Re-Engineering 3D Engine With Linear Algebra

# What is this?

This is a project guide that assist you to re-engineer a first-generation 3D rendering engine with basic linear algebra concepts. You will be surprised by how **straightforward** and **simple** it is – it does not contain a lot of code, or a tons of math concepts; with a little bit of knowledge about **mathematic modelling** and **linear algebra**, you’ll be able to construct the whole rendering engine from pieces!

# Few words about history……

Do you know that the first 3D video game ever was produced as early as 70s?

The first 3D video game ever was *Maze* – It looks like this:

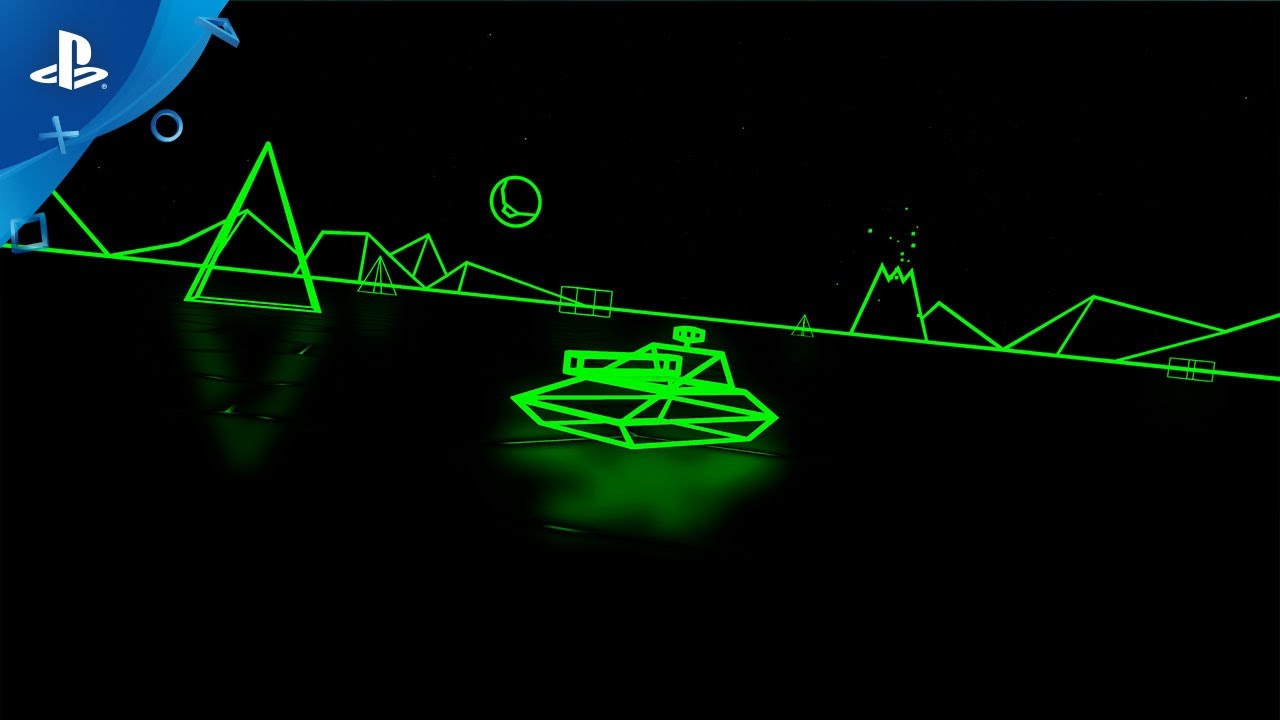


↑MIT version of *Maze* on an Imlac PDS-1D at the [Computer History Museum](https://en.wikipedia.org/wiki/Computer_History_Museum), from [Image Source: upload.wikimedia.org](https://upload.wikimedia.org/wikipedia/commons/f/f4/Maze_war.jpg)

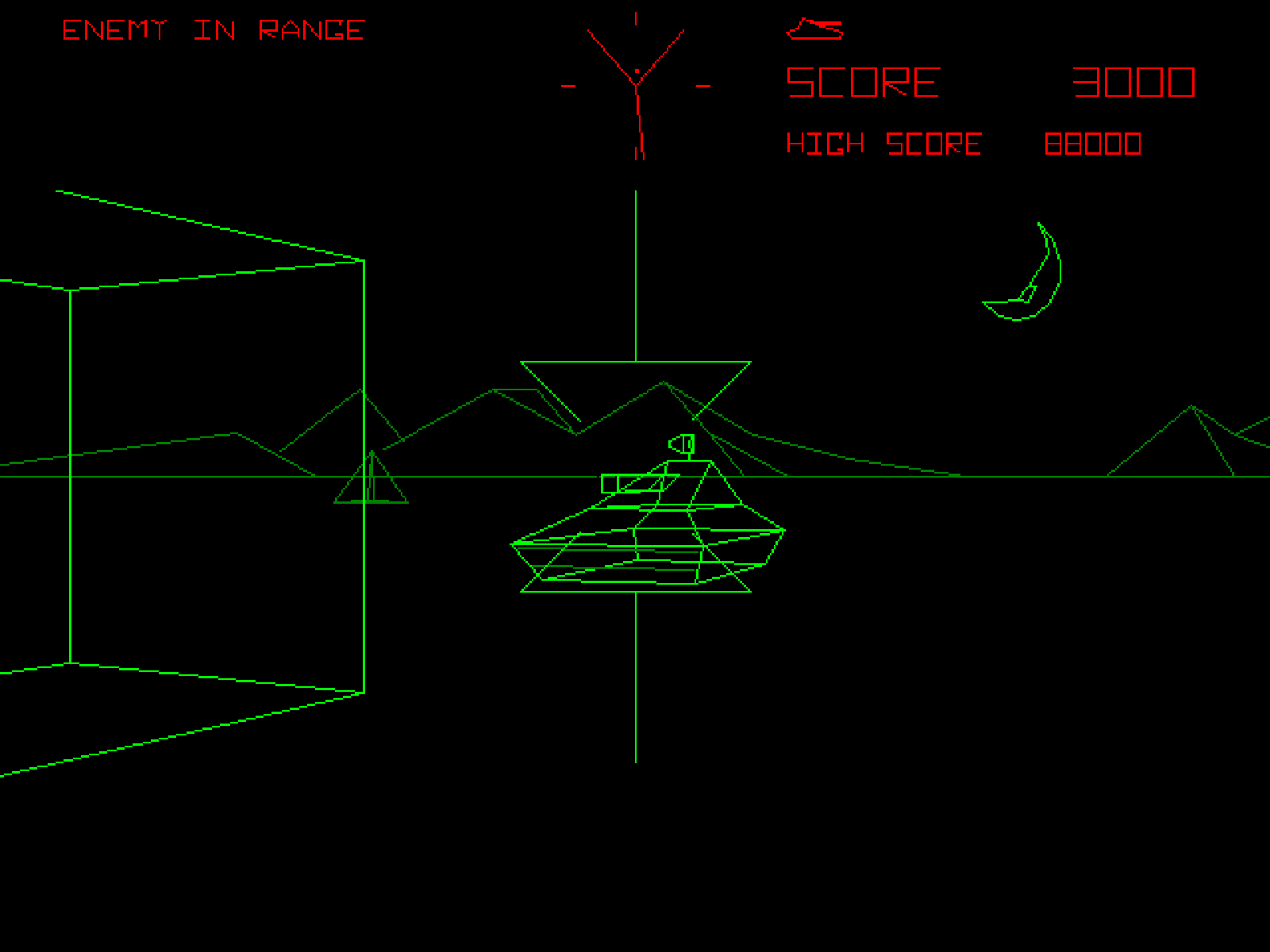
It was made in 1973 and is a first-person-shooter-game, which players will be shooting each other (yup, it supports multiplayer) in a 3D maze.

It is extremely primitive – there are only series of grids that represents walls, not texture or even image was included; the player could only turn 90-degree angle, etc. Though they didn’t stop it from being a popular game. You can find out more things about it [here](https://en.wikipedia.org/wiki/Maze_(1973_video_game)#Legacy) at Wikipedia.

Series of commercial 3D games were then released in the 80s, like [*Battlezone*](https://en.wikipedia.org/wiki/Battlezone_(1980_video_game)), which sold 15,000 copies (which means we could also be millionaires if back to 80s with this project done!)



*Battlezone* Classical mode remastered at PS VR, from [PlayStation Blog](https://blog.playstation.com/2016/12/15/battlezone-gets-classic-mode-update-on-dec-20/)



A screenshot from original *Battlezone*, from [Time](https://techland.time.com/2012/11/15/all-time-100-video-games/slide/battlezone-1980/)

# What do I need for making this 3D engine?

We’ll need some knowledge from Linear Algebra – since this is an applied math project. You need to know:

1. What is a **vector**
2. What is a **matrix**
3. What is **matrix-vector multiplication**

And, of course we’ll need knowledge from basic algebra, like trigonometry.

Programming wise, since this is intended to be an applied math project instead of a coding lab, I’ll be offering as much instructions as I can to make the coding part easier, and external links to potentially helpful third-party resources as well.

Hardware wise, you’ll need a PC (or something else that can run code) with OS that supports **Java** and its IDE (the integrated Development Environment – the nerdy name for a fancy text editor). We’ll make the engine with Java since it’s the first programming language of a good number of high school / college students, though the mathematical concepts potentially work for any other computer language as well. Feel free to try them out later.

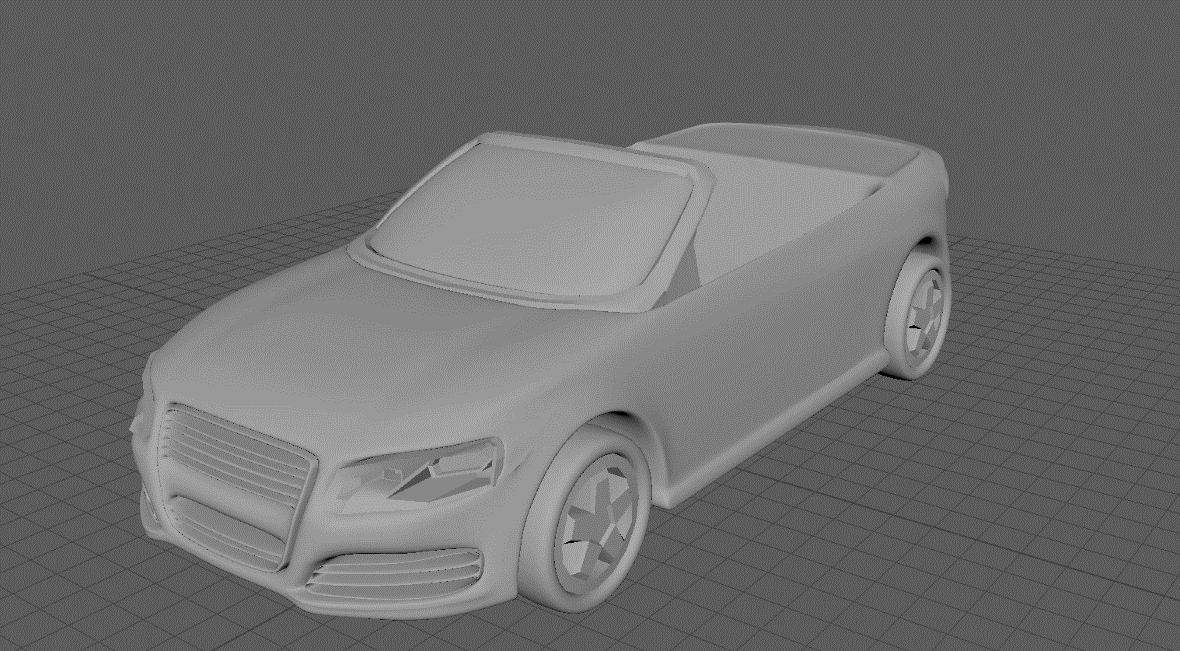
When you are ready, flip this page and we’ll begin!

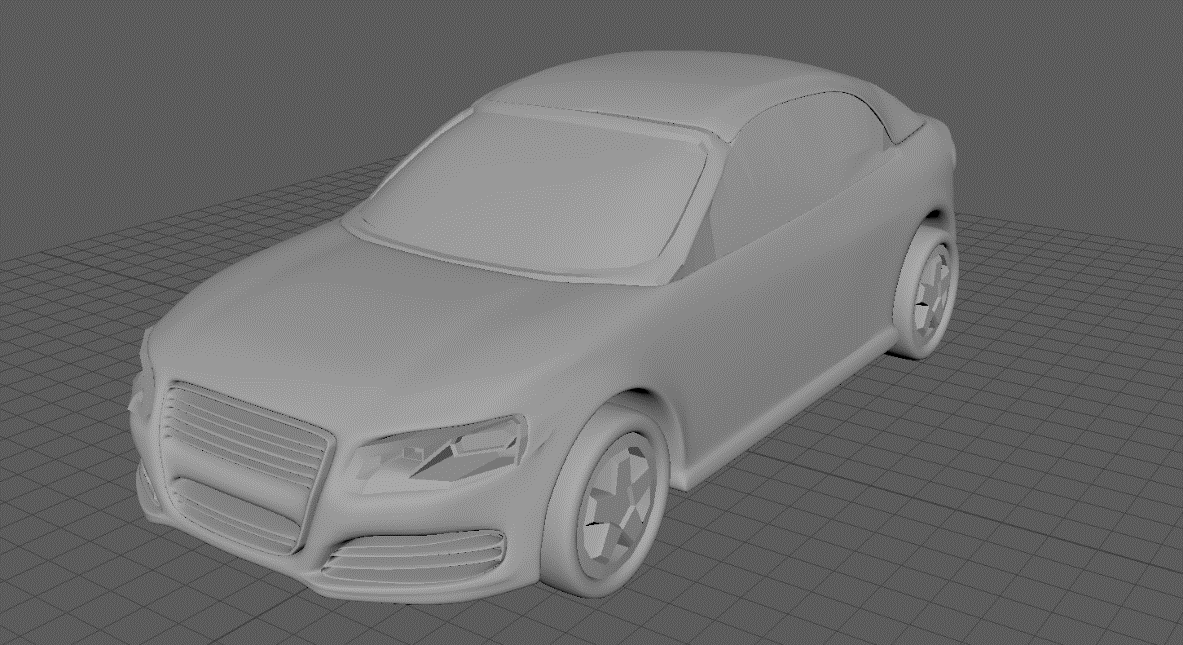
# How do I store a 3D shape?

Or in other words: How do we use numbers to represent a 3D shape?

In case you have taken a multivariable calculus class, of course we can describe a shape with 3-variable polynomials; that will yield perfect curves and save your disks.

However, won’t it be a bit too hard for artists who want to produce complex shapes? What about rigged models (which change shape according to code or “skeleton”) made for animations? What about the real-time compute power consumed when calculating a whole system of polynomials?





↑Think about this car – how much time is it going to take to use a system of polynomials to describe it, and how much compute power will it consume when rendering?

Vertices and Lines

What about starting with a lower dimension? Like, a dot (or point).

It’s easy to use a 3-variable vector to represent a point in 3D space ():

The position of this point is x units of displacement on direction, y units of displacement on direction, and z units of displacement on direction from origin.

Then, what about several points within the same ? That’ll be a set of 3D vectors:

|  |
| --- |
| A set of 3D vectors, , , , }, in other words, {, , , …} |

Now, by connecting any of the two vectors (points), we then have straight lines.

Then, since each line only consists of two points, it’s easy to store lines as array in computer – an array of 2D vectors, with each of its component referring to a point:

|  |
| --- |
| Assuming 1 stands for , 2 stands for in in a vector that represents a line in R3,  Then basically represents a line that connects point and point .  In this way, a series of lines will be {, , , …} |

You can also arrange three lines that connect three points together in another vector to represent a triangular face (surface), though that has beyond our scope today.

Now, we have a several points (vertices) in , as well as several lines that connects them. If we tell our computer to draw these lines, we’ll get a holographic “skeleton” on screen. That’s exactly the basis of the rendering technology that the 3D animation & game & movie industries are using today.

Homogeneous vectors

For simplifying the perspective mapping operations later on, we need to apply homogeneous vectors (instead of normal 3d vectors) for storing vertices (or points).

A homogeneous vector is a 4D vector that contains four components: x, y, z, w, which maps into a point in 3D space, that , , .

Normally, w has to be 1 for minimizing calculation for the 3D point that the vector maps. When w = 0, the vector becomes “virtual” and can be considered pointed towards infinity (which we won’t be using in this project).

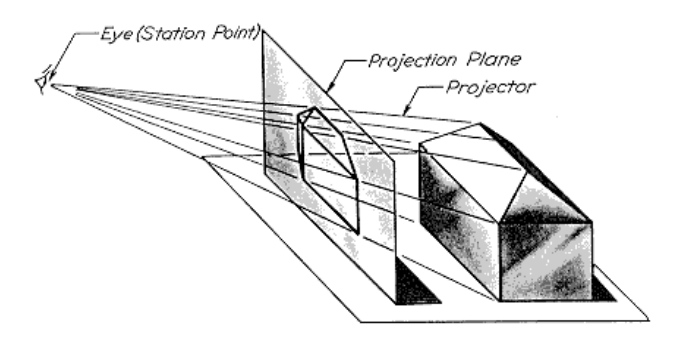
Please don’t worry if you can’t understand the significance of homogeneous vector just yet, this is just the definition for homogeneous vector, and you’ll see it’s usage in next page.

# Rendering the shape in Perspective

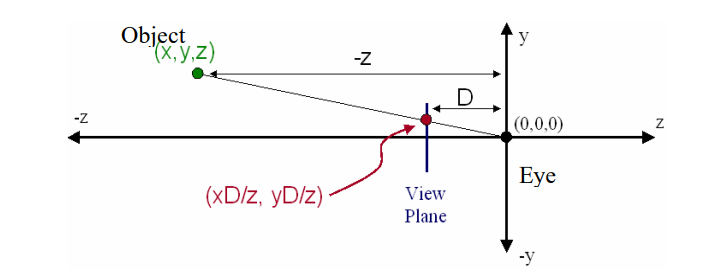
For a 3D engine that really works for animations, games, etc., for the sake of realism, we want to render our skeleton in perspective that’s similar to our eye’s.

In the perspective of human eye, the further objects tend to be smaller, and the closer objects tend to be larger. That makes 3D rendering trickier than 2D rendering.

Let’s draw a diagram to see how exactly does perspective works with a screen:



↑Perspective in 3D (Yip, 2001) (awaiting remaster)



p



↑Perspective in y-z plane (Yip, 2001) (awaiting remaster)

Basically, human eye concentrates lights from all 3D space to a single point, in our case the origin (position of eye).

See that the **green triangle** and the **red triangle** are similar – that essentially makes it possible to calculate the position of point **p** on view plane, which is .

Applying the same method to x-z plane, the position of point **p** mapped to screen will be . That gives us the 2D coordination of the point **p** on screen: .

Know how to turn this into a matrix-vector multiplication? Write down your own answer before turning to next page.

We assume the 3D coordinate of point **p** being represented by a homogeneous matrix, which is . Then we compose the equation in the basic matrix equation form :

A =

Have any idea about what will A actually look like? Try to write it yourself before you see my answer.

Our perspective matrix: A =

A little reminder of the definition of homogeneous vector – the result of now seems like , yet the point our yielded homogeneous vector actually maps to in is actually point which , , .

Therefore, the final vector is equivalents to ,

which match our initial derivation of A = .

Now, given a point in , you can easily map it to a 2D screen by multiplying our Perspective Matrix A with it, which will yield its position on screen.

Since our 3D shapes are being stored by vertices, if we perform the matrix-vector multiplication with every vertex, we can map the whole object onto our screen. Then, we can tell the computer to draw the lines between these vertices on screen – we’ll see our 3D model in perspective view.

# Move and Rotate

If it’s only about rendering, we would have stopped at the last section. However, in almost any practical 3D engines for games, animations, or physical simulation, users should be able to move and rotate objects freely in virtual space. So, let’s take a look and implement these on our own engine.

Movement (awaiting image)

Movement seems quite easy. Can you think of a matrix equation for moving a point from to in form ? Try to write down your own before you look at mine.

=

Easier than you might think, I guess? Note the significance in our fourth column – it’s made possible by the 4th component of the homogeneous vector. It’ll be more difficult to be written in form of matrix equation (though a simple vector addition will also work).

Rotation (awaiting image)

Rotation is a bit trickier comparing to Movement, and as you might guess, we will be using trigonometry in our matrix equation.