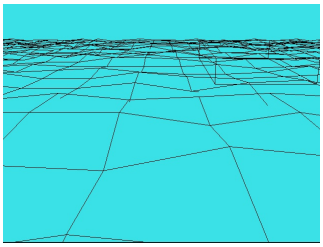
How to make ChrisGo-3D

January 2022



1 Resume

My personal aim for this small project was to learn the basics of C++. With out reading anything about 3D engines, the idea was recreating the math to transform a 2D library (SDL) into a 3D engine.

2 Project a 3D object in a 2D plane:

By default in SDL, the point $(0, 0)$ is at the top left of the screen.
We can think the view of a person like a pyramid, the further we go, the bigger is the view range, Figure 1.

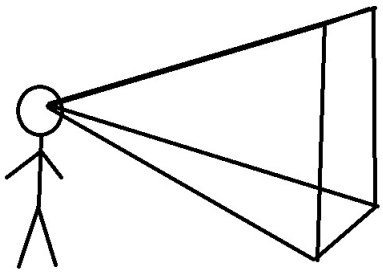


Figure 1: Field of view of the people.

The size of the screen is a constant (in this case set to 640x480), but the view range increases with z, this implies that objects further apart seem smaller. If the angle associated to the x axis of the triangle is $60°$, then $z = x$ and because the aspect ratio, the other angle has to be about $44°$. If we set a maximum height that objects have when $z \to \infty$ and make objects go to the middle, we get the following expressions

$$P_x \ \vec{v} = 320 + 640 \ \frac{x}{z}\hat{x} \tag{1}$$

$$P_y \ \vec{v} = \text{Height} + 0.75 \cdot 480 \ \frac{y}{z}\hat{y} \ . \tag{2}$$

Where P_x and P_y are the operators that act over the 3-dimensional vector.

3 Points, Objects and rotations:

To begin we are going to set an absolute set of coordinates, whose origin point $(0, 0, 0)$ is where the player starts.
A point's position is defined by a 3 dimensional vector $\vec{v}$, but the projection of the point has to be relative to the players perspective.
First we can calculate the difference between the position of a point A and the player, this gives us the position of

A in the reference system of the player. Rotating the view of the player is equivalent of rotating the object around the player Figure 2, so using the matrices of rotation

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{pmatrix} R_y = \begin{pmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{pmatrix} R_z = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{3}$$

We can get the values to project the object in the players field of view.

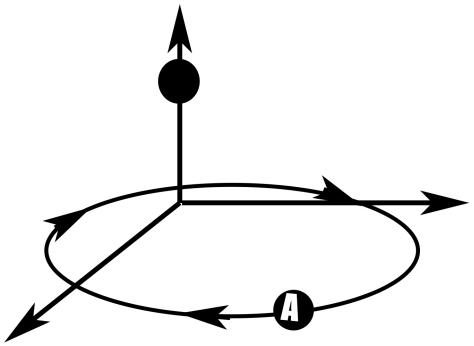


Figure 2: Object rotates around the player.

In this example, a rectangular parallelepiped was coded, having the 2 extremes of the object is easy to get all the others and render the connections. To update the position of the object, we update the position of each point as seen before.

4 Terrain:

First a terrain object is located in the Heap so it doesn't get destroyed when the game loops. This terrain object has to make new "Chuncks" if the position wasn't previously visited, and load the old ones in the other case.

4.1 Chuncks:

Chuncks in this program are blocs of 100×100 in the xz plane. Chuncks have a resolution (amount of divisions per Axis), so the amount of points in a Chunk is $resolution^2$ and the points are separated by a constant amount

$$\Delta x_\alpha = \frac{100}{resolution}\ . \tag{4}$$

This points are created by a function Create Chunck and then saved in a list. The list saves the points using a loop that adds on the x and z components, so the order of the points in the list are given by the black arrows on Figure 3.

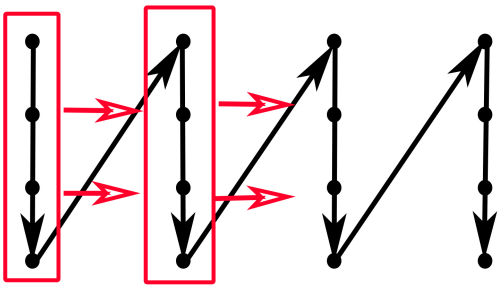


Figure 3: How points are saved in the list.

So now, we want to draw the Chunck.
If we draw the points following the pointer, except when the i-th element is congruent to -1 modulo *resolution*

$$i \equiv -1 \mod (resolution) \ , \tag{5}$$

we have all the vertical lines. Drawing the horizontal lines in a linear time requires the use of memory. Making 2 vectors of length *resolution*, let's us unite all the points with the same remainder.

4.2 Chunck Management:

The player must be seeing the terrain at all times, so 6 chuncks in front of the player are render Figure 4, when the player moves to the next Chunck, then the ones back to him are deleted while the ones in front of him are render.

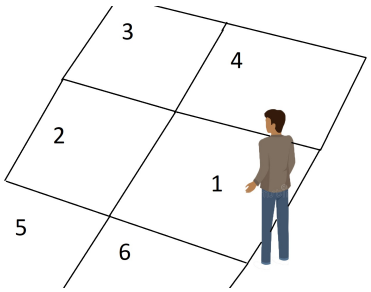


Figure 4: How chuncks are render.

If the player goes in some direction and then comes back, we expect the terrain to be exactly the same as it was, so the Chuncks must be stored in some way. If we think the Chuncks to have their own set of coordinates X and Z, we can store them in a 2 dimensional "deque", this is because we want to add new elements at the beginning or at the end of the list fast.
If we set the first coordinate of the deque with the rows and the second with the columns. It is possible to observe that if a new row is visited, then the previous ones were previously visited, This means that if we check the first coordinate, and the new X is greater than it, then the new X must be stored other side of the list and its general position will be

$$\text{Pos}(X) = X - list.at(0).at(0)[0] \ . \tag{6}$$

The Z coordinate can't be found that directly, but applying the same logic and adding the numbers in the same way, we see that the Y components for a given X are sorted, this let's us search the number with a binary search. This means that the algorithm to find the Chuncks has a complexity $\mathcal{O}(n)$, where n is the amount of rows visited.

5 Ideas on Collision (possibly in the future):

Detecting if each object is colliding with all the other objects in the map is almost impossible, but the Chunks in the terrain may be a good way to improve this. We are going to Check only the points that are on the same Chunck.
A first approximation is making one or more spheres inside the object to check if 2 objects are colliding, then calculate the torque made to the object ($\vec{\tau} = \vec{r} \times \vec{F}$). Finally, with small oscillations we have a simple expression for the rotation of the body on stable axis, and a approximation for the unstable ones that can be updated in time to make a better solution.

6 Other improvements not implemented:

Trick to improve memory on Chunck saving: instead of saving all the points in a Chunck, saving instructions could be more efficient for the memory.
Use real time: Right now the time in the code is random and depends only in the speed of the PC. Using real time, is essential to make the physics of the game that depend on the time.
Shaders: As far as i could see, SDL does not support painting figures that are not rectangles, using other libraries, could make this possible and shaders could be implemented, for example counting the intersection of lines between the light source and the object illuminated.