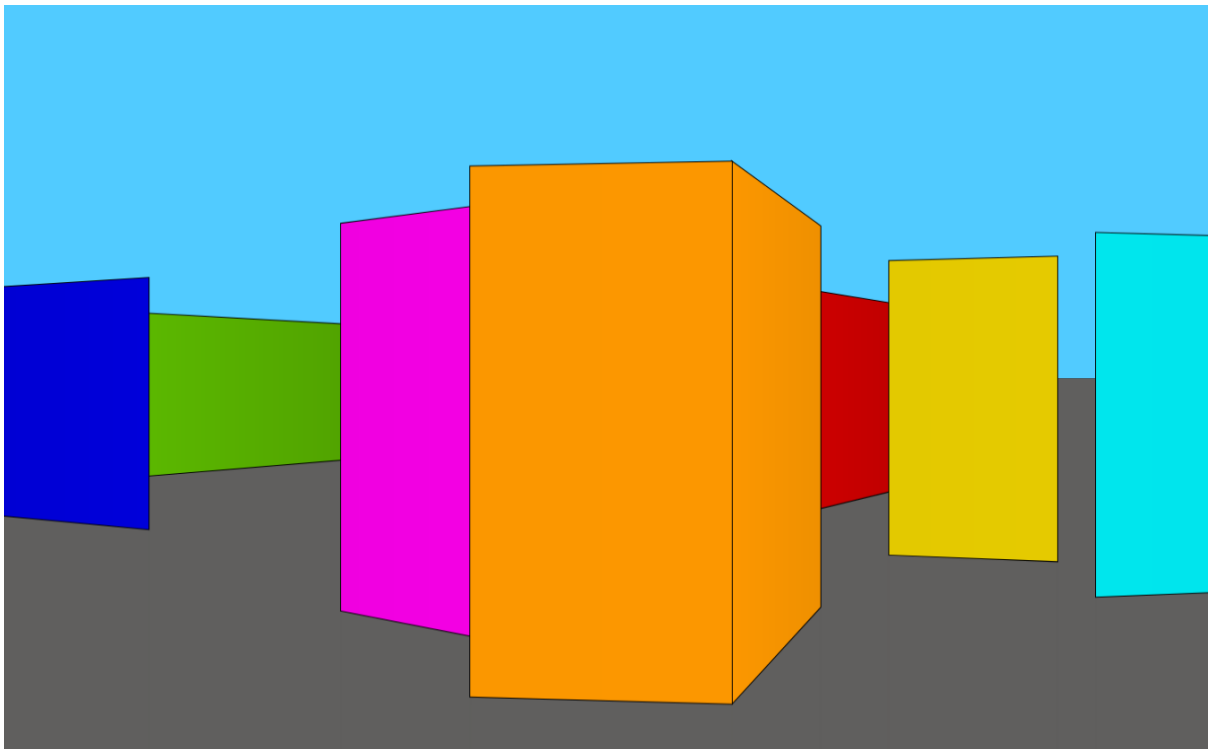


# MAPR

Création d'un algorithme de rendu d'images en 3 dimensions

**Marcus Hamelink**



Un projet d'abord présenté comme travail de maturité  
puis développé pour le Prix Informatique

Collège de Candolle  
24 mars 2022

# Table des matières

---

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Informations sur le projet</b>	<b>3</b>
2.1	Généralités . . . . .	3
2.2	Instructions pour l'utilisation de l'application . . . . .	3
<b>3</b>	<b>La méthode couramment utilisée de Raycasting</b>	<b>5</b>
<b>4</b>	<b>Les principes de ma nouvelle méthode</b>	<b>7</b>
4.1	Le concept derrière la méthode . . . . .	7
4.2	Détecter les obstacles dans le champ de vision . . . . .	7
4.3	Arranger les données des vecteurs pour chaque obstacle . . . . .	8
<b>5</b>	<b>Implémentation de la méthode</b>	<b>10</b>
5.1	Calculs de projection du 2D au 3D . . . . .	10
5.2	Calculer la taille relative d'un obstacle dans le champ de vision . . . . .	11
5.3	Tri des obstacles . . . . .	12
5.3.1	Créer les graphes . . . . .	13
5.3.2	Comparer tous les obstacles entre eux . . . . .	15
5.3.3	Interpréter le graphe . . . . .	15
5.4	Dessins des obstacles . . . . .	16
<b>6</b>	<b>Conclusion</b>	<b>18</b>
<b>7</b>	<b>Bilan personnel</b>	<b>19</b>
<b>8</b>	<b>Annexe</b>	<b>20</b>
8.1	a) Trouver l'intersection de deux segments . . . . .	20
8.2	b) Trouver si deux vecteurs sont orientés dans le sens des aiguilles d'une montre . . . . .	21
<b>9</b>	<b>Bibliographie</b>	<b>23</b>
9.1	Images . . . . .	23
9.2	Sources . . . . .	23

# 1 Introduction

---

Après avoir appris à programmer en Python en mars 2020 au début du confinement, je suis devenu fasciné par tout ce qui est informatique. Je n'ai pas arrêté d'apprendre des nouvelles notions de programmation et durant l'été j'ai commencé à apprendre Javascript pour coder des sites web. J'ai été notamment intéressé par la création de jeux.

Au début de ma troisième année du collège, j'ai découvert le Raycasting, qui est une *technique de calcul d'images de synthèse 3D*<sup>1</sup>. C'est ce qui a rendu possible les premiers jeux vidéo en 3D comme Doom ou Wolfenstein 3D au début des années 1990. Fondée sur un espace en 2D et non pas sur un monde 3D, la méthode permet de simuler une perspective en 3D sans les calculs complexes de la troisième dimension. J'expliquerai de manière plus détaillée comment cette technique fonctionne dans le développement.

J'ai alors commencé à apprendre comment implémenter cette méthode moi-même en suivant des guides sur internet et c'est alors que j'ai décidé d'en faire mon travail de Maturité. Initialement, mon plan était de créer un moteur de jeu avec un grand nombre de fonctionnalités qui auraient permis à quiconque de dessiner des environnements sur un plan et de les explorer en perspective. Il y aurait également eu un accès facile à mon code pour programmer des nouveaux jeux avec mes bases.

Peu de temps après, je me suis rendu compte que la méthode que j'utilisais ne me plaisait pas pour des raisons que je détaillerai dans mon développement. J'ai donc décidé de chercher à créer ma propre technique suivant les principes du Raycasting. Les notions de géométrie vectorielle que j'ai apprises lors de mes cours de maths en troisième année du collège m'ont énormément aidé pour appliquer les mathématiques dans mon code.

Après avoir rendu ce projet de Maturité, j'ai continué à travailler dessus dans mon temps libre. Ce document présente les explications originales avec des mentions des fonctionnalités que j'ai améliorées ou ajoutées.

Dans cette présentation, je détaillerai d'abord comment accéder à une démo du projet final ainsi qu'un guide de son utilisation. J'expliquerai ensuite les principes des méthodes actuelles du Raycasting. Enfin, je décrirai les nouvelles fonctionnalités qui composent ma méthode.

---

<sup>1</sup>Raycasting. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 22 juillet 2021 à 22 :17. [Consulté le 2 octobre 2021]. Disponible à l'adresse : <https://fr.wikipedia.org/wiki/Raycasting>

## 2 Informations sur le projet

### 2.1 Généralités

J'ai écrit le code pour ce projet avec *VSCode* qui est un éditeur de texte pour programmer. Ce code peut être exécuté en ouvrant le fichier *index.html* avec n'importe quel moteur de recherche (*Chrome, Firefox, Safari, etc...*).

Ceci est possible car le langage de programmation avec lequel j'ai écrit mon projet est le *Javascript*. En effet, ce choix était motivé par le fait que je souhaitais que la démo soit accessible à n'importe qui, peu importe l'appareil électronique utilisé.

Ainsi, j'ai dû coder une page web en *HTML* pour créer l'interface qui permet de personnaliser certaines options. J'ai également fait en sorte que l'emplacement des éléments soit adapté à la taille de l'écran pour offrir une meilleure expérience de jeu grâce à un fichier *CSS*. Ce fichier permet aussi de personnaliser la couleur et la formes des éléments sur la page comme les boutons ou les curseurs.

Pour se rendre compte de l'importance du *CSS*, voici une comparaison de l'interface pour ordinateur avec et sans style appliqué :



Page du projet avec style appliqué<sup>2</sup>



Page du projet sans style

### 2.2 Instructions pour l'utilisation de l'application

La page de mon projet est divisée en trois parties :

- Le plan à vue d'oiseau du niveau
- La projection 3D depuis la vue en première personne du joueur
- La section paramètres défilable contenant toutes les personnalisations et commandes nécessaires au programme

Sur un appareil avec un clavier, le joueur peut être déplacé avec les touches W, A, S et D. Pour orienter le regard du joueur de gauche à droite ou de haut en bas, les touches fléchées sont utilisées. Sur un écran tactile où il n'y a pas de clavier, des touches virtuelles sont dessinées sur la projection 3D.

Des obstacles peuvent être dessinés sur le plan à vue d'oiseau en cliquant et en faisant glisser la souris dessus (ou le doigt sur les écrans tactiles). En maintenant la touche *Shift* ou la touche *Majuscule* lorsque vous dessinez un obstacle, le curseur va se coller automatiquement à l'extrémité de l'obstacle le plus proche. Cette fonctionnalité existe aussi sur les écrans tactiles mais elle se fait automatiquement lorsque l'extrémité est déjà placée.

---

<sup>2</sup>Démonstration du projet hébergé sur <https://mapr.me/>

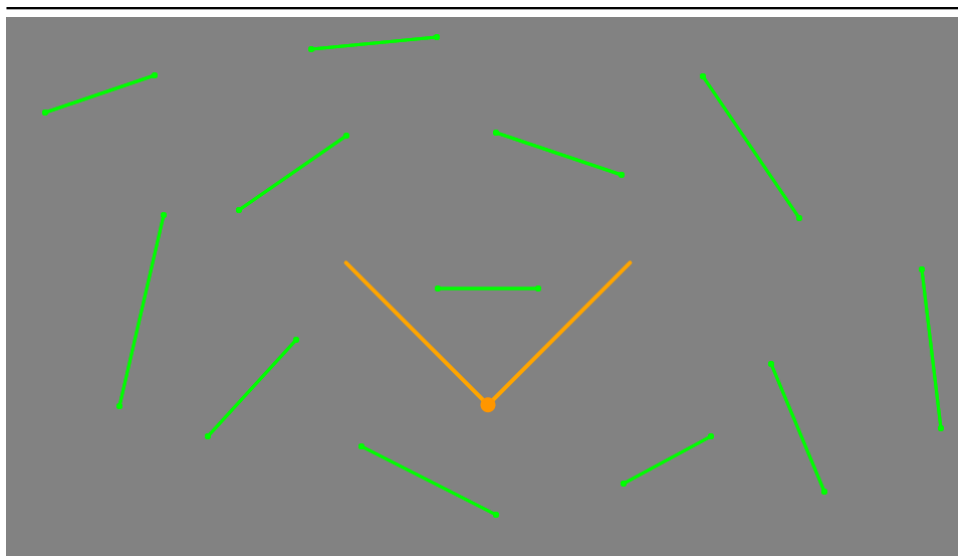
La section paramètres contient tout ce qu'il faut pour générer des obstacles de tous les types ainsi que d'autres petites configurations. Néanmoins, une sélection rapide de différents plans déjà créés sont disponibles au début de la section. Ces plans ont été dessinés pour montrer les capacités de mon programme. Si cette section est trop petite sur la page, un bouton au coin en bas à droite permet de l'élargir.

### 3 La méthode couramment utilisée de Raycasting

---

J'ai d'abord découvert le Raycasting grâce à une vidéo par Daniel Shiffman, un créateur sur YouTube qui tient la chaîne The Coding Train, où dans une de ses séries, Coding challenge, il réalise des petits projets dans le but d'éduquer ses auditeurs. Sa vidéo sur le Raycasting<sup>3</sup> a été pour moi une bonne introduction au sujet. J'ai passé de nombreuses heures à comprendre les principes afin de pouvoir les appliquer dans mes propres projets. J'ai décidé d'en faire mon travail de Maturité car j'étais curieux de savoir comment je pouvais créer mes propres jeux avec.

La méthode la plus courante et celle que j'ai codée en premier est la suivante : On part d'un plan en 2 dimensions où le joueur est représenté par un point ainsi que deux segments indiquant l'étendue de son champ de vision (utile pour la suite). Autour de lui sont placés des traits qui représentent des obstacles. Le joueur peut se déplacer et de s'orienter librement sur le plan.



Plan 2D du joueur entouré d'obstacles<sup>4</sup>

---

Dans mon programme, tous les obstacles ainsi que le joueur sont programmés sous forme d'objets. C'est-à-dire que chacun d'eux possède des propriétés qui leur correspondent individuellement. (Comme par exemple les coordonnées x, y sur le plan, l'orientation, la couleur, etc...)

Ensuite, on génère des rayons provenant du joueur entre les extrémités de son champ de vision. Lorsque les rayons intersectent avec un obstacle, on calcule le point d'intersection où se joignent les deux segments<sup>5</sup>.

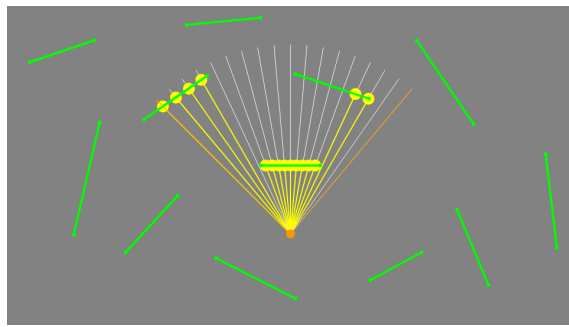
Le plan 2D n'est qu'une représentation des calculs réalisés pour le rendu 3D. Ainsi, on peut se balader sur le plan comme si on se trouvait à la place du point et on percevait tous les obstacles de nos propres yeux sur le sol.

---

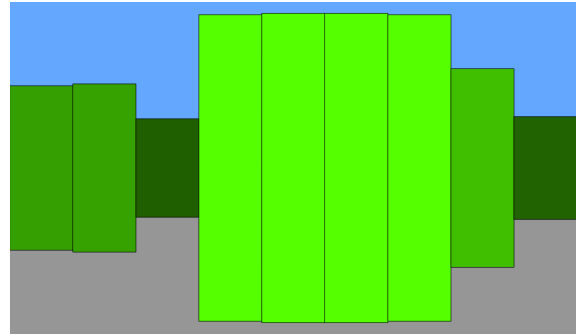
<sup>3</sup>SHIFFMAN Daniel, 2019 Coding Challenge #146 : Rendering Raycasting [enregistrement vidéo]. YouTube [en ligne]. Disponible à l'adresse : <https://youtu.be/vYgIKn7iDH8>.

<sup>4</sup>Plan 2D du joueur entouré d'obstacles ; image générée à partir de mon code, HAMELINK Marcus, 2021.

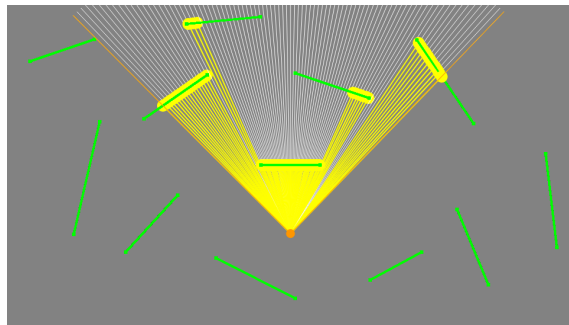
<sup>5</sup>Le développement de la formule mathématique de l'intersection se trouve en Annexe a)



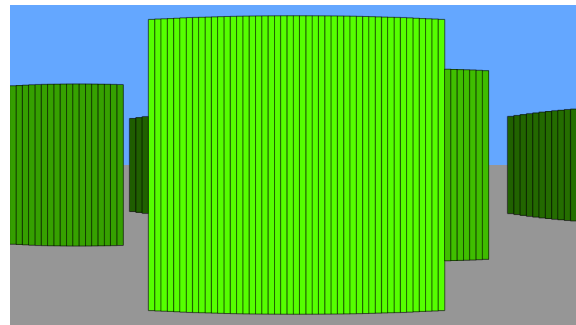
Plan 2D avec les points d'intersection calculés<sup>6</sup>



Perspective 3D calculée à partir du plan 2D



Plan 2D avec plus d'intersections<sup>7</sup>



Même perspective avec plus de détails

Sur les images ci-dessus, on peut observer une représentation (à gauche) du plan 2D et (à droite) la perspective 3D.

L'écran de la perspective 3D est divisé horizontalement par le nombre de rayons générés. Ainsi, pour chacun de ces rayons, on lui associe une partie de cet écran 3D et on utilise la distance entre le joueur et l'intersection avec un obstacle pour dessiner un rectangle à hauteur proportionnelle à la distance. C'est-à-dire, si la distance est plus grande, l'intersection se trouve plus loin du joueur donc la hauteur du rectangle sera plus petite et vice versa si la distance est plus petite. Ceci permet de simuler la perspective que l'on aperçoit de nos propres yeux.

Les images ci-dessus sont une démonstration de ce concept avec d'un côté ce qui se passe sur le plan 2D et de l'autre ce qu'on peut apercevoir après les calculs d'images. Pour calculer la hauteur du rectangle, il existe une autre manière que j'applique dans ma nouvelle méthode pour ces calculs d'images 3D.

Il existe également d'autres méthodes pour coder un moteur de rendu avec le Raycasting, comme par exemple à partir d'une grille pour présenter les obstacles comme des cubes<sup>8</sup>, le principe basique est le même. C'est à partir de ce principe expliqué précédemment que j'ai créé une nouvelle méthode qui me plaisait mieux.

<sup>6</sup>Plan 2D avec les points d'intersections de 18 rayons calculés ; image générée à partir de mon code, HAMELINK Marcus, 2021

<sup>7</sup>Plan 2D avec les points d'intersections de 90 rayons calculés ; image générée à partir de mon code, HAMELINK Marcus, 2021

<sup>8</sup>Il existe un guide complet pour réaliser cette méthode, Raycasting, Lode Vandevenne, 2004-2020, Disponible à l'adresse : <https://lodev.org/cgtutor/raycasting.html>

## 4 Les principes de ma nouvelle méthode

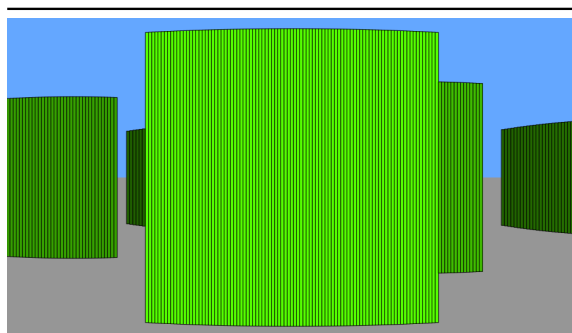
---

### 4.1 Le concept derrière la méthode

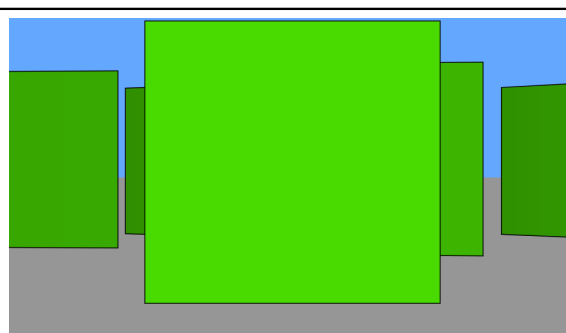
Ce que je n'aimais pas avec la méthode initiale c'était l'aspect pixelisé du rendu 3D. Lorsque le joueur se dirige dans une autre direction, on voit un changement dans les rectangles qui sont dessinés, cassant cette perspective que l'on perçoit de nos propres yeux.

En revanche, plus on utilise de rayons, plus il y a de rectangles qui sont dessinés. Seulement, une augmentation de calculs par image entraîne rapidement un ralentissement de l'expérience de jeu et limite les possibilités, surtout si en plus du nombre de rayons, on augmente également le nombre d'obstacles. De plus, on a moins de liberté si on veut, par exemple, dessiner un contour autour des obstacles (étant donné qu'il n'y a pas d'unique forme mais une multitude de rectangles qui ont chacun leur propre contour).

Ainsi, je me suis dit que je pouvais utiliser des polygones pour dessiner ces obstacles car peu importe l'orientation des obstacles, la perspective qui en résulte sera toujours un polygone à quatre coins. Le problème, c'est qu'avec la méthode précédente, on ne sait pas si deux rayons intersectent avec un même obstacle. On aurait éventuellement pu le détecter mais un autre problème se pose lorsque le coin d'un obstacle se trouve entre deux rayons. Il suffit que le joueur se déplace un tout petit peu et le coin de cet obstacle dans la perspective 3D ne sera pas au même endroit.



Perspective 3D de l'ancienne méthode 180 rayons calculés



Perspective 3D de ma nouvelle méthode 10 rayons calculés

Ce sont ces petits détails qui m'ont fait comprendre que si je voulais changer de méthode, je devais supprimer cette génération de tous ces rayons. En gardant cette idée de dessiner des polygones, on peut en déduire une nouvelle manière de détecter les obstacles.

Dans les prochaines sections, je vais présenter les différentes fonctions utilisées par mon programme. Ces fonctions sont exécutées sur chacun des obstacles.

### 4.2 Détecter les obstacles dans le champ de vision

Pour commencer, il faut regrouper tous les obstacles qui se situent dans le champ de vision du joueur. La fonction *isInsideFOV()* (qui signifie "se trouvant à l'intérieur du champ de vision" en anglais) attribue d'abord deux nouvelles variables *p* et *h* qui sont deux vecteurs qui relient respectivement la position du joueur à chacune des extrémités de l'obstacle. (*p* qui signifie position et qui correspond au vecteur du premier point de l'obstacle et *h* qui signifie header, aussi dit l'entête de l'obstacle)



Ensuite, on effectue la vérification par l'utilisation répétée de *isClockwiseOrder()*<sup>9</sup>, une fonction qui prend deux vecteurs et qui retourne vrai si le premier est orienté dans le sens des aiguilles d'une montre par rapport au deuxième.

Dans mon programme, si une des extrémités de l'obstacle est à la fois à la droite du vecteur de champ de vision de gauche et aussi à la gauche du vecteur de champ de vision de droite, alors la fonction va retourner vrai. On vérifie ceci pour les deux extrémités. De plus, si le joueur observe un obstacle de près et que les extrémités sortent du champ de vision, la fonction vérifie également s'il existe une intersection<sup>10</sup> entre l'obstacle et chacun des vecteurs de champ de vision.

Cette fonction permet d'effectuer les prochains calculs seulement sur les obstacles qui comptent.

```
function isInsideFOV() {
// Creating vector going from player to wall's first vertex
  this.p = {
    'x': this.pos.x - player.pos.x,
    'y': this.pos.y - player.pos.y
  };

// Creating vector going from player to wall's first vertex
  this.h = {
    'x': (this.pos.x + this.dir.x) - player.pos.x,
    'y': (this.pos.y + this.dir.y) - player.pos.y
  };

  this.p.dist = Math.sqrt((this.p.x)**2 + (this.p.y)**2);
  this.h.dist = Math.sqrt((this.h.x)**2 + (this.h.y)**2);

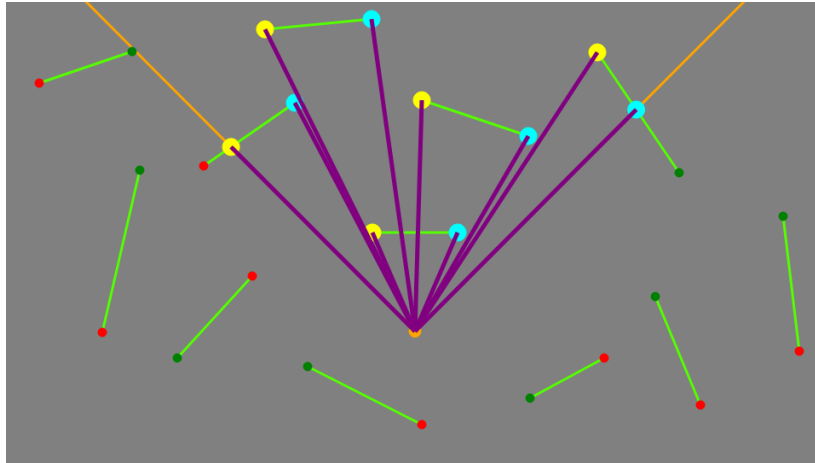
  if ((isIntersectionFovW(player.fov.v1, this) ||
    isIntersectionFovW(player.fov.v2, this)
  ) ||
    (isClockwiseOrder(player.fov.v1.dir, this.p) &&
      !isClockwiseOrder(player.fov.v2.dir, this.p) &&
      isClockwiseOrder(player.fov.v1.dir, this.h) &&
      !isClockwiseOrder(player.fov.v2.dir, this.h) &&
      (this.p.dist <= player.fov.v1.dir.length*2 &&
        this.p.dist <= player.fov.v2.dir.length*2) &&
      (this.h.dist <= player.fov.v1.dir.length*2 &&
        this.h.dist <= player.fov.v2.dir.length*2)
    )
  ) return true;
}
```

### 4.3 Arranger les données des vecteurs pour chaque obstacle

La fonction *processFOV()* qui est exécutée seulement après avoir vérifié que l'obstacle se trouve dans le champ de vision, modifie les vecteurs p et h pour que les données soient plus faciles à traiter par la suite.

<sup>9</sup>Voir point b) de l'Annexe pour une explication détaillée de cette fonction

<sup>10</sup>La fonction *isIntersection()* dont l'explication se trouve dans l'Annexe a)



Visualisation des obstacles intersectant avec les vecteurs de champ de vision <sup>11</sup>

D'abord on s'assure que  $p$  soit bien à la gauche de  $h$  ; dans le cas contraire, on inverse les données. Ensuite, on vérifie si l'obstacle croise un des vecteurs du champ de vision. Si c'est le cas, on définit à nouveau un des vecteurs pour qu'on n'ait seulement la tranche de l'obstacle qui se trouve à l'intérieur du champ de vision.

```
function processFOV() {
    // make sure v1 is always to the left of v2
    if (!isClockwiseOrder(this.p, this.h)) {
        let vtemp = this.p;
        this.p = this.h;
        this.h = vtemp;
    }

    // redefine wall if it intersects with LEFT fov ray
    if (!isClockwiseOrder(player.fov.v1.dir, this.p) &&
        isClockwiseOrder(player.fov.v1.dir, this.h)) {
        this.p = intersectionFovW(player.fov.v1, this);
        this.p.x -= player.pos.x;
        this.p.y -= player.pos.y;
    }

    // redefine wall if it intersects with RIGHT fov ray
    if (!isClockwiseOrder(player.fov.v2.dir, this.p) &&
        isClockwiseOrder(player.fov.v2.dir, this.h)) {
        this.h = intersectionFovW(player.fov.v2, this);
        this.h.x -= player.pos.x;
        this.h.y -= player.pos.y;
    }
    return true;
}
```

<sup>11</sup>Visualisation des obstacles intersectant avec les vecteurs de champ de vision par des points (jaune pour l'extrémité gauche de l'obstacle, et bleu pour l'extrémité droite) ; image générée à partir de mon code, HAMELINK Marcus, 2021

## 5 Implémentation de la méthode

### 5.1 Calculs de projection du 2D au 3D

Maintenant qu'on a toutes les données nécessaires, on peut enfin réaliser les calculs qui conduiront à une perspective 3D. On va prendre les vecteurs *p* et *h* ainsi que le vecteur directeur du joueur (celui qui indique son orientation) pour ensuite les normaliser afin de pouvoir travailler avec leur valeurs unitaires et ne plus devoir se soucier de leur longueur.

On appliquera ensuite la formule de l'angle entre deux vecteurs. L'angle *v1xangle* correspond à l'angle entre le vecteur *p* et le vecteur directeur du joueur. L'angle *v2xangle* correspond à celui entre *p* et *h*. Les angles sont calculés ainsi pour qu'on ait des valeurs allant du négatif au positif. On a un angle relatif entre *p* et *h* dont l'angle *p* qui est relatif au vecteur directeur du joueur. L'angle final de *h* n'est qu'une addition entre l'angle de *p* et l'angle relatif entre les deux.

Finalement, on définit les variables pour chacun des coins du polygone qui sera généré pour dessiner l'obstacle. On calcule *x1* et *x2* avec une proportion entre l'angle calculé précédemment et sa place relative sur l'écran et *h1* et *h2* sont calculés grâce à la fonction *calculateHeight()* dont je vais parler dans la prochaine section.

```
function calculate3D() {
    const fovamount = player.fov.xamount;

    this.p = vectorNormalize(this.p);
    this.h = vectorNormalize(this.h);
    const dir = vectorNormalize(player.dir,
        Math.sqrt((player.dir.y) ** 2 + (player.dir.x) ** 2));
    let v1xangle = Math.acos(vectorDotProduct(this.p, dir));
    let v2xangle = Math.acos(vectorDotProduct(this.h, this.p));

    // correct sign depending on side of v1/v2
    if (!isClockwiseOrder(dir, this.p)) v1xangle = -v1xangle;
    if (!isClockwiseOrder(this.p, this.h)) v2xangle = -v2xangle;

    v2xangle = v2xangle + v1xangle; // make v2 relative to v1

    this.p.dist *= Math.cos(v1xangle);
    this.x1 = degrees(v1xangle) * canvas.width / fovamount;
    this.h1 = this.calculateHeight(this.p);

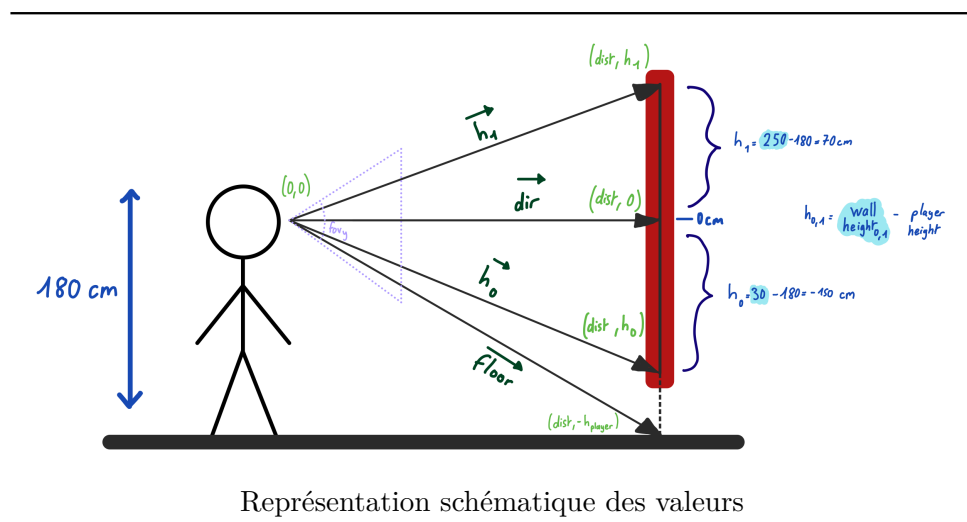
    this.h.dist *= Math.cos(v2xangle);
    this.x2 = degrees(v2xangle) * canvas.width / fovamount;
    this.h2 = this.calculateHeight(this.h);

    if (this.h1 > 10000) this.h1 = 10000;
    if (this.h2 > 10000) this.h2 = 10000;
}
```

## 5.2 Calculer la taille relative d'un obstacle dans le champ de vision

Après avoir calculé la distance entre le joueur et un point sur un obstacle, on peut alors déterminer la hauteur que cette tranche d'obstacle prendra sur la perspective 3D.

Les valeurs dont on a besoin sont :  $dist$  la distance calculée précédemment grâce à la fonction `intersection()`,  $h_0$  la différence entre le bas de l'obstacle et la hauteur du joueur,  $h_1$  la différence entre le haut de l'obstacle et la hauteur du joueur,  $floor$  la différence entre la hauteur du joueur et le sol,  $canvas_{height}$  la hauteur en pixels de l'écran en perspective 3D et  $FOV_y$  la valeur verticale maximale du champ de vision. Pour mieux illustrer ces points, on les représentera sous forme de vecteurs mais on n'a seulement besoin des valeurs verticales car la valeur horizontale est la même. La hauteur du joueur ainsi que celle du haut et le bas de chaque obstacle sont prédéfinies et peuvent être modifiées librement.



Ainsi, on peut calculer la proportion entre la hauteur de ces points et l'hauteur relative à l'écran en perspective 3D grâce aux formules suivantes :

$$floor_{height} = \frac{floor_y \cdot canvas_{height}}{\tan(FOV_y) \cdot dist} \quad h_{0height} = \frac{h_{0y} \cdot canvas_{height}}{\tan(FOV_y) \cdot dist} \quad h_{1height} = \frac{h_{1y} \cdot canvas_{height}}{\tan(FOV_y) \cdot dist}$$

Ensuite, on peut utiliser une proportion entre l'angle et la hauteur de l'écran de la perspective 3D. La fonction `calculateHeight()` retourne enfin ces valeurs calculées par cette proportion.

Au moment de rendre ce projet comme travail de Maturité, cette partie de mon programme était plus complexe que nécessaire car j'ai essayé de trouver des mesures grâce aux angles. Ceci donnait une projection qui paraissait à première vue bonne mais on se rendait vite compte que quelque chose était faux. Finalement, la proportionnalité entre les différentes hauteurs donne un calcul plus simple et des hauteurs qui ressemblent à ce qu'on percevrait de nos propres yeux

```
function calculateHeight(v) { // simple proportion calculations
  const h0 = vectorCreate(v.dist, this.height0 - player.height);
  const h1 = vectorCreate(v.dist, this.height1 - player.height);
  const floor = vectorCreate(v.dist, -player.height);
  return {
    'floor': (floor.y * canvas.height) / (Math.tan(radians(player.fov.yamount)) * v.dist,
```

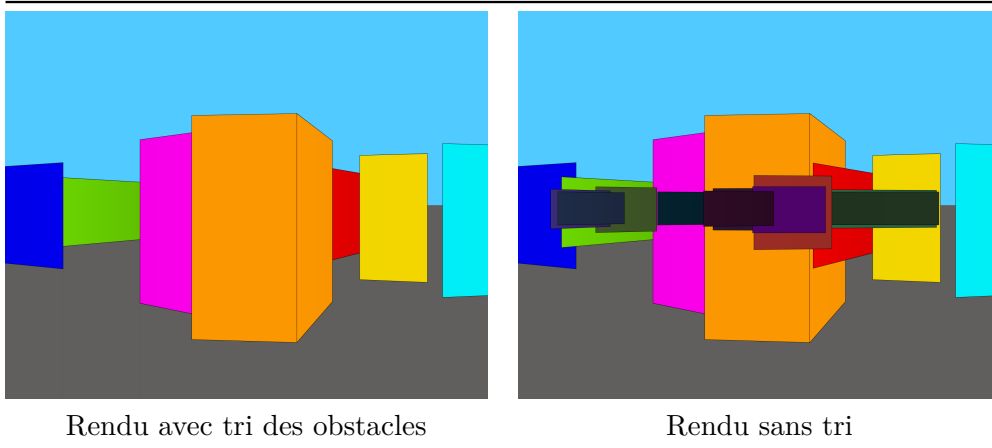
```

'h0': (h0.y * canvas.height) / (Math.tan(radians(player.fov.yamount)) * v.dist),
'h1': (h1.y * canvas.height) / (Math.tan(radians(player.fov.yamount)) * v.dist),
'sat': 1,
'dist': v.dist
};
}

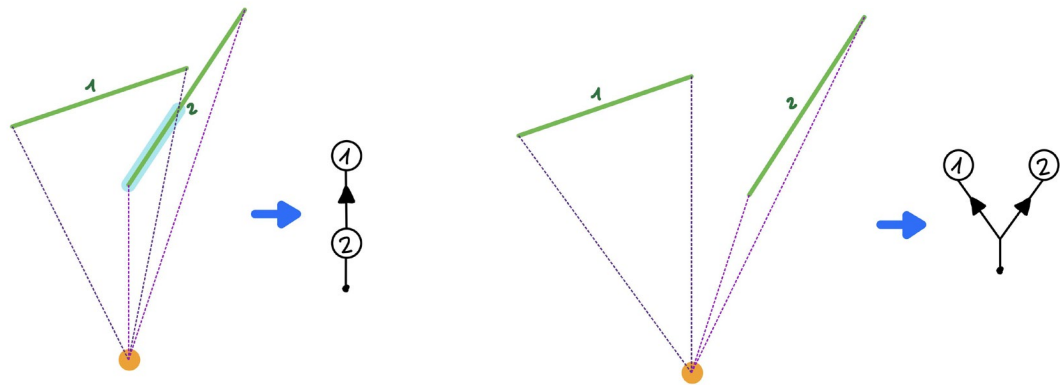
```

### 5.3 Tri des obstacles

Actuellement, on a une liste des obstacles que l'on doit dessiner dans un ordre au hasard. Sans traitement de cette liste, on peut se retrouver avec un rendu comme le suivant.



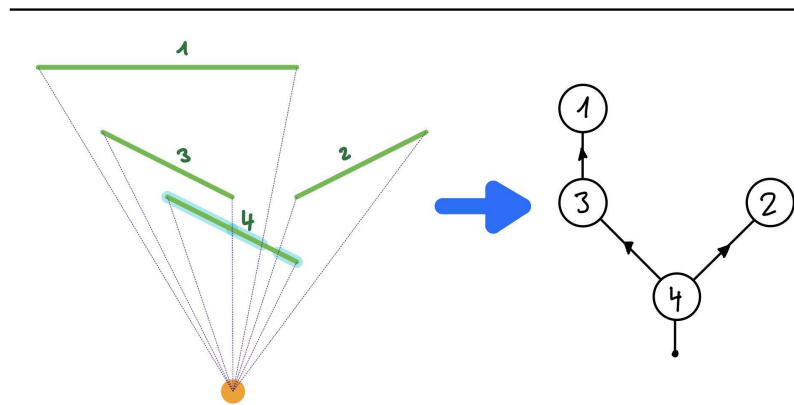
On a donc besoin de trier ces obstacles en fonction de leur distance par rapport au joueur. Ceci tient en compte qu'il faut pouvoir déterminer si un obstacle est derrière ou devant un autre par rapport au joueur. On n'a pas non plus de valeur unique à comparer car chaque obstacle possède deux coordonnées  $x$  et  $y$  et il faut que tout soit calculé relativement au joueur. Un ensemble d'ordre total est un ensemble dont tous les éléments sont comparables entre eux. Par exemple, un ensemble de nombres qui deux à deux peuvent tous être comparés entre eux pour les ordonner du petit au plus grand. Dans notre cas, l'ensemble d'obstacles n'est pas d'ordre total car il peut exister deux paires, où aucun des deux est devant l'autre. En réalité, cet ensemble s'appelle un Ensemble Partiellement Ordonné (ou POSET en anglais pour Partially Ordered Set). Pour effectuer un tri, on utilise ce qu'on appelle un graphe. C'est une manière de représenter la hiérarchie des relations entre des objets. Chaque point, appelé un sommet, représente un objet, et chacun de ces sommets est lié par ce qu'on appelle une arête. En mettant en relation chaque obstacle dans un graphe, la présentation de celui-ci ressemble à une sorte d'arbre.



Deux obstacles qui ont une relation entre eux par rapport au joueur

Deux obstacles qui n'ont aucune relation entre eux par rapport au joueur

Dans les images ci-dessus figurent deux exemples avec une paire d'obstacles ainsi que leur représentation sous forme de graphe. À gauche, l'obstacle 2 apparaît devant l'obstacle 1, donc il est placé avant le premier dans le graphe. À droite, aucun des deux n'est devant ou derrière l'autre, donc on va les représenter par deux arêtes divergentes, appelées branches, sur le graphe.



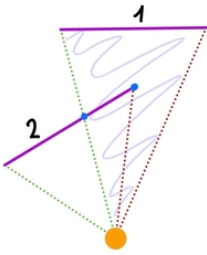
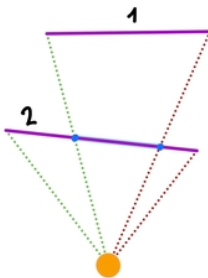
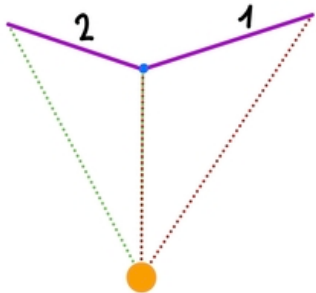
Cas d'un graphe avec plusieurs branches

Dans l'image ci-dessus, les deux cas sont fusionnés. On remarque que l'obstacle 4 paraît devant tous les autres, sans que ceux-ci aient de relation entre eux.

### 5.3.1 Créer les graphes

**5.3.1.1 Critères de comparaison de deux obstacles** Afin de créer les graphes, il faut une fonction qui peut comparer deux obstacles et déterminer s'ils sont l'un devant l'autre ou pas. Cette fonction s'appelle *v1HigherThanv2()*. La comparaison doit se faire de manière à ce que toutes les possibilités de placement d'obstacles soient pris en compte.

La version actuelle prend en compte trois cas.

Cas 1	Cas 2	Cas 3
		
Une des extrémités d'un obstacle devant l'autre	Un obstacle cachant l'autre	Obstacles collés mais pas superposés

**Cas 1)** D'abord, il y a une vérification de si une des extrémités de l'obstacle 2 se trouve devant l'autre. On peut faire ceci en considérant que les deux extrémités de l'obstacle 1 ainsi que le point où se situe le joueur forment un triangle. Ensuite, si le point de l'obstacle 2 est effectivement à l'intérieur du triangle, alors on sait que l'obstacle 2 est bien devant le 1.

**Cas 2)** On réalise aussi une vérification d'intersections entre l'obstacle 2 et les vecteurs reliant le joueur à l'obstacle 1. Si c'est le cas, ça veut dire que l'obstacle 2 est forcément devant le 1.

**Cas 3)** Finalement, on vérifie si les obstacles partagent une extrémités entre eux car ce cas ne retourne pas vrai dans les autres vérifications. En implémentant cette partie du code, j'ai eu des difficultés pour comparer les coordonnées de deux points car il y a comparaison de deux nombres décimaux.

**5.3.1.2 Erreurs de précision des points flottants** L'ordinateur stocke ces nombres, qu'on appelle aussi *points flottants*, sous une notation scientifique en base 2. Bien que cette manière nous offre la possibilité de travailler avec des décimales, elle vient également avec certains problèmes qui sont les erreurs de précision. En effet, l'addition de deux points flottants  $0.1 + 0.2$  qui équivaudrait  $0.3$  normalement, ne donne pas le même résultat dans un programme. Ceci est dû à la représentation de  $0.1$  et  $0.2$  qui sont, selon l'ordinateur équivalent à, par exemple,  $0.100000001$  ou  $0.20000012$ . Ces nombres sont des arrondis qui sont malheureusement nécessaires car un ordinateur ne peut pas représenter ces nombres avec une telle précision.

```
const EPSILON = 0.01
function isSame(v1, v2) {
  return Math.abs(v1.x - v2.x) < EPSILON &&
    Math.abs(v1.y - v2.y) < EPSILON
}
```

Pour contrer ce problème, lorsqu'on veut vérifier l'égalité entre deux points flottants, on prend la différence entre les deux et on vérifie si celle-ci est plus petite qu'une certaine constante. Dans mon programme, j'englobe cette comparaison sous la fonction `isSame()` qui utilise une constante appelée *Epsilon* qui vaut  $0.01$ . Cette fonction retourne vrai si la différence entre les coordonnées des deux points donne un nombre inférieur à la constante. Ceci permet de survenir au problème de précision car on inclut une certaine marge d'erreur.

```
function v1HigherThanv2(w1, w2) {
  // CAS 3
  if (isSame(vectorMult(w1.p, w1.p.dist), vectorMult(w2.h, w2.h.dist)) ||
    isSame(vectorMult(w1.h, w1.h.dist), vectorMult(w2.p, w2.p.dist))) {
    return false
  }
```

```

}

// CAS 1 et 2
return isIntersectionVectors(P1toW1P, W2ptoW2h, w1.index, w2.index) ||
       isIntersectionVectors(P1toW1H, W2ptoW2h, w1.index, w2.index) ||
       ptInTriangle(W2p, player.pos, W1p, W1h, w1.index, w2.index) ||
       ptInTriangle(W2h, player.pos, W1p, W1h, w1.index, w2.index)
}

```

### 5.3.2 Comparer tous les obstacles entre eux

Maintenant qu'on a établi une comparaison entre deux obstacles, on doit pouvoir associer chacun d'entre eux ensemble et ainsi créer un graphe. La fonction *wallsToGraph()* va associer chaque obstacle avec un autre de manière à ce que chaque paire soit unique. C'est une combinaison dont l'ordre n'est pas important car la comparaison avec la fonction *v1HigherThanv2()* se fait dans les deux sens afin de savoir dans quel ordre placer les obstacles dans le graphe. S'ils n'ont aucun lien entre eux, on va les ajouter au graphe mais en tant que sommets individuels plutôt qu'une arrête dans un sens.

```

function wallsToGraph(w) {
    if (w.length < 2) return [];

    const g = new Graph();
    for (let i = 0; i < w.length - 1; i++) {
        for (let j = i+1; j < w.length; j++) {
            if (v1HigherThanv2(w[i], w[j])) {
                g.addEdge(j, i);
            } else if (v1HigherThanv2(w[j], w[i])) {
                g.addEdge(i, j);
            } else {
                g.addVertex(i);
                g.addVertex(j);
            }
        }
    }

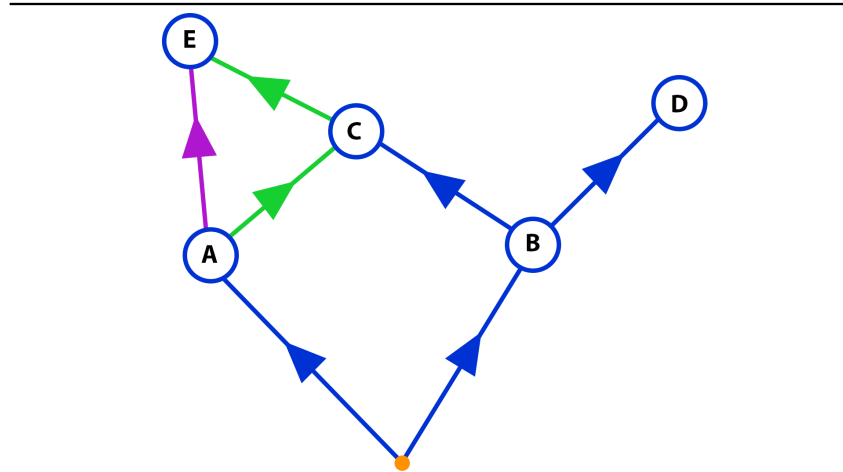
    let final = g.topologicalSort().reverse()
    return final;
}

```

### 5.3.3 Interpréter le graphe

Le graphe ainsi créé, il faut pouvoir l'interpréter afin de pouvoir dessiner les obstacles du plus loin au plus proche. Pour ordonner ce graphe, on va utiliser un algorithme de traversée appelé *Depth First Search* (qui signifie en anglais parcours en profondeur) afin d'accéder à tous les sommets dans leur ordre dans le graphe. On pourra ainsi former une liste des obstacles qu'il faut dessiner dans l'ordre du plus éloigné du joueur au plus proche. Cette traversée consiste en partir de chaque point et de traverser tous les autres sommets pour connaître le plus loin que l'on peut aller depuis ce premier point. En d'autres mots, on va attribuer à chaque point une valeur de distance qui représente le plus long parcours qu'on peut parcourir pour arriver au bout du graphe. Cette technique est possible car le graphe est acyclique. C'est à dire qu'en suivant l'ordre des arrêtes, on ne pourra pas revenir sur les sommets parcourus.





Exemple d'un parcours depuis un sommet

Ici, en partant du sommet A, il y a deux parcours possibles pour arriver au bout du graphe. En allant jusqu'à E, la distance vaut 1, tandis qu'en passant pas C pour ensuite arriver au bout, on parcourt une distance de 2. On attribue dans cet exemple une distance de 2 pour le sommet A. Ainsi, on peut alors dessiner les obstacles dans l'ordre de leur distance qu'on a évalué précédemment.

## 5.4 Dessins des obstacles

Pour finalement dessiner les obstacles sur la perspective 3D, on prend chaque obstacle dans l'ordre de la liste créée par le tri de la section précédente. On utilise les valeurs calculées précédemment  $x_1$ ,  $x_2$ ,  $h_1$  et  $h_2$  qui définissent les coins du polygone final. La couleur est déterminée par une variable `this.hex` (valeur hexadécimale de la couleur) qui est propre à l'obstacle. On utilise une fonction quadratique qui prend pour abscisse la distance des vecteurs  $p$  et  $h$  (vecteurs allant du joueur jusqu'à chacune des extrémités de l'obstacle) afin de calculer le niveau d'obscurité qu'on appliquera par un dégradé entre les deux variations de la couleur initiale. Ceci apporte un air de profondeur car plus une partie de l'obstacle est éloignée du joueur, plus celle-ci sera foncée.

```
function display3D() {
    const maxl = 0;
    const minl = -0.75;
    const a = canvas2D.width/1.2;
    const n = 2;

    let L1 = -((this.p.dist / a) ** n);
    if (L1 < minl) L1 = minl;
    if (L1 > maxl) L1 = maxl;

    let L2 = -((this.h.dist / a) ** n);
    if (L2 < minl) L2 = minl;
    if (L2 > maxl) L2 = maxl;

    const grd = ctx.createLinearGradient(
        this.x1 + canvas.width / 2, canvas.height / 2,
        this.x2 + canvas.width / 2, canvas.height / 2);
```

```
grd.addColorStop(0, shadeHexColor(this.hex, L1));
grd.addColorStop(1, shadeHexColor(this.hex, L2));
ctx.fillStyle = grd;

polygon([this.x1 + canvas.width / 2, this.h1.h0 + canvas.height / 2,
this.x1 + canvas.width / 2, this.h1.h1 + canvas.height / 2,
this.x2 + canvas.width / 2, this.h2.h1 + canvas.height / 2,
this.x2 + canvas.width / 2, this.h2.h0 + canvas.height / 2
], `grd`, 2);
}
```

## 6 Conclusion

---

Pour récapituler, l'implémentation de ma nouvelle méthode pour un moteur de rendu 3D avec le Raycasting a été réussie. Bien que le code soit plus complexe, celui-ci est capable de nous dessiner des obstacles projetés lisses sur les bords sans générer des centaines de rayons. Malgré l'augmentation des lignes de code nécessaires pour exécuter cette méthode, on atteint notre but avec plus d'efficacité ; Dans la version de ce projet comme travail de Maturité, j'avais une erreur dans la projection qui pouvait déformer les obstacles qui sont parallèles au vecteur directeur du joueur. Depuis, j'ai pu régler ce problème.

Ce travail a abouti à un projet dont je suis très fier car j'ai réussi à programmer toutes les fonctionnalités que possédait l'autre méthode mais à la manière que je désirais. Maintenant, on peut réellement se plonger dans ce monde qui semble être un espace 3D mais qui ne l'est pas vraiment.

## 7 Bilan personnel

---

Lors de la réalisation de mon projet, j'ai dû m'investir dans un travail de recherche et de logique afin de trouver les meilleures solutions aux problèmes qui se sont posés. Ma perception de ce que je voulais rendre comme travail final à beaucoup changé au cours de cette dernière année.

En effet, je voulais initialement utiliser la première méthode pour créer un moteur de jeu qui permettrait à quiconque de programmer ses propres jeux vidéo 3D avec plus de facilité. C'est seulement après avoir codé les premières versions que je me suis rendu compte que le rendu ne me plaisait pas suffisamment.

Petit à petit, j'ai réalisé des petites modifications qui ont abouti à un nouvel algorithme pour lequel j'ai dû approfondir mes connaissances dans l'application des maths dans l'informatique. Malgré les heures de recherches et d'échecs que j'ai dû traverser, le fait de voir le résultat fonctionner m'apporte une grande satisfaction.

## 8 Annexe

### 8.1 a) Trouver l'intersection de deux segments

Soit  $P_x, P_y$ , les coordonnées du point d'intersection des deux segments définis par  $(x_1, y_1)$  et  $(x_2, y_2)$  pour le premier et  $(x_3, y_3)$  et  $(x_4, y_4)$  pour le deuxième. Ainsi, on pose  $t, u$  tel que :

$$t = \frac{(x_1 - x_3)(y_3 - y_4) - (y_1 - y_3)(x_3 - x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)} \quad \text{et} \quad u = \frac{(x_1 - x_3)(y_1 - y_2) - (y_1 - y_3)(x_1 - x_2)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}$$

On note que le dénominateur est le même pour le calcul de  $u$  et de  $t$ .

Ces deux formules sont le résultat d'un développement d'un calcul de déterminants<sup>12</sup> (notion associée aux matrices).

Avant même de calculer le point  $P_x$  et  $P_y$ , là où se trouve l'intersection des deux segments, on peut vérifier si cette intersection existe.

L'intersection des deux segments existe si  $0.0 \leq t \leq 1.0$  et  $0.0 \leq u \leq 1.0$ . Ceci permet d'ignorer le résultat et de seulement savoir s'il y a une intersection afin d'optimiser le programme.

En revanche, on peut quand même calculer  $P_x$  et  $P_y$  tels que :

$$(P_x, P_y) = (x_3 + u(x_4 - x_3), y_3 + u(y_4 - y_3))$$

Dans mon code, j'ai deux fonctions en lien avec l'intersection : *Intersection()* qui détermine les coordonnées d'un point d'intersection de deux segments s'il existe, et *isIntersection()* qui fait de même sans calculer les coordonnées.

```
function isIntersection(w1, w2) {
    const x1 = w1.pos.x;
    const y1 = w1.pos.y;
    const x2 = x1 + w1.dir.x * w1.dir.length;
    const y2 = y1 + w1.dir.y * w1.dir.length;
    const x3 = w2.pos.x;
    const y3 = w2.pos.y;
    const x4 = x3 + w2.dir.x;
    const y4 = y3 + w2.dir.y;
    const den = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4);
    if (den == 0) return false;
    // Si le dénominateur est égal à 0, le calcul engendrerait une erreur
    // donc l'intersection n'existe forcément pas
    const t = ((x1 - x3) * (y3 - y4) - (y1 - y3) * (x3 - x4)) / den;
    const u = -((x1 - x2) * (y1 - y3) - (y1 - y2) * (x1 - x3)) / den;

    // Si l'intersection existe, on retourne vrai ou faux.
    return u >= 0 && u <= 1 && t >= 0 && t <= 1;
}
```

<sup>12</sup>Line-line intersection. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 6 octobre 2021 à 21 :04. [Consulté le 9 octobre 2021]. Disponible à l'adresse : [https://en.wikipedia.org/wiki/Line%E2%80%93line\\_intersection](https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection)

```

function intersection(w1, w2) {
  // On n'effectuera aucun calcul si aucune intersection n'existe!
  if (!isIntersection(w1, w2)) return;

  const x1 = w1.pos.x;
  const y1 = w1.pos.y;
  const x2 = x1 + w1.dir.x * w1.dir.length;
  const y2 = y1 + w1.dir.y * w1.dir.length;
  const x3 = w2.pos.x;
  const y3 = w2.pos.y;
  const x4 = x3 + w2.dir.x;
  const y4 = y3 + w2.dir.y;

  const den = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4);
  const u = ((x2 - x1) * (y1 - y3) - (y2 - y1) * (x1 - x3)) / den;

  const xint = x3 + u * (x4 - x3);
  const yint = y3 + u * (y4 - y3);
  const intersection = {
    'x': xint,
    'y': yint,
    // Formule pour calculer la distance entre deux points
    'dist': Math.sqrt((y1 - yint) ** 2 + (x1 - xint) ** 2)
  };
  return intersection;
}

```

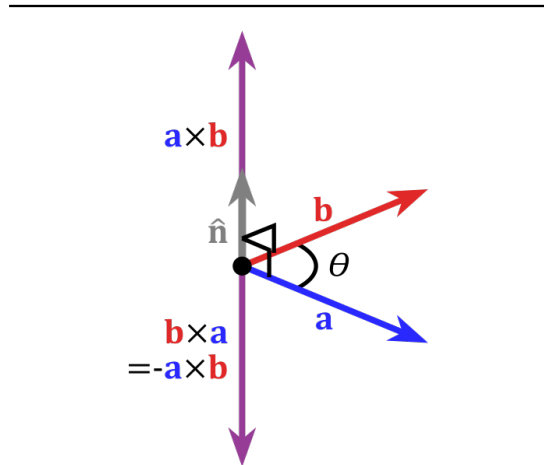
## 8.2 b) Trouver si deux vecteurs sont orientés dans le sens des aiguilles d'une montre

Une fonction beaucoup utilisée dans mon programme s'appelle *isClockwiseOrder()*. Cette fonction retourne vrai si le premier vecteur inséré est orienté de telle sorte qu'il vient avant le deuxième dans le sens des aiguilles d'une montre. Ceci me permet de comparer l'orientation de deux vecteurs sans me soucier de leur angle relatif. L'opération qui se trouve derrière cette fonction est le produit vectoriel.

Comme nous sommes sur un plan en 2D on appliquera le produit vectoriel  $a \times b$  avec  $a_z = 0$  et  $b_z = 0$  :

$$a \times b = \begin{pmatrix} a_x \\ a_y \\ 0 \end{pmatrix} \times \begin{pmatrix} b_x \\ b_y \\ 0 \end{pmatrix} = \begin{pmatrix} a_y \cdot 0 - 0 \cdot b_y \\ 0 \cdot b_x - a_x \cdot 0 \\ a_x \cdot b_y - a_y \cdot b_x \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ a_x \cdot b_y - a_y \cdot b_x \end{pmatrix}$$

Ceci simplifie les calculs à effectuer et réduit le résultat à une valeur seulement. Cette valeur représente un certain vecteur perpendiculaire au plan 2D dont la hauteur est déterminée par le calcul. Ainsi, par la règle d'anticommutativité du produit vectoriel, (c'est-à-dire que  $a \times b = -b \times a$ ), on peut facilement déterminer si deux vecteurs sont orientés dans le sens des aiguilles d'une montre.



L'anticommutativité du produit vectoriel<sup>13</sup>

En conclusion, si le résultat de  $a_x \cdot b_y - a_y \cdot b_x$  est positif, alors  $a$  et  $b$  sont orientés dans le sens des aiguilles d'une montre. S'il est négatif, alors  $a$  et  $b$  sont orientés dans le sens contraire des aiguilles d'une montre. S'il est égal à 0, alors les vecteurs sont parallèles. Dans mon code, je considère que ce cas est équivalent au premier par simplification.

```
function isClockwiseOrder(v1, v2) {
    return v1.x*v2.y - v1.y*v2.x <= 0;
}
```

<sup>13</sup>Représentation de l'anti commutativité du produit vectoriel, (auteur inconnu), 2008, Disponible à l'adresse : [https://commons.wikimedia.org/wiki/File:Cross\\_product\\_vector.svg](https://commons.wikimedia.org/wiki/File:Cross_product_vector.svg)

## 9 Bibliographie

---

### 9.1 Images

- Page de titre : Vue d'obstacles multicolores en perspective ; Image générée à partir de mon code, HAMELINK Marcus 2021
- The cross product with respect to a right-handed coordinate system (auteur inconnu), 2008, Disponible à l'adresse : [https://en.wikipedia.org/wiki/Cross\\_product#/media/File:Cross\\_product\\_vector.svg](https://en.wikipedia.org/wiki/Cross_product#/media/File:Cross_product_vector.svg)
- Représentation schématique des valeurs ; Schéma que j'ai dessiné, HAMELINK Marcus 2021
- Schémas pour illustrer la comparaison entre deux obstacles ; Schémas que j'ai dessinés, HAMELINK Marcus 2021

### 9.2 Sources

- Raycasting. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 22 juillet 2021 à 22 :17. [Consulté le 10 octobre 2021]. Disponible à l'adresse : <https://fr.wikipedia.org/wiki/Raycasting>
- SHIFFMAN Daniel, 2019. Coding Challenge #146 : Rendering Raycasting [enregistrement vidéo]. Youtube [en ligne]. Disponible à l'adresse : <https://youtu.be/vYgIKn7iDH8>
- Line-line intersection. Wikipédia : l'encyclopédie libre [en ligne]. Dernière modification de la page le 6 octobre 2021 à 21 :04. [Consulté le 14 octobre 2021]. Disponible à l'adresse : [https://en.wikipedia.org/wiki/Line%E2%80%93line\\_intersection](https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection)
- VANDEVENNE Lode, 2004-2021. Lode's Computer Graphics Tutorial : Raycasting [en ligne]. Dernière modification de la page en 2020. [Consulté le 15 janvier 2021] Lode Vandevenne, 2004-2020. Disponible à l'adresse : <https://lodev.org/cgtutor/raycasting.html>