42 PARIS - JCHOTEL

TUTORIEL CUB3D

Ce projet est inspiré du jeu éponyme mondialement connu, considéré comme le premier FPS jamais développé. Il vous permettra d'explorer la technique du ray-casting. Votre objectif est de faire une vue dynamique au sein d'un labyrinthe, dans lequel vous devrez trouver votre chemin.

Mon projet ne suit pas strictement ce tutoriel (les règles, ma compréhension, et mes connaissances ayant évoluées).

Sommaire

Sommaire 2

Principe 4

Parsing du Fichier 5

- **Taille d'écran** 5
- **Map** 5
- **Validité de la carte** 5

Minimap 6

- **MiniLibX** 6
  - Affichage d'une fenêtre
  - Affichage d'un carré
  - Affichage d'un rond
  - Utiliser les images
- **Affichage de la Minimap** 6
  - Affichage de la carte
  - Affichage du joueur

Déplacements 7

- **Récupérer la saisie clavier** 7
- **Déplacements rectilignes** 7
- **Déplacements arrières et latéraux** 7
- **Déplacements multidirectionnels** 8

Calculs Préparatoires 9

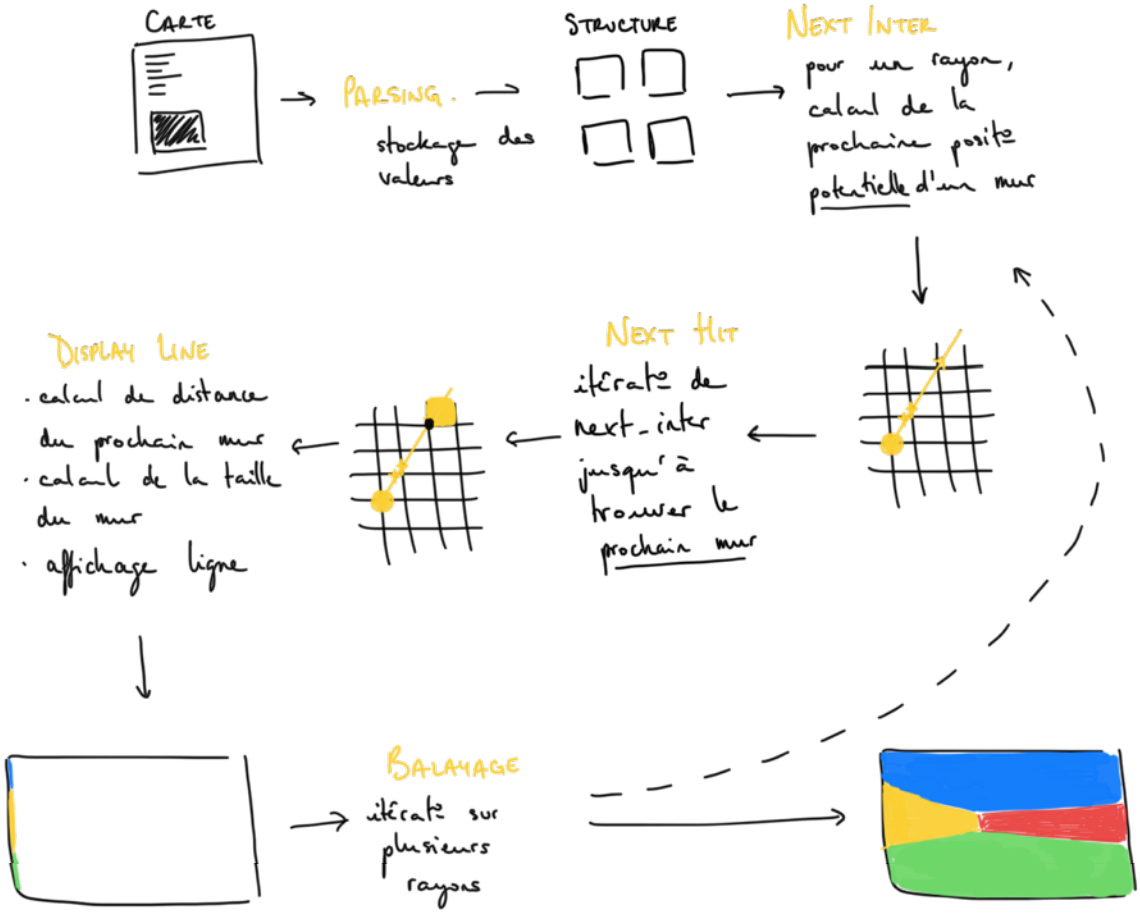
- **Premier rayon** 9
  - Tracer une droite …
  - … en fonction d'un angle …
  - … depuis le joueur…
  - … jusqu'au mur.
- **Optimisation** 10
  - Prochaine intersection…
  - … horizontales ou verticale
  - Dessiner le mur

Details et Améliorations 12

- **Textures** 12
  - N-S-E-W

Les fichiers .xpm
Mathématiques 14
Théorème de Pythagore 14
Distance 14
Les degrés et les radians 14
Cosinus et Sinus 14

Principe



1 - Le fichier d'entré doit être parser.
2 - Les valeurs doivent alors être correctement placées dans des structures.
3 - Pour une direction donnée, la prochaine intersection sera calculée - position potentiel d'un mur
4 - Ce calcul sera itéré jusqu'à la présence d'un mur
5 - L'affichage d'un mur (de taille inversement proportionnel à la distance) serra effectué.
6 - Les étapes 3-4-5 seront répétées afin d'obtenir un visuel complet.

Ci-après, des étapes pas-à-pas seront proposées et expliquer afin de progresser dans la réalisation du projet.
Les personnes qui ne cherchent qu'une aide mathématique peuvent directement se diriger vers la partie X - Calculs préparatoires ou vers la partie X - Mathématiques qui énumère et explique les formules.

Parsing du Fichier

Le parsing du fichier, càd la dissection du fichier d'entrée, peut se faire avec la fonction get_next_line(). Il faudra alors stocker toutes ces données dans une/des structure(s).
Par exemple, aux premiers abords, les structures game et player peuvent être instanciées de la manière suivante :

GAME	
INT	taille_x
INT	taille_y
PLAYER	player
CHAR[][]	map

PLAYER	
FLOAT	pos_x
FLOAT	pos_y
FLOAT	alpha

D'autres valeurs comme la santé, le type d'arme, etc. pourront être rajoutées par la suite.

Ici nous verrons les premières étapes du parsing, sans nous attarder sur les textures, puisque celles-ci ne seront utilisées que bien plus tard.

Taille d'écran

En utilisant le principe de la fonction get_next_line(), il est possible de récuperer tour à tour les informations du fichier.
Créez la fonction parsing(); pour chaque ligne, il faudra vérifier si les 2 premiers caractères correspondent à un 'R ' suivit d'un espace. Le cas échéant, il faudra donc récupérer les deux valeurs taille_x et taille_y via un simple ft_atoi().
Prenez le temps de tester des cas de fichiers invalides et de renvoyer une erreur si besoin.

Map

Dans la nouvelle version de cub3d, la carte peut prendre des formes non rectangulaires.
Une solution envisageable est donc de stocker les lignes dans une liste chainée. Puis de récupérer la taille de celle-ci et de la ligne la plus grande, afin de créer le tableau de char aux dimensions maximales.
La version du sujet a évoluer depuis : [Pour remplir le tableau map, il sera suffira de prendre un caractère sur 2 et de vérifier que le caractère d'espace est bien un espace : une ligne ne pourra pas être « 1 0 0 000 1 » ni « 1 0 0 0.0 1 ».]
Réalisez la fonction parse_map() qui remplira correctement la variable char [][] map et qui sera judicieusement appelée dans la fonction parsing().

Validité de la carte

Maintenant que les donnés sont récupérer, il est nécessaire de vérifier leur validité.

MINIMAP

L'affichage de la minimap n'est pas nécessaire mais permet d'appréhender la mlx dans un contexte simplifié - 2D. Aussi, cette étape permettra d'assurer la bonne réalisation du parsing et des déplacements du personnage.

MiniLibX

Affichage d'une fenêtre

Dans un premier temps, ouvrez une fenêtre aux dimensions voulues.

Affichage d'un carré

Maintenant, utilisez les fonctions basiques de la mlx pour créer une fonction de type draw_square(int pos_x, int pos_y, int size_x, int size_rec, int color) qui permettra d'afficher un carré / rectangle.

Affichage d'un rond

Le principe est simple, quoique, moins que pour le carré : créez une fonction draw_circle(int pos_x, int pos_y, int ray, int color). *En fait, c'est comme un carré de taille = rayon mais avant de dessiner le pixel, il faut verifier que la distance* du pixel par rapport au centre, est inférieure au rayon.*

TIPS : FAIRE UNE STRUCTURE VECTEUR OU COORD AVEC UN X ET UN Y

Utiliser les images

Plutôt que de directement modifier la fenêtre de jeu, modifier une image qui sera plus tard afficher via une fonction de type draw_image(). En fait, vous noterez plusieurs intérêts à celles-ci. Elles pourront par exemple être superposée dans un ordre donnée, mais sont notamment plus rapide à modifier (évite donc à votre jeu de laguer)

Affichage de la Minimap

Affichage de la carte

Maintenant que notre structure de donnée est prête, nous avons accès à la carte dans un format char[][]. Il faut donc parcourir ce tableau, et dans une image dessiner ou non un carré symbolisant un mur. Créez la fonction draw_mini().

TIPS : DANS UN .H FAIRE UN #DEFINE SIZE_MINI POUR SIMPLIFIER LES FUTURS REGLAGES

Affichage du joueur

Avec la fonction draw_circle(), il est maintenant très simple de rajouter le joueur dans la minimap.

Déplacements

Maintenant que le visuel est en place, les déplacements du joueur peuvent être ajouter et visualiser sur la minimap.

ATTENTION : LES MATHS PROPOSÉES SI DESSOUS SONT BASÉES DANS UN REPÈRE NORMAL

Par convention mathématiques, le x est l'axe horizontal vers la droite, et le y est l'axe vertical ascendant.
Hors, dans le cas présent, l'axe des y est descendant ; ceci nécessitera donc quelques inversions et modifications mais le principe reste correcte.

Récupérer la saisie clavier

La Minilibx fournit toutes les fonctions nécessaires pour récupérer la clé des touches utilisées.
Familiarisez vous avec celles-ci et chercher à actualiser une variable tel que l'angle de vision ou la position en x et y.

Déplacements rectilignes

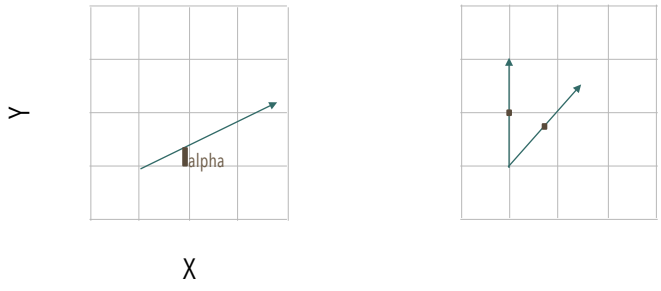
Nous conviendrons que la position du joueur évolue de la sorte à chaque itération de jeu :

$$x_{joueur} = x_{joueur} + \partial x$$
$$y_{joueur} = x_{joueur} + \partial y$$

TIPS : ∂ EST LU DELTA. IL S'AGIT D'UNE PETITE VARIATION DE LA VARIABLE

Le joueur a donc une nouvelle position égale a l'ancienne plus un petit déplacement sur les x et les y

Se déplacer suivant une direction donnée et d'une distance fixe nécessite de se plonger dans quelques petits calculs.
En effet, ∂x et ∂y ne sont pas simplement égaux à 1 comme illustré ci-après. Par exemple, pour le déplacement vertical, on a $\partial x = 0$ et $\partial y = 1$. Dans le deuxième cas, nous avons $\partial x < 1$ et $\partial y < 1$.



En fait, les déplacements équivalent sur les axes x et y sont calculables via le cosinus et le sinus*, dont toutes les explications se trouvent dans la section mathématiques.

$$x_{joueur} = x_{joueur} + \cos(alpha)$$
$$y_{joueur} = x_{joueur} + \sin(alpha)$$

Déplacements arrières et latéraux

Pour terminer la gestion des déplacements, il faudra s'assurer de gérer correctement les mouvements de recul et sur les cotés.
Ces déplacements équivalent mathématiquement aux mouvements précédents pour les angles suivants.

- recul :- alpha
- cotés : alpha ± 90°

TIPS : IL EXISTE DES SIMPLIFICATIONS TRIGONOMÉTRIQUES

pour cos/sin(-alpha) et cos/sin(alpha ± 90°)

Déplacements multidirectionnels

A preciser ….
si on veut faire des déplacements multidirectionnels : (je sais plus exactement mais il faudra utiliser une structure ou qq chose de similaire (nombre binaire) pour savoir quelles touchent sont maintenues appuyées en simultané)

Calculs Préparatoires

La difficulté de ce projet réside essentiellement en cette partie calculatoire. En effet, le principe de raycasting fait appel à plusieurs notions mathématiques détaillées ci-après.
Le principe du raycasting est d'envoyer un « rayon » qui détectera le prochain obstacle. Une fois détectée, sa distance sera calculée, et la taille dudit obstacle sera alors déduite. Pour ce faire, il faut donc savoir tracer une droite.

Premier rayon

Tracer une droite …

Vous retrouverez plus de détails et d'explications dans la partie X - Mathématiques au sujet des droites*.
L'équation d'une droite est la suivante :

$$y = mx + b$$

Avec 'm', le coefficient directeur et 'b' l'ordonnée à l'origine.
Ainsi, pour tracer une droite il suffirait de dire :

```
| pour x = début à x = fin
| |   y = mx + b
| |   tracer le point (x, y)
| |   x += ∂x
```

Réalisez une fonction qui vous permettra de tester plusieurs valeurs de **début**, **fin**, **coefficient directeur, ordonnée à l'origine** et **∂x**.

TIPS : SI POUR UN M TRÈS GRAND, LA DROITE N'EXISTE PAS, PAS DE PANIQUE

… en fonction d'un angle …

Le coefficient directeur donne l'inclinaison de notre droite comme vous avez dû le remarquer plus haut ou en partie X - Mathématiques. Il existe donc un lien entre coefficient directeur et l'angle d'inclinaison :

$$m = (y2 - y1) / (x2 - x1) = \sin(alpha) / \cos(alpha) = \tan(alpha).$$

Modifiez donc votre fonction pour quelle prenne en paramètre un angle donné - à savoir celui dans lequel regarde le joueur.
Vous devez maintenant pouovir modifier l'orientation de la droite via le clavier.

TIPS : UN 'M' TRÈS GRAND CORRESPOND À UN ALPHA = ±90

$\text{SIN}(\pm 90) = \pm 1$ && $\text{COS}(\pm 90) = 0$

$M = 1 / 0 \Longrightarrow$ pas possible

Nous traiterons ce problème plus tard

… depuis le joueur…

Puisque le coefficient m est maintenant connu, le seul paramètre manquant est donc b.
L'ordonnée à l'origine correspond au point d'intersection de la droite avec l'axe des y. Elle permet donc de déplacer, vers le haut ou vers le bas la droite. Dans le cas de ce projet, on ne se soucis pas de connaitre cette ordonnée, en revanche, le rayon doit provenir du joueur. Mathématiquement, cela signifie que les coordonnées du joueur doivent valider l'équation.

$$y_{joueur} = mx_{joueur} + b$$

$$y_{joueur} - mx_{joueur} = b$$

$$b = y_{joueur} - mx_{joueur}$$

Modifiez votre fonction pour qu'elle trace une droite en fonction de la position du joueur de sorte à ce que le rayon parte du joueur.

… jusqu'au mur.

Grace aux précédentes étapes, nous trouvons l'équation suivante :

$y = mx + b$ équation de droite

$y = \tan(alpha)\, x + b$ coefficient directeur en fonction d'alpha

$y = \tan(alpha)\, x + (y_{joueur} - mx_{joueur})$ b en fonction du joueur

$$y = \tan(alpha)\, x + (y_{joueur} - \tan(alpha)\, x_{joueur})$$

Modifiez votre fonction pour qu'elle s'arrête lorsqu'elle rencontre un mur.

TIPS : DANS LE TABLEAU DE CHAR[][] CORRESPONDANT À LA CARTE, IL FAUT REGARDER SI LA CASE CORRESPOND À UN MUR OU NON

Optimisation

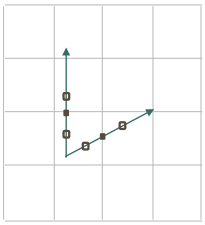
Prochaine intersection…

Jusqu'alors nous avons fixer nous même le ∂x. Il est possible de faire quelques observations à ce sujet :

- Plus le ∂x est petit, plus il y a de calculs intermédiaires, la droite est mieux tracée et plus précis est la detection de mur.
- Plus le ∂x est grand, moins il y a de calculs intermédiaires, la droite est appauvrie et la détection de mur est moins efficace

Il est donc nécessaire de choisir cette valeur de manière pertinentes afin d'éviter les calculs inutiles et couteux en temps mais d'assurer une détection exacte des murs.

Pour ce faire, nous allons délibérément choisir les positions potentielles de mur. Plutôt que de choisir des x à interval régulier (qui par la suite permettent de calculer les y), nous allons tester un couple (x,y) sélectionné pour savoir si un mur est présent. Les points que nous cherchons à tester sont sur les intersections qu'il s'agisse d'intersections verticales ou horizontales, comme illustré ci-après.



Pour une position (x, y) donnée, la prochaine intersection aura pour coordonnée la partie entière de la position (+ 1).
C'est à dire :

- si le joueur est en (1.2, 2.8) et qu'il regarde en haut, le prochain mur (potentiel) dans son champs de vision sera sur l'axe y = 2 = la partie entière de 2.8.
- si le joueur regarde cette fois à sa droite, le prochain mur (potentiel) sera sur l'axe x = 2 = la partie entière de 1.2 + 1.

Pour savoir si cette prochaine intersection est un mur, il nous faut connaitre la deuxième coordonnée (x ou y en fonction du type d'intersection). Nous utiliserons la fonction de droite précédemment établi {$y = \tan(alpha)\, x + (y_{joueur} - \tan(alpha)\, x_{joueur})$}.

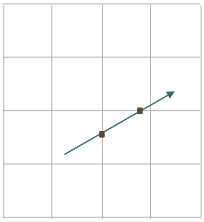
TIPS : DANS LE CAS D'UNE INTERSECTION DONT ON CONNAIT LE Y, IL SUFFIT DE RÉARRANGER L'ÉQUATION POUR ISOLER LE X

Ecrivez deux fonctions qui affiche sur la minimap le prochain point d'intersection, horizontal et vertical :

Vector inter_v(Vector v, Player p);

Vector inter_h(Vector v, Player p);

… horizontales ou verticale



Pour résumer :

- Nous avons vu comment tracer une droite partant du joueur et l'arrêter sur le prochain mur
- Nous avons vu comment calculer les prochains points d'intersections (horizontal et vertical) avec le mur potentiel.

Il nous faut donc assembler ces deux notions.

Le principe sera le suivant :

- tant que le Vector v n'est pas un mur
  - calculer l'intersection verticale suivante
  - calculer l'intersection horizontale suivant
  - Vector v = la plus proche du joueur (distance*).

Il faudra donc implémenter une petite fonction qui pour un vecteur donnée regarde s'il s'agit d'un mur ou non.

TIPS : LES INTERSECTIONS N'APPARTIENNENT PAS AU 2 CASES A LA FOIS.

Par exemple lorsque l'on regarde vers la gauche et que le joueur se trouve en (1.5, 1.5), l'intersection sera (1, 1.5).
Dans notre imaginaire, il appartient donc à la case [0, 1] et et [1, 1], mais mathématique il n'appartient qu'à [1, 1].
Il suffira donc de rajouter une petite règle en fonction de l'angle pour checker la bonne case.

Dessiner le mur

Maintenant que nous savons détecter un mur. Il est possible de calculer sa distance*. De là, on en déduit sa taille (plus le mur est loins, plus il est petit et inversement).

Créez une fonction qui dessine une ligne de la hauteur souhaité, centré sur l'écran en hauteur et à la position x donnée. Elle sera complétée par un ciel et un sol. void draw_line(int x, int height).

Aussi, nous devons itérer sur chaque pixel de l'écran. Nous aurons donc une boucle « while (x < taille_ecran) », de ce x découle un angle de vision de ±30° par rapport au player. Ce ∂alpha s'exprime de la manière suivante en fonction de x :

la bande x = 0, ∂alpha = -30°

la bande x = largeur écran, ∂alpha = +30°

x / largeur de l'écran = (∂alpha + 30°) / 60°

x * 60° / largeur de l'écran = ∂alpha + 30°

(x * 60° / largeur de l'écran) - 30° = ∂alpha

∂alpha = (x * 60° / largeur de l'écran) - 30°

Nous chercherons donc à tracer pour chaque $x_{écran}$ la bande :

- nous calculons le ∂alpha correspondant
- nous calculons les prochaines intersections jusqu'à trouver un mur
- nous dessinons le mur à la taille et à la position $x_{écran}$ sur la fenêtre

Details et Améliorations

À ce stade, vous devriez avoir un rendu visuel relativement fini : deux images composent la fenêtre, la minimap et le premier rendu 3D (celui-ci étant pour l'heure uniquement de 3 couleurs - murs, ciel et sol). Votre personnage doit pouvoir se déplacer dans la carte correctement.

Textures

N-S-E-W

Commençons par faire la distinction entre nos 4 types de murs. Pour ce faire nous devons savoir 2 choses :

- s'il s'agit d'une intersection de type horizontal (Nord ou Sud) ou vertical (Est ou Ouest)
- si les signes du cosinus et du sinus
  - cos > 0 et sin > 0 - regarde en haut à droite (Est ou Nord)
  - cos < 0 et sin > 0 - regarde en haut à droite (Nord ou West)
  - cos < 0 et sin < 0 - regarde en haut à droite (West ou Sud)
  - cos > 0 et sin > 0 - regarde en haut à droite (Sud ou Est)

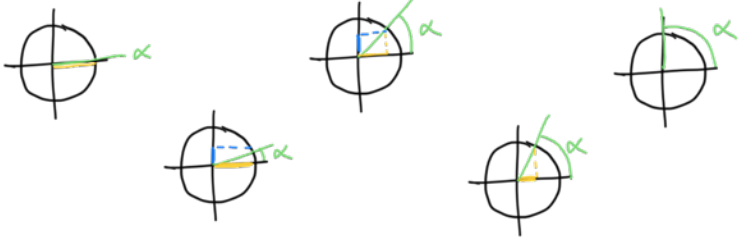
Ainsi, en recoupant les 2 infos, il est possible de connaitre le type de mur.
L'information concernant le type d'intersection était disponible dans la partie … horizontales ou verticale. Cherchez donc un moyen de la récupérer et de dessiner les murs d'une couleurs différentes pour chaque type.

Les fichiers .xpm

…chercher a afficher une image dans un premier temps sans distortion
… la suite pour plus tard…

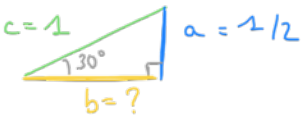
⊕ Valeurs Cos & Sin

	0°	30°	45°	60°	90°
cos	1	$\sqrt{3}/2$*	$\sqrt{2}/2$*	1/2	0
sin	0	1/2	$\sqrt{2}/2$*	$\sqrt{3}/2$*	1



* il est possible de calculer ces valeurs via le théorème de Pythagore :

• cos 30° :



$a^2 + b^2 = c^2$

$\left(\frac{1}{2}\right)^2 + b^2 = 1$

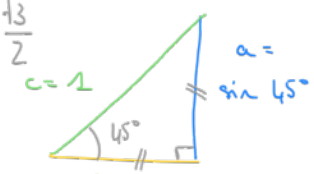
$\frac{1}{4} + b^2 = 1$

$b^2 = 1 - \frac{1}{4}$

$b = \sqrt{\frac{3}{4}} = \frac{\sqrt{3}}{\sqrt{4}} = \frac{\sqrt{3}}{2}$

| et réciproque pour sin 60°.

• cos 45° et sin 45° :



$a^2 + b^2 = c^2$

$\cos^2 45 + \sin^2 45 = 1$

$2\cos^2 45 = 1$

$\cos^2 45 = \frac{1}{2}$

$\cos 45 = \sqrt{\frac{1}{2}} = \frac{1}{\sqrt{2}}$

// cos 45 = sin 45

Théorème de Pythagore

« Les 2 petits cotés au carré égalent le carré du grand. » Il s'agit d'un fait concernant les triangles dits « rectangles » (ceux qui possède un angle droit).
Dans ce cas on appelle le coté opposé à l'angle droit ,l'hypoténuse, 'H' et les deux autres cotés 'A' et 'B'.
Les deux petits cotés au carré : $A^2 + B^2$
Égalent le carré du grand : $= H^2$
Soit :

$$A^2 + B^2 = H^2$$

On isole le H qui est alors au carré -> opération inverse du carré = racine

$$\sqrt{(A^2 + B^2)} = \sqrt{H^2}$$

Simplification possible du coté du H :

$$H = \sqrt{(A^2 + B^2)}$$

Distance

Une distance est calculée via le théorème de Pythagore. En effet, le déplacement sur l'axe des X et celui sur l'axe des Y forment un angle à 90°, soit un triangle rectangle. L'hypoténuse correspond donc à la distance recherchée.

Les degrés et les radians

Bien que les deux soient liées, une conversion doit être effectuée pour passer de l'un à l'autre (comme on passerait de kilomètre à des miles).
On sait que 360° équivalent à 2π.

360° <=> 2π

Pour convertir des degrés en radians
1° <=> 2π / 360°
x° <=> x * 2π / 360°

Pour convertir de radians en degrés
360° / 2π <=> 1 radians
x * 360° / 2π <=> x radians

Les fonctions trigonométriques (cos, sin, tan etc) utiliseront les radians.

Cosinus et Sinus

Ces deux notions peuvent être très simplement appréhendées en prenant l'image suivante :

- une personne tiens un baton de 1 mètre. Il forme un angle (que nous appellerons Alpha) avec le sol.
- si une lumière se trouve au dessus de ce baton, alors une ombre sera projetée au sol. La taille de l'ombre, en mètre, s'appellera le cosinus.
- si une lumière se trouve à coté de ce baton, alors une ombre sera projetée sur le mur. La taille de l'ombre, en mètre s'appellera le sinus.

TIPS : UN CERCLE «TRIGONOMÉTRIQUE » EST SIMPLEMENT UN CERCLE DE RAYON = 1

Plusieurs valeurs sont alors remarquables :

Degrés	0°	30°	45°	60°	90°
Radian	0	π / 6	π / 4	π / 3	π / 2
COSINUS	1				0
SINUS	0				1

Pour toutes les autres valeurs, nous appellerons simplement les fonctions mathématiques cosinus et sinus.