CSE 167: Introduction to Computer Graphics Discussion 3

TA: Jimmy Ye ft. Tutor: Kevin Huang
University of California, San Diego
Fall Quarter 2018

Announcements

- Project 2 due this Friday at 2pm
  - Grading in CSE basement labs B260 and B270
  - This time using Autograder (no whiteboard)
  - Upload code to TritonEd by 2pm

Overview

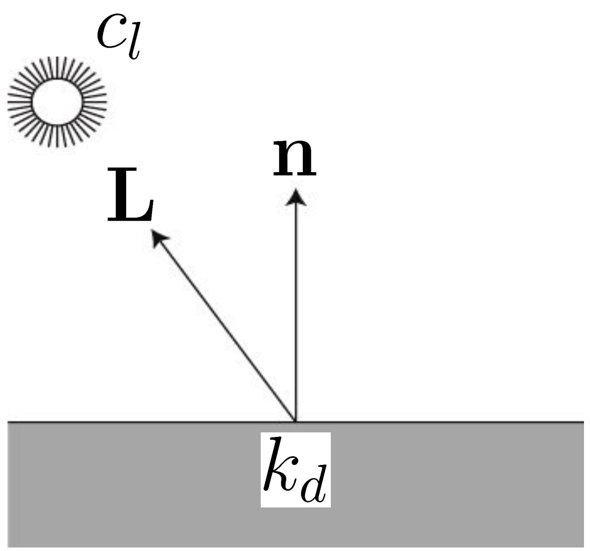
- Lecture review
- Overview of next lecture (lights) for HW2
- Common errors
- Implementation hints

Diffuse Reflection

- Given
  - Unit (normalized!) surface normal **n**
  - Unit (normalized!) light direction **L**
  - Material diffuse reflectance (material color) k_d
  - Light color (intensity) c_l

- Diffuse color c_d is:

$$c_d = c_l k_d \underbrace{(\mathbf{n} \cdot \mathbf{L})}_{\text{Proportional to cosine between normal and light}}$$



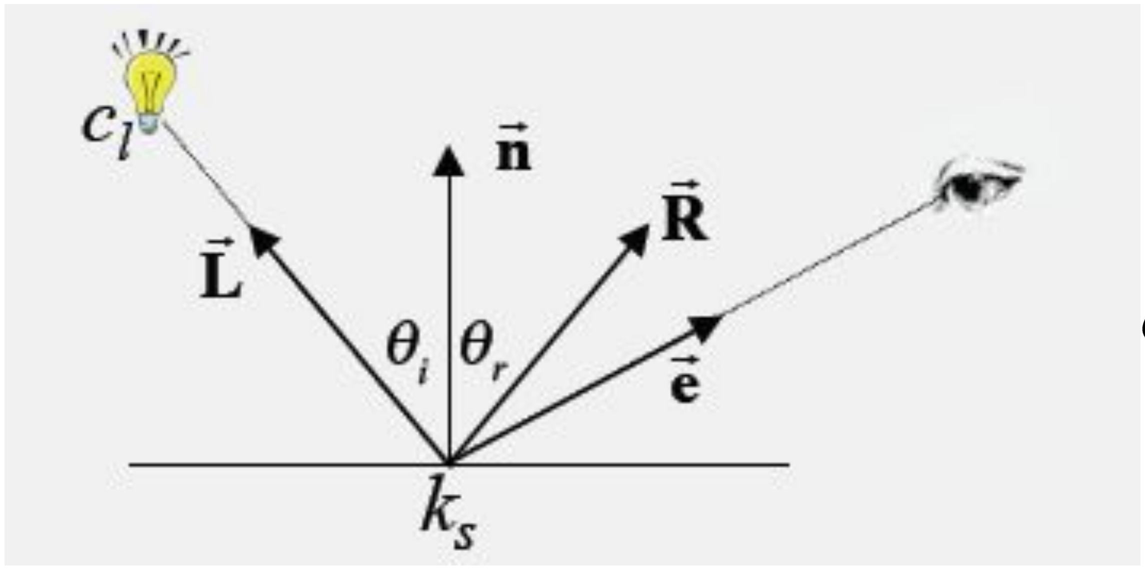
Diffuse Reflection

Notes

- Parameters k_d, c_l are r,g,b vectors
- Need to compute r,g,b values of diffuse color c_d separately
- Parameters in this model have no precise physical meaning
  - c_l: strength, color of light source
  - k_d: fraction of reflected light, material color

Phong Shading Model

- Developed by Bui Tuong Phong in1973
- Specular reflectance coefficient k_s
- Phong exponent p
  - Greater p means smaller (sharper) highlight



$$c = k_s c_l \left(\mathbf{R} \cdot \mathbf{e}\right)^p$$

Complete Phong Shading Model

- Phong model supports multiple light sources
- All light colors *c* and material coefficients *k* are 3-component vectors for red, green, blue

$$c = \sum_i c_{l_i}(k_d(L_i \cdot n) + k_s(R \cdot e)^p + k_a)$$

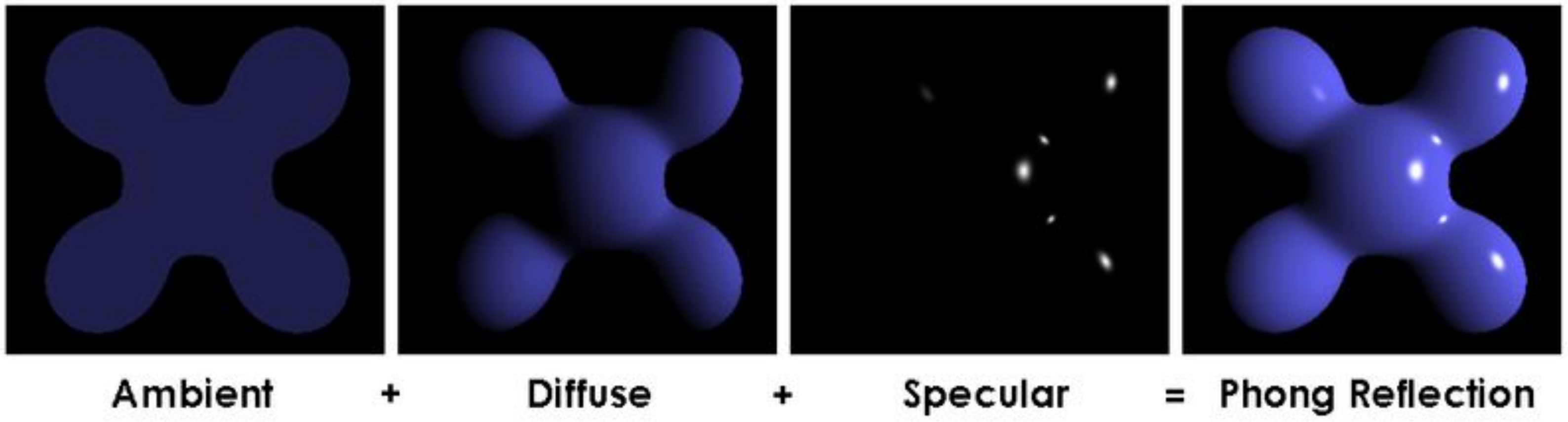


Image by Brad Smith

Lights

- Quick overview just for the basic ideas + equations needed for the lighting portion of HW2
- More in the next lecture

Light Sources

- Real light sources can have complex properties
- We use simplified model for real-time rendering

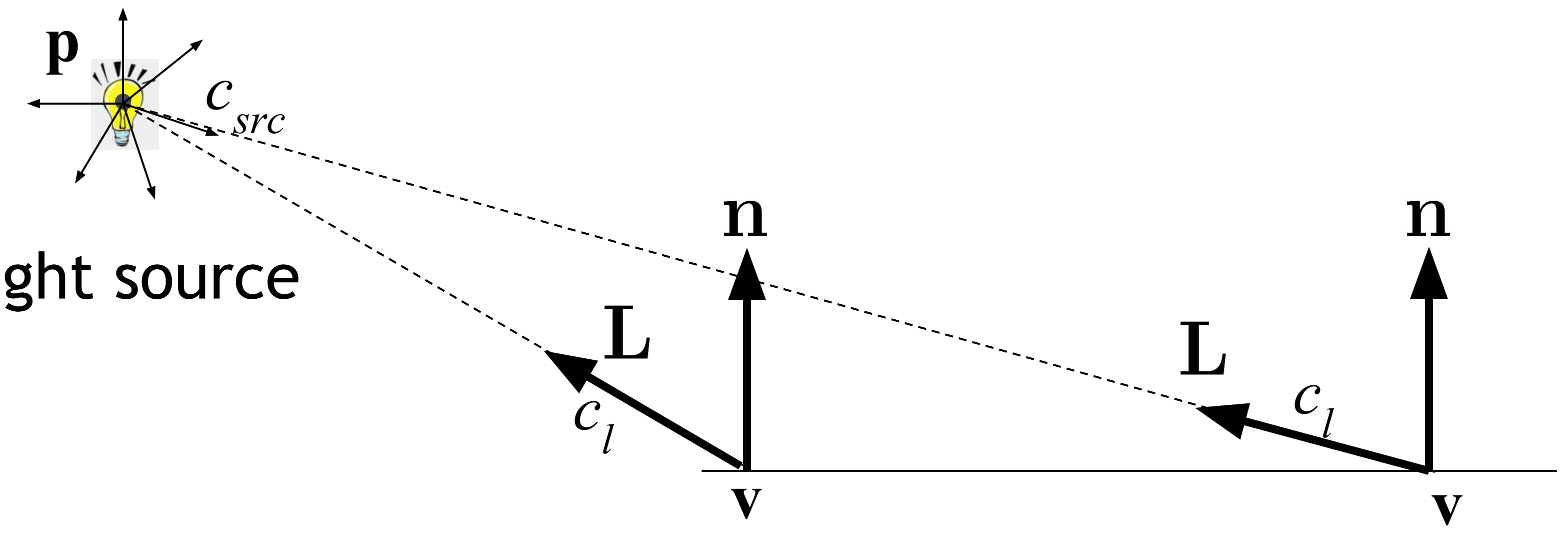
Types of Light Sources

- At each point on surfaces we need to know
  - Direction of incoming light (the $\mathbf{L}$ vector)
  - Intensity of incoming light (the c_l values)
- Three light types:
  - ~~Directional: from a specific direction~~
  - Point light: from a specific point
  - Spotlight: from a specific point with intensity that depends on direction

Point Lights

- Similar to light bulbs
- Infinitely small point radiates light equally in all directions

Point Light Math



Receiving surface

At any point v on the surface:

$$\mathbf{L} = \frac{\mathbf{p} - \mathbf{v}}{\|\mathbf{p} - \mathbf{v}\|}$$

Attenuation:

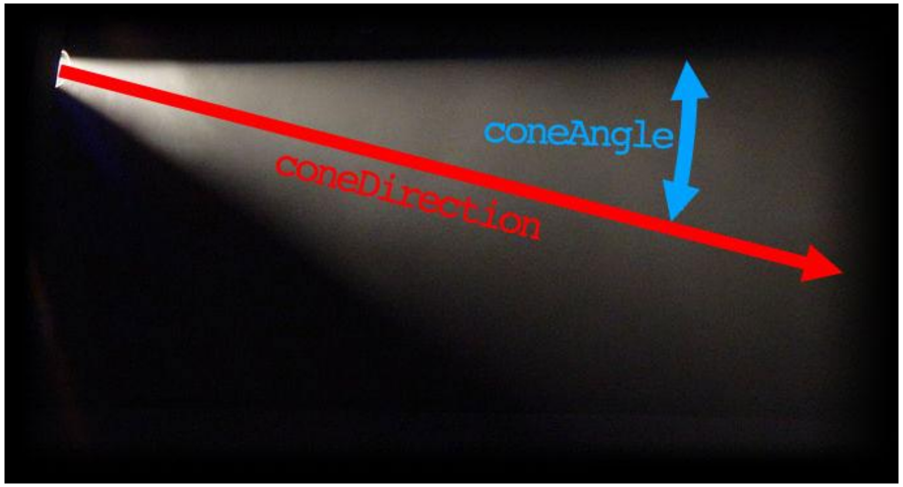
$$c_l = \frac{c_{src}}{\|\mathbf{p} - \mathbf{v}\|^2}$$

Light Attenuation

- Adding constant factor k to denominator for better control
- Quadratic attenuation: $k*(p-v)^2$
- Linear attenuation (HW2) : $k*(p-v)$
- Constant attenuation: k

Spotlights

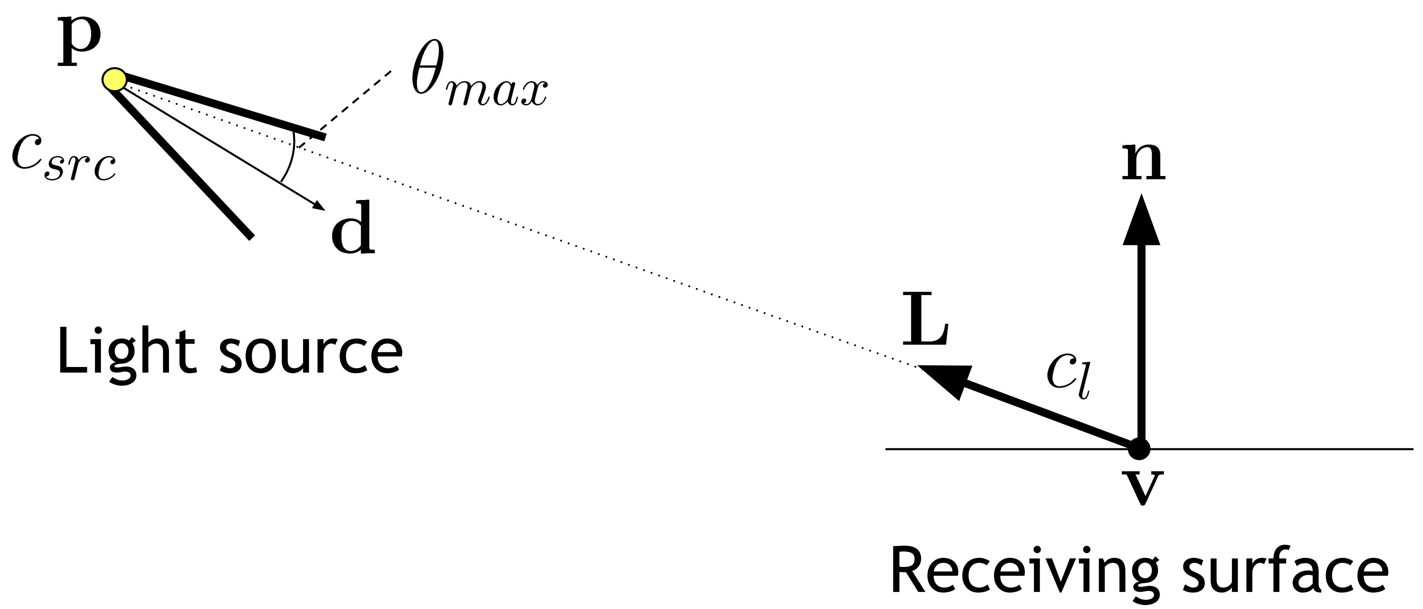
- Like point light, but intensity depends on direction



Parameters

- Position: location of light source
- Cone direction *d*: center axis of light source
- Intensity falloff:
  - Beam width (cone angle θ_{max})
  - The way the light tapers off at the edges of the beam

Spotlights



$$\mathbf{L} = \frac{\mathbf{p} - \mathbf{v}}{\|\mathbf{p} - \mathbf{v}\|}$$

$$c_l = \begin{cases} 0 & \text{if} \quad -\mathbf{L} \cdot \mathbf{d} \leq \cos(\theta_{max}) \\ c_{src}\,(-\mathbf{L} \cdot \mathbf{d})^f & \text{otherwise} \end{cases}$$

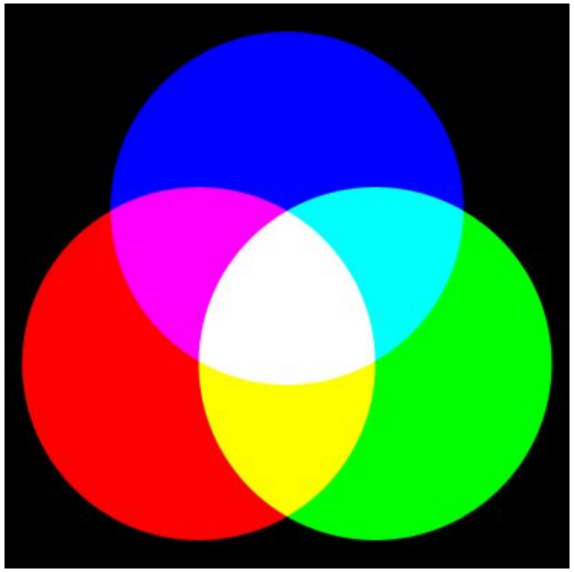
Vertex Shader

```
#version 150

// We don’t need to use textures for HW2 (but we will for HW3)
uniform mat4 camera;
uniform mat4 model;

in vec3 vert;
~~in vec2 vertTexCoord;~~
in vec3 vertNormal;

out vec3 fragVert;
~~out vec2 fragTexCoord;~~
out vec3 fragNormal;

void main()
{
~~    // Pass some variables to the fragment shader~~
~~    fragTexCoord = vertTexCoord;~~
    fragNormal = vertNormal;
    fragVert = vert;

    // Apply all matrix transformations to vert
    gl_Position = camera * model * vec4(vert, 1);
}
```

Source: Tom Dallling’s OpenGL Tutorial

Fragment Shader for Diffuse Reflection

```
#version 150

uniform mat4 model;
uniform sampler2D tex;

uniform struct Light
{
   vec4 position; // if w component=0 it’s directional
   vec3 intensities; // a.k.a the color of the light
   float attenuation; // only needed for point and spotlights
   float ambientCoefficient;
   float coneAngle;  // only needed for spotlights
   vec3 coneDirection; // only needed for spotlights
   float exponent;  // cosine exponent for how light tapers off
} light;

in vec2 fragTexCoord;
in vec3 fragNormal;
in vec3 fragVert;

out vec4 finalColor;
```



Source: Tom Dallling’s OpenGL Tutorial

Fragment Shader Part 2

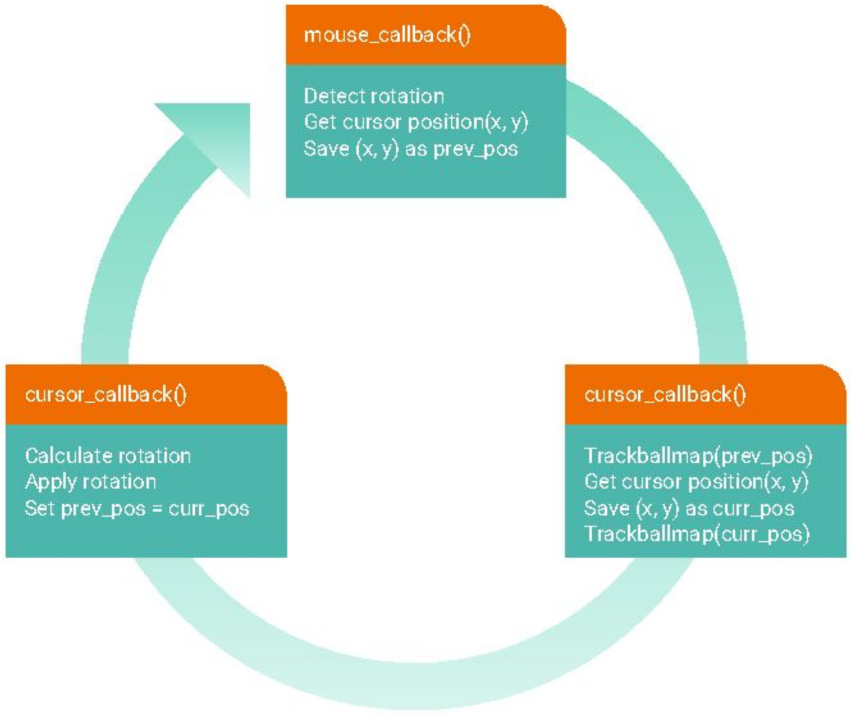
```
void main()
{
   // calculate normal in world coordinates
   mat3 normalMatrix = transpose(inverse(mat3(model)));
   vec3 normal = normalize(normalMatrix * fragNormal);

   // calculate the location of this fragment (pixel) in world coordinates
   vec3 fragPosition = vec3(model * vec4(fragVert, 1));

   // calculate the vector from this pixels surface to the light source
   vec3 surfaceToLight = light.position - fragPosition;

   // calculate the cosine of the angle of incidence
   float brightness = dot(normal, surfaceToLight) / (length(surfaceToLight) * length(normal));
   brightness = clamp(brightness, 0, 1);

   // calculate final color of the pixel, based on:
   // 1. The angle of incidence: brightness
   // 2. The color/intensities of the light: light.intensities
   // 3. The texture and texture coord: texture(tex, fragTexCoord)
   vec4 surfaceColor = texture(tex, fragTexCoord);
   finalColor = vec4(brightness * light.intensities * surfaceColor.rgb, surfaceColor.a);
}
```

Source: Tom Dallling's OpenGL Tutorial

Common Errors

- https://piazza.com/class/jmi7l0j71xg77u?cid=98
- Include shaders in working directory
- OBJ files are 1-indexed, OpenGL is 0-indexed
- Normal coloring needs to be moved from your C++ code to your shader code
- glm::length(x), not x.length()
- Initialize OBJObjects in Window::initialize_objects()
- Correct the axis of rotation for trackball if applying a series of rotations

Trackball mouse control: implementation details

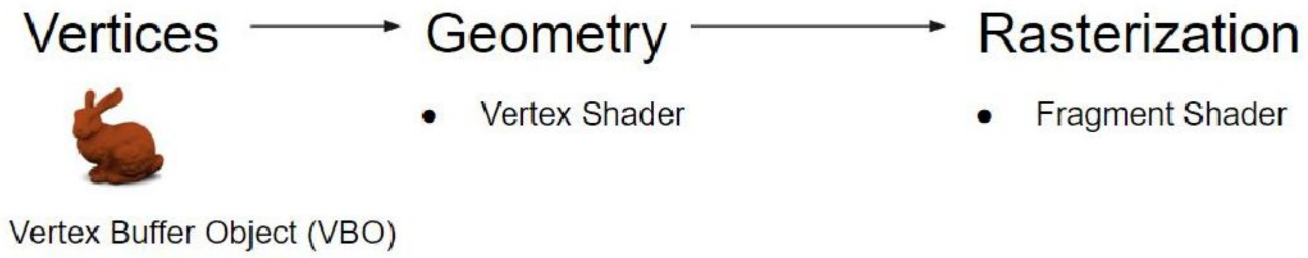


Trackball mouse control: implementation details

- http://www.glfw.org/docs/latest/input_guide.html#input_mouse
- Get mouse input
    - main.cpp: `glfwSetMouseButtonCallback(window, mouse_button_callback);`
    - window.cpp: define `void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)`
- Get cursor position
    - main.cpp: `glfwSetCursorPosCallback(window, cursor_pos_callback);`
    - window.cpp: define `static void cursor_position_callback(GLFWwindow* window, double xpos, double ypos)`

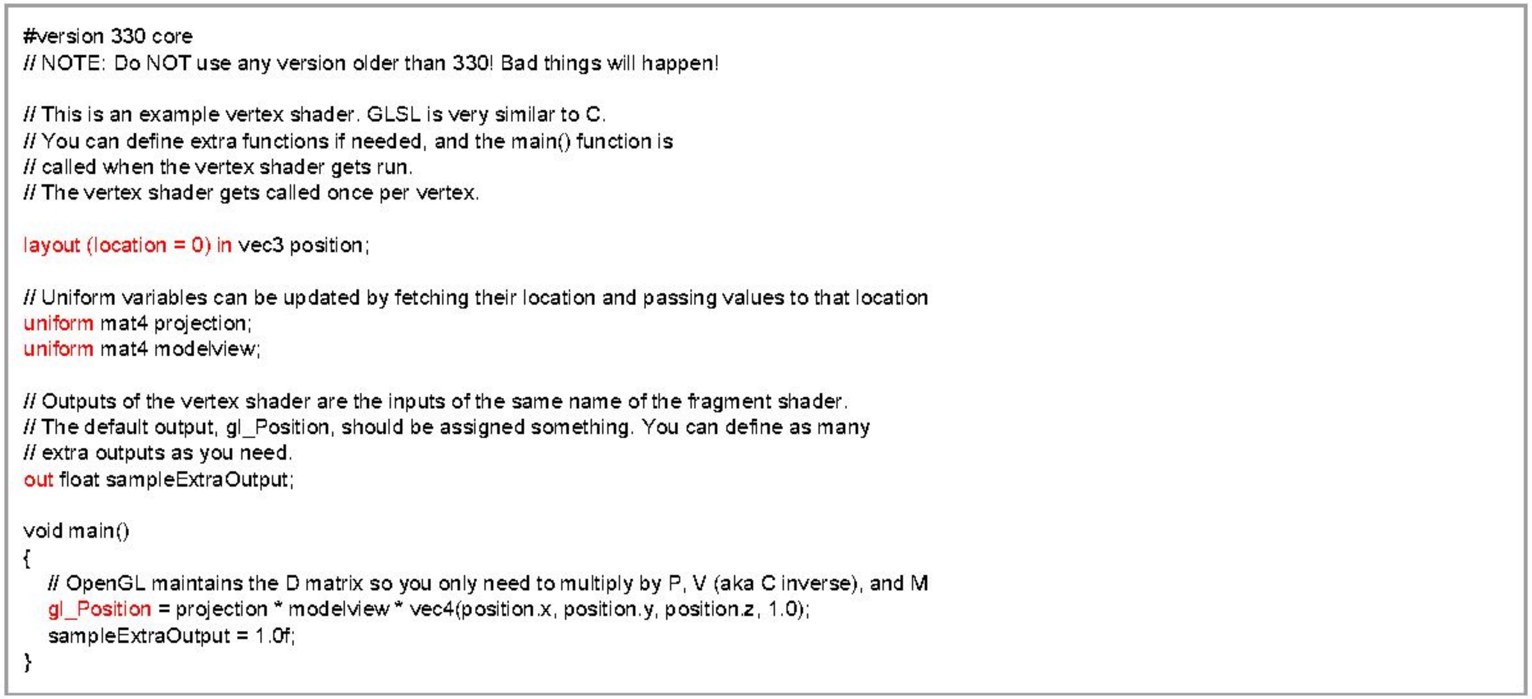
Modern OpenGL: shaders

- Modern OpenGL pipeline



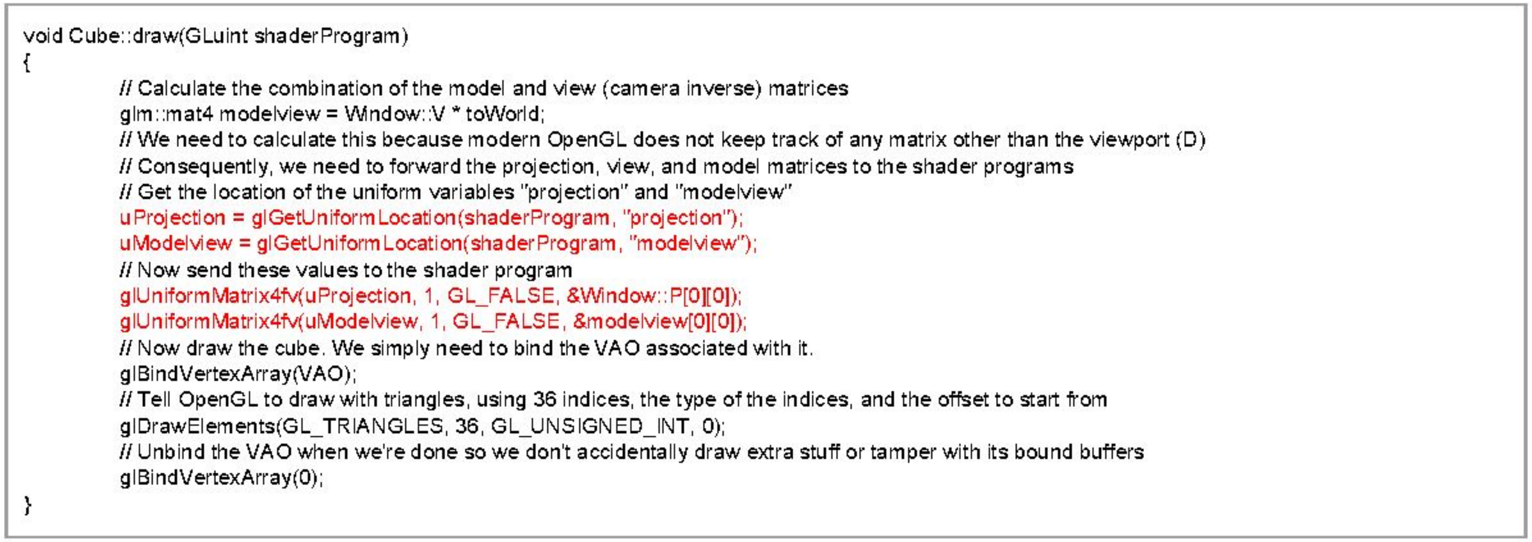
Modern OpenGL: shaders

- Vertex shader: handles processing of individual vertices



```
#version 330 core
// NOTE: Do NOT use any version older than 330! Bad things will happen!

// This is an example vertex shader. GLSL is very similar to C.
// You can define extra functions if needed, and the main() function is
// called when the vertex shader gets run.
// The vertex shader gets called once per vertex.

layout (location = 0) in vec3 position;

// Uniform variables can be updated by fetching their location and passing values to that location
uniform mat4 projection;
uniform mat4 modelview;

// Outputs of the vertex shader are the inputs of the same name of the fragment shader.
// The default output, gl_Position, should be assigned something. You can define as many
// extra outputs as you need.
out float sampleExtraOutput;

void main()
{
    // OpenGL maintains the D matrix so you only need to multiply by P, V (aka C inverse), and M
    gl_Position = projection * modelview * vec4(position.x, position.y, position.z, 1.0);
    sampleExtraOutput = 1.0f;
}
```

Modern OpenGL: shaders



```
void Cube::draw(GLuint shaderProgram)
{
            // Calculate the combination of the model and view (camera inverse) matrices
            glm::mat4 modelview = Window::V * toWorld;
            // We need to calculate this because modern OpenGL does not keep track of any matrix other than the viewport (D)
            // Consequently, we need to forward the projection, view, and model matrices to the shader programs
            // Get the location of the uniform variables "projection" and "modelview"
            uProjection = glGetUniformLocation(shaderProgram, "projection");
            uModelview = glGetUniformLocation(shaderProgram, "modelview");
            // Now send these values to the shader program
            glUniformMatrix4fv(uProjection, 1, GL_FALSE, &Window::P[0][0]);
            glUniformMatrix4fv(uModelview, 1, GL_FALSE, &modelview[0][0]);
            // Now draw the cube. We simply need to bind the VAO associated with it.
            glBindVertexArray(VAO);
            // Tell OpenGL to draw with triangles, using 36 indices, the type of the indices, and the offset to start from
            glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
            // Unbind the VAO when we're done so we don't accidentally draw extra stuff or tamper with its bound buffers
            glBindVertexArray(0);
}
```

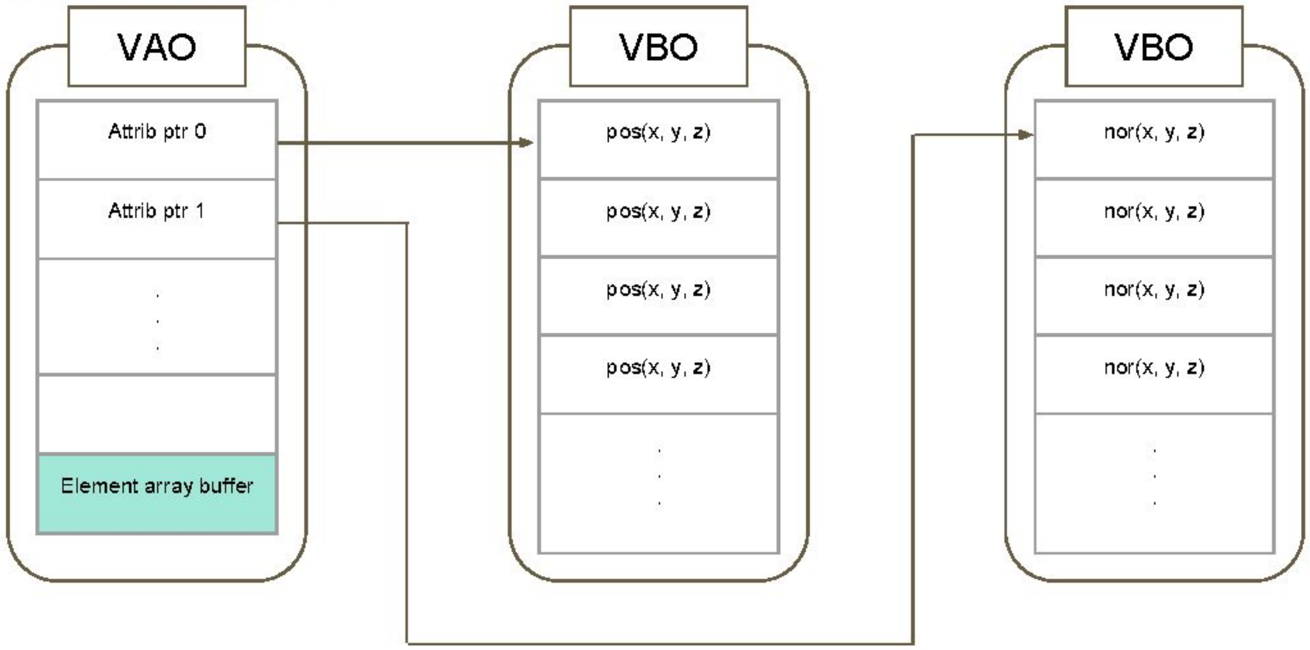
Modern OpenGL: shaders

- Fragment shader: handles processing of fragments created by rasterization

```
#version 330 core
// This is a sample fragment shader.

// Inputs to the fragment shader are the outputs of the same name from the vertex shader.
// Note that you do not have access to the vertex shader's default output, gl_Position.
in float sampleExtraOutput;

// You can output many things. The first vec4 type output determines the color of the fragment
out vec4 color;

void main()
{
    // Color everything a hot pink color. An alpha of 1.0f means it is not transparent.
    color = vec4(1.0f, 0.41f, 0.7f, sampleExtraOutput);
}
```

Modern OpenGL: VAO, VBO, EBO

Multiple VBOs, glm::vec3



Modern OpenGL: VAO, VBO, EBO

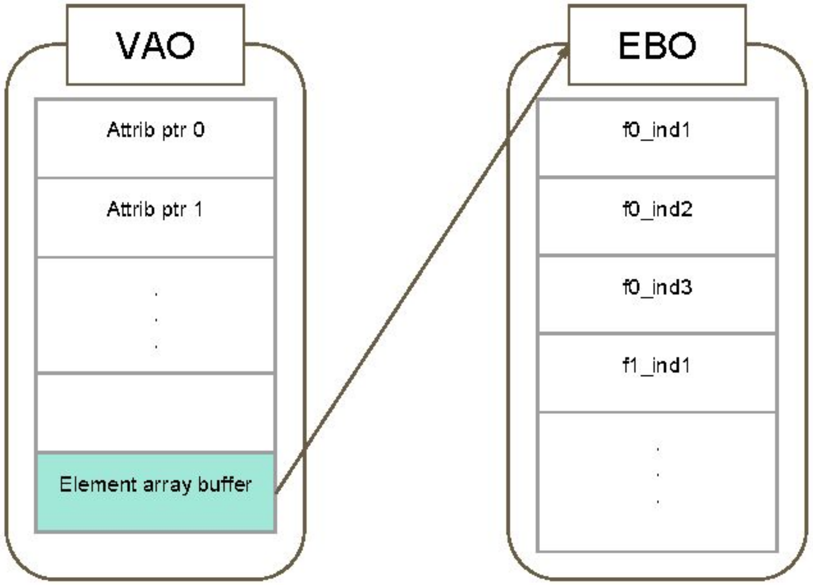
Multiple VBOs, glm::vec3

```
Cube::Cube()
{
        toWorld = glm::mat4(1.0f);

        glGenVertexArrays(1, &VAO);
        glGenBuffers(1, &VBO);
        // (1) Generate VBO for normals, e.g. VBO2
        glGenBuffers(1, &EBO);

        glBindVertexArray(VAO);

        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, ...);

        // (2) Bind VBO2
        // (3) Send buffer data of VBO2

        // (4) Enable vertex attribute array with position 1
        // (5) Define vertex attribute pointer for position 1

        ...
}
```

```
#version 330 core
// NOTE: Do NOT use any version older than 330! Bad things will happen!

layout (location = 0) in vec3 position;
// Create layout for location = 1

// Uniform variables can be updated by fetching their location and passing values
to that location
uniform mat4 projection;
uniform mat4 modelview;

// Outputs of the vertex shader are the inputs of the same name of the fragment
shader.
// The default output, gl_Position, should be assigned something. You can define
as many
// extra outputs as you need.
out float sampleExtraOutput;

void main()
{
   // OpenGL maintains the D matrix so you only need to multiply by P, V (aka C
inverse), and M
   gl_Position = projection * modelview * vec4(position.x, position.y, position.z,
1.0);
   sampleExtraOutput = 1.0f;
}
```

Modern OpenGL: VAO, VBO, EBO

EBO



```
Cube::Cube()
{
            toWorld = glm::mat4(1.0f);

            glGenVertexArrays(1, &VAO);
            glGenBuffers(1, &VBO);
            glGenBuffers(1, &EBO);

            glBindVertexArray(VAO);

            glBindBuffer(GL_ARRAY_BUFFER, VBO);
            glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

            glEnableVertexAttribArray(0);
            glVertexAttribPointer(...)

            glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
            glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);

            glBindBuffer(GL_ARRAY_BUFFER, 0);
            glBindVertexArray(0);

}
```

Lighting: Phong shading model, light

- How to determine c
- Directional light
  - Color
  - Direction
- Point light
  - Color
  - Position
  - Attenuation(linear)
- Spotlight
  - Color
  - Position
  - Direction
  - Cutoff, exponent
  - Attenuation(quadratic)

```
struct Light{
        int light_mode;

        vec3 light_color;
        vec3 light_pos;
        vec3 light_dir;

        float cons_att;
        float linear_att;
        float quad_att;

        float cutoff_angle;
        float exponent;
};
```

Lighting: Phong shading model, light

- How to determine c
- Directional light
    - Color
    - Direction
- Point light
    - Color
    - Position
    - Attenuation(linear)
- Spotlight
    - Color
    - Position
    - Direction
    - Cutoff, exponent
    - Attenuation(quadratic)

```
struct Light{
        int light_mode;

        vec3 light_color;
        vec3 light_pos;
        vec3 light_dir;

        float cons_att;
        float linear_att;
        float quad_att;

        float cutoff_angle;
        float exponent;
};
```

Lighting: Phong shading model, material

- The color at a point is: $c_d + c_s + c_a$
- Diffuse
  - cd = cl · kd(n · L)
- Specular
  - cs = cl · ks(R · e)^p
- Ambient
  - ca = cl · ka

```
struct Material{
        int object_mode;

        vec3 color_diff;
        vec3 color_spec;
        vec3 color_ambi;
};
```

- What is the difference between c and L?
- What is the datatype of each constant and term?
- What should be the datatype of color that is the output of fragment shader?
- How do we determine k’s and c? Reference link

Lighting: Phong shading model, light+model

(1) Calculate c_l
(2) Calculate c_d, c_s, c_a
(3) Multiply attenuation to each term(for point and spot lights only)
(4) Add the three terms
(5) Pass as glm::vec4!!!

Lighting: implementation tips

- Make light and material classes for cleaner coding
  - Red is in Light class, blue is in fragment shader, green is in Window.cpp

```
class Light{
        int mode;
        …

        void draw();
        ...

};
```

```
void Light::draw()
{
        …
        light_mode = glGetUniformLocation(program,
        "Light.light_mode");

        glUniform1i(light_mode, mode);
        …
}
```

```
struct Light{
        int light_mode;

        vec3 light_color;
        vec3 light_pos;
        vec3 light_dir;

        float cons_att;
        float linear_att;
        float quad_att;

        float cutoff_angle;
        float exponent;
};
```

```
…
Light* light_ptr;
Light dir_light(0);
Light point_light(1);
…
```

```
// How are you going to pass the shader program to
// be used?

light_ptr->draw();
```

Lighting: implementation tips

- Transform lights just like you transformed your objects
    - For example, when you want to rotate directional light, you just need to rotate the directional light’s direction and assign the new direction to that light.
- Pass lights’ information to the shader just like you passed your objects’ information
    - Treat lights as objects
    - Since the lights don’t need to be “lighted” and will not be lighted because of the normals, you can set the ambient color for the light models to be something high

Lighting: implementation tips

- Vertex and gl_Position
  - Vertex = vec3(model * vec4(position, 1.0f))
  - gl_Position = projection * view * model * vec4(position, 1.0f);
  - What is the difference between Vertex and gl_Position?
  - Which variable should you pass to fragment shader to calculate the color?
- Normal
  - What is the problem if you just pass normal values as they are?
  - How do you pass “correct” normal values to fragment shader?
  - Solution: Normal = mat3(transpose(inverse(model))) * normal;
  - Why? Chalkboard time! reference