Modeling Light

- Based on, but not the same as, real world lighting
  - Food for thought: why not?
- Thus, whatever you do with light in computer graphics, especially in real time, is ultimately an approximation
- Beyond some core principles (see below), modeling light is a pretty wide-open game, grounded in physics but in some ways ultimately modulated by aesthetics and practicality

Core Calculations

- A lighting model computes the light that hits a polygon, then computes how that polygon reflects this light
- Multiple light sources on a polygon are cumulative: i.e., their RGB values add up, clamped at 1.0
- A material absorbs or reflects light, based on its own color(s) and other properties: this reflection is equivalent to multiplying the light's and material's RGB values (note how having a 0.0–1.0 range is particularly helpful here)

From the Fixed Function Model

The former fixed function OpenGL light model modeled light as three main components:

- Ambient—Light that is so scattered as to appear to be coming from all directions and going in all directions
- Diffuse—Light coming from either a specific direction or a point in space
- Specular—Light that is reflected back in a focused direction; affects the perception of “shininess”

Setting Up a Lit Scene

- Define your objects so that they capture the data that affect how things get lit
  - Light sources: Colors, positions, directions
  - Material settings: Colors, other properties
- Use these settings to perform color calculations within your shaders
- An initial approach would be to assign lit color per vertex, then having OpenGL interpolate the rest

The New “Normal”

- One of the most important geometric aspects of many lighting approaches is the normal vector: that is, the “direction” that a polygon is facing, expressed as (duh) a vector
- The importance of this value makes intuitive sense—see sunrise, noon, and sunset
- Because we care mainly about direction when dealing with normal, they are generally worked with as unit vectors (i.e., lengths equal to 1)

Other Key Issues

Many other issues can get involved, all of which translate into additional shader variables, logic, and computation:

- Attenuation—Does brightness decrease as a function of distance, and if so, by how much?
- Physical properties—“Shininess” for specular reflection is one thing, but there can be many more
- Local vs. global lighting—For simplicity, we calculate lighting per object or vertex; in reality, lit objects also affect each other

Fixed Function "Recipes"

Acknowledging that these categories are all approximations—they abstract out various effects of the "one true" Rendering Equation—these are typically what's done for the original OpenGL fixed-function categories

As a prerequisite, these approaches all presume the presence of some base color—either a uniform color, a varying color interpolated from a color array, or a color that is derived from a texture map at given texture coordinates… or a combination of all three; then, given this base color:

Ambient Light

Note again that the guiding principle for these calculations is that multiple sources of light add up to a final "light contribution," and that value is multiplied to a surface/polygon/material's base color

- Typically ambient light is modeled as a single RGB value since it is, well, ambient (if you did have multiple sources, just add them up as noted above)
- Component-multiply that final RGB value with the color value of the vertex or pixel

Diffuse Light

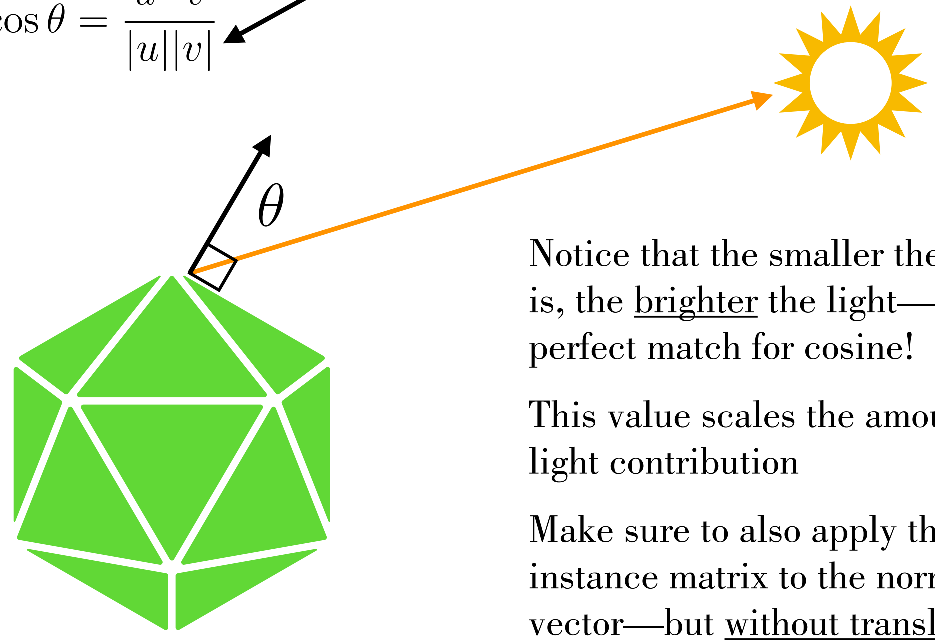
For diffuse light, you must have normal vectors (one per vertex) and light source position & color:

- Normal vectors can be computed by forming vectors from the vertices of each triangle and computing their cross product (see "Normals" in Ghayour/Cantor)
- Apply the instance matrix to the vertex, get the vector from the vertex to the light source, then determine the brightness based on the angle between the normal and light vectors (via dot product—cosine law)

The secret ingredient:

$$\cos\theta = \frac{u \cdot v}{|u||v|}$$

Use unit vectors (`normalize` function in GLSL) so that you don't have to deal with this denominator



Notice that the smaller the angle is, the brighter the light—a perfect match for cosine!

This value scales the amount of light contribution

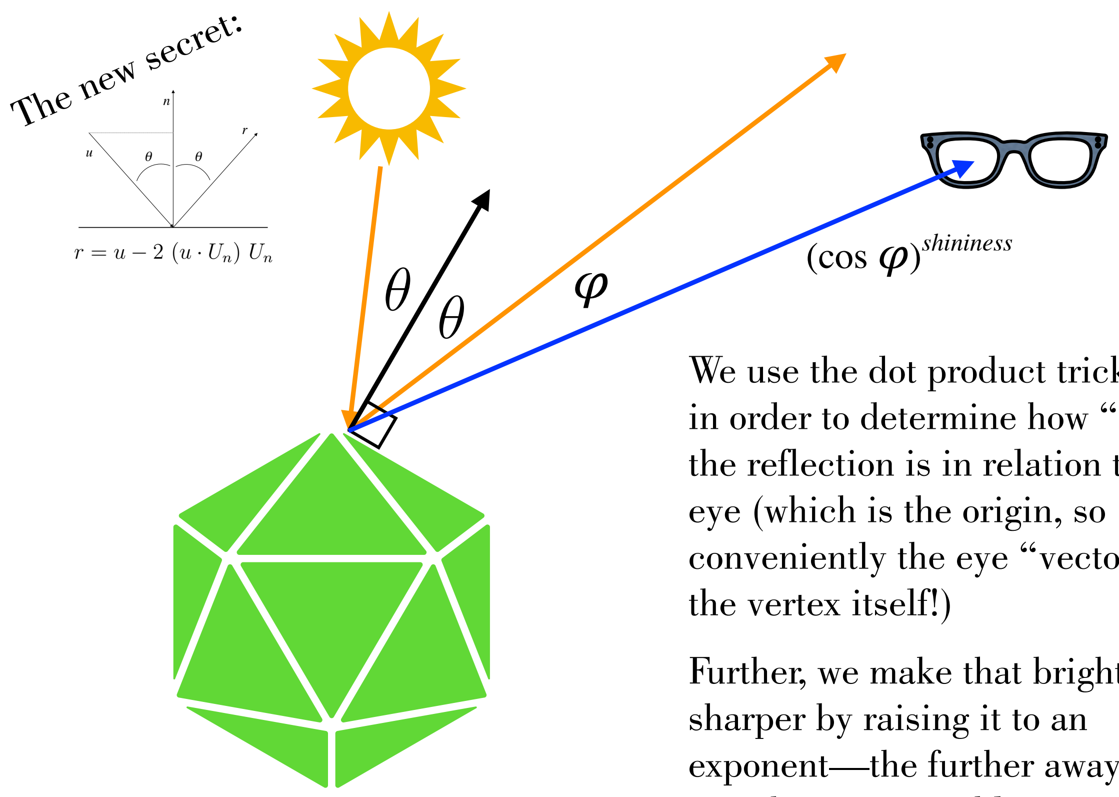
Make sure to also apply the instance matrix to the normal vector—but without translation (i.e., make the 4th element a zero)

Specular Light

The “shine” or “glare” that is associated with specular light is based on how directly <u>reflected</u> light gets into your eye—the more directly reflected, the “shinier” it is

This “shine” diminishes quickly—this can be simulated by a shininess value that we raise as an exponent

- So in additional to normal vectors and light data, we’ll want a `shininess` parameter for customization
- In terms of calculations, we now want to factor in the light vector’s <u>reflection</u> off the normal vector



We use the dot product trick again in order to determine how “bright” the reflection is in relation to the eye (which is the origin, so conveniently the eye “vector” is the vertex itself!)

Further, we make that brightness sharper by raising it to an exponent—the further away from 1.0, the more quickly it recedes

Language Notes

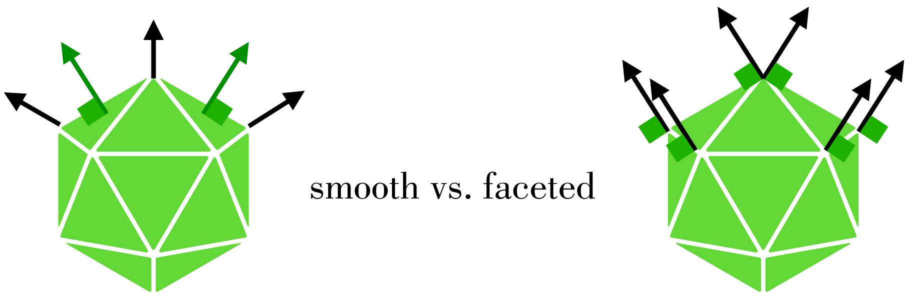
- Remember that matrix multiplications are 4×4 so you will need to use `vec4` for those
- However vertex and vector operations use `vec3`
- Converting from `vec4` to `vec3` means post-pending a component suffix like `.xyz` or `.rgb`
- Converting from `vec3` to `vec4` uses the expression `vec4(x, y, z, w)`—where `w` is almost always `1.0` except when transforming the normal vector—use `0.0` there because you want to lose translations

The helper functions and operations that you need for lighting are already built into GLSL—you just have to put them together correctly:

- `*` indicates scalar-vector or matrix-vector multiplication—inferred based on data type
- `normalize` will compute the unit vector
- `dot` will compute the dot product (cosine of the angle between two unit vectors)—remember that this is a scalar value (data type `float` in GLSL)
- `max` will calculate the maximum of given values
- `pow` will raise a value to a given exponent

Variations on the Theme

The smooth vs. faceted look comes from how the normal vectors are computed—because normals are given per vertex, one produces "smoothness" by making a vertex's normal be the average of the normals of its triangles vs. precisely the normal of each triangle:



The other variation lies in whether the lighting is done per vertex (in the vertex shader) or per pixel (in the fragment shader)—a.k.a. Gouraud vs. Phong shading

- With Gouraud (per-vertex) shading, the color is computed per vertex and that color is then interpolated across the triangles (remember the Dye Hards?)
- With Phong (per-pixel) shading, the lighting calculation is done for every pixel—higher cost but generally viewed as more realistic-looking

  (you would do this by passing the normal, vertex, and light values as `varying` variables—they are interpolated automatically and you can then do the lighting calculations in the fragment shader)
- A web image search on "Gouraud vs. Phong shading" will yield multiple visual examples