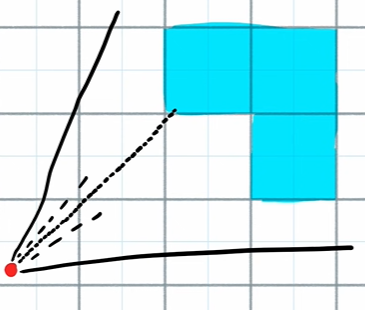
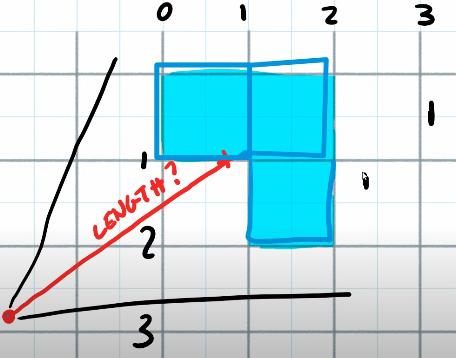
**RAY CASTING ALGORITHM (based on DDA algorithm)**

• Supponiamo di trovarci all’interno di una griglia e che alcune delle caselle siano identificate come “solidi”  
 Quello che teoricamente andrebbe a fare il ray casting è proiettare, all’interno del campo visivo del   
 personaggio, per ogni colonna sullo schermo, un raggio nella direzione in cui il personaggio sta guardando  
 per poi calcolare il punto in cui questo intercetta i punti “solidi”.  
 A quel punto andremmo a calcolare la lunghezza di ogni raggio per creare uno pseudo effetto 3D.

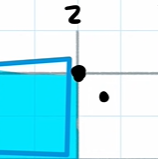
Il problema risiede nel modo in cui proiettiamo questi raggi: conoscendo l’origine e l’angolo del raggio dovremmo andare a incrementarlo di uno step infinitesimale fino a raggiungere il punto di collisione  
e questo per ogni raggio. Ovviamente il costo computazionale di queste operazioni risulterebbe elevato  
ed oltretutto impreciso, potrebbe capitare infatti che in un dato istante l’incremento risulti fuori dal “muro”  
e nell’incremento successivo al suo interno, non rilevando il “bordo” del muro stesso.



• Riassumendo il problema che stiamo cercando di risolvere è: In una mappa a scacchiera  
 qual’è la lunghezza del raggio proiettato da una data posizione, in una direzione conosciuta, fino al   
 punto di collisione con un ostacolo?

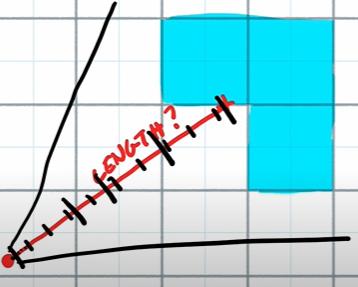


Fortunatamente l’ambiente ci permette alcune semplificazioni. Possiamo assumere che le caselle siano  
**di dimensione costante e con lato unitario**



Questo ci fornisce un sistema di riferimento in cui possiamo definire coordinate non unitarie (ad esempio 2,3) il cui valore può essere troncato per darci la coordinata della cella (che non sarà altra che la posizione della cella più vicina nella matrice della mappa)



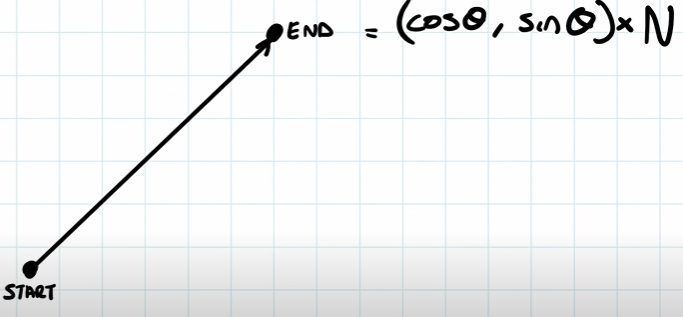
• Ma come risolvere la questione degli infiniti incrementi del raggio?  


Quello che andremo a fare sarà implementare un algoritmo che scarti una discreta mole di passi e prenda in considerazione solo i punti in cui il raffio intercetta una casella della mappa.

Questo risulterà non solo più veloce ma anche più accurato perché i nostri muri avranno il loro perimetro proprio lungo la griglia della mappa!

**IDEA**

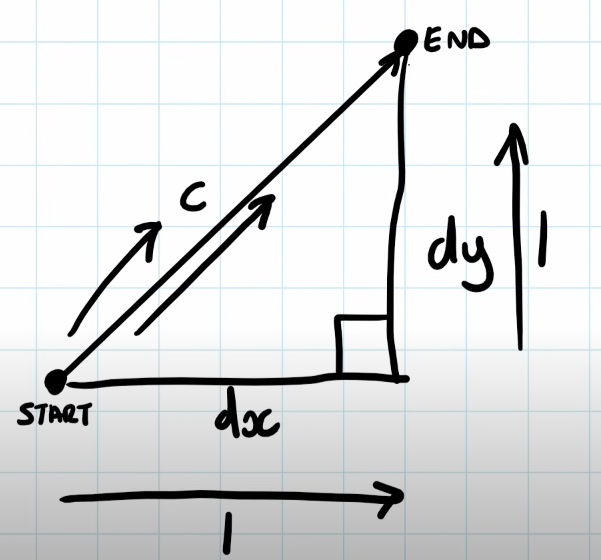
• Partiamo dal presupposto che in ogni istante abbiamo 2 informazioni sul nostro raggio:  
 - Il punto di partenza del raggio (il personaggio)  
 - il punto di arriva oppure, in caso questo non sia noto, la direzione

Nel caso in cui il punto di arrivo non sia conosciuto può essere calcolato semplicemente moltiplicando il coseno ed il seno dell’angolo (che il vettore forma con il sistema di coordinate) con un valore N.  
In realtà possiamo scartare questo valore in quanto, prendendo in considerazione il vettore unitario parallelo al raggio, N risulterà = 1



• Dato un qualsiasi vettore possiamo costruire un triangolo rettangolo con componente x = dx e componente y = dy. A questo punto l’ipotenusa (il nostro raggio) non sarà altro che

• Ora però mi chiedo: Considerando una variazione (o movimento) unitaria lungo l’asse delle x o lungo  
 l’asse delle y, di quanto cambierà la lunghezza del mio raggio?

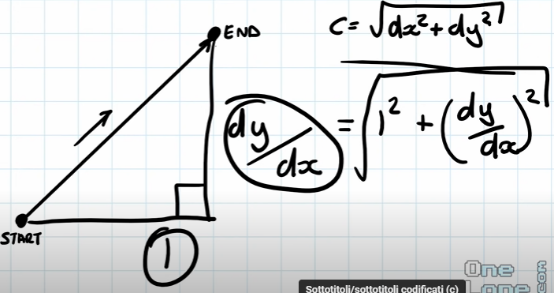


Sappiamo che il gradiente della retta è dato dall’equazione

Quindi, ad esempio, so che muovendomi di un’unità sull’asse delle x (dx = 1) mi muoverò per un intero dy verso l’alto.

Come calcolo l’ipotenusa del triangolo risultante?

Quello che so è che se dx = 1 (ed essendo posso riscrivere dy =

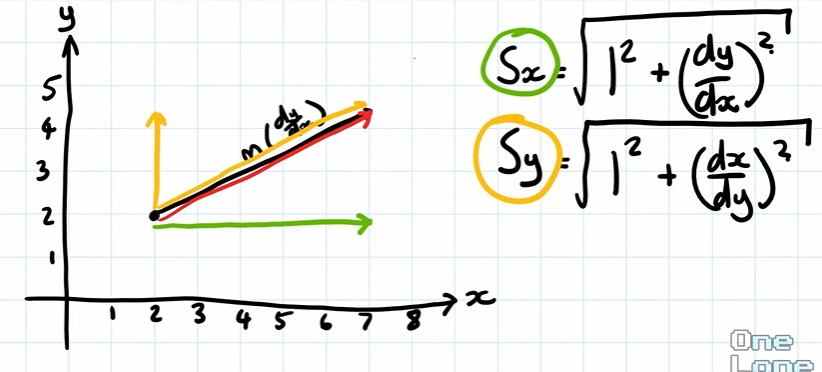
Usando il teorema di pitagora:

Quindi posso dire che **un triangolo con cateto x = 1 avrà** la corrispondente **ipotenusa** di lunghezza

Mentre **un triangolo rettangolo con cateto y di lunghezza = 1 avrà** . Quindi, sempre per il teorema di pitagora, la corrispondente **ipotenusa** avrà lunghezza

• Ho quindi dei **fattori di scala**. Infatti, dato un qualsiasi punto nel nostro spazio diviso in celle unitarie ed un vettore direzione che parta da questo punto, finche conosco la pendenza della mia retta

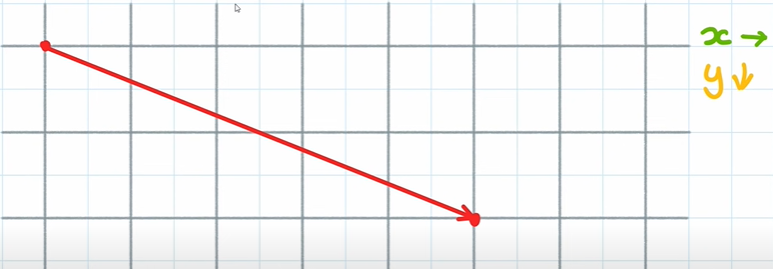
Posso prendere la lunghezza di una delle componenti del vettore e moltiplicarla per il corrispondente fattore di scala per ottenere l’ipotenusa



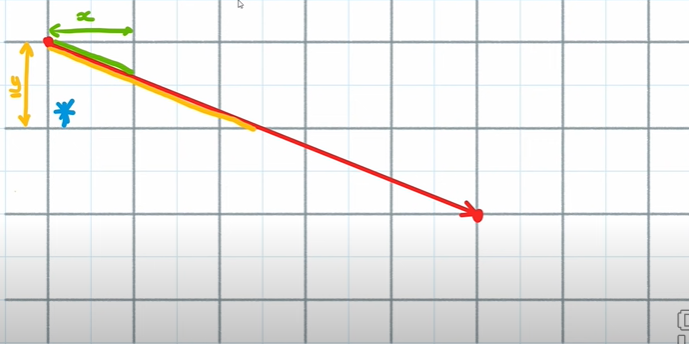
**APPLICAZIONE DEL PRINCIPIO NEL NOSTRO SPAZIO**

**Punti coincidenti con la griglia**  
• Consideriamo due punti giacenti sulla nostra griglia ed andiamo ad applicare manualmente l’algoritmo.

NOTAZIONE: Useremo una freccia verde per identificare i cambi di direzione sull’asse x, una   
 freccia verde per quelli sull’asse y ed un asterisco blu per idicare le caselle che   
 intercetteremo



• Usando i fattori di scala andiamo a calcolare **quanto è lungo il raggio per una singola linea della matrice**  
 (cioè per uno spostamento unitario dell’asse y) e, analogamente, **quanto è lungo il raggio per una singola   
 colonna della matrice**  (cioè per uno spostamento unitario dell’asse x partendo dallo stesso punto)

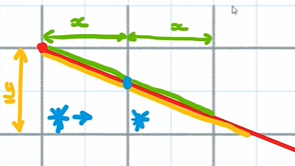


• A questo punto la regola è la seguente: Vado a comparare le due lunghezze, **l’asse lungo il quale mi   
 sposterò sarà quello con il semi-raggio più corto**

   
 Possiamo vedere dall’immagine che il semi-raggio risultate da uno  
 spostamento sull’asse delle x è il più piccolo fra i due, quindi mi muoverò   
 di un’unità lungo le x

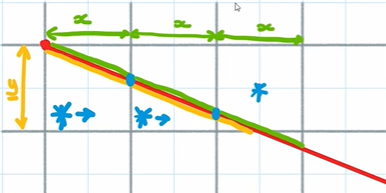
Ora avrò un nuovo punto di partenza (il punto in blu) dal quale rifare lo stesso calcolo

• **Mi muovo** quindi **di una cella nella direzione scelta** (in questo caso la x)

Seguendo lo stesso ragionamento la lunghezza dotale del segmento  
x risulta minore della lunghezza totale del semi-raggio giallo.

* Mi muovo di un’altra unità sulle x

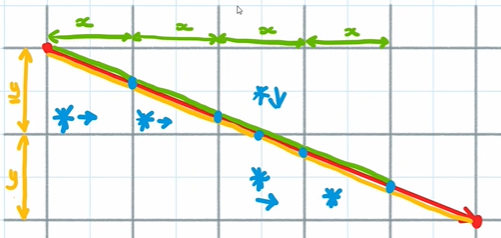
• **Mi muovo** quindi **di una cella** (partendo dal nuovo punto raggiunto) **nella direzione scelta** (anche in questo caso la x)



• Ripeto il procedimento: questa volta la lunghezza totale del semiraggio x è minore di quella del   
 semiraggio y 🡪 mi dovrò muovere di una cella lungo le y (**verso il basso** perché l’asse delle y  
 ha origine in alto a sinistra)



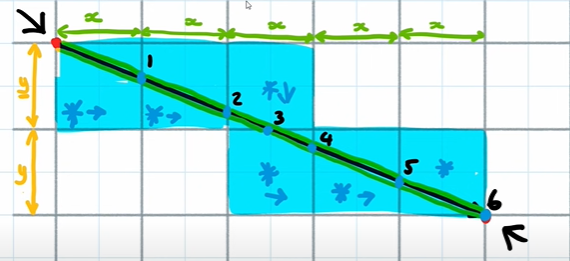
• Reitero: Confronto le due lunghezze e la più corta risulta quella associata al semi-raggio x  
 🡪 Sposto la mia x di una casella e traccio il nuovo semi-raggio partendo dall’ultimo punto  
 raggiunto



• Reitero: Di nuovo il semi-raggio x risulta più corto e mi muovo lungo questo asse

**Controllo il punto raggiunto e questo coincide con il “punto di collisione”** (la casella che volevo   
 raggiungere)

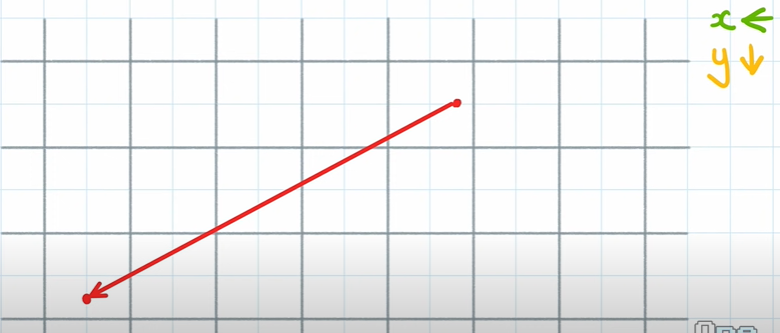
**NOTA:** Ovviamente **la verifica sul punto raggiunto va fatto ad ogni iterazione, controllando   
 se coincide con un muro oppure uno spazio vuoto**



Colorato in blu vediamo le caselle unitarie interessate dal mio raggio, o in generale il “percorso”  
attraversato nel mio spazio unitario dal raggio (non unitario)

Nel disegno vediamo che, invece di suddividere la retta in parti infinitesime e verificare l’avvenuta collisione per ogni passo, **abbiamo controllato solamente 6 punti**.

**Punti non coincidenti con la griglia**  
• Ora vediamo un altro esempio con una diversa direzione e, soprattutto, con punti che non risiedono   
 sulla nostra griglia

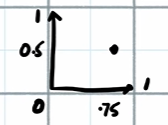


Considerando che, rispetto all’asse x, il mio punto di partenza è maggiore di quello di arrivo dovrò muovermi da destra verso sinistra (decremento la x). Mentre per le y ci muoveremo sempre dall’alto verso il basso dato che il punto di partenza è minore di quello di arrivo

NOTA: **Questa analisi sulla direzione della x e y (quindi del loro incremento o decremento) va fatta   
 prima di poter applicare l0algoritmo di ray casting**.

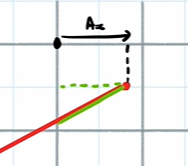
• Ora, considerando il precedente esempio, siamo partiti incrementando di un’unità ogni asse.  
 In questo caso **non possiamo farlo per la nostra prima cella** perché non siamo allineati con la griglia.  
 Cosa fare quindi?

• Ricordiamo che abbiamo supposto che le nostre caselle abbiano lati di lunghezza 1.   
 Quindi, ogni posizione all’interno di questo sistema di riferimento avrà un valore compreso fra 0 e 1  
 che **può essere letto come una percentuale**.

Stiamo cioè normalizzando le coordinate rispetto alla cella

**Cordinata x di partenza:**  
• Non essendo allineata con la griglia questa avrà un valore decimale, es x = 5.3  
 Per ottenere la distanza del nostro punto di partenza dal primo “incrocio di celle”, in modo   
 da poter calcolare la lunghezza del semi-raggio sul nostra asse delle x, ci basterà   
 **“troncare”** (**cioè prendere la parte intera**) **la coordinata x del nostro punto iniziale** (punto nero. Nel nostro esempio è 5) **ottenendo** **la posizione sulla griglia generale   
 (in che casella mi trovo)** .

Questo mi permetterò di ottenere un “nuovo punto” dal quale calcolare il dx del mio segmento  
 (guarda foto)

A questo punto la “nuova distanza iniziale” Ax rispetto alla singola cella – **che sarà il dx del mio punto di partenza** – sarà facilmente calcolabile sottraendogli l’intera coordinata x del punto di partenza

**CALCOLO**

• Per il calcolo abbiamo quindi due situazioni possibili a seconda della direzione del raggio:

**i) verso sinistra**  
Come detto dovrò calcolare manualmente la nuova posizione di partenza rispetto alla singola cella per ottenere Ax.  
Se il raggio si muove verso sinistra (la componente del vettore direzione x è negativa) avrò:

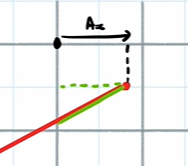
Es: = 5,3 – = 5.0 = 0,3

Calcolo quindi la prima ipotenusa moltiplicando questo valore per l’Sx visto in precedenza

**ii) verso destra**  
Muovendomi invece verso destra (la componente del vettore x è > 0) quello che cerchiamo è  
“la lunghezza del raggio dal punto di partenza all’uscita della prima cella”

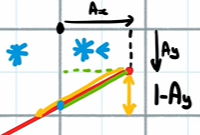
Per ottenere questo starting point relativo alla cella dovrò sottrarre il valore i della cella successiva  
() alla componente x del mio punto di partenza

Es: = 6.0) - = 5,3) = 0,7



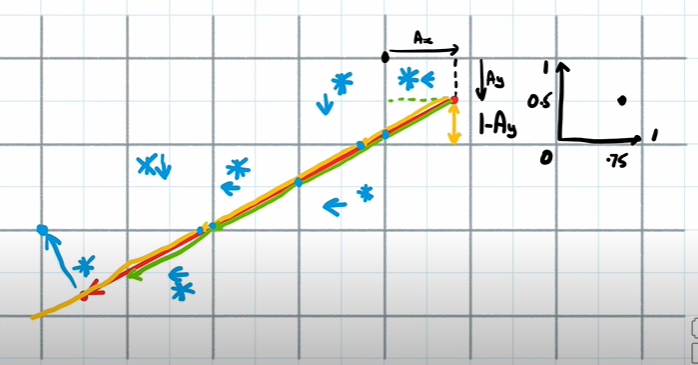
Calcolo quindi la prima ipotenusa moltiplicando questo valore per l’Sx visto in precedenza

**Coordinata y di partenza:**  
• Considerata la direzione in cui il mio raggio sta viaggiando nella sua componente y (verso il basso)  
 il calcolo del tratto iniziale è simili ma bisogna fare un’osservazione:  
 Non andrò a trovare Ax ma 1 -Ay



**CALCOLO**• Per calcolare l’ipotenusa rispetto alle x si eseguono gli stessi identici calcoli usati per la x,  
 sostituendo la componente y alla componente x

Ora possiamo applicare lo stesso algoritmo visto precedentemente muovendoci di un’intera unità  
per ogni coordinata rispettando l’ordine del “più corto” e controllando se abbiamo raggiunto un punto di collisione



NOTA: Osserviamo che nessuno dei due semi-raggi ha raggiunto il punto ma questo non ci interessa.  
 Noi siamo guardando le celle raggiunte e dall’immagine possiamo che l’ultima iterazione   
 del raggio x va a colpire proprio la casella indicata come “solida”

NOTA: Considerando che conosciamo la lunghezza del raggio ad ogni iterazione possiamo   
 facilmente calcolare il punto di collisione sommando:

**IMPLEMENTAZIONE**(Basato su un programma in C++)

- **vettore** **float** (matrice bidimensionale) **ray\_start** = posizione\_personaggio   
- **vettore float** (matrice bidimensionale**) ray\_direction** = vettore unitario che rappresenta la direzione in cui  
 guarda il personaggio  
(queste variabili avranno componente x ed y)

- creo una funzioni che calcoli il fattore di scala Sx ed Sy: **v\_ray\_unit\_step\_size**

- devo tenere traccia di quale cella mi trovo in ogni momento. Creerò un un nuovo vettore bidimensionale  
 (matrice bidimensionale): **int map\_check = ray\_start** 🡪 vado cioè ad inizializzare questa matrice   
 con le componenti x ed y di partenza del giocatore troncate alla parte intera  
 (dato che è un vettore int)

- Sappiamo che dovremo confrontare la lunghezza dei semi-raggi sia lungo l’asse x che y.  
 Avrò bisogno quindi di una nuova matrice bidimensionale che aggiungerà ad ogni iterazione  
 queste lunghezze: **ray\_len\_1d**

- Ci muoveremo lungo il raggio nella stessa direzione sia per le x che per le y ma dobbiamo comunque  
 tenere traccia di quale è l’una e quale l’altra. Creo quindi un nuovo vettore “int steps”  
  
- applico quanto detto sopra per il calcolo del punto di partenza:

if (ray\_dir.x < 0) //il raggio va verso sinistra)

{

    step.x = -1;

    ray\_len\_id.x = (ray\_start - (float)map\_check.x) \* ray\_unit\_step\_size.x;

}

else

{

    step.x = 1;

    ray\_len\_id.x = ((float)map\_check.x + 1.0 - ray\_start) \* ray\_unit\_step\_size.x;

}

if (ray\_dir.y < 0) //il raggio va verso sinistra)

{

    step.y = -1;

    ray\_len\_id.y = (ray\_start - (float)map\_check.y) \* ray\_unit\_step\_size.y;

}

else

{

    step.y = 1;

    ray\_len\_id.y = ((float)map\_check.y + 1.0 - ray\_start) \* ray\_unit\_step\_size.y;

}

- Avrò ora bisogno di un ciclo con due condizioni:  
float current\_distance = 0;

float max\_distance = 1000;

/\*ora mi serve un ciclo con condizione: "finche non ho una collisione &&

  la distanza attuale è minore di una distanza massima prestabilita"\*/

while (!(trovata una collisione) && (current\_distance < max\_distance))

{

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*WALK\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    //comparo le distanze

    if (ray\_len\_1d.x < ray\_len\_1d.y)

    {

        map\_check.x += step.x;  /\*in map\_check avevo le coordinata iniziali x ed

                                  y del personaggio. Semplicemente andrò ad  
 aggiungere al valore x + o -1 salvato in steps\*/

        current\_distance = ray\_len\_1d.x; /\*aggiorno la distanza percorsa \*/

        ray\_len\_1d.x += ray\_unit\_step\_size; /\*Come detto io dovrò sempre fare la

                                              comparazione con i valori totali

                                              dei semi-raggi ottenuti. Quindi ad

                                              ogni iterazione sommo alla lunghezza

                                              del segmento ray\_len\_1d.x il nuovo

                                              segmento ottenuto   
 ray\_unit\_size.x\*/

    }

    else

    {

        map\_check.y += step.y; /\*stessa cosa se l'ipotenusa y è quella piu lunga\*/

        current\_distance = ray\_len\_1d.y;

        ray\_leb\_1d.y += ray\_unit\_step\_size.y

    }

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CHECK\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

    /\*controllo di non uscire fuori dalla mappa\*/

    if (map\_check.x >= 0 && map\_check.x < mapsize.x && map\_check.y >= 0 &&  
 map\_check.y < mapsize.y )

    {

        if (la cella in cui mi trovo nella mappa == solido)

        {

            trovata una collisione

        }

    }

}

Una volta usciti dal ciclo controllo se ho effettivamente trovato una collisione

/\*Creo un nuovo vettore 2D\* intersection\*/

if (collisione trovata == true)

{

    /\*questo sarà il nostro punto di collisione\*/

    intersection = ray\_start + ray\_direction \* distance;

}