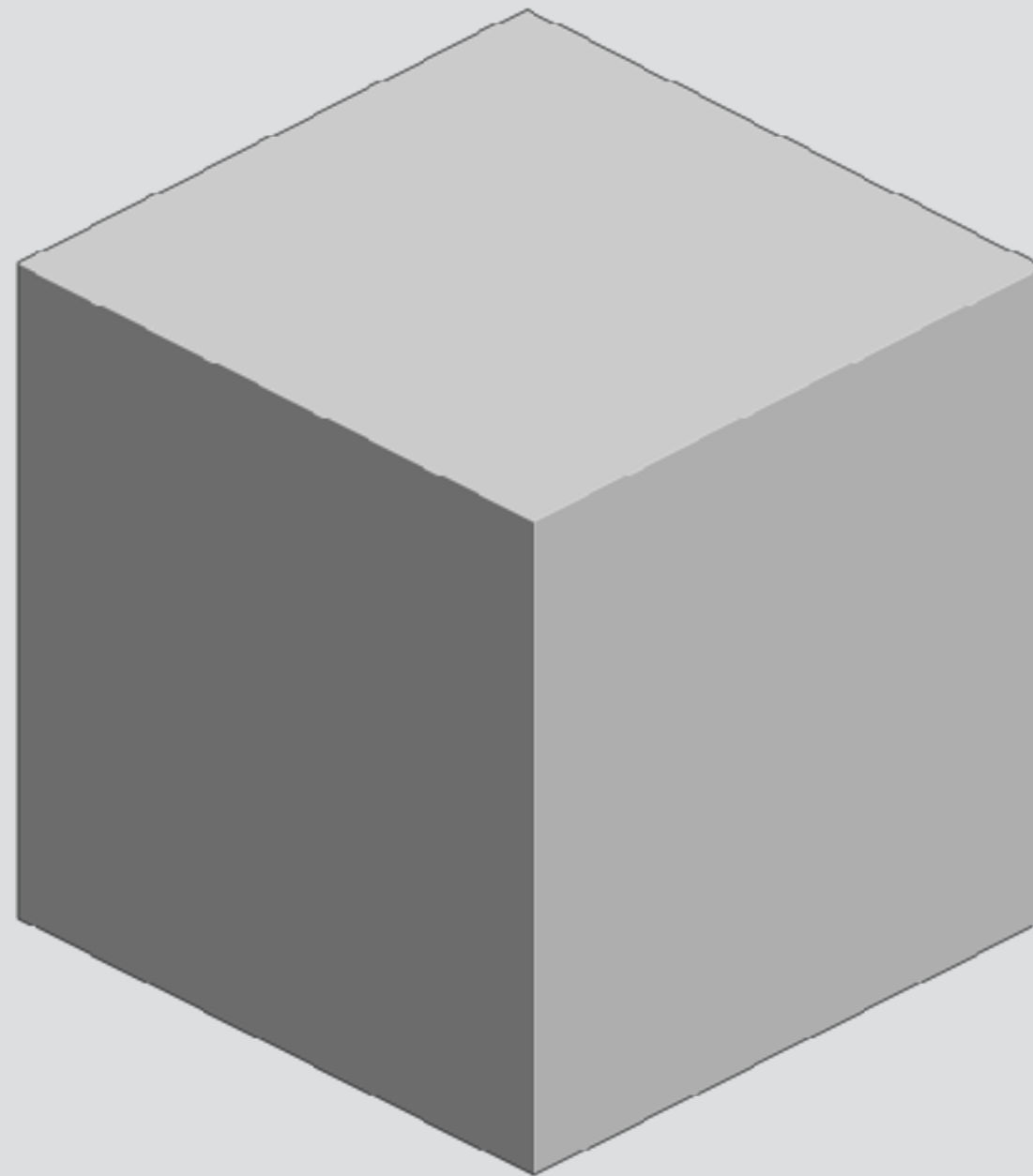


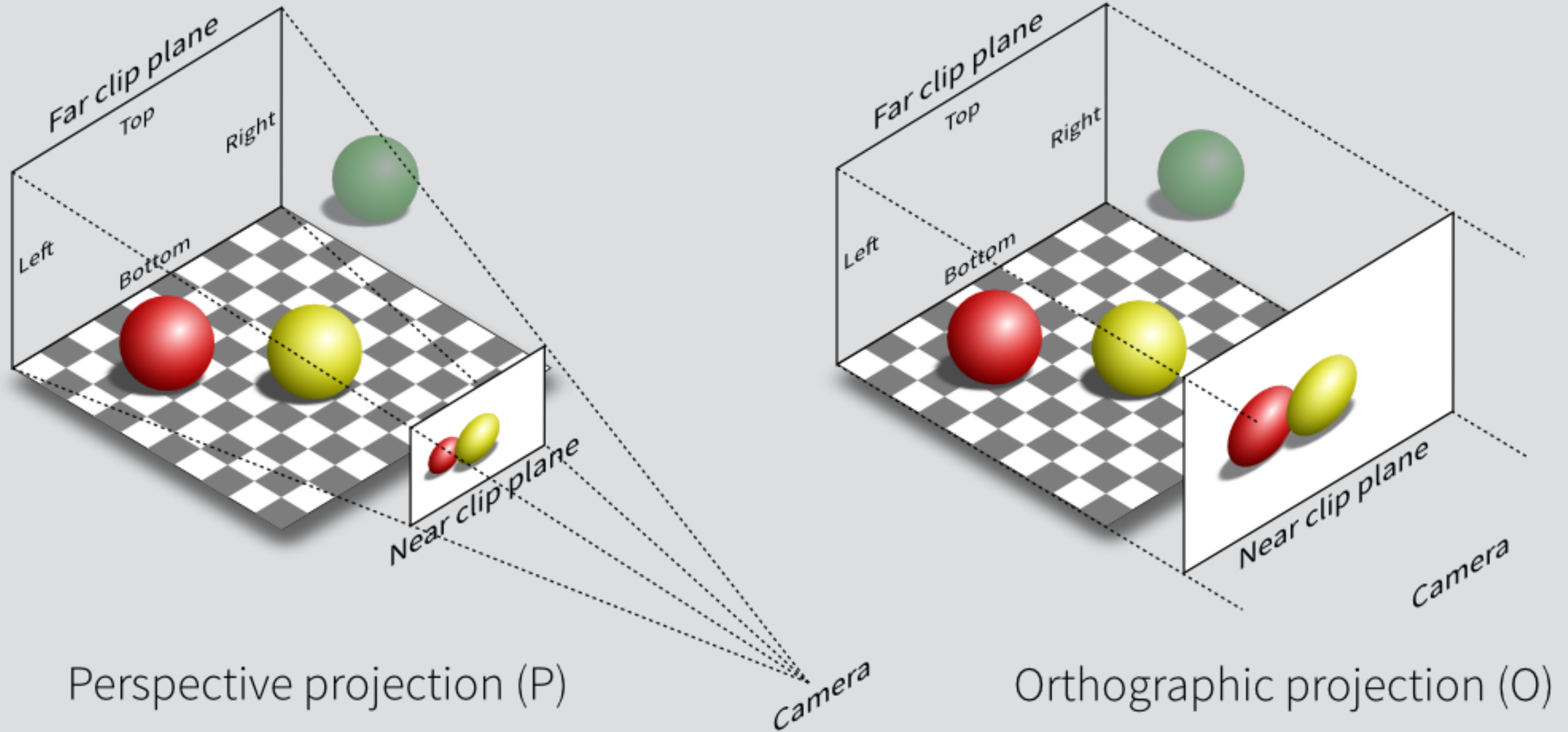
3D Graphics

Part 1



Perspective

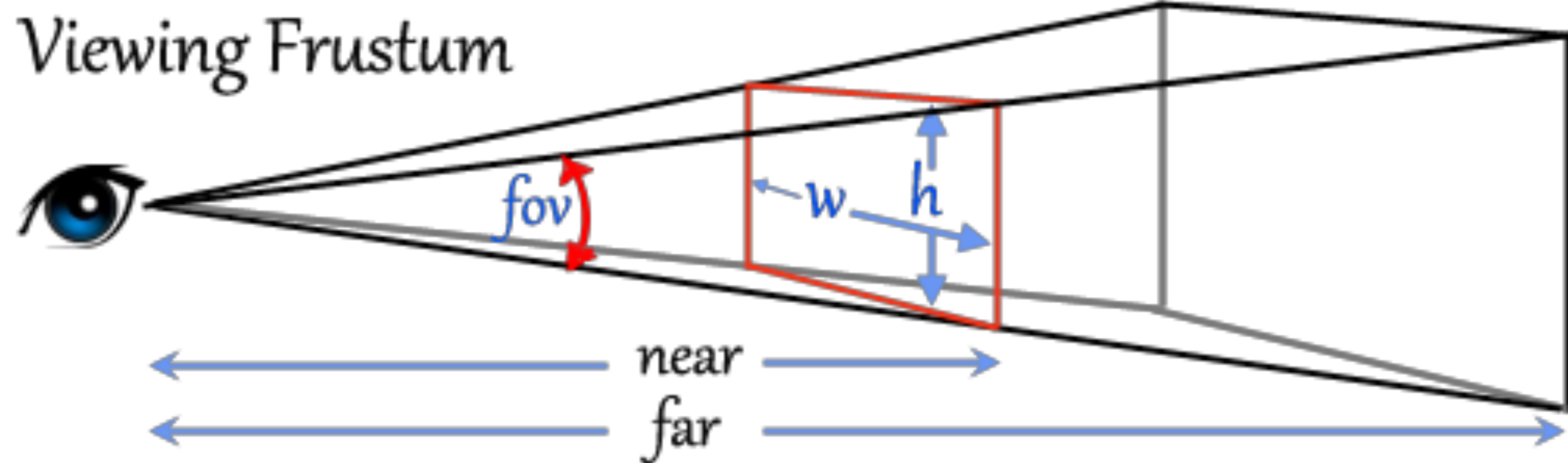
Perspective vs. Orthographic projection.

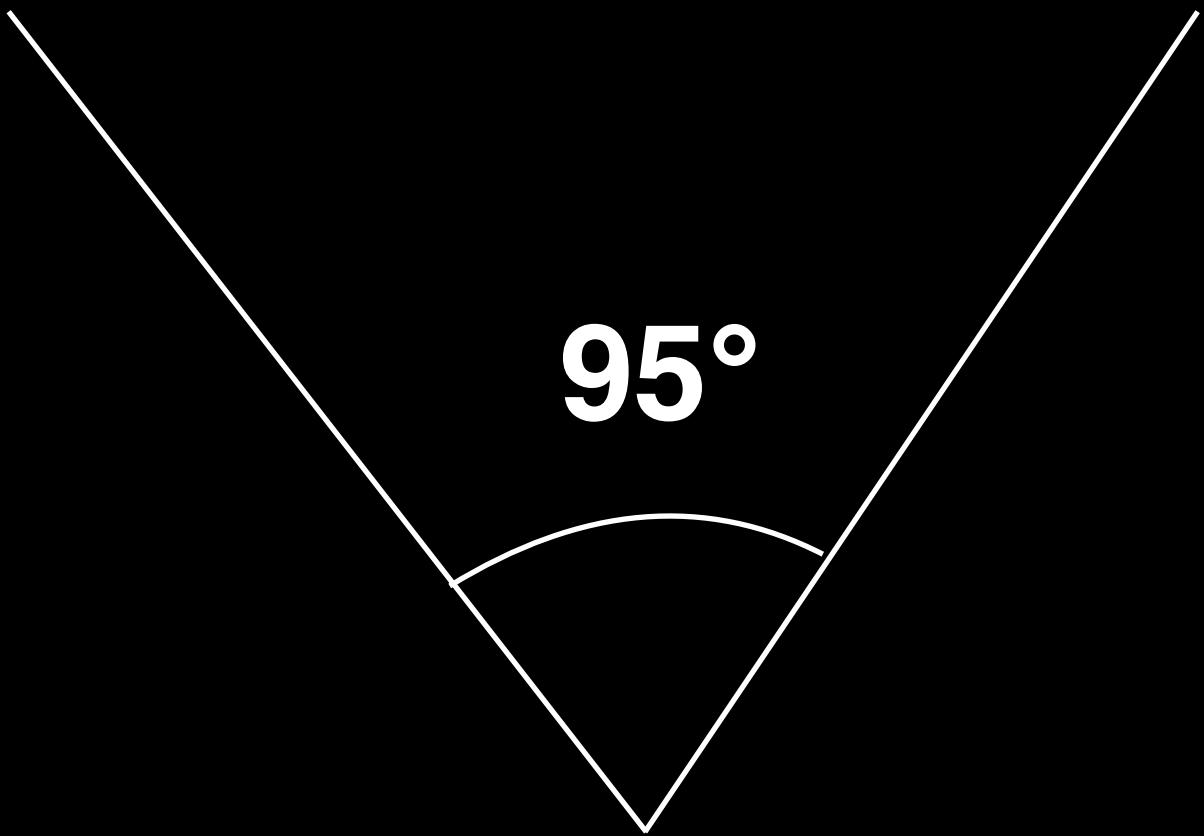
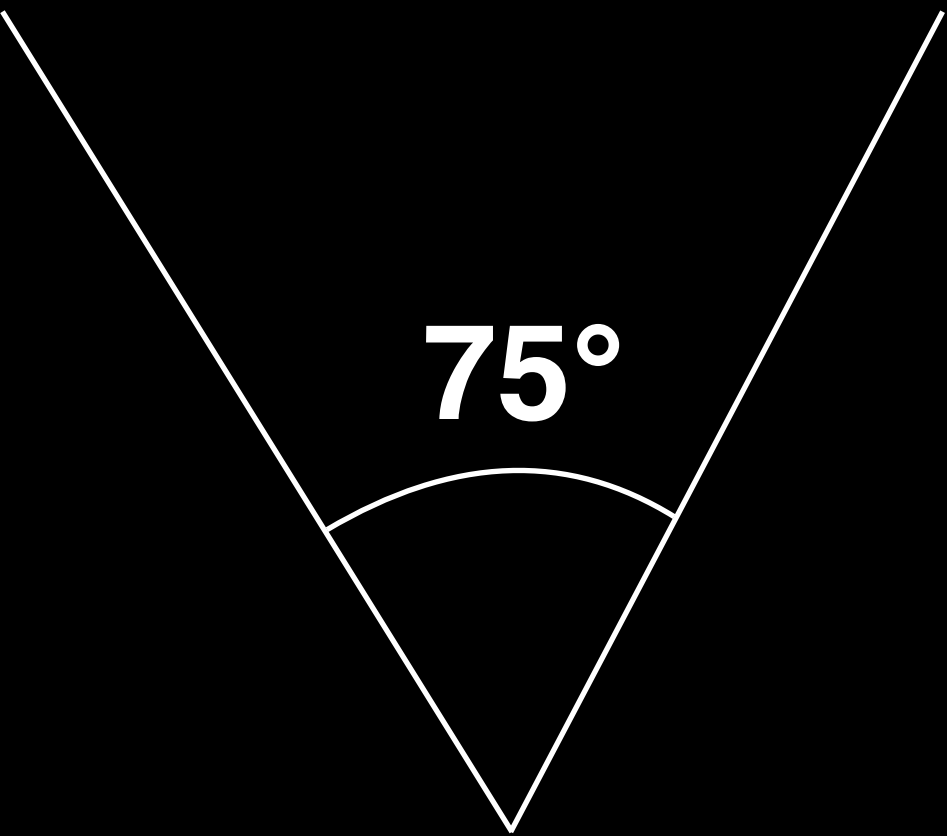
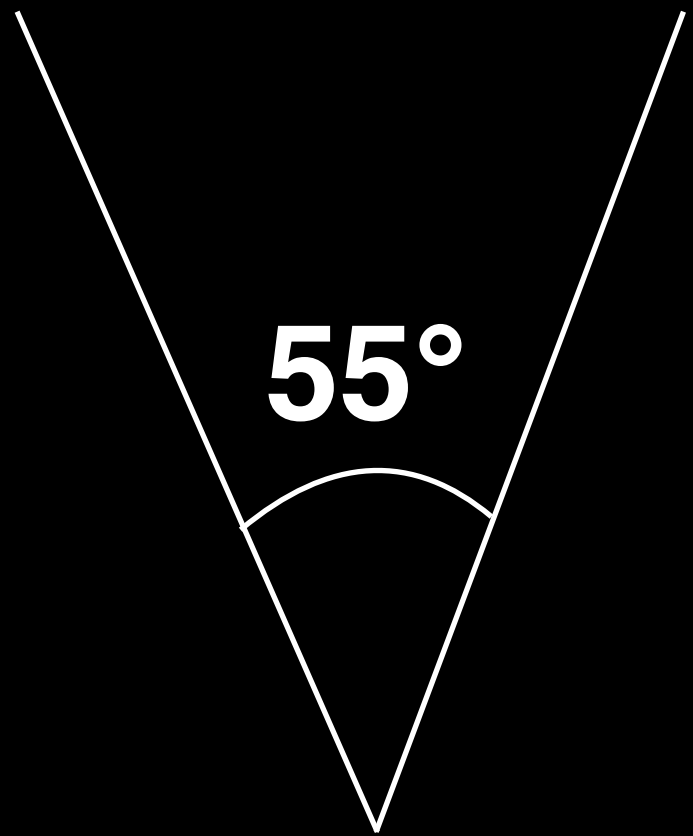


Orthographic projection matrix.

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{(r+l)}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{(t+b)}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{(f+n)}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Building a perspective projection matrix.





Perspective projection matrix.

$$P = \begin{bmatrix} \frac{\cot \frac{fovy}{2}}{aspect} & 0 & 0 & 0 \\ 0 & \cot \frac{fovy}{2} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2 * n * f}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

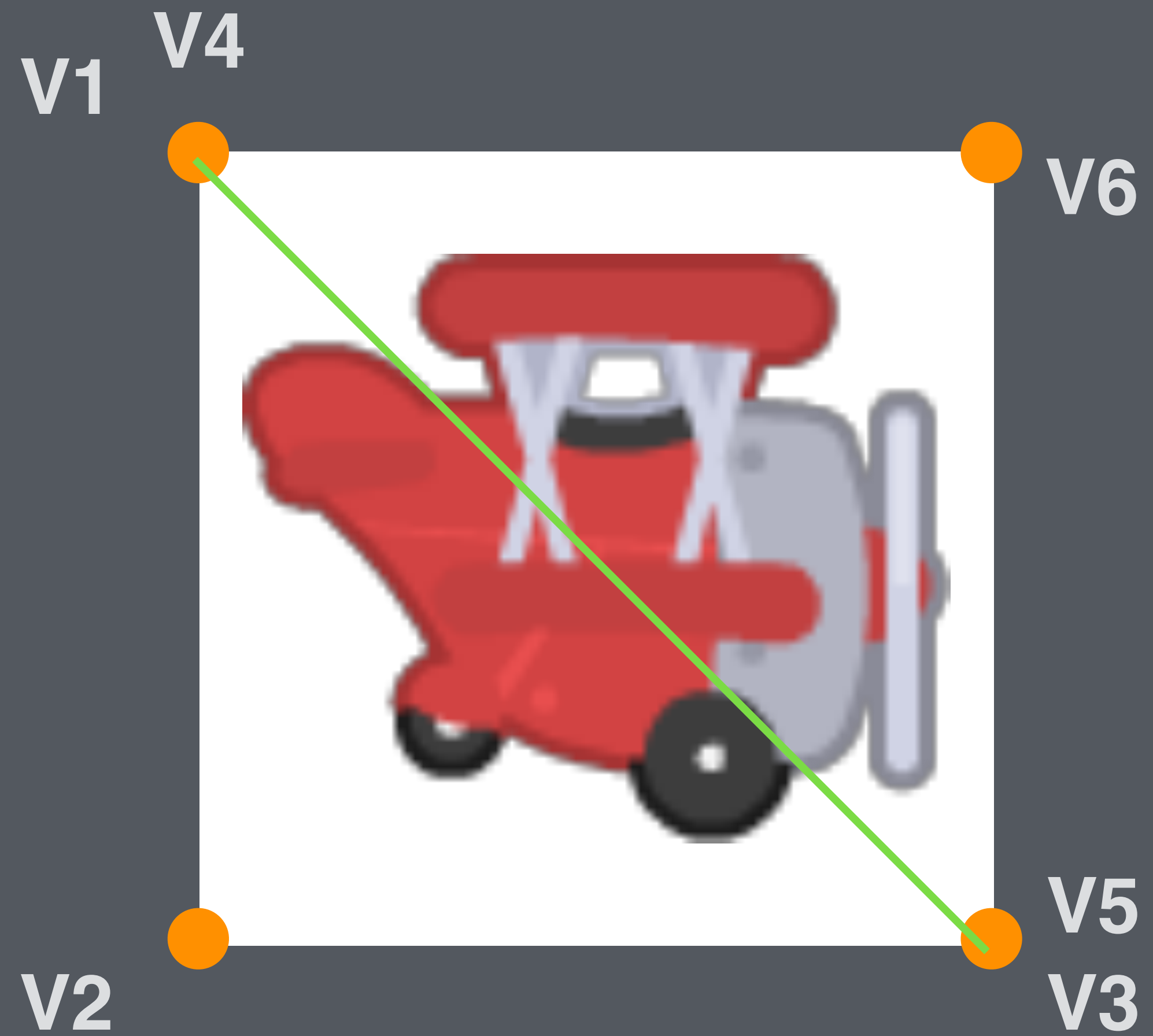
You can use `glm::perspective` to create a perspective projection matrix.

```
glm::mat4 glm::perspective(  
    float fov,  
    float aspectRatio,  
    float nearPlane,  
    float farPlane  
);
```

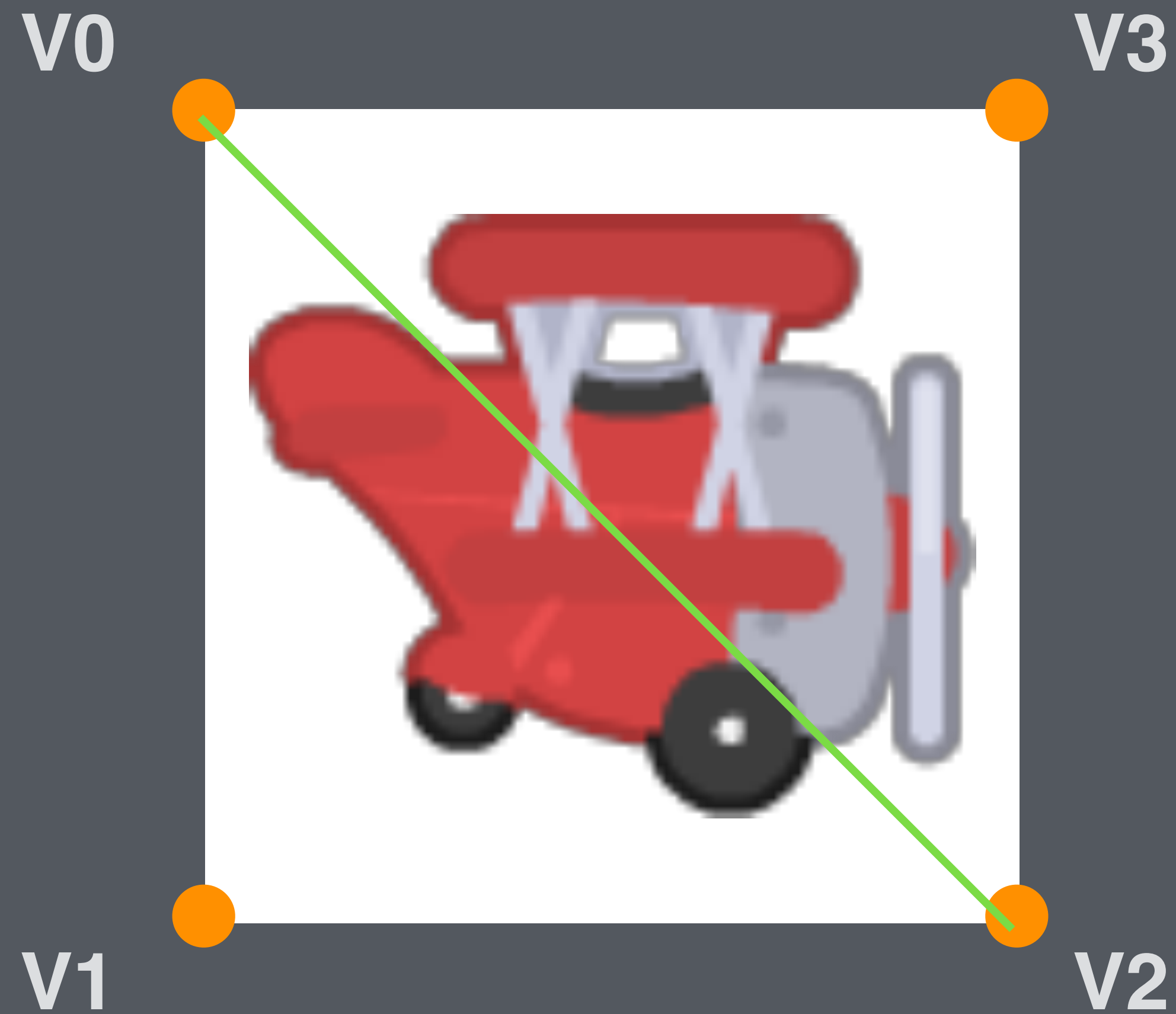
```
glm::mat4 projMatrix = glm::perspective(45.0f, 1.777f, 0.1f, 100.0f);
```

Indexed drawing.

GL_TRIANGLES



GL_TRIANGLES



V0,V1,V2

V0,V2,V3


```
void glDrawElements (GLenum mode, GLsizei  
count, GLenum type, const GLvoid *indices);
```

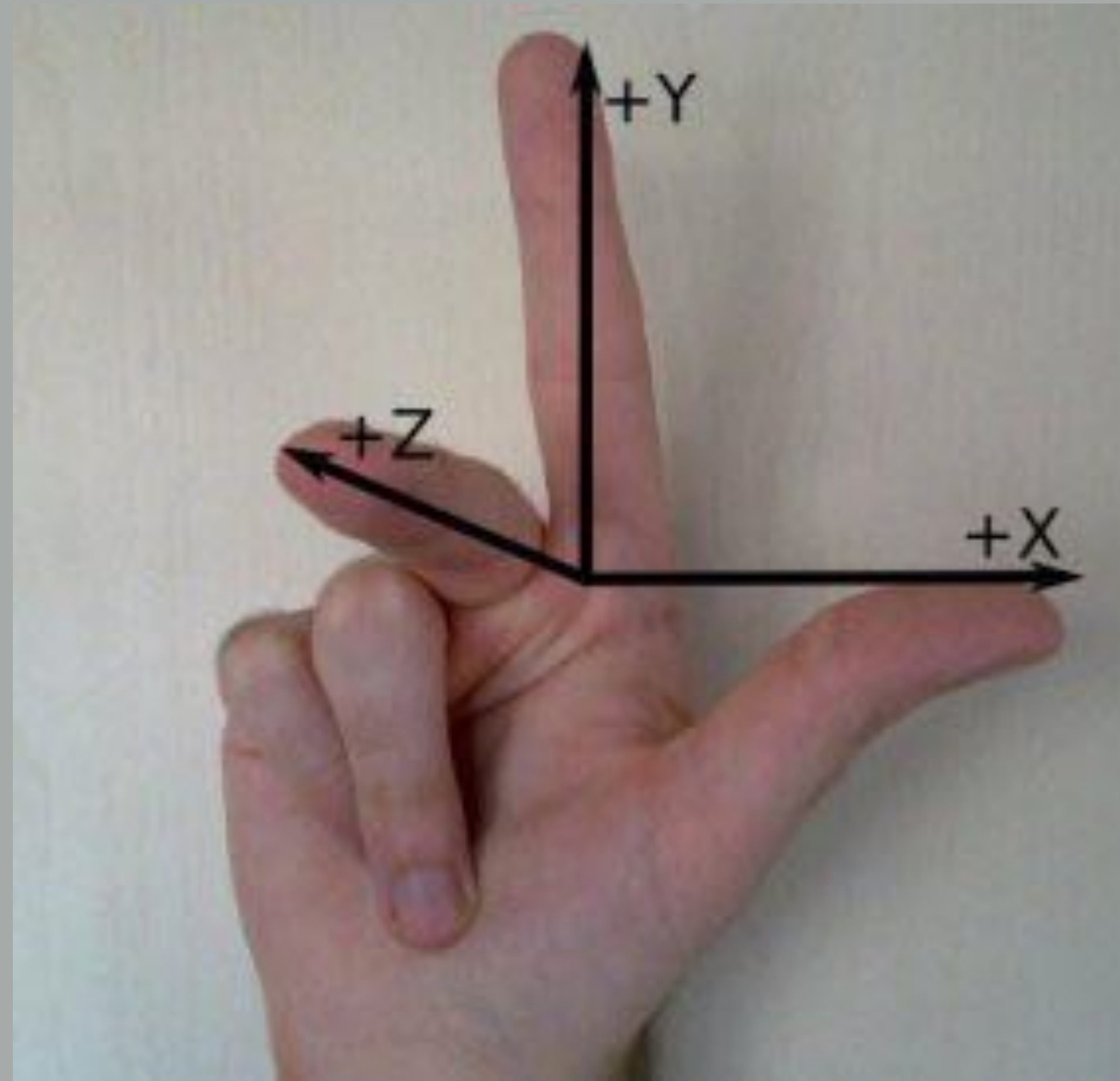
Draws vertices defined by glVertexPointer using a list of indices from that array.

```
std::vector<unsigned int> indices = {0,1,2,0,2,3};  
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, indices.data());
```

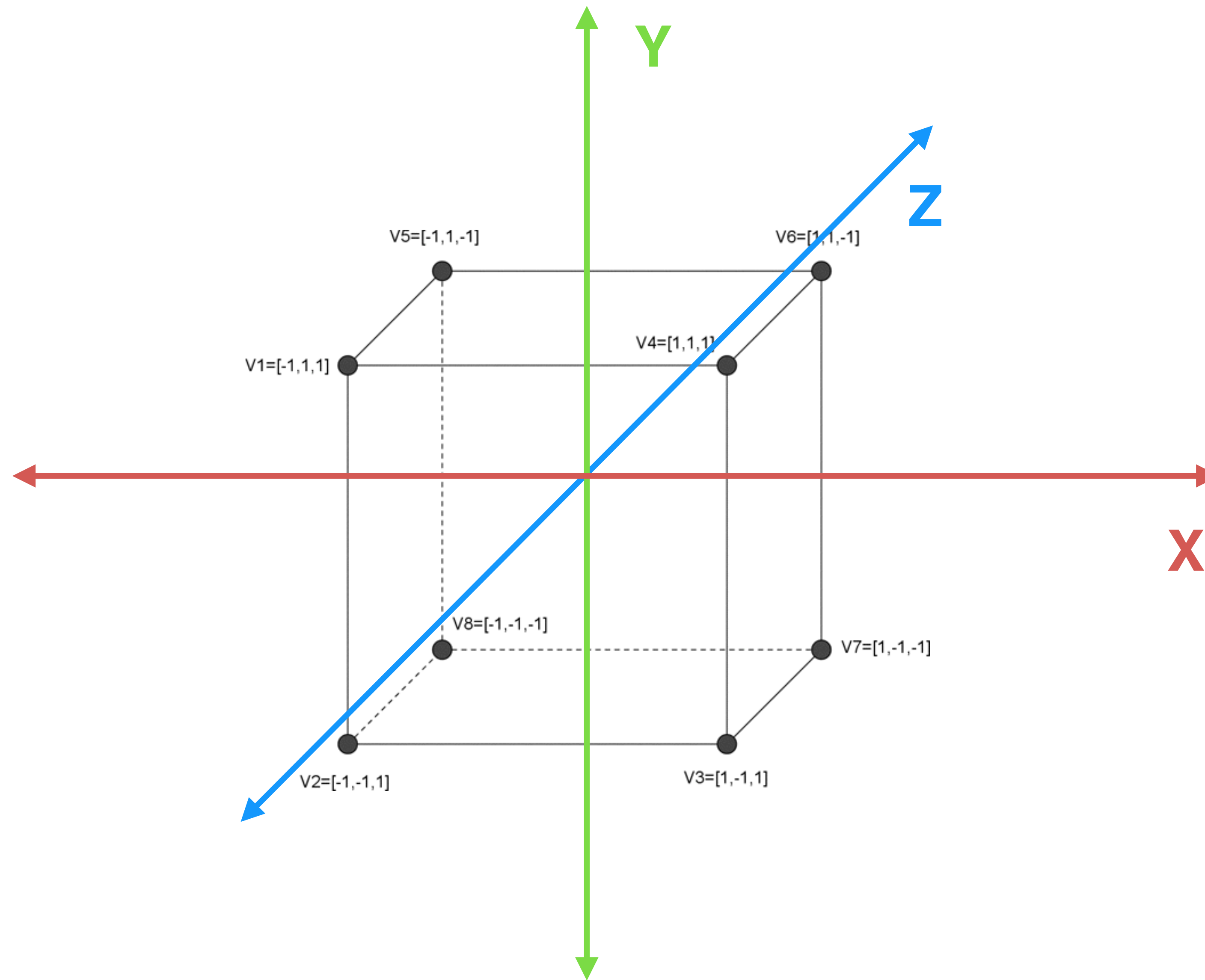
Drawing in 3D

It's the same as drawing in 2D!
We just need to specify 3 dimensions per vertex!

Right-handed coordinate system.



Drawing a cube.



```
float vertices[24] = {-1.0, -1.0, 1.0,
1.0, -1.0, 1.0,
1.0, 1.0, 1.0,
-1.0, 1.0, 1.0,
-1.0, -1.0, -1.0,
1.0, -1.0, -1.0,
1.0, 1.0, -1.0,
-1.0, 1.0, -1.0};
```

```
float colors[32] = {1.0f, 0.0f, 0.0f, 1.0,
0.0f, 1.0f, 0.0f, 1.0,
0.0f, 0.0f, 1.0f, 1.0,
1.0f, 1.0f, 0.0f, 1.0,
0.0f, 1.0f, 1.0f, 1.0,
1.0f, 0.0f, 1.0f, 1.0,
1.0f, 0.5f, 0.0f, 1.0,
1.0f, 0.0f, 0.5f, 1.0};
```

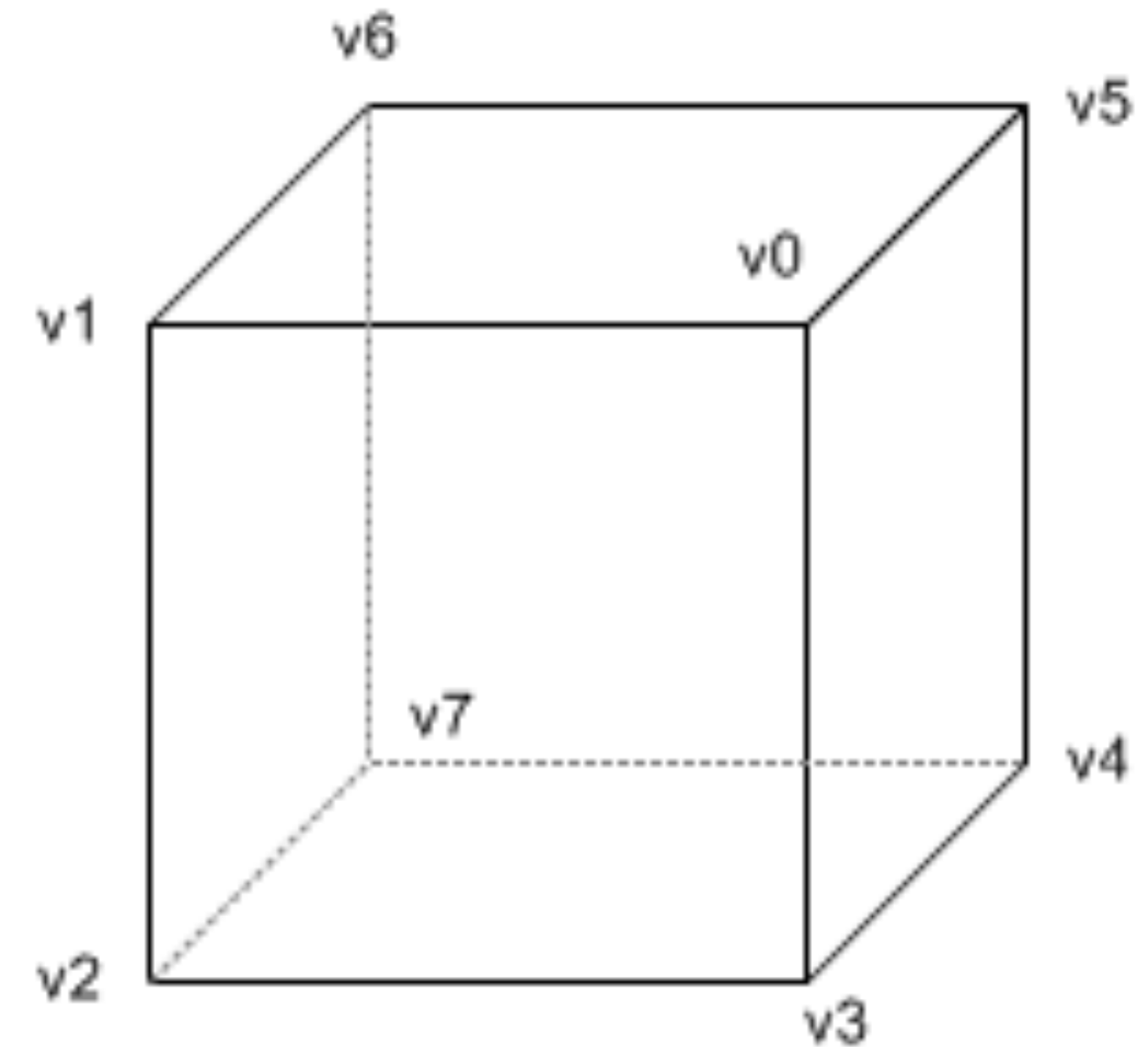
```
unsigned int indices[36] = {
0, 1, 2,
2, 3, 0,
3, 2, 6,
6, 7, 3,
7, 6, 5,
5, 4, 7,
4, 5, 1,
1, 0, 4,
4, 0, 3,
3, 7, 4,
1, 5, 6,
6, 2, 1};
```

```
glVertexAttribPointer(program.positionAttribute, 3, GL_FLOAT, false, 0, vertices);
glEnableVertexAttribArray(program.positionAttribute);
```

```
glVertexAttribPointer(colorAttribute, 4, GL_FLOAT, false, 0, colors);
glEnableVertexAttribArray(colorAttribute);
```

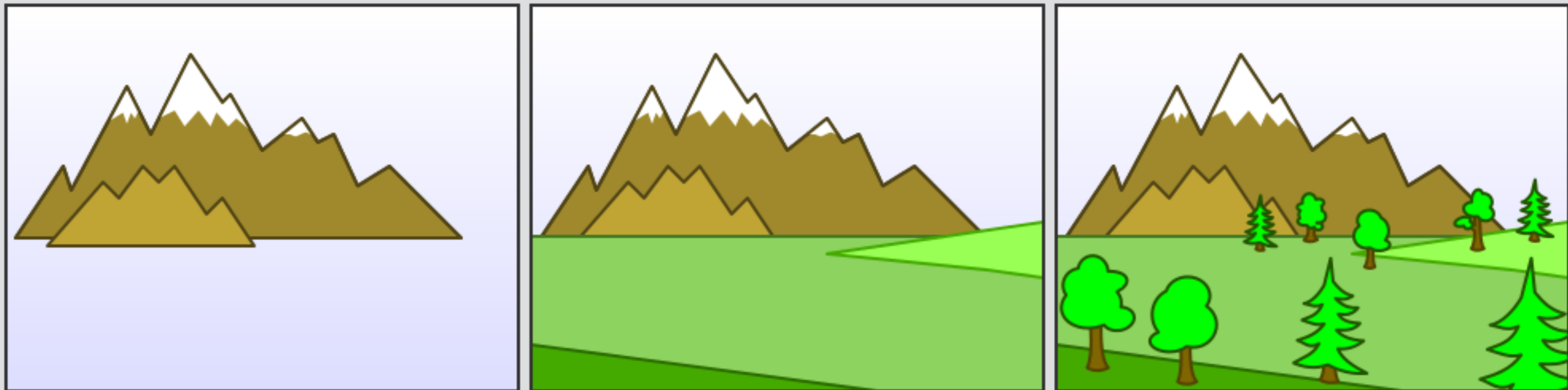
```
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, indices);
```

3 DIMENSIONS PER VERTEX

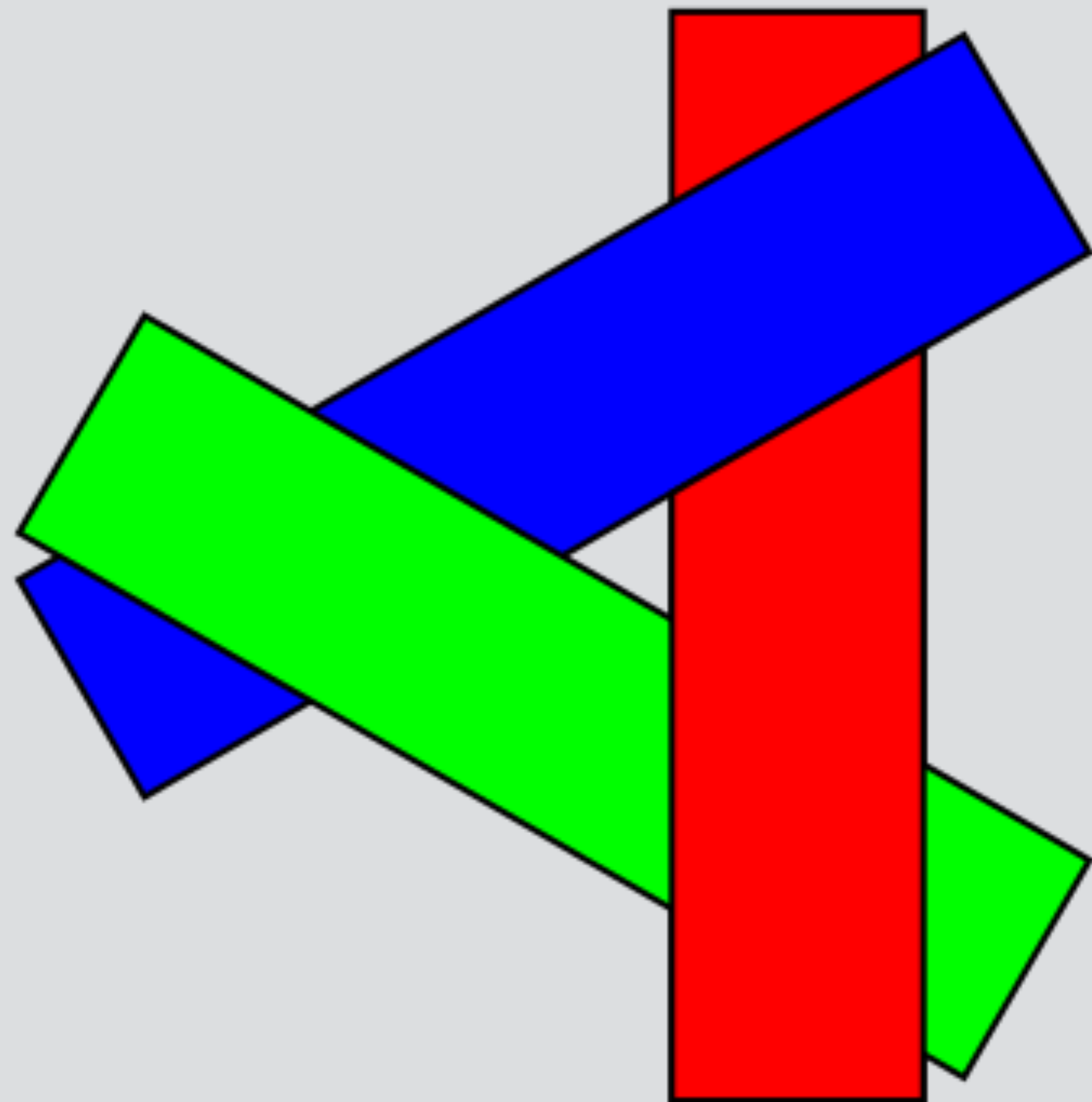


3 DIMENSIONS PER VERTEX

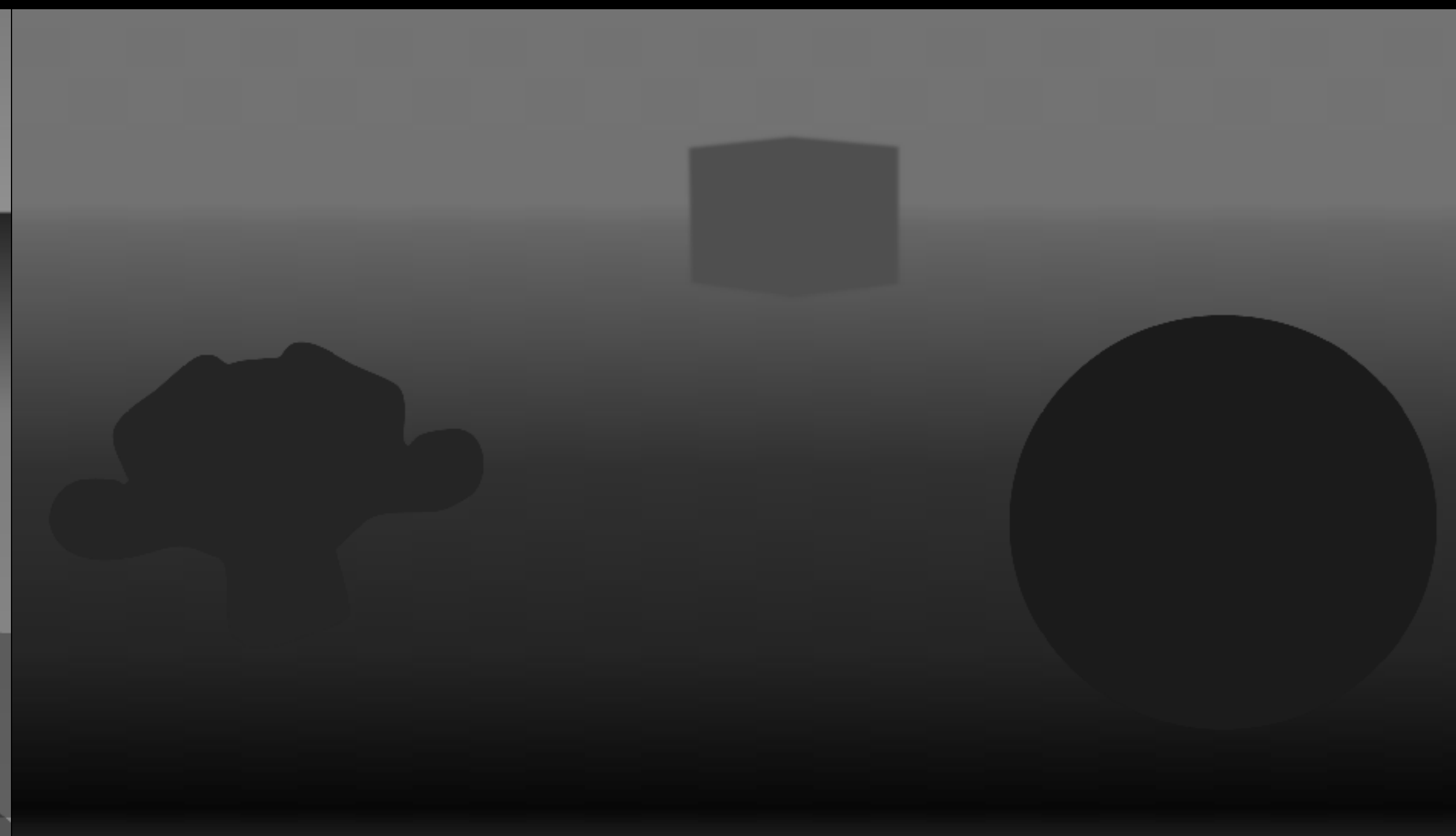
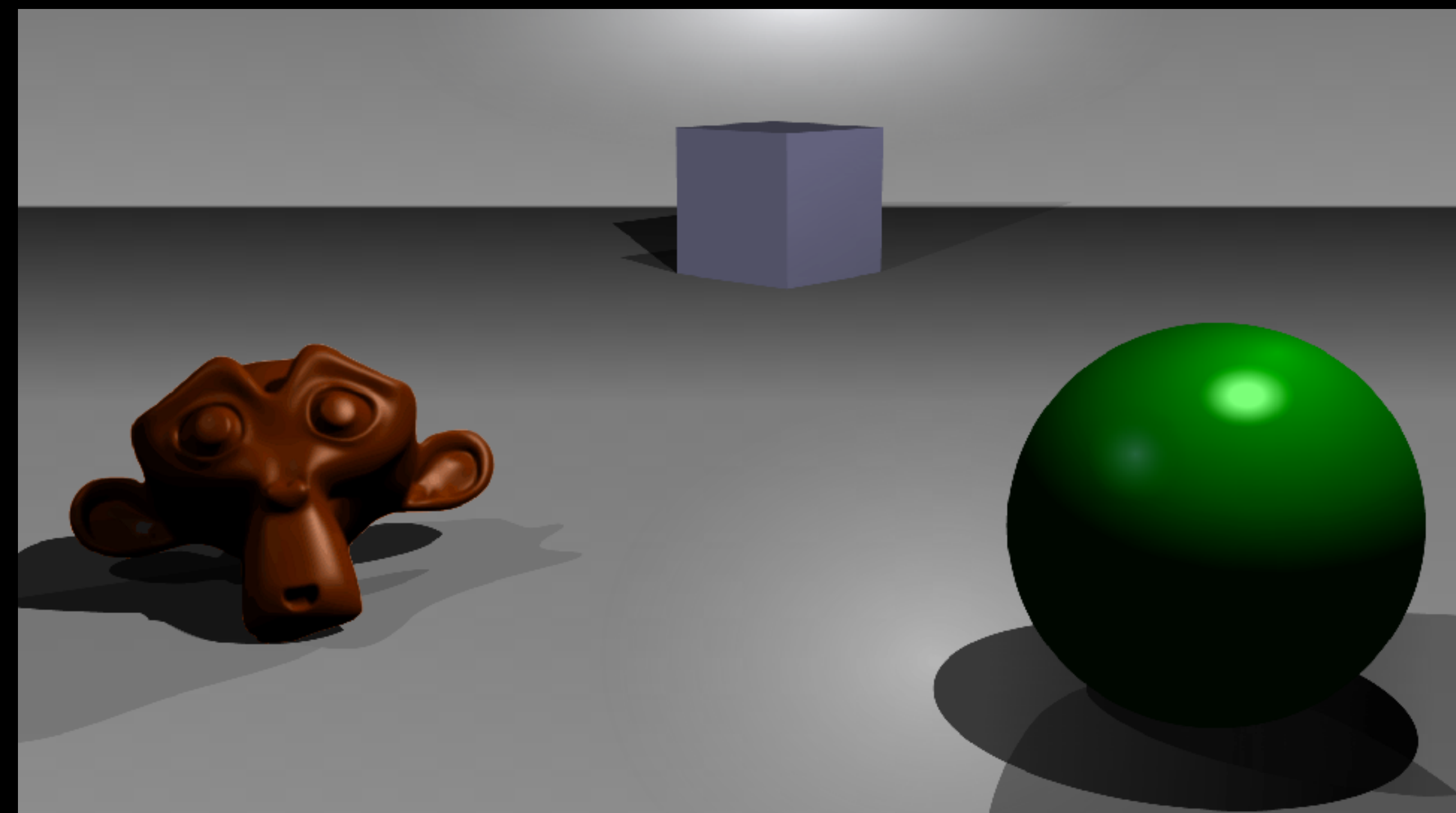
Painter's algorithm.



???



Z-Buffer



Enabling the Z-Buffer in OpenGL.


```
void glDepthMask (GLboolean flag);
```

Enables or disables depth buffer writing.

```
glDepthMask(GL_TRUE); // enable depth write  
glDepthMask(GL_FALSE); // disable depth write
```

```
void glEnable (GLenum capability);  
void glDisable (GLenum capability);
```

Enable or disable an OpenGL capability. Use GL_DEPTH_TEST to enable or disable depth test capability.

```
glEnable(GL_DEPTH_TEST); // enable depth testing  
glDisable(GL_DEPTH_TEST); // disable depth testing
```

```
void glDepthFunc (GLenum func);
```

Specifies the function used to compare each incoming pixel depth value with the depth value present in the depth buffer.

Can be one of the following:

GL_NEVER, GL_LESS, GL_EQUAL, GL_LEQUAL, GL_GREATER, GL_NOTEQUAL, GL_GEQUAL, and GL_ALWAYS

For example if the function is **GL_LEQUAL**, the pixel will only be drawn if its depth is **LESS THAN OR EQUAL** than the pixel's depth on the screen (most of the time we want this).

```
glDepthFunc(GL_LEQUAL); // only draw pixels that are closer or equal
```

Putting it together.

Enabling depth testing/writing for 3D.

```
glEnable(GL_DEPTH_TEST);  
glDepthFunc(GL_LEQUAL);  
glDepthMask(GL_TRUE);
```

Disabling depth testing and writing.

```
glDisable(GL_DEPTH_TEST);  
glDepthMask(GL_FALSE);
```

Clearing the depth buffer!

If using the depth buffer, we need to tell the OpenGL clear function to clear the depth buffer as well as the color buffer!

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Texturing in 3D

Same as 2D, but be careful of shared vertices with indexed drawing.

For example, you cannot texture an indexed 8-vertex cube because the corner vertices are shared but require differing texture coordinates.

```
float boxW = 1.0f;
float boxH = 1.0f;
float boxD = 1.0f;

std::vector<float> vertices;
std::vector<float> uvs;

vertices.insert(vertices.begin(), {
    -boxW, boxH, -boxD,
    -boxW, boxH, boxD,
    boxW, boxH,boxD,

    -boxW, boxH, -boxD,
    boxW, boxH, boxD,
    boxW, boxH, -boxD,

    boxW, -boxH, -boxD,
    boxW, -boxH, boxD,
    -boxW, -boxH,boxD,

    boxW, -boxH, -boxD,
    -boxW, -boxH, boxD,
    -boxW, -boxH, -boxD,

    -boxW, boxH, -boxD,
    -boxW, -boxH, -boxD,
    -boxW, -boxH, boxD,

    -boxW, boxH, -boxD,
    -boxW, -boxH, boxD,
    -boxW, boxH, boxD,

    boxW, boxH, boxD,
    boxW, -boxH, boxD,
    boxW, -boxH, -boxD,

    boxW, boxH, boxD,
    boxW, -boxH, -boxD,
    boxW, boxH, -boxD,

    -boxW, boxH, boxD,
    -boxW, -boxH, boxD,
    boxW, -boxH, boxD,

    -boxW, boxH, boxD,
    boxW, -boxH, boxD,
    boxW, boxH, boxD,

    boxW, boxH, -boxD,
    boxW, -boxH, -boxD,
    -boxW, -boxH, -boxD,

    boxW, boxH, -boxD,
    -boxW, -boxH, -boxD,
    -boxW, boxH, -boxD,
});
```

```
uvs.insert(uvs.begin(), {
    0.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f,

    0.0f, 0.0f,
    1.0f, 1.0f,
    1.0f, 0.0f,

    0.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f,

    0.0f, 0.0f,
    1.0f, 1.0f,
    1.0f, 0.0f,

    0.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f,

    0.0f, 0.0f,
    1.0f, 1.0f,
    1.0f, 0.0f,

    0.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f,

    0.0f, 0.0f,
    1.0f, 1.0f,
    1.0f, 0.0f,

    0.0f, 0.0f,
    0.0f, 1.0f,
    1.0f, 1.0f,

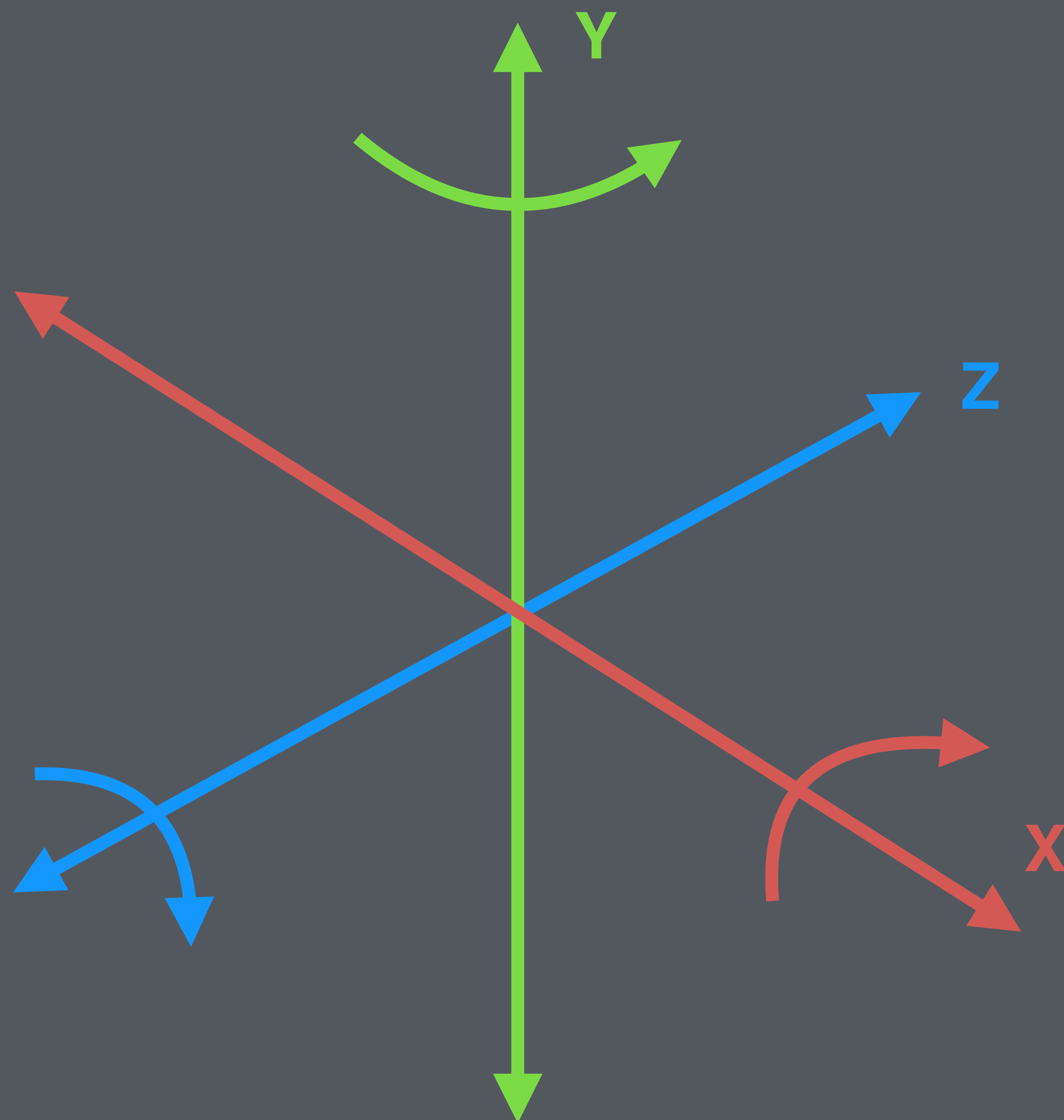
    0.0f, 0.0f,
    1.0f, 1.0f,
    1.0f, 0.0f,
});
```

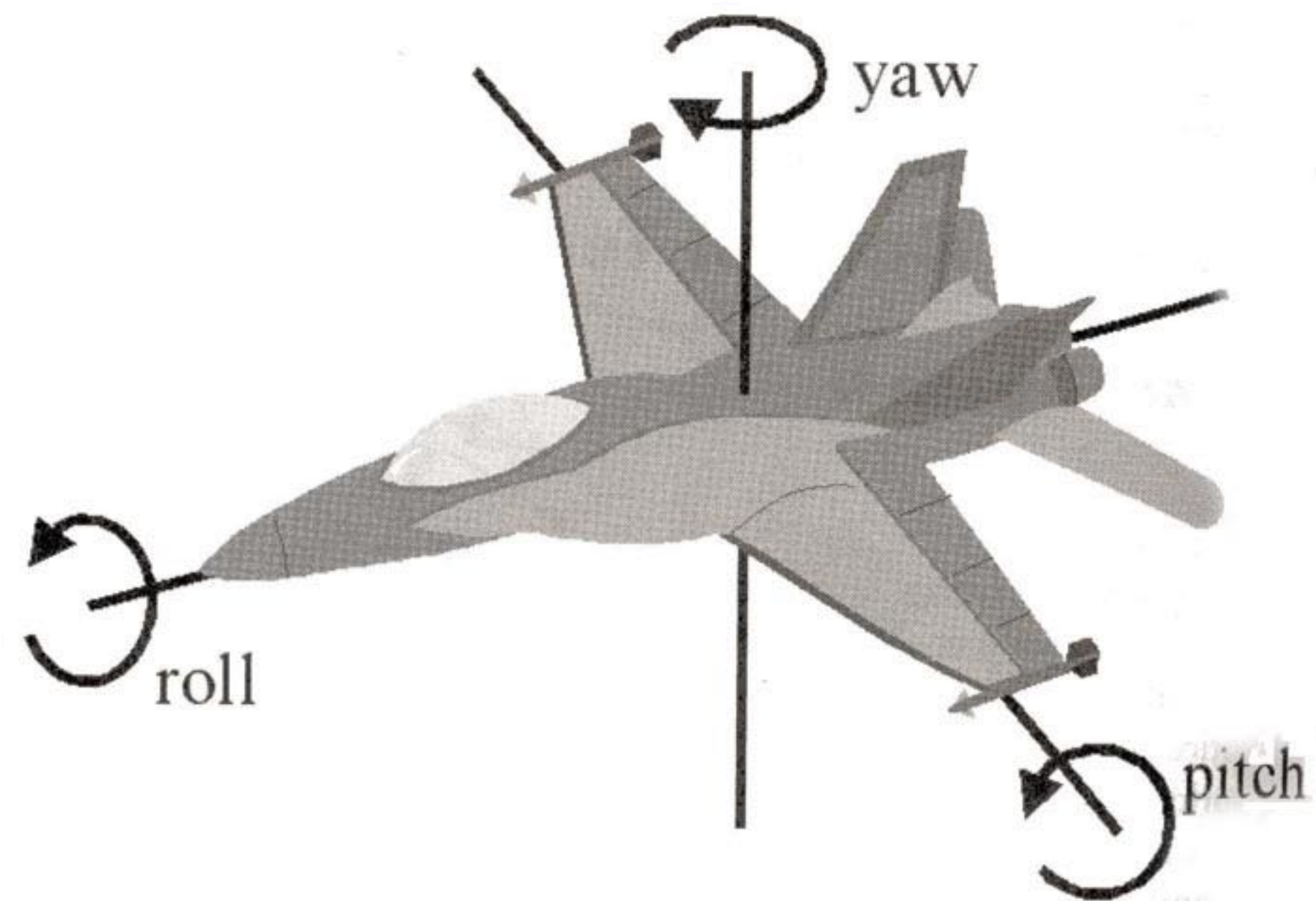
Textured cube position and texture coordinate data.

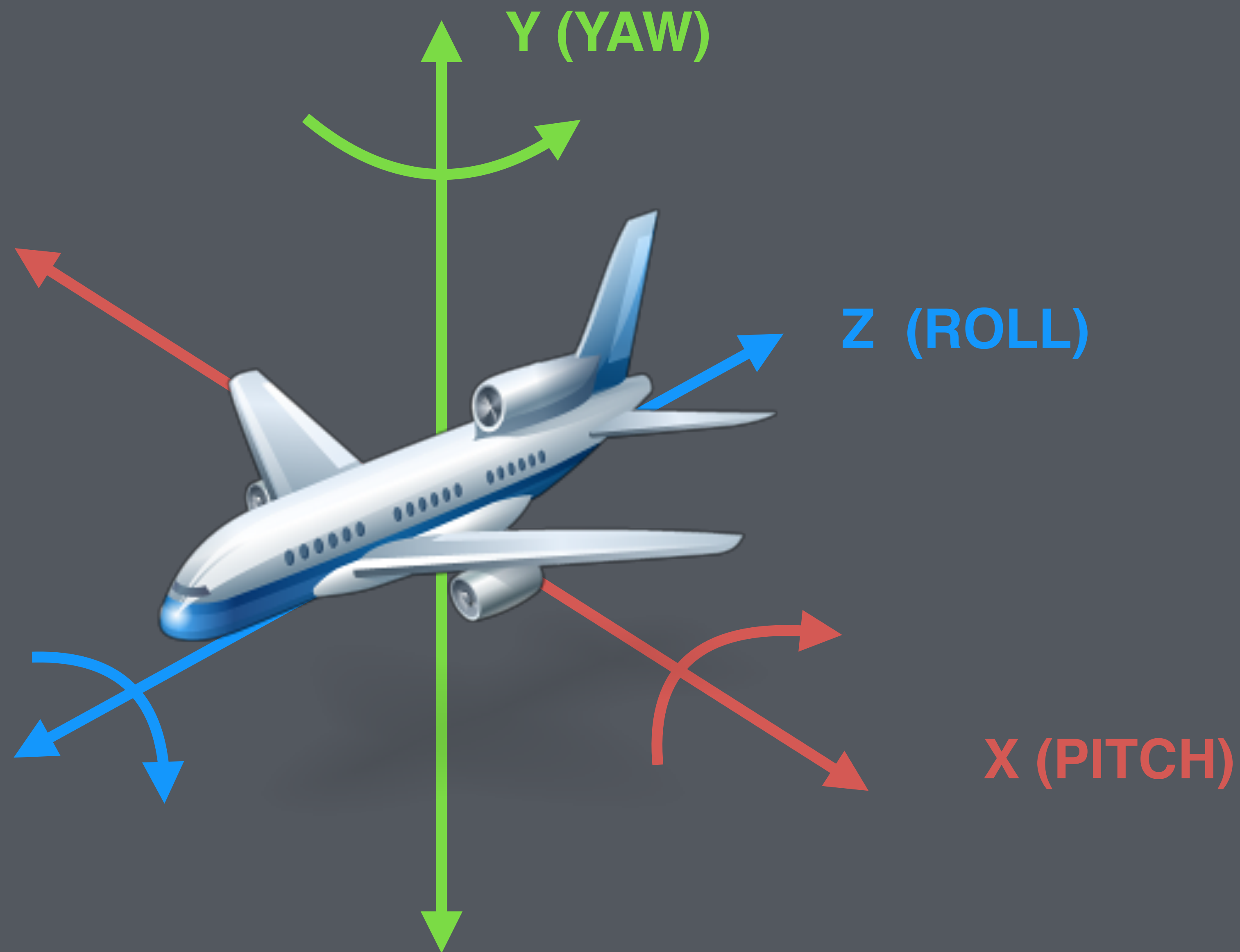


3D Entities

3 degrees of rotation.







```
class Entity {  
public:  
  
    Entity();  
    void Update(float elapsed);  
    void Render();  
    void buildMatrix();  
  
    glm::mat4 matrix;  
  
    glm::vec3 position;  
    glm::vec3 rotation;  
    glm::vec3 scale;  
  
    glm::vec3 velocity;  
    glm::vec3 acceleration;  
  
    unsigned int texture;  
    std::vector<float> vertices;  
    std::vector<float> uvs;  
  
    bool visible;  
    float friction;  
  
};
```

Extending our entity class
into the **3rd dimension**.

$$\begin{array}{c}
 \text{X-Rotation in 3D} \\
 \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{array}
 \begin{array}{c}
 \text{Z-Rotation in 3D} \\
 \begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{array}
 \begin{array}{c}
 \text{Scale in 3D} \\
 \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{array}
 (4 \times 4) * (4 \times 1) = (4 \times 1)$$

$$\begin{array}{c}
 \text{Y-Rotation in 3D} \\
 \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{array}
 \begin{array}{c}
 \text{Translation in 3D} \\
 \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{array}
 \begin{array}{c}
 \text{Matrix Multiplication} \\
 \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}
 \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ q \end{bmatrix}
 \end{array}$$

Diagram illustrating the composition of a Rotate Matrix:

$$\text{ROTATE MATRIX} = \text{ROTATE X MATRIX} * \text{ROTATE Y MATRIX} * \text{ROTATE Z MATRIX}$$

The diagram shows a sequence of four gray rectangular boxes. The first box on the left is wider than the others and contains the text "ROTATE MATRIX". To its right is an equals sign. This is followed by three boxes of equal width, each containing two lines of text: "ROTATE X MATRIX", "ROTATE Y MATRIX", and "ROTATE Z MATRIX". Asterisks are placed between the second, third, and fourth boxes to indicate matrix multiplication.

Diagram illustrating the composition of a Final Matrix:

$$\text{FINAL MATRIX} = \text{SCALE MATRIX} * \text{ROTATE MATRIX} * \text{TRANSLATE MATRIX}$$

The diagram shows a sequence of four gray rectangular boxes. The first box on the left is wider than the others and contains the text "FINAL MATRIX". To its right is an equals sign. This is followed by three boxes of equal width, each containing two lines of text: "SCALE MATRIX", "ROTATE MATRIX", and "TRANSLATE MATRIX". Asterisks are placed between the second, third, and fourth boxes to indicate matrix multiplication.

3D Camera

Our view matrix needs to be
the inverse of the camera entity's matrix!

Mixing 2D and 3D

Rendering 3D, then 2D on top of it.

- Clear color and depth buffers
- Set perspective projection in projection matrix.
- Enable depth testing and writing
- Draw 3D scene
- Set orthographic projection in projection matrix
- Disable depth testing and writing
- Render 2D scene