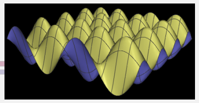
Handling Lighting in WebGL

- two-sided lighting
- materials and lights – structs
- multiple lights – arrays
- positioning lights (EC, WC, OC)

Two-Sided Lighting



- can add parameters to shaders for different front and back materials
- fragment shader variable `gl_FrontFacing` indicates which face is being drawn
- for per-vertex shading, vertex shader must compute both colors (and pass to fragment shader)
  - fragment shader uses `gl_FrontFacing` to determine which to use
- for per-pixel shading, vertex shader can pass outward normal
  - light, materials properties are `uniforms` and are available directly to the fragment shader
  - fragment shader uses `gl_FrontFacing` to determine whether to flip normal, then computes color for the desired side

GLSL Structs

defining a struct is convenient when there are multiple related shader parameters, especially if they are passed to helper functions or duplicated (e.g. multiple lights or front and back materials)

- working with GLSL structs
  - in GLSL

```
struct MaterialProperties {
    vec3 diffuseColor;
    vec3 specularColor;
    float specularExponent;
};
```

```
vec3 lightingEquation( LightProperties light,
                       MaterialProperties material,
                       vec3 eyeCoords, // Eye coordinates for the point.
                       vec3 N, // Normal vector to the surface.
                       vec3 V  // Direction to viewer.
                     ) {
    vec3 L, R; // Light direction and reflected light direction.
    if ( light.position.w == 0.0 ) { // directional light
        L = normalize( light.position.xyz );
    }
    else { // point light
        L = normalize( light.position.xyz/light.position.w - eyeCoords );
    }
    if (dot(L,N) <= 0.0) { // light does not illuminate the surface
        return vec3(0.0);
    }
    vec3 reflection = dot(L,N) * light.color * material.diffuseColor;
    R = -reflect(L,N);
    if (dot(R,V) > 0.0) { // ray is reflected toward the viewer
        float factor = pow(dot(R,V),material.specularExponent);
        reflection += factor * material.specularColor * light.color;
    }
    return reflection;
}
```


GLSL Structs

```
struct MaterialProperties {
    vec4 diffuseColor;      //
    vec3 specularColor;
    vec3 emissiveColor;
    float specularExponent;
};
```

```
uniform MaterialProperties material;
```

- working with GLSL structs
  - in javascript – each field of the struct is a separate parameter
    - need location variable (+ buffer for attributes)
      - convenient to use a javascript object to group, rather than multiple separate variables

```
u_material = {
    diffuseColor: gl.getUniformLocation(prog, "material.diffuseColor"),
    specularColor: gl.getUniformLocation(prog, "material.specularColor"),
    emissiveColor: gl.getUniformLocation(prog, "material.emissiveColor"),
    specularExponent: gl.getUniformLocation(prog, "material.specularExponent")
};
```

- need to set

```
gl.uniform3f( u_material.specularColor, 0.1, 0.1, 0.1 );  // specular properties don't change
gl.uniform1f( u_material.specularExponent, 16 );
gl.uniform3f( u_material.emissiveColor, 0, 0, 0);  // default, will be changed temporarily for some objects
```

GLSL Arrays

- working with GLSL arrays
  - in GLSL
    - syntax is like Java

```
uniform LightProperties lights[4];
```

```
vec3 color = vec3(0.0);  // Start with black (all color components zero).
for (int i = 0; i < 4; i++) {  // Add in the contribution from light i.
    if (lights[i].enabled) { // Light can only contribute color if enabled.
        color += lightingEquation( lights[i], material,
                                   eyeCoords, normal, viewDirection );
    }
}
```


GLSL Arrays

```
uniform LightProperties lights[4];
```

- working with GLSL arrays
  - in javascript – each element of the array is a separate parameter
    - need location variable (+ buffer for attributes)
      - convenient to use a javascript array to group, rather than multiple separate variables

```
u_lights = new Array(4);
for (let i = 0; i < 4; i++) {
    u_lights[i] = {
        enabled: gl.getUniformLocation(prog, "lights[" + i + "].enabled"),
        position: gl.getUniformLocation(prog, "lights[" + i + "].position"),
        color: gl.getUniformLocation(prog, "lights[" + i + "].color"),
        spotDirection: gl.getUniformLocation(prog, "lights[" + i + "].spotDirection"),
        spotCosineCutoff: gl.getUniformLocation(prog, "lights[" + i + "].spotCosineCutoff"),
        spotExponent: gl.getUniformLocation(prog, "lights[" + i + "].spotExponent"),
        attenuation: gl.getUniformLocation(prog, "lights[" + i + "].attenuation")
    };
}
```

- need to set

```
for (let i = 1; i < 4; i++) { // set defaults for lights
    gl.uniform1i( u_lights[i].enabled, 0 );
    gl.uniform4f( u_lights[i].position, 0, 0, 1, 0 );
    gl.uniform1f( u_lights[i].spotCosineCutoff, 0); // not a spotlight
    gl.uniform3f( u_lights[i].spotDirection, 0,0,-1);
    gl.uniform1f( u_lights[i].spotExponent, 5);
    gl.uniform1f( u_lights[i].attenuation, 0); // no attenuation
    gl.uniform3f( u_lights[i].color, 1,1,1 );
}
```


Positioning Lights

- common usage patterns
  - fixed light with respect to the viewer
    - e.g. viewer light, overhead light – the scene is always illuminated from the same direction regardless of the camera's orientation
    - light position is defined in EC
  - fixed light with respect to the world
    - e.g. street light – the light has a location in the world but appears in different places in the rendered scene depending on the camera's position and orientation
    - light position is defined in WC
  - fixed light with respect to an object
    - e.g. car headlights – the light has a location relative to an object but that object can be in different places in the world
    - light position is defined in OC

Positioning Lights in OpenGL

⚠ syntax/concepts are OpenGL 1.0, not WebGL

- light positions are transformed by the modelview matrix in effect when the position is set using `glLightfv`
- usage patterns
  - fixed light with respect to the viewer – light position in EC
    - set light position while modelview is identity (before any viewing or modeling transforms)
  - fixed light with respect to the world – light position in WC
    - set light position after the viewing transform but before any modeling transforms
  - fixed light with respect to an object – light position in OC
    - set light position with the same modeling transform as the object

Positioning Lights in WebGL

- lighting computations are commonly done in EC
- as implemented, must pass EC light coordinates to shaders
  - modelview passed to shaders is the one associated with the current primitive, not the light(s)

- JavaScript program must apply appropriate transform to lights before passing to shaders

```
vec4.transformMat4( transformedVector, originalVector, matrix );
```

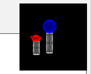
  - lights fixed relative to an object (OC)
    - transformation is modelview (for light's modeling transform)
  - lights fixed relative to the world (WC)
    - transformation is viewing transform (modeling transform is identity)
  - lights fixed relative to the viewer (EC)
    - transformation is identity

Caveat

- all lights in effect when drawing a primitive must be set before the `gl.drawArrays` or `gl.drawElements` call
  - typically means that lights must be set before any geometry is drawn

define all lights (i.e. set shader parameters for lights) here

- for lights fixed relative to the viewer (EC) – matrix is identity
- for lights fixed relative to the world (WC) – matrix is the viewing transform
  - use current `modelview` since no additional transformations have been applied
- for lights fixed relative to an object (OC) – matrix is the light's modeling transform
  - save current `modelview`
  - duplicate the modeling transforms to position the object and the light relative to the object
  - transform the OC light position using the current `modelview`
  - restore the previous `modelview`



```
// viewing and projection
modelview = rotator.getViewMatrix();
projection = mat4.create();
mat4.ortho(projection, -5, 5, -5, 5, 5, 15);

// move the "floor" down for the whole scene
mat4.translate(modelview, modelview, [0, -2, 0]);

// ---- teapot and pedestal ---------------------------
stack.push(mat4.clone(modelview));

mat4.translate(modelview, modelview, [-3, 0, 0]);

stack.push(mat4.clone(modelview));
drawPedestal(0.5, 2, [1, 1, 1]);
modelview = stack.pop();

stack.push(mat4.clone(modelview));
mat4.translate(modelview, modelview, [0, 1, 0]);
drawTeapot(2, [1, 0, 0]);
modelview = stack.pop();

modelview = stack.pop();

// ---- sphere and pedestal ---------------------------
stack.push(mat4.clone(modelview));

mat4.translate(modelview, modelview, [-2, .5, -2]);

stack.push(mat4.clone(modelview));
drawPedestal(0.5, 3, [1, 1, 1]);
modelview = stack.pop();

stack.push(mat4.clone(modelview));
mat4.translate(modelview, modelview, [0, 2.5, 0]);
drawSphere(1, [0, 0, 1]);
modelview = stack.pop();

modelview = stack.pop();
```