Raycast Engine Guide

This guide will serve as a way to learn all the different parts of the engine, and all the different components that allow it to work etc. This also helps me keep track of all the different things I have added myself as I tend to forget haha!

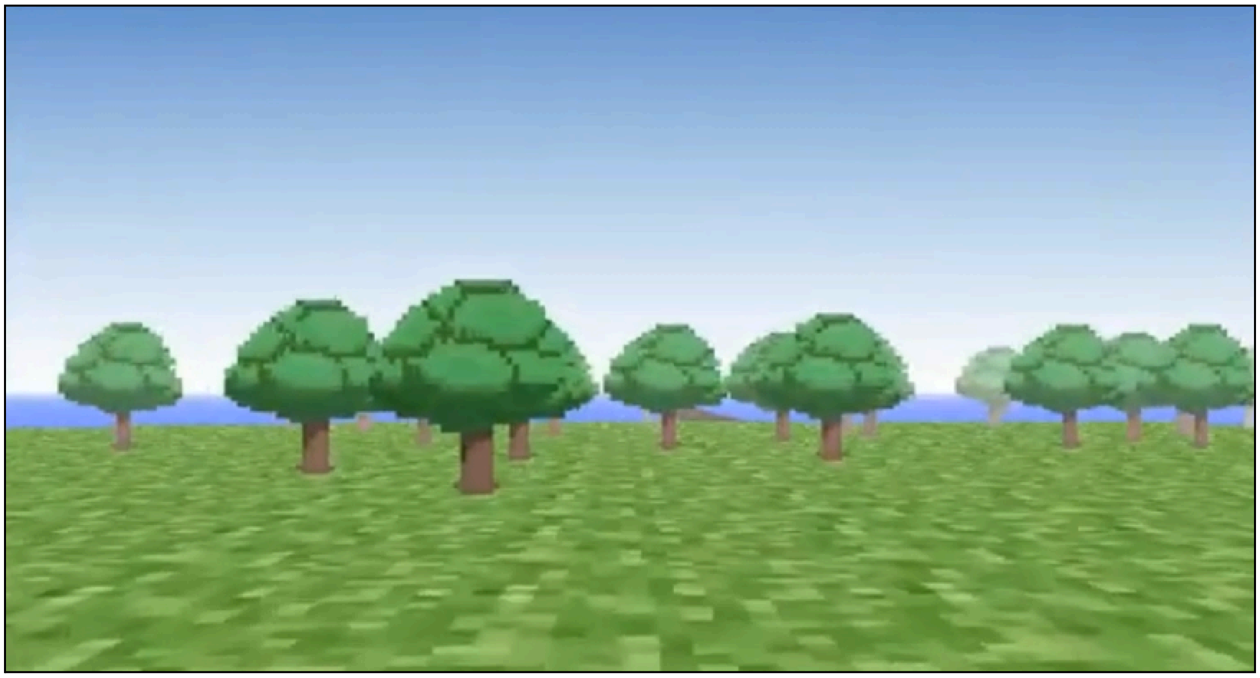
This guide will be split into 3 parts. The first part will mainly be talking about how raycasting works. Some explanation as well some links to resources so you can learn about it on your own. I will say the math can be a bit complex even for something as simple as a raycaster, so be warned!

The second part will walk you through each part of the engine in case you want to know how its insides are organized if you want to modify it, etc.

Lastly, I also will include a section at the end on how to use the editor. This editor allows you to place objects in the world and create all your games that will be read by the engine and rendered!

Ok, enough talk, let’s get started!

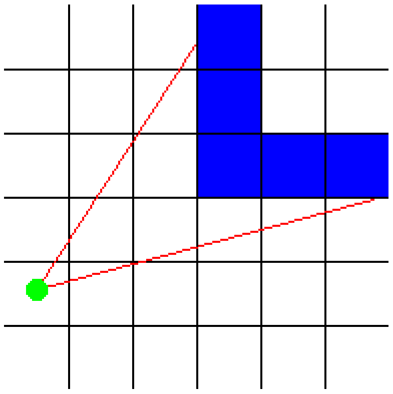
Part 1: Raycasting Main Mechanics



First of all, what the heck is raycasting? Sure, when we hear of raycasting, maybe the first thing that comes to mind is old retro graphics from the 90s. Notable titles include Wolfenstein 3D, one of the first shooters in 3D and used raycasting. There were many other games that also used raycasting, especially in the SNES, and other early consoles. I must say, there is a certain charm to its simplicity. It’s not only lightweight, but if you are creative enough, you can tell some pretty amazing narrative stories with this technique. No fancy GPUs or graphics, just lightweight software and your imagination.

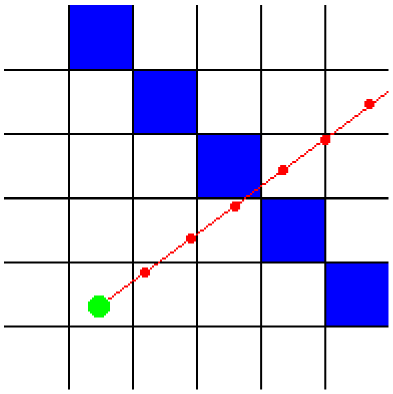
The basic idea of raycasting is as follows: the map is a 2D square grid, and each square can either be 0 (= no wall), or a positive value (= a wall with a certain color or texture).

For every x of the screen (i.e. for every vertical stripe of the screen), send out a ray that starts at the player location and with a direction that depends on both the player's looking direction, and the x-coordinate of the screen. Then, let this ray move forward on the 2D map, until it hits a map square that is a wall. If it hit a wall, calculate the distance of this hit point to the player, and use this distance to calculate how high this wall has to be drawn on the screen: the further away the wall, the smaller it's on screen, and the closer, the higher it appears to be. These are all 2D calculations. This image shows a top down overview of two such rays (red) that start at the player (green dot) and hit blue walls:

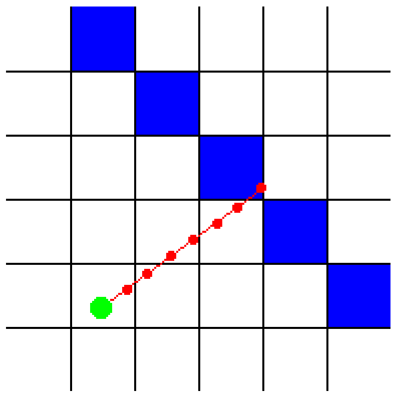


To find the first wall that a ray encounters on its way, you have to let it start at the player's position, and then all the time, check whether or not the ray is inside a wall. If it's inside a wall (hit), then the loop can stop, calculate the distance, and draw the wall with the correct height. If the ray position is not in a wall, you have to trace it further: add a certain value to its position, in the direction of the direction of this ray, and for this new position, again check if it's inside a wall or not. Keep doing this until finally a wall is hit.

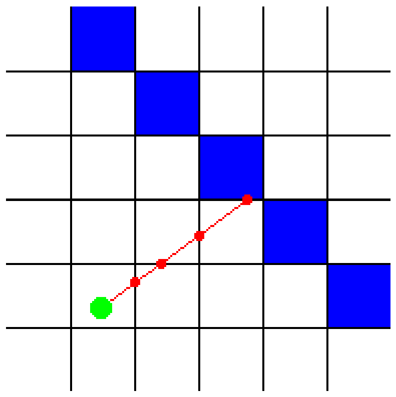
A human can immediatly see where the ray hits the wall, but it's impossible to find which square the ray hits immediatly with a single formula, because a computer can only check a finite number of positions on the ray. Many raycasters add a constant value to the ray each step, but then there's a chance that it may miss a wall! For example, with this red ray, its position was checked at every red spot:



As you can see, the ray goes straight through the blue wall, but the computer didn't detect this, because it only checked at the positions with the red dots. The more positions you check, the smaller the chance that the computer won't detect a wall, but the more calculations are needed. Here the step distance was halved, so now he detects that the ray went through a wall, though the position isn't completely correct:



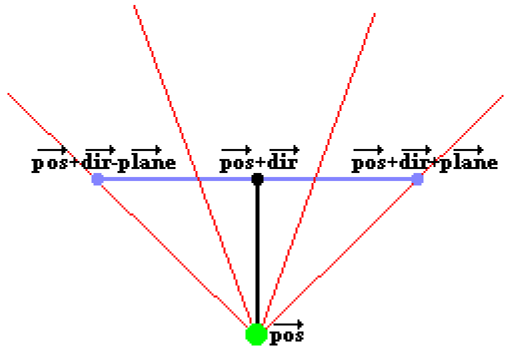
For infinite precision with this method, an infinitely small step size, and thus an infinite number of calculations would be needed! That's pretty bad, but luckily, there's a better method that requires only very few calculations and yet will detect every wall: the idea is to check at every side of a wall the ray will encounter. We give each square width 1, so each side of a wall is an integer value and the places in between have a value after the point. Now the step size isn't constant, it depends on the distance to the next side:



As you can see on the image above, the ray hits the wall exactly where we want it. In the way presented in this tutorial, an algorithm is used that's based on DDA or "Digital Differential Analysis". DDA is a fast algorithm typically used on square grids to find which squares a line hits (for example to draw a line on a screen, which is a grid of square pixels). So we can also use it to find which squares of the map our ray hits, and stop the algorithm once a square that is a wall is hit.

Some raytracers work with Euclidean angles to represent the direction of the player and the rays, and determinate the Field Of View with another angle. I found however that it's much easier to work with vectors and a camera instead: the position of the player is always a vector (an x and a y coordinate), but now, we make the direction a vector as well: so the direction is now determinated by two values: the x and y coordinate of the direction. A direction vector can be seen as follows: if you draw a line in the direction the player looks, through the position of the player, then every point of the line is the sum of the position of the player, and a multiple of the direction vector. The length of a direction vector doesn't really matter, only its direction. Multiplying x and y by the same value changes the length but keeps the same direction.

This method with vectors also requires an extra vector, which is the camera plane vector. In a true 3D engine, there's also a camera plane, and there this plane is really a 3D plane so two vectors (u and v) are required to represent it. Raycasting happens in a 2D map however, so here the camera plane isn't really a plane, but a line, and is represented with a single vector. The camera plane should always be perpendicular on the direction vector. The camera plane represents the surface of the computer screen, while the direction vector is perpendicular on it and points inside the screen. The position of the player, which is a single point, is a point in front of the camera plane. A certain ray of a certain x-coordinate of the screen, is then the ray that starts at this player position, and goes through that position on the screen or thus the camera plane.

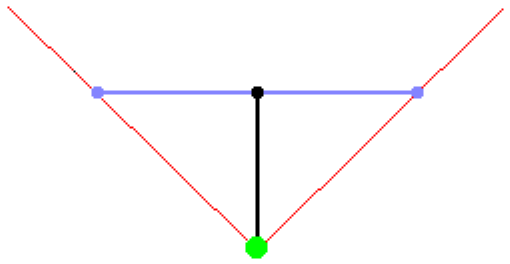

$\vec{pos}+\vec{dir}-\vec{plane}$
$\vec{pos}+\vec{dir}$
$\vec{pos}+\vec{dir}+\vec{plane}$
$\vec{pos}$

The image above represents such a 2D camera. The green spot is the position (vector "pos"). The black line, ending in the black spot, represents the direction vector (vector "dir"), so the position of the black dot is pos+dir. The blue line represents the full camera plane, the vector from the black dot to the right blue dot represents the vector "plane", so the position of the right blue point is pos+dir+plane, and the posistion of the left blue dot is pos+dir-plane (these are all vector additions).

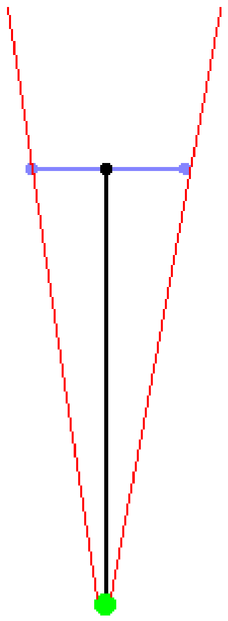
The red lines in the image are a few rays. The direction of these rays is easily calculated out of the camera: it's the sum of the direction vector of the camear, and a part of the plane vector of the camera: for example the third red ray on the image, goes through the right part of the camera plane at the point about 1/3th of its length. So the direction of this ray is dir + plane*1/3. This ray direction is the vector rayDir, and the X and Y component of this vector are then used by the DDA algorithm.

The two outer lines, are the left and right border of the screen, and the angle between those two lines is called the Field Of Vision or FOV. The FOV is determinated by the ratio of the length of the direction vector, and the length of the plane. Here are a few examples of different FOV's:

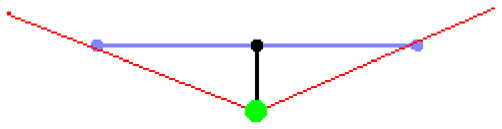
If the direction vector and the camera plane vector have the same length, the FOV will be 90°:



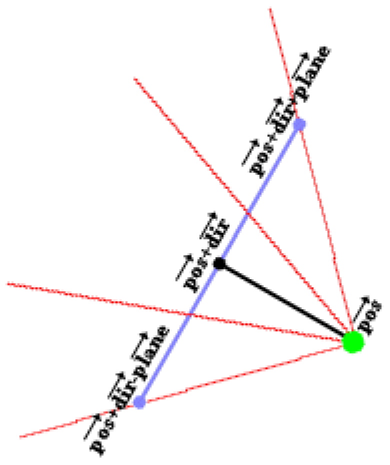
If the direction vector is much longer than the camera plane, the FOV will be much smaller than 90°, and you'll have a very narrow vision. You'll see everything more detailed though and there will be less depth, so this is the same as zooming in:



If the direction vector is shorter than the camera plane, the FOV will be larger than 90° (180° is the maximum, if the direction vector is close to 0), and you'll have a much wider vision, like zooming out:



When the player rotates, the camera has to rotate, so both the direction vector and the plane vector have to be rotated. Then, the rays will all automaticly rotate as well.


$\vec{pos}+\vec{dir}+\vec{plane}$
$\vec{pos}+\vec{dir}$
$\vec{pos}$
$\vec{pos}+\vec{dir}-\vec{plane}$

To rotate a vector, multiply it with the rotation matrix

```
[ cos(a) -sin(a) ]
[ sin(a)  cos(a) ]
```

If you don't know about vectors and matrices, try to find a tutorial with google as having a solid grasp of linear algebra can be very useful for 3D graphics in general.

Reference so you can continue reading about this amazing topic here: Raycasting

Part 2: The Raycast Engine (In Progress)

2.1: Classes

2.1.1: each class. Etc 2.1.2, etc

2.2: API

2.2.1: Talk about each part of the api what it accomplishes

Part 3: The Raycast Editor (In Progress)

3.1: the different components of the editor like creating a world etc

3.2: the other components that stuff etc