Raycster with SDL and C++

Sam Laister

November 2022

1 Introduction

Hello, Welcome to my writeup on a recent passion, a raycaster! I've been interested in DOOM's (1996) renderer for some time now. However jumping that far into the deep end isn't my style. So let's start with it's predecessor, Wolfenstein 3D (1992).

They say that the best way to learn is to teach. So I'm going to try and teach you, providing code snippets from my own program. The source code can be found on my github page[1]

2 The Basics

I want to split this writeup into 3 sections, the maths, translating that to code, and finally the SDL implementation. Lets start with the basics of the Maths involved.

Say you have a x and y coord representing a player, the goal is to shoot rays from left to right and then draw vertical pixels depending on how far the ray has travelled. But this arises questions such as how do we calculate if a ray has hit a wall? How do we know how far the ray has travelled? How do we know what direction to shoot the ray? etc. I'm going to present a simplified situation and break it down.

2.1 Setting the Scene

Imagine you have a simple 2D grid, lets say 5 by 5. The player is in the middle of the grid, so (2.5, 2.5). In said grid, a 1 represents a wall, and a 0 represents an empty space. As demonstrated below.

[1] https://github.com/Soup666/SDL-Raycaster_v1

1	1	1	1	1
1				1
1		P		1
1				1
1	1	1	1	1

The way we're going to create a raycaster, is by shooting different rays for horizontal and vertical walls, then drawing the smallest ray as either a vertical wall or horizontal wall, respectfully. The way we define a vertical wall or horizontal wall is simple. Take a 1 from the above diagram, A vertical wall is a wall that is either to the left or right and spans the length of the unit. A horizontal wall is obviously a wall that is either above or below. Wall's in this raycaster don't have a normal variable to take into account.

For this example, we're going to set the player's direction to +59.04 deg, so facing north east. And we'll shoot a single ray the same direction.

2.2 Horizontal Ray

1	1	1	1	1
1				1
1		P		1
1				1
1	1	1	1	1

Okay lets get into some maths. We want to calculate the nearest wall from the players position, (2.5,2.5). In pixel position, this would be (250,250). Assuming 0,0 is top left, and we're looking east, we can the nearest horizontal boundary. We know:

$$\begin{aligned} pa &= 59°(1.031rad) \\ px &= 250 \\ py &= 250 \end{aligned} \tag{1}$$

We can then calculate the ray's first x and y.

$$\begin{aligned} unitW &= 500/5 = 100pixels \\ ry &= floor(playerY/unitW) * unitW - 0.001 \\ rx &= playerY * \frac{1}{\tan 1.031} + playerY \end{aligned} \tag{2}$$

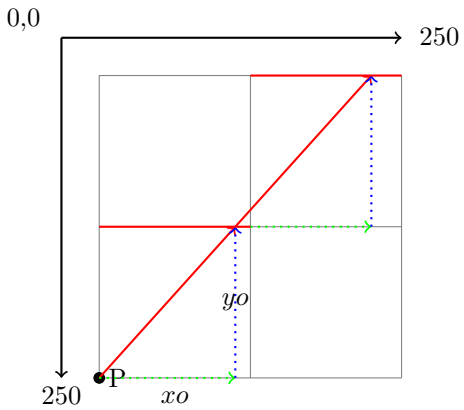
NOTE the -0.001 is to slightly push it in the bounds of the next square, so we don't check the current players square. We only do this if the player is looking in the negative y direction.

Giving us $(279.96, 200)$. Great! However we can do better. A grid is made up of squares, so what if we calculated an offset as well to repeat the proccess. Let me give you an example.

We introduce ox and oy. As per,

$$\begin{aligned} yo &= -100 \\ xo &= -yo * \frac{1}{\tan 1.031} \end{aligned} \tag{3}$$

Giving us $yo = -100$ and $xo = 38.346$. Let's demonstrate this on a graph.



Here, you can see we intersect two horizontal lines. Perfect. We'll eventually be checking if the square north of the intersection is a wall, and if it is we return the ray length.

2.3 Multidirectional Rays

Ok, so so far we have a working algorithm for calculating the angle of a ray based off the players current rotation! Great, as long as the player is looking north east. The formulae changes slightly depending on the angle of the player. For example, we substracted -0.001 so the ray landed within the box, however if the player is looking 180 deg that no longer functions as intended.

To solve this, we can set bounds for different behaviours. We'll start by assigning a *rayangle* variable

$$ra = pa + offset$$

We can decide which direction the player is looking by applying cos or sin to this variable.

$ra \leftarrow pa$
if $\cos ra \geq 0.001$ **then**
    ...
    // Looking left
end if
if $\cos ra \leq -0.001$ **then**
    ...
    // Looking right
end if
$ra \leftarrow pa$
if $\sin ra \geq 0.001$ **then**
    ...
    // Looking Up
end if
if $\sin ra \leq -0.001$ **then**
    ...
    // Looking Down
end if