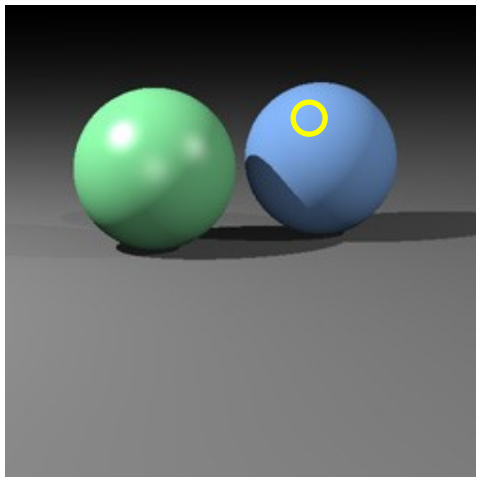
Lights and Shading

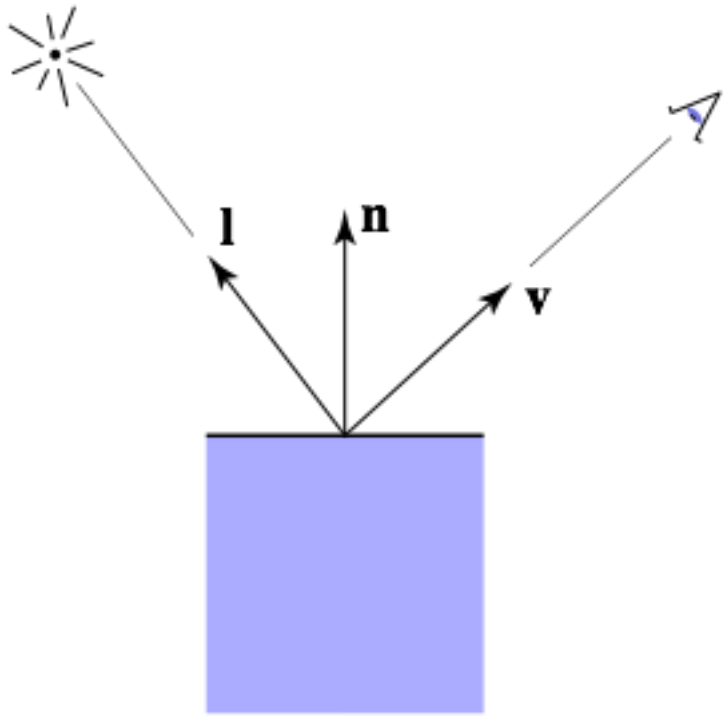
COMP557
Paul Kry



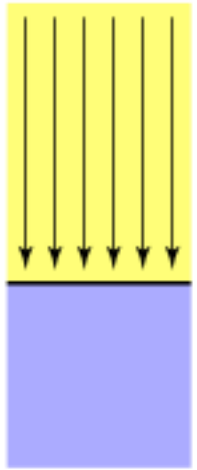
Light source
Viewer (eye)
Illumination
Projection
direction
Objects in scene

Shading

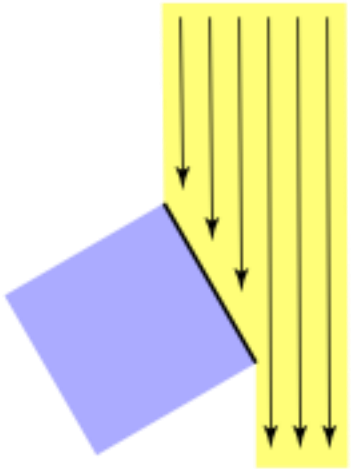
- Compute light reflected (*scattered*) toward camera
- Inputs:
  - eye direction
  - light direction
    (for each of many lights)
  - surface normal
  - surface parameters
    (color, shininess, …)



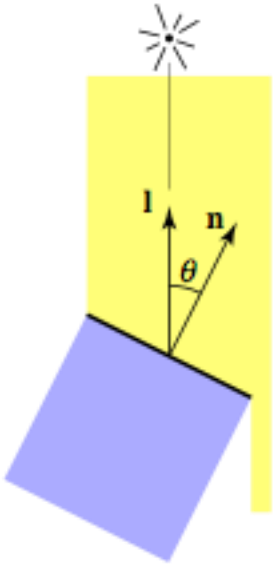
Diffuse reflection



Top face of cube receives a certain amount of light



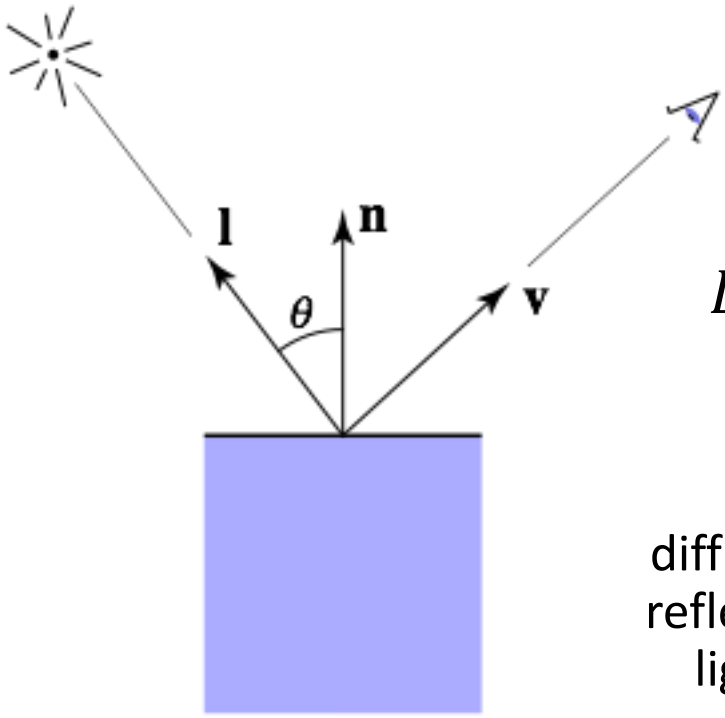
Top face of 60 deg rotated cube intercepts half the light



In general, light per unit area is proportional to $\cos\theta = l \cdot n$

Lambertian shading

- A Lambertian surface scatters light equally in all directions
- Process each colour channel independently (more on colour later)
- Let k_d and I be vectors with red greed and blue components, then see $k_d I$ as ***component wise multiplication***



$$L_d = k_d \, I \, \max(0, \boldsymbol{n} \cdot \boldsymbol{l})$$

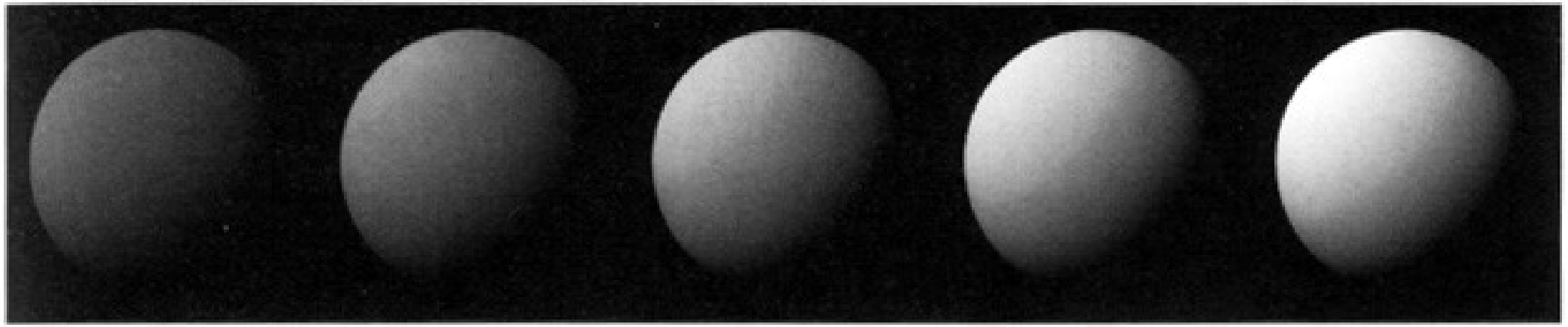
illumination from source

diffusely reflected light

diffuse coefficient

Lambertian shading

- Produces matte appearance

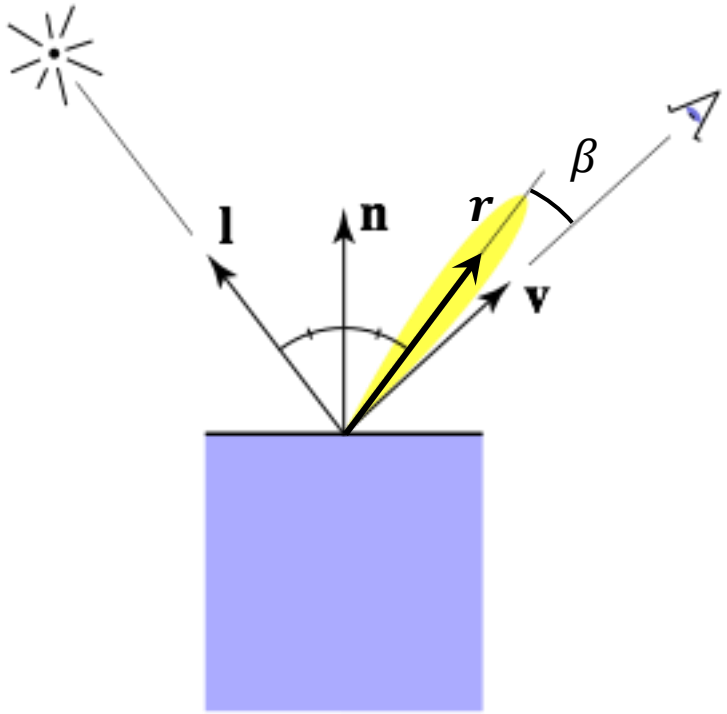


[Foley et al.]

k_d ⟶

Specular shading (Phong)

- Intensity depends on view direction
  - bright near mirror configuration



$$\boldsymbol{r} = \text{reflect}(\boldsymbol{l}, \boldsymbol{n}) = \left(2\boldsymbol{n}\boldsymbol{n}^T - \boldsymbol{I}\right)\boldsymbol{l}$$

$$L_s = k_s\, I \max(0, \cos\beta)^p = k_s\, I \max(0, \boldsymbol{r} \cdot \boldsymbol{v})^p$$

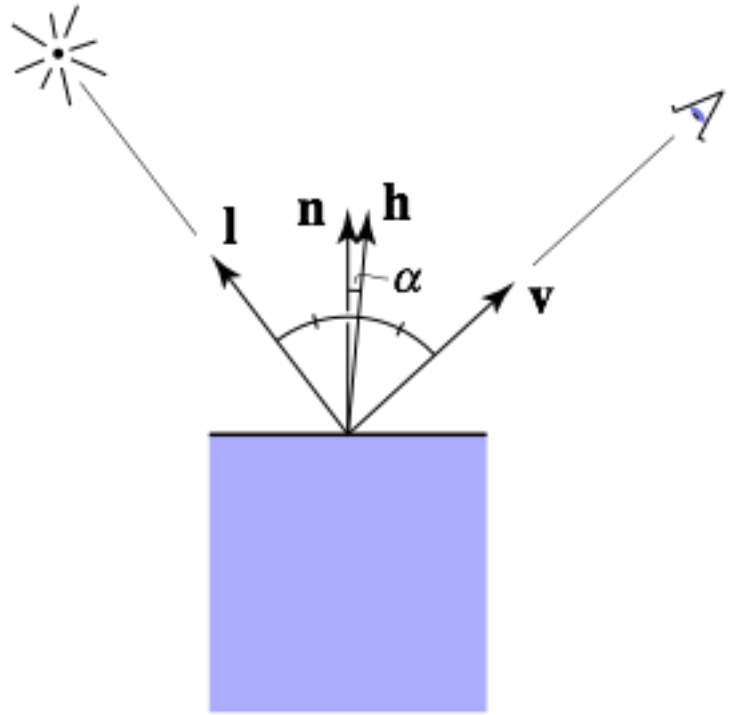
specularly reflected light

specular coefficient

shininess

Specular shading (Blinn-Phong)

- Close to mirror ⇔ half vector near normal
  - Measure “near” by dot product of unit vectors



$$\boldsymbol{h} = \text{bisector}(\boldsymbol{v}, \boldsymbol{l}) = \frac{\boldsymbol{v} + \boldsymbol{l}}{\|\boldsymbol{v} + \boldsymbol{l}\|}$$

$$L_s = k_s\, I\, \max(0, \cos\alpha)^p = k_s\, I\, \max(0, \boldsymbol{n} \cdot \boldsymbol{h})^p$$

specularly reflected light

specular coefficient

shininess

Phong model—plots

- Increasing specular exponent *p* narrows the lobe

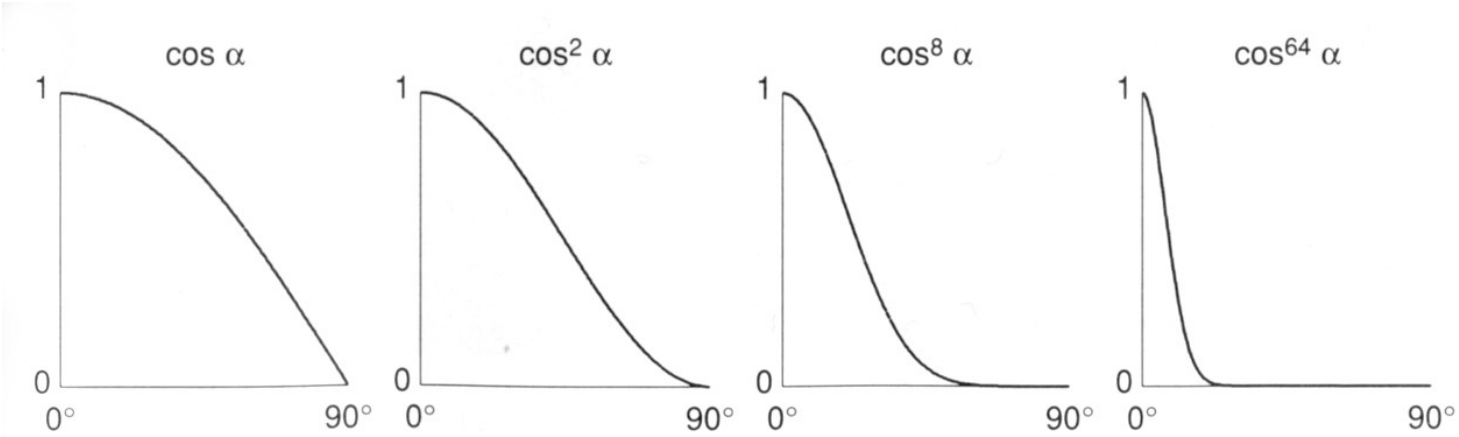
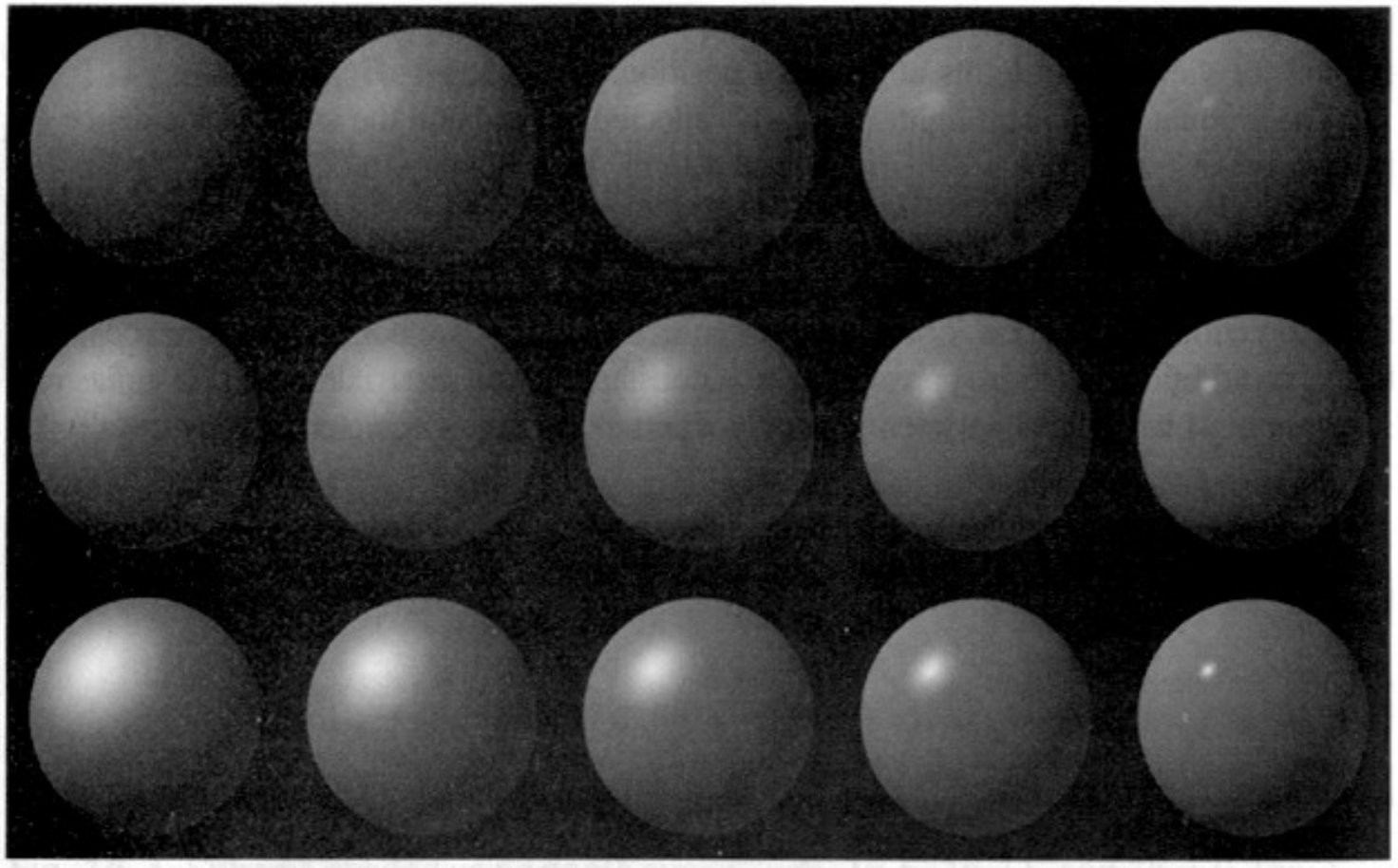


Fig. 16.9 Different values of $\cos^n \alpha$ used in the Phong illumination model.

[Foley et al.]

- What are good values for the exponent?
- Alternatively what are bad values? What can go wrong?
  - Note, the model is phenomenological… not based on physics

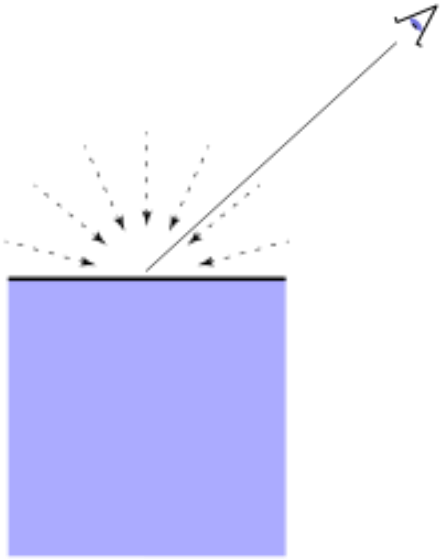
Specular shading



[Foley et al.]

Ambient shading

- Shading that does not depend on anything
  - add constant color to account for ***disregarded illumination*** and fill in black shadows
  - What is this “disregarded illumination” that we didn’t include?



$$L_a = k_a \, I_a$$

reflected ambient light

ambient coefficient

ambient light intensity

Putting it together

- Usually include ambient, diffuse, Phong in one model

$$L = L_a + L_d + L_s$$
$$= k_a I_a + k_d\, I \max(0, \boldsymbol{n} \cdot \boldsymbol{l}) + k_s\, I \max(0, \boldsymbol{n} \cdot \boldsymbol{h})^p$$

- The final result is the sum over many lights

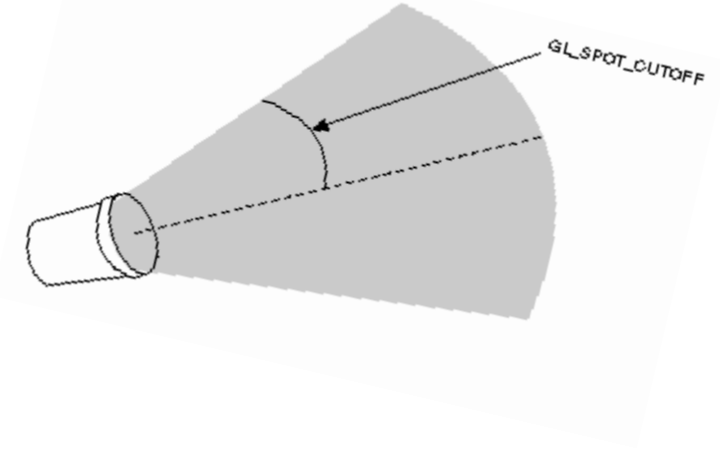
$$L = L_a + \sum_{i=1}^{N} \left((L_d)_i + (L_s)_i \right)$$

$$= k_a I_a + \sum_{i=1}^{N} (k_d I_i \max(0, \boldsymbol{n} \cdot \boldsymbol{l}_i) + k_s I_i \max(0, \boldsymbol{n} \cdot \boldsymbol{h}_i)^p)$$

Spot lights

- Can also have GLSL code for other effects
  - For instance, spotlights…

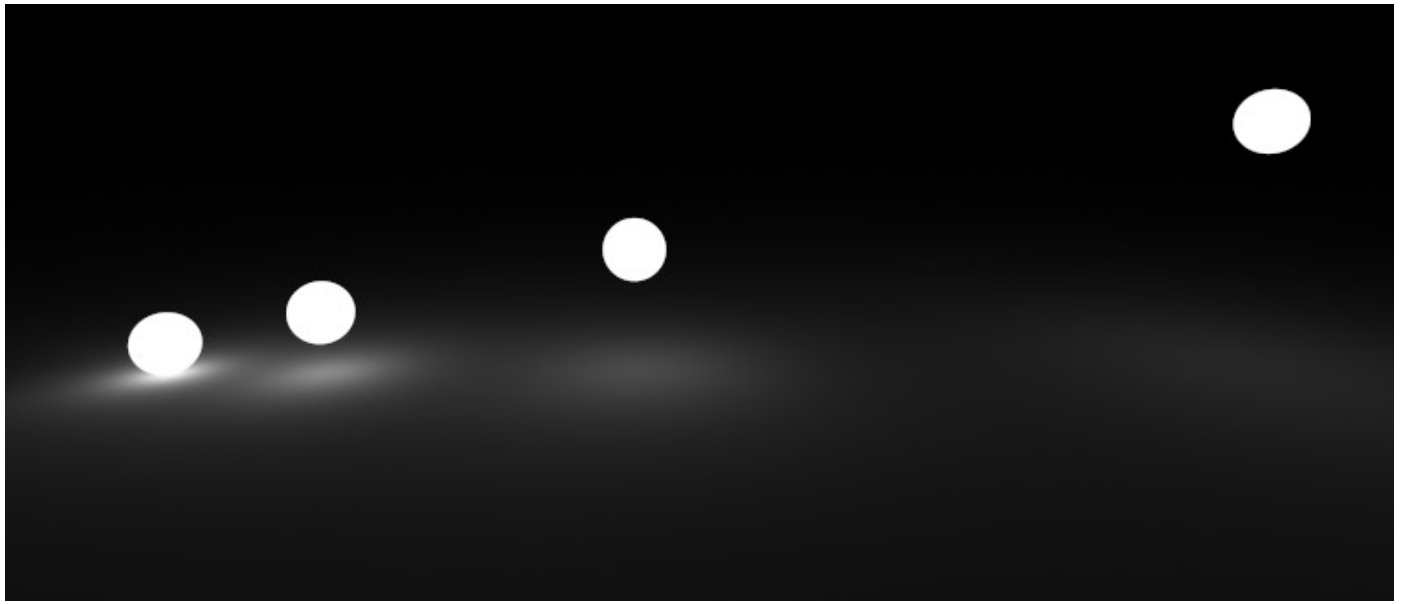
    Inspired by old OpenGL versions, can set up spotlights with parameters such as position, direction, cutoff, and an exponent to control how concentrated. Do the math or look at OGLPG8 too!



Light Attenuation

- Intensity of light decreases as distance from the light increases.
- Directional light is “infinitely far away”, so attenuation disabled.
- Constant and Linear term helps us capture the attenuation behaviour of area lights.

$$\frac{1}{k_c + k_l d + k_q d^2}$$



http://imdoingitwrong.wordpress.com/2011/01/31/light-attenuation/

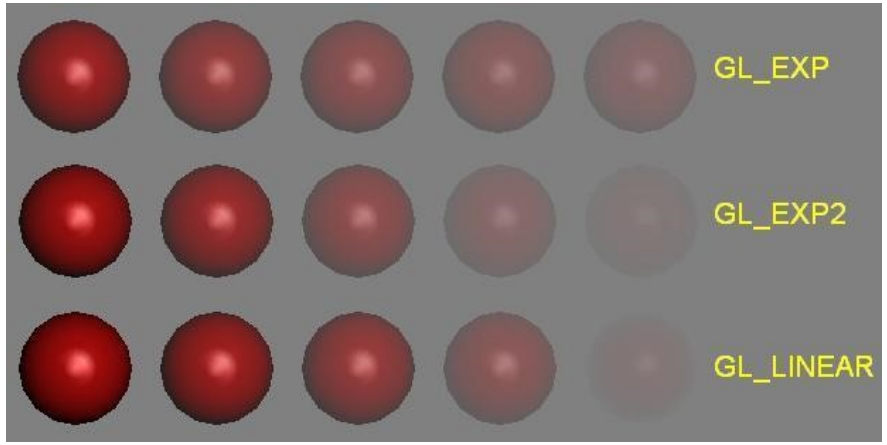
Light absorption and scattering

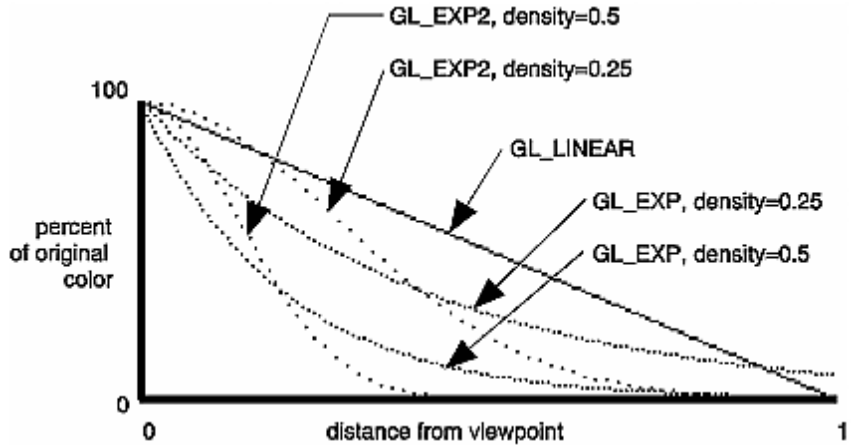
- Can mix a fog colour with the surface colour
  - Colour, exponential with density, or linear with start and end, using different equations from old-school OpenGL

$$f = \mathrm{e}^{-(density\ z)}$$

$$f = \mathrm{e}^{-(density\ z)^2}$$

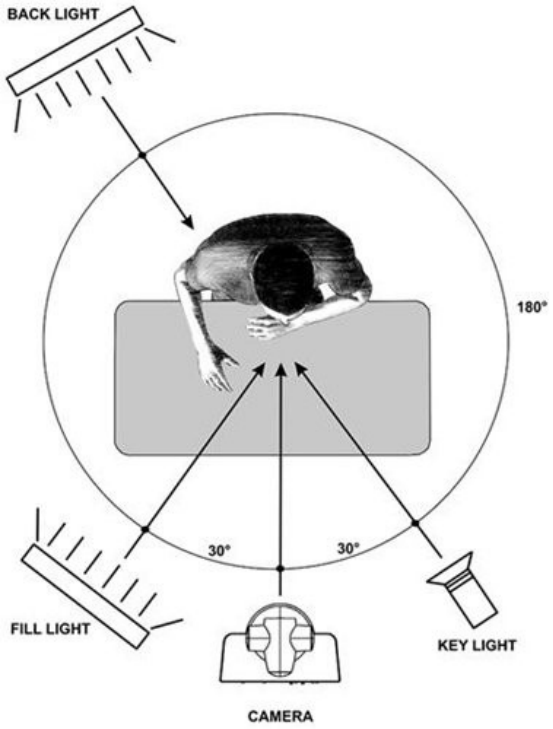
$$f = \frac{end - z}{end - start}$$



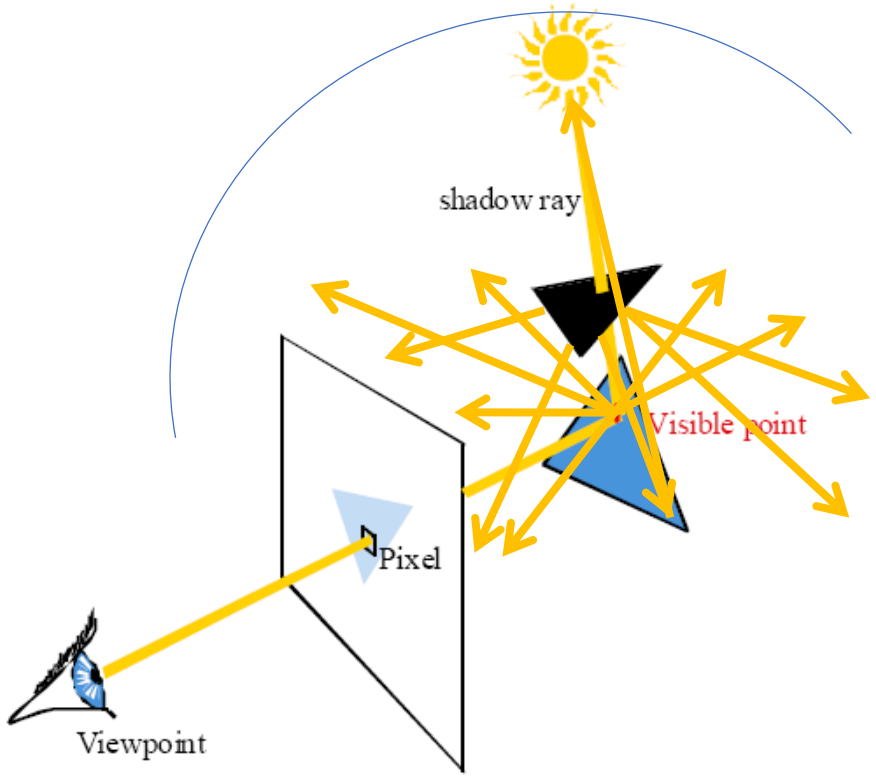


Lighting Design

- Completely black shadows are not so pleasing to the eye, so add additional lights to fill in the shadows.
  - Three lights often using in television: key, fill, back
  - Fill and back can be “area” lights, harder to produce resulting soft shadows in in object order rendering
- Loop over lights, add contributions
- Ambient shading
  - black shadows are not really right (why?)
  - one solution: dim light at camera
  - alternative: add a constant “ambient” color to the shading



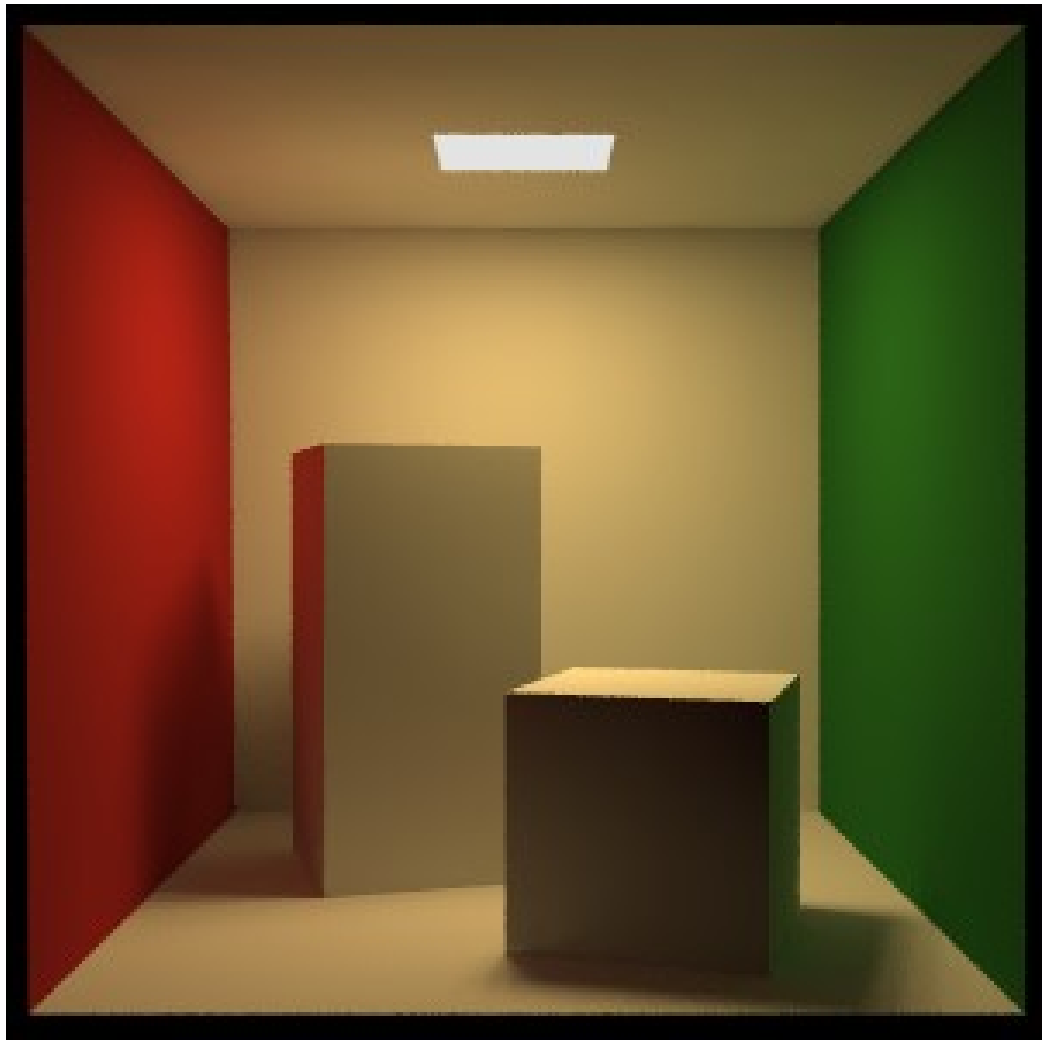
Global Illumination



- Where does the light come from?
- Can still solve this problem for special cases, or with careful sampling in both object order and image order implementations

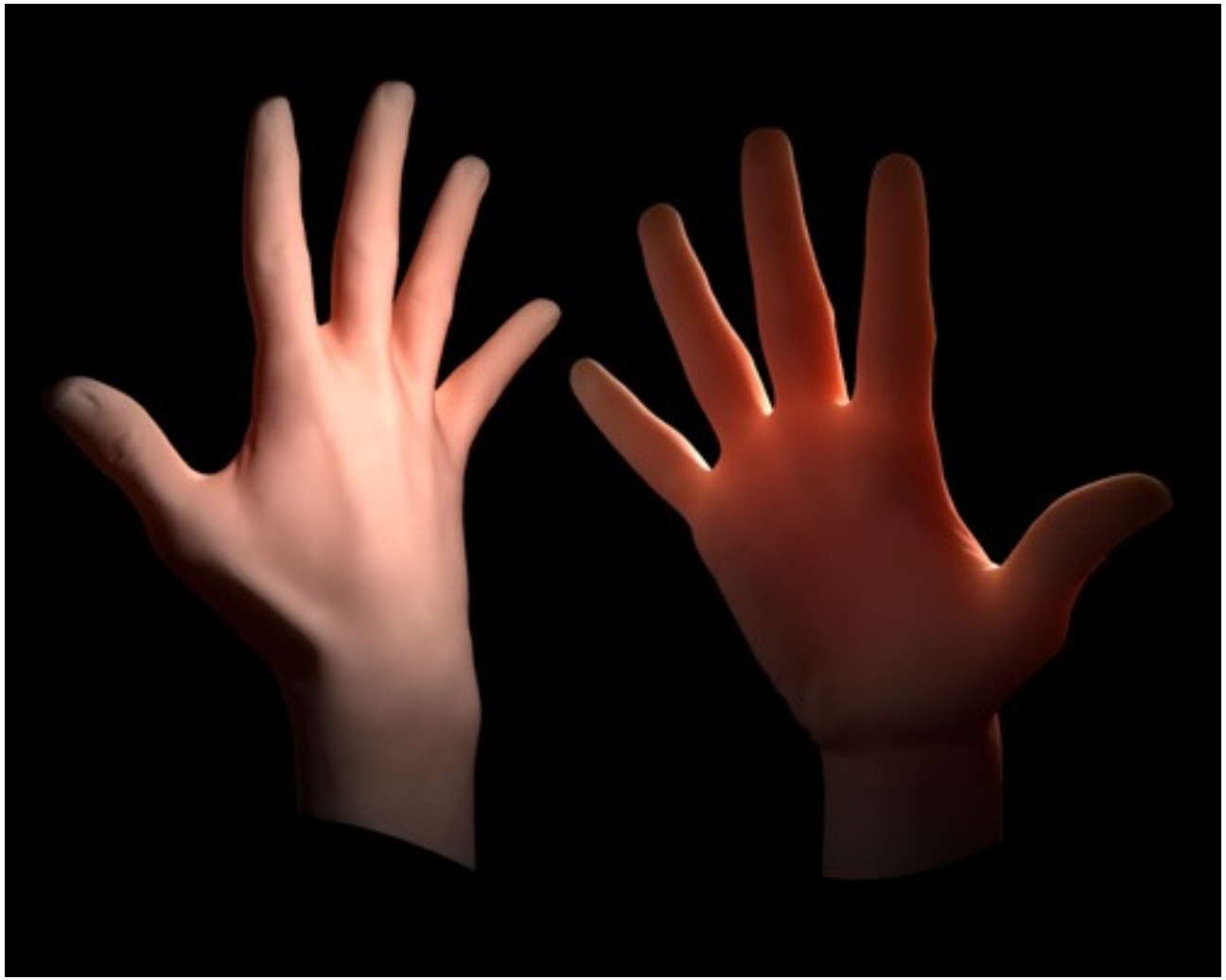
Global Illumination

- Light can bounce off walls to illuminate different surfaces
  - Red reflected light illuminates left side of left object
  - Green reflected light illuminates right side of cube on the right
- We will revisit this problem later this term!



Lighting / Shading

- The full problem is even more complex!!!
  - Light is ***everywhere*** and can even be ***scattered beneath*** the surface of materials
  - Devising a global illumination method that is fast and that correctly models all ***light transport*** is the ultimate problem in rendering.



[spin99, GELATO]

Working with OpenGL GLSL

- Need to tell our GLSL programs about materials and lights!
  - The material constants (diffuse, specular, the exponent, does the surface have a different material on the front than the back? Does it glow in the dark?)
  - How many lights, positions, or directions
- Define parameters for materials and lights as uniforms

```
#version 330 core
// can do something like this, or likewise something much simpler!!
struct LightProperties {
        bool isEnabled;
        bool isLocal;
        bool isSpot;
        vec3 ambient;
        vec3 color;
        vec3 position;
        vec3 halfVector;
        vec3 coneDirection;
        float spotCosCutoff;
        float spotExponent;
        float constantAttenuation;
        float linearAttenuation;
        float quadraticAttenuation;
};

// the set of lights to apply, per invocation of this shader
const int MaxLights = 10;
uniform LightProperties Lights[MaxLights];
uniform float Shininess;
uniform float Strength;
uniform vec3 EyeDirection
```

OpenGL Pipeline & GLSL

The graphics pipeline

- The standard approach to object-order graphics, and many versions exist
  - software, e.g., Pixar's REYES architecture
    - many options for quality and flexibility
  - hardware, e.g., graphics cards in PCs
    - amazing performance: millions of triangles per frame
- We will consider the OpenGL pipeline and consider an abstract version of hardware pipeline
- "Pipeline" because of the many stages
  - very parallelizable
  - leads to remarkable performance of graphics cards (many times the flops of the CPU at 10% to 20% the clock speed)

Simplifed pipeline overview

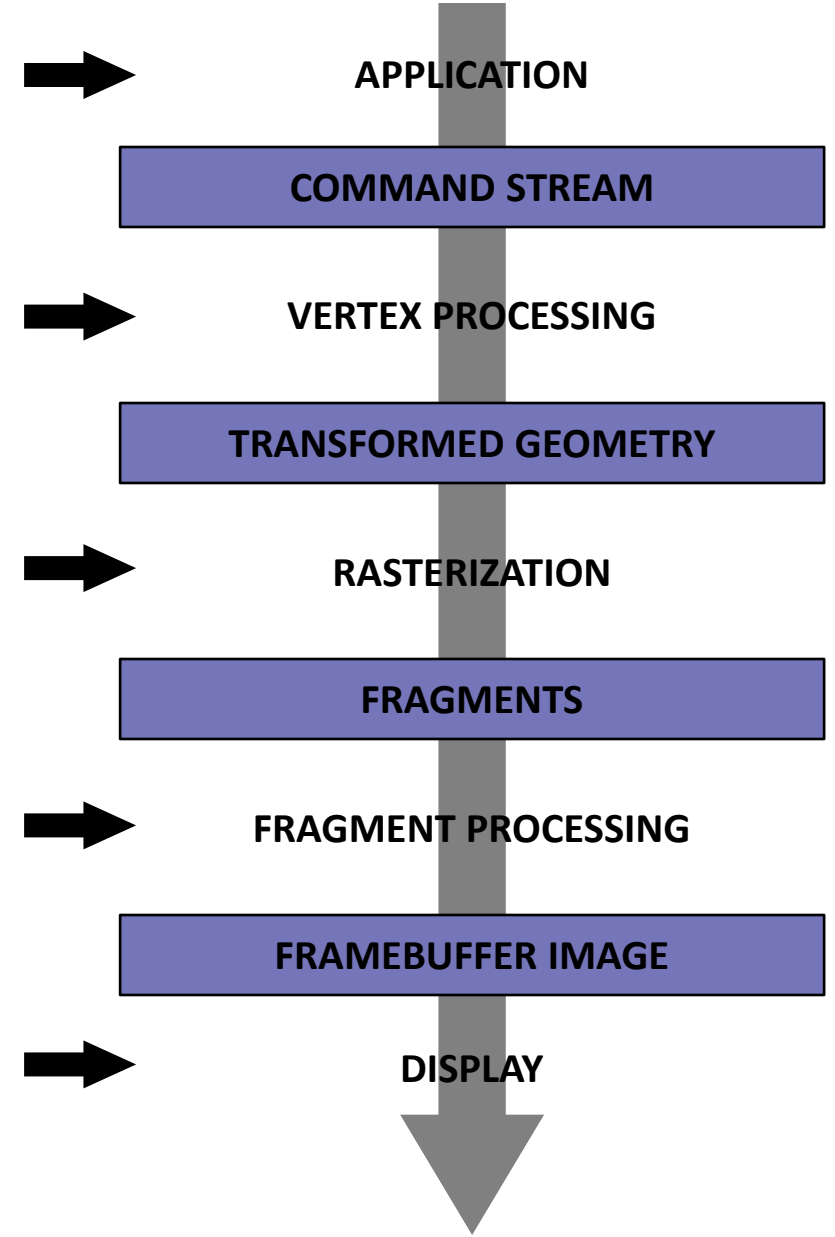
you are here

Vertex Program… 3D transformations; shading

conversion of primitives to pixels

Fragment Program… shading… followed by blending, compositing

user sees this



Primitives

- Points
- Line segments (and chains, loops of line segments)
- Triangles (and strips and fans of adjacent triangles)
- And that's all!
  - Curves? Approximate them with chains of line segments
  - Polygons? Break them up into triangles
  - Curved regions? Approximate them with triangles
- Trend has been toward minimal primitives
  - simple, uniform, repetitive: good for parallelism

Rasterization

- First job: enumerate the pixels covered by a primitive
  - simple, aliased definition: pixels whose centers fall inside
- Second job: interpolate values across the primitive
  - e.g., colors computed at vertices
  - e.g., normals at vertices

OpenGL Programmable Processors

- Shader is a program that runs on the GPU
- Shading language is appropriate for implementing both common and complex graphics calculations
- **Vertex shaders** work at the vertex level
- **Fragment shaders** work at the pixel level
- Geometry and tessellation shaders also exist

Vertex Program

```
#version 400

uniform mat4 P;
uniform mat4 V;
uniform mat4 M;

in vec4 position;

void main() {
   gl_Position = P * V * M * position;
}
```

Fragment Program

```
#version 400

out vec4 fragColor;

void main(void) {
   fragColor = vec4(1,0,0,1);
}
```

See "An Overview of the OpenGL Shading Language" in Chapter 2 "Shader Fundamentals" of OGLPG (4.3) 8th edition

GLSL Data Types

- Scalars: float, int (16 bits+), bool
  - Automatic type promotion likely works, but different syntax for casts
    - float x = 1;
    - int i = int(x);
- Vectors: vec2, vec3, vec4 (ivec and bvec too)
  - Components xyzw, rgba, or stpq
  - Easy access with swizzling: a contrived example…
    - vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
    - vec4 dup = pos.xxyy; // dup = (1,1,2,2)
    - pos.sq = dup.bg; // pos = (2,1,2,1)
- Matrices: mat2, mat3, mat4 (floating point)
    - vec4 v1, v2; mat4 m;
    - v2 = v1 * v2; // component-wise multiply
    - v2 = m*v1; // matrix vector multiply

GLSL Data Types

- Samplers, for accessing textures
  - sampler1D, sampler2D, sampler3D, samplerCube, sampler1DShadow, sampler2DShadow
  - Assigned to a texture unit, with texture units set up on the application side
    - uniform sampler2D img;
      vec2 coord = vec2(0.5, 0.5);
      vec4 color = texture2D(img, coord);
- Arrays, of any type you like, but fixed size!
  - If a shader uses a non-constant variable to index the array, that shader must explicitly declare the array with the desired size.
- Structures can also be declared

```
struct light {
   vec3 position;
   vec3 color;
};
```

GLSL Functions and syntax

- Similar to C++, but some important differences
- Functions have const, in, out, and inout parameters
    - vec3 f(const vec3 a, vec3 b, out vec3 c, inout vec3 d)
    - // a can only be read
    - // b can be read and written, but caller's variable not modified
    - // c can be read but is undefined, should be set before return
    - // d can be read, and should be written before return
    - // the function also returns a vec3
- Many useful built in functions (Appendix C OGLPG (4.3) 8th Edition)
    - radians, degrees, sin, cos, tan, asin, acos, atan…
    - pow, exp, log, epx2, log2, sqrt, inversesqrt…
    - abs, sign, floor, ceil, mod, min, max, clamp, mix…
    - length distance, dot, cross, normalize…
    - Fragment input argument derivatives: dFdx, dFdy

GLSL, variable storage qualifiers

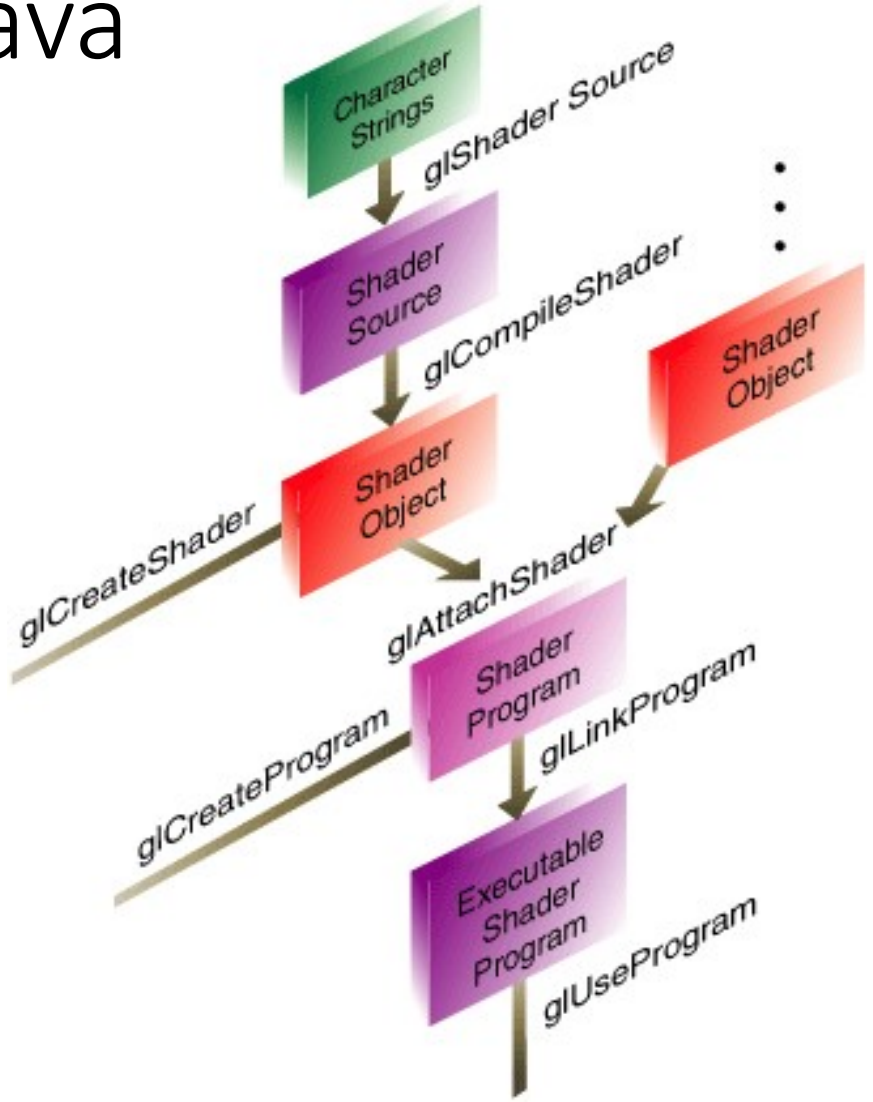
- Uniforms
  - Data that doesn't change during shader execution
  - But can change between drawing sets of primitives
    - Light positions
    - Material properties
    - Transformation matrices
- "in" and "out" declare inputs and outputs, for example
  - Vertex data, i.e., ***Attributes***, are vertex program inputs, such as position, normal, texture coordinates, skinning weights
  - Data passed OUT from vertex program for the fragment program
  - Fragment program receives IN the ***interpolated*** vertex program output
  - Fragment program output goes to the frame buffer (e.g., pixel colour)

GLSL Built-in Uniform Variables

- GLSL has built-in uniforms
  - We are likely only to use ***gl_*Position**, in the vertex program to specify the homogeneous vertex position output
  - Other built-in variables are listed in Appendix C of OGLPG8

Setting up GLSL in OpenGL / Java

- glCreateProgram, glShaderSource, glCompileShader, glAttachShader, glLinkProgram, glBindProgram
  - JOGL has ShaderCode and ShaderProgram helper classes, which attach the program to a ShaderState helper class
  - LWJGL3 more closely matches C++ API use
- Query uniforms with glGetUniformLocation
- Set uniforms with glUniform* calls
- Query attributes with getAttributeLocation
- Per vertex data typically assembled into vertex attribute buffers (note location aliasing issues)



No lighting Example (GLSL 3.30)

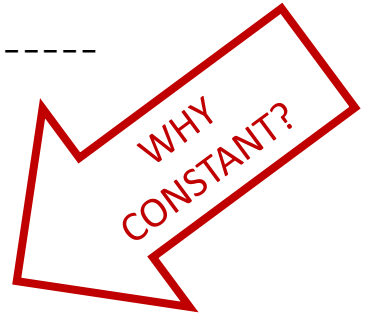
```
---------------------- Vertex Shader ------------------------
// Vertex shader with no lighting
#version 330 core
uniform mat4 MVPMatrix; // model-view-projection transform
in vec4 VertexColor; // sent from the application, includes alpha
in vec4 VertexPosition; // pre-transformed position
out vec4 Color; // sent to the rasterizer for interpolation
void main() {
    Color = VertexColor;
    gl_Position = MVPMatrix * VertexPosition;
}

---------------------- Fragment Shader ----------------------
// Fragment shader with no lighting
#version 330 core
in vec4 Color; // interpolated between vertices
out vec4 FragColor; // color result for this fragment
void main() {
    FragColor = Color;
}
```

Directional Light Example (GLSL 3.30)

```
--------------------------- Vertex Shader -----------------------------
#version 330 core
uniform mat4 MVPMatrix;
uniform mat3 NormalMatrix; // to transform normals, pre-perspective
in vec4 VertexColor;
in vec3 VertexNormal; // we now need a surface normal
in vec4 VertexPosition;
out vec4 Color;
out vec3 Normal; // interpolate the normalized surface normal
void main() {
    Color = VertexColor;
    // transform the normal, without perspective, and normalize it
    Normal = normalize(NormalMatrix * VertexNormal);
    gl_Position = MVPMatrix * VertexPosition;
}
```

Directional Light Example cont. (GLSL 3.30)

```
------------------------- Fragment Shader ---------------------------
#version 330 core
uniform vec3 Ambient;
uniform vec3 LightColor;
uniform vec3 LightDirection; // direction toward the light
uniform vec3 HalfVector; // surface orientation for shiniest spots
uniform float Shininess; // exponent for sharping highlights
uniform float Strength; // extra factor to adjust shininess
in vec4 Color;
in vec3 Normal; // surface normal, interpolated between vertices
out vec4 FragColor;
void main() {
     // compute cosine of the directions, using dot products,
     // to see how much light would be reflected
     float diffuse = max(0.0, dot(Normal, LightDirection));
     float specular = max(0.0, dot(Normal, HalfVector));
     // surfaces facing away from the light (negative dot products)
     // won't be lit by the directional light
     if (diffuse == 0.0)
          specular = 0.0;
     else
          specular = pow(specular, Shininess); // sharpen the highlight
     vec3 scatteredLight = Ambient + LightColor * diffuse;
     vec3 reflectedLight = LightColor * specular * Strength;
     // don't modulate the underlying color with reflected light,
     // only with scattered light
     vec3 rgb = min(Color.rgb * scatteredLight + reflectedLight, vec3(1.0));
     FragColor = vec4(rgb, Color.a);
}
```



Review and More Information

- OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 4.3, 8th Edition
  - Chapter 2, GLSL Shader Fundamentals (probably want to scan… it is long!)
  - Chapter 7, Light and Shadow (up to (not including) advanced lighting models)
  - Appendix C, Built in variables and functions
- FCG Chapter 10, but not Section 10.3
  - Ambient, Diffuse, Specular (Phong)
  - Also FCG Chapter 4 Section 4.5 on the basic shading equations in the context of ray tracing
- CGPP 14.9 Standard Approximations