WIKIPEDIA

Blinn–Phong reflection model

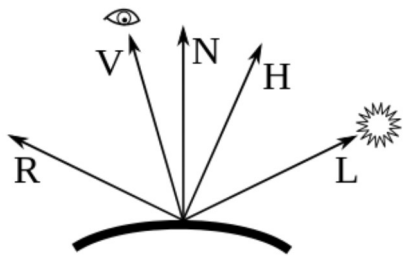
The **Blinn–Phong reflection model**, also called the **modified Phong reflection model,** is a modification developed by Jim Blinn to the Phong reflection model.[1]

Blinn–Phong is the default shading model used in OpenGL and Direct3D's fixed-function pipeline (before Direct3D 10 and OpenGL 3.1), and is carried out on each vertex as it passes down the graphics pipeline; pixel values between vertices are interpolated by Gouraud shading by default, rather than the more computationally-expensive Phong shading.[2]

Contents

Description

In Phong shading, one must continually recalculate the dot product $R \cdot V$ between a viewer (V) and the beam from a light-source (L) reflected (R) on a surface.



Vectors for calculating Phong and Blinn–Phong shading

If, instead, one calculates a *halfway vector* between the viewer and light-source vectors,

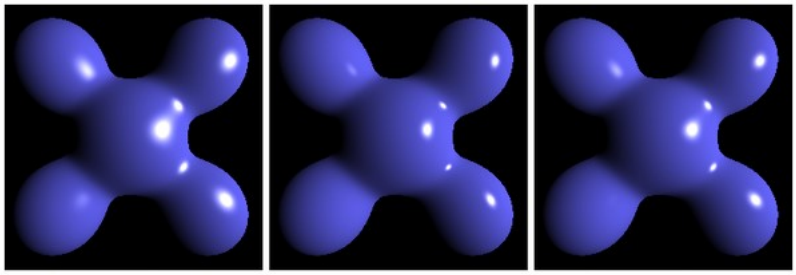
$$H = \frac{L + V}{\|L + V\|}$$

$R \cdot V$ can be replaced with $N \cdot H$, where N is the normalized surface normal. In the above equation, L and V are both normalized vectors, and H is a solution to the equation $V = P_H(-L)$, where P_H is the Householder matrix that reflects a point in the hyperplane that contains the origin and has the normal H.

This dot product represents the cosine of an angle that is half of the angle represented by Phong's dot product if V, L, N and R all lie in the same plane. This relation between the angles remains approximately true when the vectors don't lie in the same plane, especially when the angles are small. The angle between N and H is therefore sometimes called the halfway angle.

Considering that the angle between the halfway vector and the surface normal is likely to be smaller than the angle between R and V used in Phong's model (unless the surface is viewed from a very steep angle for which it is likely to be larger), and since Phong is using $(R \cdot V)^{\alpha}$, an exponent can be set $\alpha' > \alpha$ such that $(N \cdot H)^{\alpha'}$ is closer to the former expression.

For front-lit surfaces (specular reflections on surfaces facing the viewer), $\alpha' = 4\alpha$ will result in specular highlights that very closely match the corresponding Phong reflections. However, while the Phong reflections are always round for a flat surface, the Blinn–Phong reflections become elliptical when the surface is viewed from a steep angle. This can be compared to the case where the sun is reflected in the sea close to the horizon, or where a far away street light is reflected in wet pavement, where the reflection will always be much more extended vertically than horizontally.[3]



Additionally, while it can be seen as an approximation to the Phong model, it produces more accurate models of empirically determined bidirectional reflectance distribution functions than Phong for many types of surfaces.[4]

Efficiency

Blinn-Phong will be faster than Phong in the case where the viewer and light are treated to be very remote, such as approaching or at infinity. This is the case for directional lights and orthographic/isometric cameras. In this case, the halfway vector is independent of position and surface curvature simply because the halfway vector is dependent on the direction to viewer's position and the direction to the light's position, which individually converge at this remote distance, hence the halfway vector can be thought of as constant in this case. H therefore can be computed once for each light and then used for the entire frame, or indeed while light and viewpoint remain in the same relative position. The same is not true with Phong's method of using the reflection vector which depends on the surface curvature and must be recalculated for each pixel of the image (or for each vertex of the model in the case of vertex lighting). In 3D scenes with perspective cameras, this optimization is not possible.

Code samples

High-Level Shading Language code sample

This sample in High-Level Shading Language is a method of determining the diffuse and specular light from a point light. The light structure, position in space of the surface, view direction vector and the normal of the surface are passed through. A Lighting structure is returned;

The below also needs to clamp certain dot products to zero in the case of negative answers.

Without that, light heading away from the camera is treated the same way as light heading towards it. For the specular calculation, an incorrect "halo" of light glancing off the edges of an object and away from the camera might appear as bright as the light directly being reflected towards the camera.

```
struct Lighting
{
    float3 Diffuse;
    float3 Specular;
};

struct PointLight
{
    float3 position;
    float3 diffuseColor;
    float  diffusePower;
    float3 specularColor;
    float  specularPower;
};

Lighting GetPointLight(PointLight light, float3 pos3D, float3 viewDir, float3 normal)
{
    Lighting OUT;
    if (light.diffusePower > 0)
    {
        float3 lightDir = light.position - pos3D; //3D position in space of the surface
        float distance = length(lightDir);
        lightDir = lightDir / distance; // = normalize(lightDir);
        distance = distance * distance; //This line may be optimised using Inverse square root

        //Intensity of the diffuse light. Saturate to keep within the 0-1 range.
        float NdotL = dot(normal, lightDir);
        float intensity = saturate(NdotL);

        // Calculate the diffuse light factoring in light color, power and the attenuation
        OUT.Diffuse = intensity * light.diffuseColor * light.diffusePower / distance;

        //Calculate the half vector between the light vector and the view vector.
        //This is typically slower than calculating the actual reflection vector
        // due to the normalize function's reciprocal square root
        float3 H = normalize(lightDir + viewDir);

        //Intensity of the specular light
        float NdotH = dot(normal, H);
        intensity = pow(saturate(NdotH), specularHardness);

        //Sum up the specular light factoring
        OUT.Specular = intensity * light.specularColor * light.specularPower / distance;
    }
    return OUT;
}
```

OpenGL Shading Language code sample

This sample in the OpenGL Shading Language consists of two code files, or *shaders*. The first one is a so-called *vertex shader* and implements Phong shading, which is used to interpolate the surface normal between vertices. The second shader is a so-called *fragment shader* and implements the Blinn–Phong shading model in order to determine the diffuse and specular light from a point light source.

Vertex shader

This vertex shader implements Phong shading:

```
attribute vec3 inputPosition;
attribute vec2 inputTexCoord;
attribute vec3 inputNormal;

uniform mat4 projection, modelview, normalMat;

varying vec3 normalInterp;
varying vec3 vertPos;

void main() {
    gl_Position = projection * modelview * vec4(inputPosition, 1.0);
    vec4 vertPos4 = modelview * vec4(inputPosition, 1.0);
    vertPos = vec3(vertPos4) / vertPos4.w;
    normalInterp = vec3(normalMat * vec4(inputNormal, 0.0));
}
```

Fragment shader

This fragment shader implements the Blinn–Phong shading model[5] and gamma correction:

```
precision mediump float;

in vec3 normalInterp;
in vec3 vertPos;

uniform int mode;

const vec3 lightPos = vec3(1.0, 1.0, 1.0);
const vec3 lightColor = vec3(1.0, 1.0, 1.0);
const float lightPower = 40.0;
const vec3 ambientColor = vec3(0.1, 0.0, 0.0);
const vec3 diffuseColor = vec3(0.5, 0.0, 0.0);
const vec3 specColor = vec3(1.0, 1.0, 1.0);
const float shininess = 16.0;
const float screenGamma = 2.2; // Assume the monitor is calibrated to the sRGB color space

void main() {

  vec3 normal = normalize(normalInterp);
  vec3 lightDir = lightPos - vertPos;
  float distance = length(lightDir);
  distance = distance * distance;
  lightDir = normalize(lightDir);

  float lambertian = max(dot(lightDir, normal), 0.0);
  float specular = 0.0;

  if (lambertian > 0.0) {

    vec3 viewDir = normalize(-vertPos);

    // this is blinn phong
    vec3 halfDir = normalize(lightDir + viewDir);
    float specAngle = max(dot(halfDir, normal), 0.0);
    specular = pow(specAngle, shininess);

    // this is phong (for comparison)
    if (mode == 2) {
      vec3 reflectDir = reflect(-lightDir, normal);
      specAngle = max(dot(reflectDir, viewDir), 0.0);
      // note that the exponent is different here
      specular = pow(specAngle, shininess/4.0);
    }
  }
  vec3 colorLinear = ambientColor +
                     diffuseColor * lambertian * lightColor * lightPower / distance +
                     specColor * specular * lightColor * lightPower / distance;
  // apply gamma correction (assume ambientColor, diffuseColor and specColor
  // have been linearized, i.e. have no gamma correction in them)
  vec3 colorGammaCorrected = pow(colorLinear, vec3(1.0 / screenGamma));
```

```
    // use the gamma corrected color in the fragment
    gl_FragColor = vec4(colorGammaCorrected, 1.0);
}
```

The colors `ambientColor`, `diffuseColor` and `specColor` are supposed not to be gamma corrected. If they are colors obtained from gamma-corrected image files (JPEG, PNG, etc.), they need to be linearized before working with them, which is carried out by scaling the channel values to the range $[0, 1]$ and raising them to the gamma value of the image, which for images in the sRGB color space can be assumed to be about 2.2 (even though for this specific color space, a simple power relation is just an approximation of the actual transformation). Modern graphics APIs have the ability to perform this gamma correction automatically when sampling from a texture or writing to a framebuffer.[6]

See also

- List of common shading algorithms
- Phong reflection model for Phong's corresponding model
- Gamma correction
- Specular highlight

References

1. James F. Blinn (1977). "Models of light reflection for computer synthesized pictures". *Proc. 4th annual conference on computer graphics and interactive techniques*: 192–198. CiteSeerX 10.1.1.131.7741 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.7741). doi:10.1145/563858.563893 (https://doi.org/10.1145%2F563858.563893).
2. Shreiner, Dave; The Khronos OpenGL ARB Working Group (2010). "The Mathematics of Lighting". *OpenGL programming guide : the official guide to learning OpenGL, version 3.0 and 3.1* (7th ed.). Pearson Education, Inc. pp. 240–245. ISBN 978-0-321-55262-4.
3. Krus, Kristofer (2014), *Wave Model and Watercraft Model for Simulation of Sea State* (http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-102959), Linköping University, p. 97
4. Ngan, Addy; Durand, Frédo; Matusik, Wojciech (2004). "Experimental validation of analytical BRDF models" (https://people.csail.mit.edu/wojciech/BRDFValidation/index.html). *ACM SIGGRAPH 2004 Sketches on - SIGGRAPH '04*. ACM Press. doi:10.1145/1186223.1186336 (https://doi.org/10.1145%2F1186223.1186336). Retrieved 23 April 2019.
5. "WebGL Example: Phong / Blinn Phong Shading" (https://www.mathematik.uni-marburg.de/~thormae/lectures/graphics1/code/WebGLShaderLightMat/ShaderLightMat.html). *www.mathematik.uni-marburg.de*. Retrieved 2019-09-13.
6. https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_framebuffer_sRGB.txt