

Introduction to 3D Graphics

Using OpenGL 3D

Classical Polygon Graphics (H/W) Pipeline

- ▶ Mesh objects with 3D polygons (triangles or quads usually)
- ▶ Apply material properties to each object (for reflectance computation)
- ▶ Texture-map (i.e., superimpose an image on) polygons as needed
- ▶ Light scene
- ▶ Place camera
- ▶ Render (for each object/shape, for each polygon)
- ▶ Enjoy the view (map it to the display)

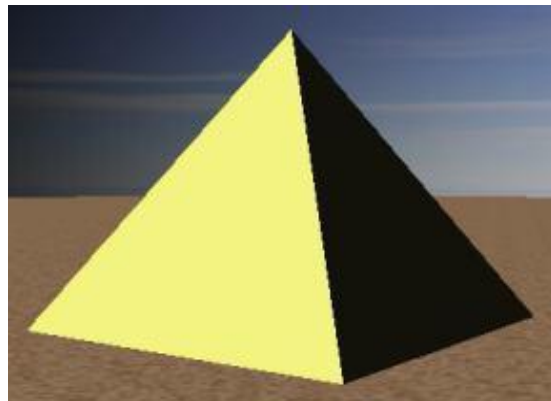
Why OpenGL for 3D?

- ▶ Widely used in industry and academia for interactive or real-time 3D graphics
- ▶ Old fixed-function API (OpenGL 1.x) assisted rapid prototyping of simple 3D scenes with “classical” lighting effects
 - ▶ Experiment with simple ideas quickly
- ▶ Modern programmable API allows for more flexibility and control
 - ▶ TAs will initially provide shaders for projects/labs; you will write your own in Labs 2 and 3

3D Polygons (1/2)

► Material specification

- Describes the light reflecting properties of the polygon
 - Color, shininess, reflectiveness, etc.
- Provided as input to shader
 - Provide values as **uniforms** to apply to entire shapes
 - Provide values as **attributes** to apply to individual vertices
- Specify yellow color of triangle as (1.0, 1.0, 0.3), an RGB triple
 - Alpha (translucency) can be specified as an additional parameter, or defaulted to 1.0

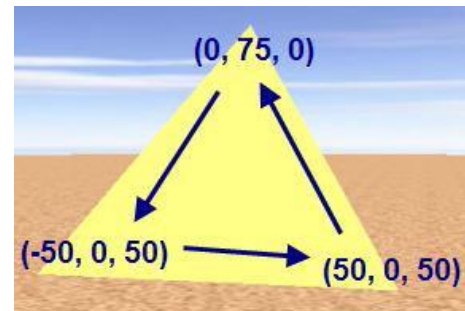
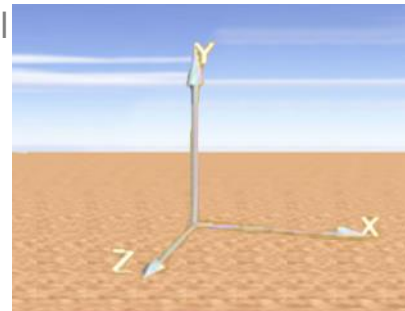


3D Polygons (2/2)

- ▶ OpenGL defaults to a right-handed coordinate system
- ▶ Polygons are defined in a single array of vertices:

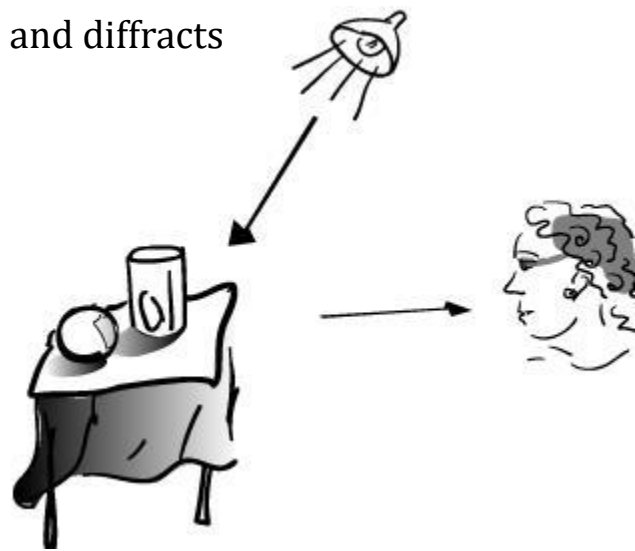
```
GLfloat vertexData[] = {  
    0, 75,  0, // Vertex 1  
   -50,  0, 50, // Vertex 2  
    50,  0, 50, // Vertex 3  
};
```

- ▶ This defines one triangle
 - ▶ A 3D shape would have multiple triangles in one array
- ▶ Coordinate values are arbitrary - can set virtual camera up capture any size scene, so use convenient values
- ▶ Remember counter-clockwise winding order!
- ▶ Surface normal uses right-hand rule: $E1 \times E2$ is normal to plane defined by edges $E1, E2$



to

Complexities of Light Reflection from Surfaces – Need to Know

- ▶ Intensity and direction of all light that strikes a point on object's surface, whether directly from light source or after multiple bounces from other objects (**global illumination, inter-object reflection**)
 - ▶ How an object's surface appears to us as it reflects, absorbs, and diffracts light (“material properties”)
 - ▶ Location of eye/camera relative to scene
 - ▶ Distribution of intensity per wavelength of incident light
 - ▶ Human visual system (HVS) and its differential, highly non-linear response to light stimuli
 - ▶ Lights may have geometry themselves
- 
- ▶ **Modern lighting/illumination models address these complexities (except for HVS)**

An Imperfect World – Model via Approximations

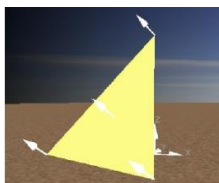
- ▶ Classic **lighting** models (also called illumination or reflection models, not to be confused with **shading** models discussed later) developed at the dawn of raster graphics in early 70s.
 - ▶ Epicenter at University of Utah in SLC where Ivan Sutherland worked with David Evans, a Mormon
 - ▶ Spawned the Evans & Sutherland flight simulator (with graphics) business
 - ▶ Other pioneers:
 - ▶ Henri Gouraud (shading model – filling in interior pixels from colors at vertices of a triangle)
 - ▶ Bui Tuong Phong (lighting and shading models)
 - ▶ Martin Newell (the Utah teapot (SIGGRAPH icon), meshing algorithms)
 - ▶ James Clark (geometry engine, Silicon Graphics, Netscape)
 - ▶ John Warnock (Hidden Surface Elimination, Adobe)
 - ▶ Ed Catmull (splines, Pixar, Disney)
 - ▶ Alvy Ray Smith (SuperPaint, HSV color space, partnered with Catmull on LucasFilm -> Pixar)
 - ▶ etc...

An Imperfect World

- ▶ Back then:
 - ▶ CPUs > 6 orders of magnitude less powerful, no GPU to speak of, just plot pixels
 - ▶ memory limited (measured in KB!)
- ▶ Even on today's machines, a physically accurate light simulation requires computational power beyond the capabilities of supercomputers!

Simple Lighting (Illumination) Models (1/2)

- ▶ Color of point on surface dependent on lighting of scene and surface material
- ▶ First approximation: model diffuse reflection from a matte surface (light reflected equally in all directions, viewer-independent) based **only** on angle of surface normal to light source



Facing light source:
Maximum reflection



In between: Some fraction of light reflected

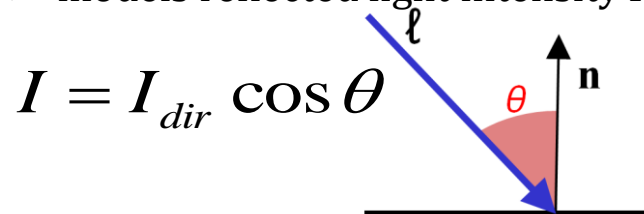


⊥ to light source:
No reflection

- ▶ Modeling light "drop-off" with angle to light

▶ Lambert's diffuse-reflection cosine law

- ▶ models reflected light intensity I



$$I = I_{dir} \cos \theta$$

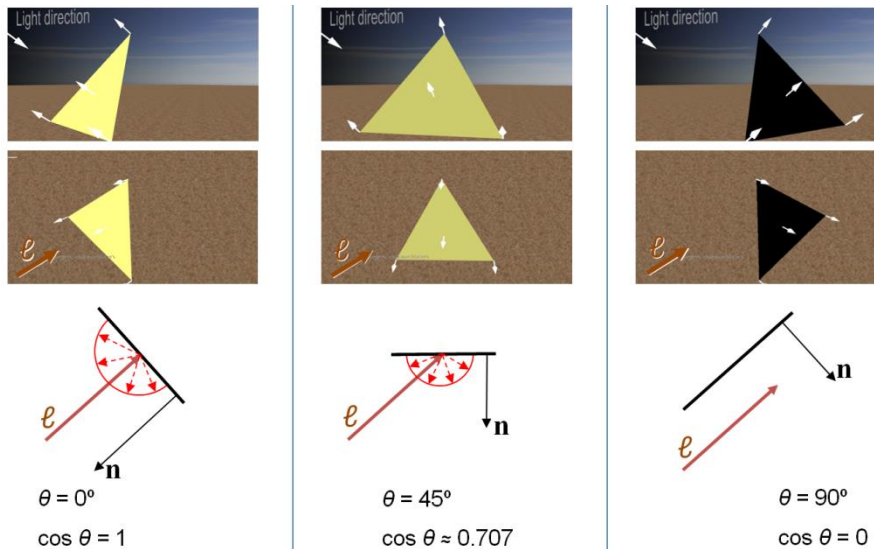
I_{dir} = measure of intensity of **directional** light (all rays parallel) at point of contact with surface, like rays from "infinitely far away" sun

θ = angle between surface normal (\mathbf{n}) and vector from light source (ℓ)

Note: I_{dir} and other quantities are fractions in $[0, 1]$. These units are convenient BUT completely arbitrary and **not physically-based!**

Simple Lighting (Illumination) Models (2/2)

- ▶ Lambert light attenuation based on surface's angle to light source
- ▶ Visualization of Lambert's law in 2D

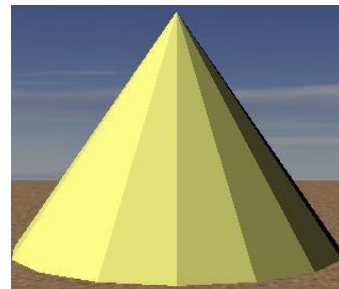
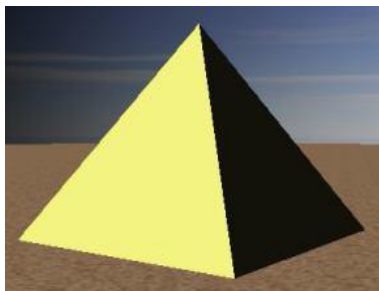


$$I = I_{dir} \cos \theta$$

- Note: crudely approximate intrinsic material properties of object with RGB values. For example, the greater the R, the more reddish the object will appear under white light.
 - In reality, need surface (micro)geometry and wavelength-dependent reflectivity, not just RGB

Shading Rule (1/6)

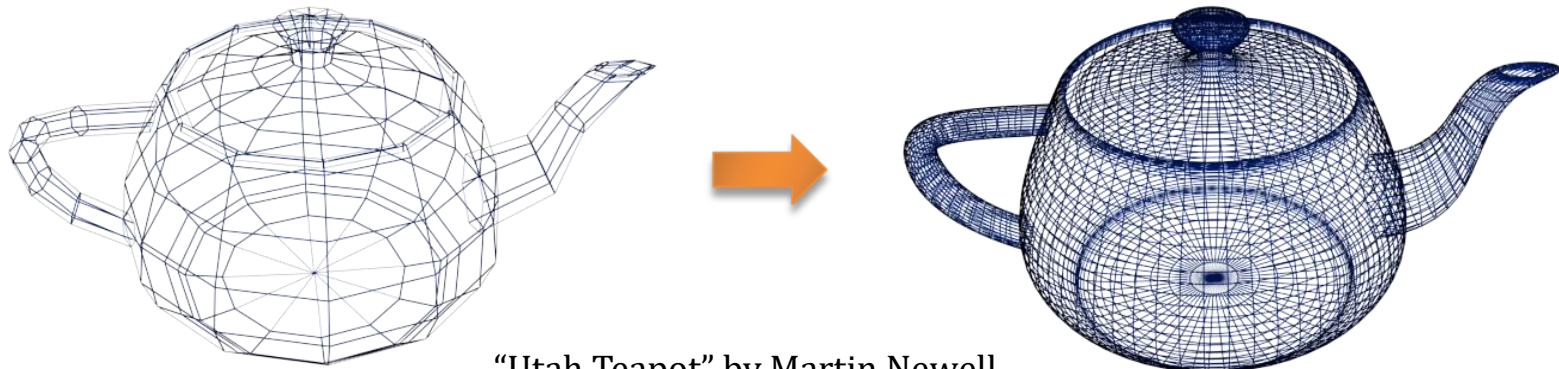
- ▶ Goal: finding color at each pixel, preferably w/o having to evaluate a full lighting model at each pixel
- ▶ First approach: Lambert's cosine law (flat/constant shading for whole facet)
 - ▶ faceted appearance, perfect for this rectangular pyramid.
- ▶ What if we want to approximate a rounded object?
 - ▶ Lambert-shaded, faceted; appearance is no longer ideal



<http://math.hws.edu/graphicsbook/demos/c4/smooth-vs-flat.html>

Shading Rule (2/6)

- ▶ First solution: increase the number of polygons
- ▶ Better shape approximation, more expensive to render
- ▶ Ultimately, still faceted when rendered (higher poly count => less faceted)
- ▶ Adaptive meshing is an improvement - more polygons in areas of high curvature



“Utah Teapot” by Martin Newell

Shading Rule (3/6)

- ▶ Get this:



faceted shading

- ▶ Want this:

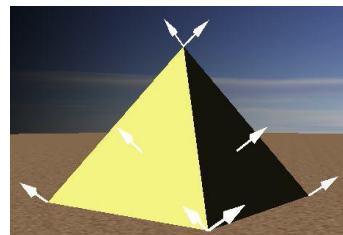


smooth shading

Shading Rule (4/6)

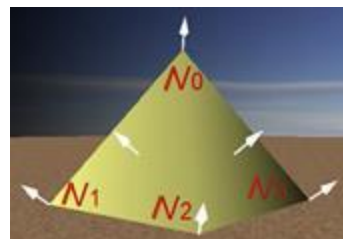
► Gouraud smooth shading

- compute lighting equation at each vertex of mesh (requires angle between normal, vector to light) for Lambertian diffuse reflection
- linearly interpolate vertex color values to get colors at all points: $C = C1 + t(C2 - C1)$
 - **weighted averaging**: the closer point is to a vertex, the more it is influenced by that vertex
- **How do we determine vertex colors? Need a normal...**
 - Vertex normals are an artifice; the normal is mathematically undefined since a vertex is a discontinuity
 - Sol'n 1: use plane normal, get faceted shading
 - Sol'n 2: hack: average face/plane normals



Faceted

The normal at a vertex is the same as the plane normal. Therefore, each vertex has as many normals as the number of planes it helps define.



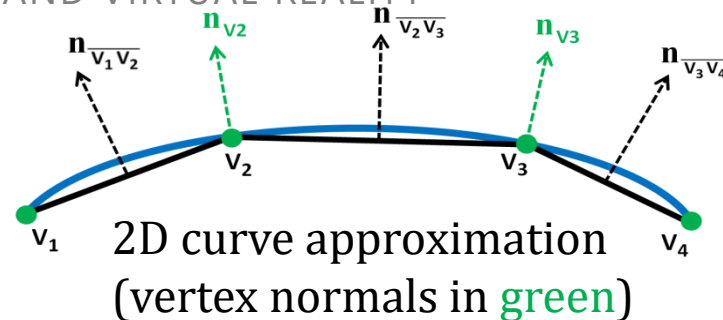
Smooth

Only one vertex normal per vertex; average of face normals of the faces the vertex is part of

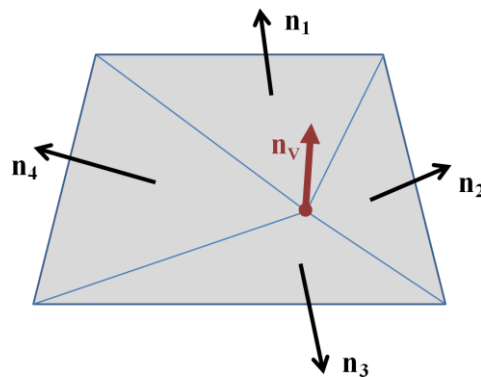
Shading Rule (5/6)

▶ Vertex normals

- ▶ if vertex used by only one face, normal is set to face's normal
 - ▶ typically computed from the face's plane equation
- ▶ otherwise, normal is set to average of normals of all faces sharing it
- ▶ if mesh is not too coarse, vertex normal is a decent approximation to the normal of modeled surface closest to that vertex
- ▶ adaptive meshing adds more triangles in areas with rapid changes in curvature



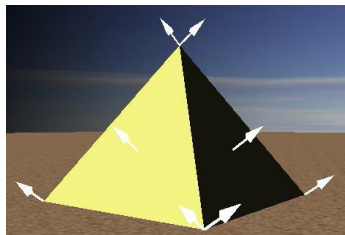
Vertex normals shown in color, face normals in black



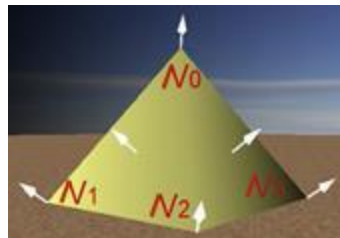
3D mesh approximation (looking down on an irregular pyramid, face normals roughly cancel each other out, hence normal points out)

Shading Rule (6/6)

- ▶ Programmable OpenGL API doesn't provide any lighting or shading. Use shaders to implement lighting model and shading rule of your choice
 - ▶ to get flat shading, specify the same surface normal for vertices of the same facet (each vertex gets n normals, where n is the number of facets it is a part of)
 - ▶ to get smooth shading, you must specify a single shared normal for each (shared) vertex in the object

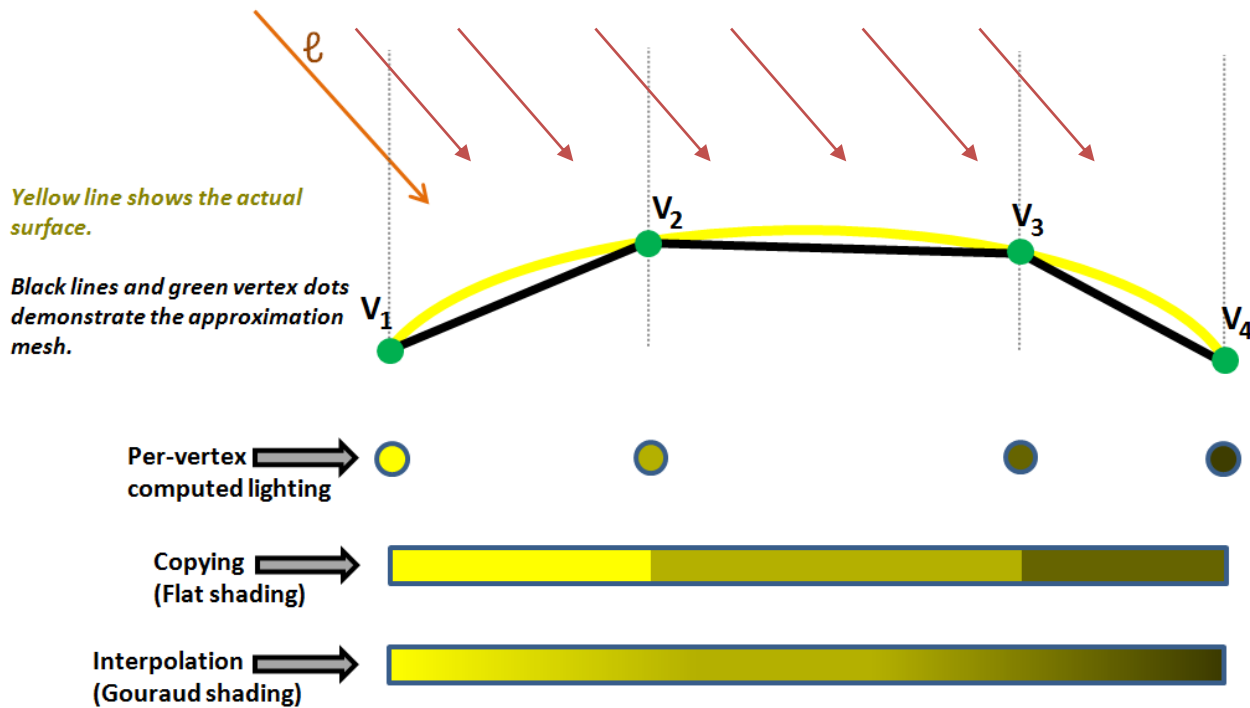


Faceted



Smooth

Interpolation vs Flat Shading Summary



Vertex Normals

- ▶ Sending vertex normal to the shader requires a small extension to the way we specify vertices
- ▶ Each vertex is now a position plus a normal, e.g.,

```
GLfloat[] vertexData = {  
    -1, 0, 0, // Position 1  
    0, 0, -1, // Normal 1  
    1, 0, 0, // Position 2  
    1, 0, 0, // Normal 2  
    ... };
```

- ▶ Normals needn't be axis-aligned, of course...
- ▶ For flat shading a shared vertex has as many (position, normal) entries as the facets it's a part of

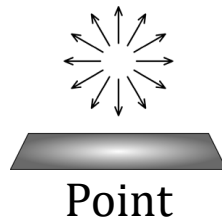
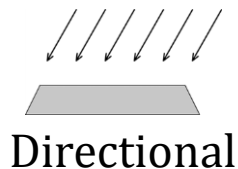
Phong Reflectance (Illumination, Lighting) Model (1/7)

▶ Non-geometric lights:

- ▶ Ambient: crudest approximation (i.e., total hack) to inter-object (“global”) reflection - all surfaces receive same light intensity. Allows all facets to be minimally visible
- ▶ Directional: illuminates all objects equally from a given direction; light rays are parallel (models sun, sufficiently far away)

▶ Geometric lights:

- ▶ Point: Originates from single point, spreads outward equally in all directions
- ▶ Spotlight: Originates from single point, spreads outward inside cone’s directions



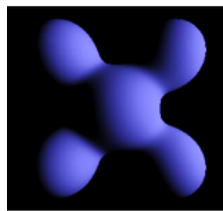
Phong Reflectance Model (2/7)

- ▶ Many models exist to approximate lighting physics – more accurate => more computation
- ▶ Fixed-function OpenGL: Phong reflection model, survives today (though crude)
 - ▶ implemented in fixed function hardware for decades, easily implemented in shaders
 - ▶ approximates lighting by breaking down into three components: ambient, diffuse, specular
 - ▶ can think of these as coincident, independent layers, each with its own characteristics, and sum them to get the final result
 - ▶ is a non-global illumination model – no inter-object reflections, **non-physically based**

**AMBIENT**

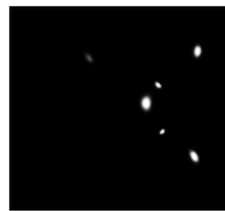
Effect of light that is non-directional, affecting all surfaces equally.

+

**DIFFUSE**

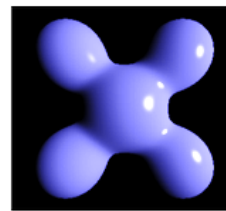
Effect of directional light on a surface with a dull/rough finish.

+

**SPECULAR**

Effect of directional light on a shiny surface when the vector to the eye-point is closely aligned to the light's reflected rays.

=

**THE COMPOSITE**

The three independent reflectivity types are accumulated to produce the result.

Phong Reflectance Model (3/7)

▶ $I_{total,\lambda} =$

Ambient Component $(I_{ambient,\lambda} k_{ambient,\lambda} O_{diffuse,\lambda}) +$

Diffuse Component $\Sigma_{directional\ lights} (I_{diffuse,\lambda} k_{diffuse,\lambda} O_{diffuse,\lambda} (\cos \theta))$
 $+ \Sigma_{geometric\ lights} (f_{att} I_{diffuse,\lambda} k_{diffuse,\lambda} O_{diffuse,\lambda} (\cos \theta)) +$

Specular Component $\Sigma_{directional\ lights} (I_{specular,\lambda} k_{specular,\lambda} O_{specular,\lambda} (\cos \delta)^n)$
 $+ \Sigma_{geometric\ lights} (f_{att} I_{specular,\lambda} k_{specular,\lambda} O_{specular,\lambda} (\cos \delta)^n)$

- ▶ Equation is wavelength-dependent; approximate with separate equations for $\lambda \in (R, G, B)$
- ▶ All values unitless real numbers between 0 and 1
- ▶ Evaluates total reflected light $I_{total,\lambda}$ at a single point, based on all lights

Phong Reflectance Model (4/7)

► Variables

- λ = wavelength / color component (e.g. R, G, and B)
- $I_{total,\lambda}$ = total amount of light reflected at the point
- $I_{ambient}$ = intensity of incident ambient light; similar for diffuse, specular incident light
- f_{att} = attenuation function for a geometric light
- O = innate color of object's material at specific point on surface (RGB approximation)
- k = object's efficiency at reflecting light
- Since both O and k are dimensionless fractions we really only need one of them

► Ambient component

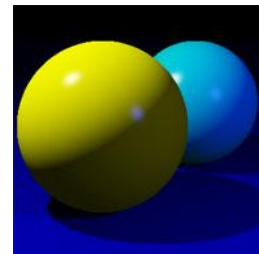
- $(I_{ambient,\lambda} k_{ambient,\lambda} O_{diffuse,\lambda})$ -- think of $k_{ambient,\lambda}$ as the fraction of $I_{ambient}$ reflected for that λ . Note that here we use $O_{diffuse,\lambda}$ for the ambient component; in Sceneview we use distinct $O_{ambient,\lambda}$
- effect on surface is constant regardless of orientation, no geometric information
- total hack (crudest possible approximation to global lighting based on inter-object reflection), but makes all objects a little visible - scene looks too stark without it

Phong Reflectance Model (5/7)

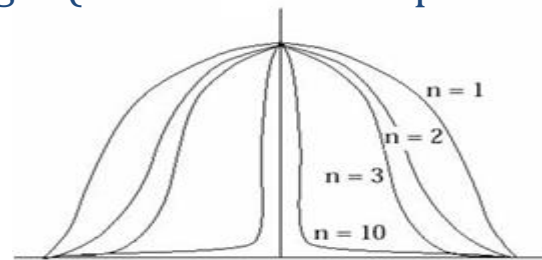
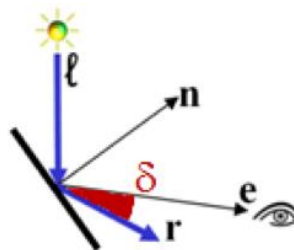
- ▶ **Diffuse component (R component shown below, same for G, B) - viewer independent!**
 - ▶ uses Lambert's diffuse-reflection cosine law
 - ▶ $\Sigma(I_{diffuse,R}k_{diffuse,R}O_{diffuse,R}(\cos \theta))$
 - ▶ $I_{diffuse}$ = light's diffuse color
 - ▶ $k_{diffuse}$ = the efficiency of incident light reflection
 - ▶ $O_{diffuse}$ = innate color of object's diffuse material property at specific point on surface
 - ▶ $\cos \theta$ = Lambert's attenuation factor where θ is the angle between normal and light vector

Phong Reflectance Model (6/7) $\Sigma_{lights} (I_{specular,\lambda} k_{specular,\lambda} O_{specular,\lambda} (\cos \delta)^n)$

- ▶ Specular Component (for R) – **viewer-dependent**
 - ▶ highlights seen on shiny objects (plastic, metal, mirrors, etc.)
 - ▶ cosine-based attenuation factor ensures highlight only visible if reflected light vector and vector to viewer are closely aligned
 - ▶ n = specular power, how "sharp" highlight is – the sharper, the more intense
 - ▶ specular highlight of most metals are the color of the metal but those on plastic, shiny apple, pearl, etc. are mostly the color of the light (see Materials chapter 27)



\mathbf{e} = viewpoint
 \mathbf{r} = reflected image of light source
 ℓ = vector from the light source
 \mathbf{n} = surface normal
 δ = angle between \mathbf{e} and \mathbf{r}
 n = specular coefficient



Specular falloff of $(\cos \delta)^n$

Note: Fixed-function OpenGL uses a slightly different lighting model called Blinn-Phong. See 14.9.3

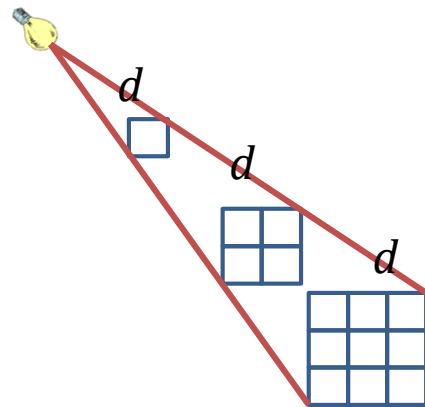
Phong Reflectance Model (7/7)

- ▶ Attenuation factor f_{att}

- ▶ Used in diffuse and specular light calculation:

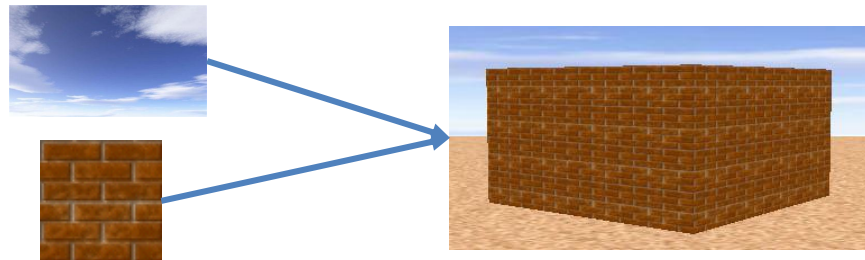
$$... + \sum_{geometric\ lights} \left(f_{att} I_{diffuse, \lambda} k_{diffuse, \lambda} O_{diffuse, \lambda} (\cos \theta) \right) + ...$$

- ▶ Directional lights have no attenuation (infinitely far away)
 - ▶ Geometric lights (point lights, spot lights) get dimmer with distance
 - ▶ Inverse square law
 - ▶ area covered increases by square of distance from light
 - ▶ thus, light intensity is inversely proportional to square of distance from light
 - ▶ light twice as far away is one quarter as intense
 - ▶ though physics says inverse square law, doesn't always look good in practice so OpenGL lets you choose attenuation function (quadratic, linear, or constant)



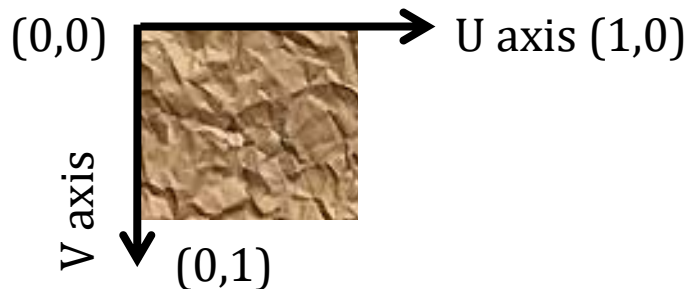
Texture Mapping (1/2)

- ▶ **Goal:** adding more detail to geometry of scene without adding more actual polygons
- ▶ **Solution:** texture mapping
 - ▶ used extensively in video games, e.g., for backgrounds, billboards
 - ▶ also used for many other techniques such as level-of-detail management
 - ▶ cover the mesh's surface in stretchable "contact paper" with pattern or image on it
 - ▶ in general, difficult to specify mapping from contact paper to every point on an arbitrary 3D surface
 - ▶ mapping to planar polygons is easy: specify mapping for each vertex and interpolate to find mapping of interior points



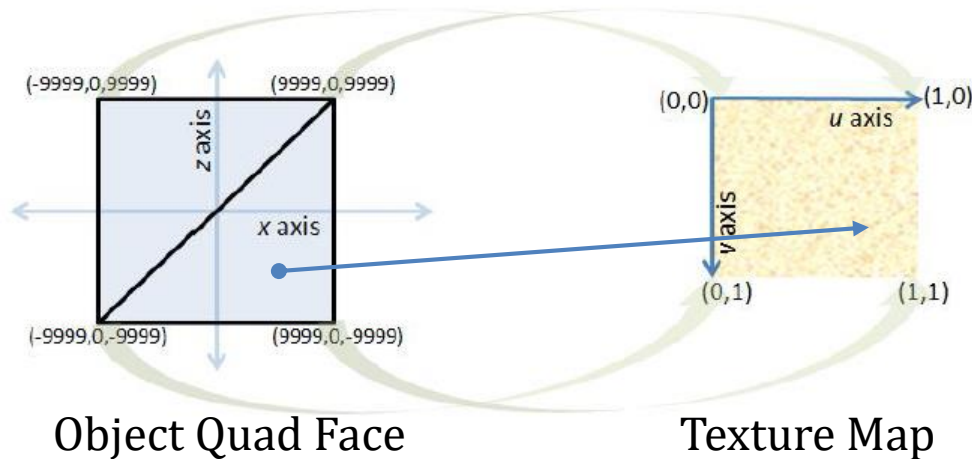
Texture Mapping (2/2)

- ▶ Specifying "texture point" mapped to particular vertex
 - ▶ requires coordinate system for referring to positions within texture image
 - ▶ convention:
 - ▶ points on pixmap described in abstract floating-point "texture-coordinate system"
 - ▶ axes labeled u and v , range 0 to 1.
 - ▶ origin located at the upper-left corner of the pixmap



Texture Mapping UV Coordinates

- ▶ Let's map from two coplanar triangles from a face in the 3D model to a texture map
- ▶ Texture map uses UV texture coordinates: just use ratios



- ▶ Texture mapping arbitrary solids is much harder – we'll study this later

Texture Mapping Example (1/2)

- ▶ We add texture coordinates* in the same way we added normals

```
GLfloat[] vertexData = {  
    -10, 0, 0,  // Position 1  
    0, 1, 0,  // Normal 1  
    0, 0,     // Texture Coordinate 1  
    10, 0, 0,  // Position 2  
    0, 1, 0,  // Normal 2  
    1, 0,     // Texture Coordinate 2  
    ... };
```

* We'll teach how to set up texture maps in Lab 3

Texture Mapping (Tiling)

- ▶ Create a brick wall by applying brick texture to plane



- ▶ Produces realistic-looking image, but very few bricks in wall



- ▶ Tiling increases number of apparent bricks



Texture Mapping (Stretching)

- ▶ Create a sky backdrop by applying a sky image to a plane



- ▶ Would look unnatural if tiled
- ▶ Stretch to cover whole plane

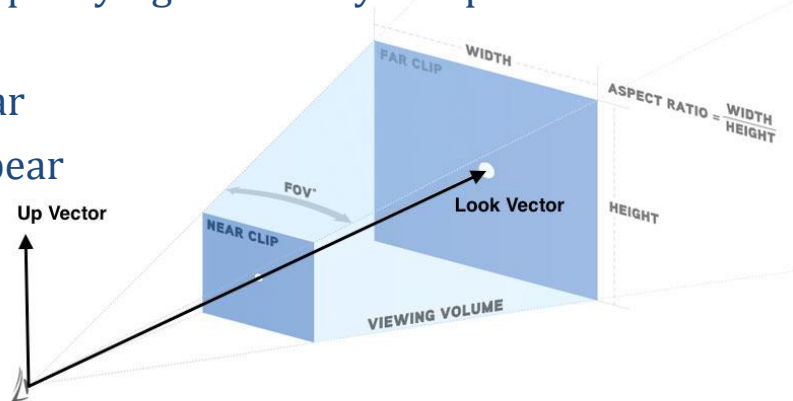


- ▶ Your texture shader can implement tiling and stretching by multiplying UV coordinates by a value >1 for tiling and <1 for stretching

Camera (1/3)

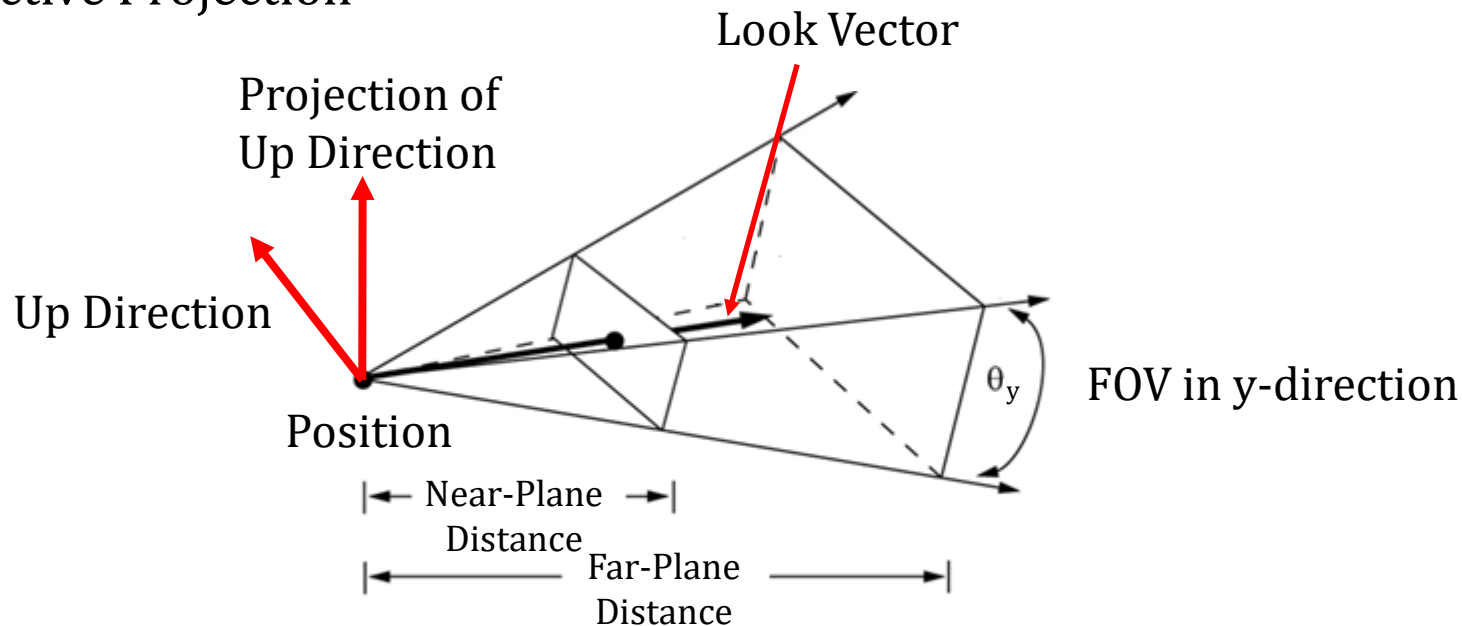
▶ Camera Properties:

- ▶ **Perspective or Orthographic**
- ▶ **Position:** placement of camera
- ▶ **Look Direction:** direction camera is aimed (vector determining lens axis)
- ▶ **Up Direction:** rotates camera about look vector, specifying which way is “up” – must not be collinear to the look vector
- ▶ **Far-Plane Distance:** objects behind do not appear
- ▶ **Near-Plane Distance:** objects in front do not appear
- ▶ **Field Of View:** (Width, height or diagonal angle)
- ▶ **Aspect Ratio** (Relative width and height)



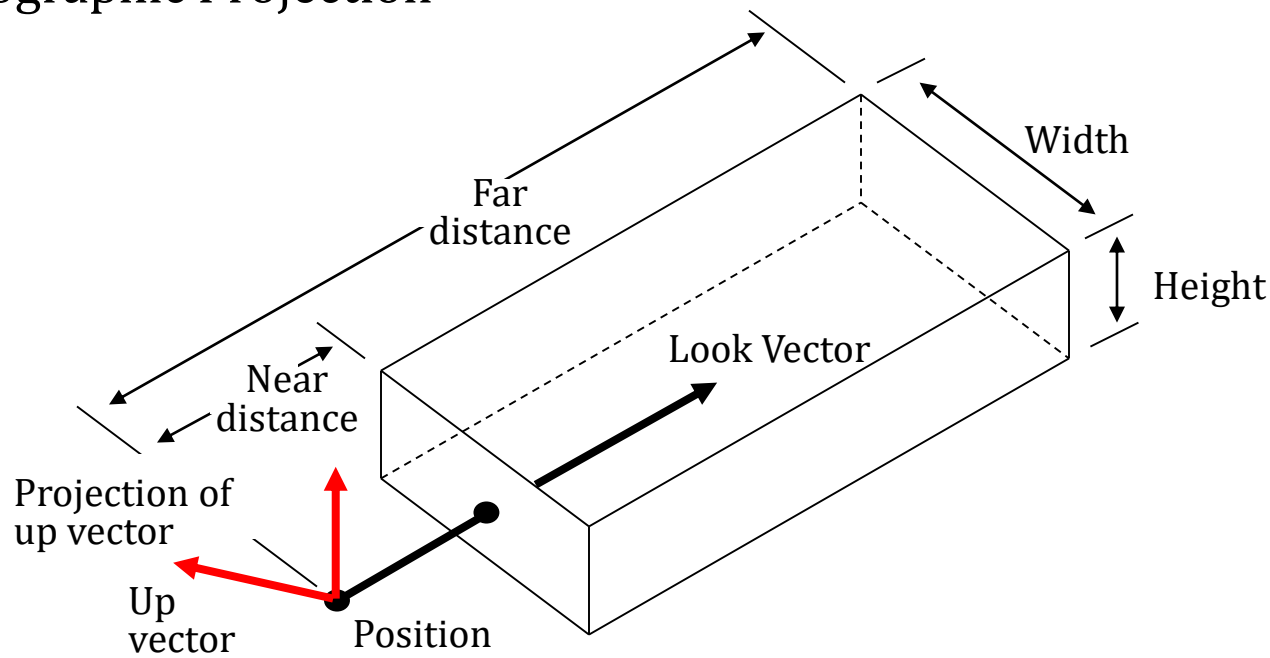
Camera (2/3)

► Perspective Projection



Camera (3/3)

► Orthographic Projection



OpenGL Camera

- ▶ Fixed-function API has support for perspective and orthographic cameras
- ▶ With the Programmable API you must construct and supply all model, view, and projection matrices, and then use them in your shaders

Rendering with OpenGL

- ▶ Pipeline of rendering with OpenGL
 - ▶ Calculate vertex data (position, normals, texture coords)
 - ▶ Calculate scene data (light position/type, camera position/orientation etc.)
 - ▶ Pass scene data to shader (specifying uniforms, in OGL parlance)
 - ▶ Pass vertex data to shader (specifying attributes, in OGL parlance)
 - ▶ Tell OpenGL to draw
- ▶ To be extra clear:
 - ▶ You write most code in C++
 - ▶ The C++ code involves using the OpenGL API to set up data structures for scene geometry, lights, and camera, which are then passed to the shaders for execution
 - ▶ You write the shaders in GLSL to process this data for rendering
- ▶ Easy enough, but just how do you pass data to shader?

Passing Data to Shader (1/5)

- ▶ What kinds of data do we have in the scene?
- ▶ Vertex data (position, normal, tex coords)
 - ▶ Pass as **attributes** in a single large array
 - ▶ Requires two OpenGL objects
 - ▶ VBOs (Vertex Buffer Objects)
 - ▶ VAOs (Vertex Array Objects)
- ▶ Also have data that remains constant across vertices (e.g., camera matrices)
 - ▶ Pass as **uniforms** using a named variable

Passing Data to Shader (2/5) -- Uniforms

- ▶ Used for data that remains constant for all vertices
 - ▶ e.g. color,* camera position, light position
- ▶ Three steps
 - ▶ 1. In GLSL shader => declare uniform variable
 - ▶ Ex: `uniform vec3 color;`
 - ▶ 2. In C++ OpenGL => Find memory address of uniform variable
 - ▶ Ex: `GLuint color_loc = glGetUniformLocation(m_shaderID, "color");`
 - ▶ 3. In C++ OpenGL => Store data in memory address
 - ▶ Ex: `glUniform3f(color_loc, 0.5, 0.9, 0.8);`
 - Note: 3f stands for 3 floats (RGB). To store 2 floats, use `glUniform2f`. To store 4 ints, use `glUniform4i`
 - See [here](#) for list of entire `glUniform` family

Passing Data to Shader (3/5) – Example Uniforms

```
// passing information for color
// ambient term is specified as RGB(A). Use glUniform4f to provide optional alpha value
// this specifies a dark grey ambient "light"
glUniform4f(<Ambient Location>, 0.2, 0.2, 0.2, 1.0 ); // 4f = 4 floats

// passing information for lighting
glUniform3f(<Position Location>, 10.0, 5.0, 8.0 ); // 3f = 3 floats
glUniform3f(<Direction Location>, 1.0, 2.0, 3.0 );

// specify an integer constant to describe type of light, here a point light
glUniform1i(<Type Location>, POINT_LIGHT_TYPE); // 1i = 1 int

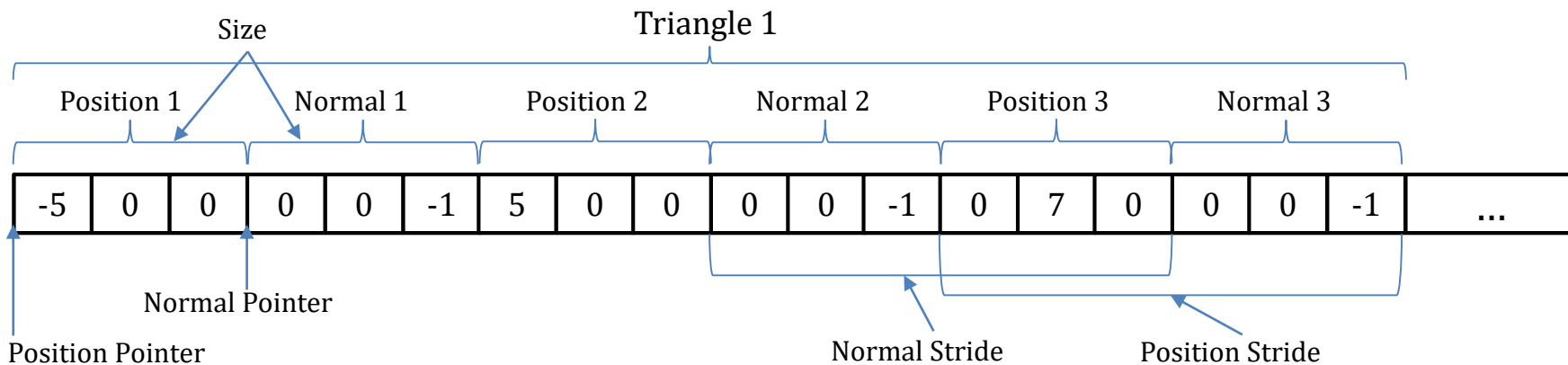
// To use a directional light
glUniform1i(<Type Location>, DIRECTIONAL_LIGHT_TYPE);
```

Passing Data to Shader (4/5) – Vertex Data

- ▶ Passing vertex data is more complicated than uniform data
- ▶ Have vertex data (pos, normal, tex) in single large array
 - ▶ Note: In OGL parlance, pos, normal, tex etc. are **attributes** each vertex has
- ▶ Two steps
 - ▶ 1. Store data in Vertex Buffer Object (VBO)
 - ▶ 2. Specify attribute layout in VBO with Vertex Array Object (VAO)

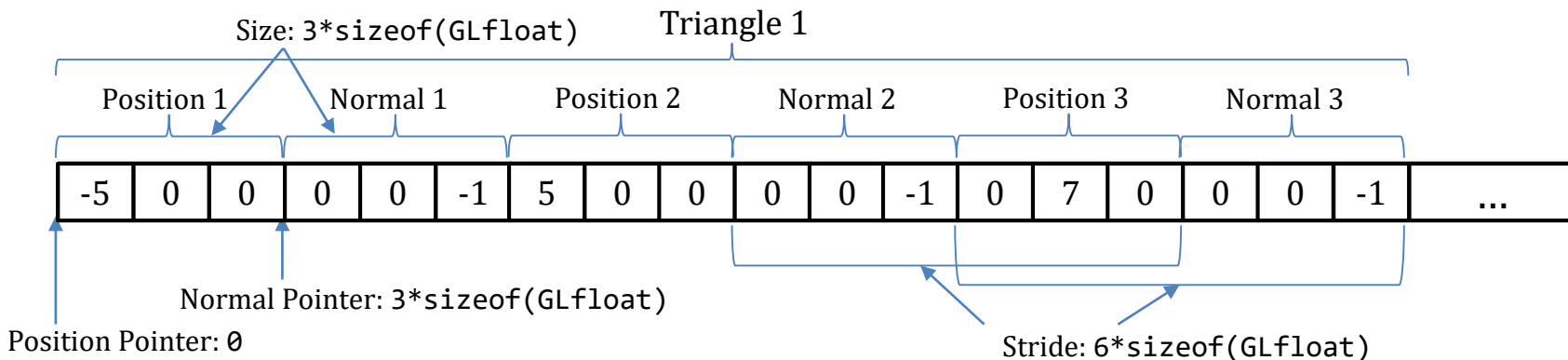
VBOs and VAOs

- ▶ VBO (Vertex Buffer Object) stores vertex data, such as position, normal, and texture coordinates. Created in C++ program, passed to shader
 - ▶ (all numbers below are really GL_FLOATs)
- ▶ Meaningless w/o interpretation - VAO tells shader how attributes are stored



Vertex Array Objects

- ▶ For each attribute, VAO takes three parameters, details in lab 1
 - ▶ **size** parameter is how many values an attribute has (e.g. 3 for position)
 - ▶ **stride** specifies how far apart values of the same type are in our array
 - ▶ **pointer** is a pointer to the index of the first value of that attribute
 - ▶ Because VBO is byte array, multiply parameters by `sizeof(GLfloat)`



The CS337 Guide to OpenGL

- ▶ TA has written a guide to OpenGL:
<http://www.cs.sjtu.edu.cn/~shengbin/course/cg/course.html>

Question 1:

- ▶ Write a program to draw a simple red cube.

Question 2:

- ▶ Write a program to draw a simple blue triangle.

- ▶ **Reference:**

- ▶ <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/>
- ▶ <https://graphics.stanford.edu/courses/cs248-99/OpenGLSession/tri.html>
- ▶ <http://antongerdelan.net/opengl/hellotriangle.html>

Assignment 1- Introduction to OpenGL(Deadline Oct. 10 8:00AM)

Question 3:

- ▶ Write a C++ class to draw and move a car using the geometrical classes. The car should be kind of similar to the one below. You can implement more complex car shapes if you want.

