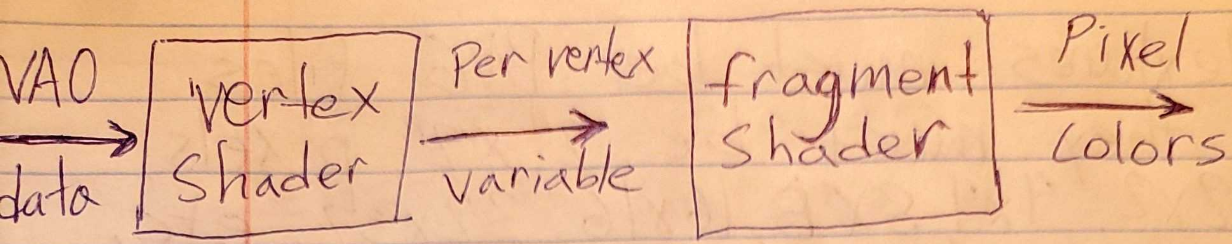


- This is our current setup, and it works relatively well. The problem with it however is that it can only take data in from the VAO.

  - We want more than just the VAO to determine how a model is rendered. for example light, fog, glows, particles, etc. Also moving and rotating the model cannot be done with just a VAO.

  - Our solution is uniform variables.

Uniform Variables:

- Variables that are in the shader code that can be set by our java code at any time.

- This means we can send data into either shader at any time to change how the models are rendered.

• we will calculate things such as light/brightness, fog density, etc. and then send that data into the shaders through uniform variables.

to-Do:

- We need to know the location of our uniform variables when load our models so we can access them.

- The first use of our uniform variables will be to move and resize our models.

- The definitions for these uses are on the next page.

Translation: $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$ Moving a model from one position to another.
- Shown with an (x, y, z) coordinate.

Rotation: (r_x, r_y, r_z) Rotating a model in a direction.
- Shown with rotation angles for each axis. (these are called euler rotations)

Scale: S Scaling a ~~n~~ model bigger or smaller. (changing it's size)
- Shown with a single scale value S. (One (1) is the normal model size)

- These three ways to alter an object are known together as an object's Transformation.

- Each object in our code will have it's own transformation.

- Transformations are stored as a 4X4 Matrix. (A transformation matrix)

- Transformation Matrices are actually a very mathematically involved subject so I will not go into any details. All we ~~kr~~ need to know how to do is turn our three factors (translation, scale, and rotation) into a single transformation matrix.

Entity: An entity is an object containing a model, a position, a rotation, and a scale. This way we can have multiple models at different sizes and positions in our screen.

- If we want to render 100 trees, instead of editing our models 100 times, causing mass lag, we can use instead an entityes values to directly change the model INSIDE our ~~vector~~ shader code. This way we can change the model without changing it's VAO.

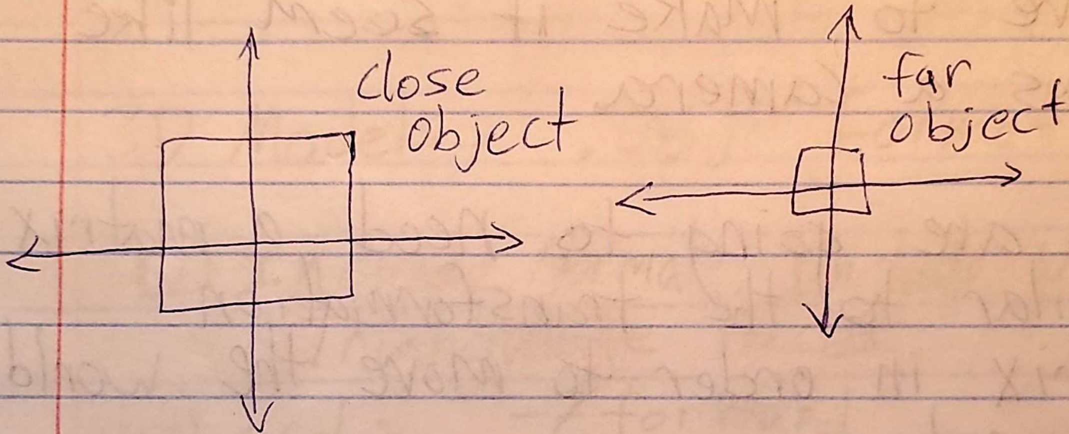
- 1 VAO, 100 trees. Because of uniform variables.

- The way our transformation Matrix can alter our vertex coordinates inside our shader code is simply by multiplying them!

$$\underbrace{\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}}_{\text{transformation matrix}} \overset{\text{multiply}}{X} \underbrace{\begin{bmatrix}x\\y\\z\\1.0\end{bmatrix}}_{\text{vertex matrix}} = \begin{bmatrix}x_t\\y_t\\z_t\\W_t\end{bmatrix} \ \text{transformed vertex}$$

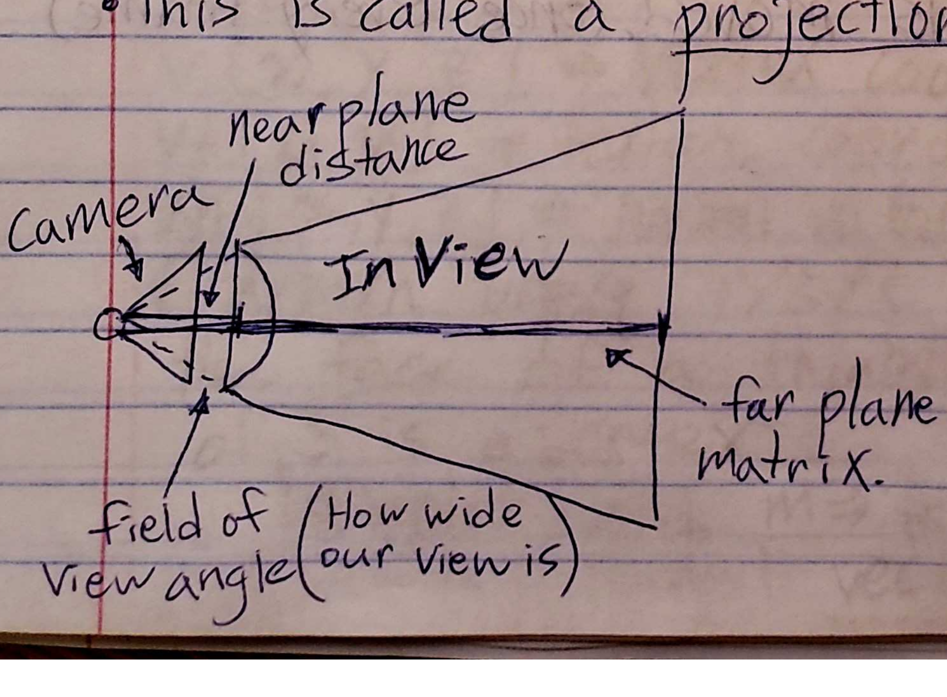
- The vertex Shader Code does this for every vertex in our model.

- When you look at the screen, you can only see the x and y because the z goes towards you. In order to make our models 3D we can scale them up and down to make them appear to have dimension.



- We can use a matrix similar to the transformation matrix that scales the model based on z values

- This is called a <u>projection Matrix</u>



- This matrix makes our scene look like the diagram It copies real life.

- A projection matrix uses a lot of matrix math to achieve a realistic view of our scene.

- In OpenGL, there is no actual camera that moves around. Instead all the world's objects move to make it seem like theres a camera.

- We are going to need a matrix similar to the transformation matrix in order to move the world around.

<u>View Matrix:</u> The transformation matrix used to move all objects.

- It will move when ~~all~~ our camera is moved. (once every frame)

Camera: this is our view of the World. It Has a position, yaw, pitch, and roll.

pitch =
yaw =
roll =

3D Models: .Obj format (wavefront)

☑ Include Normals, Include UVs, and Triangulate faces.
Forward: -Z forward
Up: Y UP
*All seams must be "Edge Split"
*Entire model use "smooth" shading
- put them in the res folder.

.Obj files:
v [x, y, z] ⇒ vertex coordinates
vt [u, v] ⇒ texture coordinates.
vn [x, y, z] ⇒ normal vectors
f l/m/n o/p/q r/s/t ⇒ f represents the face of a triangle and each block of 3 is a vertex
l ⇒ vertex coord. m ⇒ texture coord.
n ⇒ normal vector.