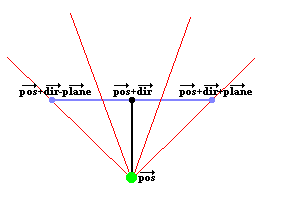
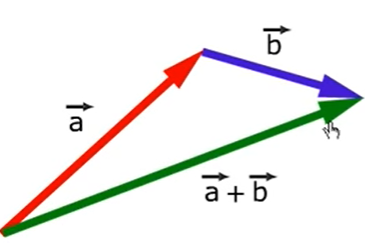
**calcolo del RayDir**



Nell’immagine sopra viene spiegato come ottenere la direzione di un raggio. Prendendo ad esempio il secondo da destra viene spiegato come la sua direzione dia

Vediamo perché:  
La somma di due vettori congiunti punta-cosa non sarà altro che il vettore che congiunge le due estremità  
dei vettori di partenza:



Considerando che il la porzione del piano presa in considerazione per la somma è 1/3 plane avrò che la direzione del vettore raggio è data dalla formula scritta sopra.

|  |
| --- |
| **NOTA:** La direzione di un vettore è indicata dalla retta o dalla linea lungo la quale il vettore si estende nel suo spazio geometrico. In altre parole, è la linea che passa attraverso l'origine (o un punto qualsiasi del vettore, a seconda del contesto) e il punto terminale del vettore stesso. Se vogliamo poi normalizzare questo vettore (renderlo unitario) dovremo dividere le sue componenti per la magnitudo del vettore stesso |

Le componenti del vettore direzione **rayDir** cosi ottenuto saranno usate per l’algoritmo DDA

**FOV**• Le due linee esterne rappresentano il bordo sinistro e destro dello schermo, e l'angolo tra queste due linee è chiamato Campo Visivo o FOV. **Il FOV è determinato dal rapporto tra la lunghezza del vettore di direzione e la lunghezza del piano**.

Ecco alcuni esempi di diversi FOV:  
- Se il vettore di direzione e il vettore del piano della telecamera hanno la stessa lunghezza, il FOV è di 90°  
- Se il vettore di direzione è molto più lungo del piano della telecamera, il FOV sarà molto più piccolo di 90° e avrai una visione molto stretta. Vedrai tutto più dettagliatamente ma ci sarà meno profondità, quindi sarà simile a fare uno zoom in avanti:

**Rotazione**  
**Quando il giocatore ruota**, la telecamera deve ruotare, **quindi sia il vettore di direzione che il vettore del piano devono essere ruotati**. A quel punto, i raggi ruoteranno automaticamente.

Per ruotare un vettore di un angolo alfa bisognerà moltiplicarlo per la matrice di rotazione:



**RAYCASTING SENZA TEXTURE  
Variabili**  
• Iniziamo dichiarando alcune variabili:

**posX e posY**: vettore posizione del giocatore   
**dirX e dirY:** direzione del giocatore  
**planeX e planeY**: piano della telecamera del giocatore.

Il piano della telecamera dovrà essere perpendicolare alla direzione ma la sua lunghezza potrà variare

Il **rapporto tra la lunghezza della direzione e del piano della telecamera determina il FOV** (Field Of Vision); in questo caso, il vettore direzione è leggermente più lungo del piano della telecamera, quindi il FOV sarà inferiore a 90° (più precisamente, il **FOV è 2 \* atan(0.66/1.0) = 66°,** il che è perfetto per un gioco sparatutto in prima persona). In seguito, quando si ruota con i tasti di input, i valori di dir e plane verranno modificati, ma rimarranno sempre perpendicolari e mantenendo la stessa lunghezza.

Le variabili **time e oldTime** verranno utilizzate per **memorizzare il tempo del frame corrente e del frame precedente**; la differenza di tempo tra questi due può essere **utilizzata per determinare quanto si dovrebbe muovere quando viene premuto un certo tasto** (per muoversi a una velocità costante indipendentemente dalla durata del calcolo dei frame), e per il contatore FPS.

  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

• Nel resto della funzione main andiamo a creare la finestra con una risoluzione a piacimento ( Se scegli una risoluzione grande, come 1280x1024, l'effetto sarà piuttosto lento, non perché l'algoritmo di raycasting sia lento, ma semplicemente perché caricare un intero schermo dalla CPU alla scheda video è molto lento.)

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

Dopo aver configurato lo schermo, inizia il ciclo del gioco. Questo è il ciclo che disegna un intero fotogramma e legge l'input ogni volta.

while(!done())

{

• Qui inizia il vero e proprio raycasting. **Il ciclo del raycasting** è un ciclo for che **attraversa ogni x**, quindi **non** viene **effettuato un calcolo per ogni pixel** dello schermo!

Prima di implementarlo dobbiamo però dichiarare e calcolare alcune variabili:

- Il **raggio** **parte dalla posizione del giocatore**(posX, posY)

- **cameraX** è la coordinata x sul piano della telecamera che la corrente coordinata x dello schermo rappresenta. Questo è fatto in modo tale che il lato destro dello schermo ottenga la coordinata 1, il centro dello schermo ottenga la coordinata 0 e il lato sinistro dello schermo ottenga la coordinata -1. In questo modo, è possibile calcolare la direzione del raggio come spiegato in precedenza: come somma del vettore direzione e di una parte del vettore piano. Questo calcolo deve essere effettuato sia per la coordinata x sia per la coordinata y del vettore (poiché sommare due vettori significa sommare le loro coordinate x e sommare le loro coordinate y).

Spiegazione:  
**cameraX**: Viene utilizzata per mappare le coordinate x dello schermo su un **piano immaginario della telecamera** (la proiezione di **plane** nel punto del player).  
La mappatura è calcolata in modo tale da farla andare da -1 a 1, dove -1 è l'estrema sinistra dello schermo, 0 è il centro, e 1 è l'estrema destra.

**Direzione del raggio**: Una volta determinata la posizione sul piano della telecamera (cameraX), la direzione del raggio può essere calcolata combinando il vettore di direzione (che rappresenta la direzione in cui la telecamera sta guardando) e una parte del vettore piano (che rappresenta il campo visivo della telecamera).

**Somma dei vettori**: Per ottenere la direzione finale del raggio, si sommano le componenti x e y dei due vettori. Questa somma determina la direzione esatta del raggio che sarà tracciato nello spazio 3D.

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

{

//calculate ray position and direction

double cameraX = 2 \* x / double(w) - 1; //x-coordinate in camera space

double rayDirX = dirX + planeX \* cameraX;

double rayDirY = dirY + planeY \* cameraX;

Nota: “w” dovrebbe essere la *width* dello schermo in pixel

• Ci servono ora altre variabili che useremo nell’algoritmo di DDA

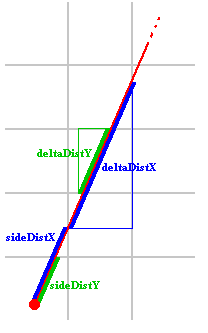
- **mapX** e **mapY** rappresentano la **cella corrente della mappa in cui si trova il raggio**.

La posizione del raggio stessa è un *float* e contiene sia informazioni su quale cella della mappa ci troviamo, sia dove in quel quadrato ci troviamo, ma mapX e mapY rappresentano solo le coordinate di quel quadrato.

- **sideDistX** e **sideDistY** sono **inizialmente la distanza che il raggio deve percorrere dalla sua posizione  
 iniziale fino al primo lato x e al primo lato y**. Più avanti nel codice, verranno incrementati mentre vengono   
 fatti i passi.

- **deltaDistX** e **deltaDistY** sono la **distanza che il raggio percorrere incrementando di un unità (cella) la x e  
 la y**. La seguente immagine mostra sideDistX, sideDistY, deltaDistX e deltaDistY iniziali:

Nota: Quando si parla di “lato x o lato y” si intende il lato x o y di una singola cella



Derivando **deltaDistX** geometricamente ottieni, tramite il teorema di Pitagora, le formule sottostanti.  
Per il triangolo blu **(deltaDistX**), un lato ha lunghezza 1 (poiché è esattamente una cella) e l'altro ha lunghezza **raydirY / raydirX** poiché descrive quante unità in direzione y il raggio percorre per ogni unità in direzione x. Per il triangolo verde (deltaDistY), la formula è simile.

    deltaDistX = sqrt(1 + (rayDirY \* rayDirY) / (rayDirX \* rayDirX));

    deltaDistY = sqrt(1 + (rayDirX \* rayDirX) / (rayDirY \* rayDirY));

Possiamo semplificare queste formule:

    deltaDistX = abs(|rayDir| / rayDirX);

    deltaDistY = abs(|rayDir| / rayDirY);

- Dove **|rayDir|** è la **lunghezza del vettore rayDirX**, **rayDirY**   
Tuttavia, **possiamo usare 1 invece di |rayDir|, perché solo il rapporto tra deltaDistX e deltaDistY è importante** per il codice DDA che segue più avanti, quindi otteniamo:

deltaDistX = abs(1 / rayDirX);

deltaDistY = abs(1 / rayDirY);

Nota: Per questo motivo, i valori di deltaDist e sideDist utilizzati nel codice non corrispondono alle lunghezze mostrate nell'immagine sopra, ma le loro dimensioni relative corrispondono comunque.

- **perpWallDist**: Questa variabile verrà **utilizzata successivamente per calcolare la lunghezza del raggio**.

- **stepX** e **stepY**: L'algoritmo DDA salterà sempre di una cella ad ogni ciclo, sia sul lato delle direzione x,   
 sia in quello delle direzione y.   
 Se la **direzione** ( in x o y) sarà **positiva o negativa** dipenderà dalla direzione del raggio,   
 e questo fatto sarà memorizzato in stepX e stepY. Che **avranno sempre valore -1 o +1**.

- **hit**: utilizzato per determinare se il **ciclo in corso può essere terminato** (muro colpito)  
  
- **side**: indica **se è stato colpito un lato x o un lato y di un muro**. Se è stato colpito un lato x, side viene   
 impostato a 0, se è stato colpito un lato y, side viene impostato a 1. Con lato x e lato y, intendo le  
 linee della griglia che sono i confini tra due quadrati.

//which box of the map we're in

int mapX = int(posX);

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);

double deltaDistY = (rayDirY == 0) ? 1e30 : std::abs(1 / rayDirY);

double perpWallDist;

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

int stepX;

int stepY;

int hit = 0; //was there a wall hit?

int side; //was a NS or a EW wall hit?

NOTA: Se rayDirX o rayDirY sono 0, la divisione per zero viene evitata impostando il valore a un numero molto alto, 1e30.

• Queste sono le variabili necessarie. Prima di far partire l’algoritmo però dobbiamo calcolare stepX, stepY e le iniziali sideDistX e sideDistY.

- **stepx** , **stepy**: Se la **direzione del raggio ha una componente x negativa, stepX è -1**; se la direzione del   
 raggio ha una **componente x positiva, stepX è +1**. **Se** la componente **x è 0**, non importa  
 quale valore abbia **stepX** poiché **non verrà utilizzato**. Lo stesso vale per la componente y.

- **sideDistX** , **sideDistY**: **Se la direzione del raggio ha una componente x negativa**, **sideDistX** **è la distanza  
 dalla posizione iniziale del raggio al primo lato a sinistra**; **se** la direzione del raggio  
 ha una **componente x positiva, viene utilizzato il primo lato a destra**.   
 Lo stesso vale per la componente y, ma ora con il primo lato sopra o sotto la  
 posizione.  
 Per questi valori, viene utilizzato il valore intero mapX e si sottrae la posizione reale  
 da esso, aggiungendo 1.0 in alcuni casi a seconda se viene utilizzato il lato sinistro o  
 destro, oppure quello superiore o inferiore. Poi si ottiene la distanza perpendicolare   
 a questo lato, quindi si moltiplica per deltaDistX o deltaDistY per ottenere la distanza   
 euclidea reale.

//calculate step and initial sideDist

if (rayDirX < 0)

{

stepX = -1;

sideDistX = (posX - mapX) \* deltaDistX;

}

else

{

stepX = 1;

sideDistX = (mapX + 1.0 - posX) \* deltaDistX;

}

if (rayDirY < 0)

{

stepY = -1;

sideDistY = (posY - mapY) \* deltaDistY;

}

else

{

stepY = 1;

sideDistY = (mapY + 1.0 - posY) \* deltaDistY;

}

**Implementazione DDA**

• L’algoritmo di DDA è costituito da un **ciclo che incrementa il raggio di una cella ad ogni iterazione**, **fino a  
 quando non colpisce un muro.**

* Ogni volta, **salta un quadrato nella direzione x (con stepX) o un quadrato nella direzione y (con stepY),** saltando sempre un quadrato alla volta.

Se la direzione del raggio avesse solo componente x, il ciclo dovrebbe saltare una cella nella direzione x ogni volta, in quanto il raggio non cambierebbe mai la sua direzione y.  
Se il raggio è leggermente inclinato verso la direzione y, allora ogni tot salti nella direzione x, il raggio dovrà saltare un quadrato nella direzione y.  
Se il raggio è esattamente nella direzione y, non dovrà mai saltare nella direzione x, ecc.

* **sideDistX** e **sideDistY** **vengono incrementati con deltaDistX a ogni salto nella loro direzione**, e **mapX** e **mapY** **vengono incrementati rispettivamente con stepX e stepY**.
* **Quando il raggio colpisce un muro, il ciclo termina** e **sapremo se è stato colpito un lato x o un lato y di un muro nella variabile** "**side**" **e quale muro è stato colpito con mapX e mapY.** Tuttavia, non sapremo esattamente dove il muro è stato colpito, ma ciò non è necessario in questo caso perché non utilizzeremo muri con texture per ora.

//perform DDA

while (hit == 0)

{

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

if (sideDistX < sideDistY)

{

sideDistX += deltaDistX;

mapX += stepX;

side = 0;

}

else

{

sideDistY += deltaDistY;

mapY += stepY;

side = 1;

}

//Check if ray has hit a wall

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

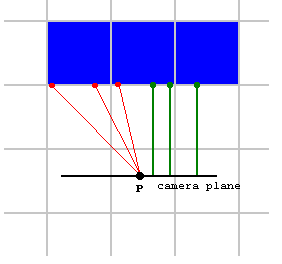
}

• Dopo aver completato l'algoritmo di Digital Differential Analyzer (DDA), dobbiamo calcolare la distanza del   
 raggio fino al muro, in modo da poter determinare l'altezza del muro che deve essere disegnata   
 successivamente.

Non **useremo** la distanza euclidea dal punto che rappresenta il giocatore, ma invece la distanza dal piano della telecamera (o **la distanza del punto proiettato nella direzione della telecamera verso il giocatore**), per evitare l'effetto fisheye.

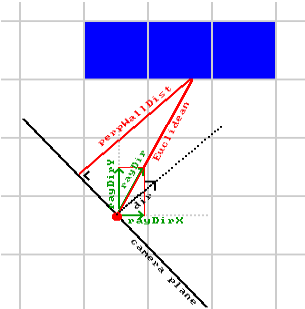
L'immagine seguente mostra perché prendiamo la distanza dal piano della telecamera invece che dal giocatore. Con P il giocatore e la linea nera il piano della telecamera: a sinistra del giocatore, i raggi rossi rappresentano i punti di impatto sul muro al giocatore, questa è la distanza euclidea.   
A destra del giocatore, troviamo invece dei raggi verdi che vanno dai punti di impatto sul muro direttamente al piano della telecamera.

Nell'immagine, il giocatore guarda direttamente il muro, e. I raggi verdi a destra hanno tutti la stessa lunghezza, quindi daranno il risultato corretto. Lo stesso vale anche quando il giocatore ruota (in questo caso il piano della telecamera non è più orizzontale e le linee verdi avranno lunghezze diverse, ma con una variazione costante fra loro) e i muri diventano linee diagonali ma dritte sullo schermo. Questa spiegazione è un po' approssimativa, ma dà l'idea.



NOTA: Questo metodo è persino più facile calcolare questa distanza perpendicolare rispetto alla distanza   
 reale, non è nemmeno necessario conoscere l'esatta posizione in cui il muro è stato colpito.

• **Questa distanza perpendicolare è chiamata "perpWallDist"** nel codice.   
 Un modo per calcolarla è utilizzare la formula per la distanza più breve da un punto a una linea, dove il  
 punto è dove il muro è stato colpito e la linea è il piano della telecamera:



Tuttavia, **può essere calcolata** **in modo più semplice**: a causa del modo in cui deltaDist e sideDist sono stati scalati per un fattore di |rayDir| sopra, **la lunghezza di sideDist è già quasi uguale a perpWallDist**. **Basta** **solo sottrargli una volta deltaDist**, tornando indietro di un passo, perché nei passaggi DDA sopra siamo andati un passo oltre per finire all'interno del muro.

A seconda che il raggio colpisca un lato X o un lato Y, la formula viene calcolata utilizzando sideDistX o sideDistY.

//Calculate distance projected on camera direction

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

else perpWallDist = (sideDistY - deltaDistY);

• Ora che abbiamo calcolato la distanza (perpWallDist), **possiamo calcolare l'altezza della linea che deve essere disegnata sullo schermo**: **lineHeight**

questa è **l'inverso di perpWallDist, moltiplicato poi per h, l'altezza in pixel dello schermo**, per portarlo a coordinate in pixel.   
Puoi ovviamente moltiplicarlo anche con un altro valore, ad esempio 2\*h, se vuoi che i muri siano più alti o più bassi.

Quindi, a **partire da questa lineHeight, vengono calcolate la posizione iniziale e finale di dove dovremmo realmente disegnare** il muro. Il centro del muro dovrebbe essere al centro dello schermo, e se questi punti si trovano fuori dallo schermo, vengono limitati a 0 o 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;

**• Infine, a seconda del numero del muro che è stato colpito, viene scelto un colore**. Se è stato colpito un lato y, lo disegneremo di un colore più scuro. E poi la linea verticale viene disegnata con il comando verLine. **Questo conclude il ciclo di raycasting**, dopo averlo fatto per ogni x almeno una volta.

//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);

}

• Dopo che il ciclo di raycasting è completato**, viene calcolato il tempo del frame corrente e di quello precedente, vengono cioè calcolati e stampati gli FPS**, **e lo schermo viene ridisegnato** in modo che tutto (tutti i muri e il valore del contatore FPS) diventi visibile.

Dopodiché il backbuffer viene cancellato con cls(), così che quando andremo a ridisegnare di nuovo i muri nel frame successivo, il pavimento e il soffitto saranno di nuovo neri invece di contenere ancora pixel dal frame precedente.

**I modificatori di velocità utilizzano frameTime e un valore costante per determinare la velocità di movimento e rotazione** dei tasti di input. Grazie all'uso di frameTime, possiamo garantire che la velocità di movimento e rotazione sia indipendente dalla velocità del processore.

//timing for input and FPS counter

oldTime = time;

time = getTicks();

//frameTime is the time this frame has taken, in seconds

double frameTime = (time - oldTime) / 1000.0;

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

**INPUT**

• L'ultima parte riguarda l'input.

🡪 Se viene premuta la **freccia verso l’alto**, il giocatore si muoverà in avanti:

Dovremo quindi **aggiungere dirX a posX e dirY a posY.**   
Questo presuppone che dirX e dirY siano vettori normalizzati (la loro lunghezza è 1), ma questo risulta vero da costruzione, quindi ci siamo.   
C'è anche un semplice rilevamento delle collisioni integrato, ovvero se la nuova posizione sarà all'interno di un muro, non ti muoverai. Tuttavia, questo rilevamento delle collisioni può essere migliorato, ad esempio controllando se un cerchio attorno al giocatore non entrerà nel muro.

Lo stesso avverrà per la **freccia verso il basso** con la differenza che la **direzione verrà sottratta**.

🡪 Per **ruotare**, se viene premuta la freccia sinistra o destra, **sia il vettore di direzione che il vettore del piano vengono ruotati utilizzando le formule di moltiplicazione con la matrice di rotazione (e l'angolo rotSpeed)**.

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);

}

}

}

**TEXTURED RAYCASTER**

• Il nucleo della versione con texture del raycaster è quasi identico; dovremo però eseguire alcuni calcoli aggiuntivi per gestire le texture. Avremo bisogno di **un ciclo nella direzione y per attraversare ogni pixel e determinare quale texel (pixel della texture) deve essere utilizzato**.

Le strisce verticali non possono più essere disegnate utilizzando il comando per le linee verticali; è necessario disegnare ogni pixel separatamente. **Il metodo migliore è usare un array 2D come buffer dello schermo e copiarlo tutto in una volta sullo schermo**.

Naturalmente, ora **è necessario un** **array aggiuntivo per le texture**. Poiché la funzione "drawbuffer" lavora con valori interi singoli per i colori (anziché con 3 byte separati per R, G e B), anche le texture sono memorizzate in questo formato. Normalmente, le texture verrebbero caricate da un file, ma per questo esempio semplice vengono generate delle texture basilari.

Nel codice che scriveremo metteremo le parti aggiuntive in grassetto.

🡪 screenWidth e screenHeight sono definiti all'inizio, poiché necessarie sia per la funzione dello schermo sia per creare il buffer dello schermo.   
Andremo ora ad **introdurre** anche la **larghezza e l'altezza delle texture**. Questi valori rappresentano chiaramente la larghezza e l'altezza delle texture in texels (pixel della texture).

**#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}**

};

🡪 Andiamo ora a dichiarare Il **buffer dello schermo e gli array delle texture**.   
 - L'array delle texture è un array di std::vector, ognuno dei quali contiene  
 un certo numero di pixel , pari a larghezza \* altezza.

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

**// y-coordinate first because it works per scanline**

**Uint32 buffer[screenHeight][screenWidth];**

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

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

Traducendo in C:

// Dichiara il buffer

Uint32 (\*buffer)[screenWidth] = malloc(screenHeight \* sizeof(\*buffer));

// Dichiara l'array delle texture

Uint32 \*texture[8];

for(int i = 0; i < 8; i++) {

    texture[i] = malloc(texWidth \* texHeight \* sizeof(Uint32));

}

// ... dopo aver finito con le texture, ricordati di liberare la memoria

for(int i = 0; i < 8; i++) {

    free(texture[i]);

}

free(buffer);

**Buffer Allocation:**

* **C++:** Uint32 buffer[screenHeight][screenWidth];
* **C:** Usa malloc per allocare dinamicamente un array bidimensionale. La dimensione è calcolata come screenHeight \* sizeof(\*buffer), dove \*buffer è un array di Uint32 di dimensione screenWidth.

**Texture Allocation:**

* **C++:** std::vector texture[8]; e texture[i].resize(texWidth \* texHeight);
* **C:** Usa malloc per allocare un array di Uint32 per ogni texture, con dimensione texWidth \* texHeight. L'array texture è un array di puntatori a Uint32, e ogni elemento viene inizializzato con un blocco di memoria di dimensioni appropriate.

**Memory Deallocation:**

* In C, è essenziale liberare la memoria allocata dinamicamente usando free per evitare perdite di memoria.

• La funzione principale ora inizia generando le texture. Abbiamo un doppio ciclo che attraversa ogni pixel delle texture, e poi il pixel corrispondente di ciascuna texture riceve un certo valore calcolato a partire da x e y. Alcune texture ottengono un pattern XOR, altre un semplice gradiente, altre ancora una sorta di pattern a mattoni; fondamentalmente sono tutti pattern piuttosto semplici, non avranno un aspetto molto bello. Per texture migliori, vedi il capitolo successivo.

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;  
   
 //flat red texture with black cross**

**texture[0][texWidth \* y + x] = 65536 \* 254 \* (x != y && x != texWidth - y);   
 //sloped greyscale**

**texture[1][texWidth \* y + x] = xycolor + 256 \* xycolor + 65536 \* xycolor;   
 //sloped yellow gradient  
 texture[2][texWidth \* y + x] = 256 \* xycolor + 65536 \* xycolor;   
 //xor greyscale**

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

**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**

**}**

• Ora ritroviamo il loop principale del gioco con il calcolo del DDA. Qui nulla è stato cambiato.

//start the main loop

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

double rayDirX = dirX + planeX\*cameraX;

double rayDirY = dirY + planeY\*cameraX;

//which box of the map we're in

int mapX = int(posX);

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 = sqrt(1 + (rayDirY \* rayDirY) / (rayDirX \* rayDirX));

double deltaDistY = sqrt(1 + (rayDirX \* rayDirX) / (rayDirY \* rayDirY));

double perpWallDist;

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

int stepX;

int stepY;

int hit = 0; //was there a wall hit?

int side; //was a NS or a EW wall hit?

//calculate step and initial sideDist

if (rayDirX < 0)

{

stepX = -1;

sideDistX = (posX - mapX) \* deltaDistX;

}

else

{

stepX = 1;

sideDistX = (mapX + 1.0 - posX) \* deltaDistX;

}

if (rayDirY < 0)

{

stepY = -1;

sideDistY = (posY - mapY) \* deltaDistY;

}

else

{

stepY = 1;

sideDistY = (mapY + 1.0 - posY) \* deltaDistY;

}

//perform DDA

while (hit == 0)

{

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

if (sideDistX < sideDistY)

{

sideDistX += deltaDistX;

mapX += stepX;

side = 0;

}

else

{

sideDistY += deltaDistY;

mapY += stepY;

side = 1;

}

//Check if ray has hit a wall

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

}

//Calculate distance of perpendicular ray (Euclidean distance would give fisheye effect!)

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

else perpWallDist = (sideDistY - deltaDistY);

//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;

• I calcoli seguenti sono nuovi e sostituiscono il selettore di colore del raycaster senza texture. La variabile **texNum** rappresenta il **valore della casella della mappa corrente meno 1**. Questo perché esiste una texture 0, ma il tile della mappa 0 non ha texture, poiché rappresenta uno spazio vuoto. Per poter utilizzare comunque la texture 0, sottrai 1 in modo che i tile della mappa con valore 1 corrispondano alla texture 0, e così via...

Il valore **wallX** rappresenta il **valore esatto del punto di impatto con il muro**, non solo le coordinate intere del muro. Questo è necessario per sapere quale coordinata x della texture utilizzare. Questo **valore viene calcolato prima determinando la coordinata x o y esatta nel mondo e poi sottraendo il valore intero del muro**. Nota che, anche se si chiama wallX, in realtà rappresenta una coordinata y del muro se side == 1, ma è sempre la coordinata x della texture.

Infine, **texX** è la coordinata x della texture, e questa viene calcolata a partire da wallX.

**//texturing calculations  
 //1 subtracted from it so that texture 0 can be used!**

**int texNum = worldMap[mapX][mapY] - 1;**

**//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;**

• Ora che conosciamo la coordinata x della texture, sappiamo che questa coordinata rimarrà la stessa, poiché restiamo nella stessa striscia verticale dello schermo. Ora abbiamo bisogno di un **ciclo nella direzione y per assegnare a ciascun pixel della striscia verticale la corretta coordinata y della texture**, chiamata **texY**.

Il **valore** di **texY** **viene calcolato incrementando di un passo predefinito** (che è costante nella striscia verticale) **per ogni pixel**. **Questo passo indica di quanto aumentare le coordinate della texture** (in virgola mobile) **per ogni pixel nelle coordinate verticali dello schermo**. **Poi è necessario convertire il valore in virgola mobile in un intero per selezionare il pixel effettivo della texture**.

Il colore del pixel da disegnare viene quindi semplicemente prelevato da texture[texNum][texX][texY], che corrisponde al texel corretto della texture giusta.

Come nel raycaster senza texture, anche qui il valore del colore sarà scurito se è stato colpito un lato verticale del muro, poiché il valore del colore non è composto da valori separati di R, G e B, ma da questi 3 byte combinati in un singolo intero, viene utilizzato un calcolo non così intuitivo.

Il colore viene scurito dividendo R, G e B per 2. Dividere un numero decimale per 10 può essere fatto rimuovendo l'ultima cifra (ad esempio, 300/10 è 30: lo zero finale viene rimosso). Allo stesso modo, dividere un numero binario per 2, che è quello che viene fatto qui, equivale a rimuovere l'ultimo bit. Questo può essere fatto spostando i bit a destra con >>1. Tuttavia, qui stiamo spostando a destra un intero a 24 bit (in realtà 32 bit, ma i primi 8 bit non sono utilizzati). Per questo motivo, l'ultimo bit di un byte diventa il primo bit del byte successivo, e questo altera i valori dei colori! Quindi, dopo lo spostamento dei bit, il primo bit di ogni byte deve essere azzerato, e questo può essere fatto eseguendo un'operazione "AND" binaria con il valore binario 011111110111111101111111, che è 8355711 in decimale. Il risultato è infatti un colore più scuro.

Infine, il pixel corrente del buffer viene impostato su questo colore, e poi si passa al pixel successivo nella direzione 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;**

**}**

**}**

• Ora il buffer deve ancora essere disegnato e successivamente svuotato (nella versione senza texture usavamo semplicemente il comando "cls"). Assicurati di farlo in ordine di scanline per migliorare le prestazioni grazie alla località della memoria e alla cache. Il resto del codice è nuovamente identico.

**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

• Ecco di nuovo i tasti; anche qui nulla è cambiato. Se vuoi, puoi provare ad aggiungere i tasti per lo strafe (per muoverti a sinistra e a destra). Questi devono essere implementati nello stesso modo dei tasti su e giù, ma utilizzando planeX e planeY invece di dirX e dirY.

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);

}

}

}

Nota: Di solito, le immagini sono memorizzate per scanline orizzontali, ma per un raycaster le texture vengono disegnate come strisce verticali. Pertanto, per utilizzare in modo ottimale la cache della CPU e evitare i page fault, potrebbe essere più efficiente memorizzare le texture in memoria per strisce verticali anziché per scanline orizzontali. Per fare ciò, dopo aver generato le texture, scambia le loro coordinate X e Y (questo codice funziona solo se texWidth e texHeight sono uguali):

//swap texture X/Y since they'll be used as vertical stripes

for(size\_t i = 0; i < 8; i++)

for(size\_t x = 0; x < texSize; x++)

for(size\_t y = 0; y < x; y++)

std::swap(texture[i][texSize \* y + x], texture[i][texSize \* x + y]);

Oppure, puoi semplicemente scambiare X e Y quando le texture vengono generate, ma in molti casi, dopo aver caricato un'immagine o ottenuto una texture da altri formati, la troverai comunque organizzata per scanline e dovrai scambiarla in questo modo.

Quando ottieni il pixel dalla texture, usa il seguente codice:  
Uint32 color = texture[texNum][texSize \* texX + texY];