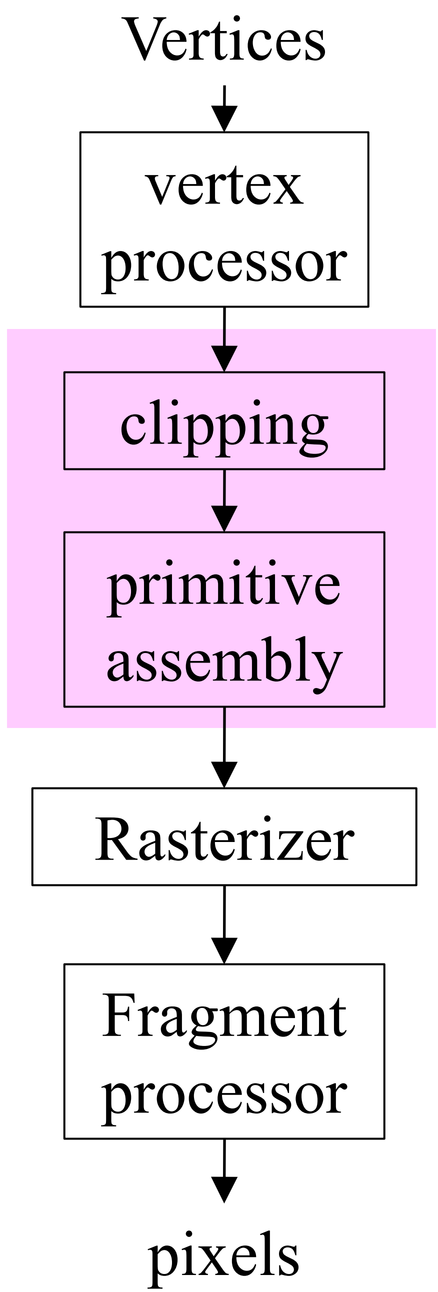
Fixed Functional Pipeline: compatibility OpenGL



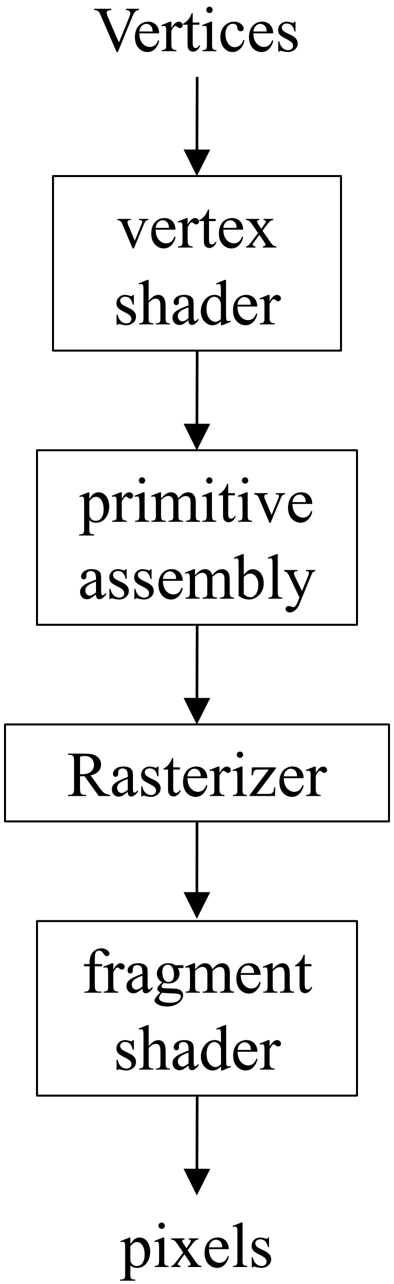
Advantages

- \+ standard, backward compatible
- \+ opaque API
  - learning, use
  - C/C++, JOGL...
- \+ good results – lighting ...
- \+ good performance

Disadvantages

- \- light calculations once / vertex interpolate pixel
- \- multiple light models
- \- 9+ lights
- \- animation and vertex blending
- \- exploit parallel computation, extensions on GPU

Programmable Pipeline: core OpenGL



Advantages

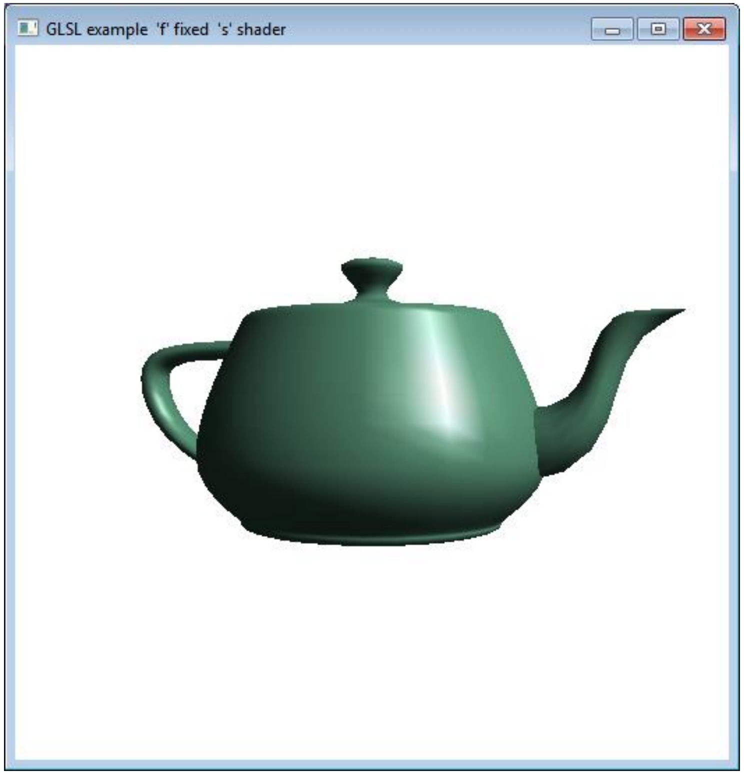
+ app specific vertex and fragment processing
    - vertex blending w/ animation
    - many lights, lighting models
+ C like variables and operations
+ built in vector and matrix operations
+ enhanced (parallel) performance

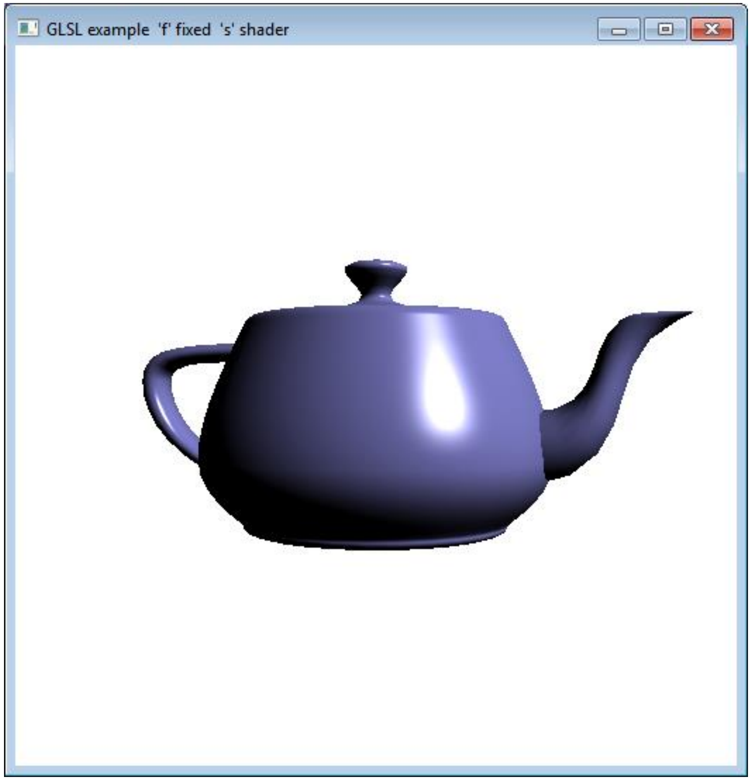
Disadvantages

- must replace fixed functionality w/in pipeline
- debugging
- multiple source files ...

GLSL	OpenGL Shading Language
Cg	C for graphics, Nvidia OpenGL and DirectX
HLSL	High Level Shading Language DirectX

Fixed (compatible) Vs. Shader (core) teapot



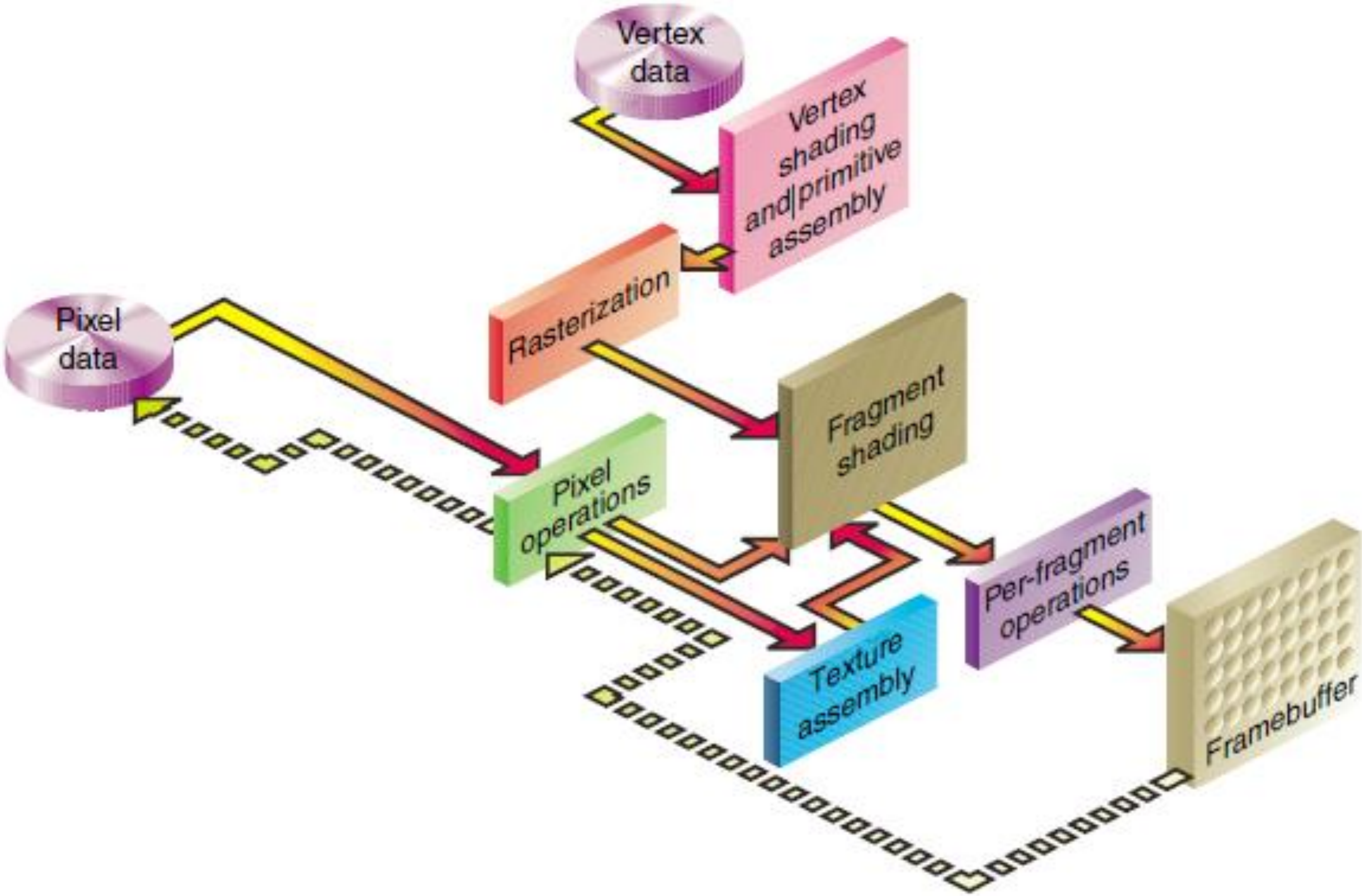


Note definition of specular light effect

This example was written in compatible mode and with an older GLSL version (2.1)

Order of operations

http://www.opengl-redbook.com/appendices/AppE.pdf
7th edition



Manage GLSL programs

Create, get source, compile, attach, link, use ...

Shader program can contain many shader programs

```
GLuint glCreateProgram();

GLuint glCreateShader(GLenum type); // create shader

   type     GL_VERTEX_SHADER, GL_FRAGMENT_SHADER

void glShaderSource(GLuint shader, GLsizei count,
   GLchar ** string, const GLint * length);

   count    # of source strings
   length   NULL if null terminated strings, ...

void glCompileShader(GLuint shader);
void glAttachShader(GLuint program, GLuint shader);
   // glDetach(GLuint program, GLuint shader);
void glUseProgram(GLuint program);
   // glUseProgram(0);  resets to fixed functional
void glLinkProgramGLuint program);
...
```

OpenGL application ➔ GLSL

Consider an prototypical OpenGL application with:

n models (*.tri) ➔ arrays of vertex[], color[], and normal[]

3 transform matrices {Model, View, Project}

OpenGL application

loads model ➔ arrays ➔ GPU buffer(s)

creates GLuint "index" variables to reference GLSL objects

"on events"

system, user ➔ set Project `// glutReshapeFunc()`

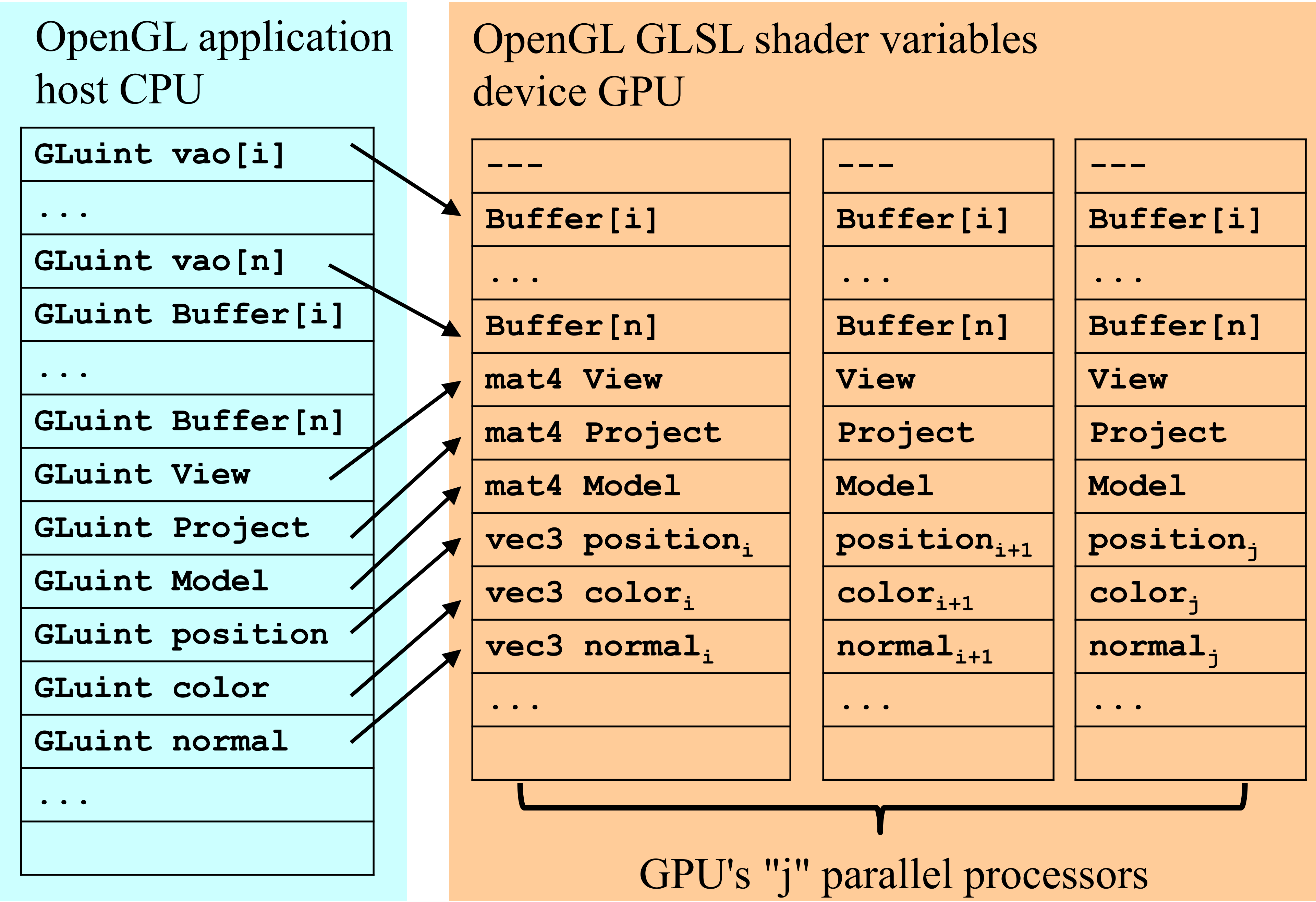
user ➔ set View `// glutKeyboardFunc()`

draw ➔ set Model, set shader variables {vertex, color, normal}

`// glutTimerFunc(), glutDisplayFunc()`

draw request `// glDrawArrays()`

GLuints maps to GLSL variables



Variables

atomic types

```
int, uint, float, double, bool
```

vectors

```
vec2, vec3, vec4  // float, ivec2 ... bvec2
vec4 color = vec4(0.0, 1.0, 1.0, 1.0);
   // r,g,b,a for "cyan"
vec3 rgb = vec3(color) // first 3 values
float red = rgb.r;
float green = rgb[1];
```

matrices

```
mat2, mat3, mat4  // float, square
            // OpenGL column storage, but:
m[1]        // second row
m[1][2]     // cell row 2 column 3
m = mat3(1.0);  //   1.0   0.0   0.0
                     0.0   1.0   0.0
                     0.0   0.0   1.0
mat4 translate = mat4(  1.0, 0.0, 0.0,  100.0,
                        0.0, 1.0, 0.0,   50.0,
                        0.0, 0.0, 1.0, -200.0,
                        0.0, 0.0, 0.0,    1.0);
```

Arrays, structs, samplers

arrays

```
[n], [i][j]  // C like, arrays of arrays
vec4 lightPos[];      // or [n]
lightPos.length();    // # values
```

struct

```
struct Particle {
   float lifeTime,
   vec3 position, velocity; };
Particle p = Particle(10.0, pos, vel);  // assume vec3s
```

samplers

```
sampler1D, sampler2D // textures
```

GLSL Built-in variables, Functions see Appendix C redbook

built-in (vec4): `gl_Position  // Vertex program`

Type Modifiers

const	read only
in	input only to shader stage
out	output only from shader stage
uniform	unknown at compile time, const w/in shader, globally declared, set in application, shared by shaders – pass information into shader `uniform sampler2D aTexture;`
buffer	read/write memory shared with application

Function argument type qualifiers:

in, const in	input only argument, must have a value
out	output only argument, set in function (no in value)
inout	input and output argument, can have null value on input

Access shader defined variables

Shader variables declared as in (input) are accessed by an index (GLuint), need to get index, enable, and set type, usage, and buffer position

```
GLuint glGetAttribLocation(GLuint program,
   const char * name);  // get

void glVertexAttribPointer(GLuint index, GLint size,
   GLenum type, GLsizei stride, GLbool normalized,
   const GLvoid * pointer);
   // set

// set up vertex arrays (after shaders are loaded)
GLuint vPosition = glGetAttribLocation( shaderProgram,
   "vPosition" );  // vPosition maps to shader var "vPosition"

glEnableVertexAttribArray( vPosition );

glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
   BUFFER_OFFSET(0) );  // set what and where vPosition maps to
```

Access uniform variables

```
GLuint glGetUniformLocation(GLuint program,
   const char * name);  // get
```

There are many "set" versions (see redbook)

```
void glUniform*(GLuint location, Type value);  // set

void glUniformMatrix*(GLuint location, GLsizei count,
   GLboolean transpose const GLfloat *  values);  // set
```

```
GLuint MVP = glGetUniformLocation(shaderProgram,
"ModelViewProject");

...

modelViewProjectionMatrix = projectionMatrix *
   viewMatrix * modelMatrix;

glUniformMatrix4fv(MVP, 1, GL_FALSE,
   glm::value_ptr(modelViewProjectionMatrix));
```

Operators

Indexes. Vectors, matrices and arrays can be indexed with [ ]

```
vec4 v = vec4(1.0, 2.0, 3.0,  4.0);
float f = v[2]; // f == 3.0
mat4 m = mat4(3.0);  // diagonals set to 3.0
v = m[1];  //  v == (0.0, 3.0, 0.0, 0.0)
           //  m's second column.
```

Swizzling. Dot or structure-member selection can "swizzle" vector components.

```
vec4 v4;
v4.rgba; // v4 vec4 with red, green, blue, alpha
v4.rgb;  // a vec3
v4.b;    // a float
v4.xy;   // a vec2

vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);
vec4 swiz = pos.wzyx // swiz == (4.0, 3.0, 2.0, 1.0)
vec4 dup = pos.xxyy; // dup == (1.0, 1.0, 2.0, 2.0)
```

Component-Wise Operators

Operator applied to vector is applied to each component.

```
vec3 v, u, w;
float f;
v = u + f;
// EQV u.x = u.x + f;  u.y = u.y + f;  u.z = u.z + f;
w = v + u;
// EQV w.x = v.x + u.x; ... w.z = y.z + u.z
```

Exceptions: vector times matrix and matrix time matrix perform standard linear algebraic multiplies.

Functions

Many built-in functions: angle, trigonometry. exponential, range, geometric, vector relationship, texture access, noise (randomness).

see redbook Appendix C

`<type>` is used to represent the same type in the function

```
vec3 radians(vec3 degree)
```

```
<type> radians(<type> degree)  // float, vec2 ... vec4
<type> degrees(<type> radians)
<type> sin(<type> radians)    // cos, tan, a*
<type> clamp(<type> x, float minVal, float maxVal)
   // returns min(max(x, minVal), maxVal) components
<type> mix(<type> x, <type> y, float a)
   // returns x * (1.0 - a) + y*a  the linear blend
float length(<type> x)  // length of vector
float distance(<type> p0, <type> p1)  // length(p0 - p1)
float dot(<type> x, <type> y)  // dot product of x and y
vec3 cross(vec3 x, vec3 y)  // cross product of x and y
<type> normalize(<type> x)
```

vertexReview1.glsl

```
# version 330 core

uniform float eyePosition;  // not used, for later version

uniform mat4 ModelView;
uniform mat4 Projection;

in vec4 vPosition;
in vec4 vColor;
in vec3 vNormal;

out vec4 vsColor;
out vec3 vs_worldpos;
out vec3 vs_normal;

void main(void) {
  vec4 position = Projection * ModelView * vPosition;
  gl_Position = position; // GLSL vertex built-in variable
  vs_worldpos = position.xyz;
  // mat3(ModelView) is a NormalMatrix
  vs_normal = mat3(ModelView) * vNormal;
  vsColor = vColor;
  }
```

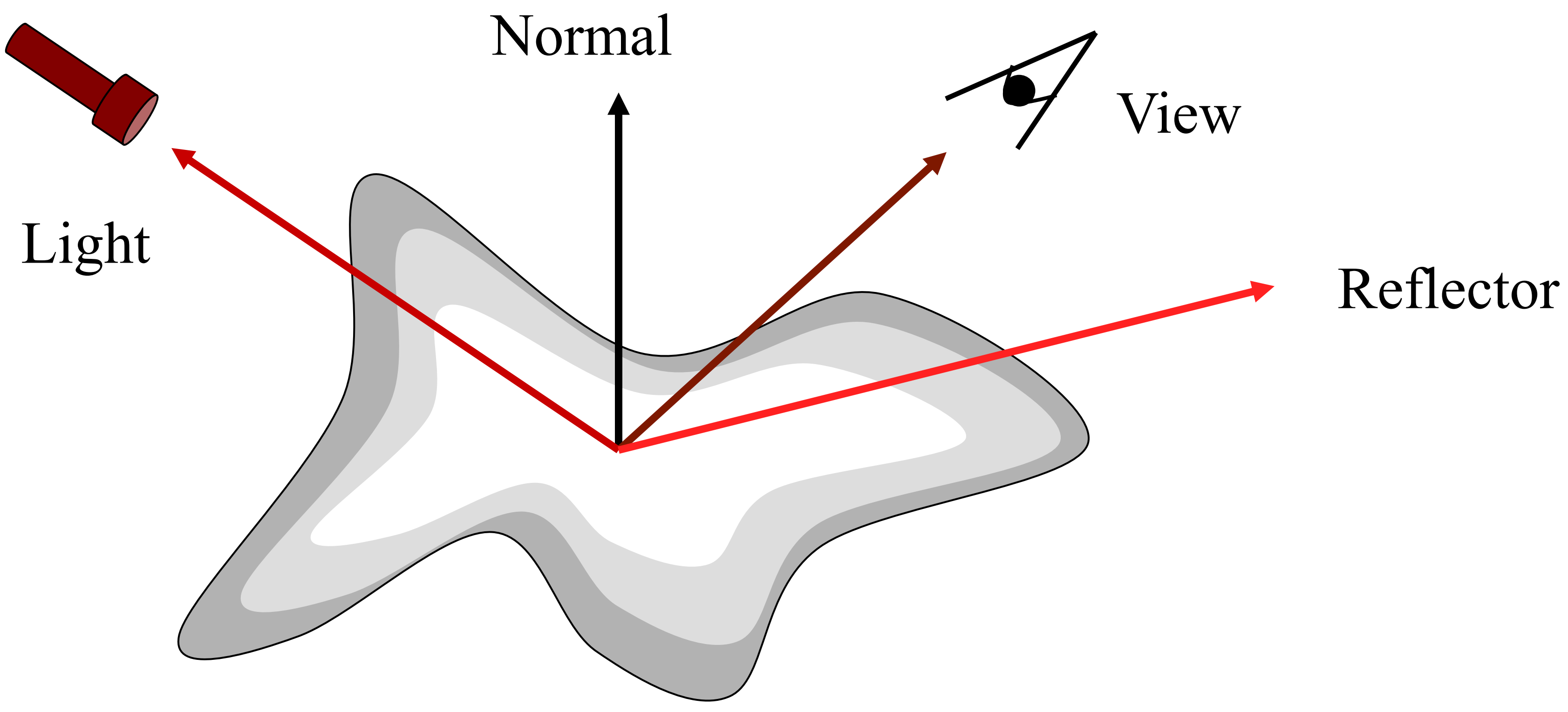
Light

A point on a surface is seen as reflected light from a view.

The light can have an intensity and a color that illuminates a surface.

The surface has color and reflective "material" properties
color or texture

Lights have no geometry (may need "shape" to represent light source)



Light model

4 contributions to the shading of a point

diffuse light reflected in all directions (!V)
ambient sum of all light source (intesity of light, !R !V)
specular shininess of reflections from smooth surface (R V)
emissive glowing – light source (!L, lights ! visible)

Unit normals must be provided for every face at the vertex if openGL lighting is to be used.

Typical light types: ambient (flat), directional, point, and spotlight
each has: diffuse, specular and ambient RGB parameters

Imagine a bare white light "bulb" in a room with green walls
diffuse and specular parameters are white
reflections are green – so the ambient parameter is green

Light source, colors, material

```
vec4 lightPos    = {10.0, 20.0, 50.0, 1.0};
vec4 diffuse[]  = {1.0, 0.0, 0.0, 1.0};   // red
vec4 specular[] = {1.0, 1.0, 1.0, 1.0};   // white
vec4 ambient[]  = {0.1, 0.1, 0.1, 1.0};   // grey
```

Ambient light has no source.
Each "sourced light" can contribute to the ambient lighting.

Directional, (distant or parallel) light has no location only direction
Most efficient lighting calculations.

Point light is like a bare light bulb.
Light emits in all directions and has an attenuation with distance

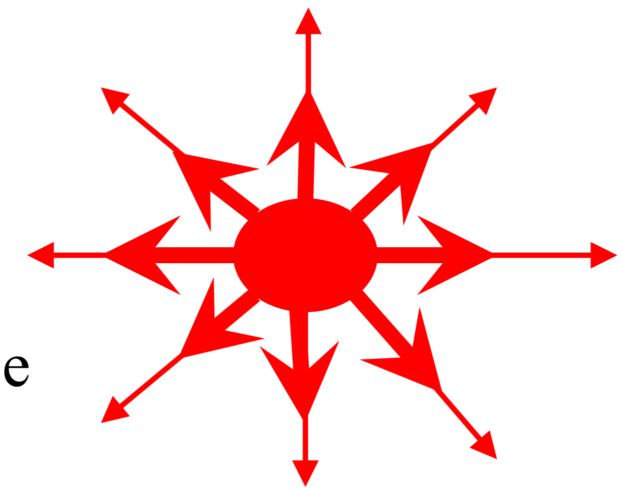
Spot light is like a flashlight, it has inner and outer cone of intensity

LightPos 4^{th} argument 1 == point source or 0 == directional source

Light sources

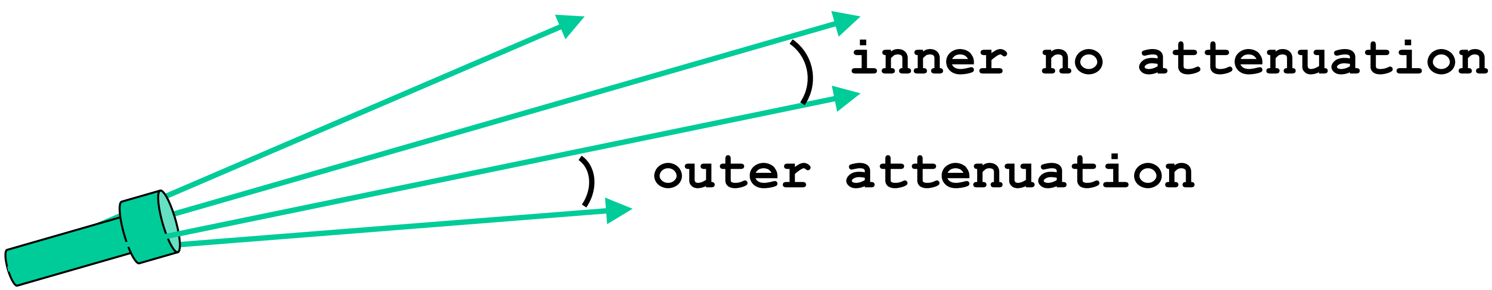
Ambient light has no source.
Each "sourced light" can contribute to the ambient lighting.

Distant (parallel) light is like the sun.
Most efficient lighting calculations.



Point light is like a lightbulb.
Light emits in all directions
and has an attenuation with distance

Spot light is like a flashlight, it has inner and outer cone of intensity



<< lightposition tutor, compatibility OpenGL >>

Materials

Materials have: ambient, diffuse, specular, and shininess values/
// material values from E. Angel, OpenGL A Primer, 2002.

Consider using a struct for a material (could also have emmissive ...)

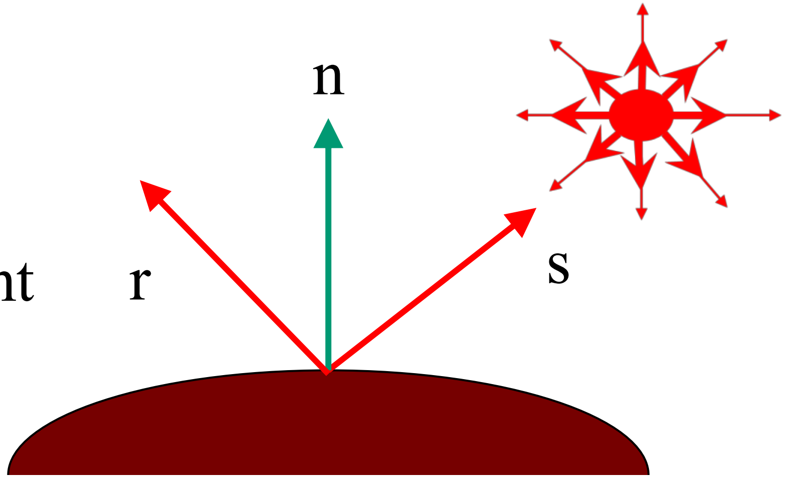
```
struct Material {  // use in shaders
   vec3 ambient, diffuse, specular;
   float shininess;
   };

Material brass = {
   vec3(0.33, 0.22, 0.03),  // ambient
   vec3{0.78, 0.57, 0.11), // diffuse
   vec3{0.99, 0.91, 0.81}, // specular
   27.8f};                 // shininess
```

or an array and a float // OpenGL → GLSL

```
glm::mat3 = glm::mat3(  // red plastic
   glm::vec3(0.3, 0.0, 0.0),
   glm::vec3(0.6, 0.0, 0.0),
   glm::vec3(0.8, 0.6, 0.6));
float shininess = 32.0f;
```

<< lightmaterial tutor >>

Ambient and Diffuse light



Ambient light intensity (I_a) at a surface point has been scattered by many reflections and is the source light's ambient color (L_a) scaled by surface's ambient (material) reflectivity constant (K_a).

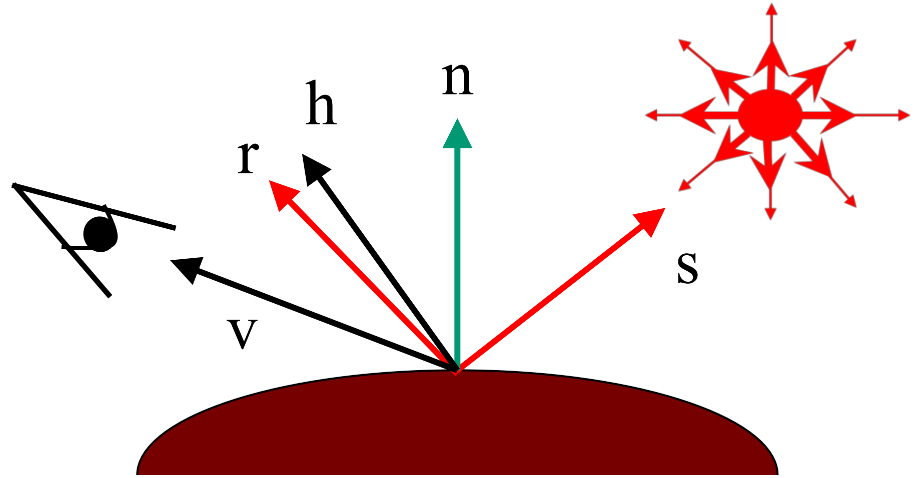
$$I_a = L_a * K_a$$

Diffuse light intensity (I_d) at a surface point is a function of the source light intensity (L_d) , the point's normal (n) to the surface, and the surface's diffuse reflectivity constant (K_d)

$$I_d = L_d * K_d * \mathrm{dot}(s,n)$$

Ambient and Diffuse lights are not affected by the position of the viewer. Diffuse light is omni-directionally reflected.

Specular light



Specular light is the "shiny surface reflection from the light source.

Reflection vector (r)
GLSL reflect function:

r = -s + 2 * dot(s,n) * n

Specular component is visible when the view (v) is aligned with the light reflection (r) (light reflects towards the view).

Specular intensity is "sharply" proportional to alignment (power of f)

$I_s = L_s * K_s * dot(r, v)^f$

Half vector (h)

Given view (v) aligned with reflection (r), the normal (n) is half way between v and s.

h = normalize(v + s)

h is a simpler computation than r, so h is often used in specular lighting

$I_s = L_s * K_s * dot(h, n)^f$

ADS vertex shader w/ functions

Shaders from D. Wolff, "function.vert" and "function.frag", pp 62-63

GLSL functions similar to "C", however, use a "call by value return" or "call by value result".

Arguments qualified by in, out, inout; out and inout values copied back into argument at return.

Arrays and structures are also passed by value – use "global scope"

```
#version 400

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
out vec3 LightIntensity;

struct LightInfo {
   vec4 Position; // Light position in eye coords.
   vec3 La;       // Ambient light intensity
   vec3 Ld;       // Diffuse light intensity
   vec3 Ls;       // Specular light intensity
   };
uniform LightInfo Light;
```

```
struct MaterialInfo {
   vec3 Ka;              // Ambient reflectivity
   vec3 Kd;              // Diffuse reflectivity
   vec3 Ks;              // Specular reflectivity
   float Shininess;      // Specular shininess factor
   };

uniform MaterialInfo Material;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void getEyeSpace( out vec3 norm, out vec4 position ){
   norm = normalize( NormalMatrix * VertexNormal);
   position = ModelViewMatrix * vec4(VertexPosition,1.0);
   }
```

```
//  approximate Phong shading in the vertex
vec3 phongModel( vec4 position, vec3 norm ) {
   vec3 s = normalize(vec3(Light.Position - position));
   vec3 v = normalize(-position.xyz);
   vec3 r = reflect( -s, norm );
   vec3 ambient = Light.La * Material.Ka;
   float sDotN = max( dot(s,norm), 0.0 );
   vec3 diffuse = Light.Ld * Material.Kd * sDotN;
   vec3 spec = vec3(0.0);
   if( sDotN > 0.0 ) // only work with visible faces
      spec = Light.Ls * Material.Ks *
         pow( max( dot(r,v), 0.0 ), Material.Shininess );
   return ambient + diffuse + spec;
   }
```

```
void main() {
   vec3 eyeNorm;
   vec4 eyePosition;

   // Get the position and normal in eye space
   getEyeSpace(eyeNorm, eyePosition);

   // Evaluate the lighting equation.
   LightIntensity = phongModel( eyePosition, eyeNorm );

   gl_Position = MVP * vec4(VertexPosition,1.0);
   }
```

Per Vertex ADS fragment shader

```
#version 400

in vec3 LightIntensity;
layout( location = 0 ) out vec4 FragColor;

void main() {
   FragColor = vec4(LightIntensity, 1.0);
   }
```

Per-vertex lighting has poor specular highlights

compute only for vertices not for each point on surface

specular should be in center of surface, calculated at vertex

where specular component might be near zero.

Per-fragment lighting preferred (Phong shading model)

Interpolate the position and normal vectors across the surface

for shading each fragment

Phong vertex shader

```
#version 400

layout (location = 0) in vec3 VertexPosition;
layout (location - 1) in vec3 VertexNormal;
out vec3 Position;   // pass position to fragment shader
out vec3 Normal;     // pass normal to fragment shader
uniform mat4 ModelViewMatrix;
uniform mat4 Normal Matrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main() {
   Normal = normalize(NormalMatrix * VertexNormal);
   Position = vec3(ModelViewMatrix *
      vec4(VertexPosition, 1.0);
   gl_Position = MVP * vec4(VertexPosition, 1.0);
   }
```

The "out" values of Position and Normal are automatically interpolated between the vertices by "fixed pipeline" assembly / rasterization stage between the shaders.

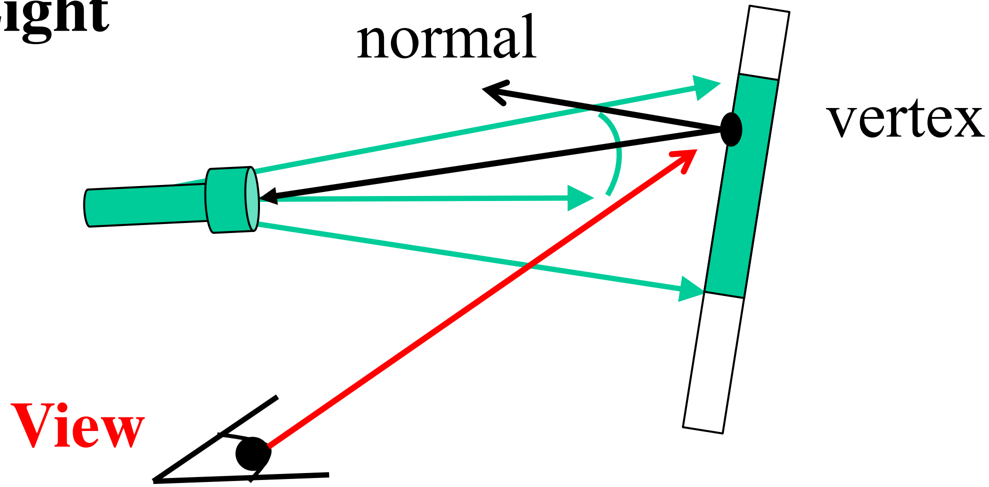
Phong fragment shader

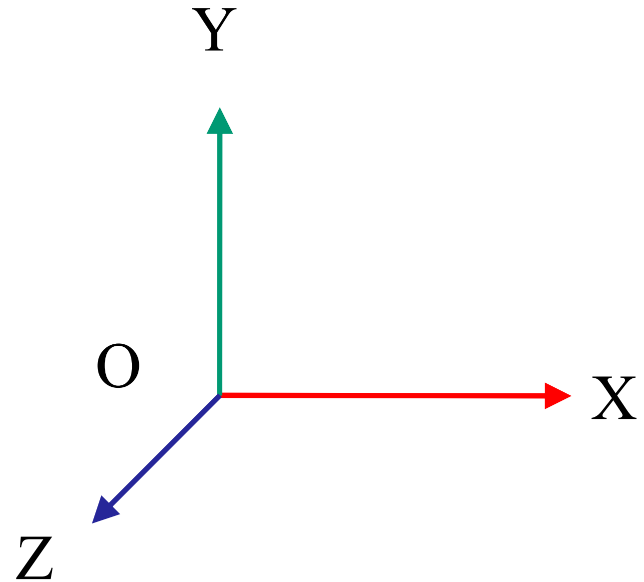
```
#version 400
in vec3 Position;
in vec3 Normal;
uniform vec4 LightPosition;
uniform vec3 LightIntensity;
uniform vec3 Kd;               // Diffuse reflectivity
uniform vec3 Ka;               // Ambient reflectivity
uniform vec3 Ks;               // Specular reflectivity
uniform float Shininess;       // Specular shininess factor
layout( location = 0 ) out vec4 FragColor;

vec3 ads( ){
   vec3 n = normalize(Normal);
   vec3 s = normalize( vec3(LightPosition) - Position );
   vec3 v = normalize(vec3(-Position));
   vec3 h = normalize (v + s);
   return  LightIntensity * ( Ka +
      Kd * max( dot(s, Normal), 0.0 ) +
      Ks * pow( max( dot(h, n), 0.0 ), Shininess ) );
   }

void main() { FragColor = vec4(ads(), 1.0); }
```

fragmentReview1.glsl

```
# version 330 core

in  vec4 vsColor;
out vec4 color;
in vec3 vs_worldpos;
in vec3 vs_normal;

// define local light propoerties
vec3 light_position = vec3(0.0f, 0.0f, 40000.0f);
vec4 color_ambient = vec4(0.5, 0.5, 0.5, 1.0);
vec4 color_diffuse = vec4(0.4, 0.4, 0.4, 1.0);
vec4 color_specular = vec4(1.0, 1.0, 1.0, 1.0);
float shininess = 50.0f;

void main(void) {
   float ambient = 0.05f;   // scale the ambient light
   vec3 light_direction = normalize(light_position - vs_worldpos);
   vec3 normal = normalize(vs_normal);
   vec3 half_vector = normalize(light_direction +
        normalize(vs_worldpos));
   float diffuse = max(0.0, dot(normal, light_direction));
   float specular = pow(max(0.0, dot(normal, half_vector)), shininess);
   color = ambient * color_ambient + diffuse * vsColor + specular *
        color_specular;
   }
```

Spot Light





Spot light has
- position in eye space
- direction, normalized in eye space
- intensity, full color
- "cutoff angle" where light is visible
- angular attenuation with distance from light's center direction

Compute "spot light factor" a scaling of the lights intensity within the cutoff angle.

Determine proportion of light visible from the view.

```
// All arguments are in eye space
// function in GLSL fragment shader
// vec3 positionEyeSpace is the "vertex"
float spotLightFactor(vec3 spotPosition, vec3 normal {
   vec3 viewDirection = normalize( - positionEyeSpace);
   vec3 direction =  spotPosition - positionEyeSpace;
   float attenuation = inversesqrt(length(direction)) * 3;
   direction = normalize(direction);
   float angle = acos(dot(direction, spotLightDirection));
   float cutoff = radians(clamp(spotLightCutoff, 0.0,
      90.0));
   float factor = 0.0f;
   if (angle < cutoff)
   factor = dot(direction, spotLightDirection) *
      attenuation;
   return max(dot(direction, viewDirection), 0.0f) *
      factor ;
// return max(dot(-direction, normal), 0.0f) *
      factor ;  // D. Wolff's version, same result

   }
```

???

White sphere on plane.

spotLightDirection =
(0.0, 0.0, -1.0)

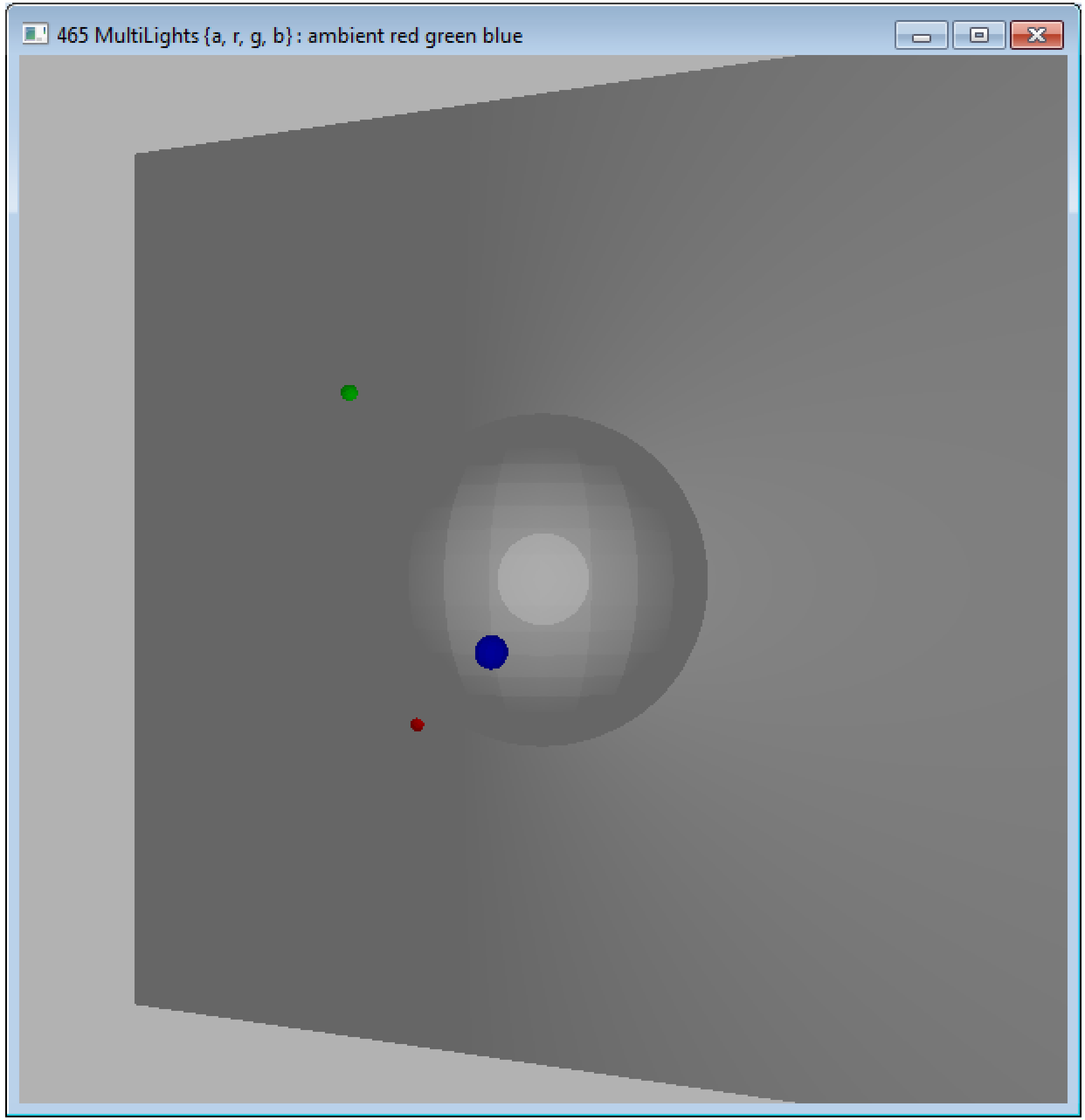
cutoff angle = 5°

red	(300, 0, 300)
green	(300, 0, 600)
blue	(300, 0, 1200)

3 spot lights rotate above sphere

Intensity attenuates with distance.

Why doesn't spot light change with light positions and lights?



References

Wolff, D., OpenGL 4.0 Shading Language Cookbook, Packt Publishing, 2011, pp 323. `http://PacktLib.PacktPub.com`

source code (uses Qt, freeGlut, GLM)

`https://github.com/daw42/glslcookbook`

There is a second edition of source code available from PacktLib site that does not use freeGlut, it uses GLM and GLFW.