CSE 167 Fall 2019

Discussion 2

Project 2

- Project specifications HERE
- DUE Friday Oct 18 2pm
  - CSE Basement 260/270
- Features to implement:
  - Display OBJ file as 3D model
  - Mouse Control
    - Rotating and scaling the model
  - Add light sources
  - Add material properties to models

Overview

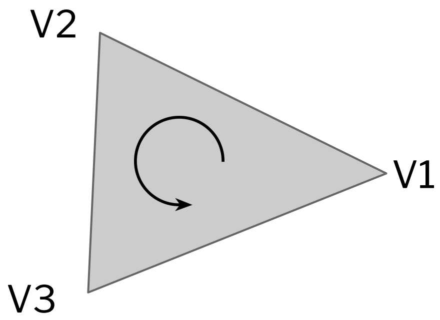
- Model Loader
  - OpenGL
- Linear Algebra
  - Homogeneous coordinates
  - Vertex transformations
    - Matrix multiplication
    - Orbit vs. spin
- Mouse Control

Model Loader

- Previously visualized using GL_POINTS
- Switch to GL_TRIANGLES
  - From the OBJ file need:
    - Vertices → store in std::vector<glm::vec3>
    - Normals → store in std::vector<glm::vec3>
    - Face indices → store in std::vector<unsigned int>
  - OpenGL needs indices of vertices in CCW order to be able to draw a face
    - CCW ordering is important so OpenGL knows the “front” of the face

OpenGL

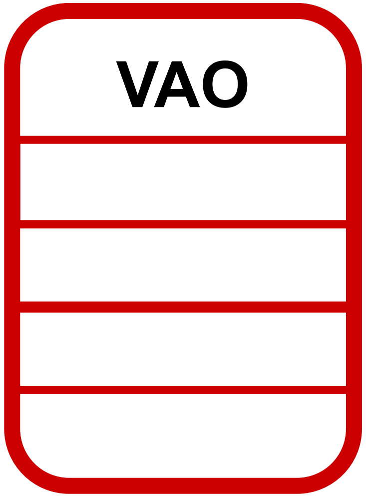
Model Loader



- Extend Parser to parse:
  - Vertex Normals
  - Faces:
    - `f 514//514 465//465 464//464`
    - `f v1//vn1 v2//vn2 v3//vn3`
  - NOTE:
    - In OBJ file indices start at 1 **NOT** 0
    - Indices are stored as unsigned ints **NOT** glm::vec3
      - Note: can also use glm::ivec3 instead of unsigned ints

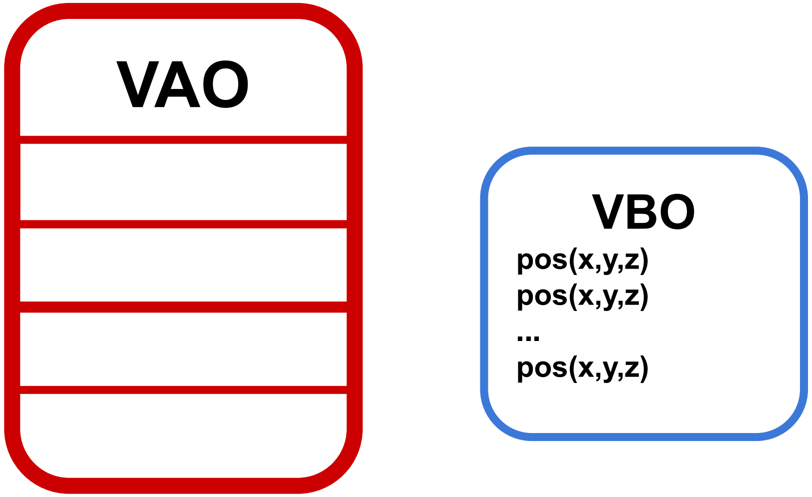
OpenGL

- VAO, VBO(s), & EBO:
  - VAO = Vertex Attribute Array
    - container for buffers
    - ie. VBO(s) and EBO
  - VBO = Vertex Buffer Object
    - hold model information
    - ie. Positions, Normals..
  - EBO = Element Buffer Object
    - hold index information
    - ie. indices for the faces



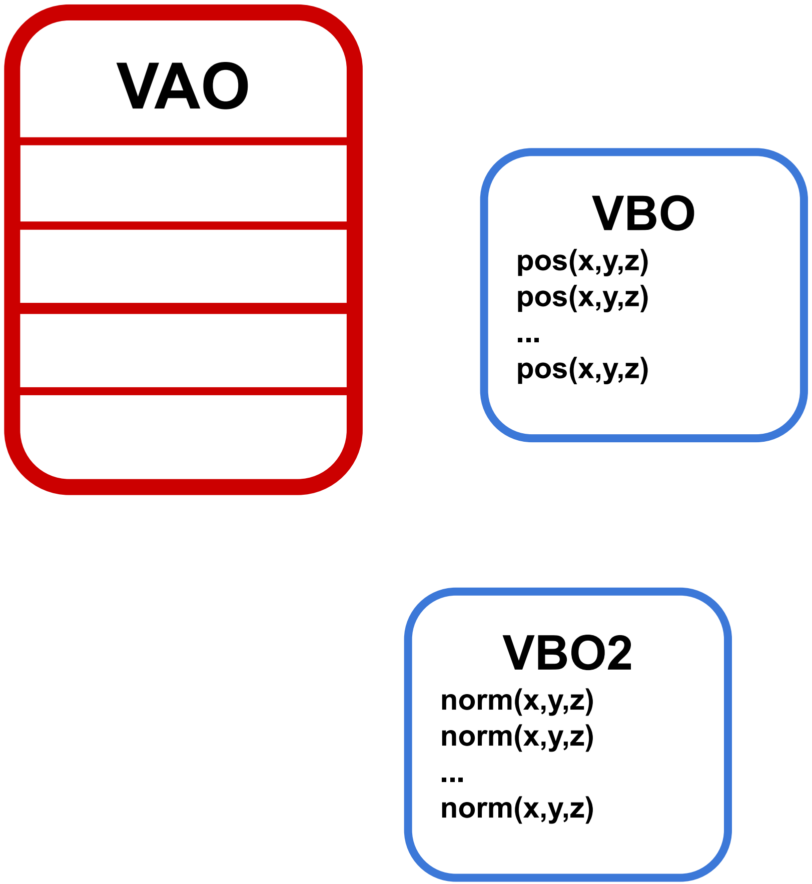
OpenGL

- VAO, VBO(s), & EBO:
  - VAO = Vertex Attribute Array
    - container for buffers
    - ie. VBO(s) and EBO
  - VBO = Vertex Buffer Object
    - hold model information
    - ie. Positions, Normals..
  - EBO = Element Buffer Object
    - hold index information
    - ie. indices for the faces



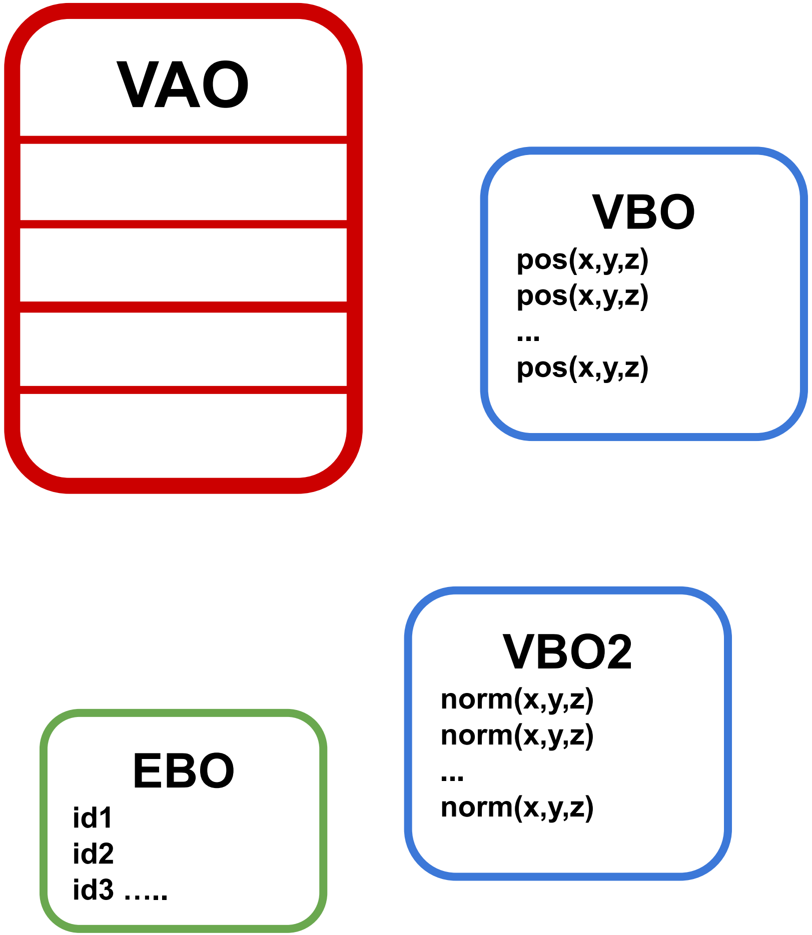
OpenGL

- VAO, VBO(s), & EBO:
  - VAO = Vertex Attribute Array
    - container for buffers
    - ie. VBO(s) and EBO
  - VBO = Vertex Buffer Object
    - hold model information
    - ie. Positions, Normals..
  - EBO = Element Buffer Object
    - hold index information
    - ie. indices for the faces



OpenGL

- VAO, VBO(s), & EBO:
  - VAO = Vertex Attribute Array
    - container for buffers
    - ie. VBO(s) and EBO
  - VBO = Vertex Buffer Object
    - hold model information
    - ie. Positions, Normals..
  - EBO = Element Buffer Object
    - hold index information
    - ie. indices for the faces



OpenGL

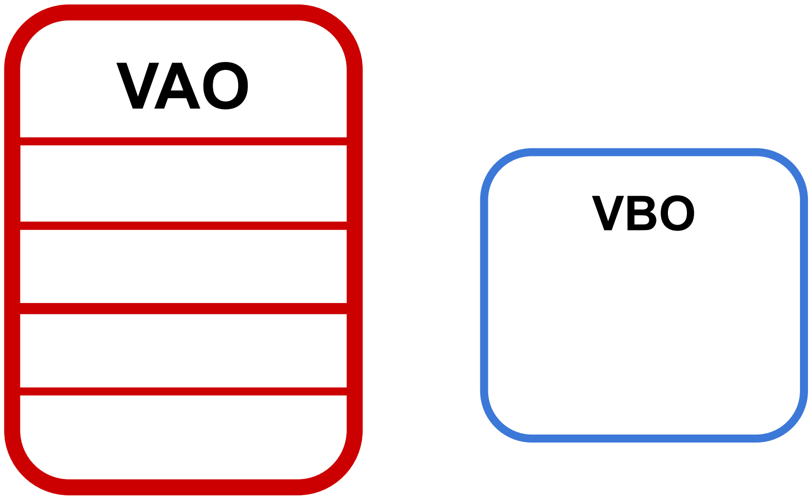
- Generate VAO and VBO(s)
- Bind VAO
  - Binding VAO "activates" it
  - Binding VBO after "attaches" it to the bound VAO

```
// Generate VAO and VBOs
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

// Bind VAO
glBindVertexArray(VAO);
```

OpenGL

- Generate VAO and VBO(s)
- Bind VAO
  - Binding VAO “activates” it
  - Binding VBO after “attaches” it to the bound VAO

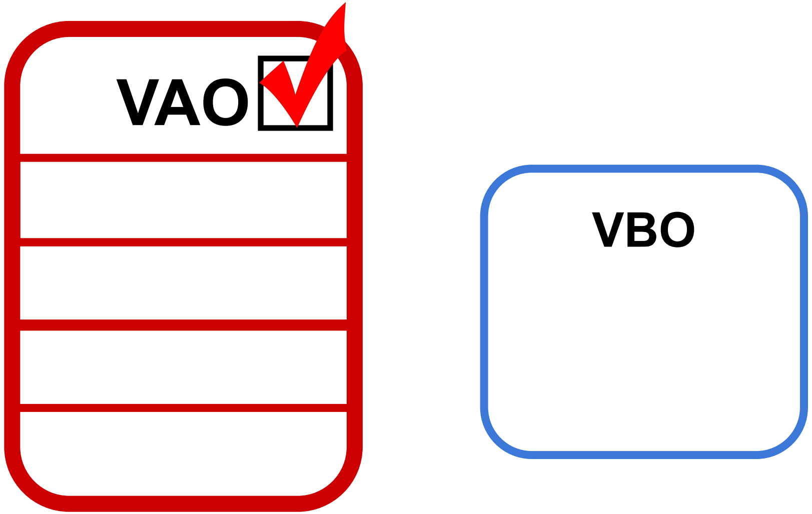
```
// Generate VAO and VBOs
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

// Bind VAO
glBindVertexArray(VAO);
```



OpenGL

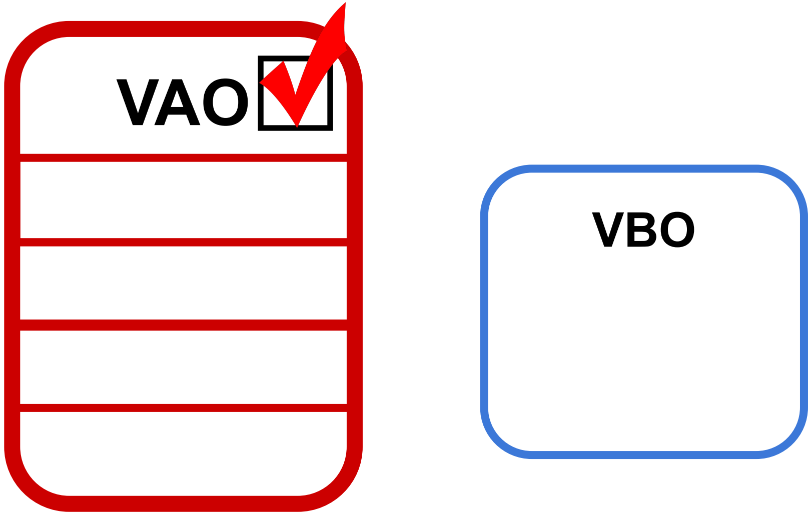
- Generate VAO and VBO(s)
- Bind VAO
  - Binding VAO “activates” it
  - Binding VBO after “attaches” it to the bound VAO

```
// Generate VAO and VBOs
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

// Bind VAO
glBindVertexArray(VAO);
```



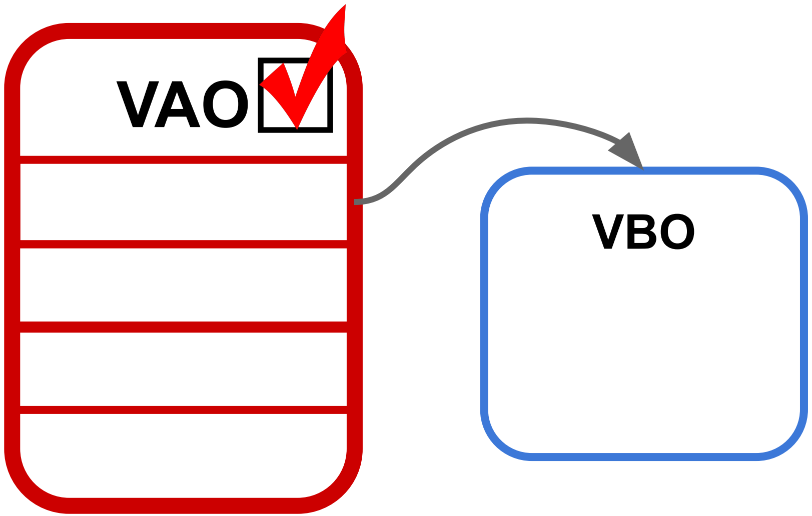
OpenGL

- For each VBO:
  a. Bind buffer to VAO
  b. Populate data in VBO
  c. Create channel between VBO and shader
  d. Tell shader how to read VBO




```
// Bind VBO to the bound VAO, and send the data
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3) * points.size(), points.data(), GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);
```

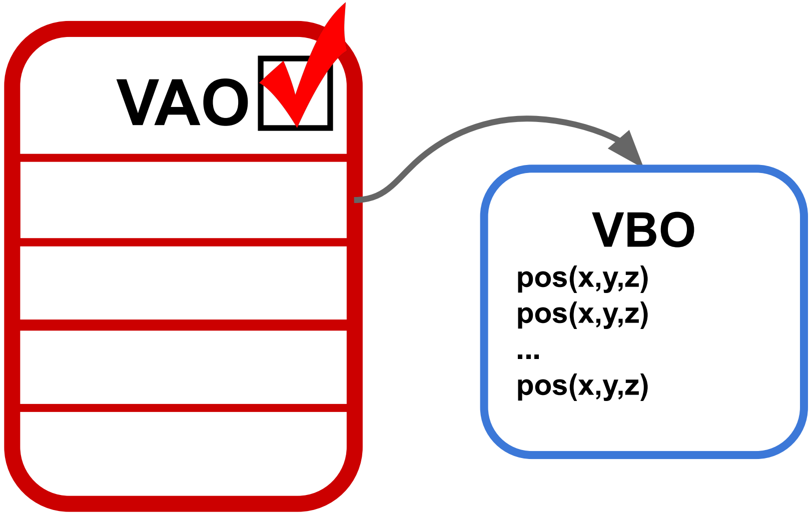
OpenGL



- For each VBO:
  a. Bind buffer to VAO
  b. Populate data in VBO
  c. Create channel between VBO and shader
  d. Tell shader how to read VBO

```
// Bind VBO to the bound VAO, and send the data
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3) * points.size(), points.data(), GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);
```

OpenGL

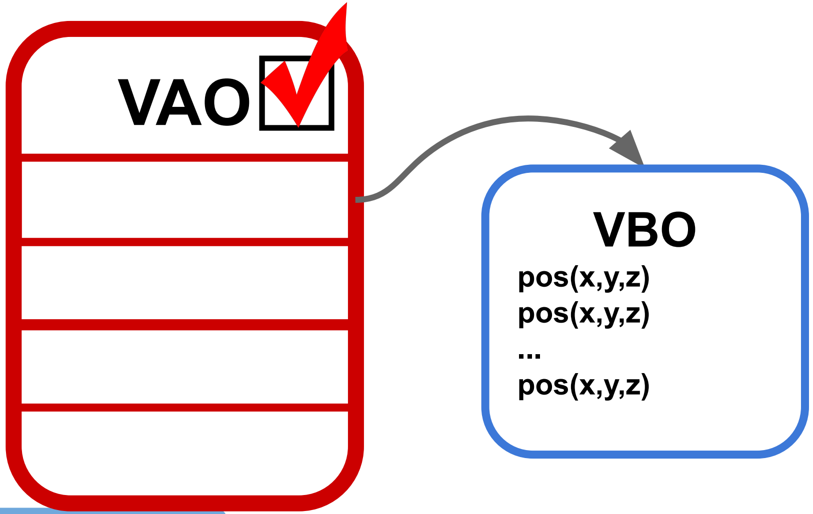


- For each VBO:
  a. Bind buffer to VAO
  b. Populate data in VBO
  c. Create channel between VBO and shader
  d. Tell shader how to read VBO

```
// Bind VBO to the bound VAO, and send the data
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3) * points.size(), points.data(), GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);
```

OpenGL

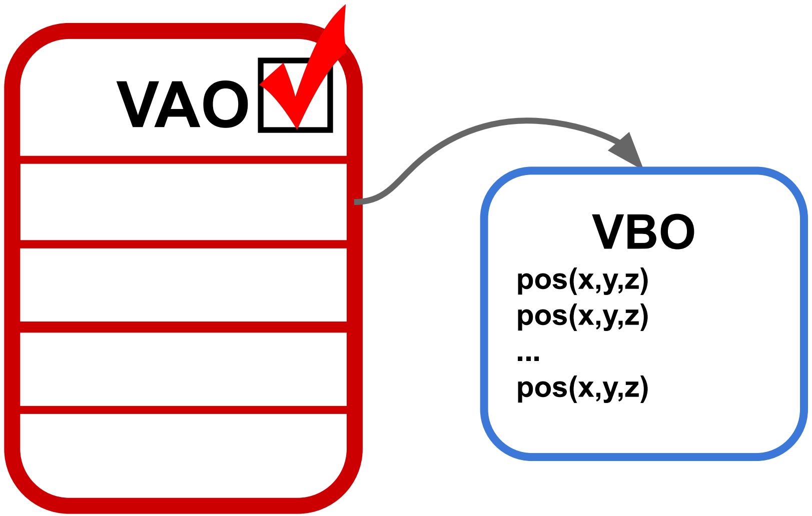
- For each VBO:
    a. Bind buffer to VAO
    b. Populate data in VBO
    c. Create channel between VBO and shader
    d. Tell shader how to read VBO




```
// Bind VBO to the bound VAO, and send the data
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3) * points.size(), points.data(), GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);
```

OpenGL

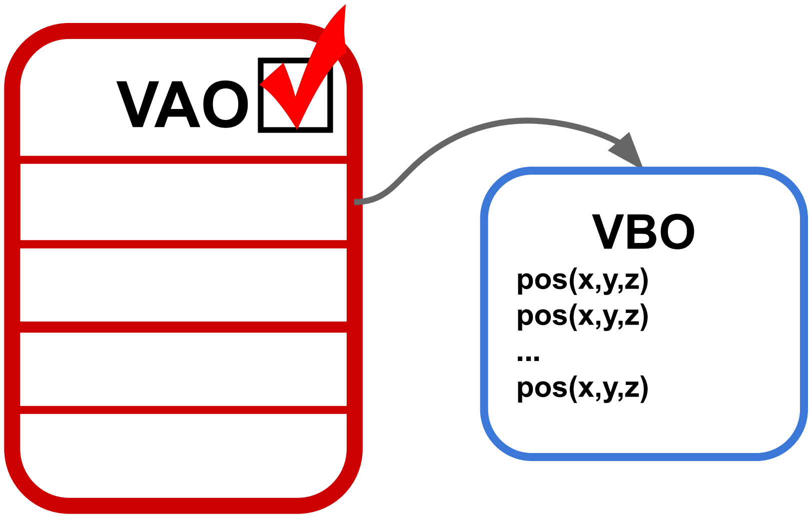
- For each VBO:
  a. Bind buffer to VAO
  b. Populate data in VBO
  c. Create channel between VBO and shader
  d. Tell shader how to read VBO




```
// Bind VBO to the bound VAO, and send the data
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3) * points.size(), points.data(), GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);
```

OpenGL

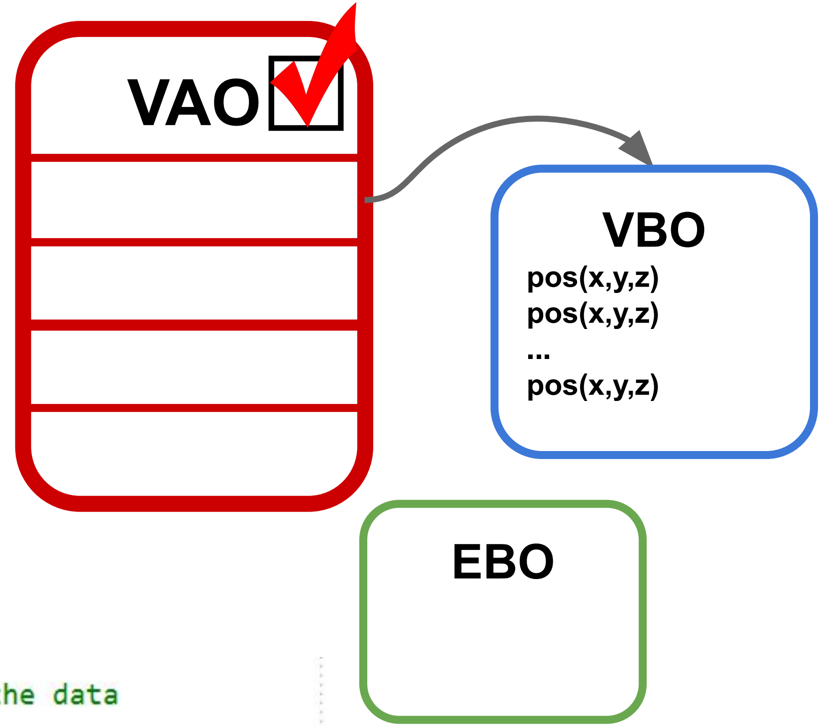
- For EBO:
  a. Generate EBO
  b. Bind buffer
  c. Populate data in EBO




```
// Generate EBO, bind the EBO to the bound VAO and send the data
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(glm::ivec3) * indices.size(), indices.data(), GL_STATIC_DRAW);
```

OpenGL

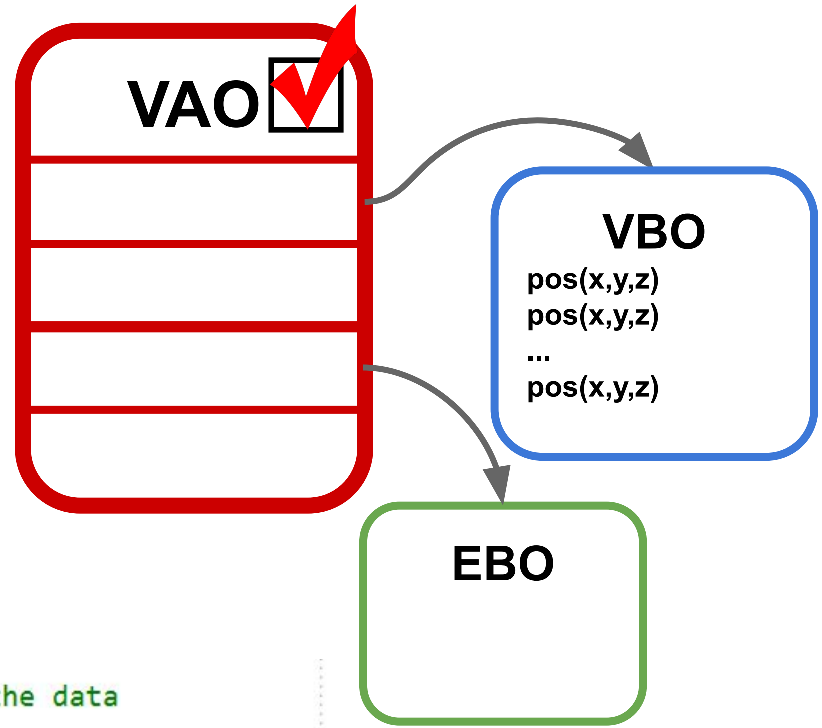
- For EBO:
  a. Generate EBO
  b. Bind buffer
  c. Populate data in EBO




```
// Generate EBO, bind the EBO to the bound VAO and send the data
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(glm::ivec3) * indices.size(), indices.data(), GL_STATIC_DRAW);
```

OpenGL

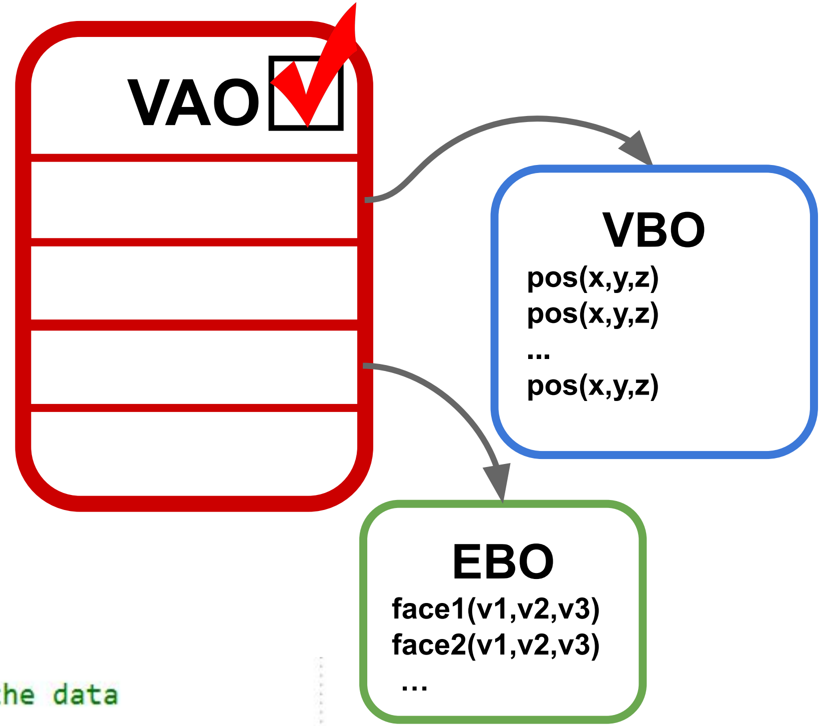
- For EBO:
  a. Generate EBO
  b. Bind buffer
  c. Populate data in EBO




```
// Generate EBO, bind the EBO to the bound VAO and send the data
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(glm::ivec3) * indices.size(), indices.data(), GL_STATIC_DRAW);
```

OpenGL

- For EBO:
  a. Generate EBO
  b. Bind buffer
  c. Populate data in EBO




```
// Generate EBO, bind the EBO to the bound VAO and send the data
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(glm::ivec3) * indices.size(), indices.data(), GL_STATIC_DRAW);
```

OpenGL

- To Draw ***POINT CLOUD***:
  - Bind VAO for the object you want to draw
  - Set the point size
  - glDrawArrays(...)
  - Unbind VAO

```
void PointCloud::draw()
{
    // Bind to the VAO.
    glBindVertexArray(VAO);
    // Set point size.
    glPointSize(pointSize);
    // Draw points
    glDrawArrays(GL_POINTS, 0, points.size());
    // Unbind from the VAO.
    glBindVertexArray(0);
}
```

OpenGL

- To Draw ***SOLID***:
  - Bind VAO for the object you want to draw
  - glDraw***Elements***(GL_TRIANGLES, ...)
  - Unbind VAO

```
void Cube::draw()
{
    // Bind the VAO.
    glBindVertexArray(VAO);
    // draw the points using triangles, indexed with the EBO
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
    // Unbind the VAO
    glBindVertexArray(0);
}
```

OpenGL

- Connecting to the shader:
  - In C++:
    - glEnableVertexAttribArray(**0**)
  - Matches with shader.vert:
    - layout (location = **0**)

```
// Bind VBO to the bound VAO, and send the dat
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE
```

```
#version 330 core
layout (location = 0) in vec3 position;

// Uniform variables
uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
```

Linear Algebra

Linear Algebra

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Homogeneous coordinates
  - Add 1 in additional dimension
  - Done so can combine together rotations/scales and translations simply
  - 3D manipulation can now all be done with 4x4 matrices
- Matrix Multiplication
  - ORDER MATTERS!

Linear Algebra

- EX: translate then scale vs scale then translate

$$S = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad a = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

$$a' = S \cdot T \cdot a = \begin{bmatrix} 4 & 0 & 4 & 1 \end{bmatrix}$$

$$a'' = T \cdot S \cdot a = \begin{bmatrix} 3 & 0 & 3 & 1 \end{bmatrix}$$

Linear Algebra

- EX: translate then scale vs scale then translate

$$S = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

$$b' = S \cdot T \cdot b = \begin{bmatrix} 4 & 0 & 0 & 1 \end{bmatrix}$$

$$b'' = T \cdot S \cdot b = \begin{bmatrix} 3 & 0 & -1 & 1 \end{bmatrix}$$

Linear Algebra

- Orbit v Spin
  - Comes down to the matrix multiplication order
  - Similar to the previous example
- Orbiting:
  - Translate away from the origin first
  - Then apply the rotation
- Spin:
  - Make sure model is at origin before applying rotation

Linear Algebra

- When parsed object:
  - Placed the OBJ in the center by subtracting the center from each of the points
  - Scaled the OBJ by multiplying each of the points by some factor
- Equivalent to using matrix multiplication:

$$S \cdot T \cdot p$$

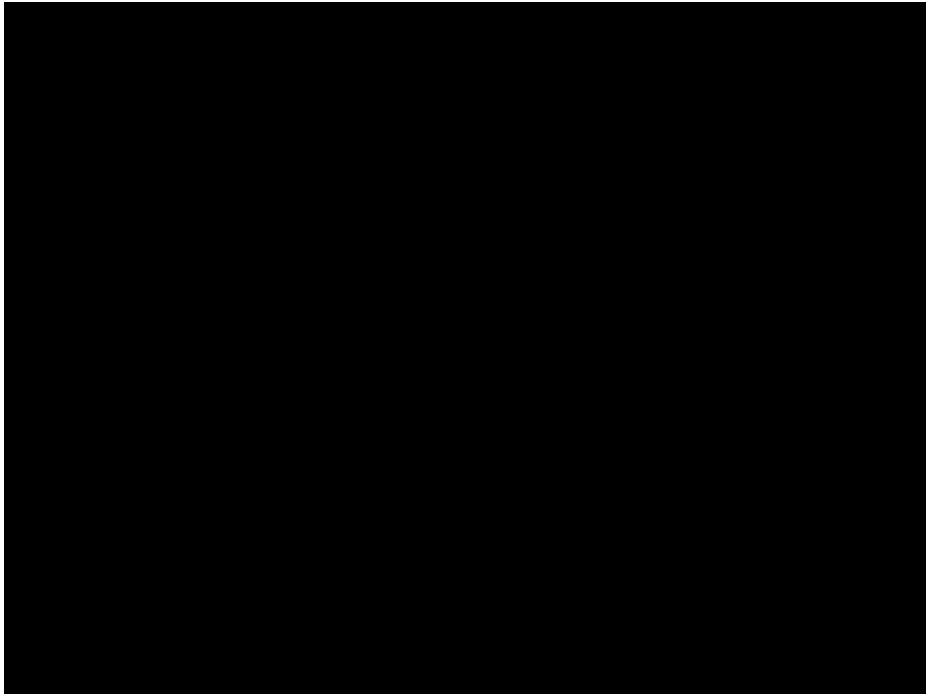
- Alternatively could have used the toWorld Matrix aka the model matrix

Linear Algebra

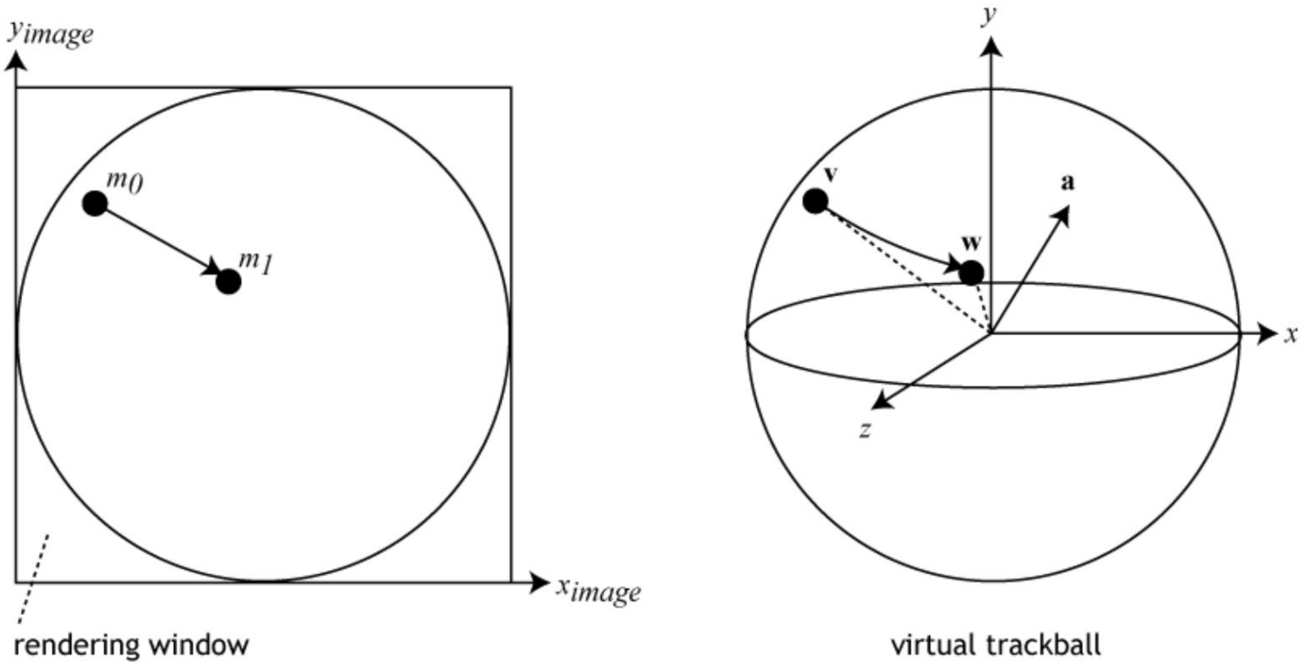
- To World Matrix (model in starter code)
  - Takes model from local space to world space
  - Places object in world
  - Changing this matrix is what rotated/spun the objects
- Instead of changing the points directly could have initialized this model matrix to $S \cdot T$
- Using matrices to change the points/position of the model will be very useful

Mouse Control

Mouse Control



Mouse Control



Mouse Control

- Trackball mapping
  - Taking two different 2D screen positions and mapping them into two 3D vectors
  - Based on these 3D vectors you find the angle and axis to rotate your model
    - Angle: angle between these two vectors
    - Axis: perpendicular to both of these vectors

Mouse Control

```
glm::vec3 Window::trackBallMapping(glm::vec2 point) {

    glm::vec3 v;    // Vector v is the 3D position of the mouse on the trackball
    float d;        // depth of the mouse location to calculate

    v.x = (2.0f * point.x - width) / width;     // mouse X position in trackball coordinates (range from -1 to 1)
    v.y = (height - 2.0f * point.y) / height;   // mouse Y position
    v.z = 0.0f;                                 // mouse Z position is initally zero


    d = glm::length(v);             // distance from the trackball's origin to the mouse location,
                                    // without considering depth (so in the plane of the trackball's origin)
    d = (d < 1.0f) ? d : 1.0f;      // this limits d to values of 1.0 or less
    v.z = sqrtf(1.001f - d*d);      // this calculates the z coordinate of the mouse position on the trackball,
                                    // based on Pythagoras: v.z*v.z + d*d = 1*1
    v = glm::normalize(v);          // need to normalize, since we only capped d, not v.
    return v;                       // v is the point in 3D of the mouse location
}
```

Mouse Control

- Need to get the Mouse Position
  - Main.cpp - setup_callbacks(...)
    - Add: glfwSetCursorPosCallback(...)
      - Tells you where the mouse is
    - Add: glfwSetMouseCallback(...)
      - Tells you what/if the mouse buttons are clicked
  - Window.cpp
    - Add: cursor_callback(...)
    - Add: mouse_callback(...)

Mouse Control

- Scaling the model:
  - Similar to rotating with trackball need another callback to get the scroll information
  - Apply a matrix transformation to scale the object up/down
    - Keep in mind the order of operations

Resources

- [LearnOpenGL]

Questions?