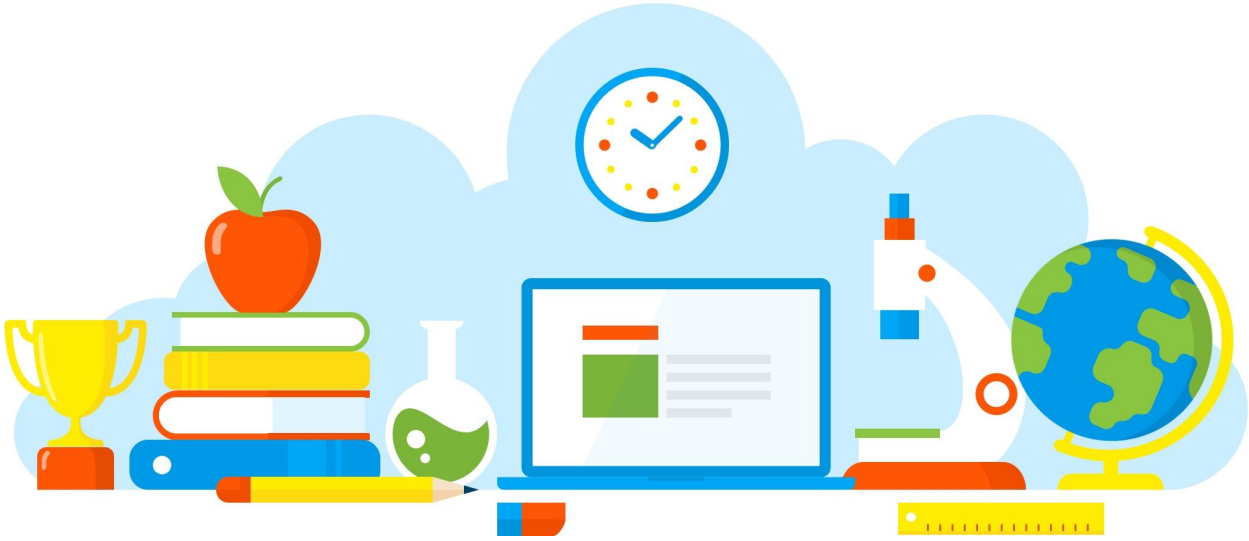


Notions mathématiques du projet Cub3D

Algorithme de Raycasting

Tuteurs 42 Lyon
Quentin PUPIER (Qpupier)
25/03/21

Introduction et utilisation de la documentation

Ce document est destiné à vous aider sur les aspects mathématiques du projet Cub3D.

Vous trouverez donc dans un premier temps une rapide introduction aux notions mathématiques mobilisées par ce projet, avant de trouver les chapitres thématiques s'appuyant directement sur les différents aspects du sujet à traiter.

Remarques et suggestions sont les bienvenues. Les tuteurs restent également disponibles pour vous expliquer ces notions, en complément du présent document.

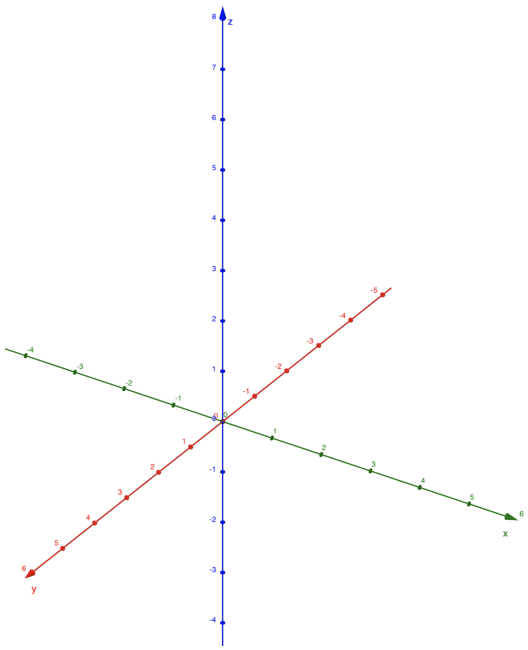
Bonne lecture et bon Cub3D à vous !

Introduction aux concepts mathématiques géométriques

Présentation des différentes nomenclatures mathématiques de la géométrie dans l'espace.

Le repère 3D orthonormé

Le repère orthonormé signifie que tous les axes ont le même ratio : une unité *x* est égale à une unité *y* et est égale à une unité *z* ; et que tous les axes sont perpendiculaires.



Dans le cas présent, pour Cub3D, je conseille vivement le repère ci-contre.

En effet, c'est vous qui choisissez votre repère. Celui-ci est intéressant car il permet de garder la logique de la map 2D avec un repère orthonormé 2D *x*/*y* qui la représente.

Figure 1.

Le point

Le point dans l'espace est représenté par 3 coordonnées *x*, *y*, *z*. Il est fixe.

Il est noté ainsi : P (x, y, z)
Il est possible de voir sa nomenclature en vertical *(voir figure ci-contre)*.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Figure 2.

Le vecteur

Le vecteur a la même notation qu'un point avec une flèche sur sa lettre. Il représente une direction. Ses 3 coordonnées *x*, *y* et *z* sont le pas de déplacement sur chacun des axes.

Un vecteur n'est pas fixe, il peut être utilisé n'importe où dans le repère.

Cas particulier, un point est un vecteur partant de 0.

Note : Il ne faut créer qu'un seul objet (structure) pour les vecteurs et les points.

Le plan

Le plan est une surface plane infinie dans l'espace tridimensionnel.

Il est représenté par l'équation suivante :
ax + by + cz + d = 0, avec *a*, *b*, *c*, et *d* des réels.

C'est-à-dire que pour *a*, *b*, *c*, et *d* choisis, cela représente un seul et unique plan avec une équation le représentant. Une équation représente un seul et unique plan.

Exemple : *3x + 2y - 5z + 42 = 0*
(voir figure ci-contre).

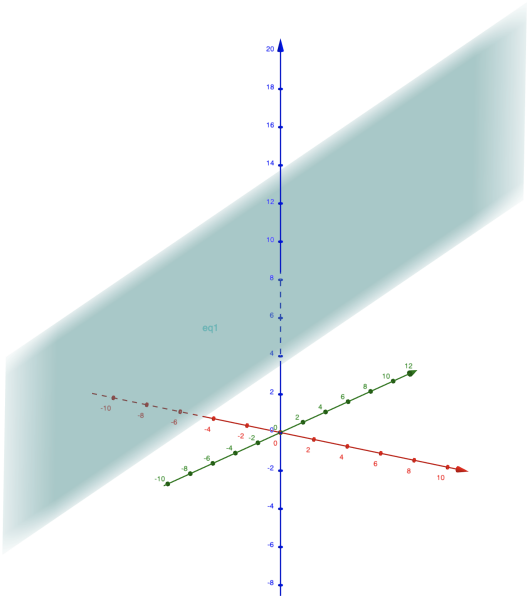


Figure 3.

Si on choisit un point et que ses coordonnées *(x, y, z)* respectent l'équation alors ce point appartient au plan.
L'équation représente l'infinité des points qui constituent un plan.

Les équations

Les équations sont les égalités qui permettent de calculer une ou plusieurs inconnues.

Exemple : $3x + 2 = 42$.

La droite

La droite dans l'espace tridimensionnel est représentée par une équation paramétrique *(voir figure ci-contre)*.

$$\begin{cases} x = O_x + u_x * t \\ y = O_y + u_y * t \\ z = O_z + u_z * t \end{cases}$$

Figure 4.

O représente un point de la droite et *u* le vecteur directeur de celle-ci.

La droite est infinie car *t* appartient aux nombres réels non nuls ($t \in \mathbb{R}^*$).

La demi-droite (le rayon)

La demi-droite est identique à la droite à la seule différence qu'elle est finie d'un côté.

Dans la *Figure 4*, *t* appartient ici aux réels strictement positifs ($t \in \mathbb{R}_+^*$).

La trigonométrie

La trigonométrie sert en géométrie à calculer des longueurs et des angles dans des triangles rectangles. Les 3 formules utilisées sont *cosinus* (*cos*), *sinus* (*sin*) et *tangente* (*tan*).

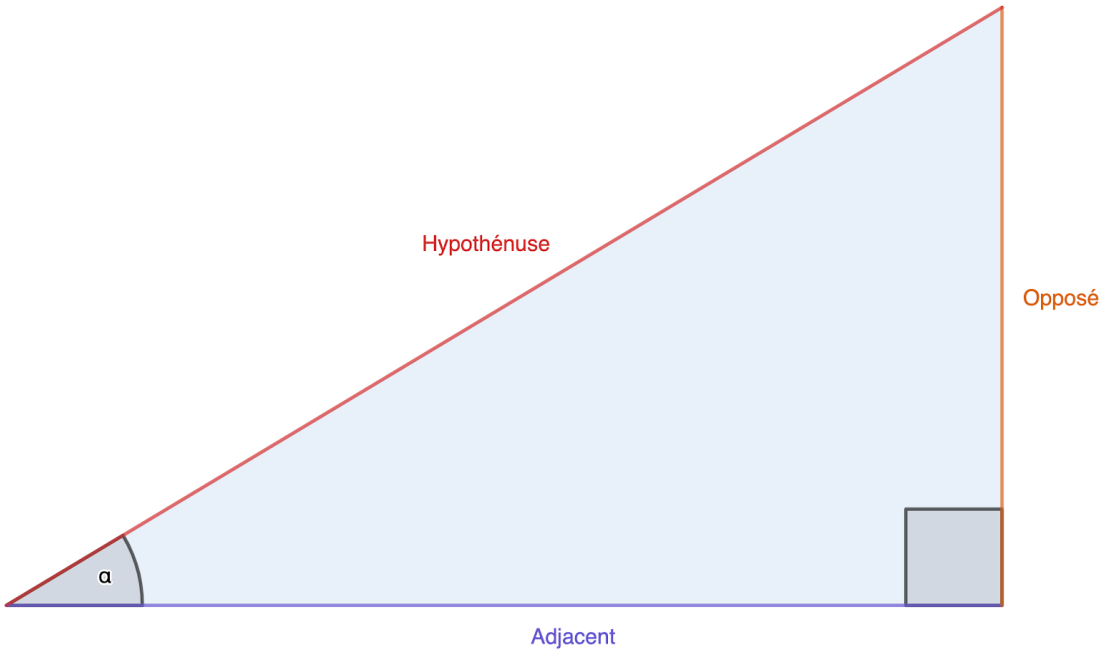


Figure 5.

Moyen mnémotechnique : *Casse-toi !* (**CAH SOH TOA**)

L'*hypoténuse (H)* est le côté opposé à l'angle droit (plus grand côté).

Le côté *adjacent (A)* représente le côté entre l'angle droit et l'*angle étudié (α)*.

Le côté *opposé (O)* est celui qui est opposé à l'*angle étudié (α)*.

Le ***cosinus*** de l'angle est égal au côté *adjacent (A)* sur l'*hypoténuse (H)* : $cos(\alpha) = A / H$.
Le ***sinus*** de l'angle est égal à l'*opposé (O)* sur l'*hypoténuse (H)* : $sin(\alpha) = O / H$.
La ***tangente*** de l'angle est égale à l'*opposé (O)* sur l'*adjacent (A)* : $tan(\alpha) = O / A$.

Note : ATTENTION à bien mettre l'angle en radian pour les fonctions informatiques.
Pour passer un angle en degré en radian : $radian = degré * \pi / 180$.

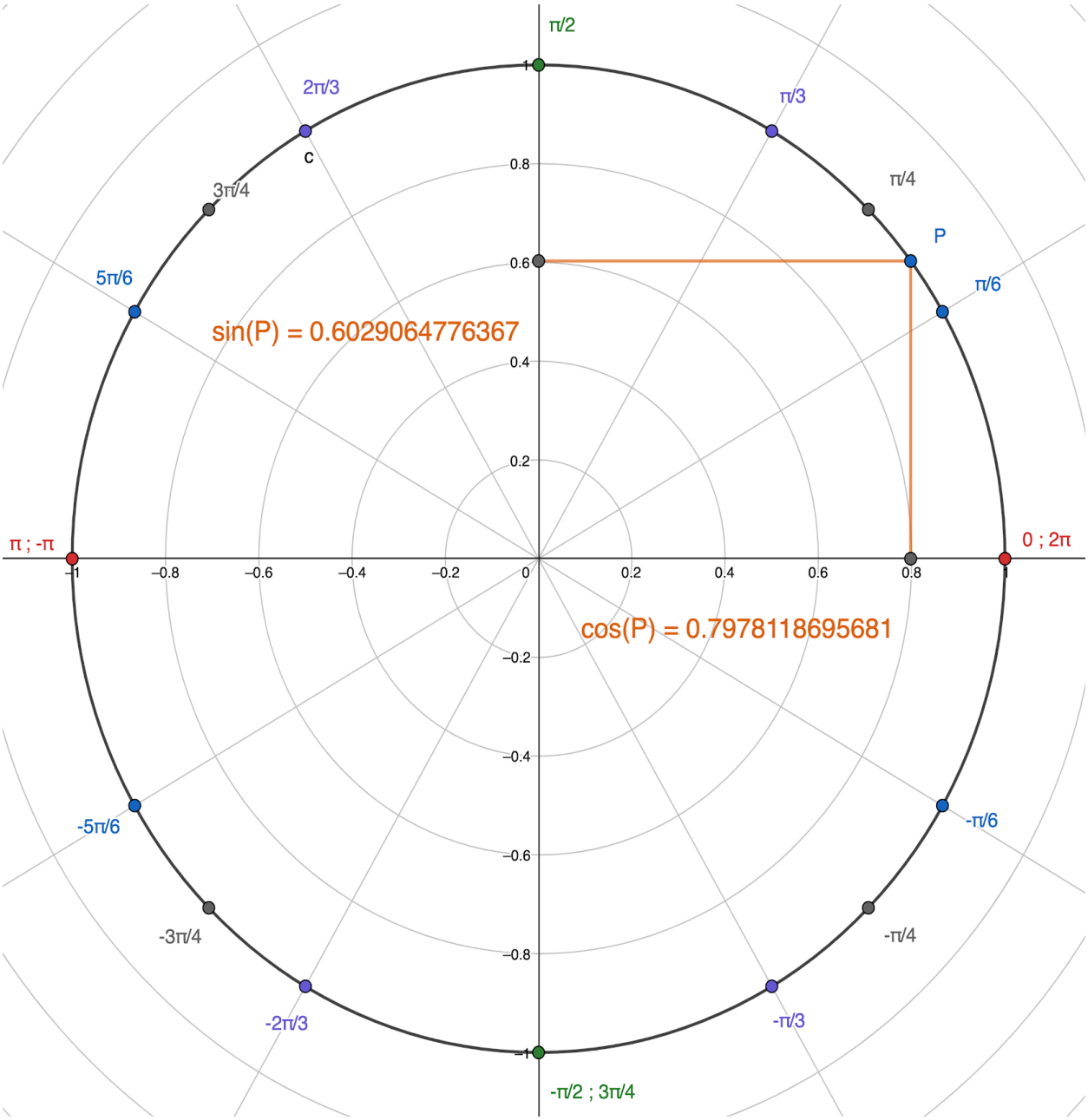


Figure 6 (Lien Geogebra).

Le repère polaire ci-dessus représente les angles en **radians** et la valeur de leur **cosinus** et **sinus**.

(Vous pouvez interagir avec le graphique Geogebra en déplaçant le point **P** via le lien de la *Figure 6*).

Création des rayons (caméra)

Présentation des différentes étapes pour créer les rayons, c'est-à-dire les vecteurs directeurs 3D de la vision du personnage qui représentent la vue du joueur.

Définition du FOV (Field Of View)

Le FOV peut être défini à 60 degrés soit π / 3 radians (champ de vue standard).

*On note **W** la largeur de la fenêtre et **H** sa hauteur.*

*Pour représenter l'entièreté du champ de vision, il faut un rayon par pixel de l'écran sur notre fenêtre. Il faut donc créer **W** * **H** rayons.*

Répartition planaire

Commençons par répartir les rayons sur un seul plan et notamment sur le plan horizontal.

Instinctivement, on divise l'angle de vision (le **FOV**) par le nombre de rayons pour tous les obtenir. Créer les rayons ainsi génère du Fish-eye ; ce n'est donc pas la bonne méthode *(voir Figure 7)*. La vision ainsi générée est l'arc de cercle représenté et non une surface plane.

En effet, si l'on fait cela, les intersections sur un plan virtuel à 1 de distance de l'origine sont à des distances différentes les unes des autres. Plus les rayons sont à l'extérieur et plus les rayons auront des distances importantes entre eux.

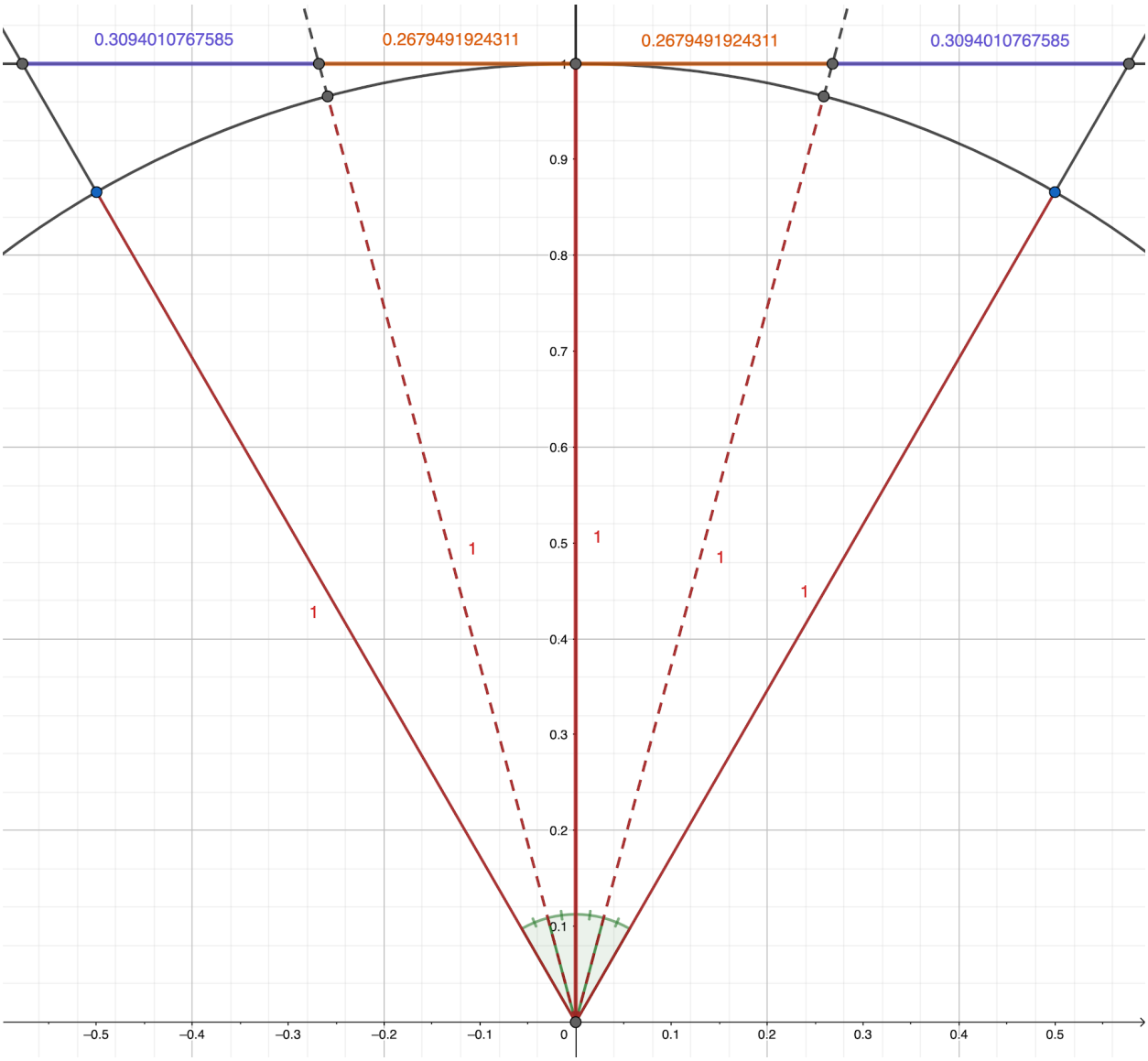

0.3094010767585
0.2679491924311
0.2679491924311
0.3094010767585
1
1
1
1
1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
−0.5
−0.4
−0.3
−0.2
−0.1
0
0.1
0.2
0.3
0.4
0.5

Figure 7.

Correction du Fish-eye

Pour pallier ce problème, il faut prendre en compte la distance obtenue sur ce-dit plan virtuel *(voir Figure 8)*.

Plutôt que de tourner nos vecteurs et donc d'obtenir des distances différentes entre eux, nous allons les créer directement avec une distance constante entre eux.

Il faut donc diviser la distance **D** par le nombre de rayons (**W**) pour obtenir l'espace exact identique entre chaque rayon.

Pour obtenir cette distance **D**, nous devons préalablement calculer la distance **O** (moitié de **D**).

Celle-ci se calcule grâce à la trigonométrie (voir triangle rectangle de droite dans la *figure 8*).
Nous connaissons l'angle **α** qui est la moitié du **FOV** et qui mesure donc ***FOV*** */ 2* radians, ainsi que la longueur du côté adjacent qui est de **1**.
Nous cherchons la longueur du côté opposé (**O**).

Nous avons donc l'équation suivante :

$$tan(\alpha) = O / 1$$

$$\Leftrightarrow tan(FOV / 2) = O.$$

Ainsi : ***O*** *=* ***tan****(****FOV*** */ 2)*.
Or ***D*** *= 2 *** ***O***
donc ***D*** *= 2 *** ***tan****(****FOV*** */ 2)*.

On peut ensuite définir **R_H**, le ratio horizontal (la distance entre les vecteurs sur l'axe horizontal) entre chaque rayon.

Nous avons donc: ***R_H*** *=* ***D*** */* ***W*** donc ***R_H*** *= 2 *** ***tan****(****FOV*** */ 2) /* ***W***
(dans l'exemple, R_H ≈ 0.29).

Le ratio vertical **R_V** se calcule grâce à **FOV_V**, le **FOV** vertical proportionnel au **FOV** horizontal. On a ***FOV_V*** *=* ***FOV*** *** ***H*** */* ***W*** donc
R_V *= 2 *** ***tan****(****FOV_V*** */ 2) /* ***H*** *= 2 *** ***tan****(****FOV*** *** ***H*** */* ***W*** */ 2) /* ***H***

donc ***R_V*** *= 2 *** ***tan****(****FOV*** *** ***H*** */ (****W*** *** *2)) /* ***H***.

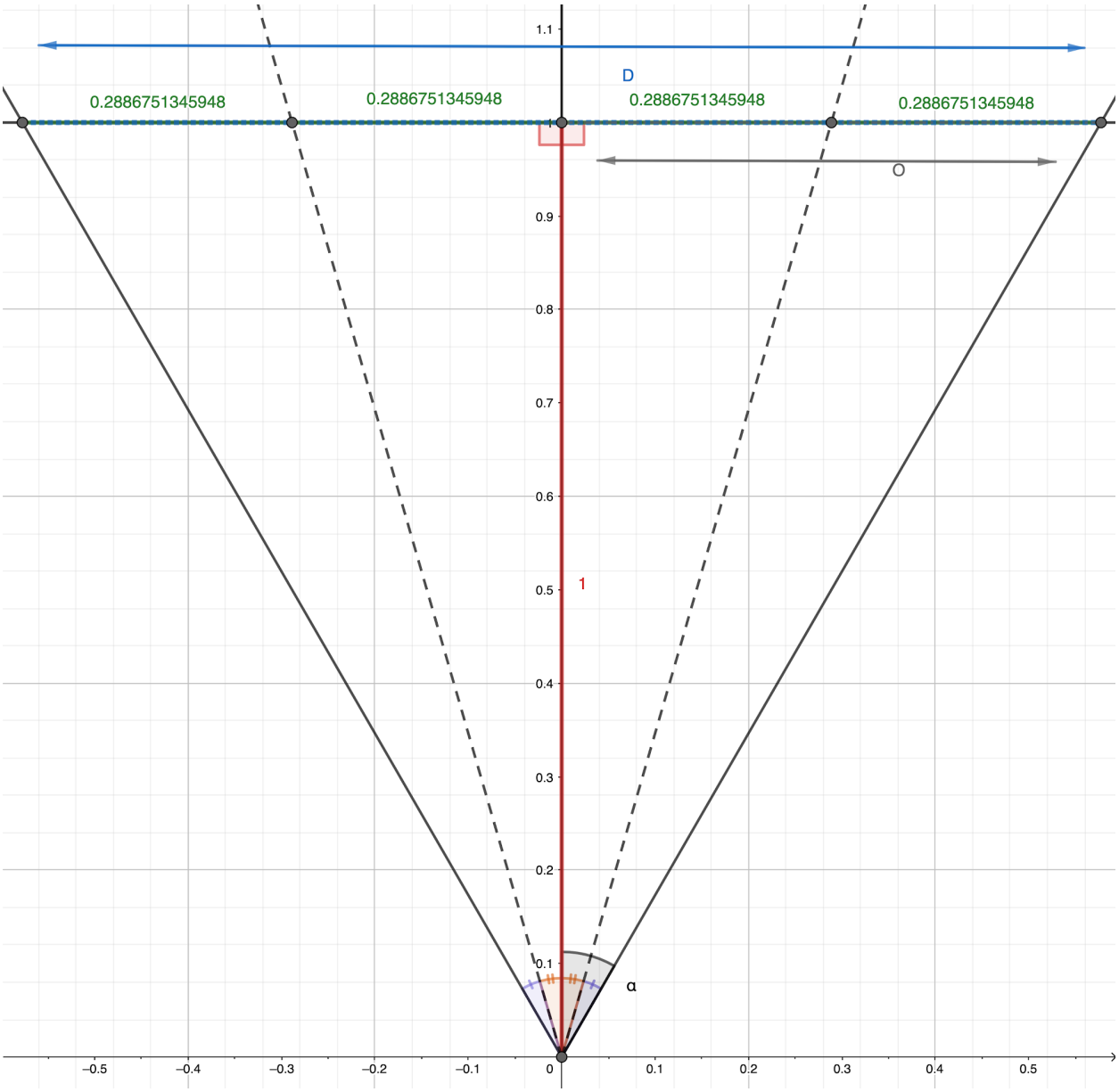

D
0.2886751345948
0.2886751345948
0.2886751345948
0.2886751345948
O
1
α
1.1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
−0.5
−0.4
−0.3
−0.2
−0.1
0
0.1
0.2
0.3
0.4
0.5

Figure 8.

Création des rayons

*Pour créer nos vecteurs directeurs, il suffit donc de mettre un incrément sur les x (que nous appellerons **i**) et un autre pour les y (**j**). Si l'on regarde au centre en direction du nord par exemple, notre vecteur directeur de rayon sera (0, -1, 0).*

Calculs

Pour simplifier et optimiser les calculs, nous choisirons de toujours créer les rayons en prenant une orientation Nord.

Donc il faut multiplier **R_H** par **i** pour la coordonnée **x**, laisser **-1** pour les **y** et multiplier **R_V** par **j** pour la coordonnée **z**.

Cependant, **i** doit être compris entre -***W*** */ 2* et ***W*** */ 2 - 1* donc pour **i** allant de *0* a ***W*** *- 1*, on doit avoir *(**i** - **W** / 2)*. Idem pour les **z** : on a donc *(**j** - **H** / 2)*.
Note : La division est plus lente à l'exécution que la multiplication.
On aura donc *((**i** - **W** * 0.5) * **R_H**, -1, (**j** - **H** * 0.5) * **R_V**)*.
Dernier point, l'axe *z* est inversement proportionnel au vecteurs verticaux de la fenêtre. Il faut donc prendre l'opposé pour la coordonnée **z** :
*-(**j** - **H** * 0.5) * **R_V*** soit *(**H** * 0.5 - **j**) * **R_V***.

Génération

Conclusion, les rayons peuvent donc être créés sous cette forme :

$$\begin{pmatrix} (i - W * 0.5) * R_H \\ -1 \\ (H * 0.5 - j) * R_V \end{pmatrix}$$

Figure 9.

Note : Il peut être utile de normaliser les vecteurs ici si vous comptez réaliser des bonus.

Création des plans (map)

Présentation des différentes étapes pour créer les plans représentant les murs sur la map et ceux représentant les sprites.

Murs

Equations de plans

Les équations des murs sont très simples *(voir Figures 10 ci-dessous)*.
Pour les murs verticaux, les murs vont de $x = 0$ a $x = w$, avec **w** la largeur de la map *(ici w = 4)*.
Et pour les murs horizontaux, les murs vont de $y = 0$ a $y = h$, avec **h** la hauteur de la map *(ici h = 3)*.

a : x = 0

b : x = 1

c : x = 2

d : x = 3

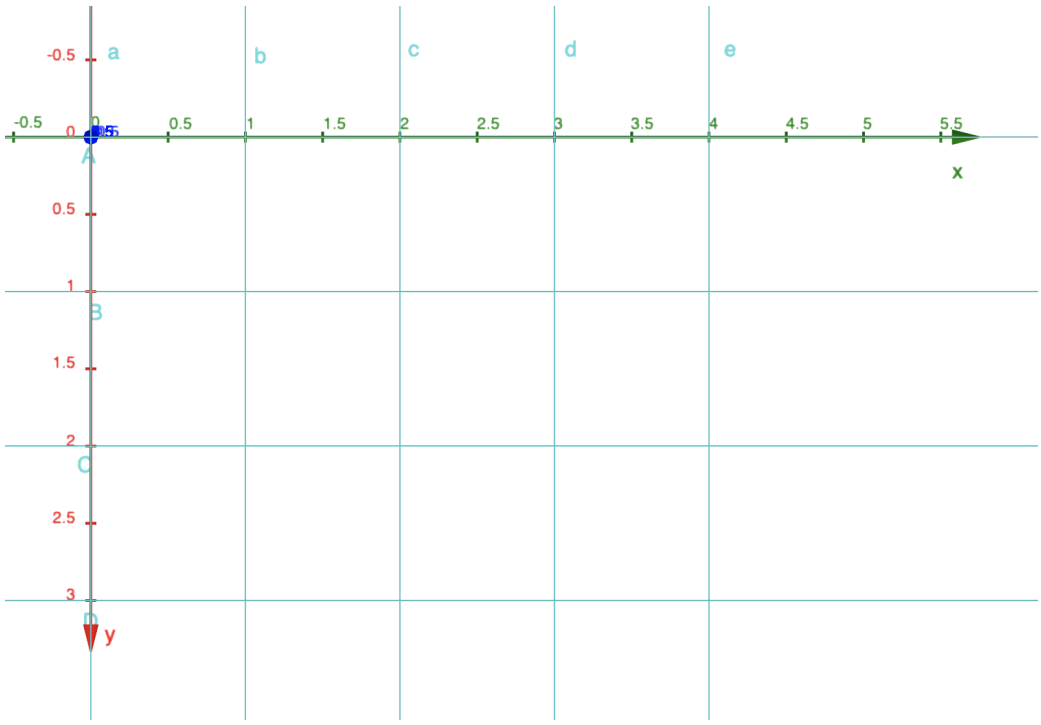
e : x = 4

A : y = 0

B : y = 1

C : y = 2

D : y = 3



Figures 10.

Pour rappel, les équations de plans sont de la forme $ax + by + cz + d = 0$.

*Par exemple, $x = 3$ deviendra donc $x - 3 = 0$, avec **a** = 1, **b** = 0, **c** = 0 et **d** = -3 ($1x + 0y + 0z - 3 = 0$).*

De manière générale, les plans horizontaux sont définis de cette manière, **d** étant un entier allant de *0* à *-**h*** *(voir Figure 11)*.

$$\begin{cases} a = 0 \\ b = 1 \\ c = 0 \\ d = [0, -h] \end{cases}$$

Figure 11.

Et les plans verticaux ainsi, **d** étant un entier allant de *0* a *-**w*** *(voir Figure 12)*.

$$\begin{cases} a = 1 \\ b = 0 \\ c = 0 \\ d = [0, -w] \end{cases}$$

Figure 12.

Début d'algorithme de raycasting

Pour commencer et tenter de minimiser les erreurs, il est préférable de tester les rayons et l'algorithme avec les 4 plans suivants : *{0, 1, 0, 3}*, *{1, 0, 0, 3}*, *{1, 0, 0, -3}* et *{0, 1, 0, -3}* et de mettre la position de la caméra (**O**) en *(0, 0, 0.5)*.

Sprites

La gestion des sprites est quelque peu plus compliquée.

Équation du plan

Le sprite est une texture plane qui est d'autant plus réaliste quand il nous fait constamment face.

Précision concernant les plans : **le vecteur de coordonnées *(a, b, c)* est un vecteur normal du plan**. C'est-à-dire que le vecteur est perpendiculaire au plan.

Or étant donné que le sprite nous fait face, son plan est exactement perpendiculaire au vecteur allant de notre position au centre de ce-dit sprite. Nous appellerons ce vecteur **v**. Avec un centre de sprite **S** et **O** la position de la caméra, nous avons donc ***v*** = ***S*** - ***O***.

On souhaite cependant que le sprite soit parallèle à l'axe vertical (z) donc ***vz*** = *0*.

Les coordonnées *a*, *b* et *c* du plan sont donc calculées : *a* = ***vx***, *b* = ***vy*** et *c* = *0*.

Il ne manque plus que *d*. Si l'on prend un point du plan connu, par exemple **S**, et qu'on remplace *x*, *y* et *z* par ses coordonnées, cela nous donne une équation à une seule inconnue :

$$ax + by + cz + d = 0$$
$$\Leftrightarrow v_x x + v_y y + 0z + d = 0$$
$$\Leftrightarrow v_x S_x + v_y S_y + d = 0$$
$$\Leftrightarrow d = -v_x S_x - v_y S_y$$

Le plan du sprite est donc ainsi :

$$\begin{cases} a = v_x \\ b = v_y \\ c = 0 \\ d = -v_x S_x - v_y S_y \end{cases}$$

Figure 13.

Intersection et texture

On note **I** le point d'intersection de notre rayon avec le plan du sprite.

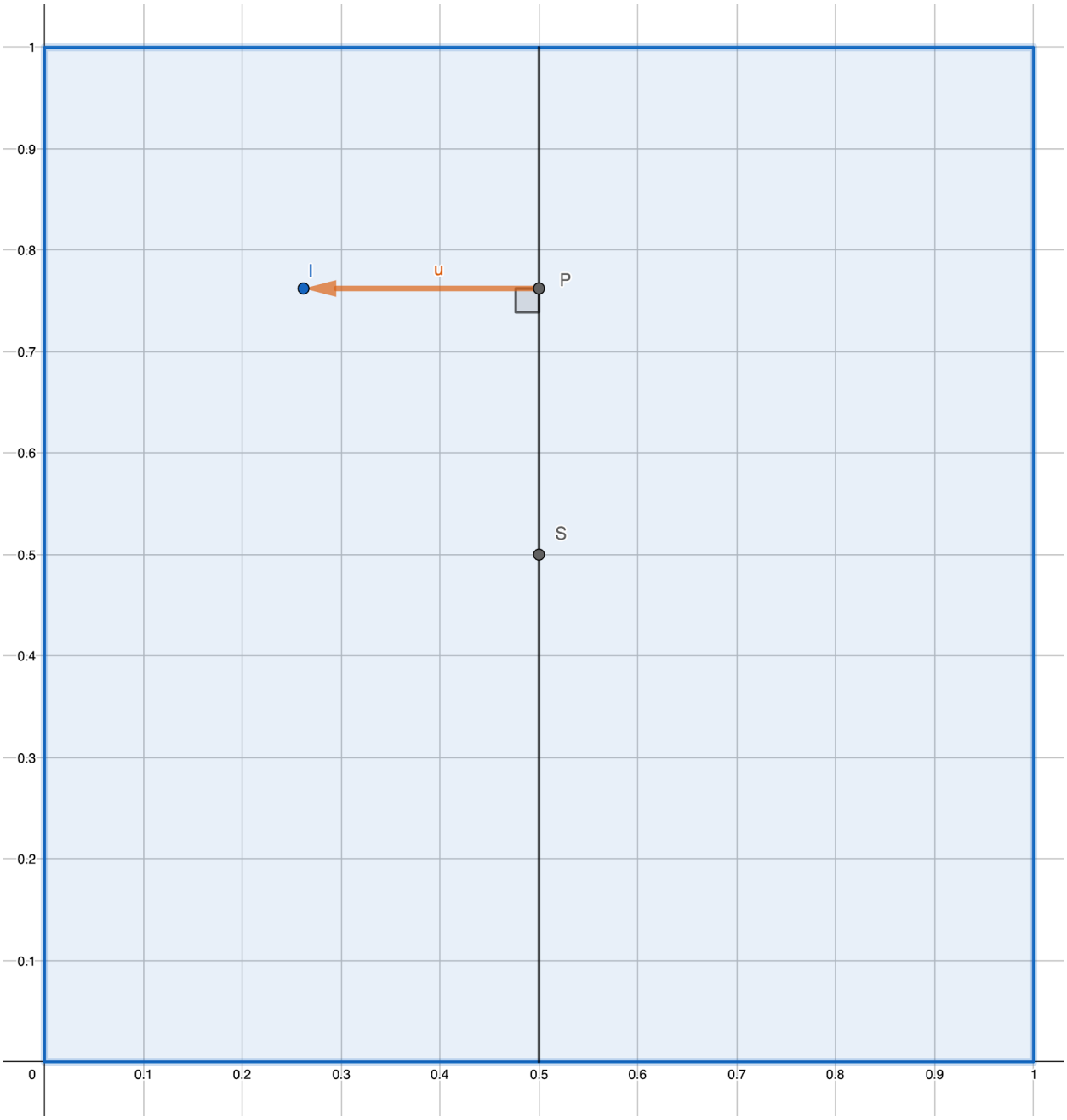


Figure 14 (Lien Geogebra).

On note **P** le projeté orthogonal de **I** sur la droite verticale au centre du sprite, passant par **S**. **P** a pour coordonnées (***Sx***, ***Sy***, ***Iz***) *(voir schéma ci-dessus -> vue de face)*.

On note **u** le vecteur allant de **P** à **I** donc ***u*** = ***I*** - ***P***.

On note **v'** le vecteur **v** tourne de *90°* dans le *sens horaire*. Pour cela, on applique au vecteur initial la matrice de rotation suivante :

$$\begin{pmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 15.

L'angle θ est donc de -π / 2 :

$$\begin{pmatrix} cos(-\frac{\pi}{2}) & -sin(-\frac{\pi}{2}) & 0 \\ sin(-\frac{\pi}{2}) & cos(-\frac{\pi}{2}) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 16.

Ce qui donne la matrice suivante :

$$\begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 17.

En multipliant cette matrice par **v**, nous obtenons **v'** :

$$\begin{pmatrix} v_y \\ -v_x \\ 0 \end{pmatrix}$$

Figure 18.

On note ensuite **v2** le vecteur **v'** normalisé.

Sur la figure ci-dessous, le sprite doit être représenté par le segment vert. Il doit toujours faire *1* de largeur ; c'est pourquoi sa représentation est en cercle (il sera toujours pile dans ce cercle quelle que soit son orientation). Pour éviter qu'il ne prenne la taille du segment rouge, il faut vérifier que la norme du vecteur **u** ne soit pas supérieure à *0.5* (cela se vérifiera automatiquement par les calculs suivants).

Pour finir, on note **r** le produit scalaire entre **u** et **v2** auquel on ajoute *0.5*. Cela nous donne ***r*** = ***u*** · ***v2*** + *0.5*. Cette distance ainsi obtenue représente le ratio de la position horizontale du point d'intersection **I** sur le sprite. Il servira ensuite à afficher la texture. Pour l'instant, il suffit de vérifier que **r** soit compris entre *0* inclus et *1 (soit* $0 <= r < 1$*)* pour pouvoir afficher le sprite correctement.

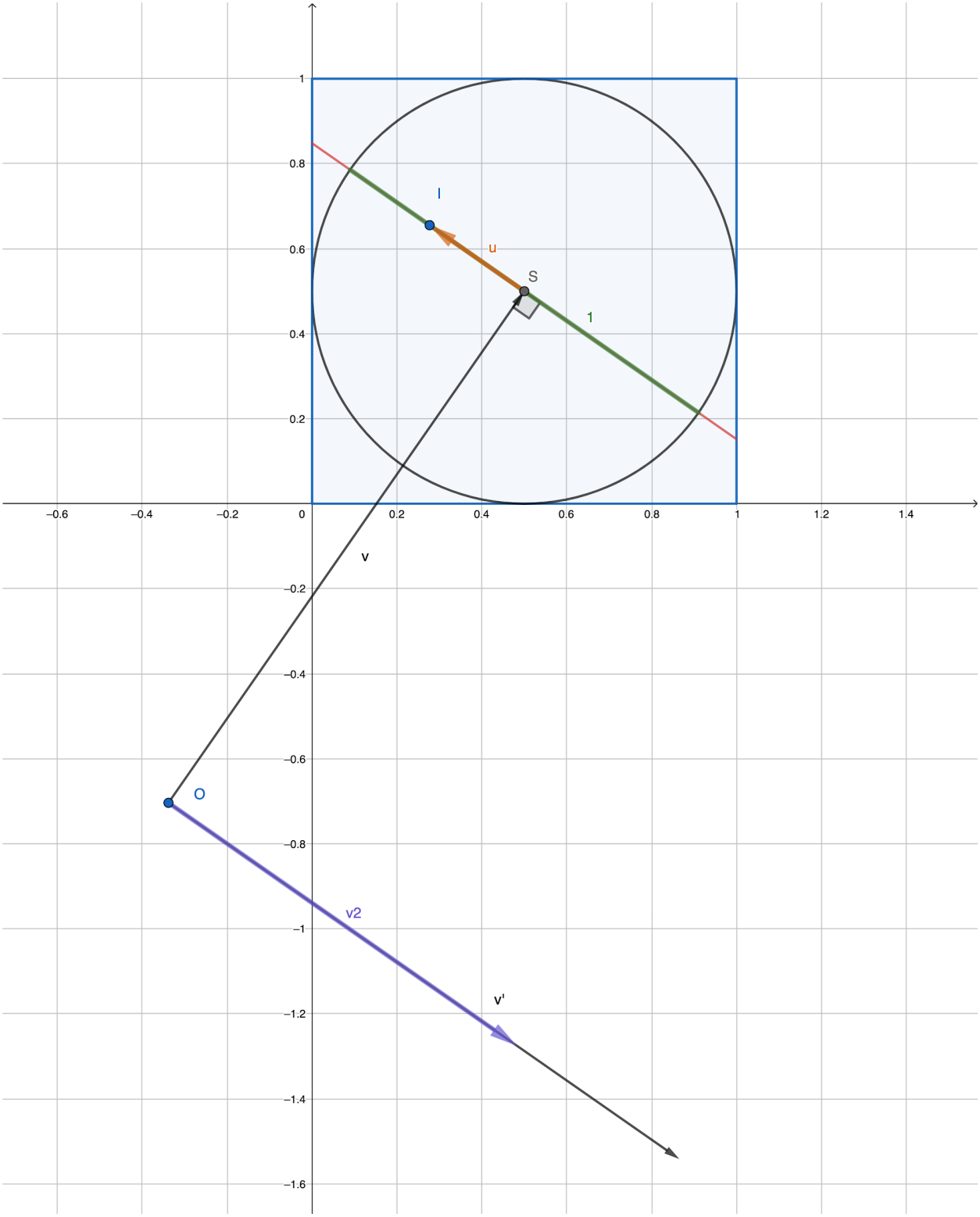


Figure 19 (Lien Geogebra).

Sol et plafond (BONUS)

Plan du sol : *{0, 0, 1, 0}*.

Plan du plafond : *{0, 0, 1, -1}*.

Skybox (BONUS)

La génération d'une skybox se fait avec 6 plans pour former un cube très grand qui englobe toute la scène. Leurs équations sont les mêmes que pour les murs et les sols et plafonds.

Il est aussi possible de la générer avec une sphère mais les calculs seront différents et plus complexes.

Note : Plus la skybox est grande et plus le rendu en sera réaliste.

Raycasting (intersections)

Le raycasting est un algorithme très utilisé en jeux vidéo. Il consiste en un lancer de rayons qui permet de détecter les objets environnants.

Intersection entre une droite et un plan

L'intersection entre une droite et un plan représente en fait le calcul d'intersection entre le rayon de vision et un mur.

Pour trouver le point **I**, intersection d'un plan et d'une droite, il faut qu'il respecte l'équation du plan et l'équation de la droite. Il faut donc lier les 2 équations.

Pour rappel, on a $ax + by + cz + d = 0$ et la *figure 4*.

Avec **x**, **y**, et **z** de l'équation paramétrique de la droite, on peut les intégrer dans l'équation du plan ainsi :

$$a(O_x + u_x t) + b(O_y + u_y t) + c(O_z + u_z t) + d = 0$$
$$\Leftrightarrow aO_x + au_x t + bO_y + bu_y t + cO_z + cu_z t + d = 0$$
$$\Leftrightarrow au_x t + bu_y t + cu_z t + aO_x + bO_y + cO_z + d = 0$$
$$\Leftrightarrow t(au_x + bu_y + cu_z) + aO_x + bO_y + cO_z + d = 0$$
$$\Leftrightarrow t(au_x + bu_y + cu_z) = -(aO_x + bO_y + cO_z + d)$$
$$\Leftrightarrow t = -(aO_x + bO_y + cO_z + d)/(au_x + bu_y + cu_z)$$

En liant ces deux équations, on trouve donc
t = -(***aO$_x$*** + ***bO$_y$*** + ***cO$_z$*** + ***d***) / (***au$_x$*** + ***bu$_y$*** + ***cu$_z$***).

En remplaçant **t** dans l'équation de la droite, on obtient le point suivant qui est le point d'intersection recherché :

$$\begin{pmatrix} O_x + u_x t \\ O_y + u_y t \\ O_z + u_z t \end{pmatrix}$$

Figure 20.

De manière générale, on a ***I*** = ***O*** + ***u*** * ***t***. **t** représente donc le nombre de vecteurs **u** qu'il faut à partir du point **O** pour parvenir au point **I**.

ATTENTION, si le dénominateur de **t** est nul, cela signifie qu'il n'y a pas d'intersection.

Note : Il est possible de précalculer le numérateur de **t** au début de chaque frame pour gagner en performances.

Application sur les rayons et la map

Notre rayon est une demi-droite. Il faut donc nous assurer que le point **I** soit devant nous et non derrière. Pour cela, il faut vérifier que **t** soit supérieur à *0*.

Ensuite, nos murs font une taille de *1*. Il faut donc aussi vérifier que $\mathbf{I}_z$ soit compris entre *0* inclus et *1*.

Matrices de rotation

Les matrices sont des tableaux de nombres. Les matrices dites "de rotation" sont des matrices spécifiques qui permettent la rotation de vecteurs.

Définition

Les matrices sont des tableaux de nombres de taille **n** * **m** (respectivement largeur et hauteur).

Exemple de matrice (3, 3) :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 21.

Note : Cette matrice particulière est une matrice identité. Cela signifie que quoi qu'on multiplie par cette matrice, le résultat sera le même que la matrice d'origine.

Multiplication

Multiplication de matrices

Par la suite, nous utiliserons les matrices **A**, de taille (**A_n**, **A_m**), et **B**, de taille (**B_n**, **B_m**).

Pour multiplier 2 matrices entre elles, il faut que la largeur de la première soit égale à la hauteur de la seconde. Soit ***A_n*** = ***B_m***.

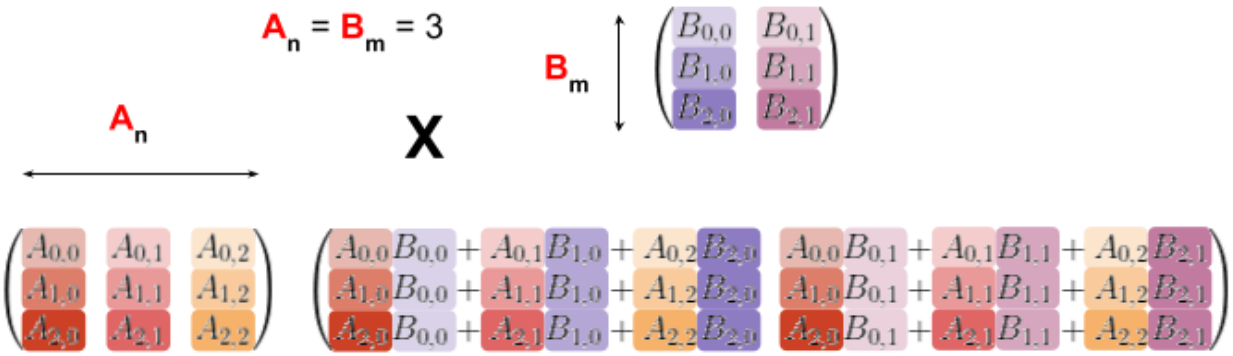


Figure 22.

Dans le cas particulier de cette *figure 22*, pour les matrices **A(3, 3)** et une matrice **B(2, 3)**, la multiplication des 2 donnera une matrice résultante **C(2, 3)**.

De manière générale, pour multiplier **A(**A_n**,** A_m**)** et **B(**B_n**,** B_m**)**, il faut que ***A_n*** = ***B_m*** et le résultat sera une matrice **C(**B_n**,** A_m**)**.

Vous vous rendrez bien compte qu'en général, il peut être possible de multiplier la matrice **A** par la matrice **B** mais l'inverse peut ne pas être possible. Quoi qu'il en soit, le résultat sera différent s'il n'y a pas de matrice identité.

Cas particulier des vecteurs

Un vecteur peut être représenté par une matrice *(1, 3)*. Si l'on multiplie une matrice *(3, 3)* par ce vecteur, nous obtiendrons donc une troisième matrice (1, 3), c'est-à-dire un nouveau vecteur.

Application

En multipliant certaines matrices spécifiques avec des vecteurs, il est possible de faire tourner ces vecteurs.

Présentation de la matrice de rotation 2D

Généralités

Voici la matrice de rotation 2D :

$$\begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix}$$

Figure 23.

θ (*theta*) représente l'angle de rotation (en *radians*).

Exemple

Pour un angle de *90* degrés, soit $\pi/2$ radians, on a la matrice suivante :

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

Figure 24.

Donc pour un vecteur ***(4, 2)*** par exemple, le vecteur résultant sera ***(-2, 4)*** (*voir figure ci-dessous*).

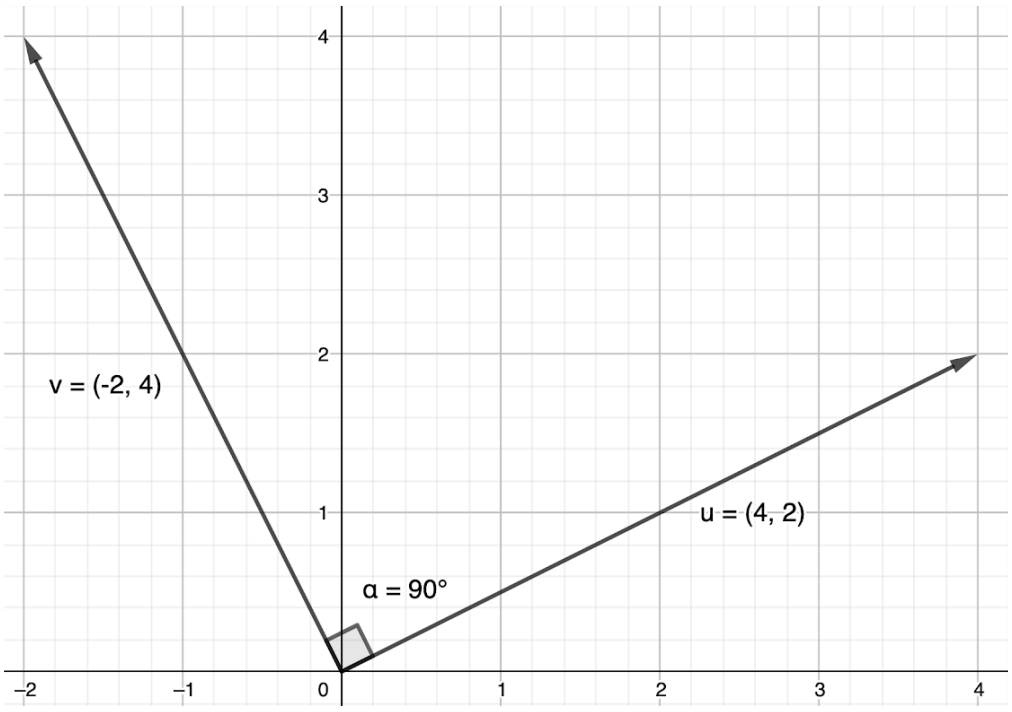


Figure 25.

Les différentes matrices de rotation 3D

En 3D, il y a 3 matrices de rotation différentes : une pour tourner autour de l'axe x, une autre pour l'axe y et une pour l'axe z.

Matrice de rotation x

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{pmatrix}$$

Figure 26.

Matrice de rotation y

$$\begin{pmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{pmatrix}$$

Figure 27.

Matrice de rotation z

$$\begin{pmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 28.

Rotation des rayons

Dans votre algo, il faudra donc créer votre rayon à partir de votre vecteur **u**. En multipliant la **matrice de rotation z** (*axe vertical*)(*figure 28*) par **u**, le rayon pointera dans la bonne direction. Cela vous permettra de vous tourner sur vous-même.

Il est aussi possible d'appliquer la **matrice de rotation x** pour pouvoir regarder en haut et en bas (voir Bonus).

Et si vous êtes à fond, vous pouvez aussi instaurer la **matrice de rotation y** pour tanguer de gauche à droite (voir Bonus).

Algorithme

Cette partie regroupe les différentes parties vues plus haut afin de les agencer dans le bon ordre. A la fin de cette partie vous serez capable de vous déplacer dans votre map.

Débuter

Pour commencer, référez-vous à ce paragraphe.

Il est préférable de commencer par 4 plans instanciés en dur. Pour chaque rayon, il faut calculer l'intersection avec chaque mur ; mettre de la couleur noire par défaut et de la couleur blanche dès qu'une intersection est considérée comme valide.

Rappel : Pour qu'une intersection soit considérée comme valide, il faut que le **t** soit positif et que $\mathbf{I}_z$ soit compris entre *0* inclus et *1*.

Ce début d'algorithme vous permettra déjà d'avoir 4 murs blancs autour de vous. Pour pouvoir tourner sur vous-même, vous n'avez plus qu'à implémenter une matrice de rotation (z) (*voir figure 28*).

Conseil : Tester les valeurs d'intersection dans Geogebra pour vérifier les calculs.

Stockage des murs

Pour penser à l'optimisation dès maintenant, le mieux est de créer 4 tableaux de plans, 1 pour chaque orientation (*N*, *S*, *E* et *W*).

Il y aura donc 2 tableaux de taille ***w*** *+ 1* (*E* et *W*) (*voir figure 12*) et 2 tableaux de taille ***h*** *+ 1* (*N* et *S*) (*voir figure 11*).

Parcours des plans

La manière la plus simple de parcourir les plans est de commencer à l'index de la position du joueur (**O**), puis d'incrémenter ou de décrémenter celui-ci selon la direction du rayon (en regardant le signe de **u**). Ainsi, il suffit de s'arrêter au premier mur et de comparer la distance (**t**) du mur le plus proche sur l'axe des *x* avec celui le plus proche de l'axe des *y*. On pourra donc afficher une couleur/texture différente selon le point cardinal.

Checks dans la map

Après avoir trouvé une intersection avec un plan, il faut regarder si un mur est présent à cet endroit sur la map.

Checks cardinaux

toutes les coordonnées de cases dans la map devront être des nombres entiers naturels.

Nord (N)

Quand on regarde au *Nord*, il faut checker la case ($\mathbf{I}_x$, $\mathbf{I}_y$ - 1).

Sud (S)

Quand on regarde au *Sud*, il faut checker la case ($\mathbf{I}_x$, $\mathbf{I}_y$).

Est (E)

Quand on regarde au *Est*, il faut checker la case ($\mathbf{I}_x$, $\mathbf{I}_y$).

Ouest (W)

Quand on regarde au *Ouest*, il faut checker la case ($\mathbf{I}_x$ - 1, $\mathbf{I}_y$).

Imprécisions

Si on laisse les checks ainsi, des imprécisions feront que des bandes ou points feront leur apparition sur l'affichage. Ce problème vient de l'imprécision des casts en *int* des coordonnées des points d'intersection. Pour pallier ce problème il vaut mieux utiliser l'opposé de la dernière coordonnée (**d**) du plan pour remplacer certaines coordonnées de cases.

Pour l'axe *x* (*W* et *E*), on doit donc remplacer $\mathbf{I}_x$ par **-d**.

Dans le cas de l'axe *y* (*N* et *S*), il faudra donc remplacer $\mathbf{I}_y$ par **-d**.

Conclusion

Voici les cases à checker : N($\mathbf{I}_x$, **-d** - 1), S($\mathbf{I}_x$, **-d**), E(**-d**, $\mathbf{I}_y$) et W(**-d** - 1, $\mathbf{I}_y$).

Textures

*Les **textures** sont les images que nous allons appliquer sur les plans.*

Calcul des textures de murs

Le calcul de texture des murs est ici très simple. Il suffit de prendre la partie décimale du point d'intersection.

Pour l'axe des *x*, les parties décimales à prendre sont celles de $\mathbf{I}_x$ et $\mathbf{I}_z$.

Pour l'axe des *y*, les parties décimales à prendre sont celles de $\mathbf{I}_y$ et $\mathbf{I}_z$.

Les ratios ainsi sélectionnés n'auront plus qu'à être multipliés par respectivement la largeur et la hauteur de la texture pour avoir la coordonnée de la case où aller chercher la couleur dans l'image de la texture.

Textures de sprites

Pour pouvoir calculer les textures, il faut au préalable avoir calculé les sprites (voir paragraphe du calcul des sprites). Ensuite, les ratios à utiliser seront **r** pour la largeur de la texture et la partie décimale de $\mathbf{I}_z$ pour sa hauteur.

Textures de sol et plafond (BONUS)

Le texturing se fait sur les sol et plafond grâce aux ratios des parties décimales de $\mathbf{I}_x$ et $\mathbf{I}_y$ cette fois.

Skybox (BONUS)

Pour texturer la skybox, il suffit de faire un ratio entre la coordonnée d'intersection et la taille de la skybox.

Optimisation des performances

Dans un tel moteur graphique, l'optimisation est très importante afin d'avoir beaucoup de FPS (Frames Per Second) et d'avoir le rendu le plus fluide possible.

Précalcul

Le précalcul, c'est-à-dire de calculer à l'avance certaines parties du moteur, est l'un des moyens les plus efficaces pour gagner en temps de calcul.

Précalcul des rayons

Premièrement, il est très important de calculer les rayons au préalable. Comme indiqué ici, les rayons peuvent être calculés avant la *loop* graphique. On les crée donc avec une orientation *Nord* puis il suffit de tourner le rayon vers notre direction juste avant de l'utiliser dans l'algorithme. On gagne ainsi tout le temps de création durant la *loop*.

Précalcul des plans

Ensuite, on peut utiliser le même procédé pour les plans (sauf sprites). Le résultat sera beaucoup moins impressionnant mais étant donné que les plans sont constants en permanence, il est inutile de les recréer à chaque fois. On peut donc, eux aussi, les créer avant même le début de la *loop*.

Précalcul des sprites

Et enfin, les sprites, qu'il est possible de précalculer au début de chaque *frame* cette fois puisque les équations sont susceptibles de changer à chaque boucle. L'intérêt ici est de ne calculer ces plans qu'une seule fois par *frame* au lieu de ***W*** * ***H*** fois (le nombre de rayons).

Minimisation des calculs

La minimisation des calculs consiste en la suppression de calculs inutiles dans certains cas afin qu'il y ait le moins d'opérations possibles.

Suppression de murs

Dans le cas des murs, il est possible de supprimer tous les plans situés entre 2 lignes/colonnes identiques. En effet, aucun plan ne sera jamais trouvé ici avec le raycasting.

Tri des sprites

Il est intéressant de trier les sprites avec leur distance (norme de **v**) par rapport au joueur (**O**) au moment de la création de leur plan.

Simplification des calculs

Pour chaque calcul d'intersection, il est possible d'enlever des paramètres au calcul. Par exemple, pour calculer un plan en *x*, il sera possible d'enlever tous les paramètres se référant à l'axe des *y* puisqu'ils seront tous a *0*. L'inverse est vrai aussi. De même pour les autres plans formés par le sol, le plafond et donc ceux de la skybox.

Le fait d'enlever ces opérations permettra de gagner un peu de temps, qui cumulé, en fera gagner beaucoup au final.

Choses à savoir

Les différentes catégories ci-dessous sont toutes très consommatrices en termes de temps d'exécution. Il vaut donc mieux les éviter le plus possible.

La division

La division est plus lente à l'exécution que la multiplication. Il est donc préférable d'utiliser cette dernière quand c'est possible. Les différents moyens d'éviter la division sont les suivants :

- Remplacer les divisions connues par des multiplications.
  Exemple : Remplacer le */ 2* par un ** 0.5*.
- Remplacer la division de fraction par la multiplication de l'inverse.
  Exemple : Remplacer *101 / 21 / 2* par *101 / 21 * (1 / 2)* soit *101 / (21 * 2)* soit *101 / 42*.
- Précalculer les divisions.
  Exemple : Si plusieurs calculs effectuent la même division, il est possible de précalculer *1* divisé par le dénominateur. Il suffira ensuite de multiplier par ce nombre au lieu de diviser.

Ce genre de petit détail, qui peut sembler insignifiant, peut faire gagner jusqu'à *10 FPS* s'il est bien utilisé.

Les angles

De même que pour la division, il est très important de stocker les angles en *radians* plutôt qu'en degrés pour éviter d'avoir à faire les conversions.

Rappel : Pour passer un angle de degrés en radians, il faut le multiplier par π */ 180*. Pour l'inverse, il faut multiplier l'angle par *180 /* π. Vous noterez que ce sont ici des divisions inutiles.

Les fonctions mathématiques

Les fonctions mathématiques (*sqrt*, *cos*, *sin* et *tan* principalement), sont elles aussi très gourmandes sur le plan temporel. Il est donc conseillé encore une fois de précalculer un maximum, d'éviter la trigonométrie et le calcul d'angle quand cela est possible (souvent) et de faire des comparaisons sur les carrés plutôt que sur les racines.

Les listes chainees

Le parcours de listes chaînées est assez long dans ce genre de circonstance (liste de sprites par exemple). Il est préférable d'utiliser un tableau pour parvenir directement à l'indice voulu.

BONUS

Ces bonus sont vivement conseillés car ils sont relativement simples à mettre en place, ils vous serviront pour la suite de vos projets et vous feront gagner énormément de FPS pour peu de temps investi.

Multithreading

Certainement la plus grosse optimisation, le *multithreading* consiste à répartir les calculs sur les différents cœurs du PC. Le CPU peut donc atteindre, théoriquement, les *400%* d'utilisation, si le processeur possède 4 cœurs.

Le plus efficace est de répartir chaque pixel et ses calculs dans le nombre de threads correspondant au nombre de cœurs du PC, équitablement (et idéalement "aléatoirement").

HUD

Et oui, le HUD permet d'optimiser ! Tous les pixels qui seront couverts par ce HUD seront autant de pixels ou l'algorithme de raycasting n'aura pas besoin de s'appliquer.

Il est donc très facile de gagner ¼ de l'écran, soit augmenter significativement les performances.

Pénombre

La pénombre qui est facilement installable est celle liée à la distance de vue du joueur. Il est ainsi possible d'assombrir le décor en s'éloignant du joueur. En clair, plus quelque chose est loin de la caméra, plus elle est sombre et moins on la voit.

En plus d'accentuer le réalisme de la scène, cela permet de ne plus calculer tout ce qui se trouve au-delà de la limite où l'on ne voit plus rien et que tout est noir.

Note : Bien penser à normaliser les vecteurs des rayons à la création afin que **t** représente bien la distance avec l'intersection ensuite.

Scale de rendu

Pour optimiser le rendu, le *scale* permet de diviser le nombre de pixels calculés par 4. Il y aura donc au final 4 fois moins de calculs. Le principe est de faire les calculs sur une fenêtre 4 fois plus petite (***W*** */ 2* par ***H*** */ 2*) puis d'agrandir le résultat (les pixels de 1 par 1 deviendront des 2 par 2) jusqu'à la fenêtre de rendu demandée.

Le résultat sera certes plus pixelisé mais ce sera à peine perceptible et surtout, le rendu sera nettement plus fluide.

Note : Il est possible d'augmenter le *scale* au-dessus de 2 pour multiplier encore les performances mais cela est déconseillé car ça affecte directement et sensiblement la qualité du rendu.

Calculs vectoriels

Les vecteurs sont très présents dans ce projet et il est important de maîtriser tous les calculs qu'ils peuvent générer.

*Dans les exemples suivants, nous utiliserons les vecteurs **u**, **v** et **w** et le nombre **k**.*

Addition vectorielle

Il est possible d'appliquer un vecteur a un point (puisqu'un point est aussi un vecteur).

Il est aussi possible d'ajouter 2 vecteurs ensemble.

Dans tous les cas, si w est l'addition vectorielle de **u** et **v**, on a :
w($\mathbf{u}_x + \mathbf{v}_x$, $\mathbf{u}_y + \mathbf{v}_y$, $\mathbf{u}_z + \mathbf{v}_z$).

$$\begin{pmatrix} u_x + v_x \\ u_y + v_y \\ u_z + v_z \end{pmatrix}$$

Figure 29.

Soustraction vectorielle

La soustraction vectorielle vectorielle est identique à l'addition.

w($\mathbf{u}_x$ - $\mathbf{v}_x$, $\mathbf{u}_y$ - $\mathbf{v}_y$, $\mathbf{u}_z$ - $\mathbf{v}_z$) ou **w**($\mathbf{v}_x$ - $\mathbf{u}_x$, $\mathbf{v}_y$ - $\mathbf{u}_y$, $\mathbf{v}_z$ - $\mathbf{u}_z$).

$$\begin{pmatrix} u_x - v_x \\ u_y - v_y \\ u_z - v_z \end{pmatrix} \quad \begin{pmatrix} v_x - u_x \\ v_y - u_y \\ v_z - u_z \end{pmatrix}$$

Figures 30.

Multiplication vectorielle

Multiplication vectorielle (multiplication d'un vecteur par un nombre)

A utiliser si l'on souhaite obtenir le même vecteur avec une longueur différente.

w(u_xk, u_yk, u_zk)

$$\begin{pmatrix} u_x k \\ u_y k \\ u_z k \end{pmatrix}$$

Figure 31.

Produit vectoriel (multiplication de 2 vecteurs)

w($u_y v_z$ - $u_z v_y$, $u_z v_x$ - $u_x v_z$, $u_x v_y$ - $u_y v_x$)

$$\begin{pmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{pmatrix}$$

Figure 32.

Produit scalaire (multiplication de 2 vecteurs)

Le produit scalaire de 2 vecteurs permet de calculer des distances, des directions et des angles. Il est noté **u · v**.

u · v = $u_x v_x$ + $u_y v_y$ + $u_z v_z$

$$u \cdot v = u_x v_x + u_y v_y + u_z v_z$$

Figure 33.

Division vectorielle

w(u_x / k, u_y / k, u_z / k) ou **w(k / u_x, k / u_y, k / u_z)**

Note : Dans le premier cas, il est préférable de multiplier par l'inverse plutôt que de diviser.

$$\begin{pmatrix} \frac{u_x}{k} \\ \frac{u_y}{k} \\ \frac{u_z}{k} \end{pmatrix} \quad \begin{pmatrix} \frac{k}{u_x} \\ \frac{k}{u_y} \\ \frac{k}{u_z} \end{pmatrix}$$

Figures 34.

Norme d'un vecteur

La norme d'un vecteur est sa taille, soit la distance qu'il parcourt dans le repère orthonormé. On la note |**u**| (ou voir figure).

Elle est calculee ainsi : |**u**| = sqrt($\mathbf{u}_x^2 + \mathbf{u}_y^2 + \mathbf{u}_z^2$).

$$\|\vec{u}\| = \sqrt{u_x^2 + u_y^2 + u_z^2}$$

Figure 35.

Normalisation d'un vecteur

Normaliser un vecteur consiste à lui donner une taille de 1.

Vecteur normalisé : **u** = **u** / |**u**| *(voir la norme et la division)*.

$$\frac{u}{\|\vec{u}\|}$$

Figure 36.

Bonus

Collisions et *slide*

Collisions

Certainement le plus **IMPORTANT** des bonus, les collisions sont indispensables. Le calcul le plus simple est de vérifier si la nouvelle position sur laquelle nous allons nous trouver n'est pas sur un mur de la map. Si c'est le cas, il ne faut pas avancer dans cette direction.

Pour un maximum d'efficacité, il faut créer un segment de notre position actuelle à la suivante. Puis il faut regarder si ce segment intersecte une case avec un mur ou un sprite.

Slide

Le *slide* concerne les collisions. Cela consiste à ne pas s'arrêter quand on rencontre un mur mais plutôt d'adapter la direction. On va devoir regarder les collisions sur *x* et *y*. Si seul l'un des 2 entre en collision, on peut garder le vecteur d'avancement du second, ce qui nous permettra de **glisser** sur le mur.

Rotations supplémentaires

Vue verticale

Pour pouvoir regarder en haut et en bas (donc de tous les côtés), il suffit d'appliquer la matrice de rotation *x*.

Tangage

Afin de tanguer (se pencher à gauche et à droite), il faut cette fois intégrer la matrice de rotation *y*.

Note : Pour implémenter les 3 rotations, il faut faire des sauvegardes temporaires.

Textures de sol et plafond

Les informations pour texturer le sol et le plafond sont ici.

Skybox

Les informations pour créer une skybox sont ici.

Multithreading

Les informations pour implémenter le *multithreading* sont ici.

Scale

Les informations pour implémenter le *scale* de rendu sont ici.

Vol

Pour voler, il suffit de désactiver la gravité et le plafond puis on peut modifier notre position à notre guise. Combiné avec les matrices de rotation, il est possible d'aller n'importe où et d'avoir l'angle de vue souhaité.

Note : Penser à mettre les collisions sur la skybox ou à limiter l'environnement pour éviter les *overflow*.

Gravité

Le fait d'implémenter la formule de gravité donne un résultat bien plus réaliste qu'une simple décrémentation linéaire.

Formule

Nous recherchons la distance que nous devons parcourir à chaque *frame*. Il faut calculer de combien descendre à chaque fois.

Voici la formule reliant la distance, avec **v** représentant la **vitesse** (en $m.s^{-1}$), **d** la **distance** (en *m*) et **t** le **temps** (en *s*) :

$$v = d/t$$

On a donc

$$d = v * t$$

Voici la formule qui relie la **vitesse**, avec **Ec** représentant l'**énergie cinétique** (en *N*) et **m** la **masse** (en *kg*).

$$Ec = 1/2mv^2$$

$$\Leftrightarrow \quad Ec/(1/2m) = v^2$$

$$\Leftrightarrow \quad Ec/(m/2) = v^2$$

$$\Leftrightarrow \quad Ec * 2/m = v^2$$

$$\Leftrightarrow \quad v^2 = 2Ec/m$$

$$\Leftrightarrow \quad v = \sqrt{2Ec/m}$$

Voici la formule reliant l'**énergie cinétique**, avec **Em** représentant l'**énergie mécanique** (en *N*) et **Epp** l'**énergie potentielle de pesanteur** (en *N*).

$$Em = Ec + Epp$$

$$\Leftrightarrow \quad Ec = Em - Epp$$

Voici la formule reliant l'**énergie potentielle de pesanteur**, avec **g** représentant la **gravité** (en $m.s^{-2}$), une constante d'accélération et **z** l'**altitude**.

$$Epp = mgz$$

Note : La gravité terrestre est d'environ ***9.81 m.s*$^{-2}$**.

$$Ec = Em - mgz$$

Le frottement de l'air est ici négligeable, on peut donc conclure la relation suivante, avec **Em_0** représentant l'**énergie mécanique** à l'instant 0 (avant que la chute ne commence).

$$Em = Em_0$$

$$\Leftrightarrow \quad Em = Ec_0 + Epp_0$$

$$\Leftrightarrow \quad Em = 1/2m_0{v_0}^2 + m_0g_0z_0$$

Or la **masse** et la **gravité** n'ont pas changé.

$$Em = 1/2m{v_0}^2 + mgz_0$$

La **vitesse initiale** était par contre nulle.

$$Em = 1/2m0^2 + mgz_0$$

$$\Leftrightarrow \quad Em = 1/2m0 + mgz_0$$

$$\Leftrightarrow \quad Em = 0 + mgz_0$$

$$\Leftrightarrow \quad Em = mgz_0$$

$$Ec = mgz_0 - mgz$$

$$\Leftrightarrow \quad Ec = mg(z_0 - z)$$

$$v = \sqrt{2(mg(z_0 - z))/m}$$

$$\Leftrightarrow \quad v = \sqrt{2mg(z_0 - z)/m}$$

$$\Leftrightarrow \quad v = \sqrt{2g(z_0 - z)}$$

$$d = \sqrt{2g(z_0 - z)} * t$$

t représente le temps d'exécution d'une *frame*. Avec **FPS**, le **nombre de *frames* par seconde**, on a la relation suivante :

$$t = 1/FPS$$

$$d = \sqrt{2g(z_0 - z)} * 1/FPS$$

$$\Leftrightarrow \quad d = \frac{\sqrt{2g(z_0 - z)}}{FPS}$$

donc ***d*** = ***sqrt***(2***g***(***z_0*** - ***z***)) / ***FPS***.

Ici, **g** représente la gravité. Vous pouvez la mettre à *9.81* (gravité terrestre) mais vous pouvez tout aussi bien l'adapter à vos besoins.

z_0 représente notre point culminant avant de retomber.

z représente notre altitude actuelle (**O_z**).

Note : Il est possible de calculer la distance différemment via les mêmes formules. Il est possible de la déterminer grâce à la vitesse précédente plutôt qu'avec le point culminant.

Note : Et pour aller encore plus loin, n'hésitez pas à aller explorer la mécanique newtonienne.

Forces et sauts

Pour gérer les sauts et les retombées, c'est la même chose. Le mieux est d'interpréter les forces en présence comme des vecteurs. La gravité est donc une force qui s'oppose à celle du saut. Quand le vecteur de gravité devient plus grand que celui du saut, les 2 s'annulent puis la gravité prend le dessus et on commence à retomber.

Raytracing

Le raytracing est sans doute le bonus qui rendra votre projet encore plus beau et réaliste, encore plus que toutes les autres techniques. Malheureusement, c'est aussi lui qui vous prendra le plus de ressources.

Il consiste en un lancer de rayons supplémentaires de lumière.

Le principe est, pour chaque point d'intersection, de calculer de nouveaux rayons en provenance de sources de lumière préétablies.

Pénombre

Les informations concernant la pénombre sont ici.

Transparence

Le bonus de transparence est très intéressant et facile à mettre en place. Il est notamment indispensable pour les sprites. Quel que soit l'objet, le but est de laisser passer le rayon au travers de cet objet si la texture est transparente. S'il n'y a qu'une transparence partielle, il faut mélanger les couleurs de la texture avec les couleurs des intersections suivantes (plus loin sur le rayon).

HUD

Les informations concernant le HUD sont ici.

Son

Pour la gestion du son, la commande la plus simple a utiliser est surement Afplay.

Libre à vous de vous amuser pour plonger le joueur dans votre jeu avec des sons d'ambiances et d'autres spécifiques aux actions que vous réalisez.

Téléportation

La téléportation est l'un des bonus les plus compliqués si vous utilisez des portails. Sinon, c'est très simple, il suffit de déplacer le point **O**, la position du joueur.

Note : Si vous utilisez les portails, il faut faire une recursive pour passer au travers eux (sans oublier la pénombre de la distance pour éviter les boucles infinies). Vous pouvez utiliser l'intersection de la droite du rayon actuel avec la droite normale au plan passant par le centre du portail pour créer le nouveau rayon et ainsi continuer les intersections.

Gameplay

Ce bonus est à votre entière discrétion : éclatez-vous ! A vous de rendre le jeu le plus fun et réaliste possible.

Et bien d'autres encore...

Bien d'autres bonus (sûrement moins mathématiques) sont autant réalisables, laissez parler votre **créativité** !

Si vous êtes fier de ce que vous avez fait, n'hésitez pas à envoyer votre code à tuteurs@42lyon.fr. Nous serons fier de discuter du code !

Lexique

a : Première coordonnée de l'équation de plan

A : Première matrice

b : Seconde coordonnée de l'équation de plan

B : Seconde matrice

c : Troisième coordonnée de l'équation de plan

C : Troisième matrice (résultante)

d : Quatrième et dernière coordonnée de l'équation de plan / Distance

D : Distance sur la vision du mur virtuel

Em : Énergie mécanique

Em_0 : Énergie mécanique initiale

Ec : Énergie cinétique

Ec_0 : Énergie cinétique initiale

Epp : Energie potentielle de pesanteur

Epp_0 : Energie potentielle de pesanteur initiale

FOV : *Field Of View* (Champ de vision horizontal)

FOV_H : *Field Of View* (Champ de vision vertical)

FPS : *Frames Per Second* (*Frames* par seconde)

g : Gravité

h : Hauteur de la map

H : Hauteur de la fenêtre

i : Incrément des *x*

I : Point d'intersection (Raycasting)

j : Incrément des *y*

m : Hauteur d'une matrice / Masse

n : Largeur d'une matrice

O : Origine (Position du joueur)

P : Projeté du point d'intersection sur l'axe vertical du sprite

r : Ratio pour le calcul des sprites

R_H : Ratio de vision horizontal

R_V : Ratio de vision vertical

S : Centre du Sprite

t : Coordonnée de l'équation paramétrique de droite / Temps

u : Vecteur allant du projeté au point d'intersection

v : Vecteur allant de la caméra au centre du sprite / Vitesse

v' : Vecteur **v** tourné de *90* degrés

v2 : Vecteur **v'** normalisé

w : Largeur de la map

W : Largeur de la fenêtre

x : Première coordonnée d'un vecteur

y : Seconde coordonnée d'un vecteur

z : Troisième et dernière coordonnée d'un vecteur / Altitude

z_0 : Altitude initiale (Point culminant du saut)

θ : Angle de rotation

Quentin PUPIER (Qpupier)