42

The meaning of my Life

## Cube3d

### Como começar

1 - Ver github do Balsa para ver README, que recomenda ver o mesmo link que o Lucas Marques: <https://lodev.org/cgtutor/raycasting.html>

2 - Ver Código do Balsa

3- Ver Vídeo Raycasting:

<https://www.youtube.com/watch?v=gYRrGTC7GtA&list=PLCWsH9Tj9oWyDM4W43VMj5yo2PdyYMGst>

4- Ver vídeos e link sugeridos pelo Lucas Marques:

<https://lodev.org/cgtutor/raycasting.html>

<https://www.youtube.com/playlist?list=PLLFRf_pkM7b6rBRoTOW64NKdltCLQNpW5>   
  
5- Ver link sugerido pelo Miguel Biltes:

<https://medium.com/@afatir.ahmedfatir/cub3d-tutorial-af5dd31d2fcf>

6 - Ver githubs do Heitor e Filipa

<https://github.com/HeitorMP/42-cub3D>

<https://github.com/fiborges/42SchoolProjects/tree/main/Cub3D>

7 – Organização

7.1. - Tendo em conta a organização do Balsa, as pastas ficaram divididas assim:

1st part (ficou para o João Silva)

- Main

- Parse

- mlx

- frees and exits

2nd part (ficou com Rodrigo Antunes):

- raycasting (chama input\_handler)

- move (contem a input\_handler)

- draw (é chamada no raycasting)

7.2. – O que vou fazer diferente do Luís

1º No main colocar:

- init\_mlx\_and\_textures

- parse\_file (2as partes:

. tokenizer é uma função que divide o texto num conjunto de carateres que tem valor para mim. Carateres ou conjuntos de carateres que têm valor. Daí o nome tokenizer, pq o token é a unidade que tem valor. Unidade ou conjunto. Este é o princípio.  
O lexer vai analisar a informação e decidir o que faz com ele.  
Basicamente o parser obtém a informação e trata-a para a entregar ao lexer.

O lexer analisa e interpreta a informação, decidindo o que fazer com ela de seguida.

- raycasting

- frees & error\_exits

2º A pasta mlx chamar-se Init

3º Meter os frees & error\_exits numa pasta frees&exits

4º Na validação do mapa há quem tenha usado o floodfill outra vez- Balsa não usou. Demorou 1 dia a fazer a validação do mapa e o parser.

-----------------------------------------------------------

No contexto de jogos e gráficos, o termo "spawn" refere-se ao ato de criar ou gerar um objeto, personagem ou entidade em um determinado local dentro do ambiente do jogo. O termo "spawn" é frequentemente usado em jogos de tiro em primeira pessoa (FPS), como os jogos de tiro com base em mapas, onde os jogadores podem "nascer" ou "aparecer" em um local específico no início de uma partida ou após morrerem.

No contexto do raycasting, que é uma técnica usada para renderizar gráficos 3D em jogos de computador e simuladores, o termo "spawn" pode ser usado para se referir ao processo de criar ou gerar um objeto, entidade ou inimigo em um determinado local dentro do ambiente do jogo. Isso pode ser feito usando algoritmos de spawn ou scripts de spawn, que determinam as condições e os critérios para a criação de novos objetos ou entidades no jogo.

Em geral, o termo "spawn" refere-se ao ato de criar ou gerar algo em um ambiente de jogo, e seu uso específico pode variar dependendo do contexto e das características do jogo em questão.

A computer screen with text and symbols

Description automatically generated

EWS e N sao a posicao do player no mapa que me e passada no ficheiro.

[14:08, 23/02/2024] Luís Balsa 42 School: Se lhe passar N o vector de direcao fica (0, -1)

[14:09, 23/02/2024] Luís Balsa 42 School: S fica (0, 1)

Esta a determinar a direcao para onde esta virado o jogador.

No inicio pode comecar com 4 direcoes diferentes.

Essa direcao e determinada pela letra que esta no mapa para representar o jogado.

Eu criei uma struct para guardar vectores.

### Tutorial <https://lodev.org/cgtutor/raycasting.html>

#### Untextured Raycaster

##### Resumido

1 – Define a posição do player (posX), a camera plane (planeX), e o direction vector (posição original dirX dir Y).

2 – Calcula o valor do X in camera space, se é 0 (centro ecrã), -1 (lado esq ecrã) ou 1(lado direito ecrã) cameraX

3 – Calcula a direção do raio (rayDirX e rayDirY)

A diagram of a triangle

Description automatically generated

4 – Verifica a nossa posição no mapa (mapX e mapY, sendo que int mapX = int(posX e int mapY = int(posY))

A grid with blue squares and green dots

Description automatically generated

5 – Cálculo da distância da posição atual até ao próximo x or y-side, e cálculo desse próximo x or y-side até ao próximo x or y-side (sideDistX, sideDistY, deltaDistX, deltaDist)

length of ray from current position to next x or y-side

//length of ray from one x or y-side to next x or y-side

A diagram of a line

Description automatically generated

6 – Cálculo de se avançamos para o lado esq ou dir (stepX – movimentamo-nos para os lados, esq (-1) e dir (1)), ou para cima/baixo (stepY – movimentamo-nos para cima (1) ou para baixo (-1)), tendo por base isto: The DDA algorithm will always jump exactly one square each loop, either a square in the x-direction, or a square in the y-direction.

7 - Cálculo de que parte do square foi atingida: o lado X ou o lado Y (Side

...é 0 se for X e 1 se for Y), e se foi hit passa a 1.

By x-side and y-side, I mean the lines of the grid that are the borders between two squares.

8 – Cálculo do DDA: O DDA serve para sabermos qual o próximo quadrado na direção do raio (isto é, o próximo quadrado para onde vamos). A comparação **sideDistX < sideDistY** no algoritmo DDA é usada para decidir se o raio atravessa primeiro uma linha **vertical** (eixo X) ou uma linha **horizontal** (eixo Y).

* **sideDistX**: Distância do raio até o próximo lado vertical.
* **sideDistY**: Distância do raio até o próximo lado horizontal.

Se **sideDistX < sideDistY**:

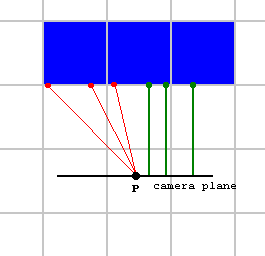
* O raio atinge uma linha vertical, atualiza-se **sideDistX**, move-se no eixo **X** (**mapX**), e define-se **side = 0**.

Se **sideDistY < sideDistX**:

* O raio atinge uma linha horizontal, atualiza-se **sideDistY**, move-se no eixo **Y** (**mapY**), e define-se **side = 1**.

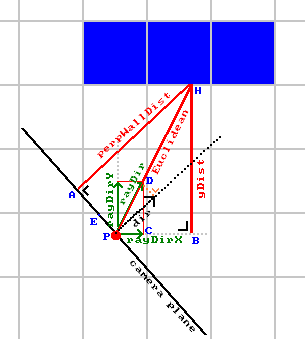
Isso permite calcular eficientemente o trajeto do raio no grid, determinando quais células ele atravessa.

9 – Evitar fisheye effect (effect you see if you use the real distance through Euclides distance, where all the walls become rounded, and can make you sick if you rotate) calculando a distância da parede até à camera plane (perpendicular distance we'll use instead of direct Euclidean distance).



10 – Cálculo da variável perpWallDist que será usada no DDA algorithm (que serve para encontrar o próximo square que atingimos) para calcular o tamanho do raio.

This perpenducular distance is called "perpWallDist" in the code. One way to compute it is to use the formula for shortest distance from a point to a line, where the point is where the wall was hit, and the line is the camera plane:

 A diagram of a line

Description automatically generated

11- Calculate height of line to draw on screen

12 - Choose wall color

13 - Timing for input and FPS counter

14 - readKeys(); - ler o input do teclado e mouse

This concludes the code of the **untextured raycaster**, the result looks like this, and you can walk around in the map:

A computer screen shot of a blue green and red cubes

Description automatically generated  
  
Here's an example of what happens if the camera plane isn't perpendicular to the direction vector, the world appears skewed:  
  
A blue and red rectangles

Description automatically generated

##### Detalhado

int main(int /\*argc\*/, char \*/\*argv\*/[])

{

double posX = 22, posY = 12; //x and y start position

double dirX = -1, dirY = 0; //initial direction vector

double planeX = 0, planeY = 0.66; //the 2d raycaster version of camera plane

double time = 0; //time of current frame

double oldTime = 0; //time of previous frame

screen(screenWidth, screenHeight, 0, "Raycaster");

while(!done())

{

for(int x = 0; x < w; x++)

{

//calculate ray position and direction

double cameraX = 2 \* x / double(w) - 1; //x-coordinate in camera space. the right side of the screen will get coordinate 1, the center of the screen gets coordinate 0, and the left side of the screen gets coordinate -1.

double rayDirX = dirX + planeX \* cameraX; // the direction of the ray can be calculated as was explained earlier: as the sum of the direction vector, and a part of the plane vector. This has to be done both for the x and y coordinate of the vector (since adding two vectors is adding their x-coordinates, and adding their y-coordinates).

double rayDirY = dirY + planeY \* cameraX;   
  
  
//which box of the map we're in

int mapX = int(posX); // mapX and mapY represent the current square of the map the ray is in.

int mapY = int(posY);

//length of ray from current position to next x or y-side

double sideDistX;

double sideDistY;

//length of ray from one x or y-side to next x or y-side

double deltaDistX = (rayDirX == 0) ? 1e30 : std::abs(1 / rayDirX); //  If rayDirX or rayDirY are 0, the division through zero is avoided by setting it to a very high value 1e30. // For the blue triangle (deltaDistX), one side has length 1 (as it is exactly one cell) and the other has length raydirY / raydirX because it is exaclty the amount of units the ray goes in the y-direction when taking 1 step in the X-direction

double deltaDistY = (rayDirY == 0) ? 1e30 : std::abs(1 / rayDirY); // For the green triangle (deltaDistY), the formula is similar.

double perpWallDist; // will be used to calculate the length of the ray.

//what direction to step in: x or y-direction (either +1 or -1)

int stepX; // The DDA algorithm will always jump exactly one square each loop, either a square in the x-direction, or a square in the y-direction. If it has to go in the negative or positive x-direction, and the negative or positive y-direction will depend on the direction of the ray, and this fact will be stored in **stepX and stepY**. Those variables **are always either -1 or +1**.

int stepY;

int hit = 0; //was there a wall hit? // Finally, **hit is used to determinate whether or not the coming loop may be ended**,

int side; //was a NS or a EW wall hit? // and **side will contain if an x-side or a y-side of a wall was hit**. If an x-side was hit, side is set to 0, if an y-side was hit, side will be 1. By x-side and y-side, I mean the lines of the grid that are the borders between two squares.

//calculate step and initial sideDist

if (rayDirX < 0) // ou seja, o raio vai pa esquerda. estamos a olhar pa esquerda. O raio está apontado para a esquerda.

{

stepX = -1; // a direção para onde estamos a ir é pa esquerda.

sideDistX = (posX - mapX) \* deltaDistX; // cálculo da distância do player até ao próximo X-side. Se rayDirX é negativo, sideDistX será a distância do player ao primeiro x-side (sideDistX) e é calculado usando a distância até o lado esquerdo da célula. Isso é feito pegando a diferença entre posX e mapX (a posição real menos a posição do grid), e multiplicando por deltaDistX (a distância proporcional para o próximo lado X). Ver explicação em sideDistX = (mapX + 1.0 - posX) \* deltaDistX; para perceber melhor.

}

Else // senão estamos a olhar pa direita...o raio vai pa direita

{

stepX = 1; // ...e a direção para onde estamos a ir é pa direita.

sideDistX = (mapX + 1.0 - posX) \* deltaDistX; // **Se** rayDirX **é positivo**: sideDistX usa o lado direito da célula, então pegamos (mapX + 1.0 - posX) e multiplicamos por deltaDistX. A razão para adicionar 1.0 é para mover a referência para o lado direito da célula.  
Vejamos, map (x,y) será o valor do lado esquerdo do quadrado. Ao somar 1 obtemos o valor do lado direito da célula. E a este é que devemos retirar o valor da posição do jogador, para que assim saibamos qual a distância do jogador até ao lado direito da célula. No caso do jogador estar a olhar po lado esquerdo, tal não é necessário, porque o valor do map(x,y) já é correspondente ao valor do lado esquerdo do quadrado. Logo, só necessitamos de subtrair a posição do jogador pelas coordenadas do map para obter a distância ao lado esquerdo (inicial, base) se o jogador estiver a olhar po lado esquerdo. Mesmo raciocínio se faz para o Y.

O deltaDistX é multiplicado e não adicionado porque o que queremos não é somar a distância q falta do nosso quadrado até ao próximo. Nós queremos é saber a direção do nosso raio. O Cálculo sideDistX = (mapX + 1.0 - posX) dá-nos a distância dentro da célula entre o jogador e o limite da célula para o lado em que ele está a olhar (no caso o direito), mas não nos dá a direção do raio, logo não nos diz em que direção o jogador está efetivamente a olhar. Logo, se não multiplicarmos por deltaDistX estamos a cometer pelo menos 2 erros: 1 – não sabemos a direcão real para onde o jogador está a olhar; 2 – o cálculo da distância não está correto pq não está a calcular a distância tendo em conta a direção certa. Logo, podemos estar a calcular na zona errada do quadrado e na direção errada, o que nos dará outro valor que não o correto.

-------

| / |

| / |

| . D |

-------

^ map2(1,0)

|

map (0,0)

O map é a posição “base” do quadrado (à qual necessita de ser somado 1 para obtermos o ponto mais à direita – o map2, para dps podermos subtrair e obter a distância do jogador à parede direita do quadrado). O ponto é a posição do jogador. Os / são a direção do raio. O D a distância do jogador à parede do lado direito (reta horizontal do jogador até à parede do lado direito). Para calcular a distância correta, deveremos multiplicar por DeltaDistX, q nos dará a reta na diagonal e não na horizontal. Assim, estaremos a calcular a “ real Euclidean distance.”

}

if (rayDirY < 0)

{

stepY = -1; // estamos a olhar para baixo.

sideDistY = (posY - mapY) \* deltaDistY; // cálculo da distância do player até ao próximo Y-side. O mesmo processo usado para o X é aplicado para sideDistY, dependendo do sinal de rayDirY, para calcular a distância até o primeiro lado no eixo Y (superior ou inferior). Ver explicação em sideDistX = (mapX + 1.0 - posX) \* deltaDistX; para perceber melhor.

}

else

{

stepY = 1; // estamos a olhar para cima.

sideDistY = (mapY + 1.0 - posY) \* deltaDistY; // Ver explicação em sideDistX = (mapX + 1.0 - posX) \* deltaDistX; para perceber melhor.

}

Exemplo Visual:

Imagine que posX está numa posição decimal dentro da célula (mapX). Ao fazer (mapX + 1.0 - posX), você está calculando a distância exata do ponto atual posX até o lado direito da célula, se o raio se mover para a direita. Essa distância é então ajustada para o valor em sideDistX para o próximo lado X que o raio cruzará.

Resumo

Esses cálculos preparam o DDA para **saber a direção do próximo "passo" do raio** e o **ponto exato onde ele cruzará o próximo lado da célula na grade**. O algoritmo DDA, então, usará esses valores para avançar de célula em célula no grid, identificando quando e onde o raio colide com uma parede.

//perform DDA: It's **a loop that increments the ray with 1 square every time**, until a wall is hit.

while (hit == 0)

{

//jump to next map square, either in x-direction, or in y-direction

if (sideDistX < sideDistY) // Se **sideDistX < sideDistY**, o raio atingirá uma linha **vertical** primeiro (lado X). Se **sideDistY < sideDistX**, o raio atingirá uma linha **horizontal** primeiro (lado Y). Sabendo que  **sideDistX**: A distância do ponto atual do raio até o próximo lado **vertical** (parede no eixo X) que ele atingirá.

 **sideDistY**: A distância do ponto atual do raio até o próximo lado **horizontal** (parede no eixo Y) que ele atingirá. Isso determina se o próximo movimento do raio será:

* **No eixo X (mapX)**: Movendo para a próxima coluna no grid.
* **No eixo Y (mapY)**: Movendo para a próxima linha no grid.

{

sideDistX += deltaDistX; // aqui já somamos o deltaDistX (ao invés de multiplicarmos, pq aqui já sabemos a direção do raio), pq queremos adicionar a distância até ao próximo quadrado.

mapX += stepX; // a posição do quadrado avança (para a esq se stepX for -1 e dir se 1) de acordo com o avanço do jogador na grid.

side = 0; // o lado X do quadrado foi atingido.

}

else

{

sideDistY += deltaDistY;

mapY += stepY; // a posição do quadrado avança (para baixo se stepY for -1 e cima se 1) de acordo com o avanço do jogador na grid.

side = 1; // o lado Y do quadrado foi atingido.

}

//Check if ray has hit a wall

if (worldMap[mapX][mapY] > 0) hit = 1;

}

1. **O que acontece após a comparação?**
   * Se o próximo lado a ser atingido é vertical (sideDistX < sideDistY):
     + Atualizamos **sideDistX**, adicionando **deltaDistX** (distância proporcional ao próximo passo no eixo X).
     + Movemos para o próximo quadrado no eixo X: **mapX += stepX**.
     + Definimos **side = 0** para indicar que foi uma colisão com um lado vertical.
   * Se o próximo lado a ser atingido é horizontal (sideDistY < sideDistX):
     + Atualizamos **sideDistY**, adicionando **deltaDistY** (distância proporcional ao próximo passo no eixo Y).
     + Movemos para o próximo quadrado no eixo Y: **mapY += stepY**.
     + Definimos **side = 1** para indicar que foi uma colisão com um lado horizontal.
2. **Por que isso é importante no Raycasting?** A ideia do Raycasting é simular o caminho do raio no mundo, verificando quais partes do grid ele atinge (como paredes ou objetos). O **DDA** permite determinar de maneira eficiente os pontos no grid que o raio atravessa. Ao comparar **sideDistX** e **sideDistY**, sabemos exatamente qual célula deve ser acessada a seguir, evitando cálculos desnecessários.

**Exemplo:**

Suponha que o raio esteja em (2.5,3.2)(2.5, 3.2)(2.5,3.2) no grid e que:

* **sideDistX = 1.0** (o raio atingirá o próximo lado X em 1 unidade).
* **sideDistY = 0.8** (o raio atingirá o próximo lado Y em 0.8 unidades).

Como **sideDistY < sideDistX**, o raio atinge primeiro um lado horizontal:

* Movemos o raio para a próxima linha no grid (mapYmapYmapY).
* Atualizamos a distância acumulada no eixo Y com **sideDistY += deltaDistY**.

**Resumo:**

A comparação **sideDistX < sideDistY** decide em qual direção o raio deve se mover no próximo passo do algoritmo. Isso mantém o algoritmo eficiente e garante que o caminho do raio seja calculado corretamente no grid.

// **After** the **DDA** is done, we have to **calculate the distance of the ray to the wall**, so that we can **calculate how high the wall has to be drawn** after this.

------------------------------ PerpWallDist -------------------------------

//Calculate distance projected on camera direction (Euclidean distance would give fisheye effect!)

if(side == 0) perpWallDist = (sideDistX - deltaDistX);

else perpWallDist = (sideDistY - deltaDistY);

--------------------Linha a desenhar / Altura da parede -------------------

//Calculate height of line to draw on screen

int lineHeight = (int)(h / perpWallDist);

//calculate lowest and highest pixel to fill in current stripe

int drawStart = -lineHeight / 2 + h / 2;

if(drawStart < 0)drawStart = 0;

int drawEnd = lineHeight / 2 + h / 2;

if(drawEnd >= h)drawEnd = h - 1;

---------------------------------- Color ----------------------------------

//choose wall color

ColorRGB color;

switch(worldMap[mapX][mapY])

{

case 1: color = RGB\_Red; break; //red

case 2: color = RGB\_Green; break; //green

case 3: color = RGB\_Blue; break; //blue

case 4: color = RGB\_White; break; //white

default: color = RGB\_Yellow; break; //yellow

}

//give x and y sides different brightness

if (side == 1) {color = color / 2;}

//draw the pixels of the stripe as a vertical line

verLine(x, drawStart, drawEnd, color);

}

------------------------------------FPS e Time ----------------------------

//timing for input and FPS counter

oldTime = time;

time = getTicks();

double frameTime = (time - oldTime) / 1000.0; //frameTime is the time this frame has taken, in seconds

print(1.0 / frameTime); //FPS counter

redraw();

cls();

//speed modifiers

double moveSpeed = frameTime \* 5.0; //the constant value is in squares/second

double rotSpeed = frameTime \* 3.0; //the constant value is in radians/second

------------------------------ Read input Keys ----------------------------

readKeys();

//move forward if no wall in front of you

if (keyDown(SDLK\_UP))

{

if(worldMap[int(posX + dirX \* moveSpeed)][int(posY)] == false) posX += dirX \* moveSpeed;

if(worldMap[int(posX)][int(posY + dirY \* moveSpeed)] == false) posY += dirY \* moveSpeed;

}

//move backwards if no wall behind you

if (keyDown(SDLK\_DOWN))

{

if(worldMap[int(posX - dirX \* moveSpeed)][int(posY)] == false) posX -= dirX \* moveSpeed;

if(worldMap[int(posX)][int(posY - dirY \* moveSpeed)] == false) posY -= dirY \* moveSpeed;

}

//rotate to the right

if (keyDown(SDLK\_RIGHT))

{

//both camera direction and camera plane must be rotated

double oldDirX = dirX;

dirX = dirX \* cos(-rotSpeed) - dirY \* sin(-rotSpeed);

dirY = oldDirX \* sin(-rotSpeed) + dirY \* cos(-rotSpeed);

double oldPlaneX = planeX;

planeX = planeX \* cos(-rotSpeed) - planeY \* sin(-rotSpeed);

planeY = oldPlaneX \* sin(-rotSpeed) + planeY \* cos(-rotSpeed);

}

//rotate to the left

if (keyDown(SDLK\_LEFT))

{

//both camera direction and camera plane must be rotated

double oldDirX = dirX;

dirX = dirX \* cos(rotSpeed) - dirY \* sin(rotSpeed);

dirY = oldDirX \* sin(rotSpeed) + dirY \* cos(rotSpeed);

double oldPlaneX = planeX;

planeX = planeX \* cos(rotSpeed) - planeY \* sin(rotSpeed);

planeY = oldPlaneX \* sin(rotSpeed) + planeY \* cos(rotSpeed);

}

}

}

This concludes the code of the **untextured raycaster**, the result looks like this, and you can walk around in the map:  
  
A computer screen shot of a blue green and red cubes

Description automatically generated  
  
Here's an example of what happens if the camera plane isn't perpendicular to the direction vector, the world appears skewed:  
  
A blue and red rectangles

Description automatically generated

##### Dúvidas e Respostas

1 – Não percebi pq é que perpWallDist fica simplificado para ser perpWallDist = (sideDistX - deltaDistX); ou perpWallDist = (sideDistY - deltaDistY);

2 – Não percebi pq é que é perpWallDist = (sideDistX - deltaDistX)

Se side atingido for X ou perpWallDist = (sideDistY - deltaDistY);

Se for Y.

Ou seja, n percebi esta explicação:

Meaning of the points:

* P: position of the player, (posX, posY) in the code
* H: hitpoint of the ray on the wall. Its y-position is known to be mapY + (1 - stepY) / 2
* yDist matches "(mapY + (1 - stepY) / 2 - posY)", this is the y coordinate of the Euclidean distance vector, in world coordinates. Here, (1 - stepY) / 2) is a correction term that is 0 or 1 based on positive or negative y direction, which is also used in the initialization of sideDistY.
* dir: the main player looking direction, given by dirX,dirY in the code. The length of this vector is always exactly 1. This matches the looking direction in the center of the screen, as opposed to the direction of the current ray. It is perpendicular to the camera plane, and perpWallDist is parallel to this.
* orange dotted line (may be hard to see, use CTRL+scrollwheel or CTRL+plus to zoom in a desktop browser to see it better): the value that was added to dir to get rayDir. Importantly, this is parallel to the camera plane, perpendicular to dir.
* A: point of the camera plane closest to H, the point where perpWallDist intersects with camera plane
* B: point of X-axis through player closest to H, point where yDist crosses the world X-axis through the player
* C: point at player position + rayDirX
* D: point at player position + rayDir.
* E: This is point D with the dir vector subtracted, in other words, E + dir = D.
* points A, B, C, D, E, H and P are used in the explanation below: they form triangles which are considered: BHP, CDP, AHP and DEP.

The actual derivation:

* 1: Triangles PBH and PCD have the same shape but different size, so same ratios of edges
* 2: Given step 1, the triangles show that the ratio yDist / rayDirY is equal to the ratio Euclidean / |rayDir|, so now we can derive perpWallDist = Euclidean / |rayDir| instead.
* 3: Triangles AHP and EDP have the same shape but different size, so same ratios of edges. Length of edge ED, that is |ED|, equals length of dir, |dir|, which is 1. Similarly, |DP| equals |rayDir|.
* 4: Given step 3, the triangles show that the ratio Euclidean / |rayDir| = perpWallDist / |dir| = perpWallDist / 1.
* 5: Combining steps 4 and 2 shows that perpWallDist = yDist / rayDirY, where yDist is mapY + (1 - stepY) / 2) - posY
* 6: In the code, sideDistY - deltaDistY, after the DDA steps, equals (posY + (1 - stepY) / 2 - mapY) \* deltaDistY (given that sideDistY is computed from posY and mapY), so yDist = (sideDistY - deltaDistY) / deltaDistY
* 7: Given that deltaDistY = 1 / |rayDirY|, step 6 gives that yDist = (sideDistY - deltaDistY) \* |rayDirY|
* 8: Combining steps 5 and 7 gives perpWallDist = yDist / rayDirY = (sideDistY - deltaDistY) / |rayDirY| / rayDirY.
* 9: Given how cases for signs of sideDistY and deltaDistY in the code are handled the absolute value doesn't matter, and equals (sideDistY - deltaDistY), which is the formula used.

Resposta:  
**1 e 2 - Por que** perpWallDist **pode ser simplificado para** (sideDistX - deltaDistX) **ou** (sideDistY - deltaDistY)**?**

A fórmula de perpWallDist surge da necessidade de calcular a distância perpendicular da posição do jogador à parede atingida pelo raio. Vamos aos pontos principais:

* **DDA (Digital Differential Analyzer)** avança o raio passo a passo até atingir uma parede. Cada passo acumula uma distância (no eixo X ou Y) chamada sideDistX e sideDistY.
* O problema de "distância perpendicular" vem do facto de sideDistX e sideDistY não serem perpendiculares ao raio. Estes acumulam distâncias ao longo do caminho, mas incluem um "passo inicial" calculado com deltaDistX e deltaDistY.

Fórmulas:

* Para o eixo X:

perpWallDist=sideDistX−\deltaDistX\text{perpWallDist} = \text{sideDistX} - \deltaDistXperpWallDist=sideDistX−\deltaDistX

* Para o eixo Y:

perpWallDist=sideDistY−\deltaDistY\text{perpWallDist} = \text{sideDistY} - \deltaDistYperpWallDist=sideDistY−\deltaDistY

Essas fórmulas ajustam sideDist ao remover o deslocamento inicial (o primeiro passo). Esse ajuste é necessário porque o algoritmo começa deslocado.

Por que diferenciar X e Y?

* **Se a parede atingida estiver no eixo X (horizontal):** A distância perpendicular será calculada com base em sideDistX.
* **Se a parede atingida estiver no eixo Y (vertical):** A distância será baseada em sideDistY.

3 – Não percebi para que temos que calcular a altura da linha a desenhar no ecrã? Que linha é esta? É a que define a altura da parede? Não percebi. Ou seja, não percebi a explicação:

Now that we have the calculated distance (perpWallDist), we can calculate the height of the line that has to be drawn on screen: this is the inverse of perpWallDist, and then multiplied by h, the height in pixels of the screen, to bring it to pixel coordinates. You can of course also multiply it with another value, for example 2\*h, if you want to walls to be higher or lower. The value of h will make the walls look like cubes with equal height, width and depth, while large values will create higher boxes (depending on your monitor).  
  
Then out of this lineHeight (which is thus the height of the vertical line that should be drawn), the start and end position of where we should really draw are calculated. The center of the wall should be at the center of the screen, and if these points lie outside the screen, they're capped to 0 or h-1.

|  |
| --- |
| //Calculate height of line to draw on screen  int lineHeight = (int)(h / perpWallDist);  //calculate lowest and highest pixel to fill in current stripe  int drawStart = -lineHeight / 2 + h / 2;  if(drawStart < 0)drawStart = 0;  int drawEnd = lineHeight / 2 + h / 2;  if(drawEnd >= h)drawEnd = h - 1;  Não percebi nenhum destes cálculos acima, nomeadamente as divisões por 2, somas a h, etc.  Resposta:  Essa linha representa a parede no jogo, projetada no ecrã. A altura depende de quão perto ou longe o jogador está da parede. Vamos às fórmulas: **Altura da linha:** lineHeight=altura do ecra˜ (h)perpWallDist\text{lineHeight} = \frac{\text{altura do ecrã (h)}}{\text{perpWallDist}}lineHeight=perpWallDistaltura do ecra˜ (h)​   * Quanto menor a perpWallDist (mais próximo da parede), maior será a lineHeight. * Multiplicar por h converte a distância para coordenadas de pixels.  **Pontos de início e fim:** A linha precisa ser desenhada no centro do ecrã. Para isso:  drawStart=−lineHeight2+h2\text{drawStart} = -\frac{\text{lineHeight}}{2} + \frac{h}{2}drawStart=−2lineHeight​+2h​ drawEnd=lineHeight2+h2\text{drawEnd} = \frac{\text{lineHeight}}{2} + \frac{h}{2}drawEnd=2lineHeight​+2h​   * A divisão por 2 ajusta a altura para ficar centrada. * Se o ponto for negativo ou maior que o ecrã, é ajustado para não ultrapassar os limites. |

4 – Não percebi de que forma a cor muda com a profundidade da parede. Podes explicar melhor? E pq é dividido por 2 se o side for igual a 1, pq é que dividindo por 2 dá um aspecto mais escuro, certo(//give x and y sides different brightness

if (side == 1) {color = color / 2;}

)?

Resposta: Para criar um efeito de profundidade e sombras:

* Quando side == 1 (eixo Y), a cor da parede é dividida por 2.
* Isso faz a cor parecer mais escura, criando um efeito visual de profundidade (parece mais realista).

5- Não percebi para que servem of FPS(frame per second)? Qual a utilidade efetiva?

Resposta: Os FPS medem a quantidade de imagens desenhadas no ecrã por segundo. Servem para:

* Avaliar o desempenho do jogo (FPS baixo = jogo lento).
* Garantir que a jogabilidade é fluida (FPS ideal é 60+).

6 - E não percebi pq é que se usa a fórmula de frametime e pq é que essa fórmula assegura que o movimento e rotating speed é independente da velocidade do processador.

double frameTime = (time - oldTime) / 1000.0;

Resposta: A fórmula:

frameTime=time−oldTime1000.0\text{frameTime} = \frac{\text{time} - \text{oldTime}}{1000.0}frameTime=1000.0time−oldTime​

* Mede o tempo entre dois frames (em segundos).
* **Objetivo:** Tornar a jogabilidade consistente, independente do poder do processador.

7 – E não percebi pq usamos o moveSpeed e o rotSpeed e pq estes valores têm valores contantes de 5 e de 3 na multiplicação pelo framTime? Também não percebo pq é que para obter esses valores usamos o frameTime?

double moveSpeed = frameTime \* 5.0;  
double rotSpeed = frameTime \* 3.0;

Resposta: Eles ajustam a velocidade de movimento e rotação com base no frameTime. Isso assegura que a velocidade do jogo não dependa da rapidez do processador.

Fórmulas:

moveSpeed=frameTime×5.0\text{moveSpeed} = \text{frameTime} \times 5.0moveSpeed=frameTime×5.0 rotSpeed=frameTime×3.0\text{rotSpeed} = \text{frameTime} \times 3.0rotSpeed=frameTime×3.0

* **Multiplicar por constantes (5 e 3):** Define uma velocidade padrão. Se frameTime for maior (FPS baixo), as velocidades aumentam para compensar.

#### Textured Raycaster

Diferenças para o untexture (**o que está a negrito foi acrescentado ou alterado**):

1. Mapa é diferente (tem mais números/texturas), e deve haver definição de width e height.

**#define screenWidth 640**

**#define screenHeight 480**

**#define texWidth 64**

**#define texHeight 64**

#define mapWidth 24

#define mapHeight 24

int worldMap[mapWidth][mapHeight]=

{

**{4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,7,7,7,7,7,7,7,7},**

**{4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,0,0,0,0,0,0,7},**

**{4,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7},**

**{4,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7},**

**{4,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,7,0,0,0,0,0,0,7},**

**{4,0,4,0,0,0,0,5,5,5,5,5,5,5,5,5,7,7,0,7,7,7,7,7},**

**{4,0,5,0,0,0,0,5,0,5,0,5,0,5,0,5,7,0,0,0,7,7,7,1},**

**{4,0,6,0,0,0,0,5,0,0,0,0,0,0,0,5,7,0,0,0,0,0,0,8},**

**{4,0,7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,7,7,7,1},**

**{4,0,8,0,0,0,0,5,0,0,0,0,0,0,0,5,7,0,0,0,0,0,0,8},**

**{4,0,0,0,0,0,0,5,0,0,0,0,0,0,0,5,7,0,0,0,7,7,7,1},**

**{4,0,0,0,0,0,0,5,5,5,5,0,5,5,5,5,7,7,7,7,7,7,7,1},**

**{6,6,6,6,6,6,6,6,6,6,6,0,6,6,6,6,6,6,6,6,6,6,6,6},**

**{8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4},**

**{6,6,6,6,6,6,0,6,6,6,6,0,6,6,6,6,6,6,6,6,6,6,6,6},**

**{4,4,4,4,4,4,0,4,4,4,6,0,6,2,2,2,2,2,2,2,3,3,3,3},**

**{4,0,0,0,0,0,0,0,0,4,6,0,6,2,0,0,0,0,0,2,0,0,0,2},**

**{4,0,0,0,0,0,0,0,0,0,0,0,6,2,0,0,5,0,0,2,0,0,0,2},**

**{4,0,0,0,0,0,0,0,0,4,6,0,6,2,0,0,0,0,0,2,2,0,2,2},**

**{4,0,6,0,6,0,0,0,0,4,6,0,0,0,0,0,5,0,0,0,0,0,0,2},**

**{4,0,0,5,0,0,0,0,0,4,6,0,6,2,0,0,0,0,0,2,2,0,2,2},**

**{4,0,6,0,6,0,0,0,0,4,6,0,6,2,0,0,5,0,0,2,0,0,0,2},**

**{4,0,0,0,0,0,0,0,0,4,6,0,6,2,0,0,0,0,0,2,0,0,0,2},**

**{4,4,4,4,4,4,4,4,4,4,1,1,1,2,2,2,2,2,2,3,3,3,3,3}**

};

1. The screen buffer and texture arrays are declared here. The texture array is an array of std::vectors, each with a certain width \* height pixels.

**Uint32 buffer[screenHeight][screenWidth]; // y-coordinate first because it works per scanline**

**std::vector texture[8];**

**for(int i = 0; i < 8; i++) texture[i].resize(texWidth \* texHeight);**

**em** int main(int /\*argc\*/, char \*/\*argv\*/[])

{

double posX = 22.0, posY = 11.5; //x and y start position

double dirX = -1.0, dirY = 0.0; //initial direction vector

double planeX = 0.0, planeY = 0.66; //the 2d raycaster version of camera plane

double time = 0; //time of current frame

double oldTime = 0; //time of previous frame

**Uint32 buffer[screenHeight][screenWidth]; // y-coordinate first because it works per scanline**

**std::vector texture[8];**

**for(int i = 0; i < 8; i++) texture[i].resize(texWidth \* texHeight);**

1. Geração das texturas na main function: loop duplo that goes through every pixel of the textures, and then the corresponding pixel of each texture gets a certain value calculated out of x and y.

screen(**screenWidth,screenHeight**, 0, "Raycaster");

**//generate some textures**

**for(int x = 0; x < texWidth; x++)**

**for(int y = 0; y < texHeight; y++)**

**{**

**int xorcolor = (x \* 256 / texWidth) ^ (y \* 256 / texHeight);**

**//int xcolor = x \* 256 / texWidth;**

**int ycolor = y \* 256 / texHeight;**

**int xycolor = y \* 128 / texHeight + x \* 128 / texWidth;**

**texture[0][texWidth \* y + x] = 65536 \* 254 \* (x != y && x != texWidth - y); //flat red texture with black cross**

**texture[1][texWidth \* y + x] = xycolor + 256 \* xycolor + 65536 \* xycolor; //sloped greyscale**

**texture[2][texWidth \* y + x] = 256 \* xycolor + 65536 \* xycolor; //sloped yellow gradient**

**texture[3][texWidth \* y + x] = xorcolor + 256 \* xorcolor + 65536 \* xorcolor; //xor greyscale**

**texture[4][texWidth \* y + x] = 256 \* xorcolor; //xor green**

**texture[5][texWidth \* y + x] = 65536 \* 192 \* (x % 16 && y % 16); //red bricks**

**texture[6][texWidth \* y + x] = 65536 \* ycolor; //red gradient**

**texture[7][texWidth \* y + x] = 128 + 256 \* 128 + 65536 \* 128; //flat grey texture**

**}**

1. texX is the x-coordinate of the texture, and this is calculated out of wallX.

**//texturing calculations**

**int texNum = worldMap[mapX][mapY] - 1; //1 subtracted from it so that texture 0 can be used!**

**//calculate value of wallX**

**double wallX; //where exactly the wall was hit**

**if (side == 0) wallX = posY + perpWallDist \* rayDirY;**

**else wallX = posX + perpWallDist \* rayDirX;**

**wallX -= floor((wallX));**

**//x coordinate on the texture**

**int texX = int(wallX \* double(texWidth));**

**if(side == 0 && rayDirX > 0) texX = texWidth - texX - 1;**

**if(side == 1 && rayDirY < 0) texX = texWidth - texX - 1;**

1. current buffer pixel is set to this color, and we move on to the next y.

**// How much to increase the texture coordinate per screen pixel**

**double step = 1.0 \* texHeight / lineHeight;**

**// Starting texture coordinate**

**double texPos = (drawStart - h / 2 + lineHeight / 2) \* step;**

**for(int y = drawStart; y<drawEnd; y++)**

**{**

**// Cast the texture coordinate to integer, and mask with (texHeight - 1) in case of overflow**

**int texY = (int)texPos & (texHeight - 1);**

**texPos += step;**

**Uint32 color = texture[texNum][texHeight \* texY + texX];**

**//make color darker for y-sides: R, G and B byte each divided through two with a "shift" and an "and"**

**if(side == 1) color = (color >> 1) & 8355711;**

**buffer[y][x] = color;**

**}**

**}**

1. buffer still has to be drawn, and after that it has to be cleared.

**drawBuffer(buffer[0]);**

**for(int y = 0; y < h; y++) for(int x = 0; x < w; x++) buffer[y][x] = 0; //clear the buffer instead of cls()**

//timing for input and FPS counter

oldTime = time;

time = getTicks();

double frameTime = (time - oldTime) / 1000.0; //frametime is the time this frame has taken, in seconds

print(1.0 / frameTime); //FPS counter

redraw();

//speed modifiers

double moveSpeed = frameTime \* 5.0; //the constant value is in squares/second

double rotSpeed = frameTime \* 3.0; //the constant value is in radians/second

Here's a few screenshots of the result:  
  
A video game screen with a green and red cubes

Description automatically generated A screenshot of a video game

Description automatically generated A screenshot of a video game

Description automatically generated

##### Wolfenstein 3D Textures

A blue and grey brick wall

Description automatically generated  
  
Just replace the part of the code that generates the texture patterns with the following (and make sure those textures are in the correct path). You can download the textures [here](https://lodev.org/cgtutor/files/wolftex.zip).

|  |
| --- |
| //generate some textures  unsigned long tw, th;  loadImage(texture[0], tw, th, "pics/eagle.png");  loadImage(texture[1], tw, th, "pics/redbrick.png");  loadImage(texture[2], tw, th, "pics/purplestone.png");  loadImage(texture[3], tw, th, "pics/greystone.png");  loadImage(texture[4], tw, th, "pics/bluestone.png");  loadImage(texture[5], tw, th, "pics/mossy.png");  loadImage(texture[6], tw, th, "pics/wood.png");  loadImage(texture[7], tw, th, "pics/colorstone.png"); |

A brick wall with a colorful design

Description automatically generated with medium confidence A video game screen with blue and brown blocks

Description automatically generated  
  
In the original Wolfenstein 3D, the colors of one side was also made darker than the color of the other side of a wall to create the shadow effect, but they used a seperate texture every time, a dark and a light one. Here however, only one texture is used for each wall and the line of code that divided R, G and B through 2 is what makes the y-sides darker.

### Proposta de Estrutura

#### Ficheiro.h

#### Estrutura Organizacional

Esta é a proposta de estrutura organizacional de pastas e funções que tenho:

cub3d/

├── include/

│ ├── cub3d.h

├── src/

│ ├── main.c

│ ├── parser/

│ │ ├── config.c

│ │ ├── map\_loader.c

│ │ └── validation.c

│ ├── init/

│ │ ├── setup\_game.c

│ │ ├── setup\_mlx.c

│ │ └── setup\_textures.c

│ ├── movement/

│ │ ├── player\_input.c

│ │ ├── movement\_utils.c

│ │ └── rotation.c

│ ├── rendering/

│ │ ├── draw.c

│ │ ├── raycasting.c

│ │ └── textures.c

│ ├── cleanup/

│ │ ├── free\_resources.c

│ │ └── exit\_game.c

1 - main.c

Responsável por inicializar o jogo e manter o loop principal.

2 - Parser

Funções relacionadas à análise do arquivo e do mapa. Foi subdividido em arquivos de configuração, carregamento e validação.

2.1. - config.c: Lê e carrega as configurações principais, como texturas e cores. Responsabilidade de carregar e processar o arquivo de configuração, que pode incluir informações como cores, texturas e configurações do mapa.

exemplo de funções:

// Função que carrega o arquivo de configuração, incluindo cores e configurações de textura.  
int load\_config(t\_game \*game, const char \*config\_file);

// Função que analisa uma linha do arquivo de configuração e atribui os valores apropriados.

void parse\_config\_line(t\_game \*game, char \*line);

2.2. - map\_loader.c: Lida com o carregamento do mapa a partir do arquivo .cub e armazena numa estrutura de dados apropriada.

exemplo de funções:

// Função que carrega o mapa do arquivo para a estrutura t\_map.

int load\_map(t\_game \*game, const char \*map\_file);

// Função para liberar a memória usada pelo mapa (array 2D).

void free\_map(t\_map \*map);

2.3. - validation.c: Valida o mapa, verificando paredes, posições do jogador, etc, ou seja, valida a integridade do mapa e do arquivo de configuração, garantindo que estão corretos antes de iniciar o jogo.

exemplo de funções:   
// Função que valida a estrutura do mapa: se as paredes estão fechadas, etc.  
int validate\_map(t\_map \*map);

// Função que valida as configurações carregadas do arquivo de configuração (como texturas).int validate\_config(t\_game \*game);

3 - Init

3.1. - setup\_game: Iniciar o jogo. Configura e inicializa o estado do jogo, como o jogador, o mapa, e outros componentes necessários.

exemplo de funções:

// Função que configura a estrutura do jogo: inicializa o jogador, mapa e outras variáveis.  
void setup\_game(t\_game \*game);

// Função para inicializar a posição e propriedades do jogador.

void setup\_player(t\_player \*player);

// Função para inicializar as propriedades do mapa.

void setup\_map(t\_map \*map);

3.2. - setup\_mlx.c: inicializar a biblioteca. Configura e inicializa a biblioteca MLX (MinilibX) para gestão de gráficos e interação com a janela do jogo.

exemplo de funções:

// Função para inicializar o sistema MLX, incluindo a criação da janela.

void setup\_mlx(t\_game \*game);

// Função para criar uma imagem a ser renderizada na janela (geralmente para a tela do jogo).

void create\_image(t\_game \*game);

// Função para inicializar as texturas do jogo, associando-as com a estrutura t\_texture.

void init\_textures(t\_game \*game);

3.3. - setup\_textures: carregar o mapa, configurar o jogador, e iniciar o loop principal de renderização e entrada, ou seja, Carrega e configura as texturas usadas no jogo (como paredes e sprites e o player que é uma sprite).

exemplo de funções:   
// Função que “carreg”a as texturas do jogo a partir de arquivos de imagem.

void load\_textures(t\_game \*game);

// Função que liberta a memória das texturas.  
void free\_textures(t\_game \*game);

4 - Movement

Movimentos do jogador, incluindo lógica de rotação e movimentação.

4.1. - player\_input.c: Gere a entrada do jogador, como os controlos de movimento e rotação, ou seja, basicamente processa as teclas pressionadas e chama as funções de movimento.

exemplo de funções:

// Função que gerencia as teclas pressionadas pelo jogador para movimento e rotação.

void handle\_keys(t\_game \*game);

// Função para processar o movimento do jogador, modificando a posição e a direção.

void process\_movement(t\_game \*game);

4.2. - movement\_utils.c: Implementa funções auxiliares para o movimento e cálculos de física do jogador tais como move\_forward, move\_backward, strafe\_left e strafe\_right.

exemplo de funções:  
// Função para mover o jogador para frente.

void move\_forward(t\_game \*game);

// Função para mover o jogador para trás.

void move\_backward(t\_game \*game);

// Função para mover o jogador para a esquerda.

void strafe\_left(t\_game \*game);

// Função para mover o jogador para a direita.

void strafe\_right(t\_game \*game);

4.3. - rotation.c: Manipula a rotação do jogador.

exemplo de funções:  
// Função para girar a visão do jogador para a esquerda.

void rotate\_left(t\_game \*game);  
 // Função para girar a visão do jogador para a direita.

void rotate\_right(t\_game \*game);

-------TUA parte Abaixo---------

5 - Rendering

Funções de renderização e raycasting, ou seja, funções responsáveis pela renderização dos gráficos, como desenhar o mapa e as paredes.

5.1. - draw.c: Renderiza a imagem na janela.

exemplo de funções:   
// Função principal de renderização da cena do jogo (mapa, paredes, etc).

void draw\_scene(t\_game \*game);

// Função para desenhar um minimapa no canto do ecrã/janela (opcional).

void draw\_minimap(t\_game \*game);

5.2. raycasting.c: Implementa os cálculos de raycasting, ou seja, são as funções responsáveis pela técnica de raycasting para calcular as distâncias e ângulos das paredes no mapa e projetá-las na tela.

exemplo de funções:

// Função que executa o raycasting, calculando as interações dos raios com o mapa.

void raycast(t\_game \*game);

// Função para calcular onde o raio atinge uma parede e definir a cor para renderizar.

void calculate\_wall\_hit(t\_game \*game, int x);

5.3. - textures.c: Lida com upload e aplicação das texturas nas paredes e outros objetos do jogo.

exemplo de funções:

// Função para carregar uma textura a partir de um arquivo.  
void load\_texture(t\_game \*game, t\_texture \*texture, const char \*file);

// Função para aplicar a textura na tela numa posição específica.

void apply\_texture(t\_game \*game, int x, int y, t\_texture \*texture);

6 - Cleanup

Funções para libertar memória e finalizar o jogo.

6.1. - free\_resources.c: Liberta os recursos alocados (mapa, texturas, etc.).

exemplo de funções:

// Função para libertar todos os recursos do jogo (mapa, texturas, etc).

void free\_game(t\_game \*game);

// Função para libertar a memória do mapa.  
void free\_map\_resources(t\_map \*map);

6.2. - exit\_game.c: Finaliza o jogo e fecha janelas.

exemplo de funções:

// Função para terminar o jogo com uma mensagem de erro ou sucesso e limpar os recursos.  
void exit\_game(t\_game \*game, const char \*message);

// Função para fechar a janela do jogo e libertar os recursos ao final.

void close\_game(t\_game \*game);