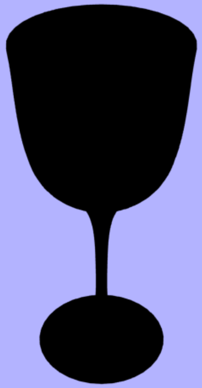
Lighting

Motivation

- Real objects interact with light
- We need to account for this to get realistic scenes

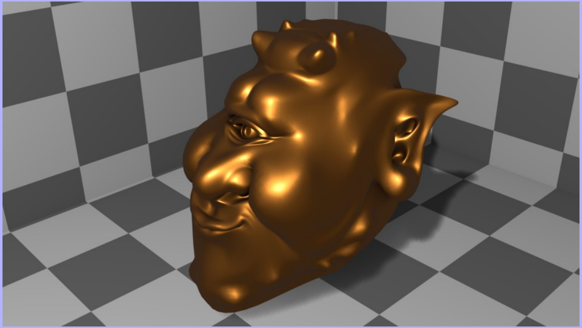
Ambient

- Easiest way: Make it constant over whole surface
- Approximates solution to *global illumination equation*



Problem

- That's not very realistic!
  - Loses lots of detail on object
- Ex: Consider Keenan Crane's "Jerry" model



Ambient

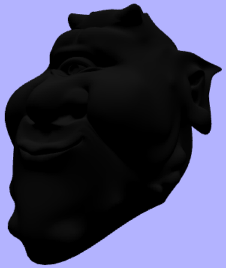
- What happens if we render with constant ambient light?

Problem

- We lose a lot of detail!
- Observe: In real life, we have different shades of light coming from different parts of the environment
  - Blueish from above (sky), brown or green from below (earth)
  - Or lighter from above, darker from below (think a room with overhead lights)
- We'd like to simulate this

Example

- Here's what we're after:



- How can we get this?

Information

- We need to know the *surface normal* at any point on the mesh
- OBJ files contain this information already
  - In the “vn” lines
- Mesh.py: Needs to read in normal data from file
  - Like how we’re doing with positions and texcoords
- And: Put in a buffer
- And: Associate buffer with a vertex puller input
- And: Tell vertex puller about the data type, size, stride
- And: Tell VS about this input

Lighting

- Lighting needs to happen in fragment shader
- But FS can't see the buffer inputs (only VS can do that)
- So we must pass data from VS to FS

VS

- Our basic VS structure:
  - Top of VS:

```
layout(location=0) in vec3 position;
layout(location=1) in vec2 texcoord;
layout(location=2) in vec3 normal;      //<--- NEW
```

  - Make sure vertex puller uses slot 0 for position, 1 for texcoord, and 2 for normal

VS

- VS now needs to pass data to FS
- Declare a global in VS:
  out vec3 v_normal;
- We need to send *world space normal* to FS
- In VS main:

```
vec4 n = vec4( normal, 0.0 );
n = n * worldMatrix;
v_normal = n;
```

FS

- In FS, declare an input (global):
  in vec3 v_normal;
- And then we use it
- Right now, you probably have something like this:

```
out vec4 color;

void main(){
    vec4 c = texture( tex, vec3( v_texcoord, 0.0 ) );
    color = c;
}
```

- How to fit ambient into this?

FS

- We'd typically multiply light color by object color
- First, suppose we have simple flat ambient
- Define a uniform:
  vec3 ambientColor;
  - This would usually be something like (0.1, 0.1, 0.1)

FS

- Now use it:

```
void main(){
    vec4 texColor = texture( tex, vec3( v_texcoord, 0.0 ) );
    color.rgb = ambientColor * texColor.rgb;
    color.a = texColor.a;
}
```

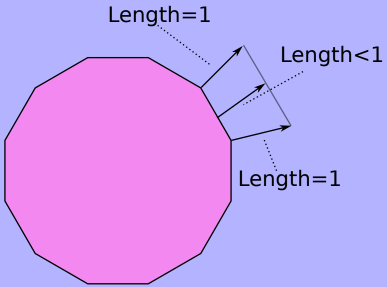
FS

- What about the two-tone ambient?
- Define two uniforms:
  vec3 ambientColor1;
  vec3 ambientColor2;
- Use them:

```
void main(){
    vec4 texColor = texture( tex, vec3( v_texcoord, 0.0 ) );
    vec3 N = normalize(v_normal);
    float mappedY = 0.5*(N.y + 1.0);    //put in 0...1 range
    float ambientColor = mix( ambientColor1, ambientColor2,
        mappedY);
    color.rgb = ambientColor*texColor.rgb;
    color.a = texColor.a;
}
```

FS

- Why do we normalize v_normal?
  - OBJ file had unit-length normals, after all
  - Normal might not be unit length after rasterizer interpolates it

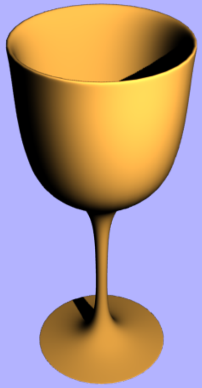


Diffuse

- Ambient is a good start, but it's not enough
  - Notice: It doesn't depend on light source position
  - And it's usually set to a fairly dark value
- We also need to account for light that directly hits an object
- This is *diffuse* reflectance

Diffuse

- Perfectly diffuse reflectors obey *Lambert's cosine law*
  - If surface normal points toward the light $\rightarrow$ Maximum illumination
  - If surface normal points 90° from light $\rightarrow$ No illumination
  - If surface normal points away from light $\rightarrow$ Also no illumination (backface)
  - As normal goes from "pointing toward light" to "pointing perpendicular to light," illumination falls off on a cosine curve



Note

- Can compute illumination percentage at surface point p with $N \cdot L$
  - Where L = vector from p to light source
  - And N = surface normal at p
- If $N \cdot L < 0$: Must clamp to zero
  - Geometric interpretation: Light is on “other side” of object
- One way:

```
float dp = dot(N,L);
if( dp < 0.0 )
    dp = 0.0;
```

- Or:

```
float dp = max( dot(N,L), 0.0 );
```

- Second way is better (branchy code can mean poor performance)

Question

- We already know N
- How can we get L?
  - Need a uniform: Light position:
    vec3 lightPosition;
  - And one for light color:
    vec3 lightColor;
- How do we compute L in the shader?

VS/FS

- Again, we compute lighting in FS
- Need to know surface point in world space
- VS must output it
- Declare a global:
  out vec3 v_worldPos;
- Set it in VS
- Use it in FS

FS

- **Extend the FS:**

```
void main(){
    vec4 texColor = texture( tex, vec3( v_texcoord, 0.0 ) );
    vec3 N = normalize(v_normal);
    float mappedY = 0.5*(N.y + 1.0);    //put in 0...1 range
    float ambientColor = mix( ambientColor1, ambientColor2, mappedY);

    L = lightPos - v_worldPos;  //FROM surface TO light
    L = normalize(L);
    float diffusePct = dot(N,L);
    diffusePct = max( diffusePct, 0.0 );

    color.rgb = texColor.rgb * (ambientColor + lightColor * diffusePct);
    color.a = texColor.a;
}
```

Specular

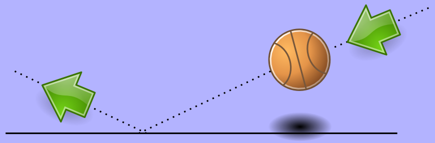
- Most surfaces are not purely diffuse
- Almost every surface looks more-or-less “shiny”
  - Think plastic or greasy hair (yuck!)
- Several ways to approximate this
  - Tradeoff: Complexity/realism for speed

Phong

- 1970’s: Phong Bui Tuong developed this model
- Not physically based
  - Not energy conserving: Total outgoing energy > Total incoming
- But it looks “reasonable”
- And it’s simple to compute

Idea

- Models specular reflection something like basketball bounce
  - Photons come in, bounce off surface, reach the eye



Mirrors

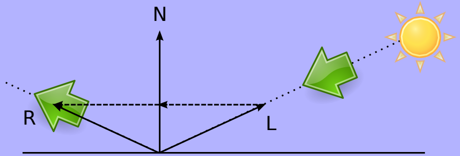
- Only perfect mirrors behave in this way
- On real surfaces, photons get *scattered* somewhat when they hit
- So Phong approximated this effect too

Reflection

- Direction of maximum specularity: *Reflection vector* R
- To find R:

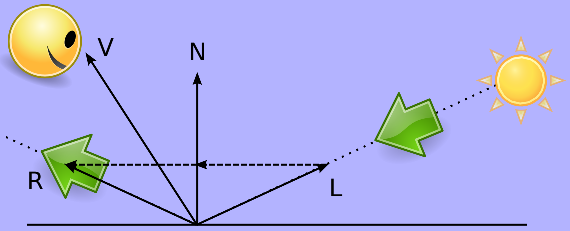
  $$T = (L \cdot N)N$$
  $$Q = T - L$$
  $$R = L + 2Q$$

  - L = "to light", N = normal
- GLSL has intrinsic: reflect(*incoming* , N)
  - vec3 R = reflect(-L, N)



Specular

- Next, we determine how close the viewer's eye is to location where maximum specularity is seen
- Use dot product: $V \cdot R$
  - V = vector to viewer [next slide]
  - R = reflection vector [previous slide]



V

- How can we get V?
  - We need to know viewer's position
  - So add a uniform:
    vec3 eyePos;
  - Set this in Camera.setUniforms()
- Then compute V:
  V = normalize(eyePos - v_worldPos);

Note

- If we used $V \cdot R$ directly: Specular highlight would be too large
  - Looks unrealistic
- So we raise $V \cdot R$ to a power
  - Since $V \cdot R \leq 1$: Raising to any power > 1 will make it smaller
  - Bigger exponent $\rightarrow$ Faster falloff to zero
- This is often called *shininess* exponent
- Thus, specPct = $(V \cdot R)^s$

Putting It Together

- Your FS will then output the combined ambient + diffuse + specular using one of these:
  - color.rgb = texColor.rgb * (ambientColor + lightColor * (diffusePct+specularPct));
    - Modulates specular by texture color $\rightarrow$ Less prominent highlight
  - color.rgb = texColor.rgb * (ambientColor + lightColor * diffusePct) + specularPct * lightColor;
    - Specular is added without regard for object color $\rightarrow$ More obvious highlight

Example

- Exponents of 4, 16, 64
- Notice: *Bigger* exponent gives *smaller* highlight

Note

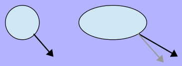
- A few other concerns:
  - $V \cdot R$ could be negative
    - Clamp to zero: Else, NaN's
  - $V \cdot R$ could be > 0 while $N \cdot L < 0$
    - Physically, you can't get specular highlights on surface backside
    - Force specular to zero if diffuse is zero

Color

- Using white for specular color gives “plastic” appearance
- If we want to model surfaces like metal: Use object color for specular (i.e., multiply by texture color)
- Ex: Let diffuse color = 1, 0.71, 0.29
- Compare: Specular of 1,1,1 vs. specular of 1, 0.71, 0.29

Note

- One last note: If our world matrix does a nonuniform scale: We will get an incorrect transformed normal (and thus incorrect lighting)

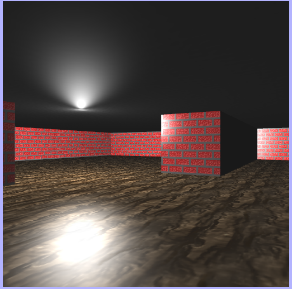


Fix

- If W is the world matrix, instead of computing $n' = nW$, must compute $n' = n(W^{-1})^T$
- Note that for orthogonal matrices, inverse is the transpose
  - So $(W^{-1})^T = W$
- As long as our world matrix only has translation, rotation, and uniform scale, we don't need to worry about this

Assignment

- Add lighting using a single light source at a fixed position in the world.



Bibliography

- D. Brackeen. Programming Games in Java.
- D. Hearn & M. P. Baker Computer Graphics in C.
- F. Luna. Introduction to 3D Game Programming with DirectX 9.0.
- F. Luna. Introduction to 3D Game Programming with DirectX 10.
- Eric Lengyel. Mathematics for 3D Game Programming & Computer Graphics. Charles River Media.

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org