**National University of Computer & Emerging Sciences**

**Karachi Campus**



**Primitive 3D Engine Parallelization Code and Decomposition**

**Group Members:**

**Saud Ahmed Abbasi 19K-0229  
Zaid Bin Shahab 19K-1512  
Umer Ahmed 19K-0181**

**INSTRUCTORS:**  
**Muhammad Danish Khan**

# Introduction

3D engines have been around for ages, and have gone through waves of optimization. To test our mathematical skills, we take up the challenge to project, rotate, and transform a cube onto the console, as well as optimize the results using optimization.

The engine we have constructed can display 3 sorts of objects, a cube, rectangular cuboid and an open rectangular cuboid.

# Existing System

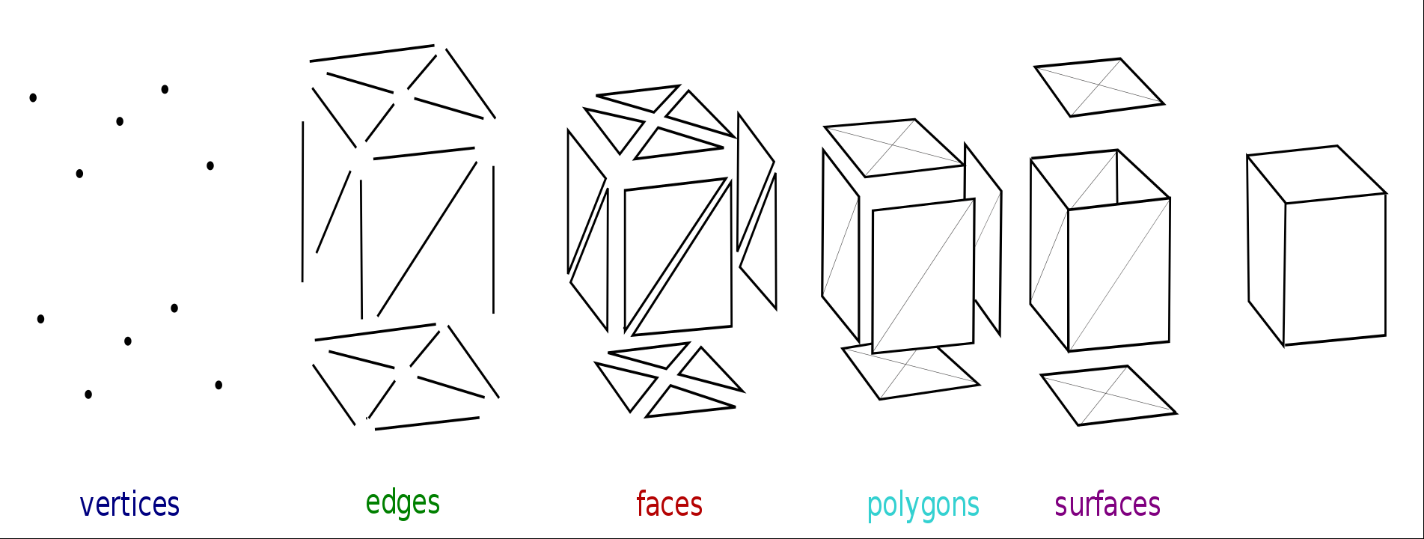
Inspired by the implementations in Blender, Maya, Zbrush, 3DSMax etc

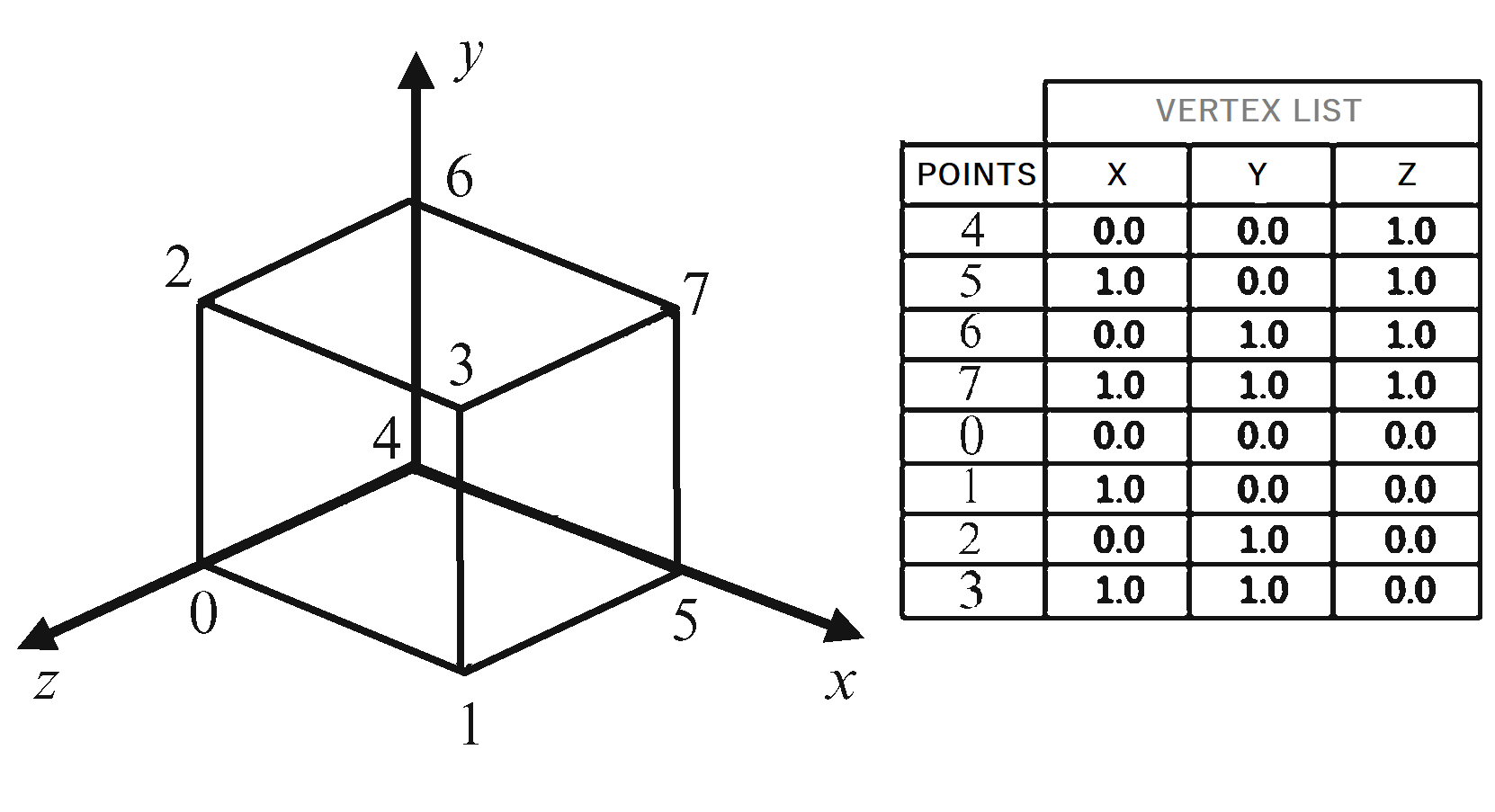
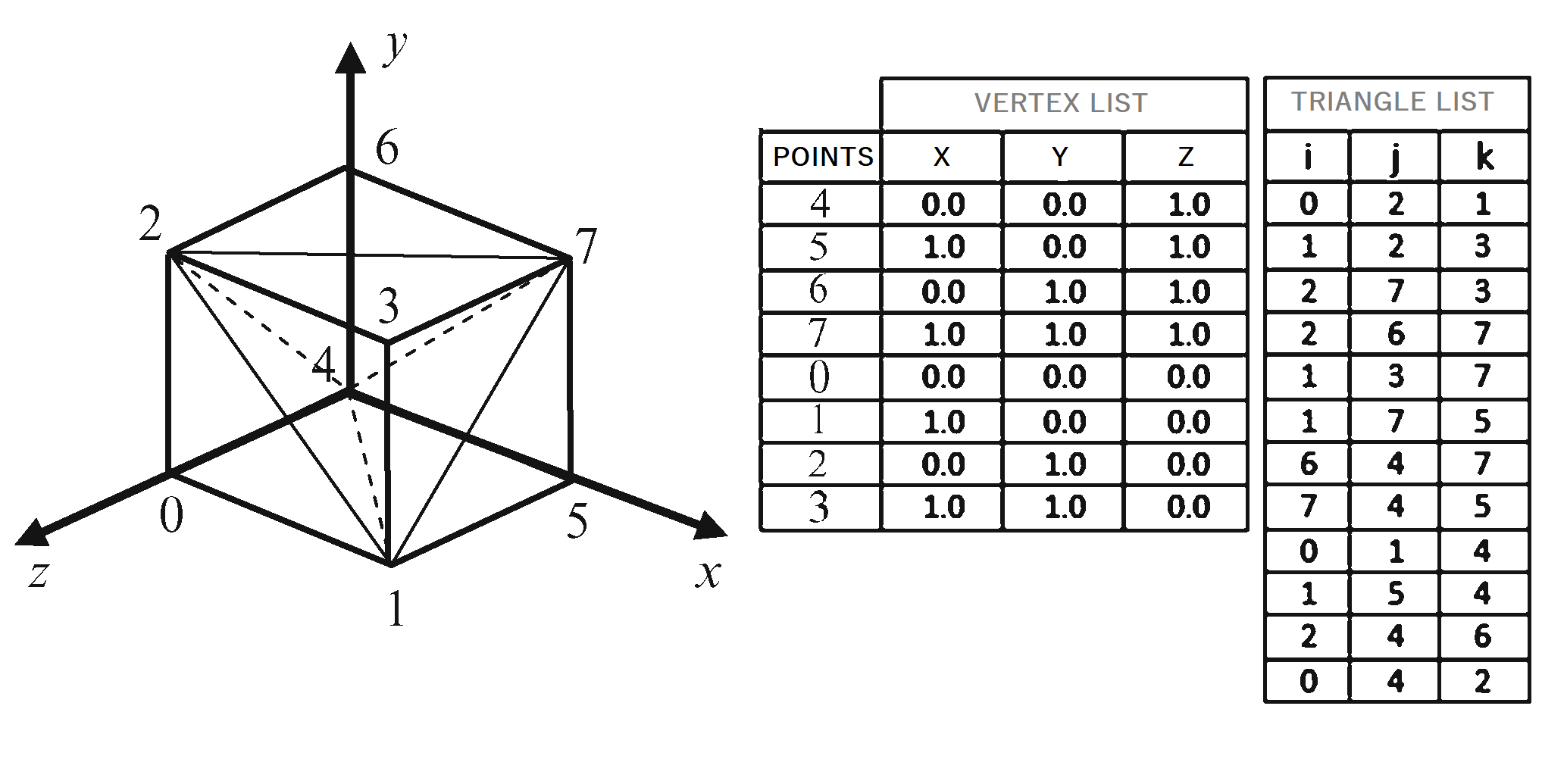
# Problem Statement

The real issue with the 3D engine all lands in the optimization category, how do we adequately rasterize a 3D object without landing ourselves below 20 FPS?

# Proposed Solution

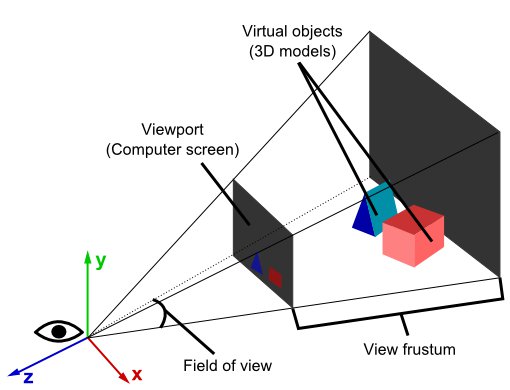
Layout Of the Object:   
Moving forward, keep these terminologies in mind:



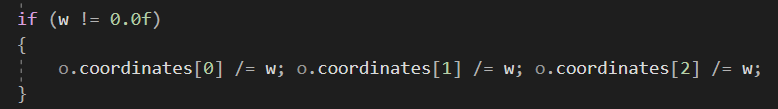
The original cube is defined as follows:  
  
  
Input decomposition is then applied to subdivide each surface into triangles faces, this makes it easier to draw on the console.  


Where each triangle is marked in a clockwise fashion, (the original cube used in the project has subdivision with a positive slope, to differentiate from the image given above where in the image the subdivision is at a negative slope, this is done to check consistency)

Intermediate Decomposition:

3D to 2D Projection:  
  
  
Homogeneous Coordinates are used to represent 2D/3D transformations, where [x,y,w] is used for 2D, and [x,y,z,w] is used for 3D.  
Since there are 3 axes in 3D as well as translation, this information fits perfectly in a 4x4 transformation matrix. A column-major matrix notation is used in this explanation.

The stages from 3D points and to a line, polygon or rasterized point is:

1. Transform your 3D points using the inverse camera matrix, followed with whatever transformations they need. If there are surface normals, transform them as well but set w to zero, as normals don’t need to be translated.  
     
   The normals must be transformed on isotropic matrices; since scaling and shearing makes the normals malformed.
2. Transform the point with a clip space matrix. This matrix scales x and y with the aspect ratio and FOV(field-of-view), z is scaled by the far and near clipping planes, and the 'old' z values are put into w.   
     
   After the transformation, x, y and z should be divided by w. This is known as a perspective divide.  
   
3. Sutherland-Hodgeman clipping is the most widespread clipping algorithm in use, this clipping is used so that you don’t render any pixels outside the viewport bounds.
4. X and Y should be transformed with respect to w and the half of both the width and height, this will land the x and y coordinates into the viewport coordinates. ‘w’ is discarded in its entirety but we retain 1/w and z, as 1/w is used for perspective-correct interpolation across the surface of the polygon and z is stored in the z buffer for depth testing.

Because z isn't used as a component in the position any longer, this stage is the actual projection.  
  
Calculate the field of view as follows:

fov = 1.0 / tan(angle/2.0)

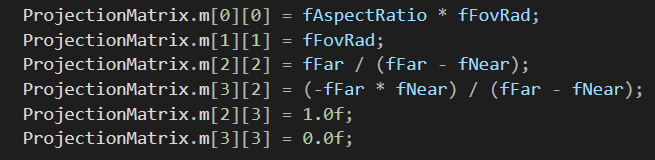
The clip matrix is as follows:

[fov \* aspectRatio] [ 0 ] [ 0 ] [ 0 ]

[ 0 ] [ fov ] [ 0 ] [ 0 ]

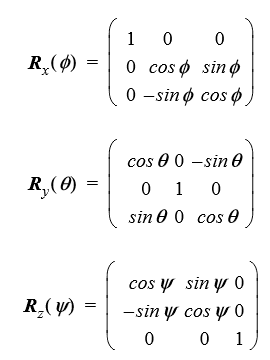
[ 0 ] [ 0 ] [(far+near)/(far-near) ] [ 1 ]

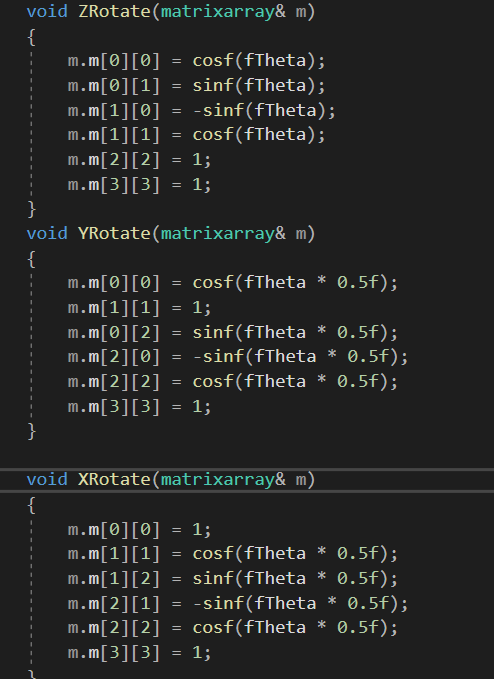
[ 0 ] [ 0 ] [(2\*near\*far)/(near-far)] [ 0 ]

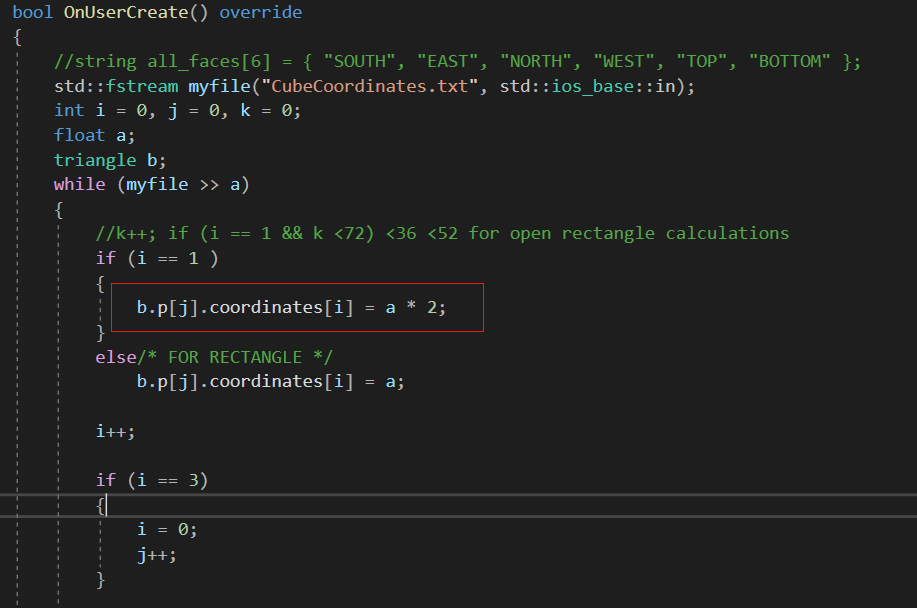
Where the aspectRatio is width/height, therefore the X component is scaled based on FOV for y.  
(Some adjustments after rref form and different interpretations)  


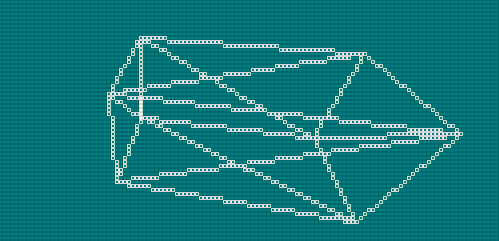
To get our screen coordinates this is the final transformation:  
new\_x = (x \* Width ) / (2.0 \* w) + halfWidth;

new\_y = (y \* Height) / (2.0 \* w) + halfHeight;

Rotation:  
  
Before transformation, to justify that the object is in fact, 3D in nature, we may use the rotation matrices before transformation:  


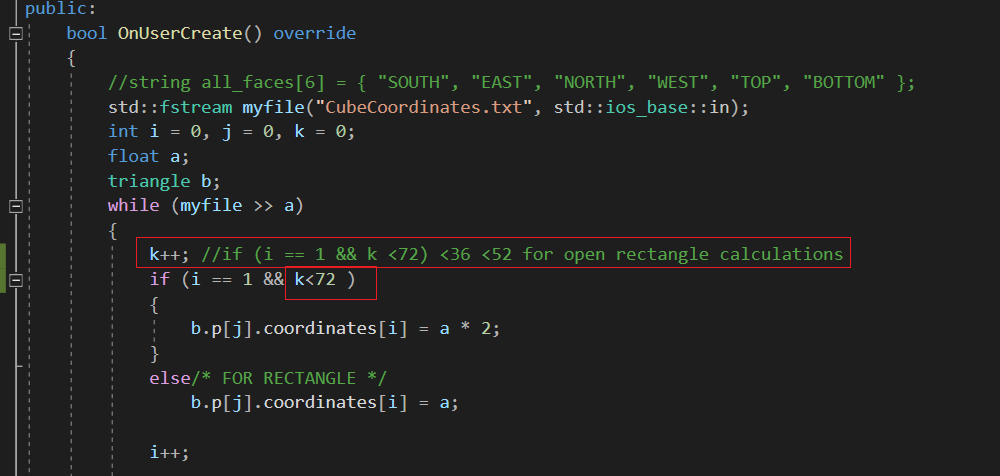
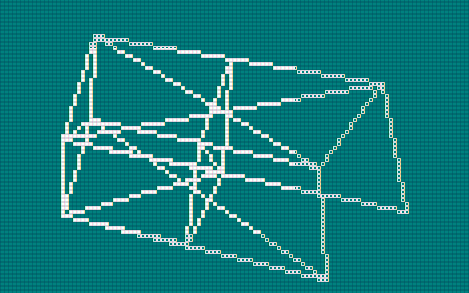


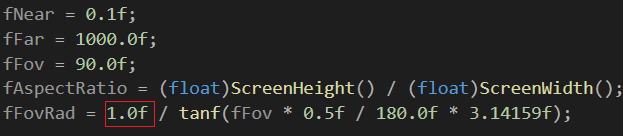
How to get a rectangular cuboid:  
  
You can manually set each y coordinate of every triangle vertex to a different value, this easier for us as we extract the values from a file, so before the values are assigned, we check if it’s a y coordinate then just multiply it by a specific value.  
  


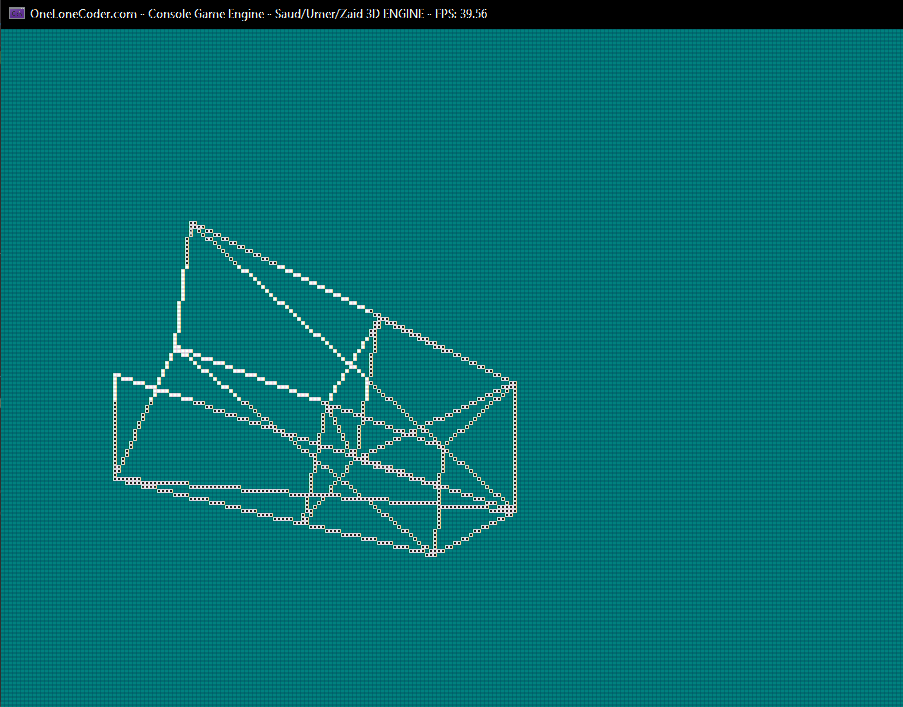


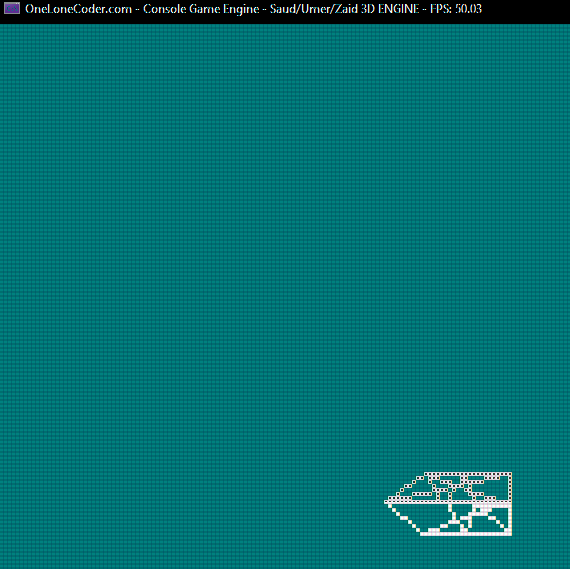
## How to get an open rectangular cuboid:

To make an open rectangular cuboid, I simply restrict the modification of the y coordinate and scale it only when k reaches a certain value after incrementing in each loop.

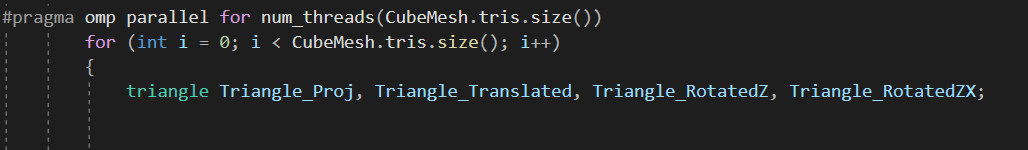
  
  
  
How to scale objects:  
  
1) You can manually alter all the points to have different values, this will scale this object  
2) You can change the Field-of-View angle so that in a way, the camera is further away from the object, scaling it accordingly.

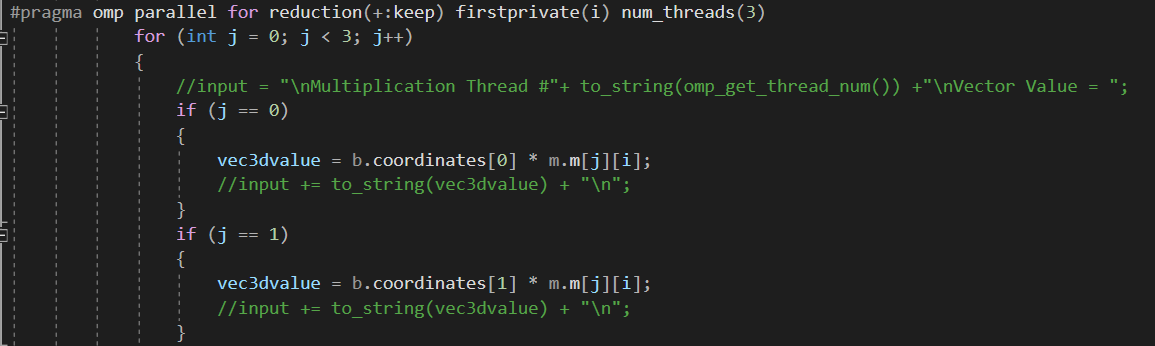


Parallelization:  
  
Done using OpenMP, on multiplication for clipping, and multiplication for transformations and rotations as well as projections for all vertices of all triangles.





# Tools & Technologies

Exclusively used C++ language, the “oldConsoleGameEngine.h” header library is used to display on console (particularly the DrawTriangle() Method), we also override the update methods for our own convenience.