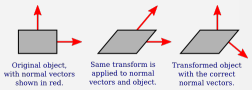
Lighting and Shading in WebGL

- programmable pipeline means programmer can specify any lighting and shading models they want
  - lighting model = calculation of illumination for a surface point
  - shading model = interpolation technique across a polygon
- per-vertex effects go into vertex shader
- per-pixel effects go into fragment shader
- shader parameters are determined by what your lighting equation needs
- uniforms vs attributes are determined by whether values are per-vertex or per-primitive
- vertex shader gets coordinates in OC
  - applies modelview and projection to compute `gl_Position` in CC
- lighting computations are commonly done in EC

Transforming Normals

- surface normals are defined in object coordinates
  - i.e. without any transforms applied
- lighting is done in eye coordinates
  - i.e. with modeling and viewing transforms applied
- the normal for a transformed surface is not necessarily the same as the transformed normal



- can compute the normal matrix from the modelview matrix
  - `mat3.normalFromMat4(normalmat,modelview)` – sets `normalmat` (must have been previously allocated)
    - 3x3 is sufficient because 4x4 was needed only to support translation, which doesn't affect surface normals
  - transformed normals must be normalized – can be done in the shader
    - scaling affects length

Simple Lighting and Shading

- same material on both sides of the polygon
- single viewpoint light
  - directional, along negative z axis (EC)
  - white
- only diffuse reflection
- flat or Gouraud shading
  - lighting equation applied per-vertex
  - flat vs Gouraud determined by the vertex normals supplied

points along the camera's lookat vector

directional lights assume orthographic projections

(orthographic projection matrix involves translate, scale, flip z – doesn't affect directional viewpoint light)

- lighting model $I = md\, I_s\, max(0, (N \cdot L))$
- parameters needed
  - material – diffuse color (md)
  - N – surface normal (OC)
  - modelview and normal matrices

L = (0,0,1) [towards light]
N·L = (0,0,N_z)

substitute abs(N·L) to compute for lit side

Simple Lighting – Vertex Shader

```
attribute vec3 a_coords;            // Object coordinates for the vertex.
uniform mat4 modelviewProjection;   // Combined transformation matrix.
uniform bool lit;                   // Should lighting be applied?
uniform vec3 normal;                // Normal vector (in object coordinates).
uniform mat3 normalMatrix;          // Transformation matrix for normal vectors.
uniform vec4 color;                 // Basic (diffuse) color.
varying vec4 v_color;               // Color to be sent to fragment shader.
void main() {
    vec4 coords = vec4(a_coords,1.0);
    gl_Position = modelviewProjection * coords;
    if (lit) {
        vec3 N = normalize(normalMatrix*normal); // Transformed unit normal.
        float dotProduct = abs(N.z);
        v_color = vec4( dotProduct*color.rgb, color.a );
    }
    else {
        v_color = color;
    }
}
```

L = (0,0,1) [towards light]
N·L = N_z

computes color for the lit side of the polygon

diffuse term

lighting equation only applies to RGB components of color, not alpha

needed to transform geometry (OC→CC), pass color to fragment shader

flat shading (normal is `uniform`)

$I = md\, I_s\, max(0, (N \cdot L))$

Simple Lighting – Perspective

point lights are appropriate for perspective projections

- single viewpoint light
  - point light at (0,0,0) [EC]

$$I = md\, I_s \max(0, (N \cdot L))$$

L = -eyeCoords [eyeCoords = vertex coords in EC]

need separate modelview to transform a_coords from OC to EC for lighting

```
attribute vec3 a_coords;      // Object coordinates for the vertex.
uniform mat4 modelview;       // Modelview transformation matrix
uniform mat4 projection;      // Projection transformation matrix.
uniform bool lit;             // Should lighting be applied?
uniform vec3 normal;          // Normal vector (in object coordinates).
uniform mat3 normalMatrix;    // Transformation matrix for normal vectors.
uniform vec4 color;           // Basic (diffuse) color.
varying vec4 v_color;         // Color to be sent to fragment shader.
void main() {
    vec4 coords = vec4(a_coords,1.0);
    vec4 eyeCoords = modelview * coords;
    gl_Position = projection * eyeCoords;
    if (lit) {
        vec3 L = normalize( - eyeCoords.xyz ); // Points to light.
        vec3 N = normalize(normalMatrix*normal); // Transformed unit normal.
        float dotProduct = abs( dot(N,L) );
        v_color = vec4( dotProduct*color.rgb, color.a );
    }
    else {
        v_color = color;
    }
}
```


Adding Specular Reflection

allows for Gouraud shading

$$I = md\, I_s \max(0, (N \cdot L)) + ms\, I_s \max(0, (R \cdot V))^{mh}$$

parameters for specular component

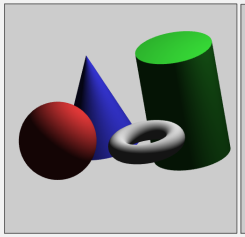
shader uses fixed light color [diffuse (0.8,0.8,0.8), specular (0.4,0.4,0.4)] but position/direction can be specified (EC)

uses OpenGL convention of w = 1 for position, w = 0 for direction

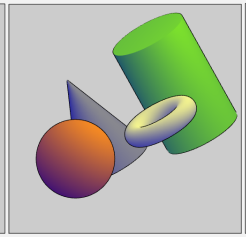
```
attribute vec3 a_coords;
attribute vec3 a_normal;
uniform mat4 modelview;
uniform mat4 projection;
uniform mat3 normalMatrix;
uniform vec4 lightPosition;
uniform vec4 diffuseColor;
uniform vec3 specularColor;
uniform float specularExponent;
varying vec4 v_color;
void main() {
    vec4 coords = vec4(a_coords,1.0);
    vec4 eyeCoords = modelview * coords;
    gl_Position = projection * eyeCoords;
    vec3 N, L, R, V;  // Vectors for lighting equation.
    N = normalize( normalMatrix*a_normal );
    if ( lightPosition.w == 0.0 ) { // Directional light.
        L = normalize( lightPosition.xyz );
    }
    else { // Point light.
        L = normalize( lightPosition.xyz/lightPosition.w - eyeCoords.xyz );
    }
    R = -reflect(L,N);
    V = normalize( -eyeCoords.xyz);  // (Assumes a perspective projection.)
    if ( dot(L,N) <= 0.0 ) {
        v_color = vec4(0,0,0,1);  // The vertex is not illuminated.
    }
    else {
        vec3 color = 0.8 * dot(L,N) * diffuseColor.rgb;
        if (dot(R,V) > 0.0) {
            color += 0.4 * pow(dot(R,V),specularExponent) * specularColor;
        }
        v_color = vec4(color, diffuseColor.a);
    }
}
```


Other Lighting Models

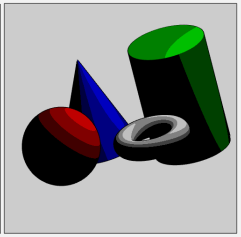
- the programmable pipeline means we can use different lighting models
  - e.g. more sophisticated models for greater photorealism
  - e.g. non-photorealistic models for special kinds of effects



standard lighting (ambient, diffuse, specular)



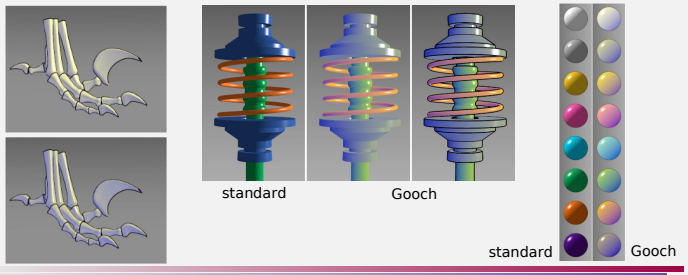
Gooch (cool-to-warm) shading



cel shading

Gooch Shading (Cool-to-Warm Shading)

- non-photorealistic rendering technique often used in technical illustrations
  - reduces contrast in the shading so that silhouettes and edge lines are distinct
  - color shift helps preserve sense of shape in spite of lower contrast in shading



https://www.cs.utah.edu/~shirley/papers/gooch98.pdf

Gooch Shading (Cool-to-Warm Shading)

- standard diffuse term

$$I = md\, I_s \max(0, (N \cdot L))$$

- Gooch diffuse term

$$I = \left(\frac{1+N\cdot(-L)}{2}\right) k_{cool} + \left(1 - \frac{1+N\cdot(-L)}{2}\right) k_{warm}$$

$$k_{cool} = (0, 0, b) + \alpha\, md$$

$$k_{warm} = (y, y, 0) + \beta\, md$$

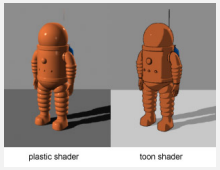
(also include the standard ambient and specular terms)

– should be used with Phong (per-pixel) shading

  - b, y determine the strength of the temperature shift
  - α, β determine the prominence of the material diffuse color

Cel Shading

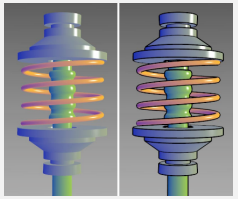
- non-photorealistic rendering technique reminiscent of comic books or cartoons



- implementation
  - use standard (or desired) lighting model, then *quantize* the result (reduce to one of a small palette of colors)
    - simple quantization – divide RGB values into equal-sized ranges
      - floor(color.rgb*steps)/steps where *steps* is the number of color levels
    - fixed palette – choose closest palette color according to Euclidean distance $\sqrt{(r_1-r_2)^2+(g_1-g_2)^2+(b_1-b_2)^2}$
  - should be used with Phong (per-pixel) shading

Edge Highlighting

- illustrations often emphasize edges by drawing them in black
- strategy – two passes
  - pass 1: draw back faces in black
    - alternative #1: draw wireframe with thick lines
    - alternative #2: draw solid triangles using vertices displaced along vertex normals
      - vertex shader works with displaced coordinates
    - fragment shader sets gl_FragColor to black
  - pass 2: draw front faces with normal illumination
  - implementation
    - shaders support both passes, with parameter to indicate which
    - draw primitive twice in JavaScript program, with culling so only desired faces are drawn



```
vec4 coords = vec4(a_coords,1.0);
if ( u_silhouette ) {
    coords += vec4(.03*a_normal,0);
}
```

`a_coords`, `a_normal` are OC vertex coords, normal

`u_silhouette` is a boolean parameter specifying whether to draw a silhouette or not

body of the `if` displaces the original coordinates a small amount along the normal

Edge Highlighting

- utilize culling to draw only back or front faces in each pass
  - enable culling
  - set the cull face to the side you don't want to draw
- JavaScript settings
  - `gl.enable(gl.CULL_FACE)` – discard one set of faces (don't even bother to draw)
  - `gl.cullFace(gl.BACK)`, `gl.cullFace(gl.FRONT)` – which side to cull (default is `gl.BACK`)