OPENGL

Workshop

Oscar

Agenda

1. Shaders and the rendering pipeline
2. Translation, Rotation, Scale and Uniform variables
3. Interpolation, Indexed Draws and Projections
4. Phong Lighting

Shaders and the rendering pipeline

What is the rendering pipeline?

Series of stages to render an image.
Programmable with GLSL (OpenGL Shading language).

1. Vertex Specification
2. Vertex Shader (programmable)
3. Tessellation (programmable)
4. Geometry Shader (programmable)
5. Vertex Post-Processing
6. Primitive Assembly
7. Rasterization
8. Fragment Shader (programmable)
9. Per-Sample Operations

Vertex Specification

Stage to set up the data of the vertices for the primitives we want to render.

Vertex: Coordinate (x,y,z).

Primitive: Simple shape with 1 or more vertices.

Vertex Specification

Uses VAOs (Vertex Array Objects), VBOs (Vertex Buffer Objects) and Attribute Pointers.

VAO defines WHAT data a vertex has (position, colour,...)

VBO defines the data itself.

Attribute Pointers define where and how shaders can access vertex data.

Vertex Specification

To create a VAO and VBO:

1. Generate a VAO ID
2. Bind the VAO with that ID
3. Generate a VBO ID
4. Bind the VBO with that ID (now you’re working on the chosen VBO attached to the chosen VAO)
5. Attach the vertex data to that VBO
6. Define the Attribute Pointer formatting
7. Enable the Attribute Pointer
8. Unbind the VAO and VBO, ready for the next object to be bound.

Vertex Specification

Initiating Draw:

1. Activate Shader Program you want to use.
2. Bind VAO of object you want to draw.
3. Call glDrawArrays.

Vertex Shader

Handles vertices individually.

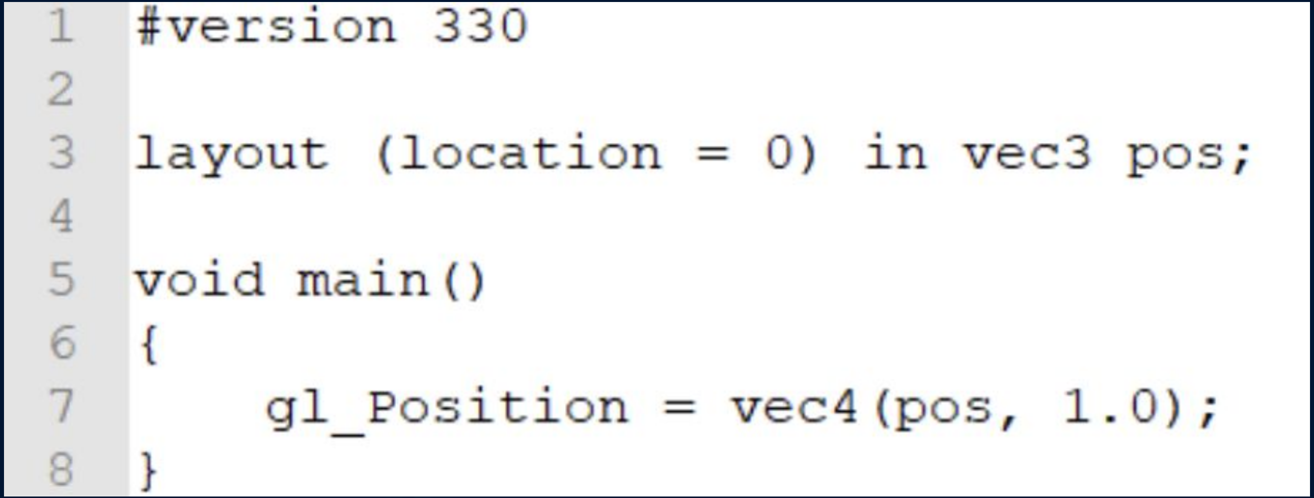
Must store something in gl_Position as it is used by later stages.

Can specify additional outputs.

Inputs consist of the vertex data itself.

Vertex Shader

Example



```
#version 330

layout (location = 0) in vec3 pos;

void main()
{
    gl_Position = vec4(pos, 1.0);
}
```

Tessellation

Divide up data in to smaller primitives.

Geometry Shader

Handles primitives (groups of vertices).

Vertex Post-Processing

Clipping: Primitives that won’t be visible are removed

Positions converted from “clip-space” to “window space”

Primitive Assembly

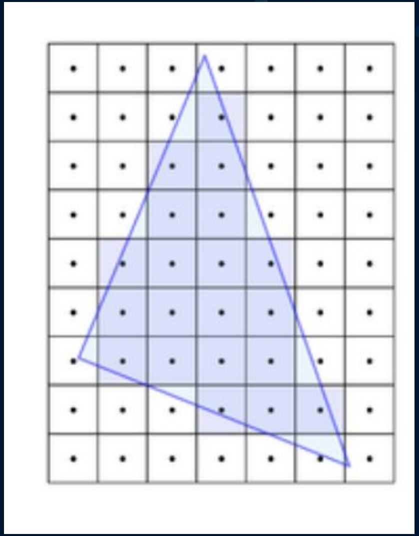
Vertices are converted into a series of primitives (ex: Triangles).

Face culling is the removal of primitives that can’t be seen because they are facing “away”

Rasterization

Converts primitives in to fragments.

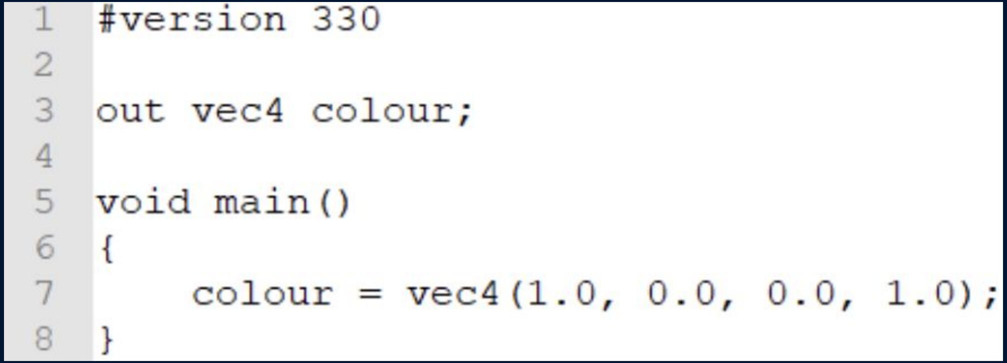
Fragments are pieces of data for each pixel.



Fragment Shader

Handles data for each fragment.

Most important output is the colour.



```
#version 330

out vec4 colour;

void main()
{
	colour = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Per-Sample Operations

Series of tests run to see if the fragment should be drawn (Depth test, Colour Blending, …).

Fragment data written to currently bound Framebuffer

Create a Shader Program

1. Create empty program.
2. Create empty shaders.
3. Attach shader source code to shaders.
4. Compile shaders.
5. Attach shaders to program.
6. Link program (creates executables from shaders and links them together).
7. Validate program.

Using a Shader Program

When you create a shader, an ID is given.

Call glUseProgram(shaderID).

Translation, Rotation, Scale and Uniform variables

Matrix multiplication

Multiplying a vector by a matrix

Identity matrix

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 5 \\ 1 \\ 8 \end{bmatrix} = \begin{bmatrix} 4 \\ 47 \\ 5 \\ 68 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Translation

Moves a vector

Position: x, y, z

Movement: X, Y, Z

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + X \\ y + Y \\ z + Z \\ 1 \end{bmatrix}$$

Scale

Resizes a vector

Position: x, y, z

Resize: SX, SY, SZ

$$\begin{bmatrix} SX & 0 & 0 & 0 \\ 0 & SY & 0 & 0 \\ 0 & 0 & SZ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} SX \cdot x \\ SY \cdot y \\ SZ \cdot z \\ 1 \end{bmatrix}$$

Rotation

Rotates a vector on a specific axis.

X rotation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{bmatrix}$$

Y rotation:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{bmatrix}$$

Z rotation:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{bmatrix}$$

Uniform variables

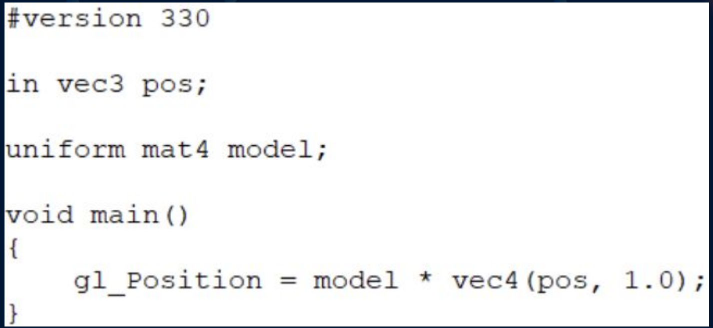
Type of variable in shader.

Is global.

Has location ID

Need to find the location so we can bind a value to it.

glUniform1f(location, 3.5f);



```
#version 330

in vec3 pos;

uniform mat4 model;

void main()
{
    gl_Position = model * vec4(pos, 1.0);
}
```

Uniform variables

glUniform1f – Single floating value.

glUniform1i – Single integer value.

glUniform4f – vec4 of floating values.

glUniform4fv – vec4 of floating values, value specified by pointer.

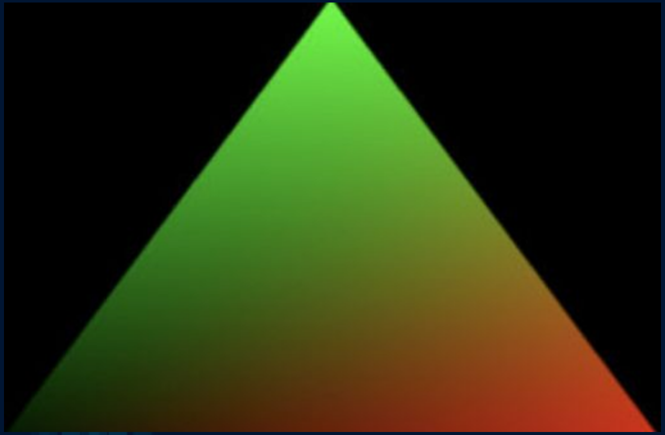
glUniformMatrix4fv – mat4 of floating values, value specified by pointer.

Interpolation, Indexed Draws and Projections

Interpolation

Per-vertex attributes passed on are “interpolated” using the other values on the primitive (weighted average).

Fragment Shader picks up the interpolated value.



Indexed Draws

Define vertices to draw a cube.

Cube will consist of 12 triangles (two for each face).

12 x 3 vertices per triangle = 36 vertices.

But a cube only has 8 vertices!

Indexed Draws

Define 8 vertices with indexes 1 to 8 to refer them.

Use glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, IBO);

Better to use 3D modeling software.

Projections

2D or 3D look.

Used to convert from “View Space” to “Clip Space”.

Coordinate systems:

1. Local Space: Raw position
2. World Space: Position relative to origin
3. View Space: Position relative to camera and orientation
4. Clip Space: Only what camera sees
5. Screen Space: Image in coordinate system of the window

Projections

Model Matrix * Local Space = World Space

View Matrix * World Space = View Space

Projection Matrix * View Space = Clip Space

Projections

To create Clip Space we define an area (frustum) of what is not to be clipped.

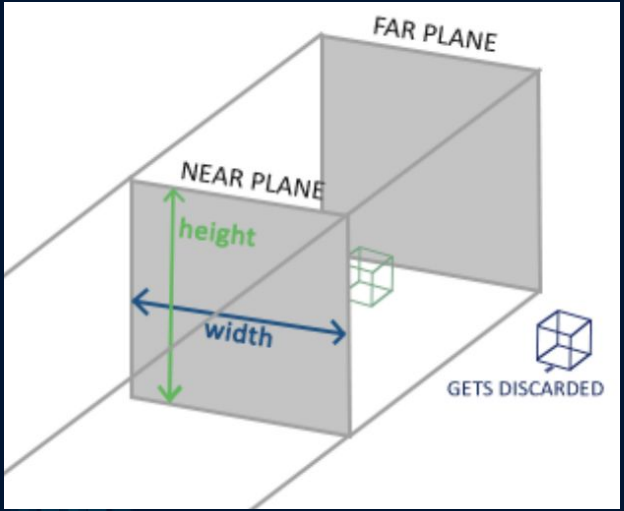
Types:

- Orthographic
- Perspective

Projections

Orthographic has cuboid frustum.

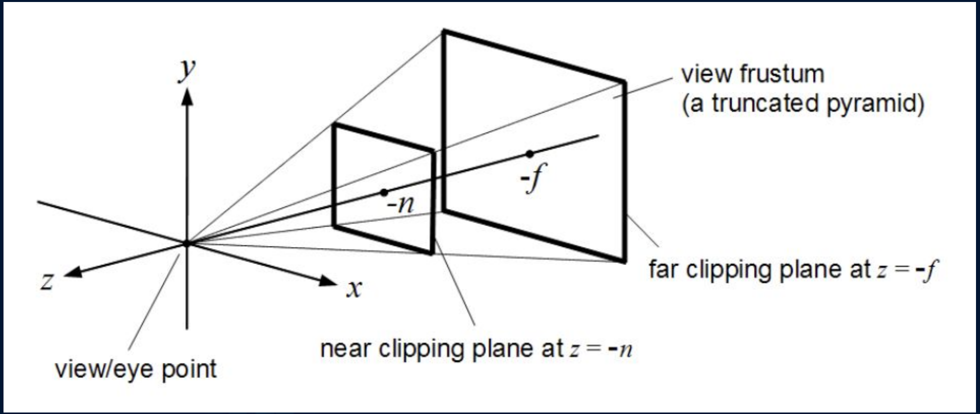
No difference close/far.



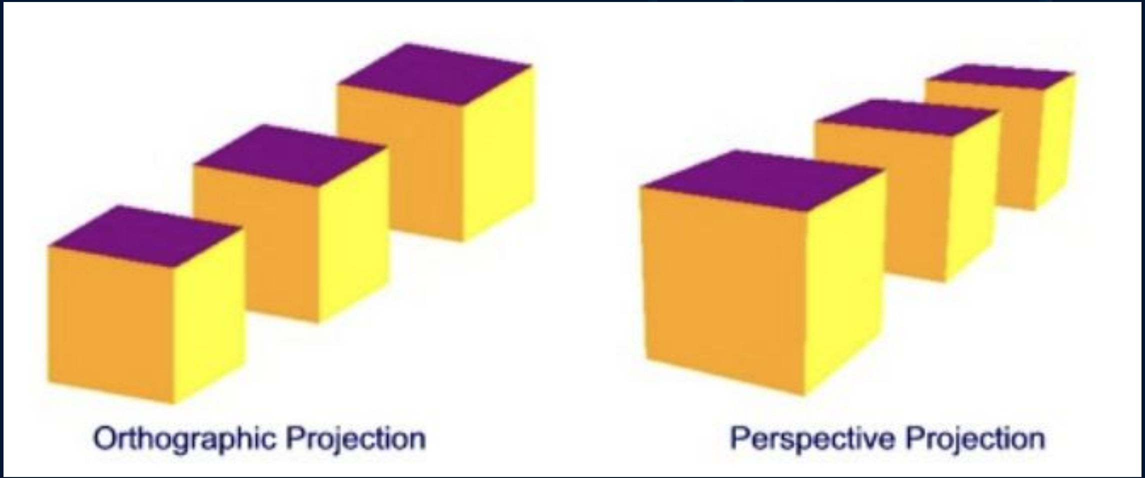
Projections

Perspective has a truncated pyramid frustum.

Illusion of depth.



Projections



Projections

glm::mat4 proj = glm::perspective(fov, aspect, near, far);

- fov = field-of-view, the angle of the frustum.
- aspect = aspect ratio of the viewport (usually its width
- divided by its height).
- near = distance of the near plane.
- far = distance of the far plane.

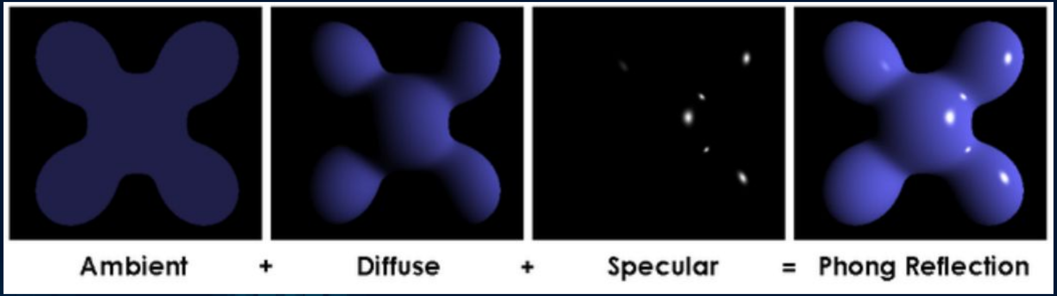
gl_Position = projection * view * model * vec4(pos, 1.0);

Phong Lighting

Phong Lighting

Simulates light

- Ambient Lighting: Light that is always present
- Diffuse Lighting: Light determined by direction of light source.
- Specular Lighting: Light reflected from the source to the viewer’s eye.



Ambient Lighting

Simulates light bouncing off other objects.

It needs a factor

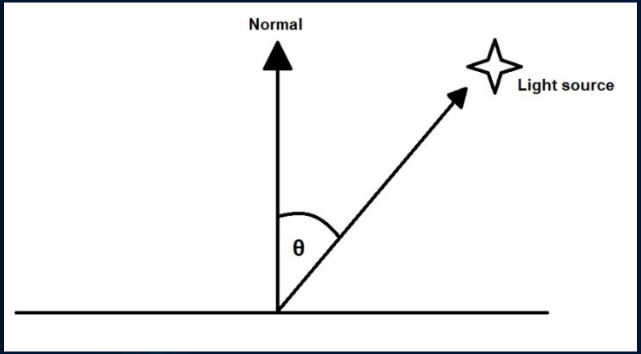
```
Ambient = lightColour * ambientStrength;


fragColour = objectColour * Ambient;
```

Diffuse Lighting

Simulates the drop-off of lighting based on the angle of lighting.

Angle between normal and light source.



Diffuse Lighting

Dot Product -> $v1 \cdot v2 = |v1| \times |v2| \times \cos(\theta)$

If normalized, $|v1| = |v2| = 1$

So, $v1 \cdot v2 = \cos(\theta)$

cos(0 degrees) = 1, and cos(90 degrees) = 0

Diffuse factor = $v1 \cdot v2$

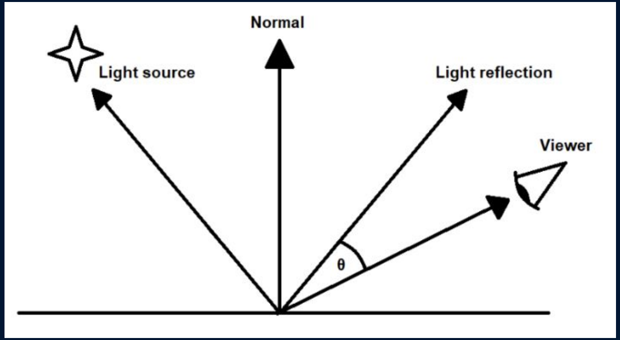
```
fragColour = objectColour * (ambient + diffuse);
```

Specular Lighting

It is the direct reflection of the light source hitting the viewer's eye.

Angle between view vector and light reflection.

- Light vector
- Normal vector
- Reflection vector (around normal)
- View vector



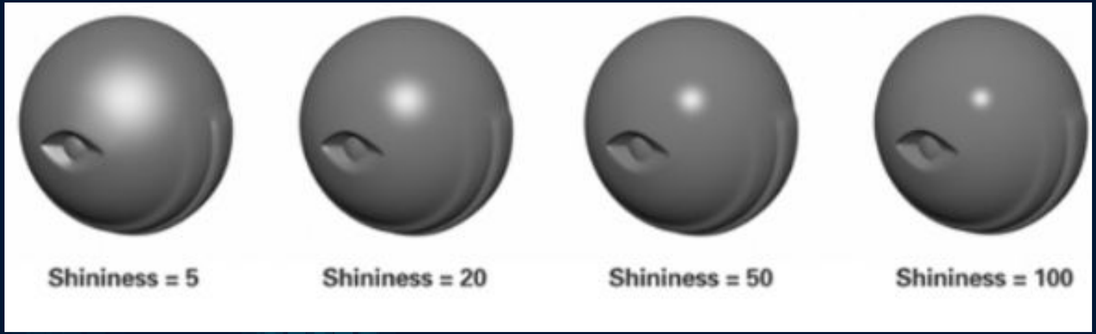
Specular Lighting

Calculate specular factor with dot product.

Shininess for better reflections.

specular = (view · reflection)^shininess

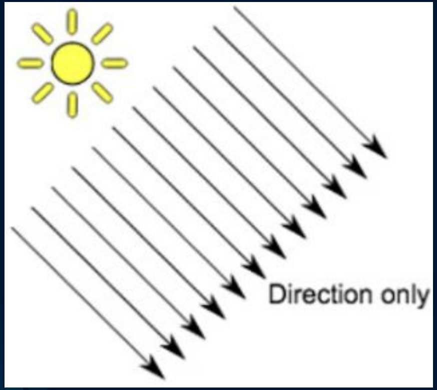
fragColour = objectColour * (ambient + diffuse + specular);



Directional Light

Simplest type of light (Ex: Sun).

Requires basic information (colour, ambient, diffuse, specular…) and a direction.



Thanks for coming!