CSE 167 Fall 2020

Discussion 3

Important Dates

- Project 1 due @ **Oct 25 - 11:59 pm**
  - If late, 25% deduction!
- Project 2 posted: due @ **Nov 8th - 11:59pm**
  - http://ivl.calit2.net/wiki/index.php/Project2F20

Overview

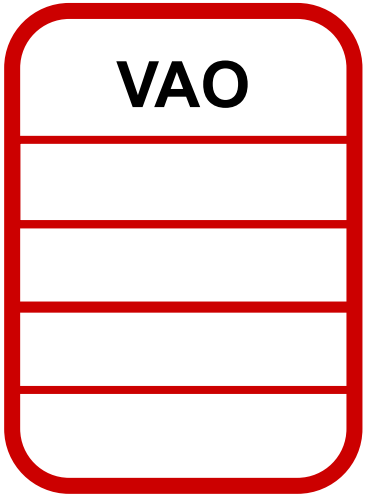
- Model Loader
  - OpenGL
- Linear Algebra
  - Homogeneous coordinates
  - Vertex transformations
    - Matrix multiplication
    - Orbit vs. spin

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

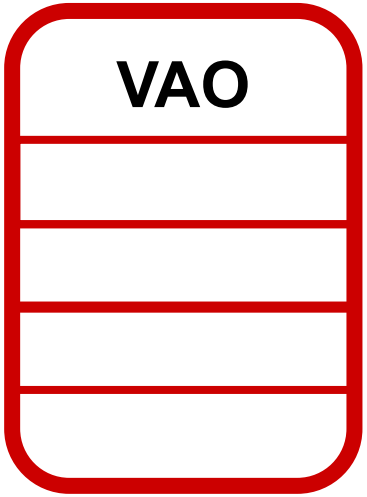
OpenGL

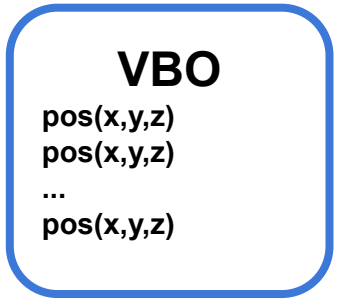


- VAO, VBO(s), & EBO:
  - VAO = Vertex Attribute Array
    - container for buffers
    - ie. VBO(s) and EBO
  - VBO = Vertex Buffer Object
    - hold model information
    - ie. Positions, Normals..
  - EBO = Element Buffer Object
    - hold index information
    - ie. indices for the faces

OpenGL

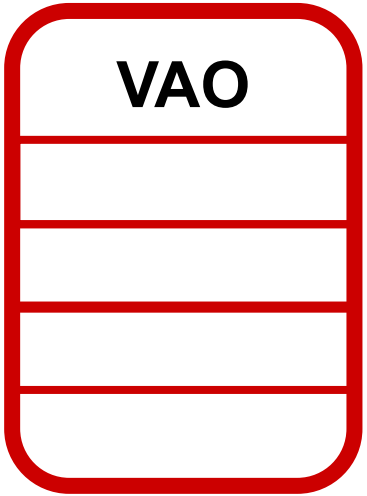
- VAO, VBO(s), & EBO:
  - VAO = Vertex Attribute Array
    - container for buffers
    - ie. VBO(s) and EBO
  - VBO = Vertex Buffer Object
    - hold model information
    - ie. Positions, Normals..
  - EBO = Element Buffer Object
    - hold index information
    - ie. indices for the faces

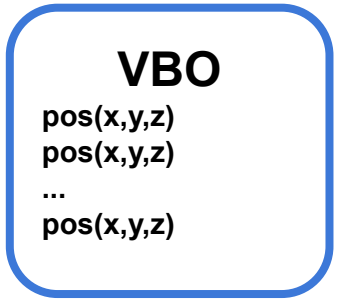


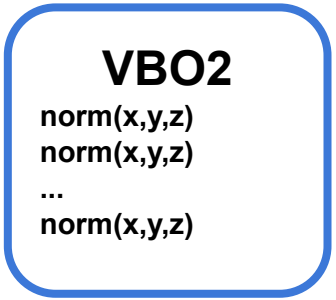


OpenGL

- 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

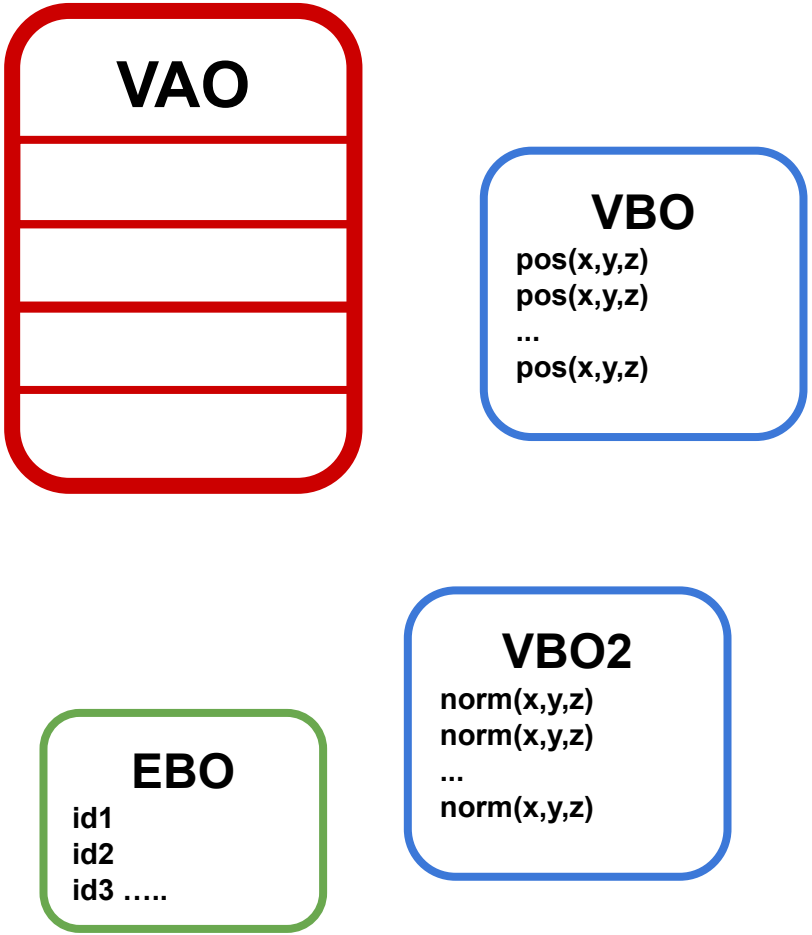






OpenGL

- 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



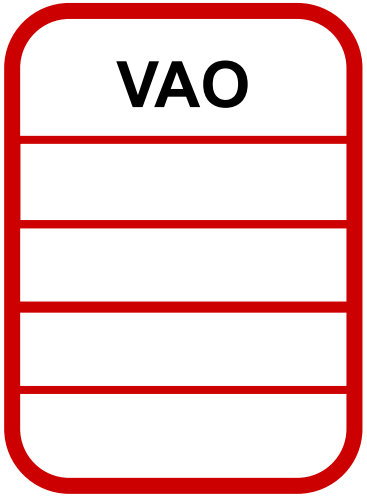
OpenGL

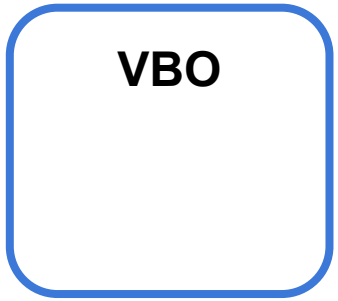
- Generate VAO and VBO(s)
- Bind VAO
  - Binding VAO “activates” it
  - Binding VBO after “attaches” it to the bound VAO

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

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

OpenGL



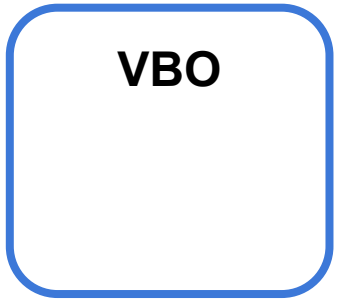


- Generate VAO and VBO(s)
- Bind VAO
  - Binding VAO “activates” it
  - Binding VBO after “attaches” it to the bound VAO

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

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

OpenGL

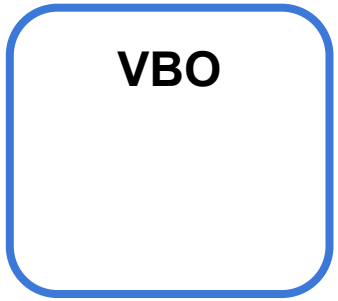


- Generate VAO and VBO(s)
- Bind VAO
  - Binding VAO “activates” it
  - Binding VBO after “attaches” it to the bound VAO

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

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

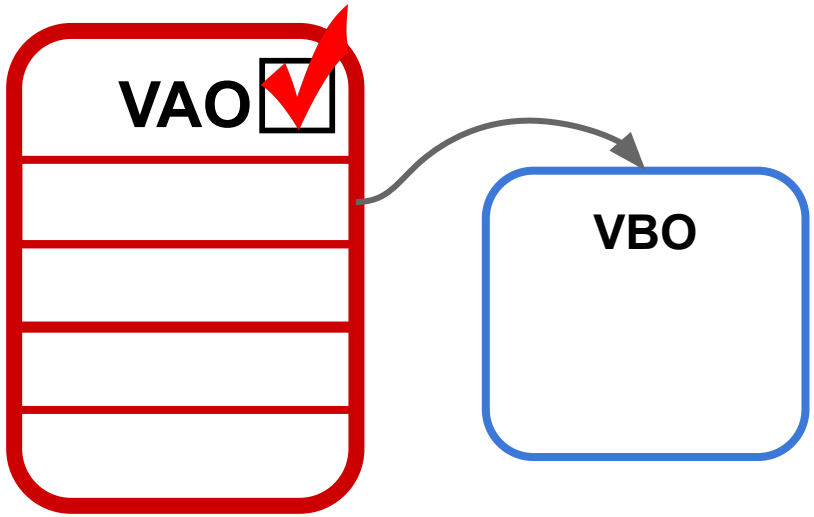
OpenGL



- 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);
```

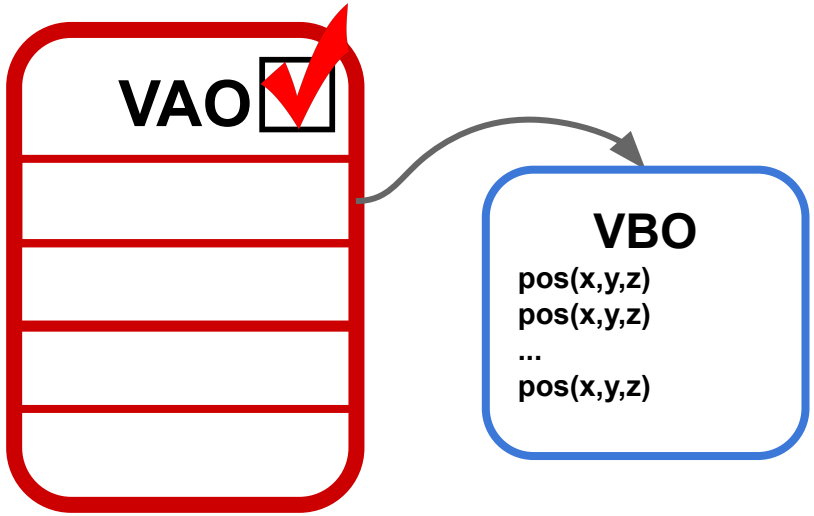
OpenGL



- 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);
```

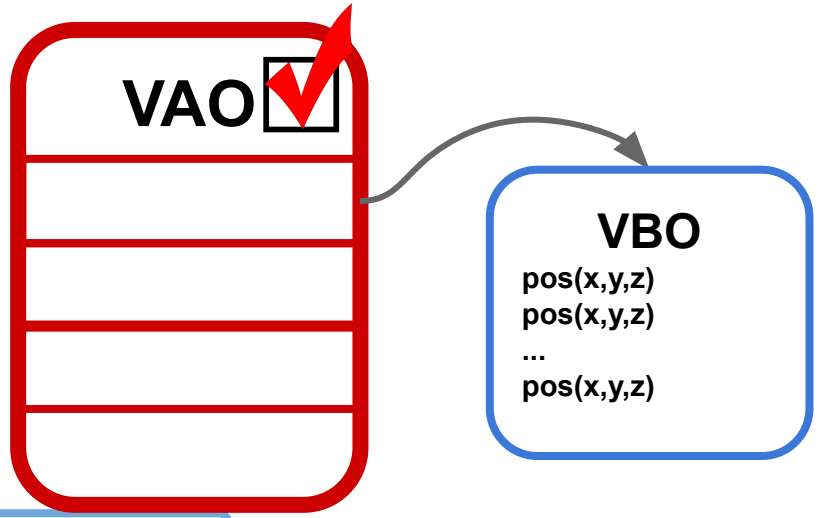
OpenGL



- 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);
```

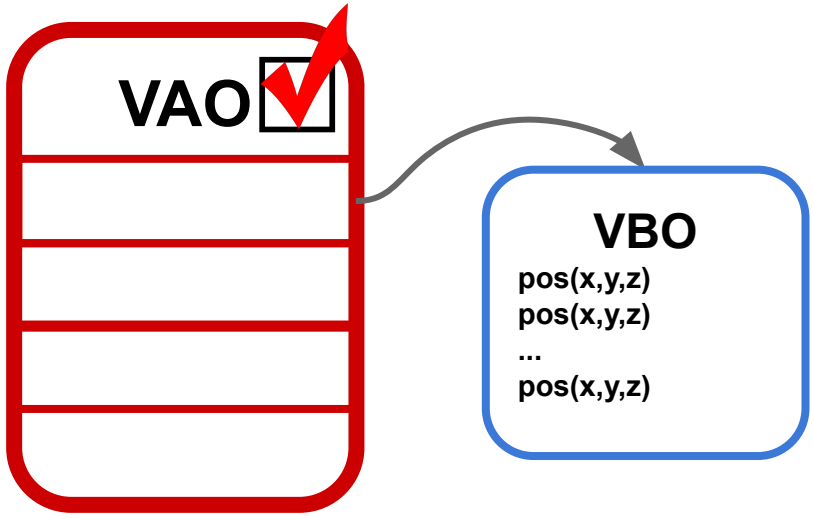
OpenGL



- 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

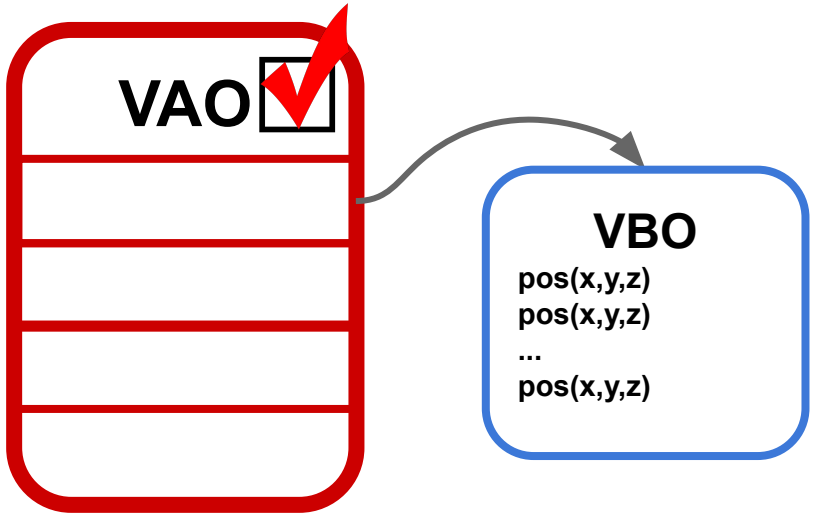


- For each VBO:
  a. Bind buffer to VAO
  b. Populate data in VBO
  c. Create channel between VBO and shader
  d. Tell shader how to read VBO

```
// 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

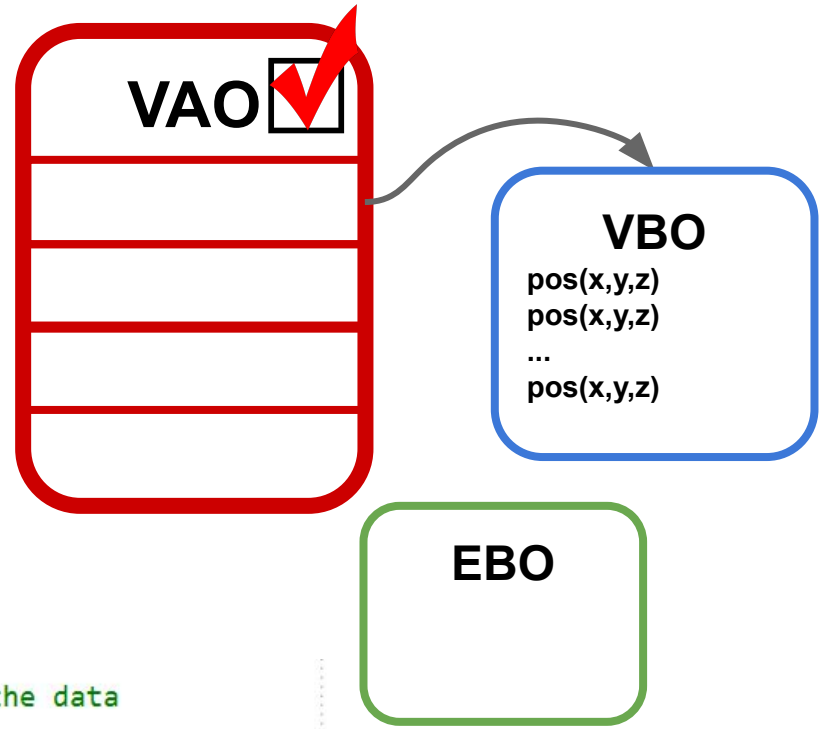
- For EBO:
  a. Generate EBO
  b. Bind buffer
  c. Populate data in EBO




```
// 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

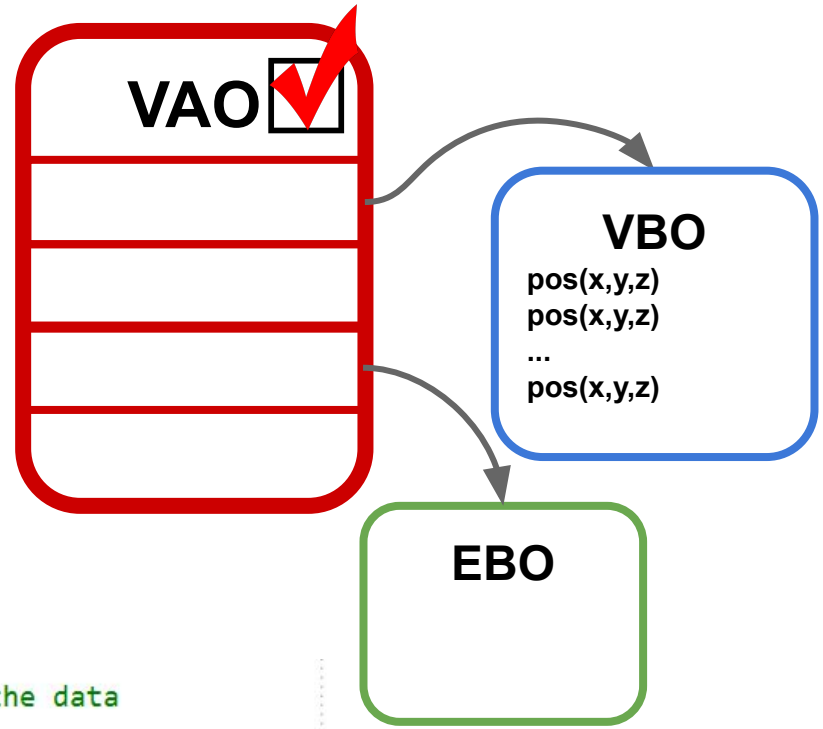
- For EBO:
  a. Generate EBO
  b. Bind buffer
  c. Populate data in EBO




```
// 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

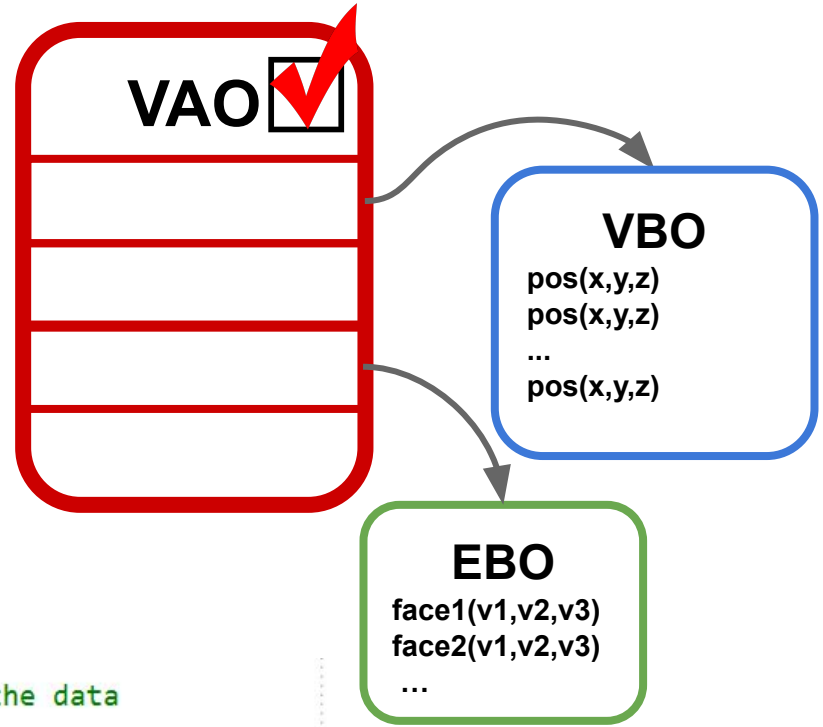
- For EBO:
  a. Generate EBO
  b. Bind buffer
  c. Populate data in EBO




```
// 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 ***A TRIANGLE MESH***:
  - 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 for point
  - Add 0 for vector
  - 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
  - For example, pa1 models.

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

GLM: Translate

- glm::translate(mat4 M, vec3 t) returns a ***copy of M*** that has been **right-multiplied** by the translation matrix made from t
- You can get the translation matrix by itself just by passing in
  - glm::translate(glm::mat4(1.0f), t)*vec4

$$M * T = \begin{bmatrix} x_1 & y_1 & z_1 & t'_x \\ x_2 & y_2 & z_2 & t'_y \\ x_3 & y_3 & z_3 & t'_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

GLM: Scale

- glm::scale(glm::mat4 M, glm::vec3 s) returns a **copy of M** where the **first three columns are scaled by v.x, v.y, and v.z** respectively
- You can get the scale matrix by itself just by passing in
  - glm::scale(glm::mat4(1.0f), s)

$$\begin{bmatrix} s_x * x_1 & s_y * y_1 & s_z * z_1 & t_x' \\ s_x * x_2 & s_y * y_2 & s_z * z_1 & t_y' \\ s_x * x_3 & s_y * y_3 & s_z * z_1 & t_z' \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

GLM: Rotate

- glm::rotate(glm::mat4 M, float angle, glm::vec3 axis) returns a **copy of m right-multiplied by the rotation matrix** made from the axis and angle (degrees) passed in
- You can get the rotation matrix by itself just by passing in
  - glm::rotate(glm::mat4(1.0f), float angle, glm::vec3 axis)

$$M * R = \begin{bmatrix} x_1 & y_1 & z_1 & t'_x \\ x_2 & y_2 & z_2 & t'_y \\ x_3 & y_3 & z_3 & t'_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} r_1 & r_2 & r_3 & t_x \\ r_4 & r_5 & r_6 & t_y \\ r_7 & r_8 & r_9 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Questions?