Cub3D: Raycasting

Y.P.Smeding

September 2023

Introduction

The purpose of this document will be to explain the math behind ray-casting needed to complete the Cub3D project from the 42 Cursus.

Ray-casting is necessary to determine the distance between the current position in the map, which we will refer to as the player position from here on and the walls that the player can see. We need this distance to determine how high the wall we will draw on the window needs to be to get an accurate representation of the map passed to the program.

The following document only highlights some methods you can use to for this project but it is far from being the only way to do it. I also strongly encourage you to seek more information about the mathematics used as I only give brief recaps of some of the subject needed for the project but if you haven't done much trigonometry or vector manipulations it might be useful to look at some more detailed explanations than the ones given below.

The necessary variables.

Before we start casting our rays, we need to have stored some values:

- The players x position, x_pos.
- The players y position, y_pos.
- The angle of the direction in which the player is looking, α.
- The unit vector in the direction that the player is looking, $\overrightarrow{u}$.

In the rest of this document we may use the names $xpos$, $ypos$, α and $\overrightarrow{u}$ to refer to these variables.

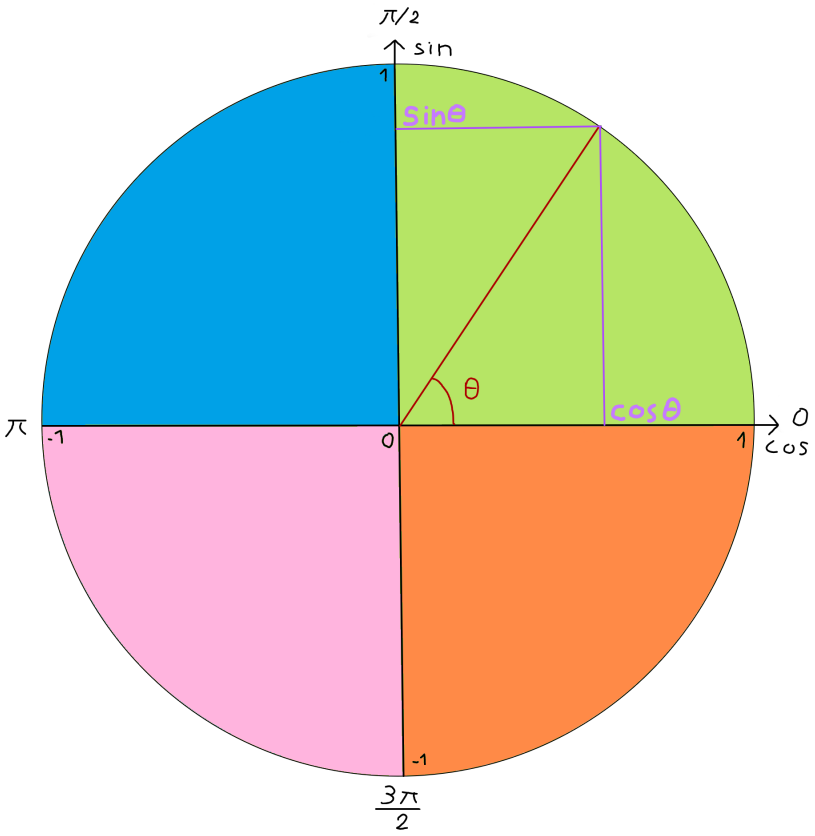
The players position.

Although it may seem unconventional we will us that the top left corner of the map is the point with coordinates $(x, y) = (0, 0)$, then the x coordinate increases when moving the the right and the y coordinate increases moving down. Of course, you may choose a different coordinate system but we chose this one because we stored the map in 2 dimensional array such that at the array position [0][0], we store the top left corner of the map.

The angle of the player's point of view

This is the angle between the direction in which the player is looking and the half line starting at the player's position going in the positive x direction. The smart thing to do would be to make sure that this angle always stays in the same interval of length 2π to avoid overflow at in your program. One could for example keep the angle in the interval $[0, 2\pi)$, which means that when the player rotates, and the angle is lower than 0, we add 2π to it or when the angle is larger or equal to 2π, we subtract 2π from it.

We note that we use the trigonometric convention for the direction in which the angle increases which means that it increases in the counter clockwise direction. Also, you are free to work with angles in degrees or radians but if you use values in degrees you need to convert them to radians when you use the sine and cosine functions of the math library in c as these take value in radians and you will get wrong results if you don't convert your angles. Below, I include a drawing of the trigonometric circle with the information necessary to complete the project.



On the image, we have put the value of the angle in the direction of the four cardinal points and we see that they increase in the counterclockwise direction. We also observe that if you rotate by an angle of 2π, you end up at the same position on the circle and that is why we can as we said earlier just take an interval of length 2π and keep our angle value inside of it.

As can be observed, the circle is divided into 4 quarters. These are called quadrants and conventionally, we call the green one the first quadrant, the blue one the second quadrant, the pink one the third quadrant and finally the orange one the fourth quadrant. For each angle within a quadrant, we know that the signs of the cosine and sine will be the same. In the upper quadrants, so the first and the second, the sine will always be positive and in the third and fourth the sine will always be negative. For the cosine, we have that it is positive in the first and fourth quadrant and negative in the other two.

The image also includes a visual representation of what the cosine and sine of a function are. We have drawn a line at an angle θ in the first quadrant. To find the cosine we take the point where the red line touches the edge of the circle and from this point we project a horizontal line (in purple) to the vertical line (in black) and we project a vertical line (in purple) to the horizontal line (in black). The sine is the value at which it touched the black vertical line, and the cosine is the value where the projection touched the black horizontal line.

We observe that in the circle we have marked the left most point where the horizontal line touches the edge of the circle as -1, the center as 0 and the right most point where the horizontal line touches the edge of the circle as 1. This indicates the value of the cosine at $-\pi$, $\frac{\pi}{2}$ or $\frac{3\pi}{2}$ and 0 respectivly. Similarly the -1, 0 and 1 on the vertical line represent the sine value of the angles $\frac{3\pi}{2}$, π or 0 and $\frac{\pi}{2}$.

The unit vector in the direction of the player's point of view

We begin with a little reminder of what vectors are.

Since we are only interested in the direction on the plane of the map, the vector will only have 2 dimensions. One entry holding the x coordinate and one for the y coordinate. For example, we could have the 2 dimensional vector

$$\overrightarrow{v} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \end{bmatrix}$$

From the player's position (x, y), we can also have a vector from the origin $(0, 0)$ to where the player is. If we want yo change the player's position in the direction of the vector described above, we can simply add

the x entries and the y entries to obtain the new position. So if we say that our player is at the coordinates $(4, 3)$ and we want to add the vector $\overrightarrow{v}$ to it, we can make the player's position into the following vector

$$\overrightarrow{p} = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$$

and perform a vector addition to get the new position

$$\overrightarrow{p} + \overrightarrow{v} = \begin{bmatrix} 4 \\ 3 \end{bmatrix} + \begin{bmatrix} 2 \\ 5 \end{bmatrix} = \begin{bmatrix} 6 \\ 8 \end{bmatrix}.$$

Thus, now the player is at position $(6, 8)$.

A unit vector is a vector of length one meaning that if I add this vector to my current position, I will be at distance one of my original position. This is very useful because it also means that if I multiply this vector by a scalar representing a distance and add this new vector to my position I can move that distance in the direction of the vector.

Since the player begins looking directly north, south, east or west, we can easily get the initial unit vector in the direction of the pov.

Using the coordinate system we described earlier, we have that if the player initially is looking north, the unit vector int the direction fo the player's pov is the vector

$$\begin{bmatrix} 0 \\ -1 \end{bmatrix}.$$

Indeed, we are looking in the negative y direction which means that if we would take a step of size 1 in that direction, the new y coordinate would be the old one minus 1 and we would not change the x coordinate.

Now, we want to be able to look in any direction of the plane, so we need to be able to get the unit vector for any direction. There are two different ways this could be done.

The first way is to use a rotation matrix. The rotation matrix in two dimensions is the matrix

$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}.$$

If we multiply a vector by this matrix we obtain that vector rotated at an angle θ in the counterclockwise direction. This means that whenever, we rotate the player in our program we would multiply our unit vector by the matrix above using the angle with which we rotated the player. We note that multiplying a vector by this matrix does not affect the length of said vector so if the length of the vector is one before the multiplication, it will still be once the operation is performed. As a reminder, we show how a multiplication with a matrix and a vector works

$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} cos\theta \cdot x - sin\theta \cdot y \\ sin\theta \cdot x + cos\theta \cdot y \end{bmatrix}.$$

Alternatively, we can use the angle of the direction of the point of view to determine the unit vector in that direction. Let say that the angle is given by α, then we simply use the sine and cosine of α to get the unit vector in the required direction. This would give the unit vector

$$\begin{bmatrix} cos\alpha \\ -sin\alpha \end{bmatrix}.$$

We add the minus in front of the sinus because we inverted the y direction from the conventional coordinate system be careful about the signs if your are using another one.

Casting rays

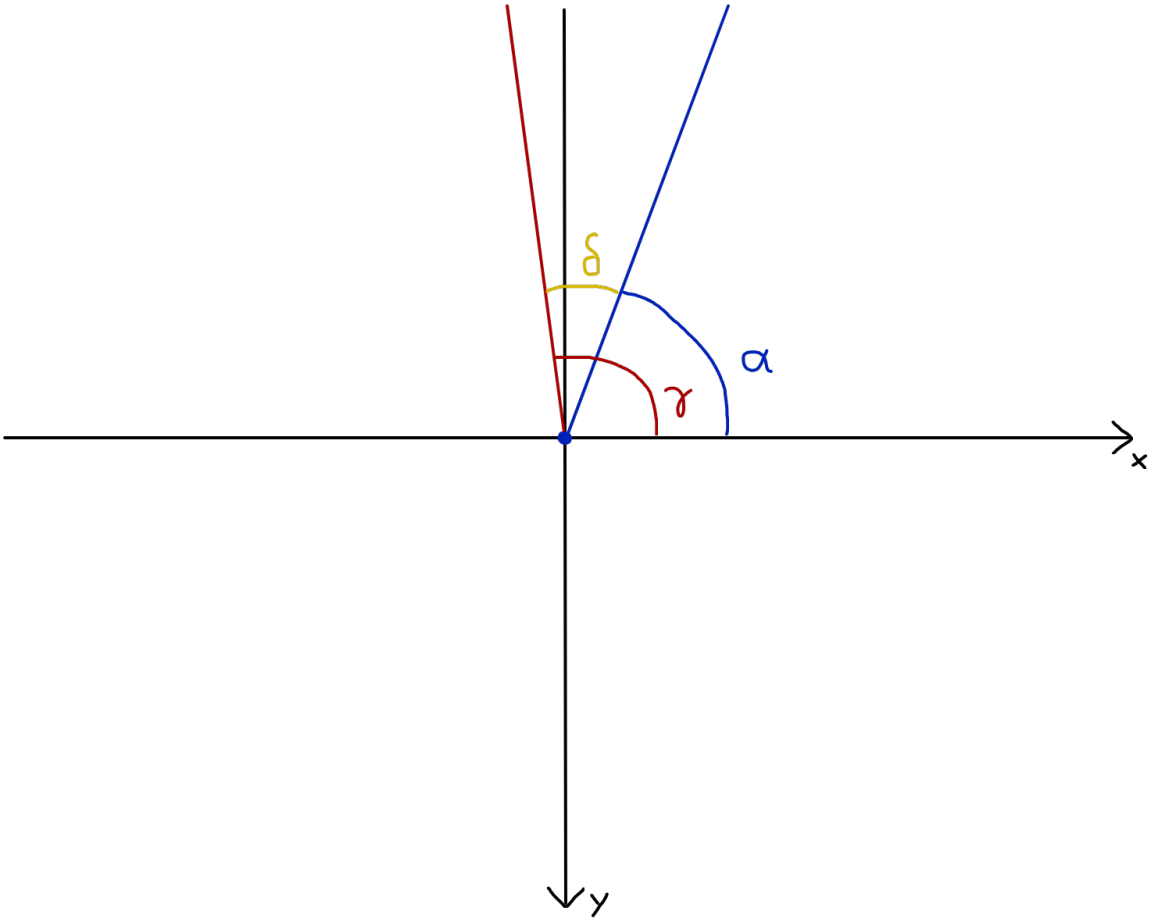
Now that we have all the necessary variables, we can start casting rays. We want to cast a ray for every column of pixels of the image of our game. We are only interested in the distance between the player an the wall it touches on a 2 dimensional plane so we only need one ray for each columns. You are free to choose the width (i.e. the number of columns of pixels) of your image, so henceforth we will use WID to refer to the number of pixels in every row of the image.

Now, we also need to decide the amplitude we want to use for our angle of sight. This amplitude determines how much to the right and left of our pov direction we can see. We took an angle of $\frac{\pi}{2}$ or $90°$, so that is what I will use in the example but you may choose another angle if you wish. A right angle as amplitude means that we can see $\frac{\pi}{4}$ radians to the left and $\frac{\pi}{2}$ to the right of our direction angle, so we need to cast WID rays at an angle of $\alpha + \delta$ where δ goes from $\frac{\pi}{4}$ to $-\frac{\pi}{4}$.

In the following sections we explain how to divide this angle such that we avoid a fish eye effect and how to cast a ray and determine at which distance it touches a wall.

Casting a single ray

Given a value for δ as described above, we can obtain the angle of our ray by adding it to our direction angle α. We will call the angle of our ray $\gamma = \alpha + \delta$. In the image below, there is a visual representation of this where the blue line represents the direction of the player's pov and the red line is the ray we cast at an angle δ from the player's pov.

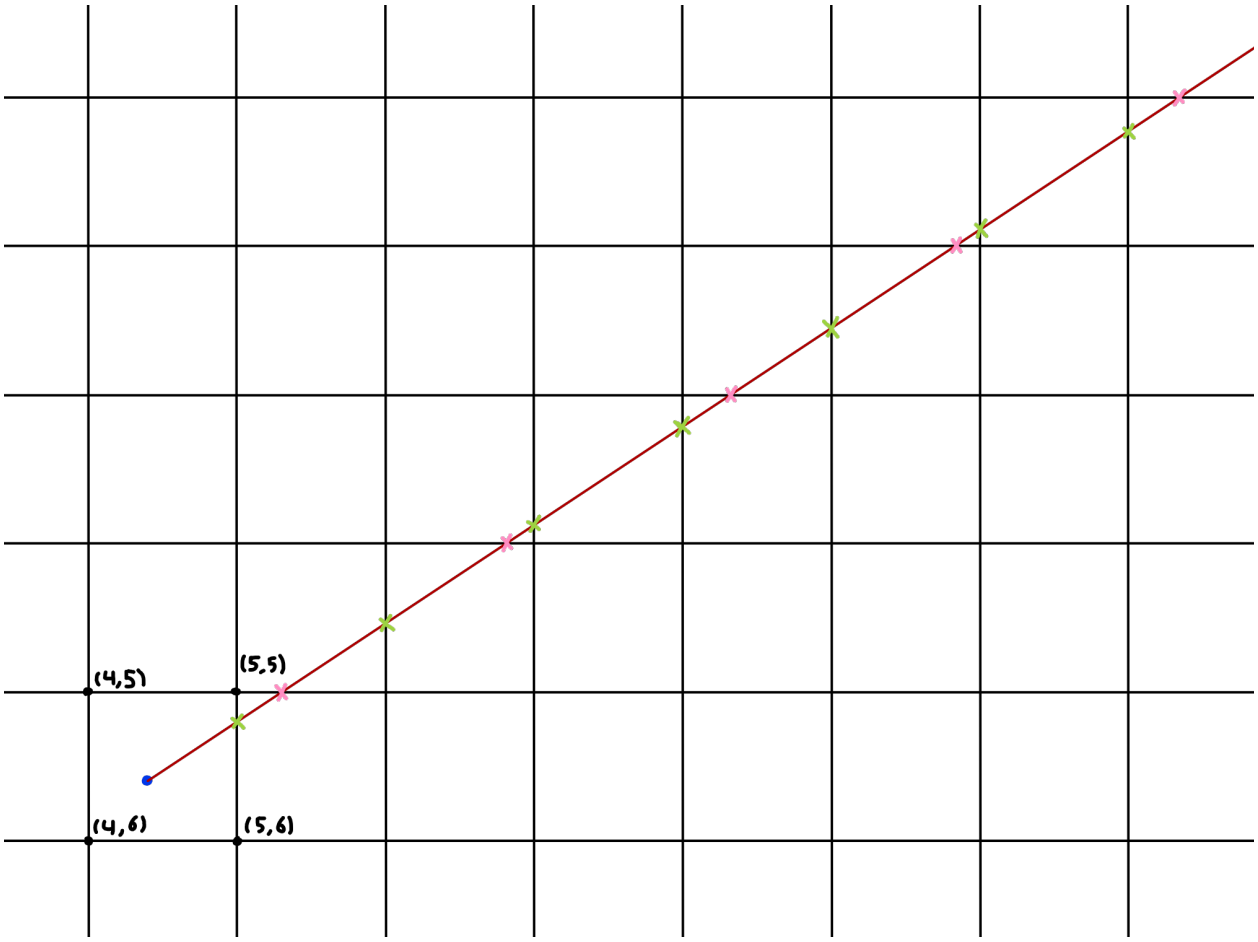


To determine the distance at which the ray touches a wall, we use the fact that a wall always starts or ends at an integer value. In the map given to the program we have that 1's represent walls, 0's represent open floor and spaces represent areas outside of the map. So if we take that the top left character represents what is inside the range $[0, 1] \times [0, 1]$ of the grid, we have that each character always represents what is inside a square with sides of length 1 that has it's corner at integer coordinates.

This means we only need to check the points where the ray crosses coordinates where the x coordinate is an integer (we will call this x-crossings) and where the y coordinate is an integer (we will call this y-crossings). Then we only need to check if in the next square that the ray is gonna pass whether it is a wall, in which case we stop and note the distance or not in which case we continue casting checking the crossings. In the image below, we have a player at the blue dot that has coordinates $(4.4, 5.6)$ and a ray represented by the red line. the x-crossings are marked by the green crosses and the y crossings are marked by the pink crosses.

One may observe that the distance between the green crosses is always the same and similarly the distance between any two pink crosses is also equivalent. So to compute the distance between the player and the nearest wall in the direction of the ray, we can check when the first x-crossing touches a wall and when the first y crossing touches a wall and take the smallest of these two distances. For the x-crossing what we would have to do is find the distance between the player (blue dot) an the nearest x-crossing (green cross), check if it is a wall and if it is not, we keep adding the distance between two green crosses in the direction of the ray until we find a wall. We work in the same way to find the distance for the y-crossings.

In the image below, we can see the triangles we use to determine the distances. The distance from the player to the first x-crossing is the hypotenuse of the green triangle and the distance between two x-crossings is the hypotenuse of the orange triangle.

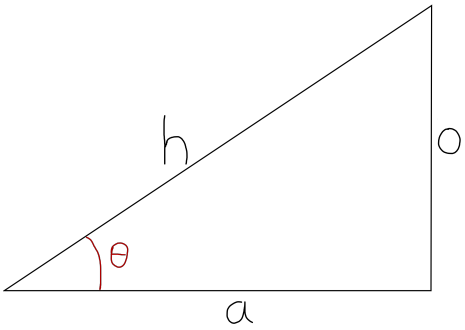


The green and orange triangles are similar triangles, meaning that they have the same angles. In this case the angle highlighted by the red arcs in the triangles are the angle of the ray in the trigonometric circle. We also know the length of the horizontal side of both triangles. For the green triangle, we use the player's position $(4.4, 5.6)$ and see that the length of the horizontal side is 0.6. Let's call this distance to the next integer x-value x_d. For the orange triangle the length of the horizontal side is 1 because that is the size of the side of one square in the grid.

Using sine and cosine, we can compute the hypotenuse of a right-angled triangle with one extra angle and the length of one of the other sides. Take the image to the right, then we have

$$sin\theta = \frac{o}{h} \Rightarrow h = \frac{o}{sin\theta}$$

$$cos\theta = \frac{a}{h} \Rightarrow h = \frac{a}{cos\theta}$$

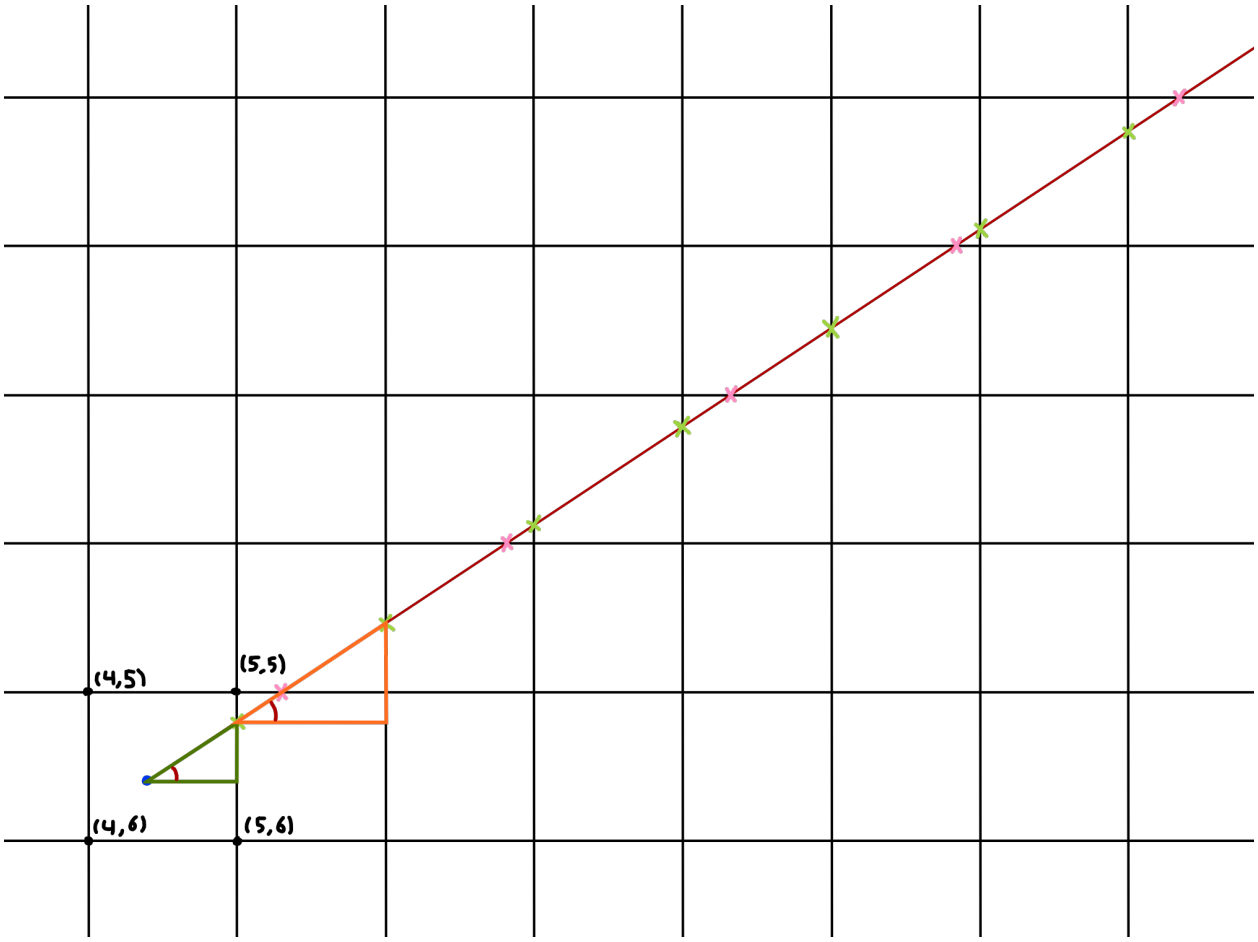


So given that the angle γ of the ray gives the angle of the triangles and that we have the adjacent sides to the angle, we use cosine to get the hypotenuse of the green triangle that we call $x_{first\ step}$ and the hypotenuse of the orange triangle that we call x_{step} as follows

$$x_{first\ step} = \frac{x_d}{cos\gamma}$$

$$x_{step} = \frac{1}{cos\gamma}$$

Now for the y-crossings, we have a similar way of getting the distance to the first y-crossing and the distance between two y-crossings. We will denote these distances $y_{first\ step}$ and y_{step}.



Similarly to the previous image, we have that the distance $y_{first\ step}$ is the length of the hypotenuse of the pink triangle and y_{step} is the hypotenuse of blue triangle. Also the red highlighted angles are still the same as in the green and orange triangles from earlier. Contrary to the triangles for the x-crossings, we now don't know the length of the horizontal side of the triangles, however we do know the length of the vertical sides. The vertical side of the pink triangle has size $y_d = 0.6$ since the player is at the position $(4.4, 5.6)$ and the vertical side of the blue triangle has length 1since it has the same size as a side of one of the squares of the grid.

So using the sine formula from before, we get

$$y_{first\ step} = \frac{y_d}{sin\gamma}$$
$$y_{step} = \frac{1}{sin\gamma}$$

Now, to add these steps to the player's position, we need to have the unit vector in the direction of the ray, multiply it by the length we want to move in that direction and add it to the player position vector. To get the unit vector in the direction of the ray, that we denote $\overrightarrow{r}$ we can either use the vector $\overrightarrow{u}$ and multiply int with the rotation matrix using the angle δ that gives the angle between the player's pov direction and the ray direction or we immediately use the angle of the ray γ to get the unit vector

$$\overrightarrow{r} = \begin{bmatrix} cos\gamma \\ -sin\gamma \end{bmatrix}.$$

So if we have the vector of the player's position

$$\overrightarrow{p} = \begin{bmatrix} 4.4 \\ 5.6 \end{bmatrix}$$

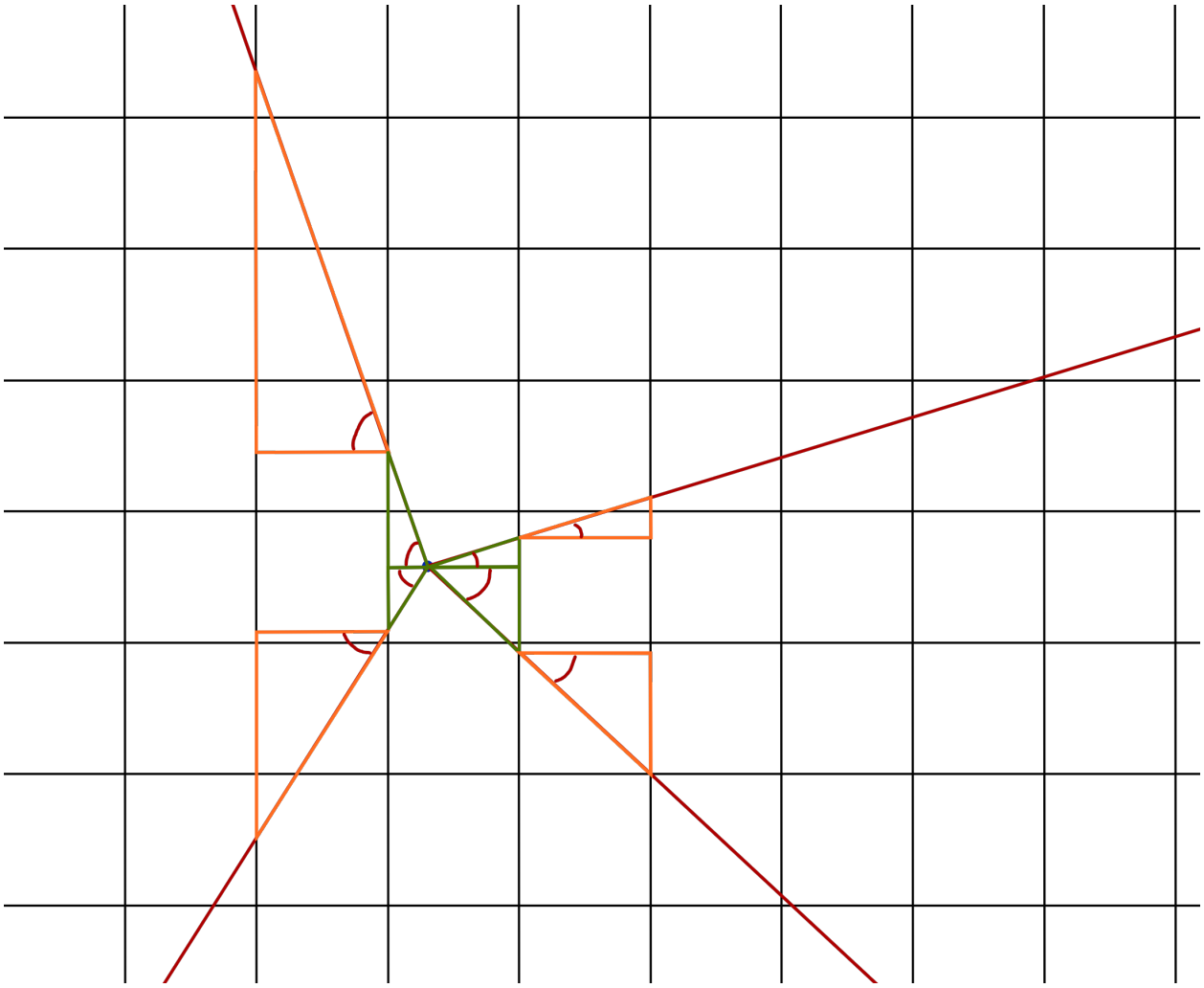
and we want to add the first step to get to the first x-crossing we do

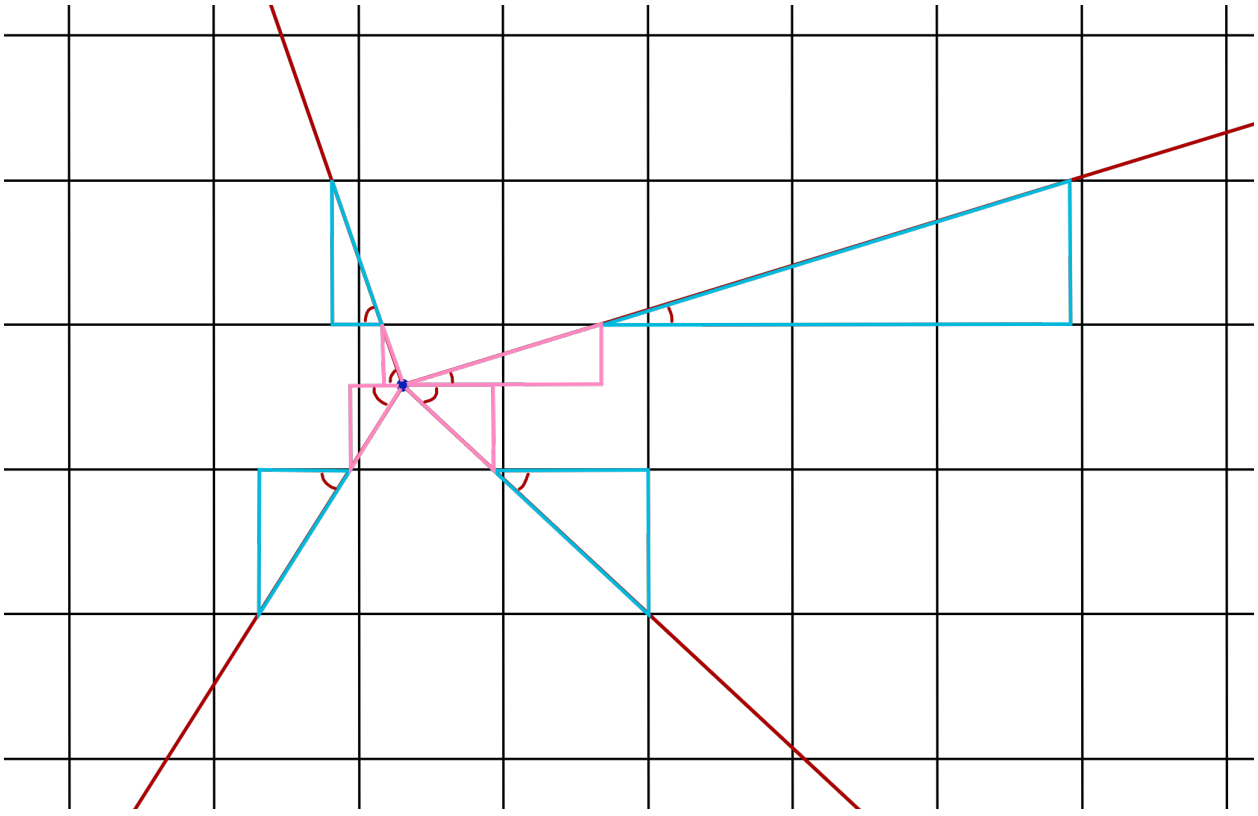
$$\overrightarrow{p} + x_{first\ step} \cdot \overrightarrow{r} = \begin{bmatrix} 4.4 + x_{first\ step} \cdot cos\gamma \\ 5.6 - x_{first\ step} \cdot sin\gamma \end{bmatrix}$$

which gives us the coordinates of the first x-crossing. Now, we would check whether the square on the right of this point is a wall or not. If it's not, we proceed in a similar way but with the x_{step} value to check the next crossing and the next until we reach a wall or until we go outside of the map.

In the example, we just saw, we had a ray with an angle in the first quadrant. As we will observe now, we can use a similar method for rays in the other quadrants but we will observe that there are some details we need to take into consideration based on which quadrant we are looking at.

Looking at the images below, we see that the angle we need to compute the hypotenuse of the sides of the triangle, is not always the angle of the ray in the trigonometric circle but the angle of the ray with the horizontal half line that goes in the same x-direction as the ray.





We can observe that the angle we need to use the formulas from before depends on the quadrant in which the angle γ of the ray is located. Let's call the angle of the triangle ψ. If the ray is in the first quadrant, we simply have $\psi = \gamma$. If the ray is in the second quadrant, meaning that it's value is in between $\frac{\pi}{2}$and π (check in the trigonometric circle above if you have any doubts), we take $\psi = \pi - \gamma$. For the third quadrant, we need $\psi = \gamma - \pi$ and finally for the fourth quadrant, we use $\psi = 2\pi - \gamma$.

We also observe that the values of x_d and y_d depend on the quadrant of the ray. The distance x_d represents the horizontal distance between the player and the vertical line in which the ray has its first x-crossing. So in the first and fourth quadrant we want the difference between x_{pos} and the smallest integer bigger than x_{pos}, so $x_d = 1 - (x_{pos} - \lfloor x_{pos} \rfloor)$ where $\lfloor x_{pos} \rfloor$ indicates the integral part of the number x_{pos}. In the second and third quadrants, we can simply take $x_d = x_{pos} - \lfloor x_{pos} \rfloor$ as we want the difference between x_{pos} and the biggest integer smaller than x_pos. In a similar fashion, we can get that in the first and second quadrant, we have $y_d = y_{pos} - \lfloor y_{pos} \rfloor$ and in the third and fourth quadrants, we take $y_d = 1 - (y_{pos} - \lfloor y_{pos} \rfloor)$.

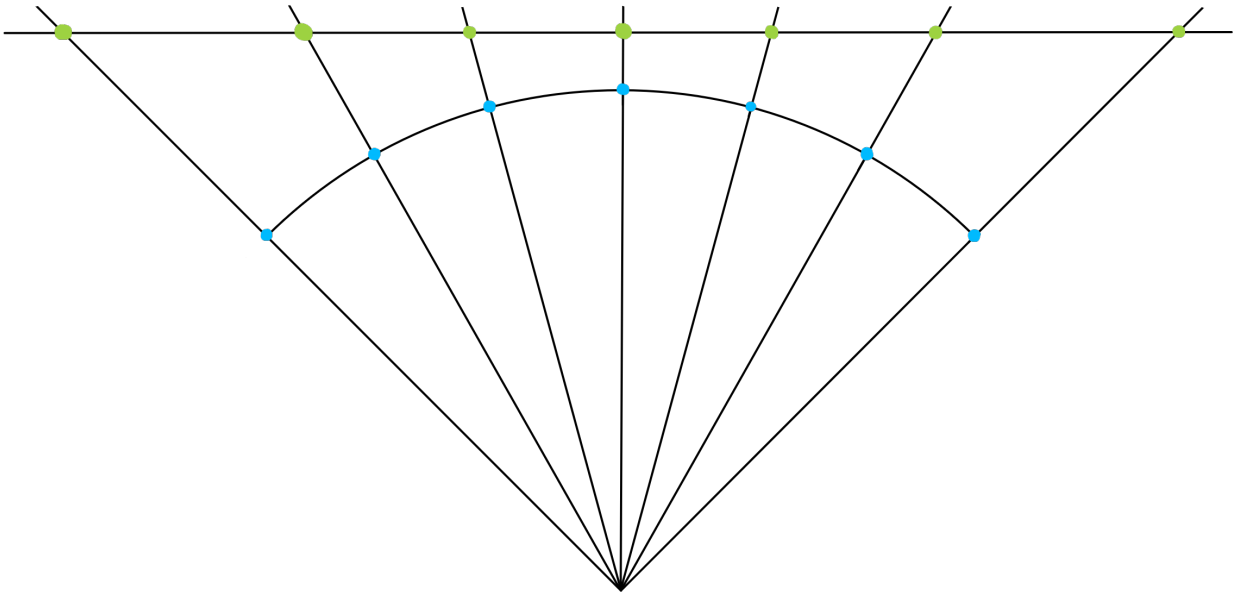
Let's call the distance at which a x-crossing impacted a wall $dist_x$ and the distance at which a x-crossing impacted a wall $dist_y$. We want to take the smallest of these two as that is the wall that the player sees first in that direction. Let's call the smallest of these distances $dist_h$.

Now, to avoid the having a fish-eye effect, we actually need to take the distance from the impact point to the line perpendicular to the vector of direction of the player. this is equivalent to taking the cosine of the angle difference between the vector of direction and the ray and multiplying it by the distance at which the ray impacts a wall. So the distance we will use to scale our walls will be

$$dist = dist_h \cdot cos\delta$$

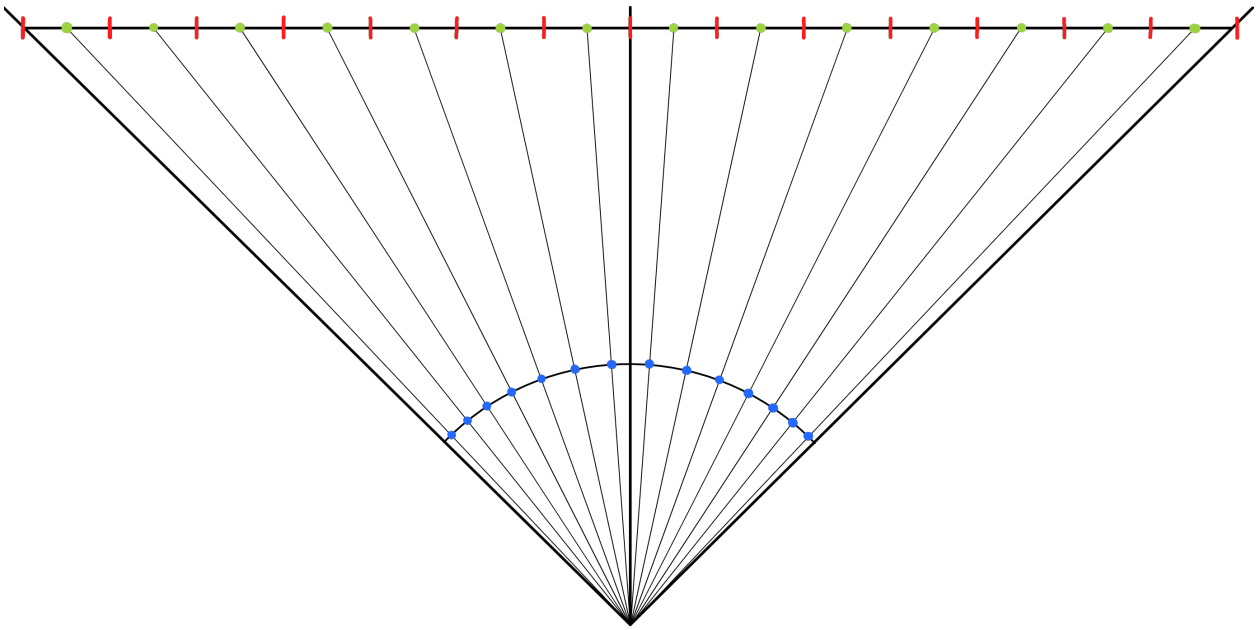
Dividing the angle of sight

To divide amplitude of the angle of sight we cannot just divide the angle by the number of rays and cast a ray at every cut. increasing (or decreasing) the angle of the ray by the same number would give the following situation.



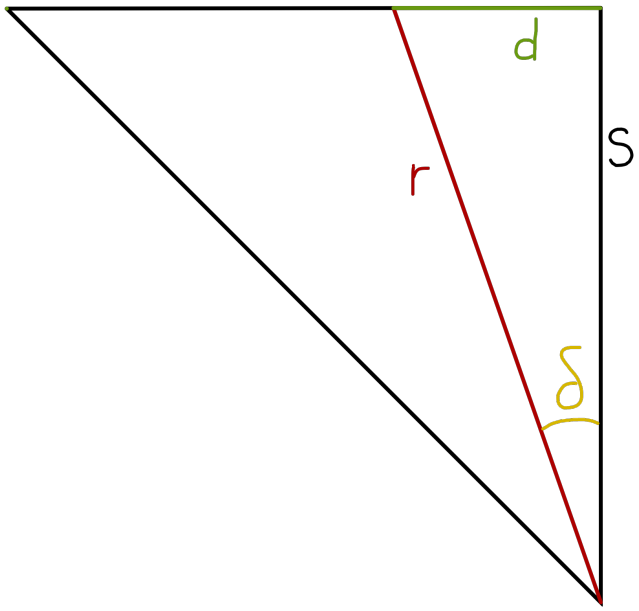
We observe that the arcs between the blue dots all have the same length however when we project this onto a flat surface, we see that the green dots are not equidistant on the line. This will cause your image to have a fish-eye effect where lines that should be straight appear curved.

Thus we want to cast rays such that their intersections with a flat surface perpendicular to the direction of the player's point of view are equidistant.



In the image above, the segments delimited by the red lines are equidistant. We observe that the arcs between the blue dots do not all have the same length.

Using the image below, we will see how to obtain the angle between the ray and the vertical axis based on the knowledge we have of our triangles.



$$\delta = arcsin\left(\frac{d}{r}\right) = arcsin\left(\frac{d}{\sqrt{d^2+s^2}}\right)$$