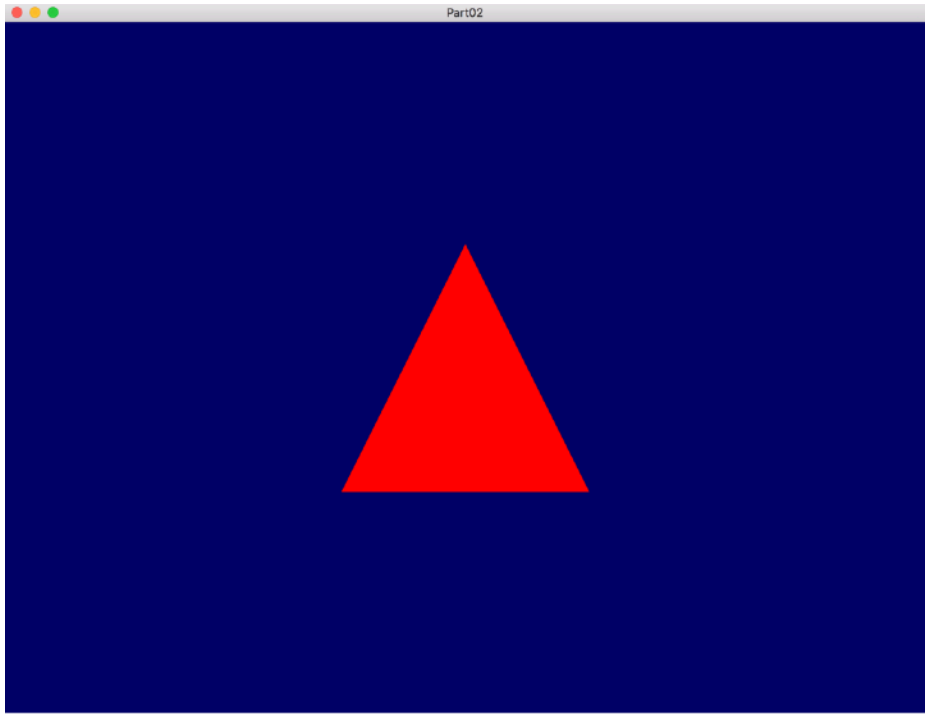
Recap Last Lecture



Vertex Shader



Vertex Shader



camera.position
vec3(0,0,5)

camera.lookat
vec3(0,0,0)

Vertex Shader

```
   //position
   glm::vec3 position = m_camera->getPosition();
   // Up vector
   glm::vec3 up = glm::cross( right, direction );
   // set camera's lookat
   m_camera->setLookAt(position,position+direction,up );

  //in camera class definition
  m_viewMatrix       = glm::lookAt(
                                   m_position,           // Camera is here
                                   m_lookat, // and looks here : at the same position, plus "direction"
                                   m_up                  // Head is up (set to 0,-1,0 to look upside-down)
                                   );

//in shader class - passing to vertex shader
void Shader::updateMVP(glm::mat4 MVP){
   glUniformMatrix4fv(m_MVPID, 1, GL_FALSE, &MVP[0][0]);
}
```

Camer

camera.position
vec3(0,0,5)

camera.lookat
vec3(0,0,0)

Example combining Shader.cpp and redTriangle.cpp

Vertex Shader

Simple vertex shader apply the model view project transformation:

```
// Pass MVP to shader - in cpp file
glm::mat4 MVP;
GLint m_MVPID = glGetUniformLocation(programID, "MVP");
glUniformMatrix4fv(m_MVPID, 1, GL_FALSE, &MVP[0][0]);
```

```
#version 330 core

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;

// Values that stay constant for the whole mesh.
uniform mat4 MVP;
void main(){
    gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
}
```

Example combining Shader.cpp and basicShader.vert

Fragment Shader

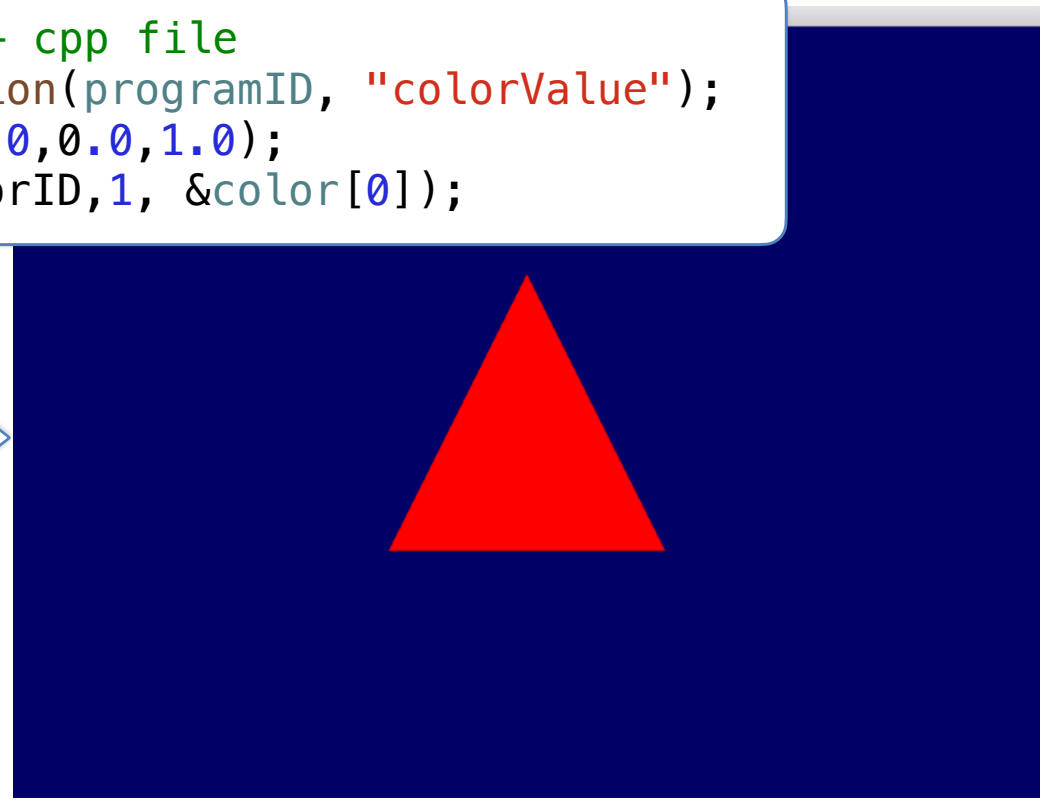
Example combining ColourShader.cpp and basicShader.frag

Simple Fragment Shader outputting gl_FragCoord:

```
#version 330 core

// Ouput data
out vec3 color;

uniform vec4 colorValue;
void main()
{
  // Output color = red
  color = colorValue.rgb;

}
```

basicShader.frag

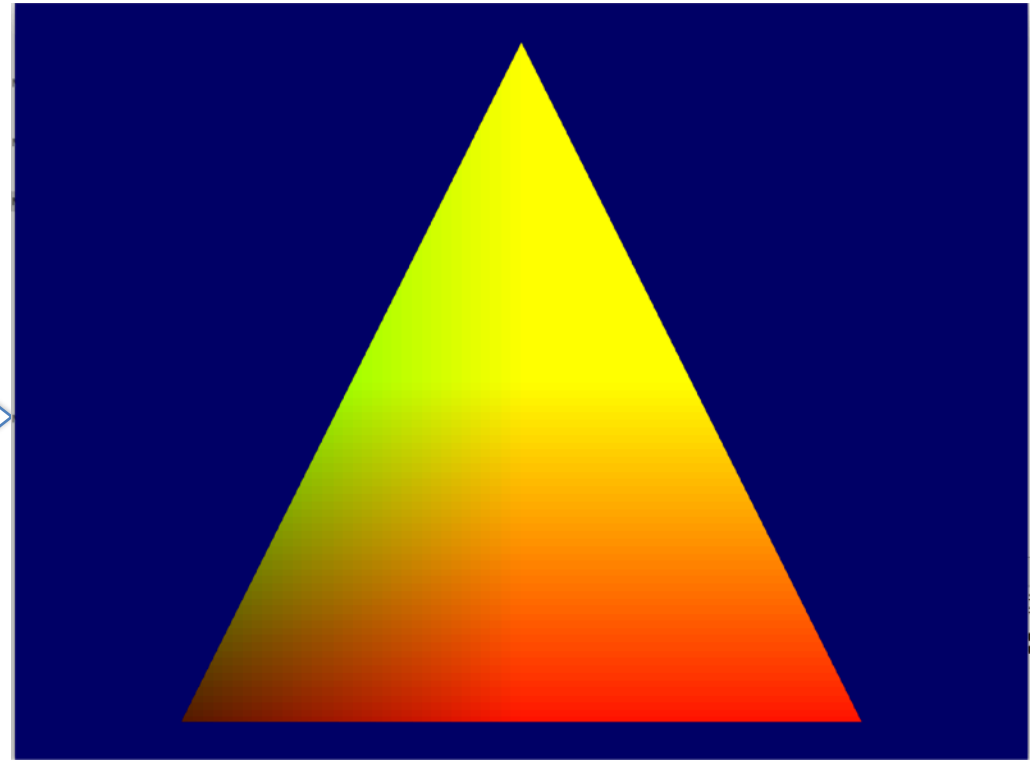
```
// add color parameter to shader - cpp file
GLint colorID = glGetUniformLocation(programID, "colorValue");
glm::vec4 color = glm::vec4(1.0,0.0,0.0,1.0);
glProgramUniform4fv(programID,colorID,1, &color[0]);
```



Fragment Shader

Simple Fragment Shader outputting gl_FragCoord:

```
#version 330 core

// Ouput data
out vec3 color;

void main()
{
  // Output color = screen coord
          color =  vec3(gl_FragCoord.r/1024,
          gl_FragCoord.g/768, 0.0);

}
```



https://registry.khronos.org/OpenGL-Refpages/gl4/html/gl_FragCoord.xhtml

Datatypes GLSL

- Basic Types
  - int, uint, float, bool: scalar numeric and logical types
  - ivec2, ivec3, ivec4: integer vectors
  - uvec2, uvec3, uvec4: unsigned integer vectors
  - bvec2, bvec3, bvec4: boolean vectors
- Floating-Point Vectors
  - vec2, vec3, vec4: 2D/3D/4D float vectors
  - Commonly used for positions, colors, and directions
- Matrices
  - mat2, mat3, mat4: 2×2, 3×3, 4×4 float matrices
  - Mixed forms also exist: mat2x3, mat3x4, etc.
  - Used for transforms (model, view, projection)

Datatypes GLSL

- Sampler and Image Types
  - sampler2D, samplerCube, sampler2DShadow, etc.
  - Represent textures bound to shader units
  - Accessed via functions like texture()
- Special Types
  - struct: custom user-defined types
  - array: fixed-size arrays of any GLSL type
  - in, out, uniform qualifiers specify data flow between stages
- Precision Qualifiers (ES / optional in desktop)
  - highp, mediump, lowp: control numeric precision

GLSL Built-in Variables

- Predefined variables automatically provided by the OpenGL shading pipeline
- Used to communicate data between pipeline stages (vertex -> fragment)
- Exist in addition to user-defined inputs/outputs (in, out, uniform)
- Purpose:
  - Give access to system-generated information (e.g., vertex index, fragment coordinates)
  - Let shaders output mandatory data (e.g., vertex position or fragment color

Vertex Shader Built-in Variables

- Purpose:
  - Process each vertex to produce a clip-space position
  - Pass attributes (like normals or UVs) to the fragment shader
- Examples:

`gl_Position`	`vec4`	out	Clip-space position — must be written.
`gl_PointSize`	`float`	**out**	Controls size of point primitives.
`gl_VertexID`	`int`	**in**	Index of the current vertex.

Vertex Shader Built-in Variables

- Purpose:
  - Process each vertex to produce a clip-space position
  - Pass attributes (like normals or UVs) to the fragment shader
- Examples:

```
void main() {
    gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4(position, 1.0);
    gl_PointSize = 4.0;
}
```

Fragment Shader Built-in Variables

- Purpose:
  - Compute the final colour and depth of each fragment (potential pixel)
  - Access screen-space or geometric information
- Examples:

gl_FragCoord	vec4	in	Window-space coordinates (x, y, z, 1/w).
gl_FrontFacing	bool	in	true for front-facing triangles.
gl_FragDepth	float	out	Overrides default depth value (optional).

Fragment Shader Built-in Variables

- Purpose:
  - Compute the final colour and depth of each fragment (potential pixel)
  - Access screen-space or geometric information
- Examples:

```
void main() {
    float depth = gl_FragCoord.z;
    fragColor = vec4(vec3(depth), 1.0);
}
```

3-Min Discussion:

What Makes a Shader Expensive? Two fragment shaders produce the same visual result, one runs at 60 FPS, the other at 20 FPS. What could cause this difference?

Example Shader

```
void main()
{
    vec4 color = texture(diffuseTex, TexCoords);

    // Multiple trigonometric operations per fragment
    for (int i = 0; i < 20; i++) {
        color.rgb += vec3(
            sin(time * 0.1 * float(i)) * color.r,
            cos(time * 0.2 * float(i)) * color.g,
            tan(time * 0.05 * float(i)) * color.b
        );
    }

    // Expensive normalization and power operations
    color.rgb = normalize(pow(color.rgb, vec3(2.2)));

    FragColor = color;
}
```

Shader Optimisation

- Avoid expensive math operations: replace pow(), tan(), or repeated normalize() calls with simpler or approximate versions
- Minimise loops and branching: keep shader control flow uniform across fragments to avoid divergence
- Precompute and reuse values: move repeated calculations to the CPU or vertex shader when possible
- Reduce overdraw: render opaque objects front-to-back or use depth testing to skip hidden fragments
- Optimise for the GPU’s strengths -> parallelism, coherence, and simple arithmetic

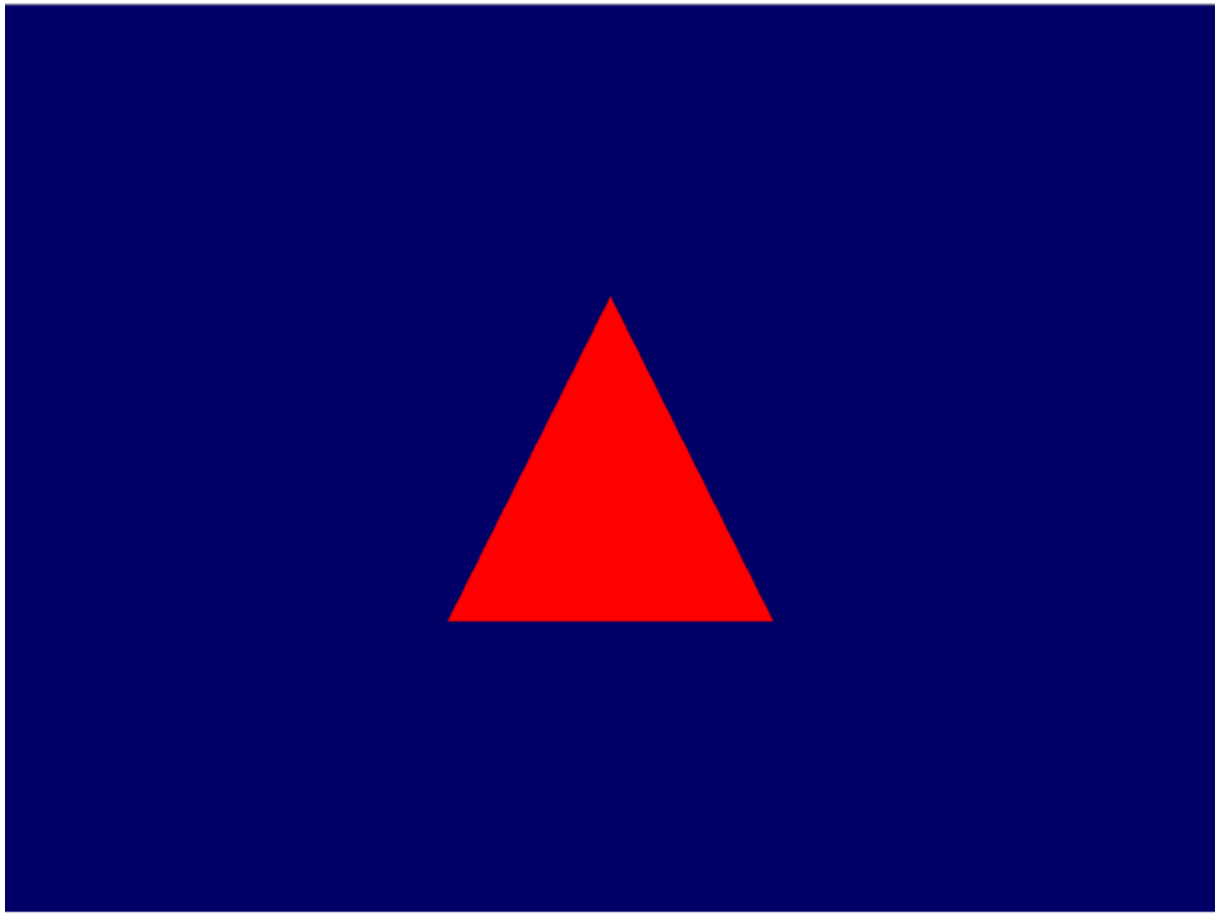
Additional Shader Stages in the OpenGL Pipeline

- Beyond the basic Vertex Shader and Fragment Shader, OpenGL includes optional programmable stages for finer control over geometry

- Tessellation Control Shader (TCS):
  - Defines how much a patch should be subdivided, enabling adaptive surface detail
- Tessellation Evaluation Shader (TES):
  - Computes vertex positions of the tessellated patch, allowing smooth curved surfaces
- Geometry Shader:
  - Operates after vertex processing and can generate, modify, or discard entire primitives (e.g., expand points into quads, create outlines).
- These stages provide greater flexibility for procedural geometry, LOD, and displacement mapping

- When unused, the pipeline runs directly from the vertex shader to the rasteriser for efficiency

What is missing?



Why Illumination?

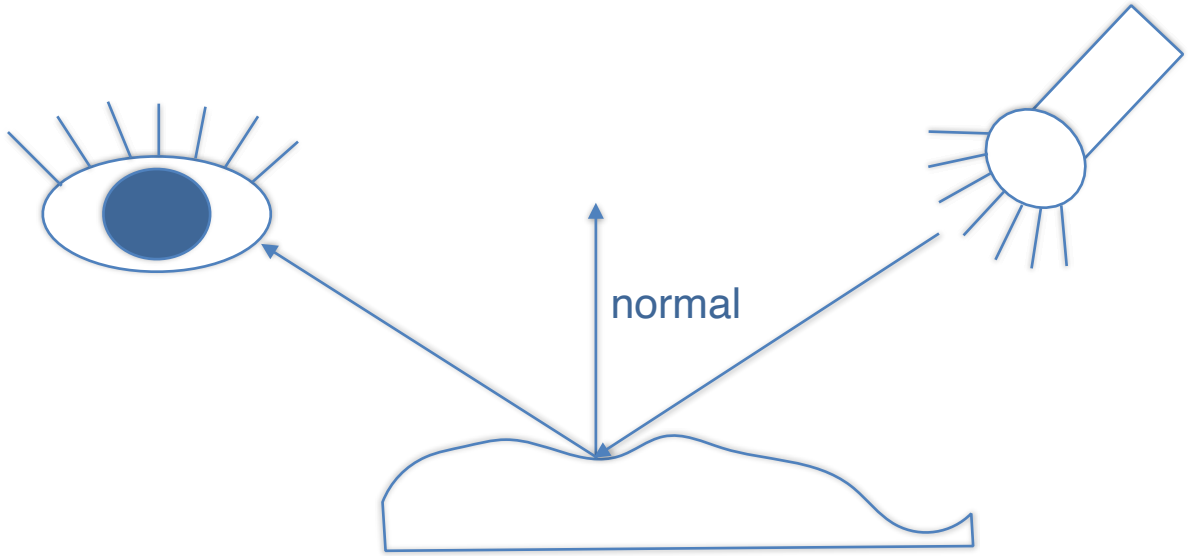
- Illumination is important for perception and understanding of 3D scenes
- Has visual cues for humans
- Provides information about
- Positioning of light sources
- Characteristics of light sources
  - Materials
  - Viewpoint

Illumination Model

- Can be complex
- Equation for computing illumination
- Includes:
  - Light attributes (intensity, colour, position, direction, shape)
  - Surface attributes (colour, reflectivity, transparency)
  - Interaction between lights and objects
- General rendering equation
  - Introduced 1986 by Kajiya
  - Global illumination model

Local Illumination Model

- OpenGL cannot render full global illumination
- We need a simplified approximation
- Local illumination model
- Does not consider light reaching after bouncing off other objects
- Function of:
  - Viewer position
  - Light source
  - Surface material properties
  - Geometry



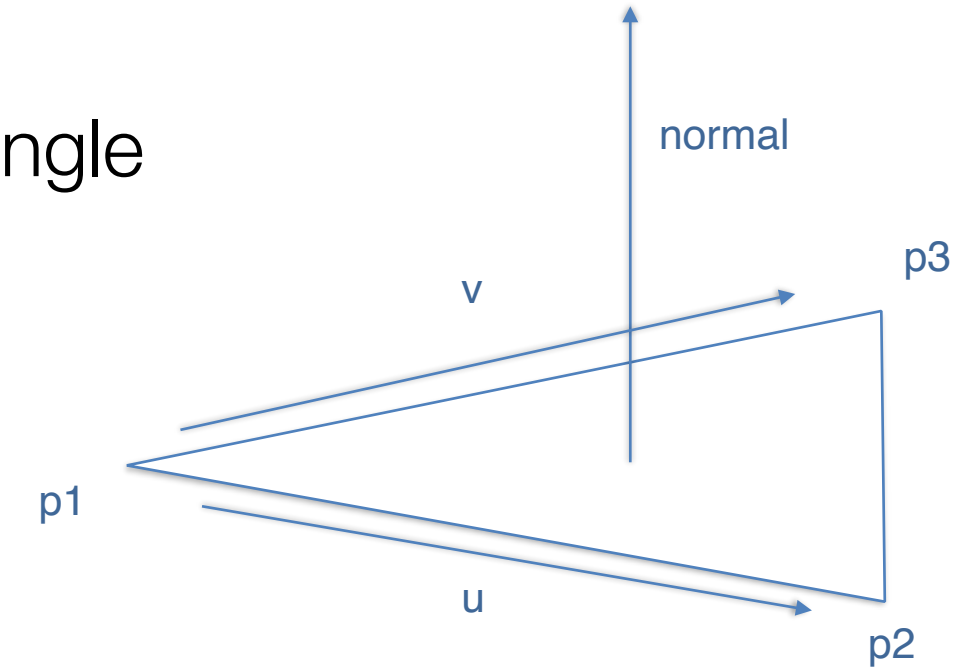
Reminder: Normal

Perpendicular to tangent plane of surface

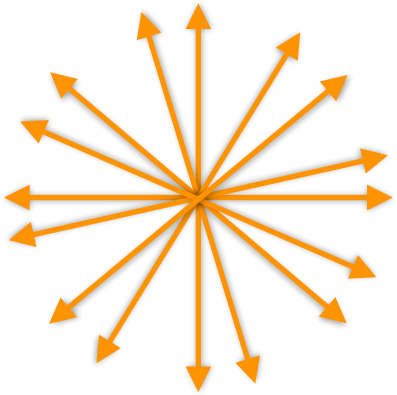
For triangles:

Cross product of two edges of that triangle

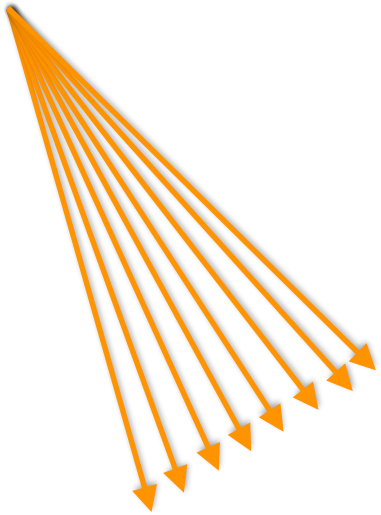
$$n = u \times v$$
$$u = p_2 - p_1$$
$$v = p_3 - p_1$$
$$n_x = u_y v_z - u_z v_y$$
$$n_y = u_z v_x - u_x v_z$$
$$n_z = u_x v_y - u_y v_x$$



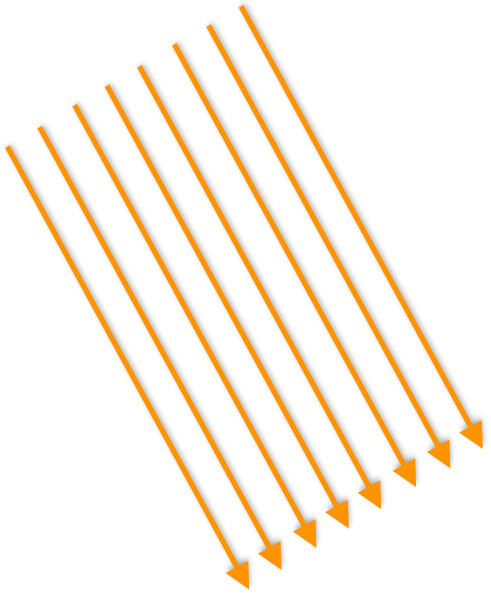
Light Sources



Point Light



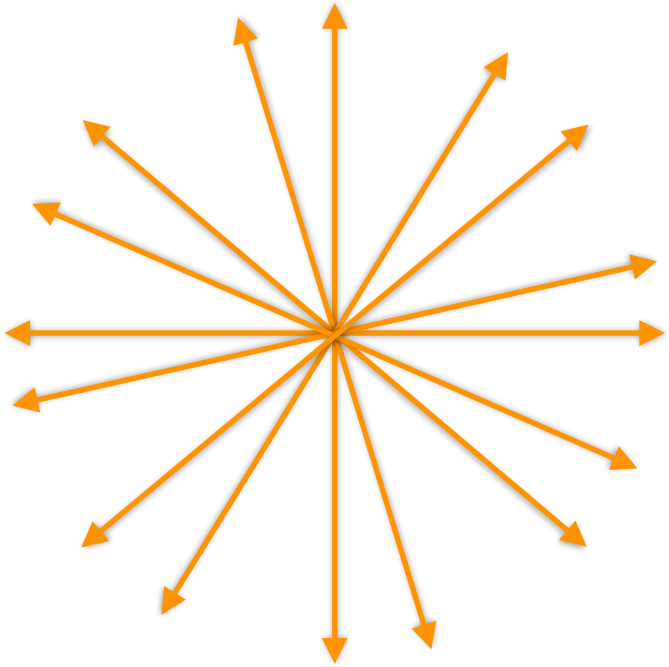
Spot Light



Directional Light

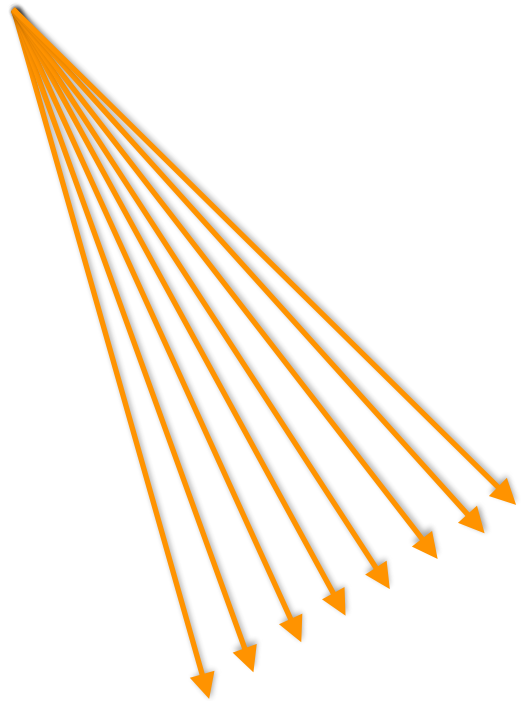
Point Light

- Starts at one point and spreads out in all directions
- Defined by position
- Intensity decreases with the square of distance
- Direction is different at each vertex (light direction = light position – vertex position)
- Example: light bulb



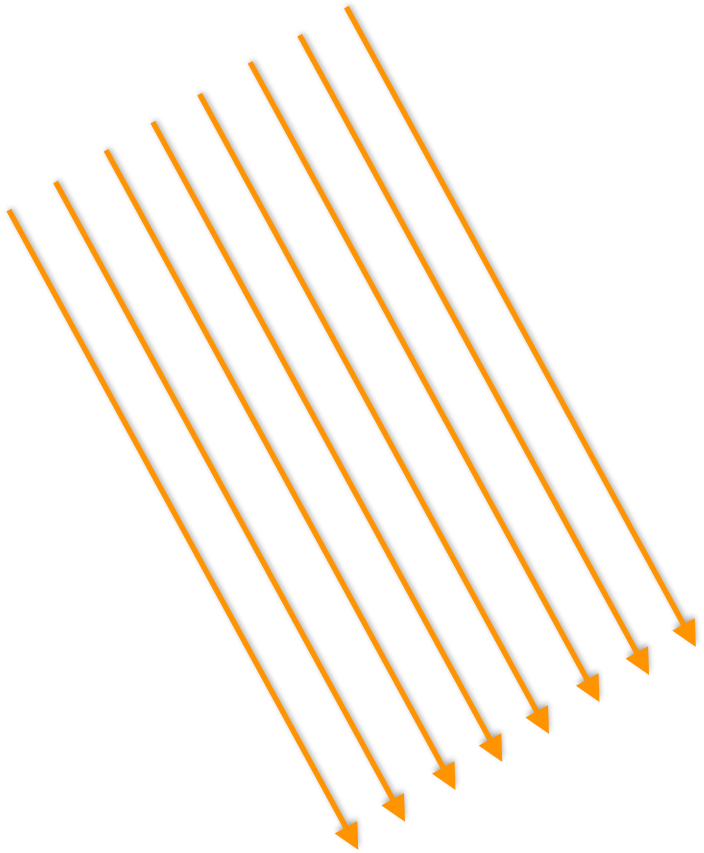
Spot Light

- Light starts at one point and spreads out as cone with defined angle
- Described by position, direction and width of beam
- Useful for dramatic light effects (e.g. theatre spot light)

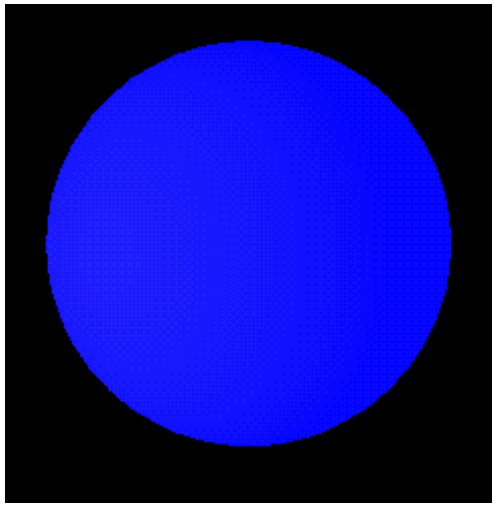


Directional Light

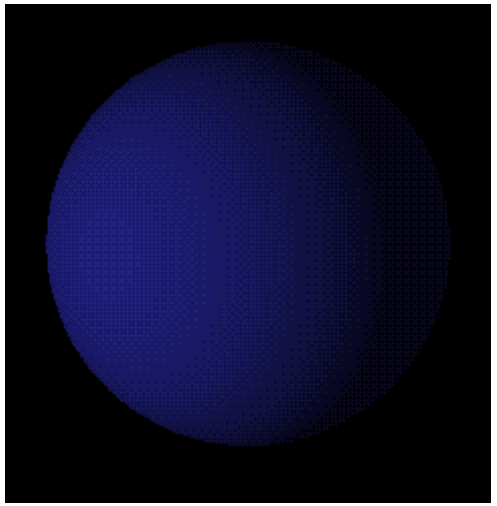
- Described by direction only
- No position
- Direction is same for all points
- Used for light sources that are infinitely far away
- Intensity does not change depending on distance
- Used for modelling sun light



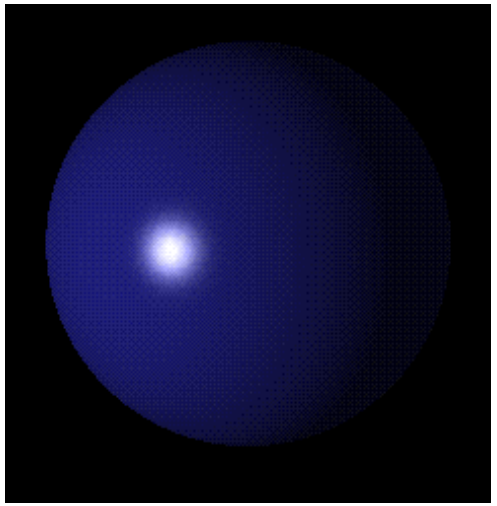
Reflection model



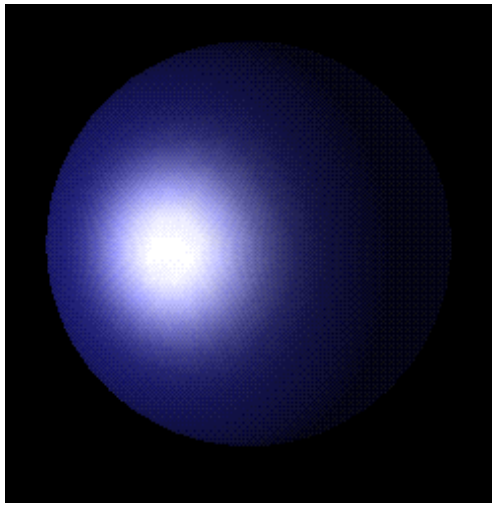
Ambient



Diffuse



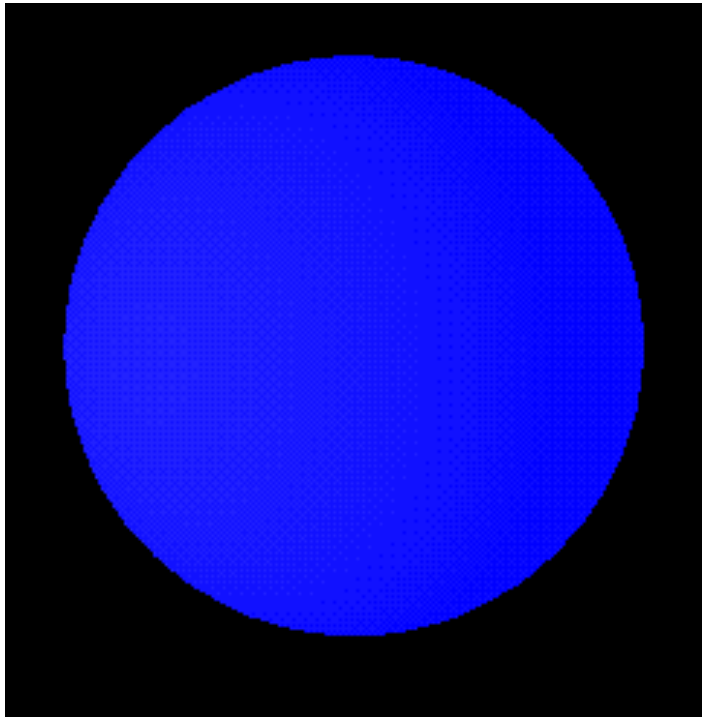
Specular



Combined

Ambient Component

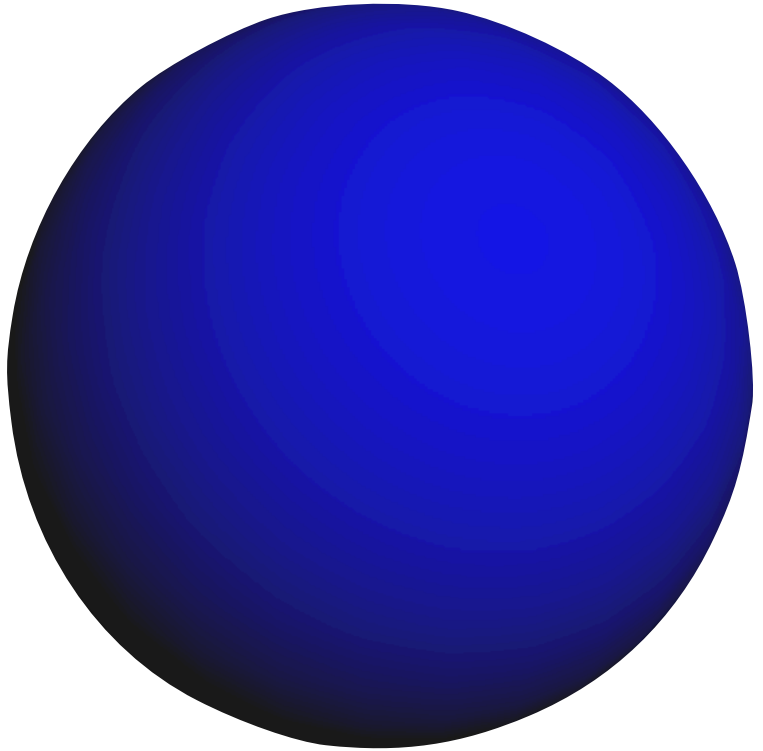
- Indirect illumination from light that has been reflected multiple times
- Does not come from a specific direction
- “Base” lighting
- Consists of:
  - Ambient light component Ia
  - Ambient material factor ka



$I_{ambient} = I_a k_a$

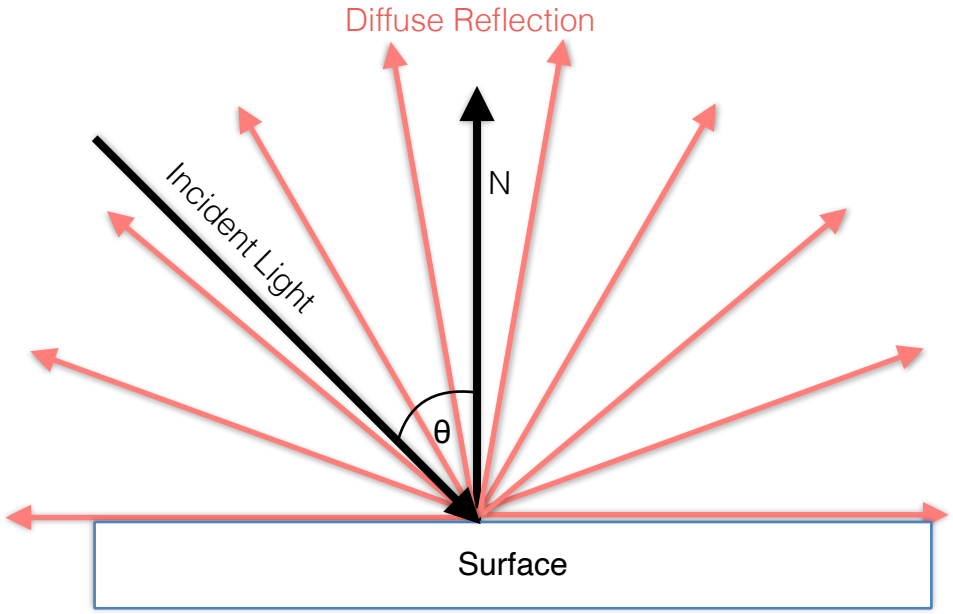
Diffuse Component

- Also called Lambertian reflection
- Ideal diffuse surface reflects light equally in all directions
- Incident ray is reflected in many directions
- Independent of view angle (reflects equally in all directions)
- But dependent on direction of incoming light (angle between normal N and incident light L : angle of incidence θ)



Diffuse Component

- Also called Lambertian reflection
- Ideal diffuse surface reflects light equally in all directions
- Incident ray is reflected in many directions
- Independent of view angle (reflects equally in all directions)
- But dependent on direction of incoming light (angle between normal N and incident light L : angle of incidence θ)

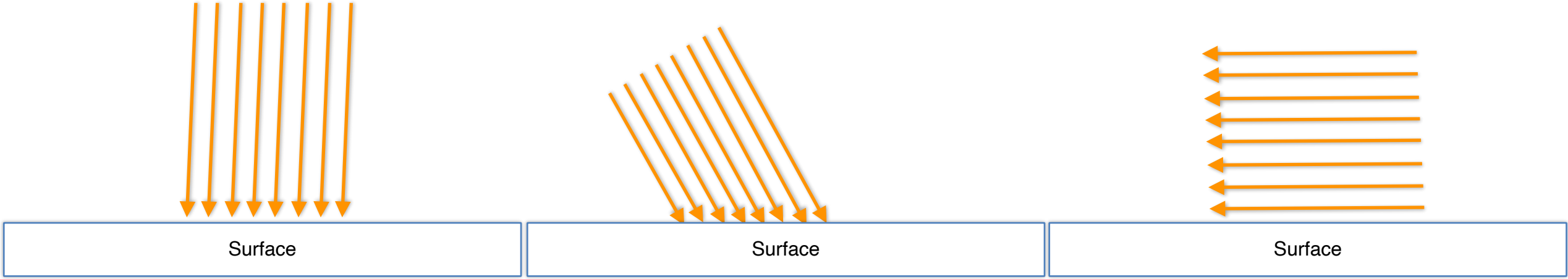


Diffuse Component

- Incoming light rays with perpendicular angle to the surface reflect more light
- The larger the angle θ between normal and incoming light rays, the less light is reflected

Diffuse Component

- Incoming light rays with perpendicular angle to the surface reflect more light
- The larger the angle θ between normal and incoming light rays, the less light is reflected



θ = 0°: Maximum Brightness

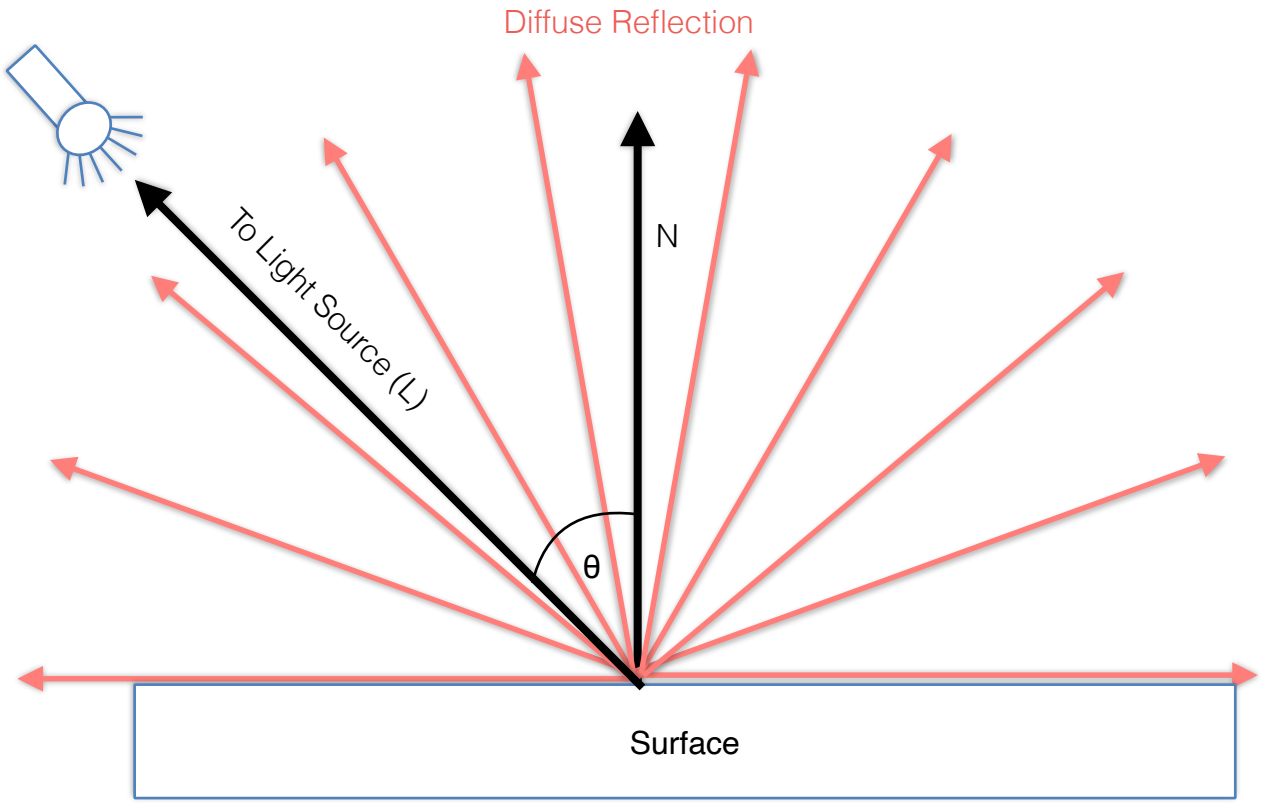
θ = 45°:

θ = 90°: Dark

Diffuse Component

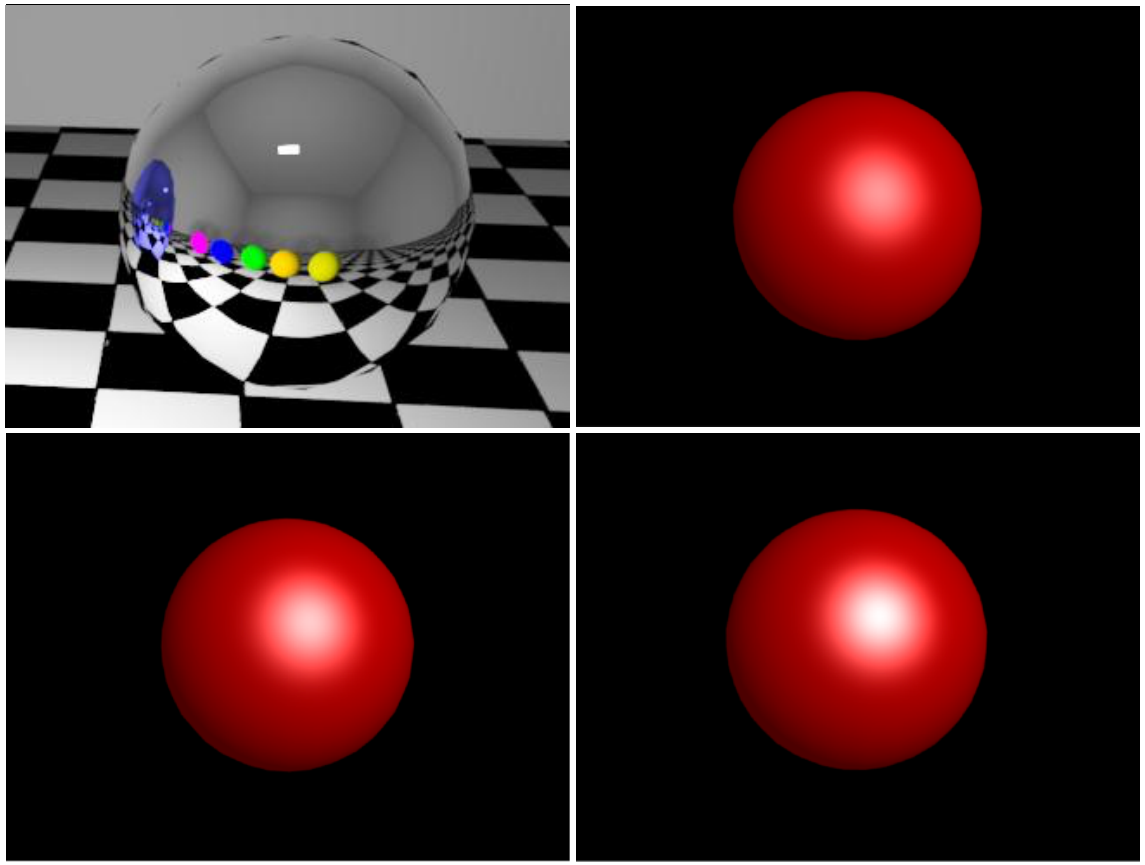
$$I_{diffuse} = I_d k_d (\hat{N} \cdot \hat{L}) = I_d k_d \cos\theta$$

- Diffuse light component Id
- Diffuse material factor kd
- Light direction L
- Surface normal N



Specular Component

- Simulates highlights from shiny objects
- Called specular highlight
- For ideal reflectors: angle of incidence equals angle of reflection (only visible if R equals V)
- For non-perfect reflectors: highlight is visible from a range of angles

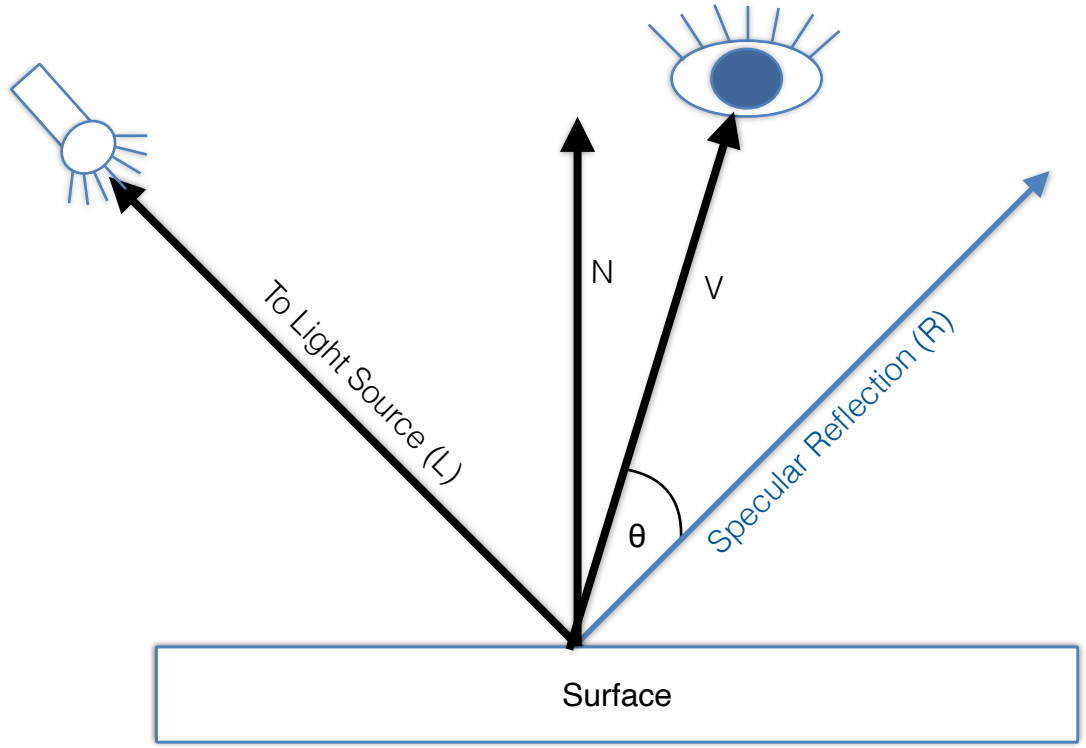


Specular Component

- Consists of:
  - Direction to light source
  - Reflected ray (in GLSL using reflect method)

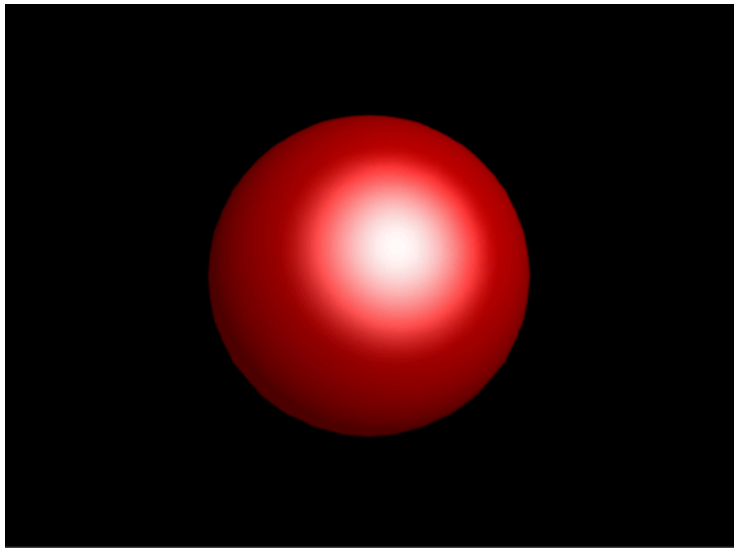
$$\hat{R} = 2(\hat{N} \cdot \hat{L})\hat{N} - \hat{L}$$

  - Specular material factor ks
  - Specular exponent ns (the larger, the smaller the highlight)
  - ks and ns have no physical meaning (a lot of tweaking required to achieve desired result)

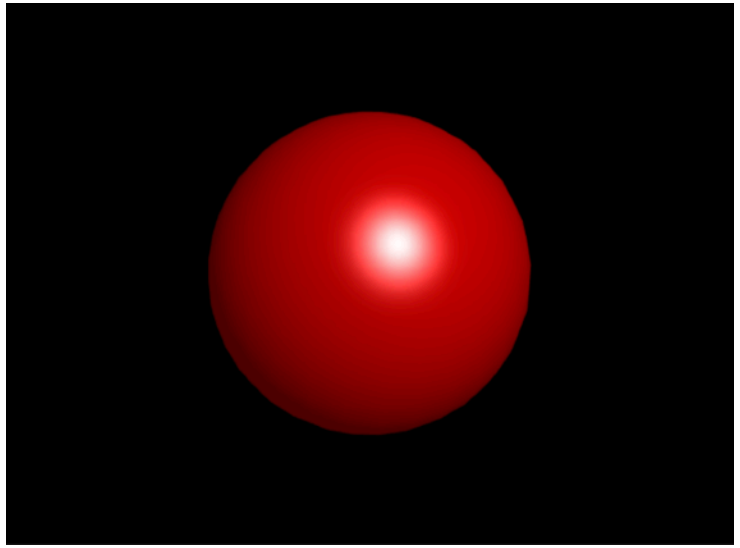


$$I_{specular} = I_s k_s (\hat{R} \cdot \hat{V})^{n_s}$$

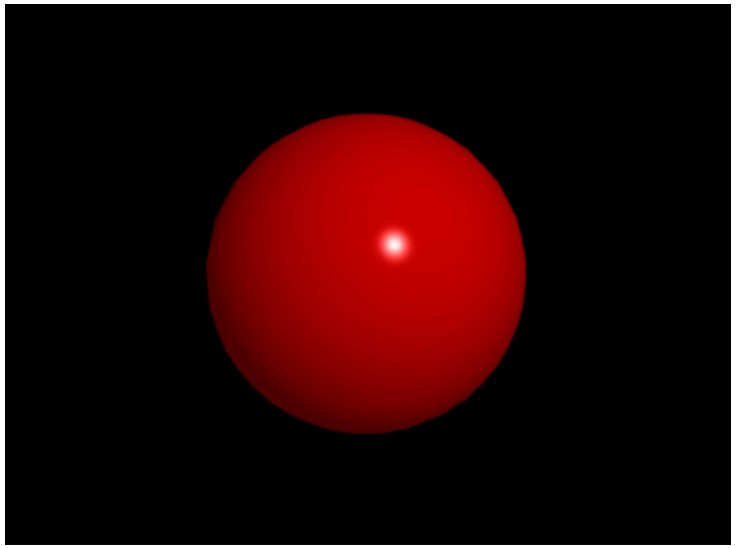
Specular Component



$n_s = 2.0$



$n_s = 10.0$



$n_s = 100.0$

Reflection Models

- Reflection models define how to combine components
- Different ways to do the computation
  - Phong
  - Blinn-Phong
  - Cook-Torrance
  - Oren-Nayar

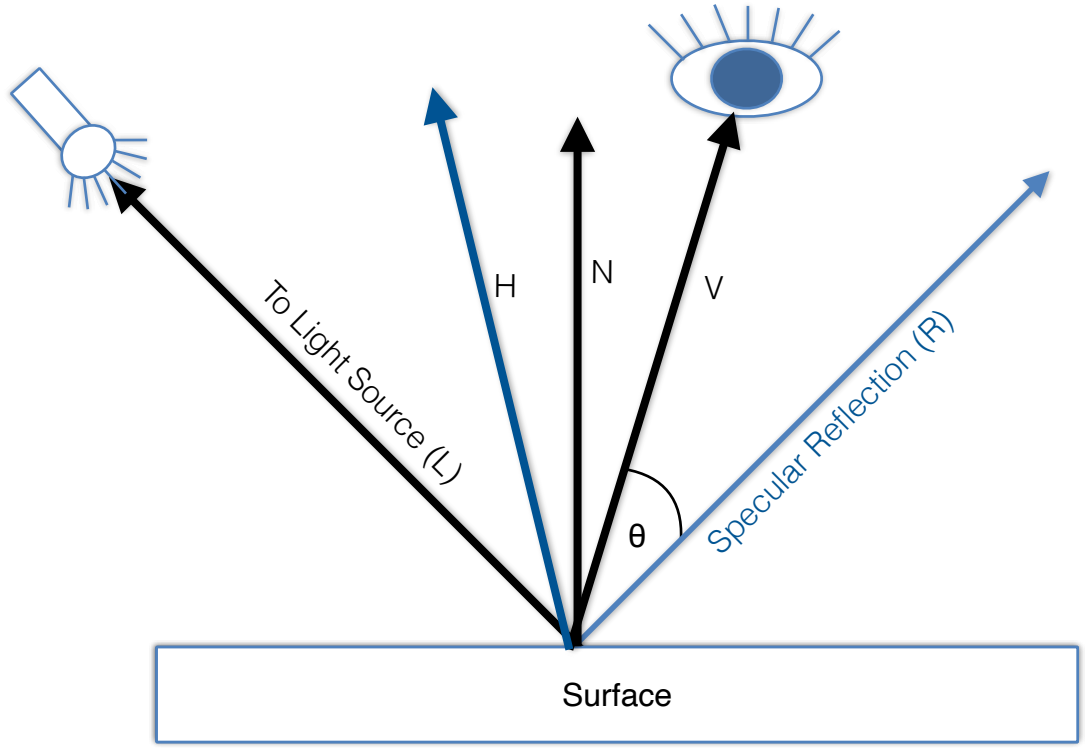
Phong Reflection Model

Illumination = $I_{ambient} = I_a k_a$ $I_{diffuse} = I_d k_d (\hat{N} \cdot \hat{L})$ $I_{specular} = I_s k_s (\hat{R} \cdot \hat{V})^{n_s}$

Illumination = $I_a k_a + I_d k_d (\hat{N} \cdot \hat{L}) + I_s k_s (\hat{R} \cdot \hat{V})^{n_s}$

Blinn-Phong Reflection Model

- Modification of phong reflection model by Jim Blinn
- Phong requires to recalculate the dot product (R dot V)
- Blinn-Phong uses halfway vector (H) between the viewer and light-source vectors

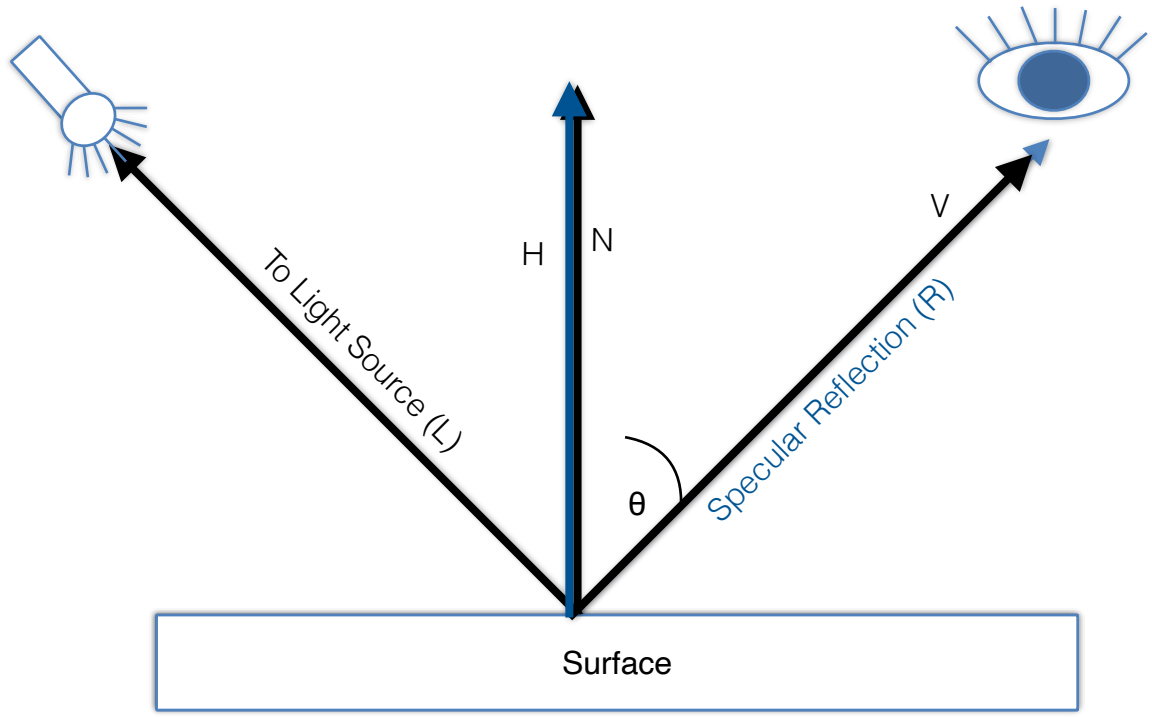


$$H = normalize(L + V) = \frac{(L + V)}{|L + V|}$$

$$I_{specular} = I_s k_s (\hat{H} \cdot \hat{N})^{n_s}$$

Blinn-Phong Reflection Model

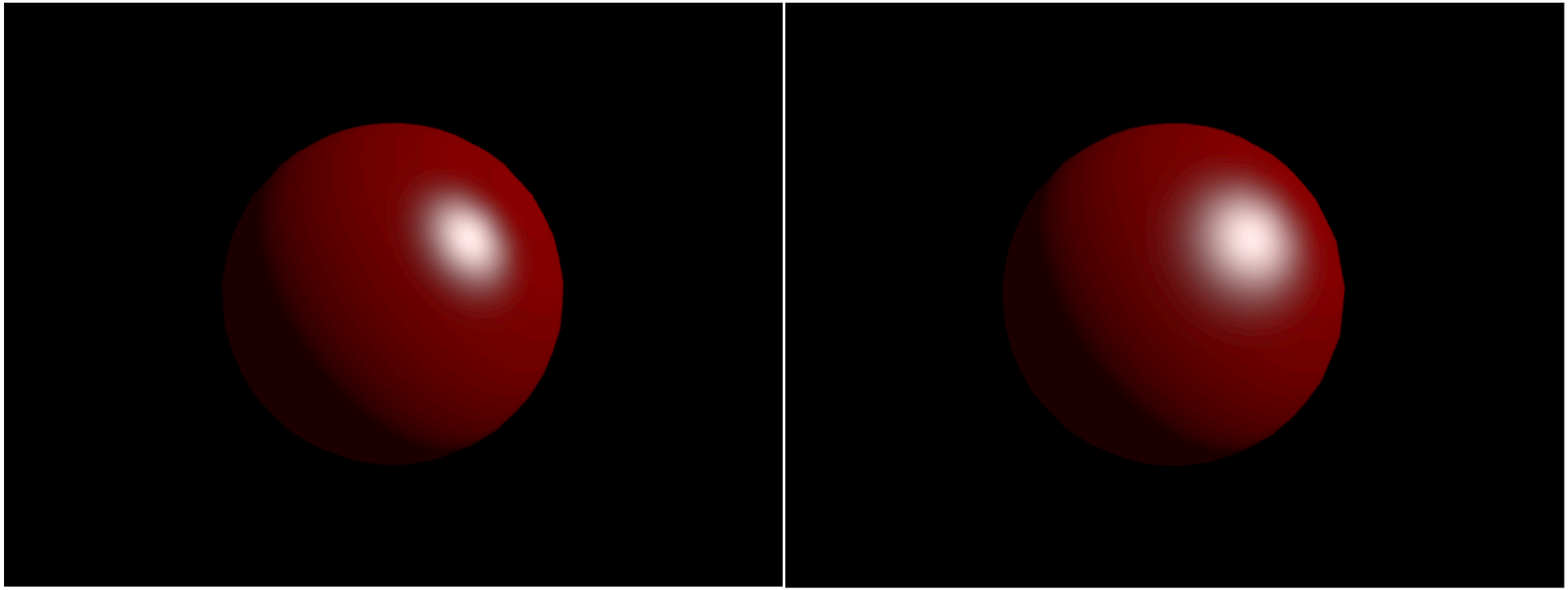
- Modification of phong reflection model by Jim Blinn
- Phong requires to recalculate the dot product (R dot V)
- Blinn-Phong uses halfway vector (H) between the viewer and light-source vectors



$$H = normalize(L + V) = \frac{(L + V)}{|L + V|}$$

$$I_{specular} = I_s k_s (\hat{H} \cdot \hat{N})^{n_s}$$

Blinn-Phong Reflection Model

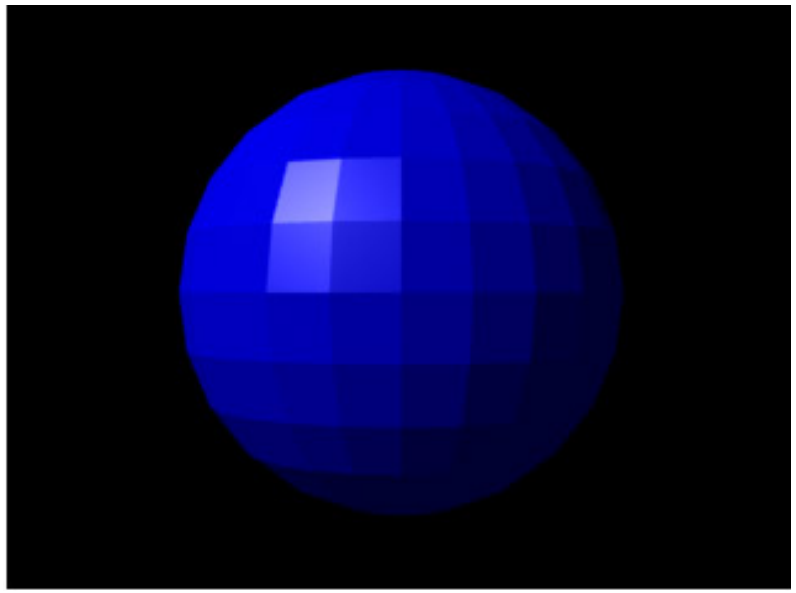


Phong

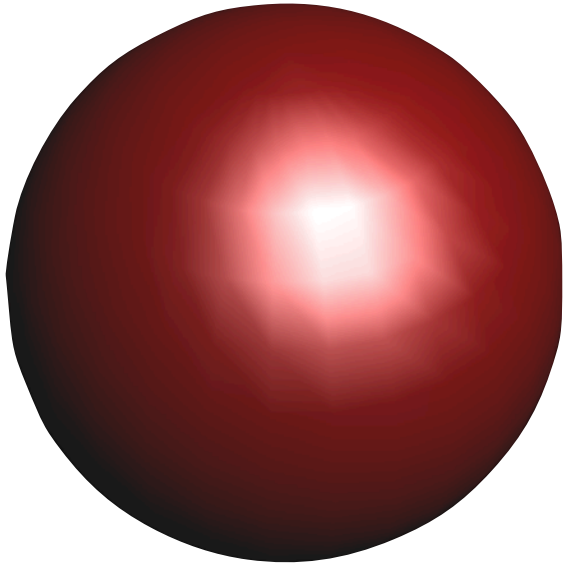
Blinn-Phong

Shading Models

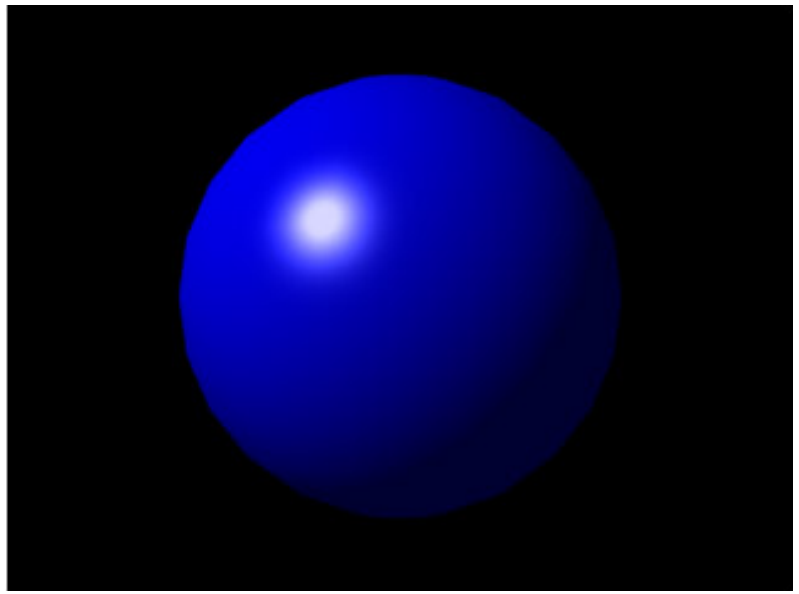
- Shading model determines on which shader stage and with which quality lighting for triangles is calculated



Flat Shading



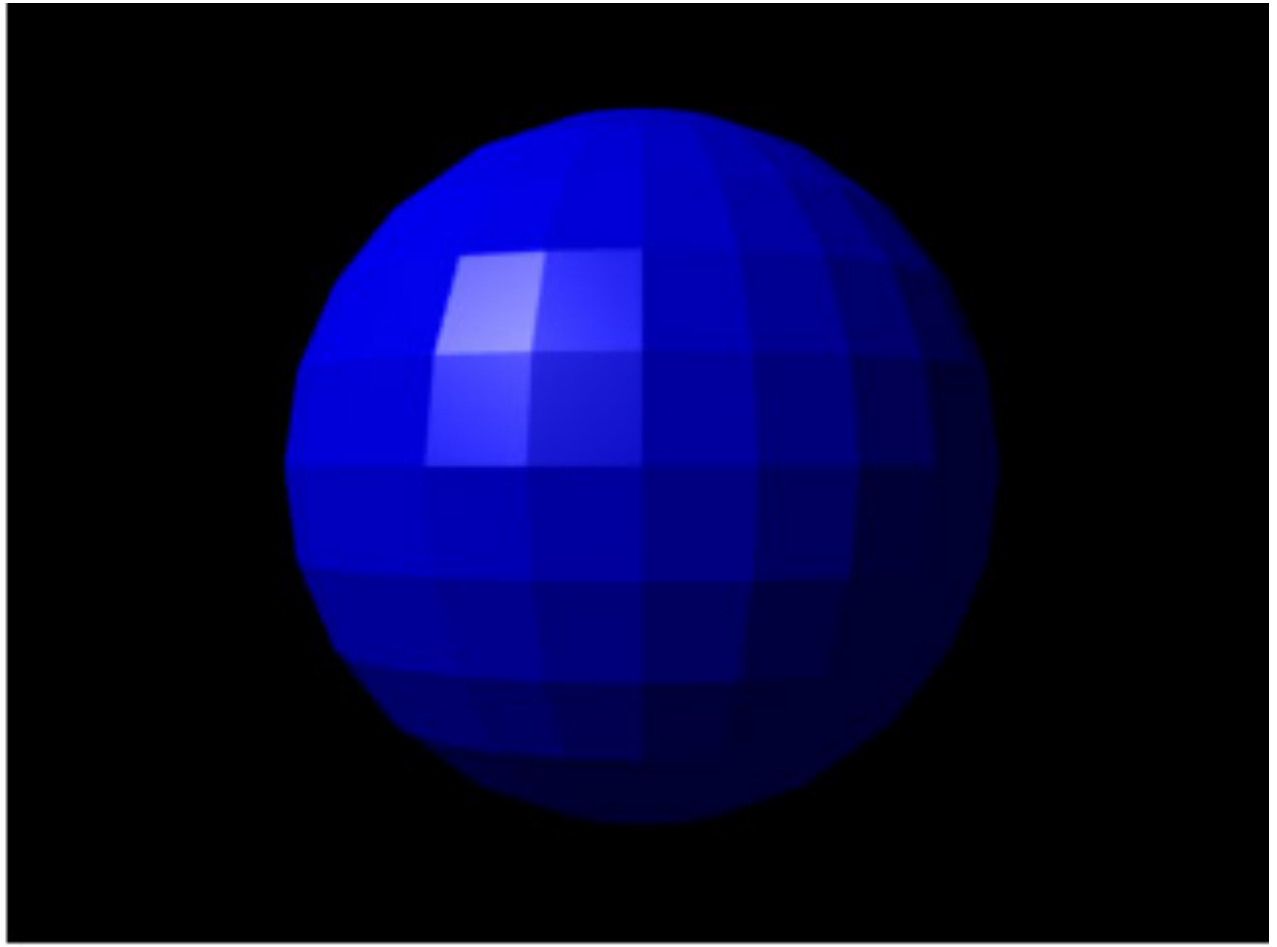
Gouraud Shading



Phong Shading

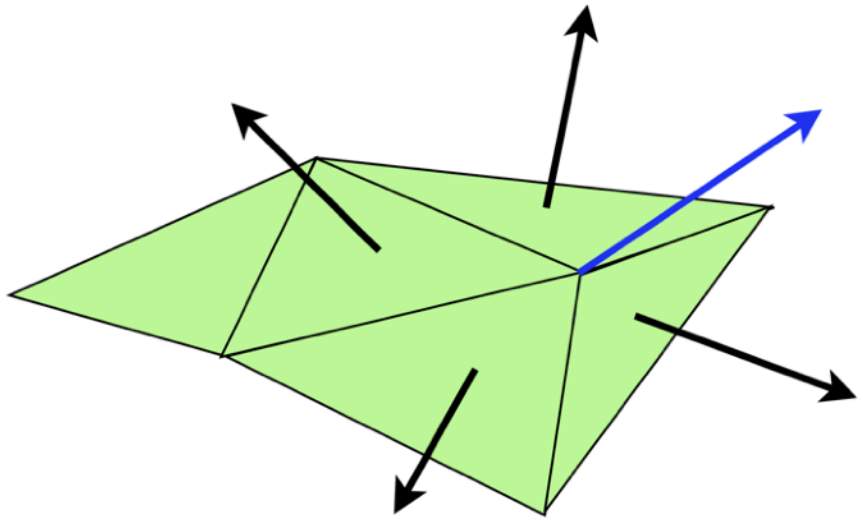
Flat Shading

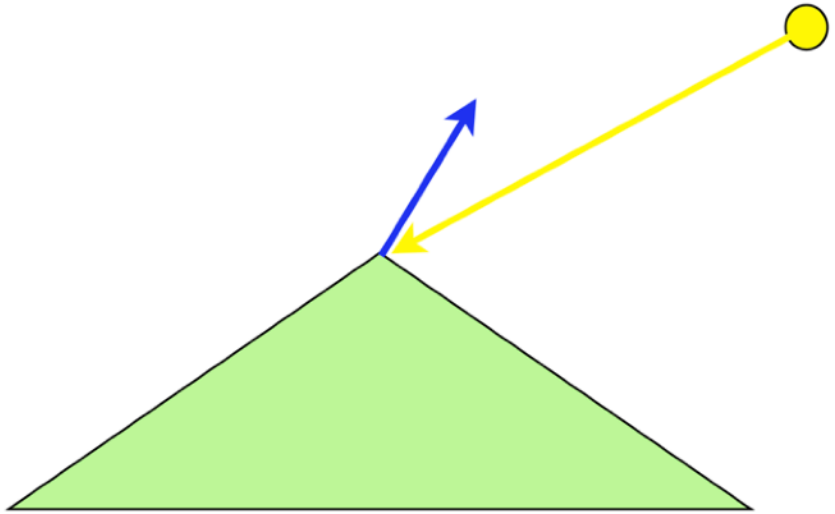
- Per polygon
- Was used for high speed rendering
- All vertices of one polygons have the same colour
- Difference between polygons
- No smooth transitions



Gouraud shading

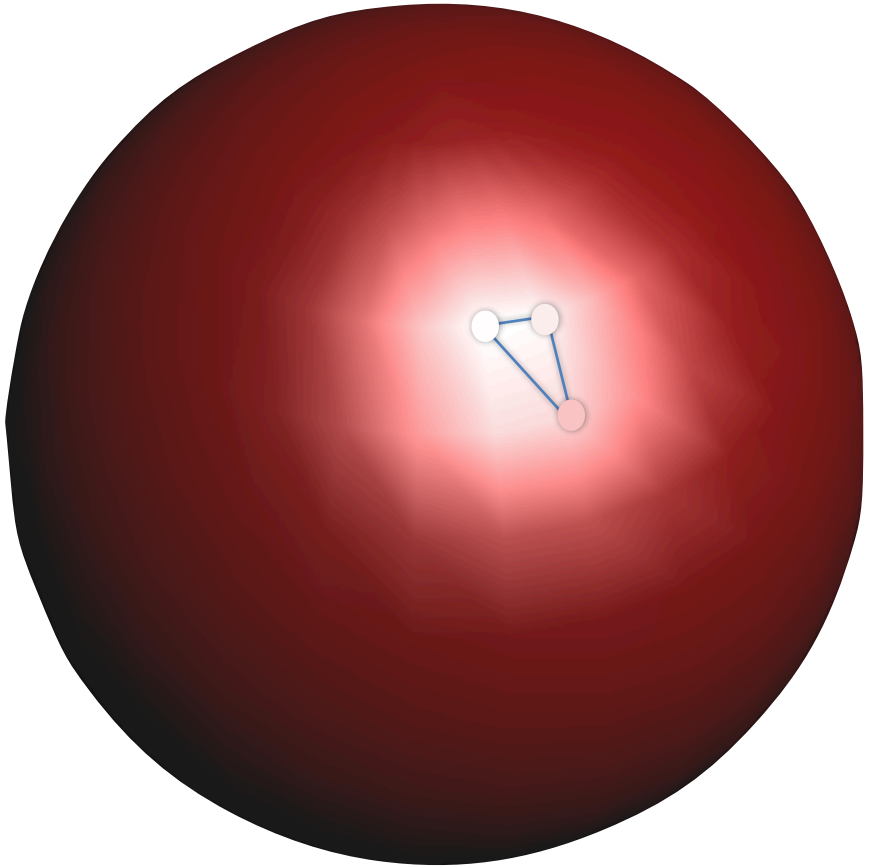
- Per vertex
- Interpolative shading
- Calculate polygon vertex colour
- Interpolate colours for interior points





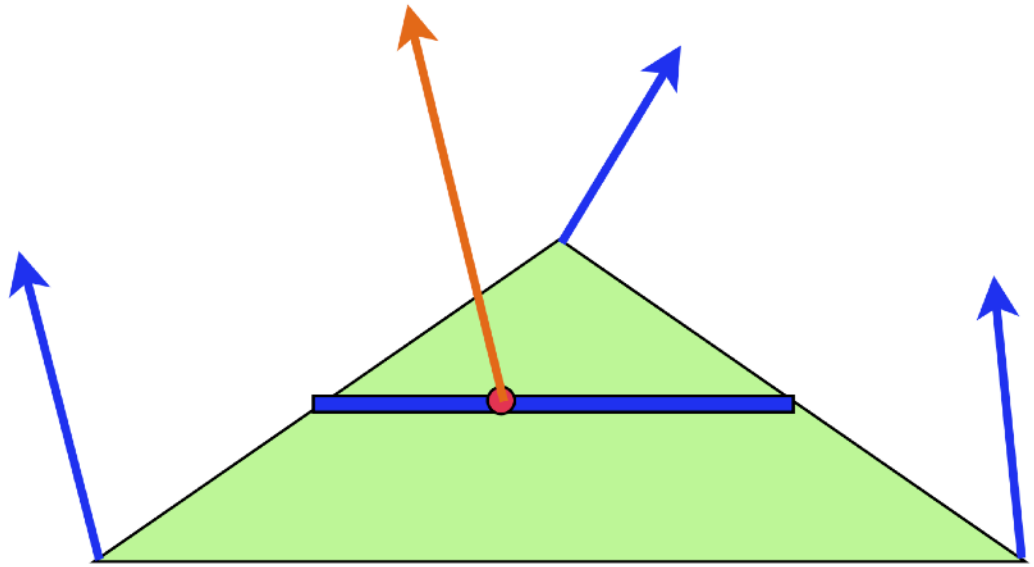
Gouraud shading

- Per vertex
- Interpolative shading
- Calculate polygon vertex colour
- Interpolate colours for interior points



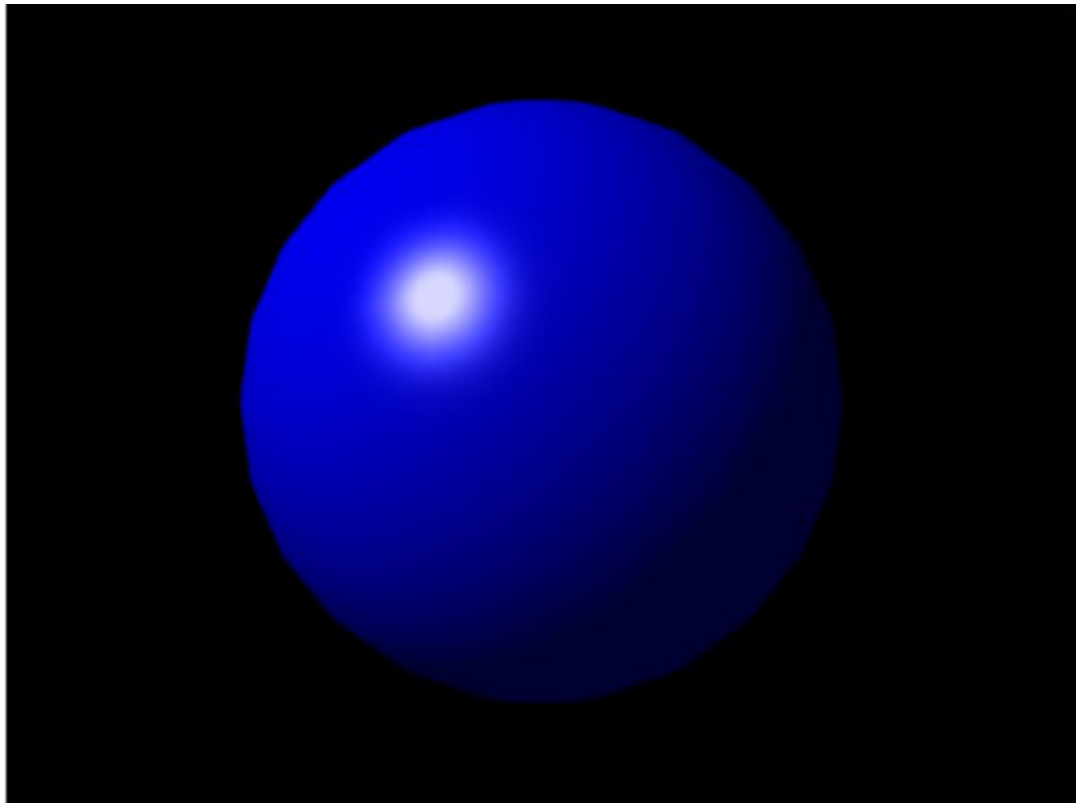
Phong Shading

- This is NOT Phong illumination
- Per fragment
- Interpolates the surface normals instead of the intensity values
- Then do calculation of intensities using the interpolated normal
- Gives better results, especially for highlights



Phong Shading

- This is NOT Phong illumination
- Per fragment
- Interpolates the surface normals instead of the intensity values
- Then do calculation of intensities using the interpolated normal
- Gives better results, especially for highlights



Shading Example 1

- Vertex shader (basicMaterialShader.vert)

```
float cosTheta = clamp( dot( N,L), 0,1 );
float cosAlpha = clamp( dot( V,R ), 0,1 ); //V or E vector for eye vector
vec3 diffuseComponent = diffuseLightColor* diffuseMatColor * textureVal * cosTheta;
vec3 ambientComponent = ambientLightColor * ambientMatColor * textureVal; //for simplification
we reuse the diffuse texture map for the ambient texture map
vec3 specularComponent = specularLightColor * specularMatColor  * pow(cosAlpha,ns);

color =
  // Ambient : simulates indirect lighting
ambientComponent +
  // Diffuse : "color" of the object
diffuseComponent +
  // Specular : reflective highlight, like a mirror
specularComponent;
```

Shading Example 2

- Fragment shader (basicMaterialShader.frag)

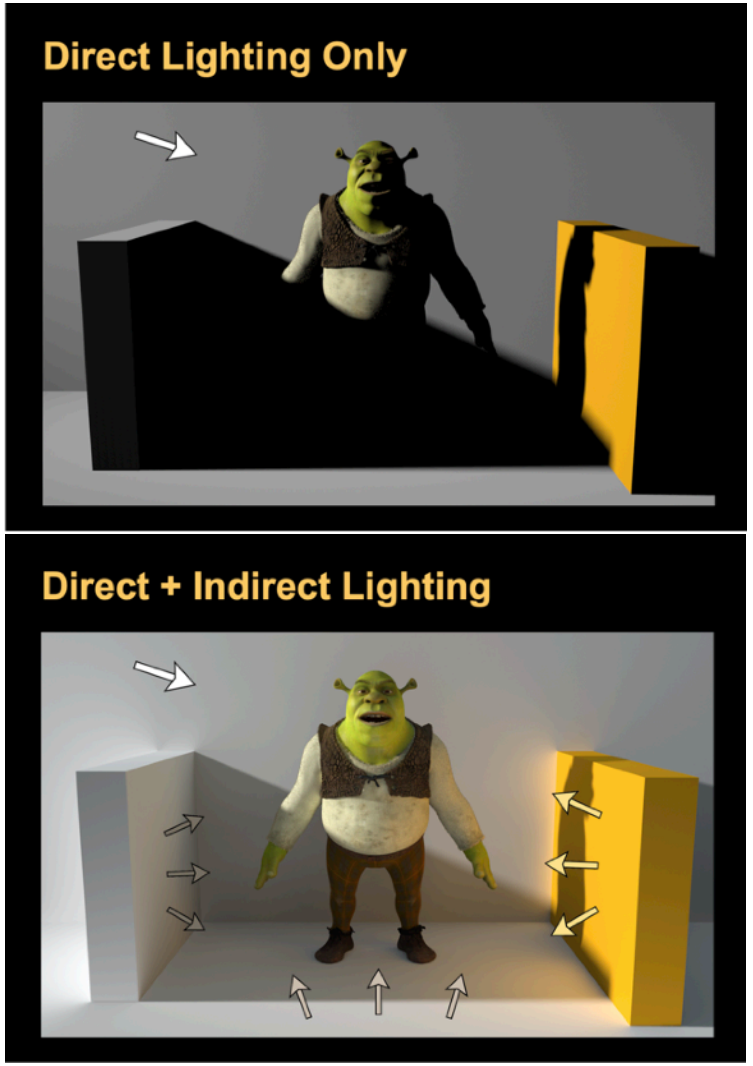
```
float cosTheta = clamp( dot( N,L), 0,1 );
float cosAlpha = clamp( dot( V,R ), 0,1 ); //V or E vector for eye vector
vec3 diffuseComponent = diffuseLightColor* diffuseMatColor * textureVal * cosTheta;
vec3 ambientComponent = ambientLightColor * ambientMatColor * textureVal; //for simplification
we reuse the diffuse texture map for the ambient texture map
vec3 specularComponent = specularLightColor * specularMatColor  * pow(cosAlpha,ns);

color =
  // Ambient : simulates indirect lighting
ambientComponent +
  // Diffuse : "color" of the object
diffuseComponent +
  // Specular : reflective highlight, like a mirror
specularComponent;
```

3-Min Discussion:

What types of shading are implemented in Shader Example 1 and in Shader Example 2?

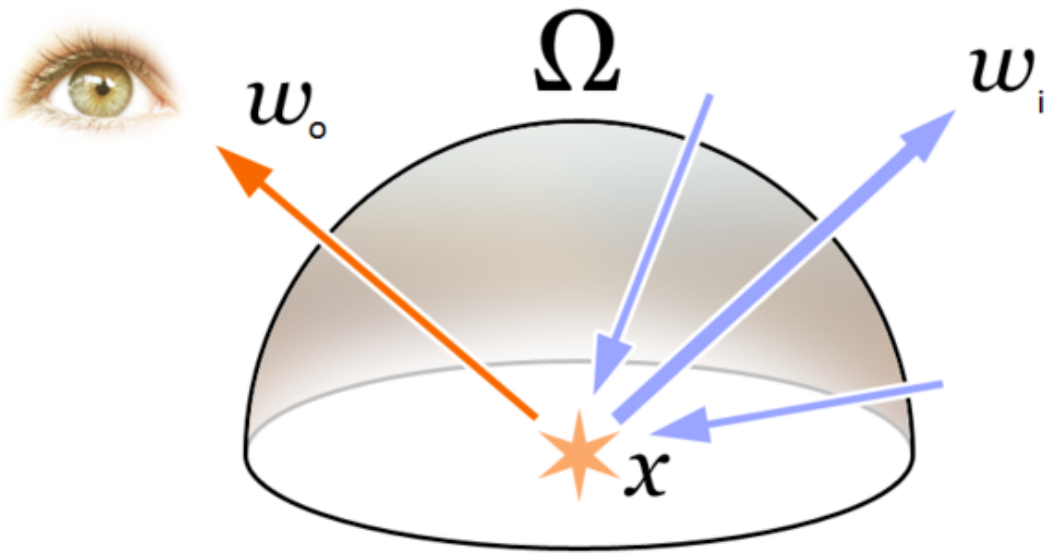
Outlook: Global Illumination

- Direct and indirect lighting
- Option in Unity: Real-time Global Illumination
  - Shadows
  - Inter-object reflections
  - Rendering equation
  - Radiosity



SIGGRAPH 2010 Course: “Global Illumination Across Industries

Rendering Equation



https://commons.wikimedia.org/wiki/File:Rendering_eq.png

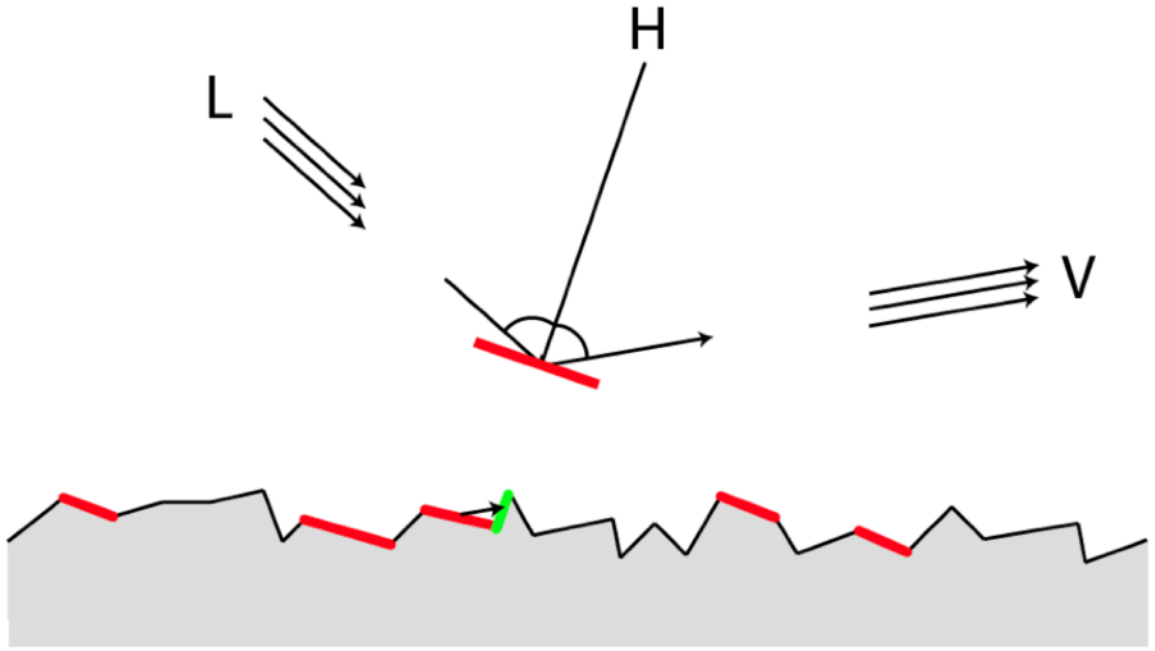
Kajiya 1986

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t)(\omega_i \cdot \mathbf{n}) \, \mathrm{d}\, \omega_i$$

Spectral radiance	Emitted spectral radiance	BRDF	Spectral radiance of wavelength (λ) coming inward toward point x
outgoing radiance (what we see)	emitted light (e.g., from a lamp or emissive material)	(bidirectional reflectance distribution function)	incoming radiance from direction

Physically Based Rendering (PBR)

- Approximation of Kaya's equation
- Render materials based on real-world light behaviour, not ad-hoc artistic parameters.
- Core principles:
  - Energy conservation:
    - Light reflected ≤ light received
  - Microfacet model -> surfaces have tiny roughness details that shape reflections
  - Metallic–Roughness workflow -> materials defined by metalness and roughness, not specular color



Material Settings Across 3D Model File Formats

- Base color / diffuse color
- Specular / metallic
- Roughness / glossiness
- Normal / bump maps
- Transparency / opacity
- Emission / self-illumination
- Texture references (image maps, UVs)

```
newmtl EarthMaterial

Ka 0.640000 0.640000 0.640000
Kd 0.640000 0.640000 0.640000
Ks 0.050000 0.050000 0.050000
Ns 30.0000
d 0.5
illum 2
map_Kd ColorMap.bmp
```

Example

Material Definitions

- Material Template Library (MTL) can be used to define material settings
- Defines ambient (Ka), diffuse (Kd), specular (Ks) colours and the specular exponent (Ns)
- Also allows to define opacity (d) - 1.0 means fully opaque
- Set texture maps (map_Kd)

```
newmtl EarthMaterial

Ka 0.640000 0.640000 0.640000
Kd 0.640000 0.640000 0.640000
Ks 0.050000 0.050000 0.050000
Ns 30.0000
d 0.5
illum 2
map_Kd ColorMap.bmp
```

Example

Material Definitions in GLTF

- Uses physically based rendering (PBR) parameters
- Separates metallic and roughness for realistic reflections
- Texture indices link to binary or image data (e.g., .glb or .bin)
- Enables consistent results across modern engines (WebGL, Unity, Unreal, etc.)

```
"materials": [
  {
    "name": "Bronze",
    "pbrMetallicRoughness": {
      "baseColorTexture": { "index": 0 },
      "baseColorFactor": [0.714, 0.428, 0.181, 1.0],
      "metallicFactor": 0.9,
      "roughnessFactor": 0.4
    },
    "normalTexture": { "index": 1 },
    "emissiveFactor": [0.0, 0.0, 0.0],
    "alphaMode": "OPAQUE"
  }
]
```

Example GLTF

The end!