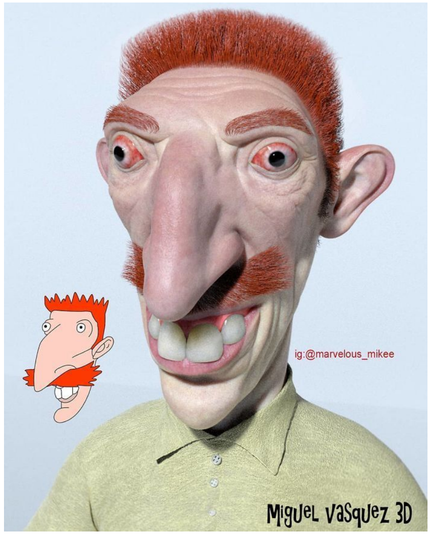
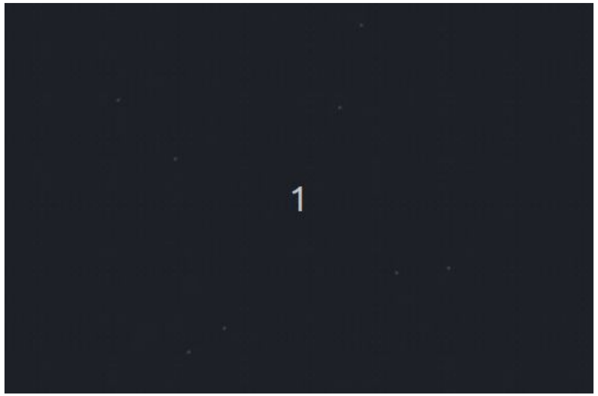
3D Graphics Programming from scratch in C w/ SDL

Part 2

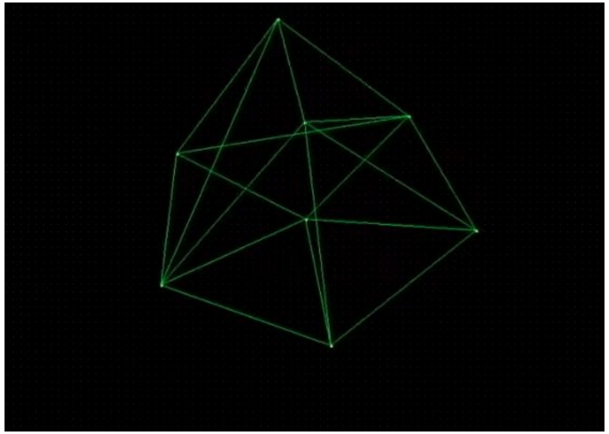


Part 1: https://docs.google.com/presentation/d/1jsQpXoaLXGmbb8Vo-ZesJN26rTRw1VaYu-ImviJOhQE/edit?usp=sharing

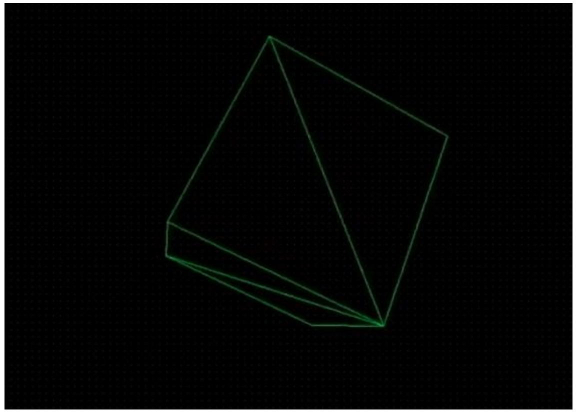
Part 1 Recap



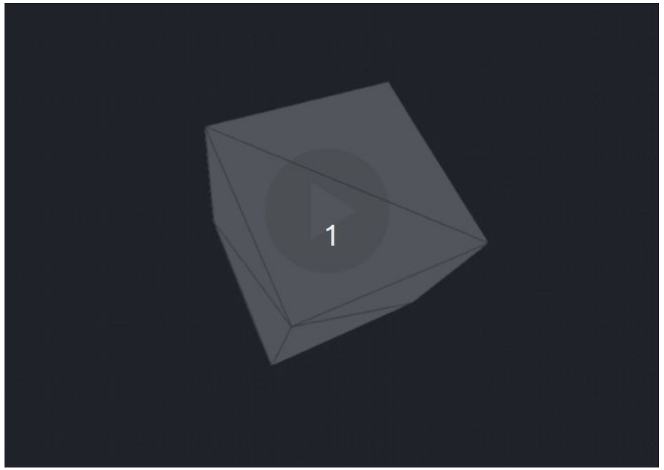
1. Vertices



2. Lines between vertices



3. Backface culling

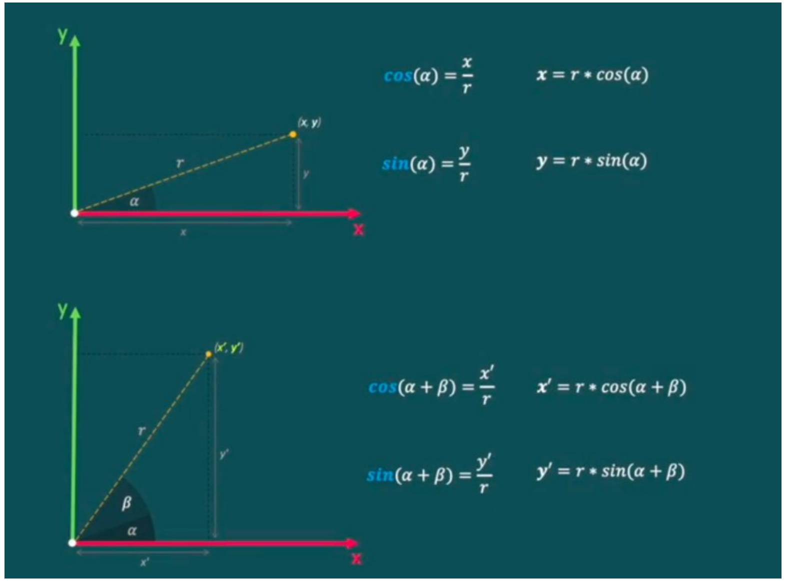


4. Filled With Horizontal White Lines

Matrix Operations

Before we can start texturing an object with an image file, the next step is to take all the naive vector manipulation operations that were being done during part 1, and turn them into matrix operations.

This trigonometry...

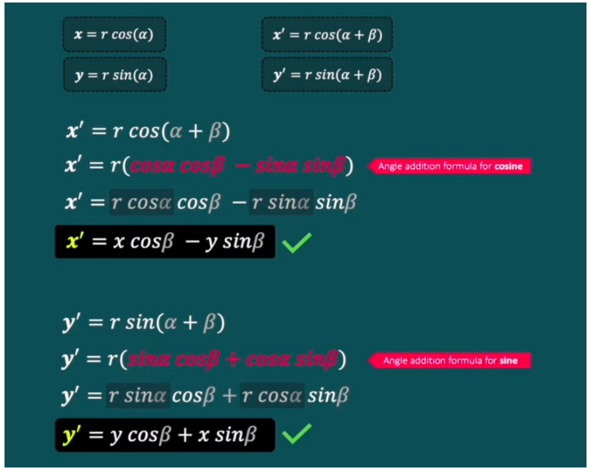


$$\cos(\alpha) = \frac{x}{r} \qquad x = r * \cos(\alpha)$$

$$\sin(\alpha) = \frac{y}{r} \qquad y = r * \sin(\alpha)$$

$$\cos(\alpha+\beta) = \frac{x'}{r} \qquad x' = r * \cos(\alpha+\beta)$$

$$\sin(\alpha+\beta) = \frac{y'}{r} \qquad y' = r * \sin(\alpha+\beta)$$



$$\boxed{x = r\cos(\alpha)} \qquad \boxed{x' = r\cos(\alpha+\beta)}$$

$$\boxed{y = r\sin(\alpha)} \qquad \boxed{y' = r\sin(\alpha+\beta)}$$

$$x' = r\cos(\alpha+\beta)$$

$$x' = r(\cos\alpha\cos\beta - \sin\alpha\sin\beta)$$ Angle addition formula for cosine

$$x' = r\cos\alpha\cos\beta - r\sin\alpha\sin\beta$$

$$\boxed{x' = x\cos\beta - y\sin\beta}$$ ✓

$$y' = r\sin(\alpha+\beta)$$

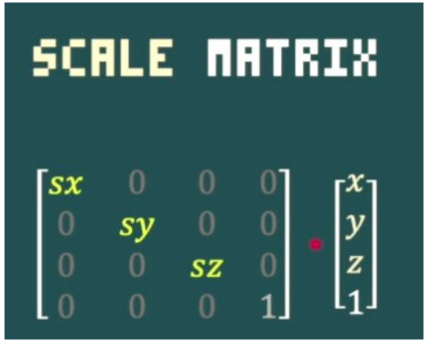
$$y' = r(\sin\alpha\cos\beta + \cos\alpha\sin\beta)$$ Angle addition formula for sine

$$y' = r\sin\alpha\cos\beta + r\cos\alpha\sin\beta$$

$$\boxed{y' = y\cos\beta + x\sin\beta}$$ ✓

Matrix Operations

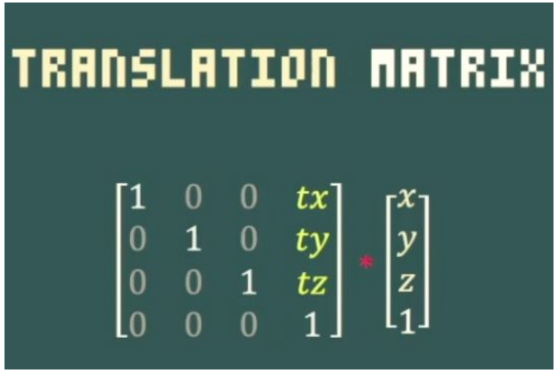
… became this matrix multiplication.



SCALE MATRIX

$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

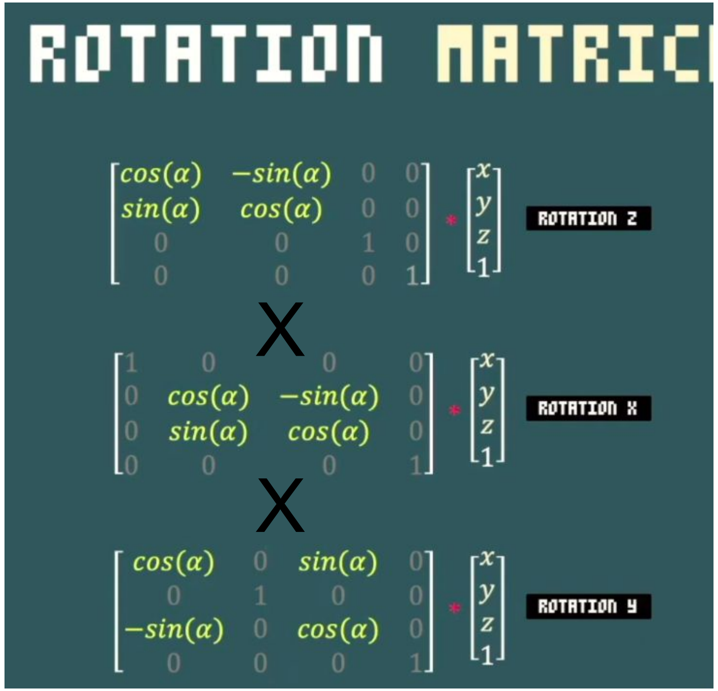
X



TRANSLATION MATRIX

$$\begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

X



ROTATION MATRIC

$$\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$ ROTATION Z

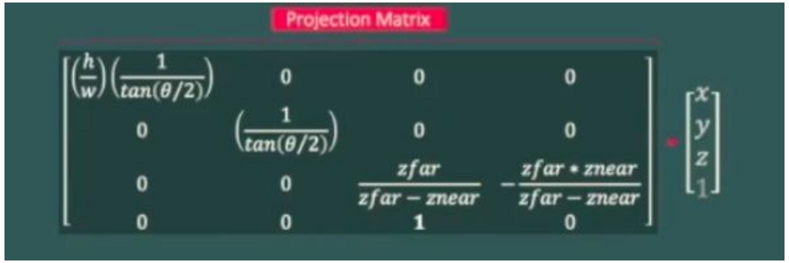
X

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$ ROTATION X

X

$$\begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$ ROTATION Y

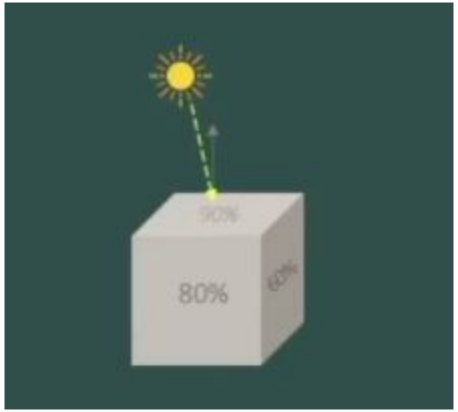
X

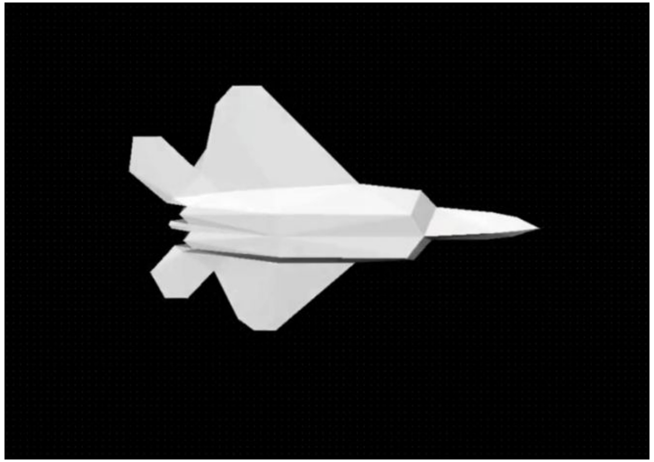


Projection Matrix

$$\begin{bmatrix} \left(\frac{h}{w}\right)\left(\frac{1}{\tan(\theta/2)}\right) & 0 & 0 & 0 \\ 0 & \left(\frac{1}{\tan(\theta/2)}\right) & 0 & 0 \\ 0 & 0 & \frac{zfar}{zfar - znear} & -\frac{zfar * znear}{zfar - znear} \\ 0 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Lighting

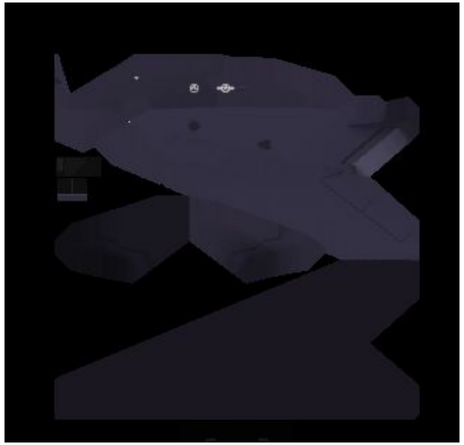


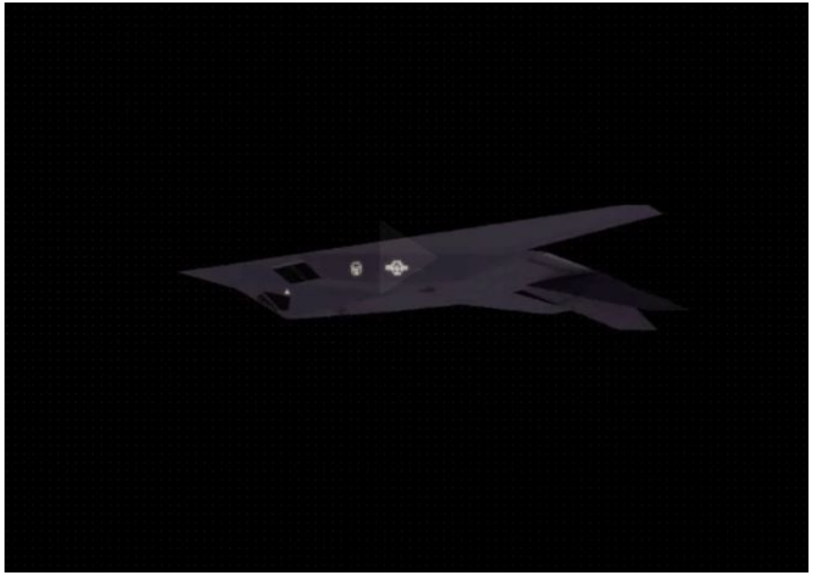


To use a single global lightsource, place a lightsource and compare where it is pointing to the direction each triangle face is pointing (the ‘normal’ vector of the triangle).

The dot product (ratio of similarity) between these two angles gives how much the triangle is pointing toward the light, and therefore how well-lit, or how intense its color, should be.

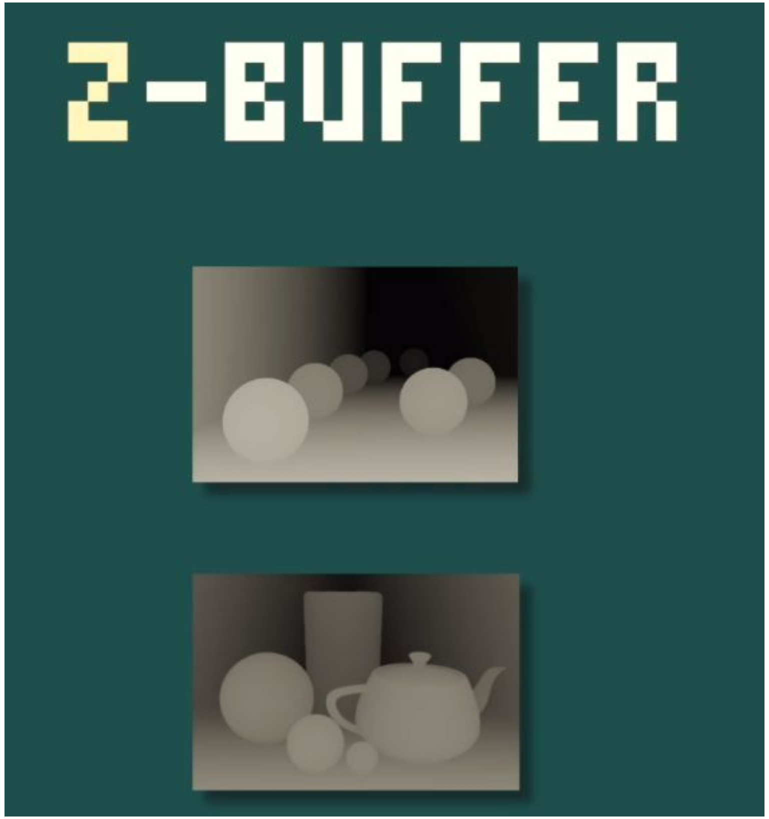
Texturing





I’m oversimplifying and sparing your from Barycentric weights and flat-top/flat-bottom triangles, but applying a texture is similar to drawing a solid color, except you read each pixel’s color-value from a texture image file.

Z-Buffer

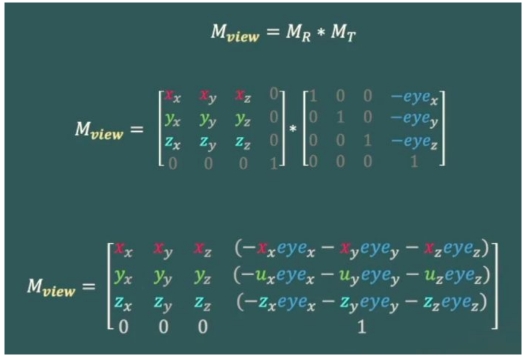


At this point we implement a z-buffer, which is a massive array which stores the depth of every pixel, in order to filter out any pixels which are behind another pixel.

This is a memory-performance tradeoff. Memory is eaten up by the z-buffer, but performance is gained by skipping a bunch of operations on the pixels which are found to be behind another.

Of course, the memory allocation needed is not a problem for modern hardware with the amount of pixels we're rendering.

Camera



$$M_{view} = M_R * M_T$$

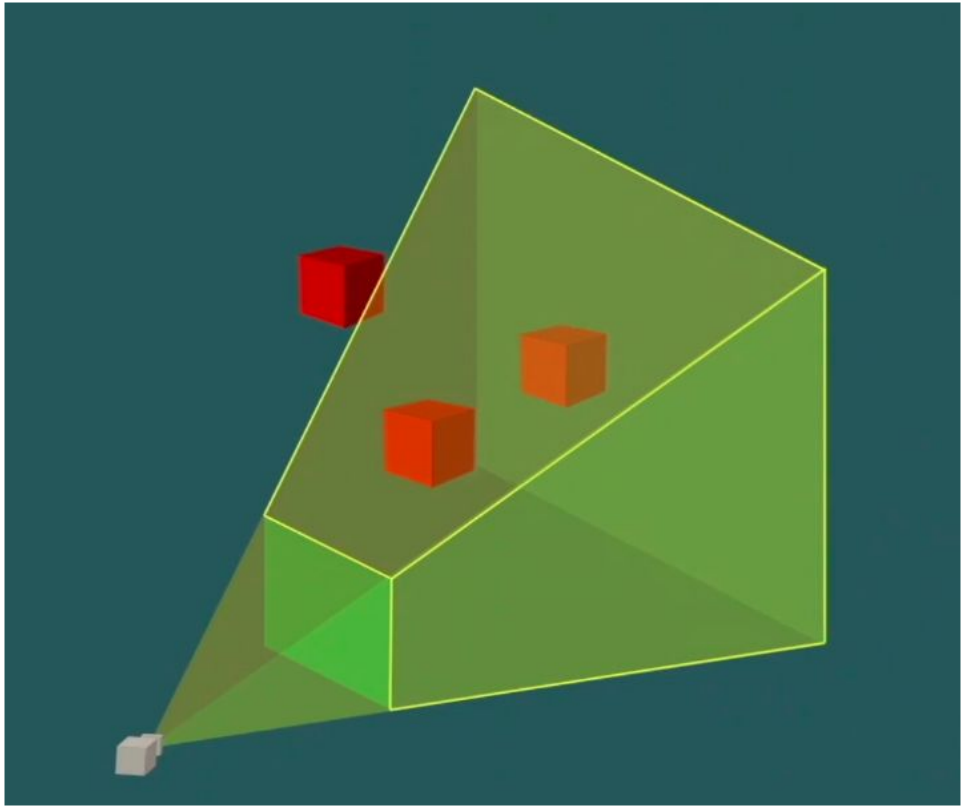
$$M_{view} = \begin{bmatrix} x_x & x_y & x_z & 0 \\ y_x & y_y & y_z & 0 \\ z_x & z_y & z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -eye_x \\ 0 & 1 & 0 & -eye_y \\ 0 & 0 & 1 & -eye_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{view} = \begin{bmatrix} x_x & x_y & x_z & (-x_x eye_x - x_y eye_y - x_z eye_z) \\ y_x & y_y & y_z & (-u_x eye_x - u_y eye_y - u_z eye_z) \\ z_x & z_y & z_z & (-z_x eye_x - z_y eye_y - z_z eye_z) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

While it seems like a camera in a game is rotating around the scene, in reality the camera is always the ‘origin’ point of the scene and everything in the scene is matrix transformed, scaled and rotated based on what the camera is doing.

Imagine when you press left on the camera, the entire world is picked up and shifted one increment to the left (or right depending on how old-school you are).

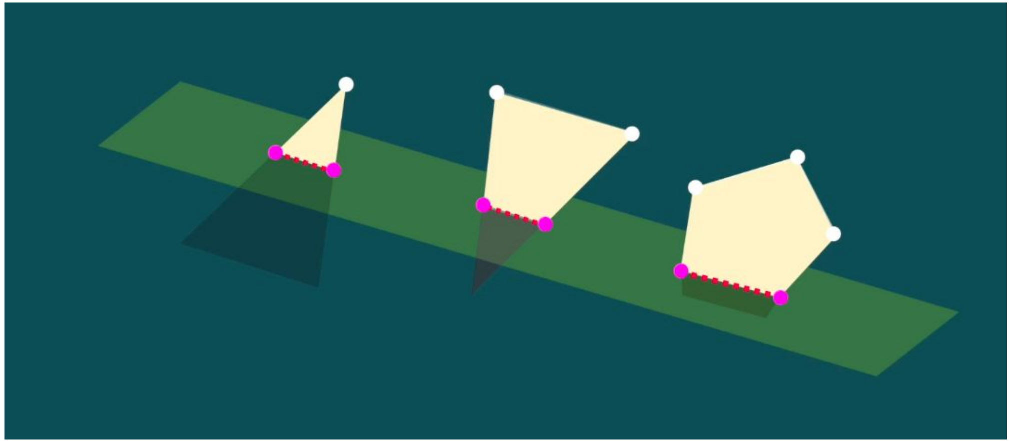
Frustum clipping

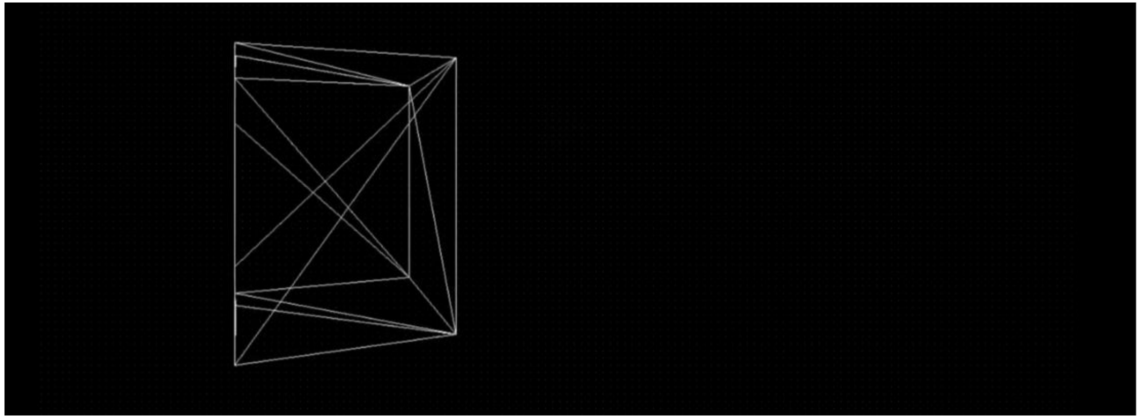


At this point, the executable will crash with a segmentation fault (trying to do something crazy with memory) if the 3D model leaves the visible screen space.

To fix this, we have to implement frustum clipping, which clips out any triangles or parts of triangles which are not inside the left, right, up, down, forward (too close) or back (too far away) frustum planes of the 3D space.

Frustum clipping





This is stupidly complicated because, when you clip triangles against edges, you can end up with shapes that have more sides than a triangle. You have to introduce the idea of several-sided polygons into your graphics pipeline, which then *eventually* get split back into new triangles for rendering.

The final result!