# Tutorial 09 Lighting

This tutorial introduces the use lighting/reflection model in WebGL applications. The *Phong lighting/reflection model* is implemented in the vertex shader. Because vertex shader performs per operations, the lighting evaluation will be carried out on each vertex, hence the name of *per-vertex shading*.

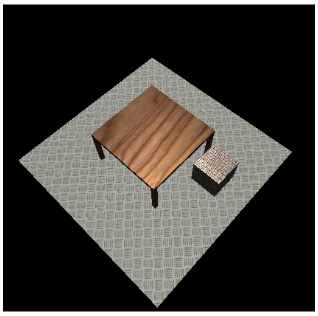

Figure A

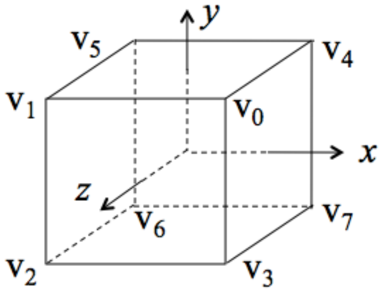


Figure B

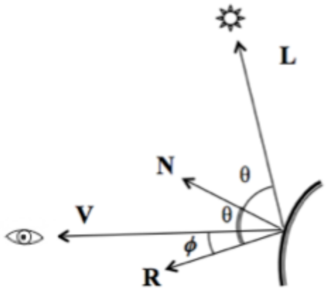


Figure C

These vectors are calculated: **L, N, V** and **R**, as shown in Figure C. They are used to evaluate the Phong reflection model. A single point light is used to illuminate the scene.

$$I = c_{ambient} + \sum_{i=1}^{all\_lights} \left( c_{i_{diffuse}} \mathbf{L}_i \bullet \mathbf{N} + c_{i_{specular}} (\mathbf{V} \bullet \mathbf{R})^{\alpha} \right) \quad \text{with} \quad \mathbf{R} = 2(\mathbf{L}_i \bullet \mathbf{N})\mathbf{N} - \mathbf{L}_i$$

Colours $c_{ambient}$, $c_{diffucse}$, $c_{specular}$ and the position of the point light are set via uniform variables:

```
uniform vec3 uLightPosition;
uniform vec3 uAmbientLightColor;
uniform vec3 uDiffuseLightColor;
uniform vec3 uSpecularLightColor;
```

The diffuse term of the model is calculated from ***L dot N***, as a (positive) weighting factor.

```
diffuseLightWeighting = max(dot(normalEye, vectorToLightSource),
                    0.0);
```

The specular term involves ***V dot R*** and a power operation:

```
rdotv = max(dot(reflectionVector, viewVectorEye), 0.0);
specularLightWeighting = pow(rdotv, shininess);
```

The overall reflection *at a vertex* is the weighted sum of the colours $c_{ambient}$, $c_{diffucse}$, $c_{specular}$ and is stored in a *varying* variable, `vLightWeighting`, to be send to the fragment shader:

```
vLightWeighting = uAmbientLightColor +
                  uDiffuseLightColor * diffuseLightWeighting +
                  uSpecularLightColor * specularLightWeighting;
```

In the fragment shader, the fragment colour obtained by interpolating the vertex colours, `vLightWeighting`, is used to modulate the texel colour for a fragment:

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

Notice that a special transformation `uNMatrix`, which equals the ***transpose of inverse*** of the modelview matrix, is used to transform vertex normals from local coordinate system to the camera coordinate system (see lecture note), `normalEye`.

```
uniform mat3 uNMatrix;
  . . .
vec3 normalEye = normalize(uNMatrix * aVertexNormal);
```

**Exercise**: Finish the partially finished program by adding the shader programs and the highlighted statements or functions.

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title> Phong Lighting </title>
<script src="webgl-debug.js"></script>
<script type="text/javascript" src="glMatrix.js"></script>
<script src="webgl-utils.js"></script>
<meta charset="utf-8">


<script id="shader-vs" type="x-shader/x-vertex">

  attribute vec3 aVertexPosition;
  attribute vec3 aVertexNormal;
  attribute vec2 aTextureCoordinates;

  uniform mat4 uMVMatrix;
  uniform mat4 uPMatrix;
  uniform mat3 uNMatrix;

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

  varying vec2 vTextureCoordinates;
  varying vec3 vLightWeighting;

  const float shininess = 32.0;

  void main() {
    // Get the vertex position in camera/eye coordinates and convert
    // the homogeneous coordinates back to the usual 3D coordinates for
    // subsequent calculations.
    vec4 vertexPositionEye4 = uMVMatrix * vec4(aVertexPosition, 1.0);
    vec3 vertexPositionEye3 = vertexPositionEye4.xyz /
                        vertexPositionEye4.w;

    // Calculate the vector (L) to the point light source
    // First, transform the coordinate of light source into
    //eye coordinate system
    vec4 lightPositionEye4 = uMVMatrix * vec4(uLightPosition, 1.0);
    vec3 lightPositionEye3 = lightPositionEye4.xyz /
                        lightPositionEye4.w;
    // Calculate the vector L
    vec3 vectorToLightSource = normalize(lightPositionEye3 -
                        vertexPositionEye3);

    // The following line of code provides a different way to calculate
    // vector L. What is the difference between the two approaches?
    // Try it out.
    //vec3 vectorToLightSource = normalize(uLightPosition -
    //                          vertexPositionEye3);

    // Transform the 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) that is needed for specular
    // light. Function reflect() is the GLSL function for calculation
    // of the reflective vector R
    vec3 reflectionVector = normalize(reflect(-vectorToLightSource,
                      normalEye));

    // In terms of the camera coordinate system, the camera/eye
    // is alway located at in the origin (0.0, 0.0, 0.0) (because the
    // coordinate system is rigidly attached to the camera)
    // Calculate view vector (V) in camera 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;

     // Finally transform the geometry
     gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
     vTextureCoordinates = aTextureCoordinates;
  }
</script>

<script id="shader-fs" type="x-shader/x-fragment">
  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);
}
</script>


<script type="text/javascript">

. . .

function setupShaders() {
   . . .

   gl.useProgram(shaderProgram);

   pwgl.vertexPositionAttributeLoc = gl.getAttribLocation(shaderProgram,
                      "aVertexPosition");
   pwgl.vertexTextureAttributeLoc = gl.getAttribLocation(shaderProgram,
                      "aTextureCoordinates");
   pwgl.uniformMVMatrixLoc = gl.getUniformLocation(shaderProgram,
                      "uMVMatrix");
```

```
  pwgl.uniformProjMatrixLoc = gl.getUniformLocation(shaderProgram,
                         "uPMatrix");
  pwgl.uniformSamplerLoc = gl.getUniformLocation(shaderProgram,
                         "uSampler");

  pwgl.uniformNormalMatrixLoc = gl.getUniformLocation(shaderProgram,
                         "uNMatrix");
  pwgl.vertexNormalAttributeLoc = gl.getAttribLocation(shaderProgram,
                         "aVertexNormal");
  pwgl.uniformLightPositionLoc =gl.getUniformLocation(shaderProgram,
                         "uLightPosition");
  pwgl.uniformAmbientLightColorLoc = gl.getUniformLocation(shaderProgram,
                         "uAmbientLightColor");
  pwgl.uniformDiffuseLightColorLoc = gl.getUniformLocation(shaderProgram,
                         "uDiffuseLightColor");
  pwgl.uniformSpecularLightColorLoc = gl.getUniformLocation(shaderProgram,
                         "uSpecularLightColor");

  gl.enableVertexAttribArray(pwgl.vertexNormalAttributeLoc);
  gl.enableVertexAttribArray(pwgl.vertexPositionAttributeLoc);
  gl.enableVertexAttribArray(pwgl.vertexTextureAttributeLoc);
  . . .
}


function setupFloorBuffers() {
  . . .

  // Specify normals to be able to do lighting calculations
  pwgl.floorVertexNormalBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.floorVertexNormalBuffer);

  var floorVertexNormals = [
       0.0,   1.0,  0.0,  //v0
       0.0,   1.0,  0.0,  //v1
       0.0,   1.0,  0.0,  //v2
       0.0,   1.0,  0.0]; //v3

  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(floorVertexNormals),
                               gl.STATIC_DRAW);
  pwgl.FLOOR_VERTEX_NORMAL_BUF_ITEM_SIZE = 3;
  pwgl.FLOOR_VERTEX_NORMAL_BUF_NUM_ITEMS = 4;

  . . .
}

function setupCubeBuffers() {
  . . .

  // Specify normals to be able to do lighting calculations
  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

        // Left face
        -1.0,  0.0,  0.0, //v1
        -1.0,  0.0,  0.0, //v5
        -1.0,  0.0,  0.0, //v6
        -1.0,  0.0,  0.0, //v2

        // Right face
        1.0,  0.0,  0.0, //0
        1.0,  0.0,  0.0, //3
        1.0,  0.0,  0.0, //7
        1.0,  0.0,  0.0, //4

        // Top face
        0.0,  1.0,  0.0, //v0
        0.0,  1.0,  0.0, //v4
        0.0,  1.0,  0.0, //v5
        0.0,  1.0,  0.0, //v1

        // Bottom face
        0.0, -1.0,  0.0, //v3
        0.0, -1.0,  0.0, //v7
        0.0, -1.0,  0.0, //v6
        0.0, -1.0,  0.0, //v2
        ];

  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;
}
. . .


function setupLights() {
  gl.uniform3fv(pwgl.uniformLightPositionLoc, [0.0, 20.0, 5.0]);
  gl.uniform3fv(pwgl.uniformAmbientLightColorLoc, [0.2, 0.2, 0.2]);
  gl.uniform3fv(pwgl.uniformDiffuseLightColorLoc, [0.7, 0.7, 0.7]);
  gl.uniform3fv(pwgl.uniformSpecularLightColorLoc, [0.8, 0.8, 0.8]);
}

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

}

. . .

function drawFloor() {
  . . .

  // Bind normal buffer
  gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.floorVertexNormalBuffer);
  gl.vertexAttribPointer(pwgl.vertexNormalAttributeLoc,
                         pwgl.FLOOR_VERTEX_NORMAL_BUF_ITEM_SIZE,
```

```
                        gl.FLOAT, false, 0, 0);
  . . .

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, pwgl.floorVertexIndexBuffer);
  gl.drawElements(gl.TRIANGLE_FAN, pwgl.FLOOR_VERTEX_INDEX_BUF_NUM_ITEMS,
                       gl.UNSIGNED_SHORT, 0);
}

function drawCube(texture) {
  . . .

  // Bind normal buffer
  gl.bindBuffer(gl.ARRAY_BUFFER, pwgl.cubeVertexNormalBuffer);
  gl.vertexAttribPointer(pwgl.vertexNormalAttributeLoc,
                        pwgl.CUBE_VERTEX_NORMAL_BUF_ITEM_SIZE,
                        gl.FLOAT, false, 0, 0);

  . . .

  // Bind index buffer and draw cube
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, pwgl.cubeVertexIndexBuffer);
  gl.drawElements(gl.TRIANGLES, pwgl.CUBE_VERTEX_INDEX_BUF_NUM_ITEMS,
                       gl.UNSIGNED_SHORT, 0);
}



function draw() {

   . . .

   // draw floor
   uploadModelViewMatrixToShader();
   uploadProjectionMatrixToShader();
   uploadNormalMatrixToShader();
   . . .
   drawFloor();

   // Draw table
   . . .
   uploadModelViewMatrixToShader();
   uploadNormalMatrixToShader();
   drawTable();
   . . .

   // Draw cube
   . . .
   uploadModelViewMatrixToShader();
   uploadNormalMatrixToShader();
   drawCube(pwgl.boxTexture);
   . . .

}

. . .

function init() {
   // Initialization that is performed during first startup and when the
   // event webglcontextrestored is received is included in this function.
   setupShaders();
   setupBuffers();
```

```
    setupLights();
    setupTextures();
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.enable(gl.DEPTH_TEST);

    // Initialize some varibles for the moving box
    pwgl.x = 0.0;
    pwgl.y = 2.7;
    pwgl.z = 0.0;
    pwgl.circleRadius = 4.0;
    pwgl.angle = 0;

    // Initialize some variables related to the animation
    pwgl.animationStartTime = undefined;
    pwgl.nbrOfFramesForFPS = 0;
    pwgl.previousFrameTimeStamp = Date.now();


    mat4.perspective(60, gl.viewportWidth / gl.viewportHeight,
                        1, 100.0, pwgl.projectionMatrix);
    mat4.identity(pwgl.modelViewMatrix);
    mat4.lookAt([8, 12, 8],[0, 0, 0], [0, 1,0], pwgl.modelViewMatrix);
}


</script>

</head>
<body onload="startup();">
<canvas id="myGLCanvas" width="500" height="500"></canvas>
<div id="fps-counter"> FPS: <span id="fps">--</span></div>
</body>
</html>
```