CS 432 – Interactive Computer Graphics

Lecture 6 – Part 2

Lighting and Shading in OpenGL

OpenGL Shading

- Need
  - Vertex Normals
  - Material properties
  - Lights
  - Position of viewer

Specifying a Point Light Source

- For each light sources we can set an RGBA for the diffuse, specular, and ambient components and for the position

```
vec4 diffuse = vec4(…);
vec4 ambient = vec4(…);
vec4 specular = vec4(…);
vec4 light_pos = vec4(…);
```

Distance and Direction

- The source colors are given in RGBA
- The position is given in homogenous coordinates
  - If $w = 1$, we are specifying a finite location
  - If $w = 0$ we are specifying a parallel source with a given direction vector
- The coefficients in distance terms are usually quadratic $\frac{1}{d^2}$ where d is the distance from the point being rendered to the light source

Material Properties

- Material properties should match the terms in the light model
  - w component gives opacity

```
vec4 ambient = vec4(…);
vec4 diffuse = vec4(…);
vec4 specular = vec4(…)
GLfloat shine = 100.0
```

Emissive Term

- We can simulate a light source by giving a material an *emissive* component
- This component is unaffected by any sources or transformations

Our Drawable Objects

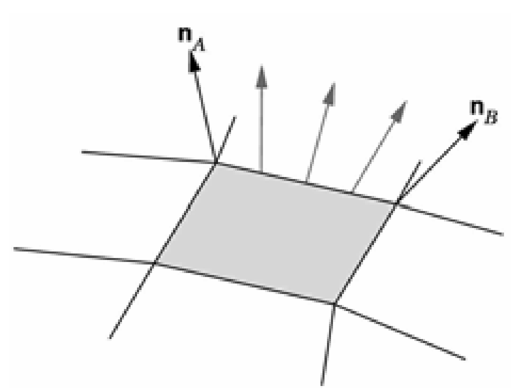
- Note: Now you may want/need to store additional things in your drawable objects
  - Normals (put in some buffer)
  - Material properties

Gouraud and Phong Shading

- There are two common ways to do shading.
  - Based on where to interpolate the shading
  - Ends up with doing our calculations in the vertex shader vs fragment shader
- Phong Shading
  - Apply modified Phong model at each vertex to get vertex shade
  - Interpolate vertex shades across each polygon (via passing to fragment shader)
- Gouraud Shading
  - Interpolate vectors need for shading by passing them to fragment shader
  - Apply modified Phong model at each *fragment*

Gouraud vs Phong

- If the polygon mesh approximates surfaces with high curvature, Gouraud may look smoother
- Gouraud shading requires much more work
  - Can now be done reasonably in fragment shader



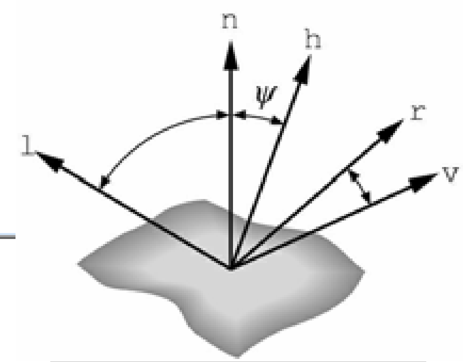
Phong Shading

Vertex Lighting Shaders 1

- The vertex shader… continued on next slide

```
#version 150

in  vec4 vPosition;
in  vec3 vNormal;

out vec4 color;

uniform mat4 model_matrix;
uniform mat4 camera_matrix;
uniform mat4 proj_matrix;


uniform vec4 lightPos;
uniform vec4 lightAmbient, lightDiffuse, lightSpecular;
uniform vec4 matAmbient, matDiffuse, matSpecular;
uniform float matAlpha;
```

Vertex Lighting Shaders II




```
void main()
{

        //the vertex in camera coordinates
        vec3 pos = (camera_matrix*model_matrix*vPosition).xyz;

        vec3 lightPosInCam = (camera_matrix*lightPos).xyz;


        //the ray from the vertex towards the light
        vec3 L = normalize(lightPosInCam.xyz-pos);
        float dist = 1.0;

        //the ray from the vertex towards the camera
        vec3 E = normalize(vec3(0,0,0)-pos);


        //normal in camera coordinates
        vec3 N = normalize(camera_matrix*model_matrix*vec4(vNormal,0)).xyz;

        vec3 H = normalize(L+E);


        vec4 ambient = lightAmbient*matAmbient;

        float Kd = max(dot(L,N),0.0);
        vec4 diffuse = Kd*lightDiffuse*matDiffuse;

        float Ks = pow(max(dot(N,H),0.0), matAlpha);
        vec4 specular = Ks*lightSpecular*matSpecular;
        if(dot(L,N)< 0.0)
                specular = vec4(0,0,0,1);

        color = (ambient+diffuse+specular)*(1/pow(dist,2));
        color.a = 1.0;



        gl_Position = proj_matrix*camera_matrix*model_matrix*vPosition;


}
```

$$\mathbf{h} = \frac{1}{\|\mathbf{v}+\mathbf{I}\|}[\mathbf{v}+\mathbf{I}]$$

$$I_a = L_a k_a$$

$$I_d = R_d L_d k_d = (l \cdot n) L_d k_d$$

$$I_s = R_s L_s k_s = (h \cdot n)^{\alpha} L_s k_s$$

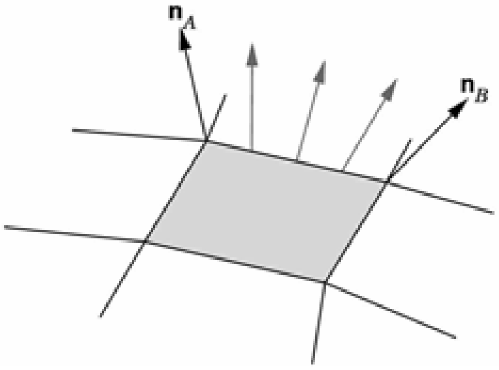
Vertex Lighting Shaders IV

- The fragment shader

```
//frag shader
//just the default passthrough
in vec4 color;
out vec4 fColor;

void(){
   fColor = color;
}
```

Shading by Fragment

- Alternatively we can do per-fragment shading
- We can compute the pass the vectors (normal, to eye, to light) from the vertex shader to the fragment shader
  - Resulting in interpolated values
- Then in the fragment shader we compute the fragment's color



Fragment Lighting Shaders I

- The vertex shader

```
#version 150

in  vec4 vPosition;
in  vec3 vNormal;

out vec3 fN;
out vec3 fE;
out vec3 fL;

uniform mat4 model_matrix;
uniform mat4 camera_matrix;
uniform mat4 proj_matrix;


uniform vec4 lightPos;


void main()
{
        //the vertex in camera coordinates
        vec3 pos = (camera_matrix*model_matrix*vPosition).xyz;

        //the light in camera coordinates
        vec3 lightPosInCam = (camera_matrix*lightPos).xyz;

        //normal in camera coordinates
        fN = normalize(camera_matrix*model_matrix*vec4(vNormal,0)).xyz;

        //the ray from the vertex towards the camera
        fE = normalize(vec3(0,0,0)-pos);

        //the ray from the vertex towards the light
        fL = normalize(lightPosInCam.xyz-pos);

        gl_Position = proj_matrix*camera_matrix*model_matrix*vPosition;

}
```

Fragment Lighting Shaders III

- The fragment shader…

```
#version 150

in vec3 fN;
in vec3 fL;
in vec3 fE;

uniform vec4 lightAmbient, lightDiffuse, lightSpecular;
uniform vec4 matAmbient, matDiffuse, matSpecular;
uniform float matAlpha;

out vec4 fColor;


void main()
{
        vec3 N = normalize(fN);
        vec3 E = normalize(fE);
        vec3 L = normalize(fL);

        vec3 H = normalize(L+E);

        vec4 ambient = lightAmbient*matAmbient;

        float Kd = max(dot(L,N),0.0);
        vec4 diffuse = Kd*lightDiffuse*matDiffuse;

        float Ks = pow(max(dot(N,H),0.0),matAlpha);
        vec4 spec = Ks*lightSpecular*matSpecular;

        if(dot(L,N)<0.0)
                spec = vec4(0,0,0,1);


        fColor = ambient + diffuse + spec;
        fColor.a = 1.0;

}
```

Performance Notes

- If possible, it may be advantageous to pass in `prodAmbient=lightAmbient*matAmbient`, etc.. Instead of the light/material properties
  - This way it would only be computed once on the CPU
  - As opposed to being computed for every vertex (or fragment)

Example: Shading a Sphere

- Let's play with shading by shading a sphere
- First we need to model/build a sphere!
- In a previous lecture and assignment you saw two ways to generate a sphere
  - Parametrically
  - Recursive Subdivision
- Either way, in the end we have vertices used to specify triangles.

Example: Building a Sphere (header file)

```
class Sphere:public Drawable {
public:
    Sphere();
    void draw(Camera, vector<Light>);
    void setMaterial(vec4, vec4, vec4, float);
    ~Sphere();
private:
    //(4 triangular faces per tetrahedron)^(numDivisions+1)*3 vertices per triangle
    static const unsigned int numVertices = 3072;
    vec4 vertexLocations[numVertices];
    vec3 vertexNormals[numVertices];
    GLuint vpos, npos, mmpos, vmpos, pmpos, diffuse_loc, spec_loc, ambient_loc, alpha_loc;
    vec4 diffuse, specular, ambient;
    float shine;

    unsigned int index;
    void build();
    void assignGouradVertices();
    float sqrt2, sqrt6;
    void tetrahedron(int);
    void divideTriangle(vec4, vec4, vec4,int);
    void triangle(vec4, vec4, vec4);
    vec4 unit(vec4);

};
```

Example: Lights

```
class Light {
    public:
        Light(vec4 p, vec4 a, vec4 s, vec4 d) : position(p),
            ambient(a),  specular(s), diffuse(d) {}
        vec4 getPosition(){return position;}
        vec4 getAmbient(){return ambient;}
        vec4 getDiffuse(){return diffuse;}
        vec4 getSpecular(){return specular;}
    private:
        vec4 position;
        vec4 ambient;
        vec4 diffuse;
        vec4 specular;
};
```

Example: Building a Sphere (Building)

```
Sphere::Sphere() {
    index = 0;
    glGenVertexArrays(1, &VAO);
    // Create and initialize a buffer object
    glGenBuffers(1, &VBO);

    build();

    program = InitShader("../vshader00_v150.glsl", "../fshader00_v150.glsl");
    glUseProgram(program);
    vpos = glGetAttribLocation(program, "vPosition");
    npos = glGetAttribLocation(program, "vNormal");
    mmpos = glGetUniformLocation(program, "model_matrix");
    vmpos = glGetUniformLocation(program, "view_matrix");
    pmpos = glGetUniformLocation(program, "proj_matrix");
    diffuse_loc = glGetUniformLocation(program, "matDiffuse");
    spec_loc = glGetUniformLocation(program, "matSpecular");
    ambient_loc = glGetUniformLocation(program, "matAmbient");
    alpha_loc = glGetUniformLocation(program, "matAlpha");
```

Example: Building a Sphere (Building, Continued…)

```
glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexLocations)+sizeof(vertexNormals), NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertexLocations), vertexLocations);
glBufferSubData(GL_ARRAY_BUFFER, 0+sizeof(vertexLocations), sizeof(vertexNormals), vertexNormals);

glEnableVertexAttribArray(vpos);
glVertexAttribPointer(vpos, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

glEnableVertexAttribArray(npos);
glVertexAttribPointer(npos, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(vertexLocations)));

translation = Translate(0, 1, -4);
modelmatrix = translation;
```

Example: Building a Sphere (Building)

```
void Sphere::build() {
	sqrt2 = (float)sqrt(2.0);
	sqrt6 = (float)sqrt(6.0);

	index = 0;
	tetrahedron(4);
	//assignGouradVertices();
}
```

```
void Sphere::triangle(vec4 ai, vec4 bi, vec4 ci) {
	vec3 a = vec3(ai.x,ai.y,ai.z);
	vec3 b = vec3(bi.x, bi.y, bi.z);
	vec3 c = vec3(ci.x, ci.y, ci.z);

	vec3 N = normalize(cross(b - a, c - a));


	vertexLocations[index] = a;
	vertexNormals[index] = N;
	index++;

	vertexLocations[index] = b;
	vertexNormals[index] = N;
	index++;

	vertexLocations[index] = c;
	vertexNormals[index] = N;
	index++;
}
```

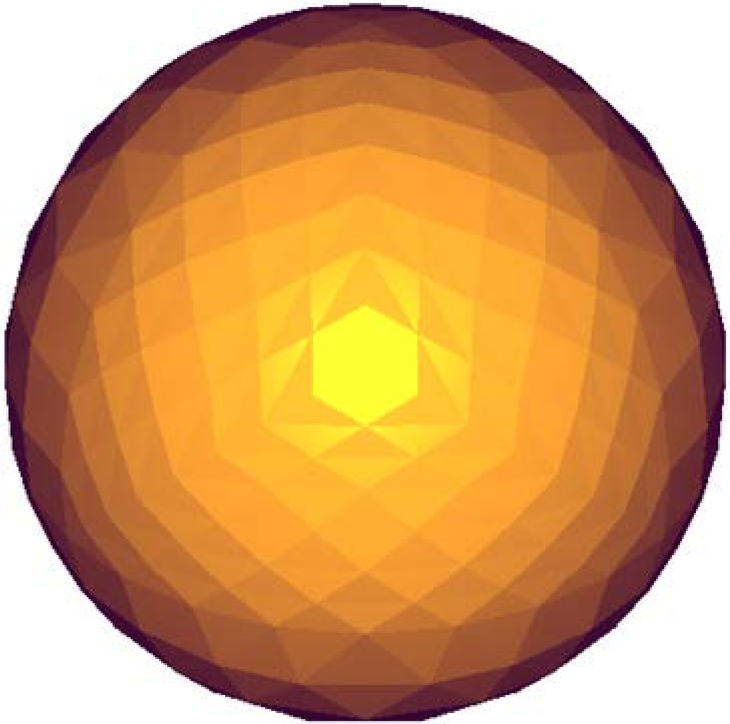
Example: Building a Sphere (material)

```
void Sphere::setMaterial(vec4 a, vec4 d, vec4 s, float sh) {
     diffuse = d;
     specular = s;
     ambient = a;
     shine = sh;
}
```

Example: Building a Sphere (draw)

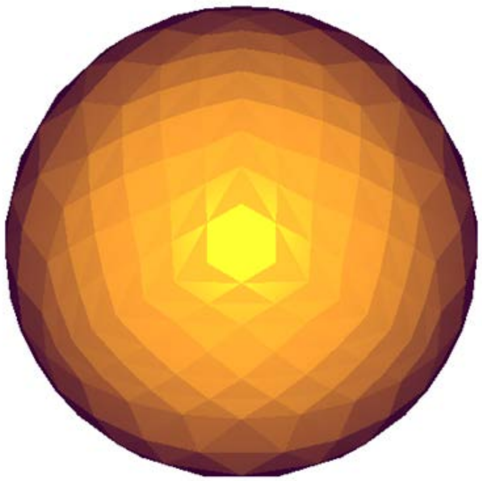
```
void Sphere::draw(Camera cam, vector<Light> lights) {
    glBindVertexArray(VAO);
    glUseProgram(program);

    glUniformMatrix4fv(mmpos, 1, GL_TRUE, modelmatrix);
    glUniformMatrix4fv(vmpos, 1, GL_TRUE, cam.getViewMatrix());
    glUniformMatrix4fv(pmpos, 1, GL_TRUE, cam.getProjectionMatrix());
    glUniform4fv(diffuse_loc, 1, diffuse);
    glUniform4fv(spec_loc, 1, specular);
    glUniform4fv(ambient_loc, 1, ambient);
    glUniform1f(alpha_loc, shine);

    GLuint light_loc = glGetUniformLocation(program, "lightPos");
    glUniform4fv(light_loc, 1, lights[0].getPosition());
    GLuint ambient_loc2 = glGetUniformLocation(program, "lightAmbient");
    glUniform4fv(ambient_loc2, 1, lights[0].getAmbient());
    GLuint diffuse_loc2 = glGetUniformLocation(program, "lightDiffuse");
    glUniform4fv(diffuse_loc2, 1, lights[0].getDiffuse());
    GLuint specular_loc2 = glGetUniformLocation(program, "lightSpecular");
    glUniform4fv(specular_loc2, 1, lights[0].getSpecular());


    glDrawArrays(GL_TRIANGLES, 0,numVertices);

}
```

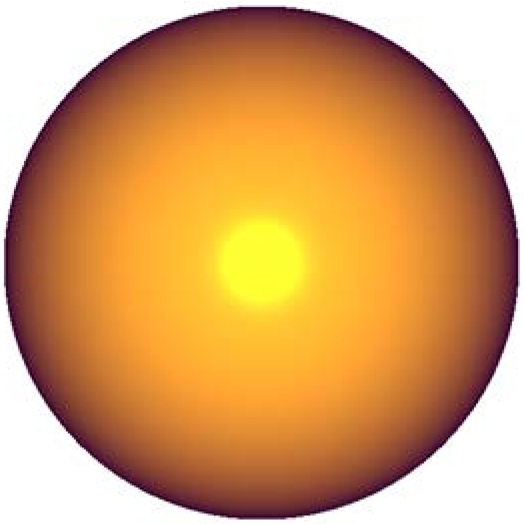
Shading with the Vertex Shader



Ways to make this look better?

- More subdivisions→ more triangles
  - 4 subdivisions = 3,072 vertices
  - 10 subdivisions = 12,582,912 vertices

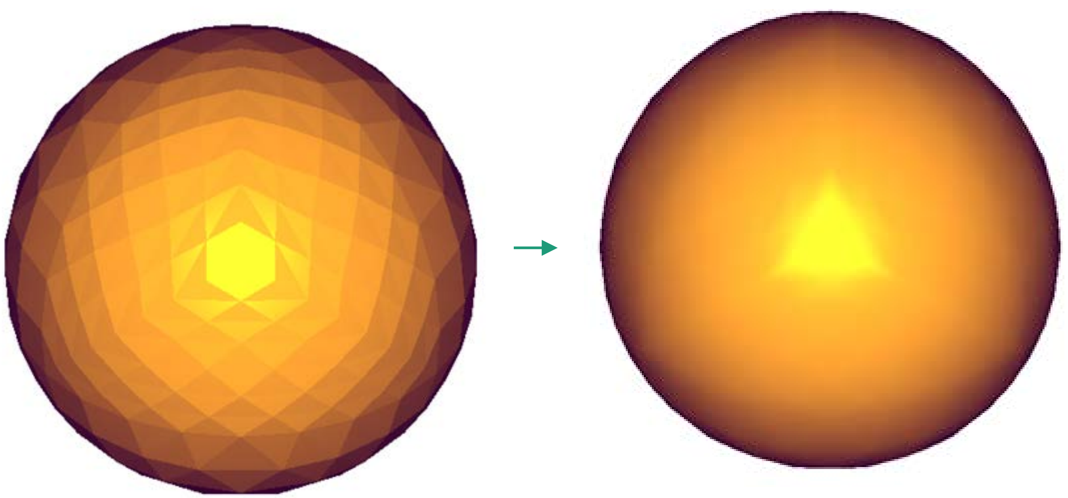




Ways to make this look better?

- Use Gouraud's approach
  - Assign a vertex's normal to be the average of the normals assigned to it by different polygons

```
void assignGouraudVertices(){
    vec3 normalSum[numSphereVertices];
    int counts[numSphereVertices];

    for(int i=0; i<numSphereVertices;i++){
        normalSum[i] = vec3(0,0,0);
        counts[i] = 0;
    }

    for(int i=0;i<numSphereVertices;i++){
        int count=0;
        for(int j=0;j<numSphereVertices;j++){
            if((sphereVertices[i].x==sphereVertices[j].x)&&
                (sphereVertices[i].y==sphereVertices[j].y)&&
                (sphereVertices[i].z==sphereVertices[j].z)){
                count++;
                normalSum[i]+=sphereNormals[j];
            }
        }
        counts[i] = count;
    }
    for(int i=0; i < numSphereVertices; i++)
        sphereNormals[i] = normalSum[i]/counts[i];

}
```



Ways to make this look better?

- Since we have a formula for a sphere, can we assign normals by a formula?
- We know the parametric form of a sphere:

$$p(\theta,\phi) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} cos\theta sin\phi \\ sin\theta sin\phi \\ cos\phi \end{bmatrix}$$

- To get the normal at point p we can take the cross product of the gradient with respect to each parameter:

$$n = \frac{dp}{d\phi} \times \frac{dp}{d\theta}$$

- For our equation of a sphere this becomes

$$n = sin\phi \begin{bmatrix} cos\theta sin\phi \\ sin\theta sin\phi \\ cos\phi \end{bmatrix} = sin\phi \boldsymbol{p}$$

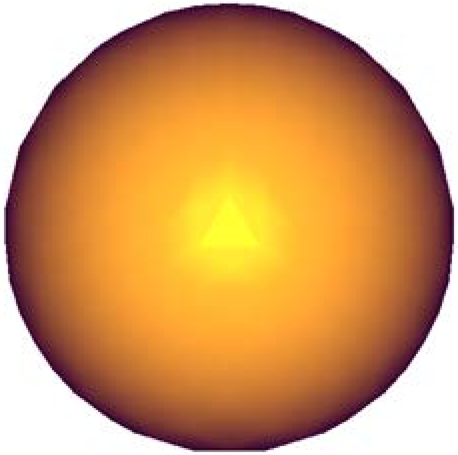
- $sin\phi$ is just a scalar so our unit normal vector is just:

$$\boldsymbol{n} = \boldsymbol{p}$$

Ways to make this look better?

- So let's just assign each vertex's normal to be the

```
void assignParametricNormals(){
    for(int i=0; i < numSphereVertices;i++){
        sphereNormals[i] = normalize(vec3(sphereVertices[i].x, sphereVertices[i].y, sphereVertices[i].z));
    }
}
```



Multiple Lights

- Often we'll have multiple lights in a scene
  - In fact this will be asked of you in a homework assignment.
- Some shader versions allow us to create structures in our shaders
  - However that's not supported in GLSL version 1.3 ☹
- Also some shader versions allow for looping through arrays of structures
  - However this tends to work system-to-system
- So to ensure maximum compatibility, we'll just hard-code in support for multiple lights

Multiple Lights

```
#version 150
…
uniform vec4 lightPos1;
uniform vec4 lightAmbient1, lightDiffuse1, lightSpecular1;
uniform int enabled1;
uniform vec4 lightPos2;
uniform vec4 lightAmbient2, lightDiffuse2, lightSpecular2;
uniform int enabled2;
out vec4 color;
…
void main(){
		vec4 color1 = vec4(0,0,0,0);
		if(enabled1==1)
				color1= //compute color due to light1
		vec4 color2 = vec4(0,0,0,0);
		if(enabled2==1)
				color2= //compute color due to light2
		color=color1+color2;
		color.a=1.0;
}
```