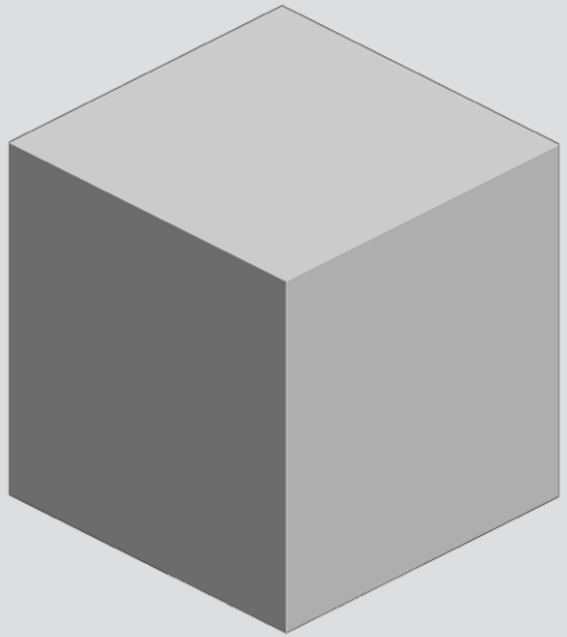
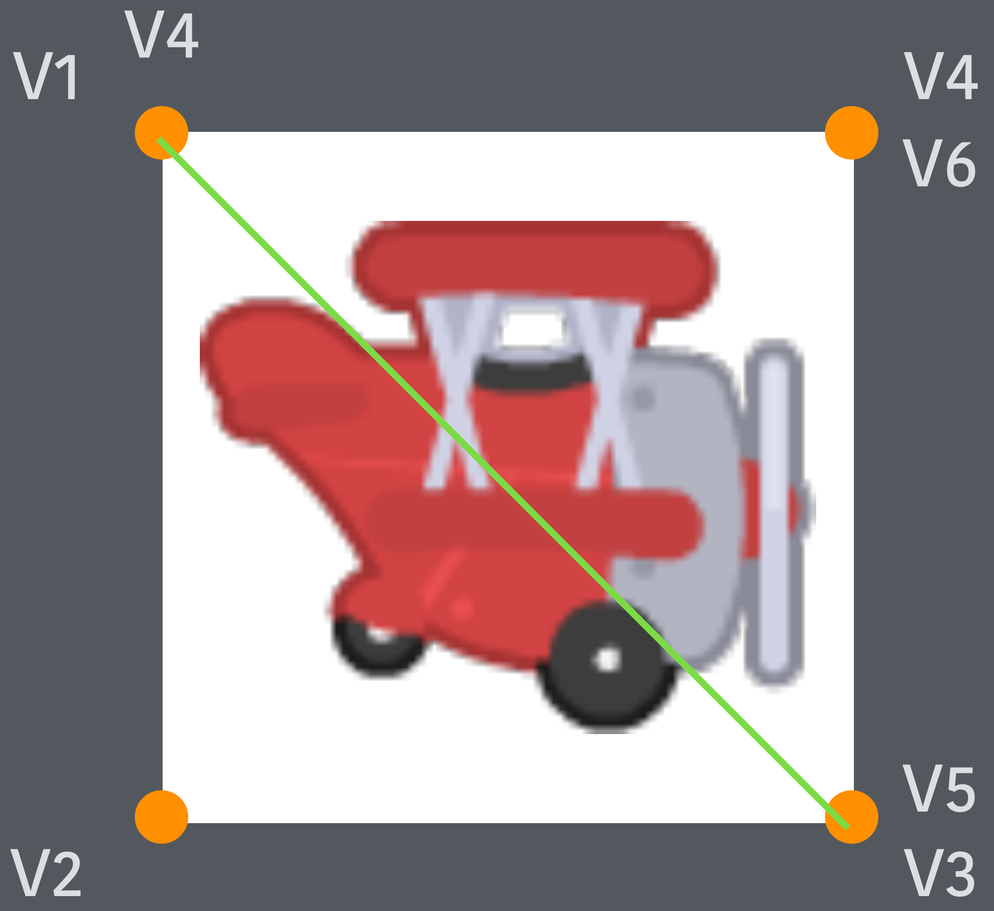
3D Graphics

Part 1

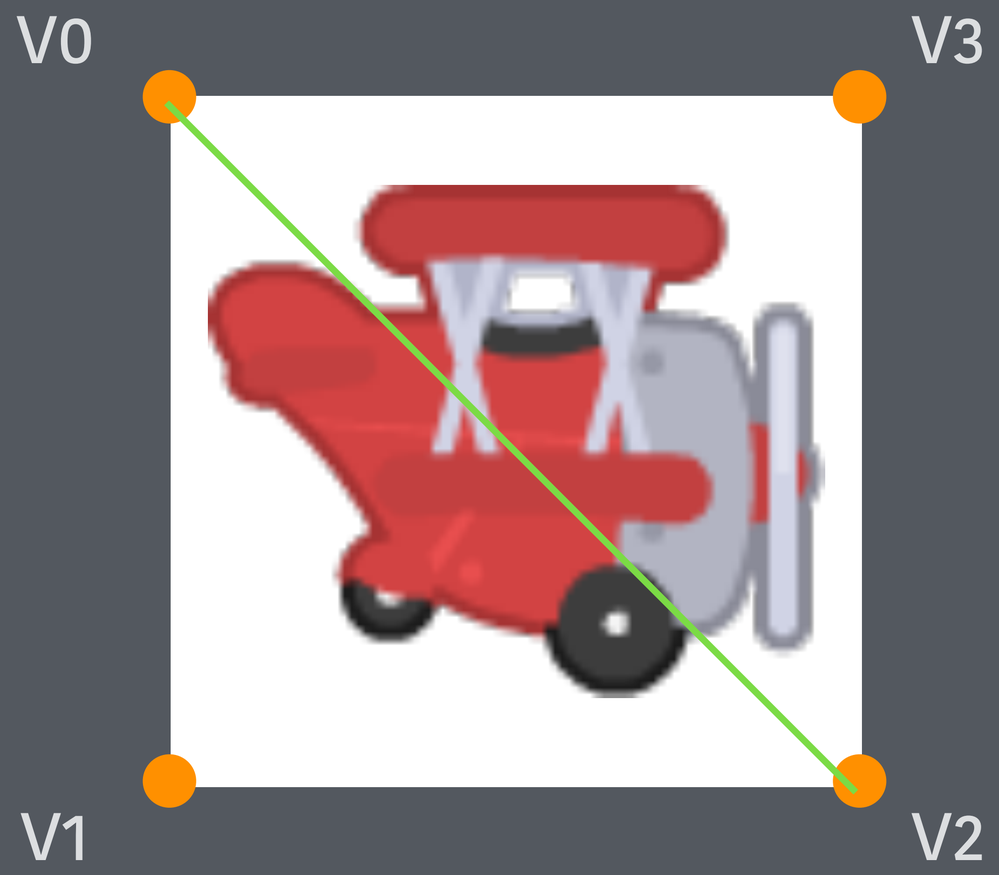


Indexed drawing.

GL_TRIANGLES



GL_TRIANGLES



V0,V1,V2

V0,V2,V3

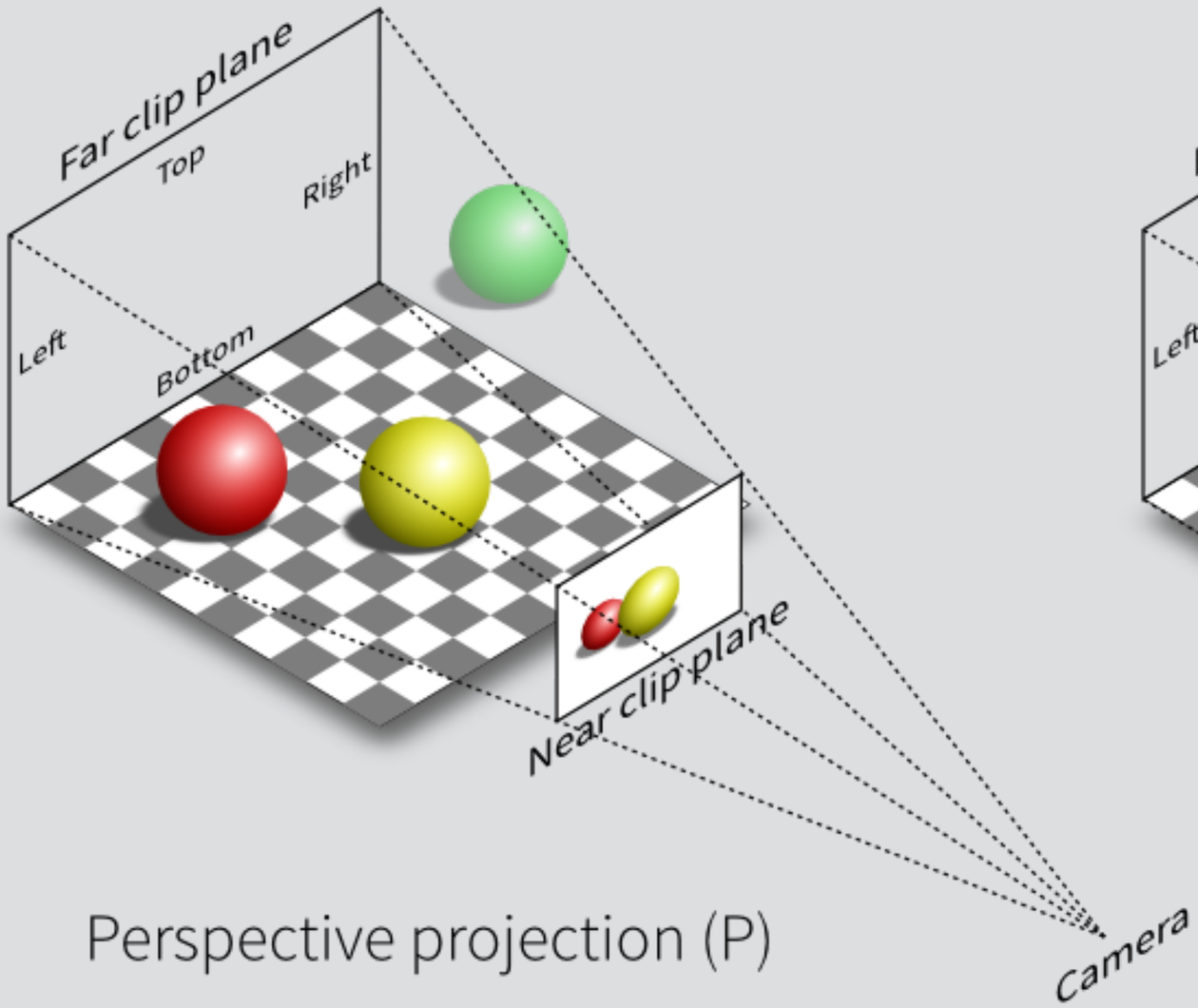
```
void glDrawElements (GLenum mode, GLsizei
count, GLenum type, const GLvoid *indices);
```

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

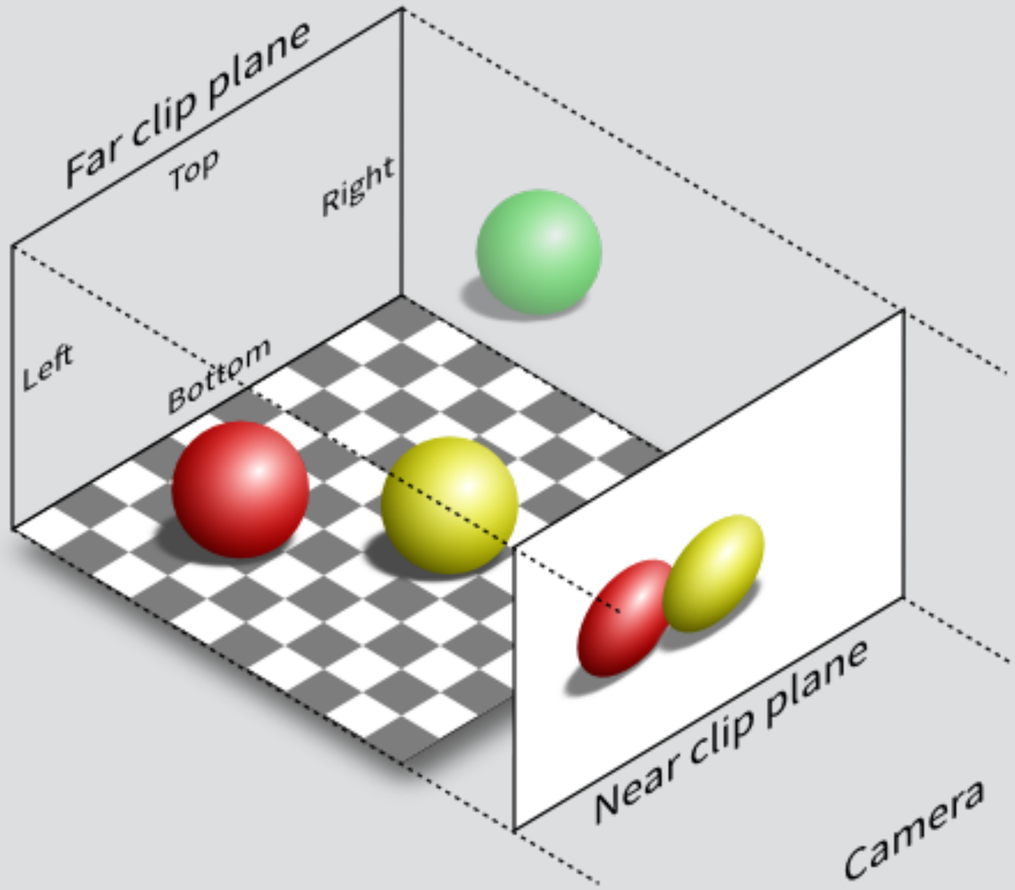
```
std::vector<unsigned int> indices = {0,1,2,0,2,3};
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, indices.data());
```

Perspective

Perspective vs. **Orthographic** projection.



Perspective projection (P)

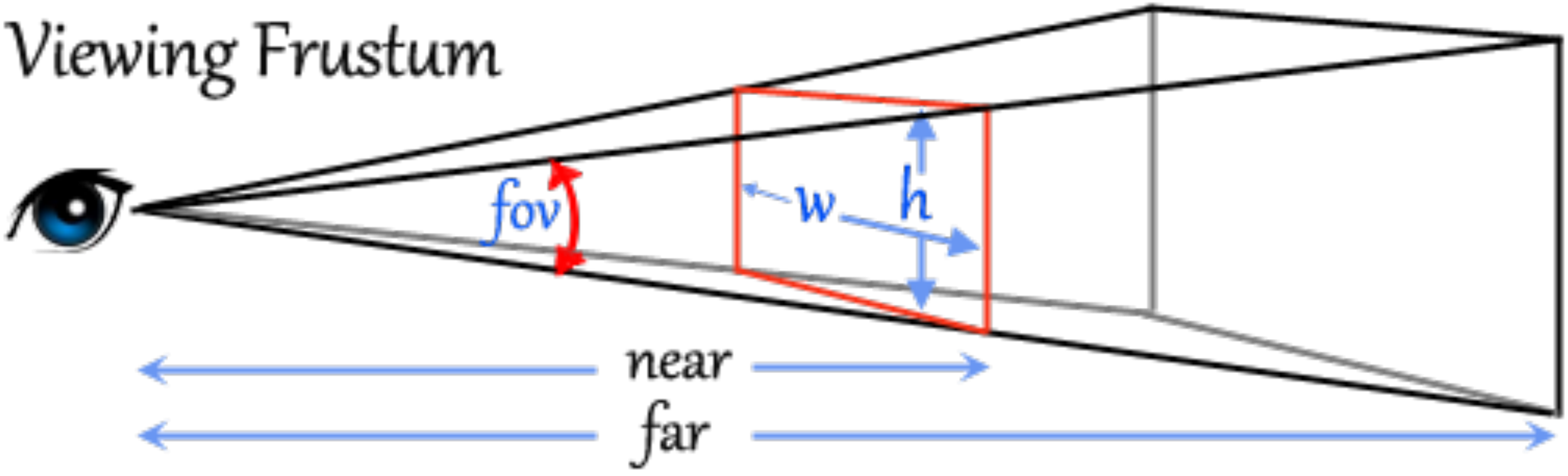


Orthographic projection (O)

Orthographic projection matrix.

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

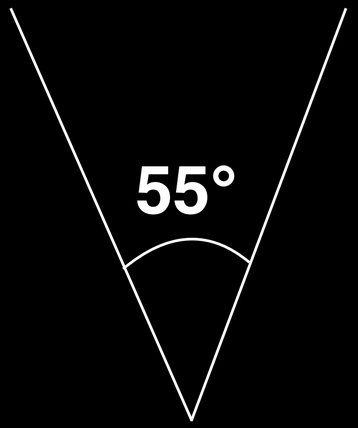
Building a **perspective** projection matrix.

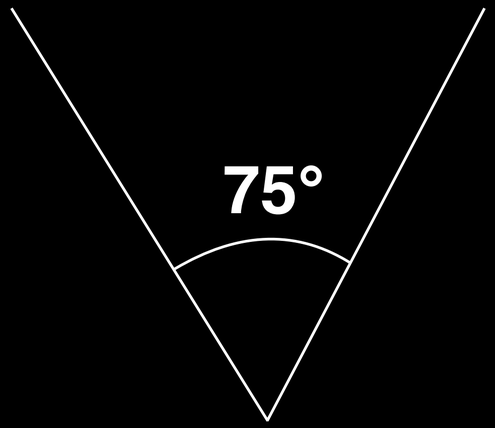

Viewing Frustum
fov
w
h
near
far

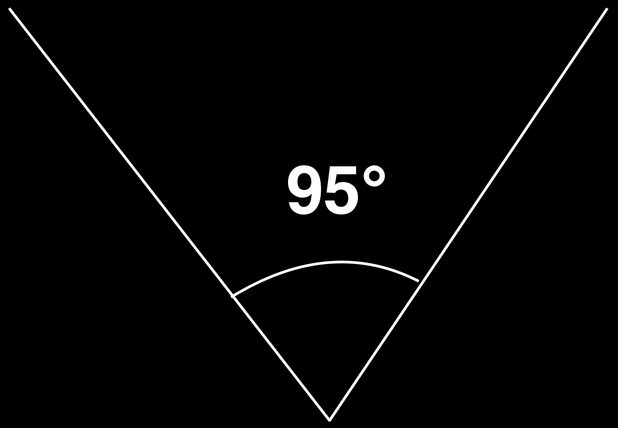

55
030


75
030


95
030


55°


75°


95°

Perspective projection matrix.

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

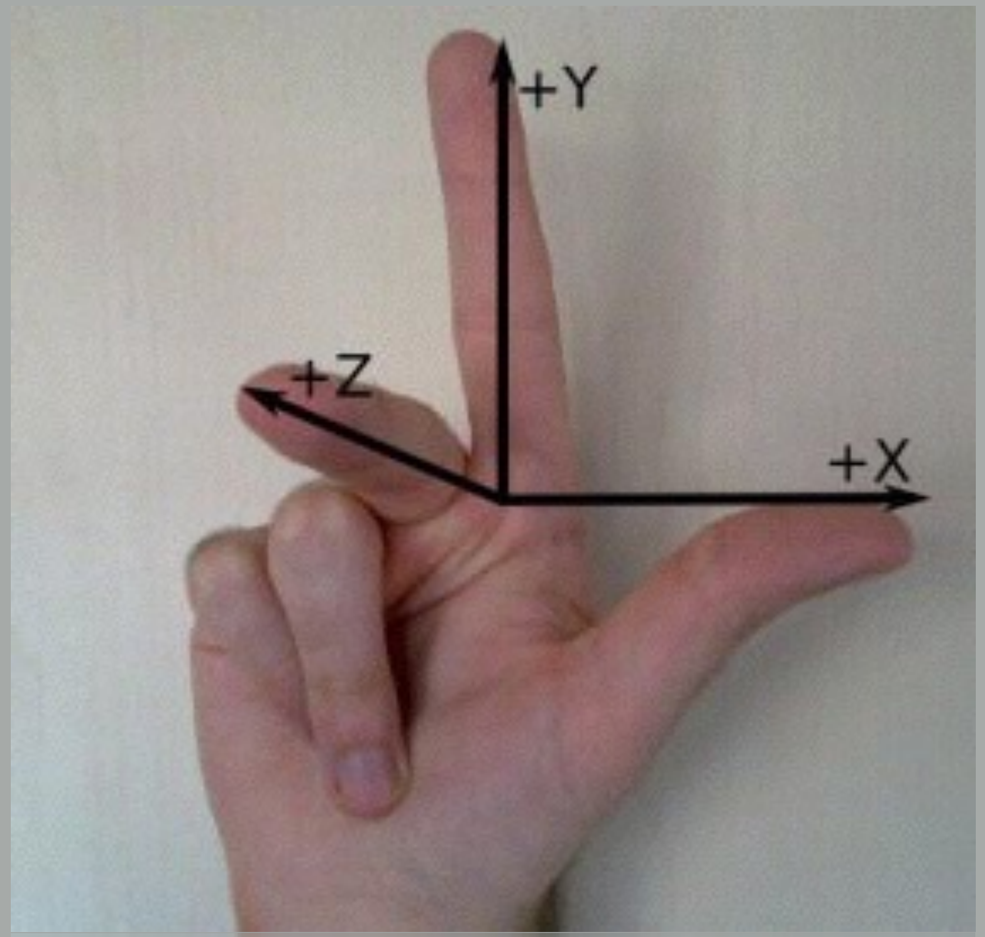
```
void Matrix::setPerspectiveProjection(float fov, float aspect,
float zNear, float zFar) {

    m[0][0] = 1.0f/tanf(fov/2.0)/aspect;
    m[1][1] = 1.0f/tanf(fov/2.0);
    m[2][2] = (zFar+zNear)/(zNear-zFar);
    m[3][2] = (2.0f*zFar*zNear)/(zNear-zFar);
    m[2][3] = -1.0f;
    m[3][3] = 0.0f;
}
```

Drawing in 3D

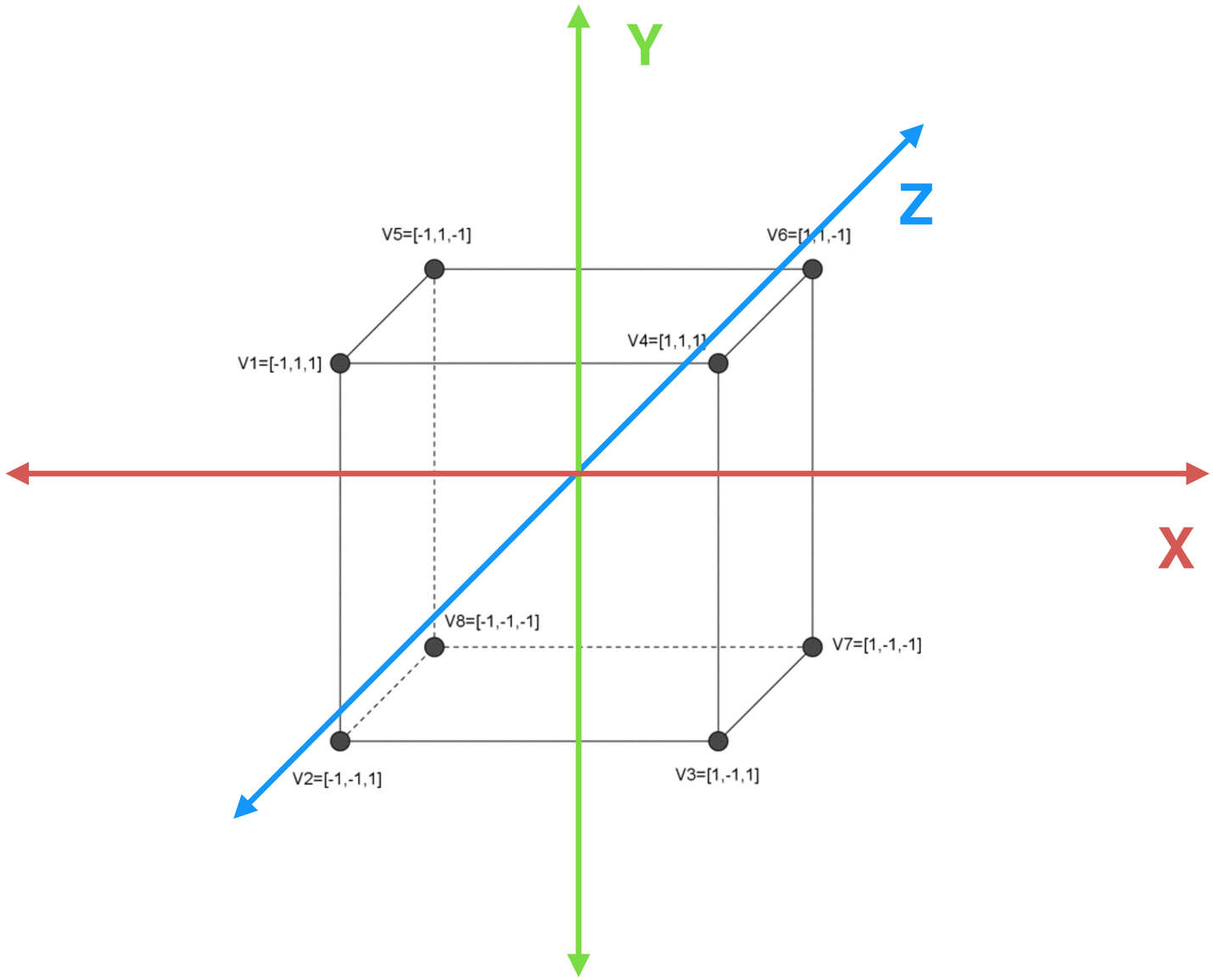
It’s the same as drawing in 2D!

We just need to specify **3 dimensions per vertex**!

Right-handed coordinate system.

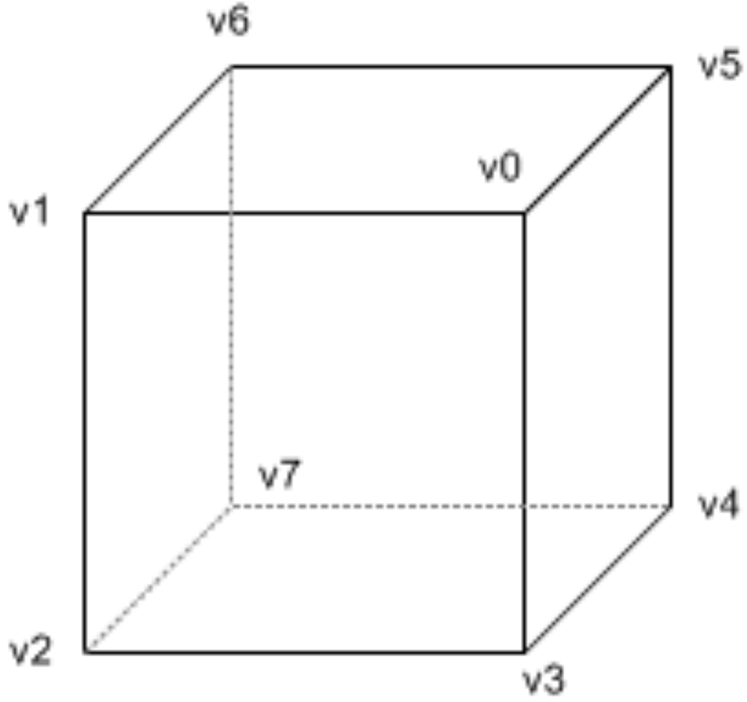


Drawing a **cube**.


Y
Z
X
V5=[-1,1,-1]
V6=[1,1,-1]
V4=[1,1,1]
V1=[-1,1,1]
V8=[-1,-1,-1]
V7=[1,-1,-1]
V2=[-1,-1,1]
V3=[1,-1,1]

```
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);
```

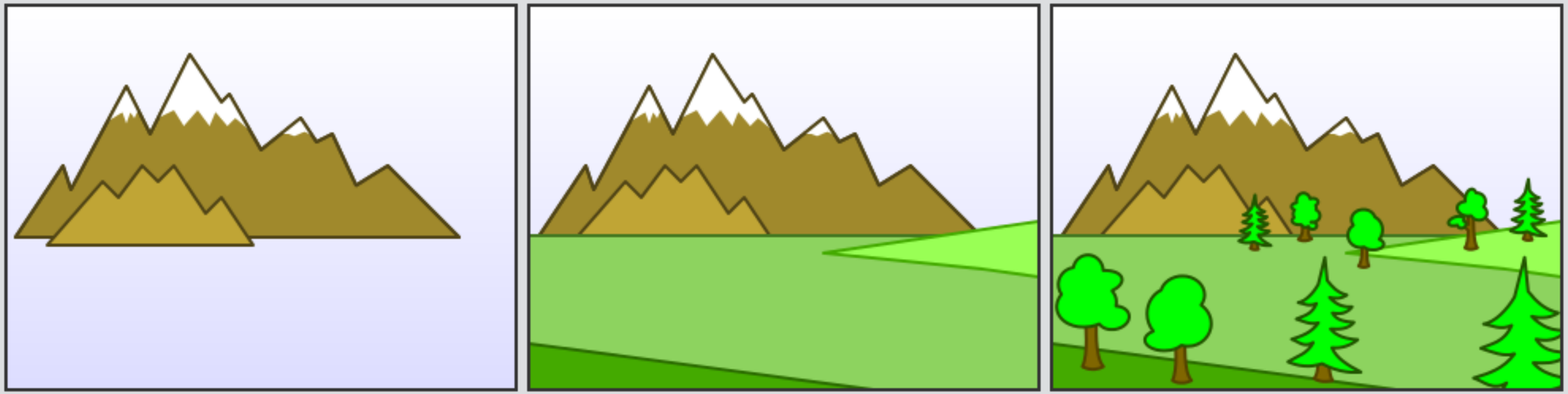
3 DIMENSIONS PER VERTEX



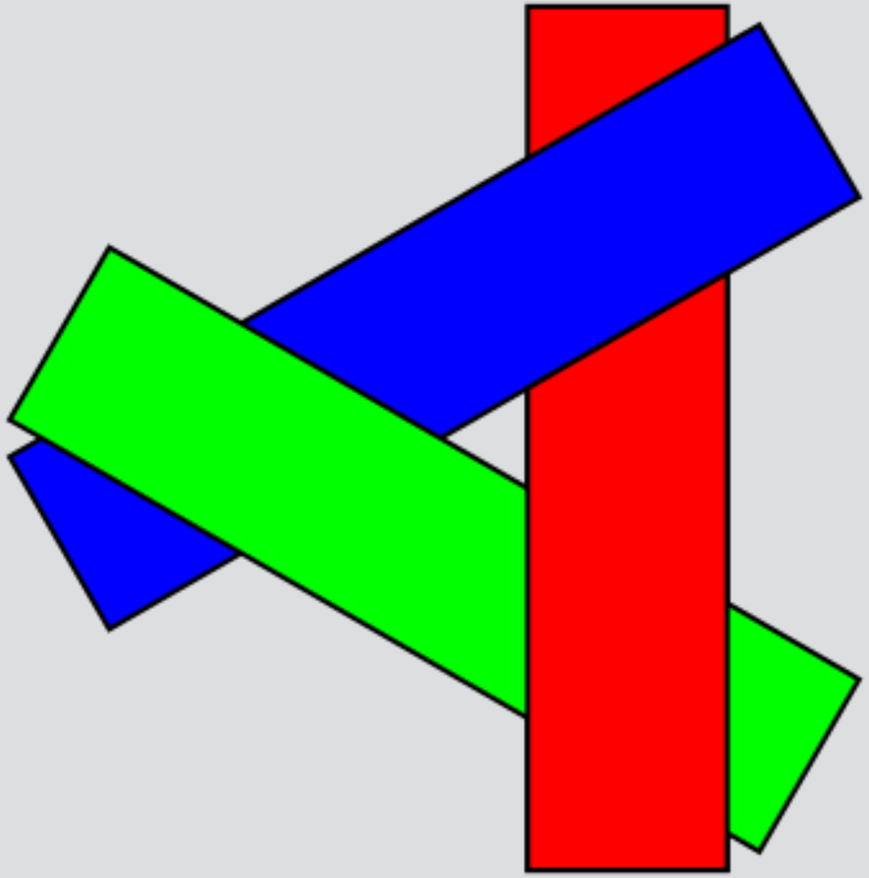
3 DIMENSIONS PER VERTEX

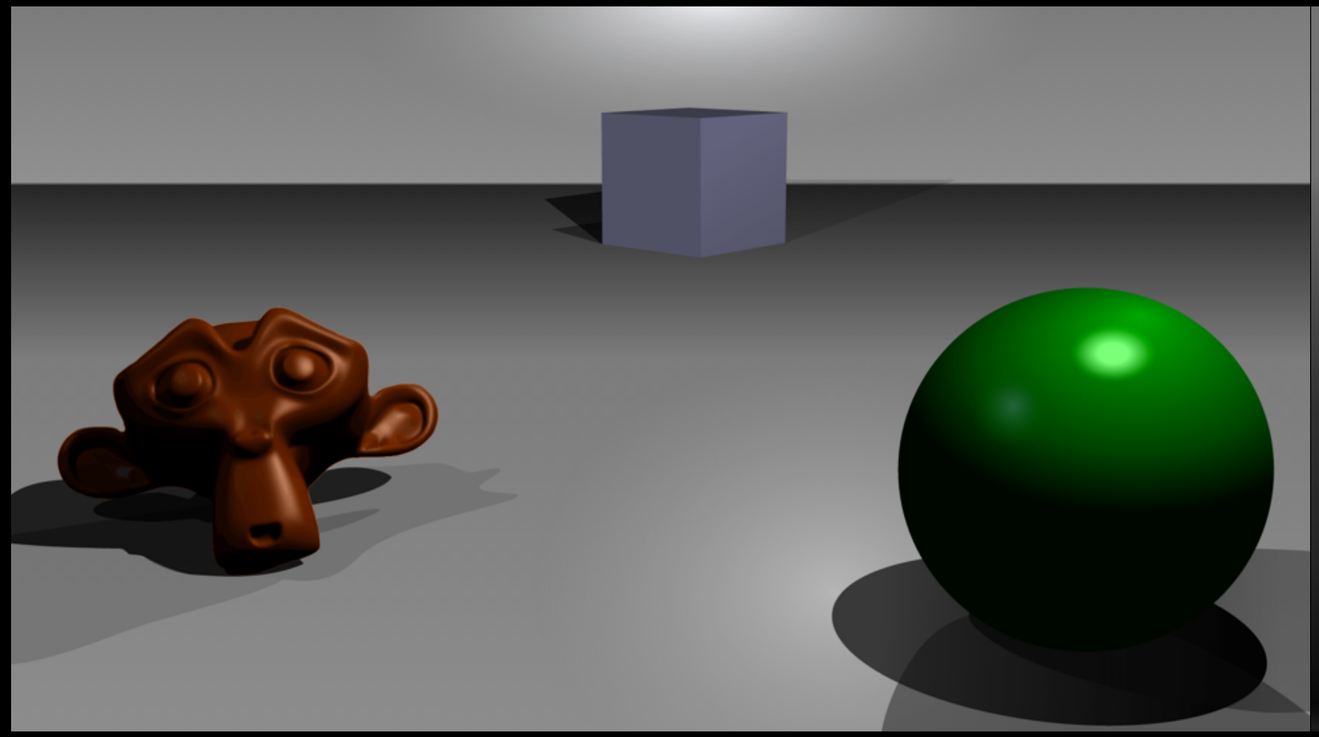
Z-Buffer

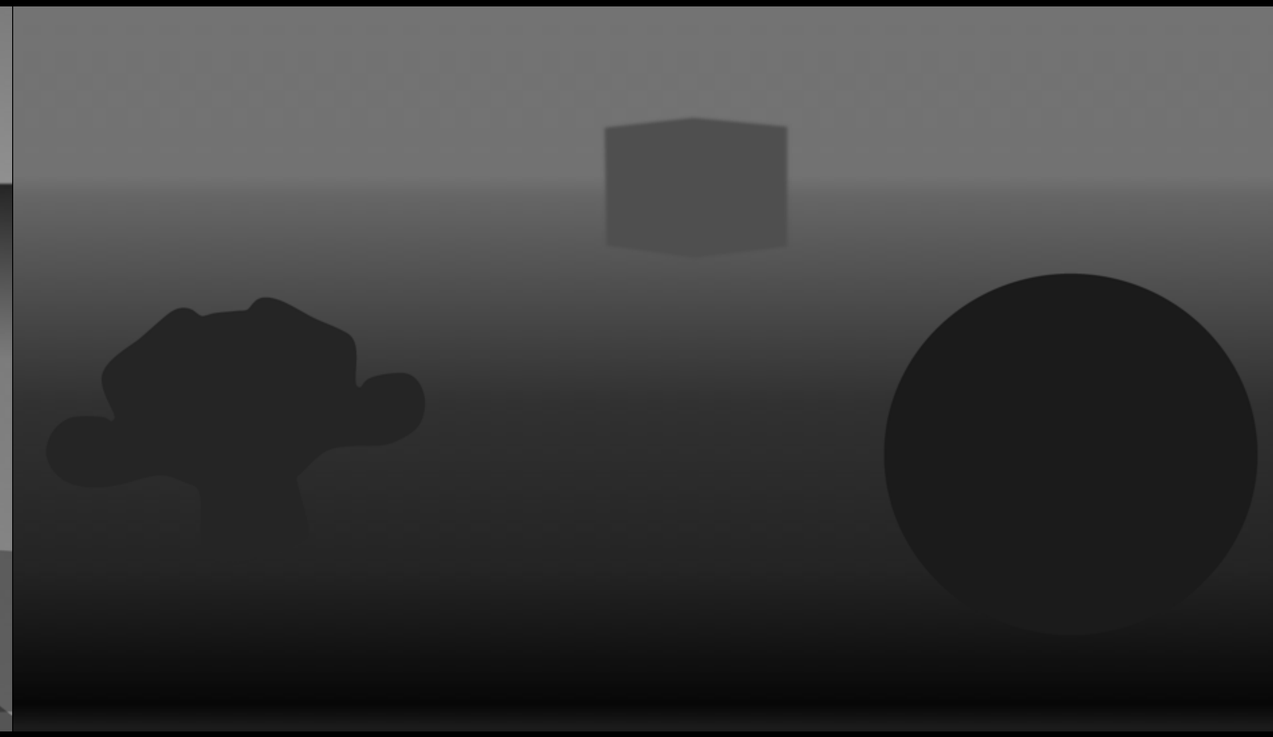
Painter's algorithm.



???







Enabling the **Z-Buffer** in **OpenGL**.

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

Enabes 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!

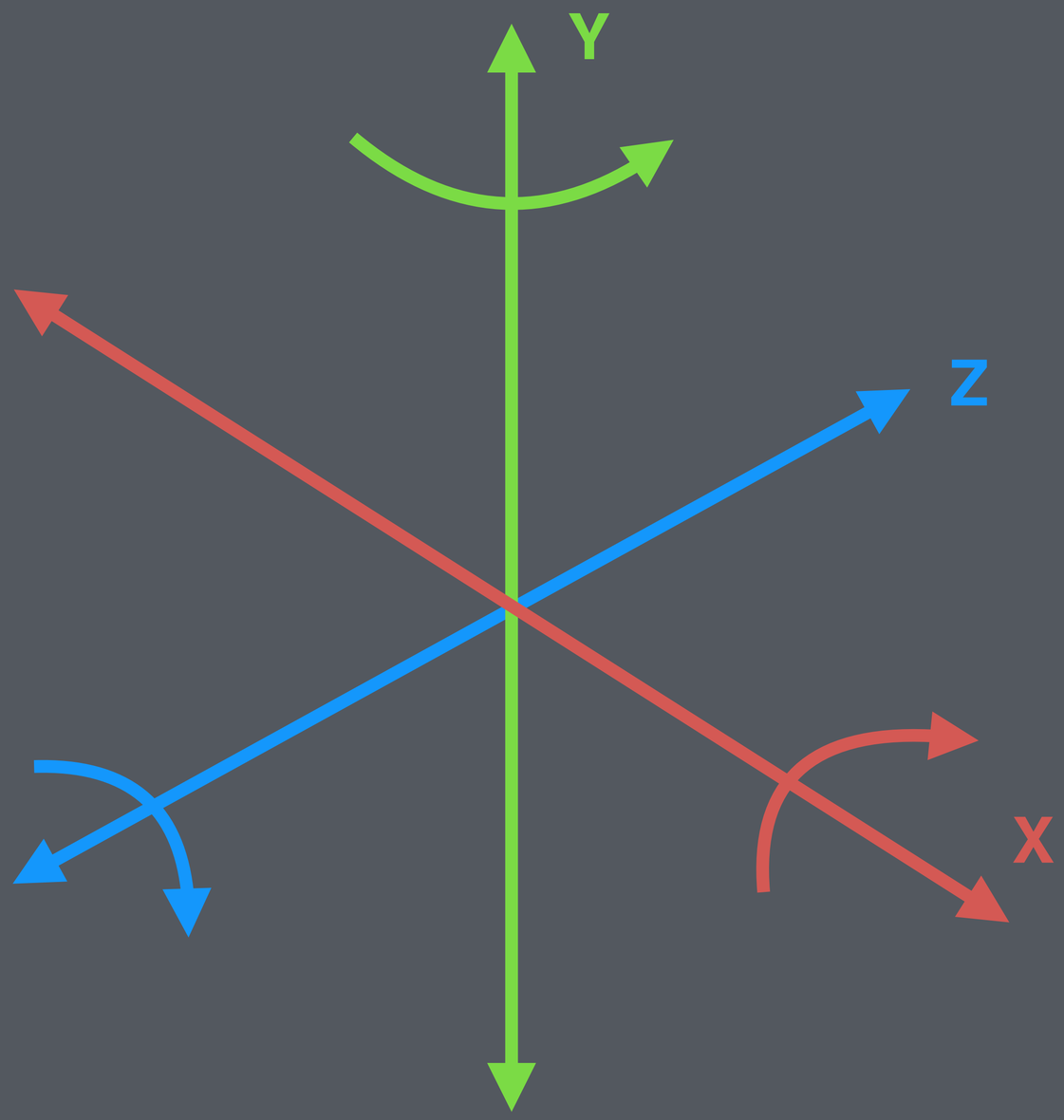
If using the depth buffer, we need to tell the OpenGL clear function to clear the depth buffer as well as the color buffer!

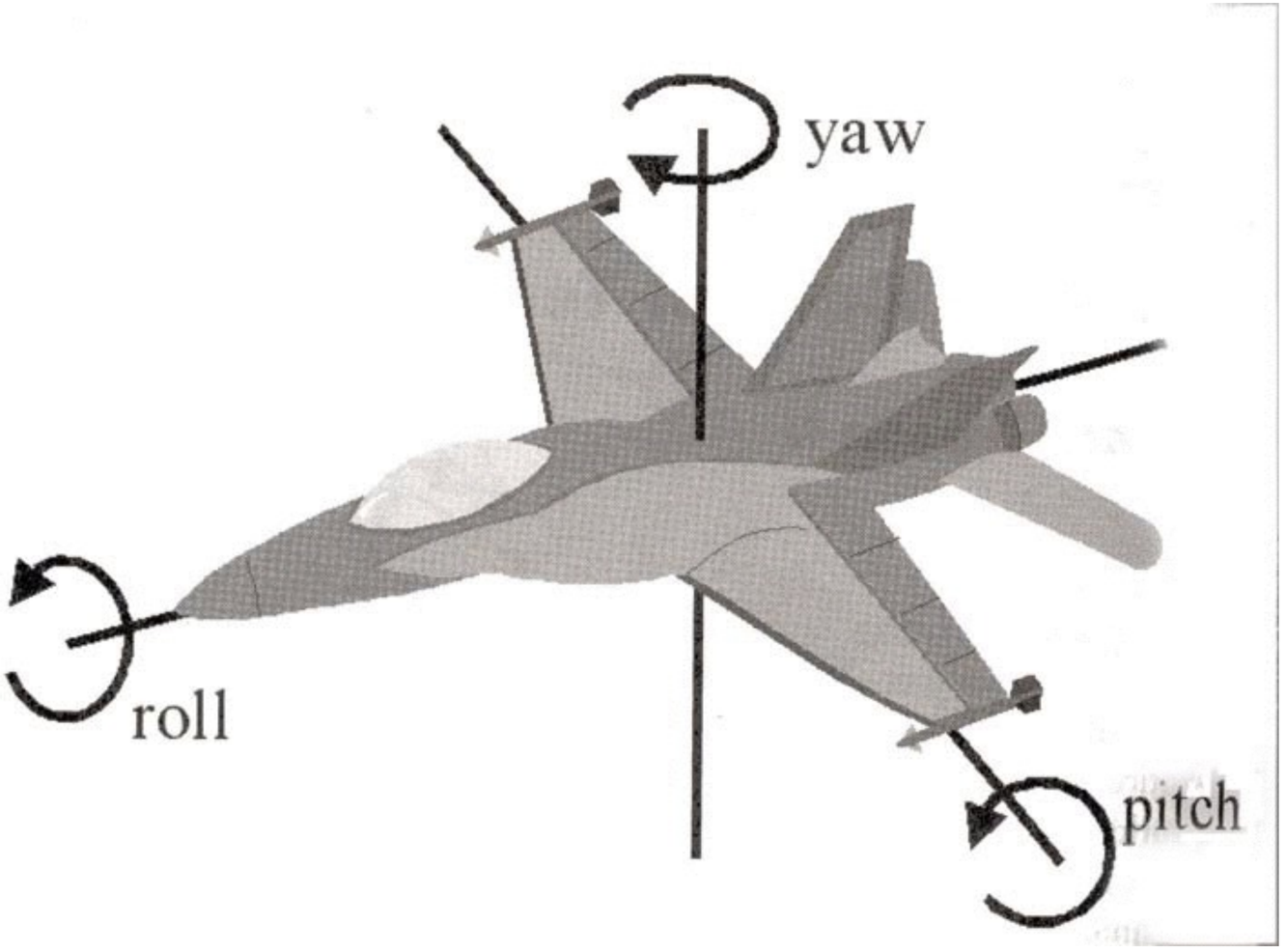
```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

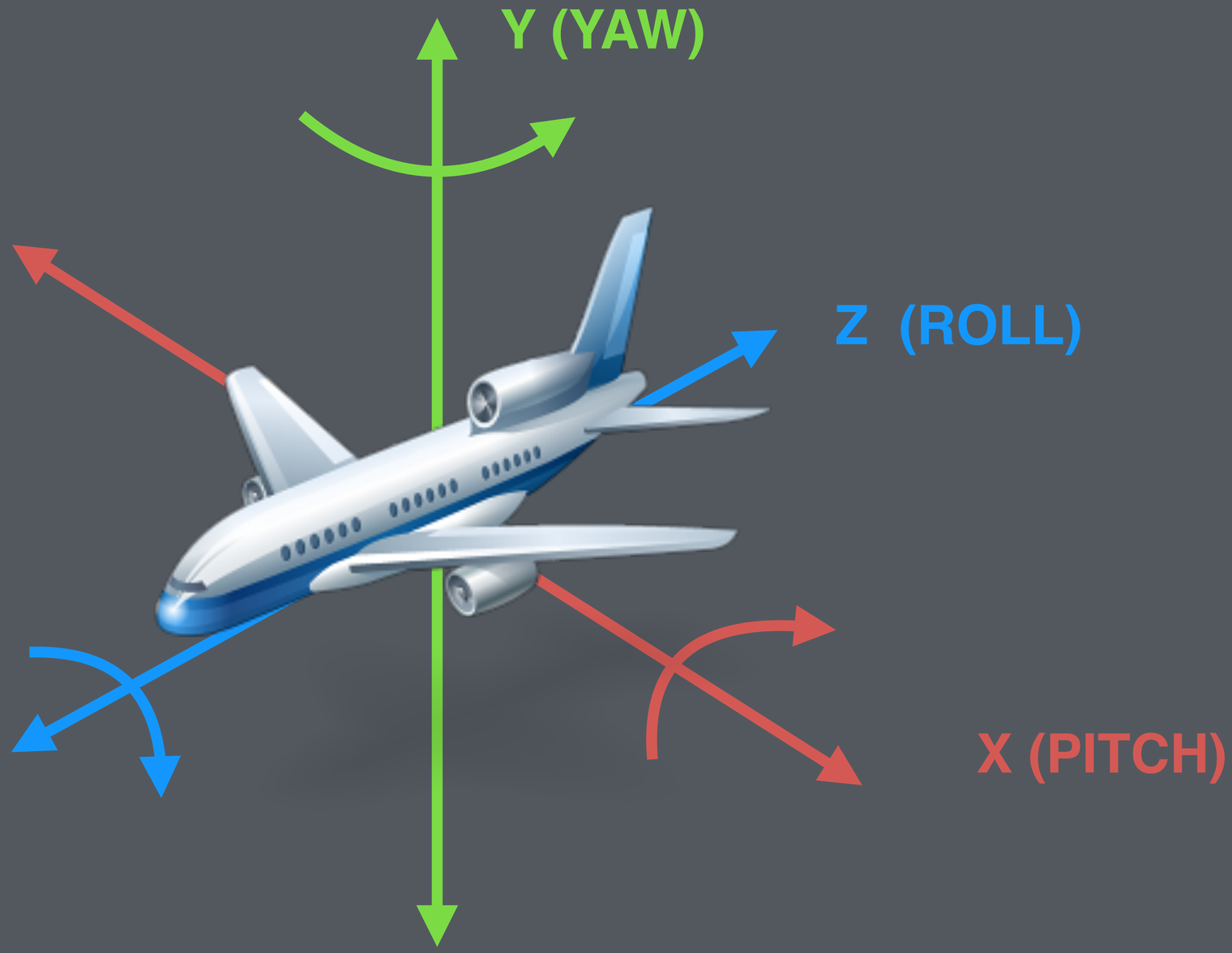
Texturing in 3D

3D **Entities**

3 degrees of rotation.


Y
Z
X


yaw
roll
pitch


Y (YAW)
Z (ROLL)
X (PITCH)

```
class Entity {
public:

    Entity();
    void Update(float elapsed);
    void Render();
    void buildMatrix();

    void FixedUpdate();

    Matrix matrix;

    Vector position;
    Vector rotation;
    Vector scale;

    Vector velocity;
    Vector acceleration;

    unsigned int texture;
    std::vector<float> vertices;
    std::vector<float> uvs;

    bool visible;
    float friction;

};
```

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

X-Rotation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z-Rotation in 3D

$$\begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale in 3D

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(4x4)*(4x1) = (4x1)

Y-Rotation in 3D

$$\begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix Multiplication

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ q \end{bmatrix}$$

ROTATE MATRIX = ROTATE X MATRIX * ROTATE Y MATRIX * ROTATE Z MATRIX

FINAL MATRIX = SCALE MATRIX * ROTATE MATRIX * TRANSLATE MATRIX

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