Cégep du Vieux Montréal

ENGIN DE RAYCASTING

Projet synthèse

Réalisé par :
Frédéric Bélanger

Présenté à :
Jean-Christophe Demers

Dans le cadre du cours :
420-B65-IN - Synthèse

En date du
24 février 2020

Table des matières

Préface

Ce document est conçu avec une approche académique dans le but de simuler un cours sur une technologie donné à des étudiants de niveau intermédiaire. La présentation visuelle mettra donc l'emphase sur le lien entre la conception 2D des niveaux et la représentation en 3D. Le texte en rouge ne serait pas visible pour d'éventuels étudiants. Il consiste principalement en des réponses aux questions qui leur sont posées ou des notes concernant le projet synthèse. Les algorithmes sur fond noir seraient également cachés.

Résumé du projet

Ce projet vise à produire une démo technique d'un engin de *raycasting*. Cette technique de rendu a été utilisée dans la production des premiers jeux de tirs à la première personne (*first person shooter* ou FPS en anglais). Le plus connu d'entre eux est sans nul doute *Wolfenstein 3D*. Vu qu'il ne s'agit pas d'un jeu complet vous n'aurez qu'à produire un seul niveau et les contrôles seront relativement limités.

La réalisation devra se faire en Java et sous forme de programmation orientée objet. Vous devrez fournir un schéma de vos classes (UML ou autre) ainsi que vos réponses aux différentes questions de ce document avant de commencer à coder.

Le raycasting, présentation sommaire

Rendu 3d moderne

Dans les jeux vidéo modernes, la méthode de rendu 3d majoritairement utilisée consiste à générer les surfaces des objets en assemblant plusieurs polygones auxquels on ajoute des textures. Les effets visuels comme la luminosité, la réflexion ou les ombres sont ajoutés manuellement aux scènes à l'aide de filtres. Une fois le rendu 3D terminé, la carte graphique agit un peu comme une caméra qui filmerait la scène d'un certain point de vu pour produire l'image qui sera affichée à l'écran. C'est ce que l'on nomme la rastérisation. Cette méthode de rendu est très exigeante pour le matériel car il faut évaluer l'état (couleur et intensité) de chaque pixel. Pour un écran 1080p cela représente 2,073 600 pixels (1920x1080) à évaluer à une cadence moyenne de 30 à 60 trames par seconde (voir même plus).

Rendu 3d en raycasting

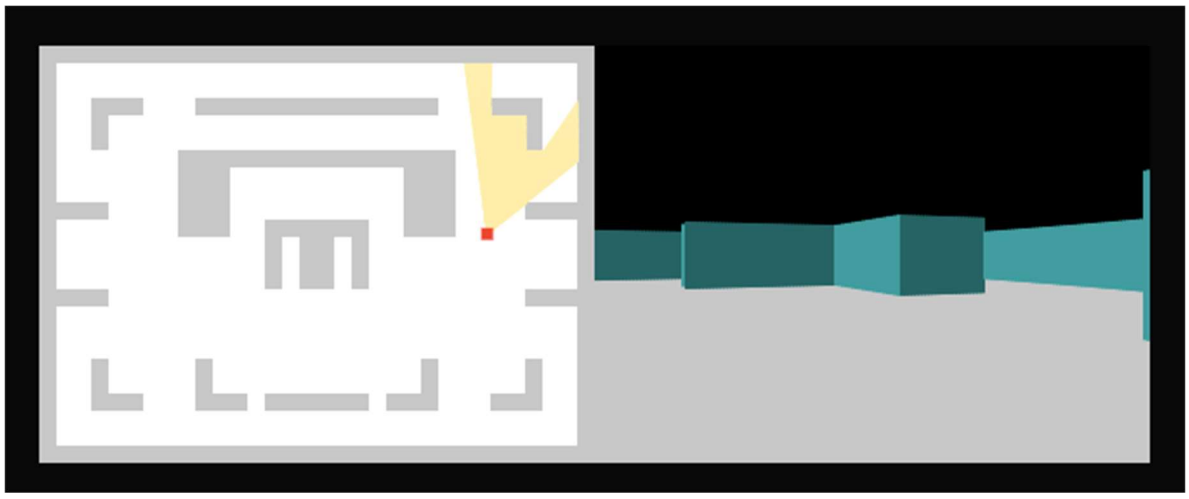

Figure 1 : Engin de raycasting (vue 2D à droite et conversion 3D à gauche)

À l'époque des premiers jeux 3D, les ordinateurs n'étaient pas suffisamment performants pour produire de véritables graphiques tridimensionnels. Le *raycasting* fut utilisé pour contourner ce problème. Cette technique est la forme la plus basique de rendu 3D. Le premier algorithme fut présenté en 1968, par Arthur Appel (aucun lien avec Apple). Le terme fut utilisé pour la première fois en graphisme informatique dans un article de Scott Roth pour décrire une méthode de rendu de modèles géométriques solides.

On surnomme parfois le *raycasting* « pseudo 3D ». En effet tout le jeu est créé et se déroule sur une carte 2D vue de haut (*top-down view*). Contrairement au « vrai » 3D il n'y a aucun calcul effectué sur l'axe Z. Ce n'est que lors de l'envoi des données pour l'affichage que l'on convertit les éléments 2D en une vue 3D.

Pour ce faire on divise la partie de l'écran où sera affichée la représentation 3D du jeu en colonne de pixels. On trace un rayon à partir du joueur en direction de son champs visuel pour chaque colonne de l'affichage. On fait évoluer ce rayon jusqu'à ce qu'il rencontre un mur.

Le milieu horizontal des murs correspond à celui de l'écran. Dans une colonne de pixel le mur est donc toujours situé exactement au centre. Il ne reste qu'à déterminer la longueur du segment du mur dans cette colonne de pixel en fonction de la distance entre le joueur et l'obstacle. Plus l'obstacle est éloigné, plus la partie représentant le mur dans la colonne de pixels sera petite. Les sections restantes après et avant le mur seront respectivement le plafond et le plancher. Tout cela peut sembler complexe mais des explications imagées et plus détaillées suivront sous peu.

Interface usager

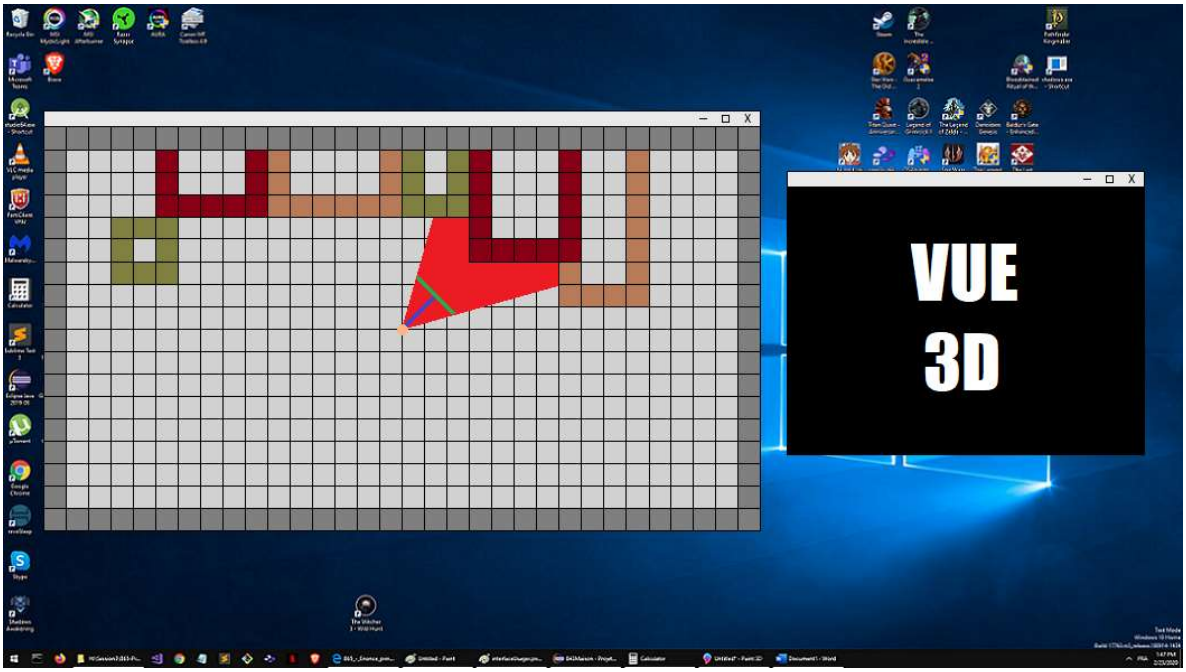

Figure 2 : Interface usager

Puisqu'il s'agit d'une démo technique l'interface usager sera très simple. Il n'y aura pas de menu, le programme s'ouvrira directement sur la vue ci-haut. La fenêtre de gauche contiendra la vue 2D, également appelée vue de haut (*top-down view* en anglais). La fenêtre de droite sera réservée pour la représentation 3D de ce qui est dans le champs visuel du joueur (cône rouge dans la vue 2D). Vu les talents artistiques limités de votre auteur, voir la Figure 1 pour un exemple de représentation visuelle 3D.

Cas d'utilisation

Encore une fois vu que nous ne réalisons qu'une démo technique, les possibilités de contrôle par l'usager seront relativement limitées. Toutes les commandes se feront sur la fenêtre 2D. En voici les détails :

Touche du clavier :

- W Faire avancer le joueur
- A Faire glisser le joueur vers la gauche (*strafe left*)
- S Faire reculer le joueur
- D Faire glisser le joueur vers la droite (*strafe right*)
- R Alterner entre le visuel des rayons avec et sans correction de la distorsion
- P Pauser et repartir la démo

Souris :

- Déplacement latéral gauche rotation de la caméra vers la gauche
- Déplacement latéral droit rotation de la caméra vers la droite

Patron de conception

Observateur

Une classe nommée « Observateur » (ou *Observer* si vos noms sont en anglais) devra être présente pour gérer les commandes de l'usager. Cette classe fera son observation sur la fenêtre de la vue 2D, ou plus précisément sur son *JFrame*. Les événements de type commandes seront emmagasinés dans un dictionnaire (*hashmap*) qui sera utilisé ensuite par l'engin du jeu.

C'est également cette classe qui s'occupera de mettre à jour les trames (*frames* en anglais) du jeu selon l'horloge à tous les 16 millièmes de seconde. Cela nous donnera un taux de rafraichissement d'image d'environ 60 trames par seconde (ou *fps*), ce qui est très commun dans l'industrie des jeux vidéo.

Modèle – Vue – Contrôleur

Votre projet devra avoir une classe EnginDeJeu (ou *GameEngine* en anglais) qui servira de modèle. Elle se chargera de la logique du jeu et mettra à jour les données pour les affichages.

Normalement la partie 2D de l'engin n'est pas affichée à l'écran et sert uniquement à effectuer les calculs pour la représentation 3D. Il arrive que cette vue soit convertie en carte de jeu permettant à l'utilisateur de visualiser sa position dans le niveau. Étant donné que notre projet a un but éducatif, nous utiliserons 2 fenêtres distinctes qui seront simultanément affichées à l'écran. La vue sera donc constituée de deux classes, une pour la fenêtre 2D (Vue2D ou *View2D*) et l'autre pour la fenêtre 3D (Vue3D ou *View3D*). L'objectif est que l'utilisateur puisse comprendre très rapidement les mécanismes impliqués. Cela permet également de faciliter le débogage en apportant un indice visuel aux différentes problématiques potentielles.

Une classe nommée Horloge (ou *GameClock* en anglais) agira comme contrôleur. À l'aide de l'observateur elle fera la mise à jour de l'engin selon les commandes de l'usager à une fréquence régulière de 60 *fps*. Elle demandera également la mise à jour des 2 fenêtres de la vue.

Les calculs

Notes importantes

- Dans votre code, toutes les dimensions devront être calculées en fonction de la taille et de la largeur des deux fenêtres d'affichage.
- Dans tous vos calculs le degré 0 devra se situer sur l'axe nord.
- Les fonctions Math.[tan() ,sin() ,cos()] de Java prennent en paramètre des angles en radian. Pour les calculs trigonométriques il vous faudra donc convertir les angles de degré à radian en utilisant la formule :

  AngleRad = AngleDeg * PI / 180

- Les fonctions Math.[atan(), asin(), acos()] de Java donnent des angles en radian. Pour les convertir en degré il faudra utiliser la formule :

  AngleDeg = AngleRad * 180 / PI

Règles de calcul trigonométrique

- La tangente d'un angle est égale à la longueur de son côté opposé divisée par la longueur de son côté adjacent :

  Tan(angleRad) = côté opposé/côté adjacent

- Le sinus d'un angle est égal à la longueur de son côté opposé divisée par la longueur de l'hypoténuse. :

  Sin(angleRad) = côté opposé/hypoténuse

- Le cosinus d'un angle est égal à la longueur de son côté adjacent divisée par la longueur de l'hypoténuse :

  Cos(angleRad) = côté adjacent/hypoténuse

- On utilise arctan, arcsin et arccos (atan, asin et acos en Java) pour trouver la valeur d'un angle à partir de sa tangente, son sinus ou son cosinus.

  Ex : angleRad = arctan(tan(angleRad))

Étapes de conception

La grille

Comme nous le verrons plus loin, pour faciliter l'évolution des rayons vers les obstacles il est essentiel que les niveaux soient conçus sur une grille. Les murs doivent toujours occuper entièrement des cases de celle-ci. La première étape du projet consiste donc à calculer les dimensions de la grille et de ses cases en fonction de la largeur et de la hauteur de la fenêtre 2D. Vu que nos cases seront carrées il est important de choisir une largeur et une hauteur de fenêtre qui permettront des divisions sans reste. Voici le détail des valeurs initiales suggérées :

- Largeur de la fenêtre 2D : 1280 px
- Hauteur de la fenêtre 2D : 720 px
- Nombre de case de la grille 2D sur une largeur de fenêtre : 32
- Largeur de la fenêtre 3D : 800 px
- Hauteur de la fenêtre 3D : 600 px

Ces 5 valeurs seront les seules dimensions fixes de notre projet. Toutes les autres tailles seront dérivées de celles-ci. Par exemple la largeur et la hauteur d'une case seront égales à la largeur de la fenêtre 2D divisée par le nombre de cases pouvant être affichées sur sa largeur. Il sera possible d'ajuster la taille des cases pour obtenir des murs de dimensions réalistes une fois affichés en 3D.

Chaque case de la grille sera un objet qui devra au minimum être composé de deux points : haut-gauche, et bas-droit. Cette classe devra également inclure une fonction mettant à jour les deux points lors d'une translation de la grille (nous y reviendrons plus loin).

Position du joueur dans la grille

Il existe plusieurs méthodes pour faire déplacer un élément dans un jeu 2d. Pour simplifier nos calculs nous opterons pour une **position du joueur fixe au centre de l'écran**. Tous les objets du jeu et la grille se déplaceront donc par translation autour du joueur. **Important** : La case de départ du joueur doit être connue en tout temps. Lors de translation de la grille il faut calculer la nouvelle position (case) du joueur dans la grille.

La carte de jeu

Initialement notre carte fera exactement la largeur et la hauteur de la fenêtre 2D. Le joueur pourra seulement changer de point de vu (rotation de la caméra) sans être en mesure de se déplacer. Nous n'aurons donc pas à nous soucier des translations des objets de la carte qui seront ajouter plus tard. L'objectif est de faciliter l'implantation des premiers éléments plus complexes de notre projet.

La carte sera représentée sous forme de tableau d'entiers de deux dimensions dont chaque index correspond à une case de la grille. Celles dont la valeur est 0 sont vides (ce sont des tuiles de plancher). Les autres sont occupées par des murs dont le type est déterminer par la valeur de l'entier (ex : 1 = mur brun, 2 = mur bleu, etc.). Chaque case de notre grille aura une couleur définie par le type de mur qui l'occupe ou par celle des planchers (case blanche = 0 = plancher). Nous créerons éventuellement un petit générateur de carte élémentaire dans Excel et nous chargerons les fichiers dans notre logiciel lors du début d'un niveau. Pour le moment notre carte sera implantée directement dans le code.

Voici à quoi elle ressemblera :

```
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
{1,0,0,0,0,2,0,0,0,2,3,0,0,0,0,3,4,0,4,2,0,0,0,2,0,0,3,0,0,0,0,1},
{1,0,0,0,0,2,0,0,0,2,3,0,0,0,0,3,4,0,4,2,0,0,0,2,0,0,3,0,0,0,0,1},
{1,0,0,0,0,2,2,2,2,2,3,3,3,3,3,3,4,4,4,2,0,0,0,2,0,0,3,0,0,0,0,1},
{1,0,0,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,2,0,0,3,0,0,0,0,1},
{1,0,0,4,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2,2,2,2,0,0,3,0,0,0,0,1},
{1,0,0,4,4,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,0,0,3,0,0,0,0,1},
{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,3,3,3,0,0,0,0,1},
{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}
```

La caméra

La caméra est l'élément clé d'un engin de *raycasting*. Son bon fonctionnement est essentiel si on veut avoir une représentation 3D qui n'est pas déformée.

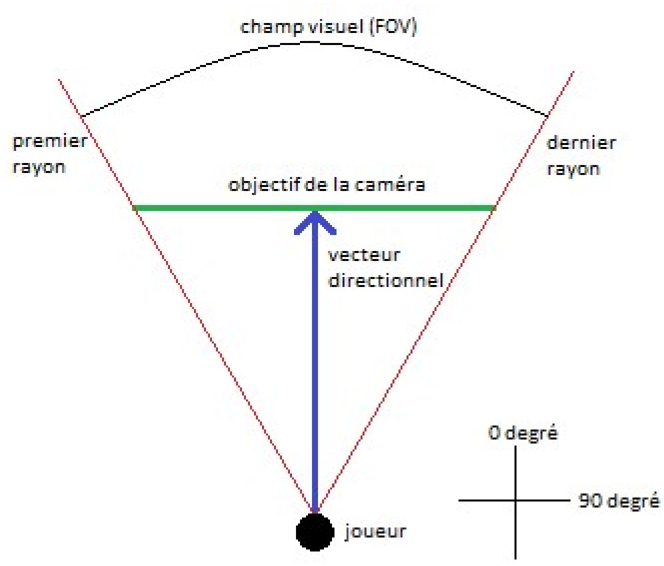

Figure 3 : Les différentes composantes de la caméra

Le champ visuel est l'angle du plan horizontal dans lequel la perception visuelle du joueur est possible. En anglais il est souvent abrégé sous l'acronyme FOV (*Field of View*). Une valeur de départ de 60 degrés est suggérée. On fait varier la longueur du vecteur directionnel et la longueur de l'objectif de la caméra pour

obtenir l'angle de champ visuel désiré. Par exemple un objectif de caméra 2 fois plus long que le vecteur directionnel donnera un FOV de 90^0.

L'objectif de la caméra pourrait également être appelée l'écran de l'ordinateur. Chaque point de ce segment représente une colonne de pixel de l'affichage. La largeur de l'objectif sera 4 fois la largeur d'une case. Cela permettra éventuellement de le diviser en parts égales pour former les colonnes la vue 3D.

Le vecteur directionnel représente la distance entre le joueur et l'objectif de la caméra (ou l'écran).

Question : Connaissant la longueur de l'objectif et l'angle de FOV désiré, calculez la longueur de ce vecteur. Indice : En incluant les deux rayons des extrémités on observe 2 triangles rectangles.

Nous avons comme éléments connus un angle (la moitié du FOV) et le côté opposé (le vecteur directionnel). Nous cherchons le côté adjacent = côté opposé / tan(angle).

Rotation de la caméra

Comme dans les jeux de tirs modernes, les rotations gauche/droite (pas de rotation haut/bas) de la caméra se feront à l'aide de déplacements latéraux de la souris. Il faudra donc ajouter un écouteur sur les mouvements de souris dans le canevas 2d. Chaque déplacement de 1 pixel sur l'axe des x équivaux à une rotation de 1 degré de la caméra. Les mouvements vers la droite feront tourner la caméra dans le sens horaire et vice-versa pour la gauche. On peut jouer sur le ratio nombre de pixel/degré de rotation pour augmenter ou diminuer la sensibilité de la souris.

Formules pour calculer la nouvelle position NP d'un point A effectuant une rotation d'angle R (en RAD) autour d'un point C :

NPx = cos(R) * (Ax - Cx) - sin(R) * (Ay - Cy) + Cx

NPy = sin(R) * (Ax - Cx) + cos(R) * (Ay - Cy) + Cy

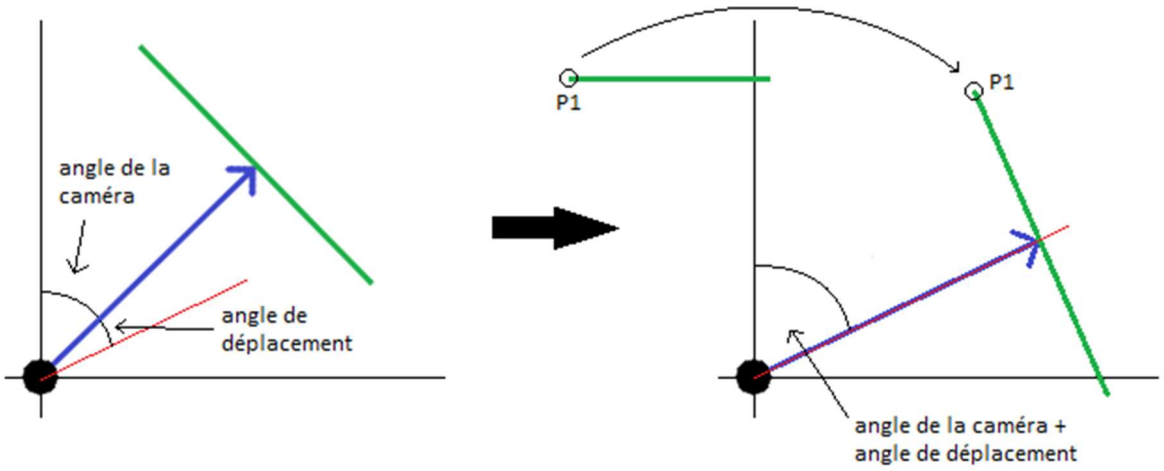

Figure 4 : Rotation horaire de la caméra

Important :

Des accumulations d'erreurs dues aux arrondissements des valeurs pourraient déformer notre caméra. En effet chaque fois qu'on déplace la souris on effectue plusieurs petites rotations successives. Au cours d'une partie cela peut représenter plusieurs milliers de rotations. Des erreurs infimes s'accumuleront et finiront par faire en sorte que l'objectif et le vecteur de direction ne soient plus alignés correctement.

Question : Comment peut-on éviter cela?

> Lors de la création de la caméra les valeurs initiales des positions de tous les éléments seront conservées dans des variables. Lors de mouvement de la souris la rotation sera faite à partir des positions initiales de tous les éléments. L'angle de rotation sera l'angle équivalent au mouvement de la souris plus l'angle actuel de la caméra.

Évolution d'un rayon

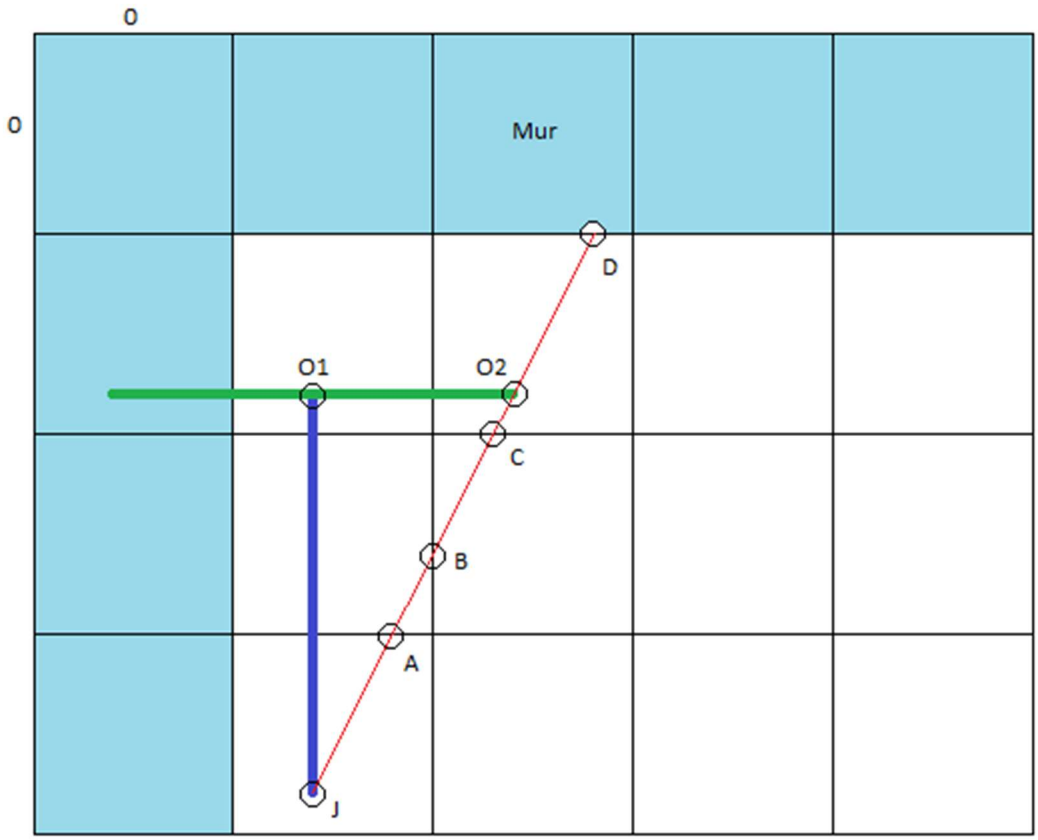

Figure 5 : Évolution d'un rayon case par case

La position du point J (le joueur) dans la grille est connue. Les positions à l'écran des points J, O1 et O2 sont également connues. Nous voulons tracer un rayon à partir du point J et le faire évoluer case par case jusqu'à ce qu'il rencontre un mur. La longueur totale du rayon (J-D) nous permettra éventuellement de calculer la section de colonne qui sera de la couleur du mur frappé par celui-ci.

Évolution du rayon dans une case

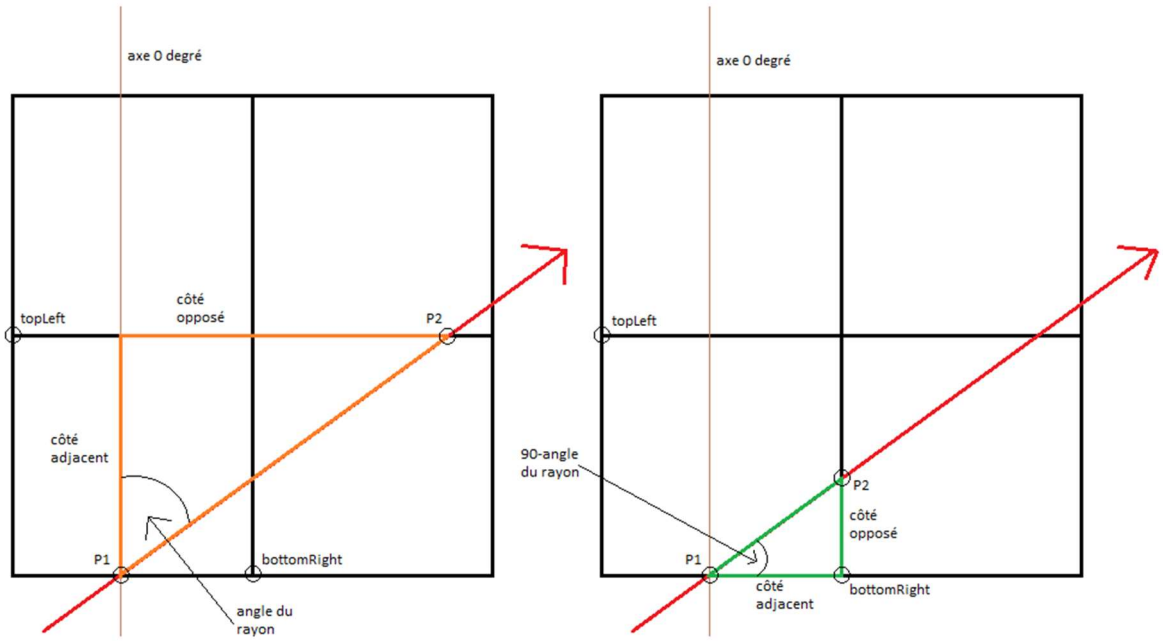

Figure 6 : Évolution d'un rayon dans une case, orientation nord-est (1-89 degrés)

Il y a huit possibilités d'évolution selon l'angle du rayon. La figure ci-dessus nous en montre un dont l'angle se situe entre 0 et 90 degrés. Comme on peut le voir il faut évaluer 1 à 2 points pour trouver le bord de la case de départ qui croisera le rayon (le point P2). On commence par vérifier si le croisement se fera sur l'axe des X (image droite, triangle orange).

Question : Connaissant la longueur du côté adjacent (hauteur d'une case) et l'angle, comment trouver la longueur du côté opposé?

Côté opposé = tan(angle du rayon) * côté adjacent.

Question : La position du point P1 étant connue, comment trouver la position X du point P2?

P2.x = P1.x + côté opposé.

Question : Nous devons maintenant vérifier si P2 touche à une bordure de la case de départ. Comment procède-t-on?

Si P2.x < bottomRight.x alors notre point de croisement se situe dans le haut de la case de départ.

Question : Puisque dans notre exemple le point x ne croise pas un bord de la case de départ, comment calculer le point de croisement sur l'axe des Y (image gauche, triangle vert)?

Nous devons trouver la longueur du côté adjacent du triangle vert de la figure 5. Donc : Côté adjacent = bottomRight.x-P1.x. L'angle de notre triangle n'est plus celui du rayon. Il est de :

Angle = 90^0 – angle du rayon.

Nous avons l'angle et le côté adjacent, nous cherchons le côté opposé :

Côté opposé = tan(angle) * côté adjacent.

Puisque nous savons que le croisement ne se fait pas sur l'axe X il est certain que P2 est sur l'axe Y. Nous n'avons pas à le vérifier.

Question : Nous devons maintenant vérifier si la case d'arrivée est un mur. La position de la case de départ sur la grille est connue. Quelle sera la position de la case d'arrivée si le rayon croise le haut de la case de départ?

Case d'arrivée rangée = case de départ rangée - 1.

Et si le rayon croise le côté droit?

Case d'arrivée colonne = case de départ colonne + 1.

Question : Pouvez-vous écrire l'algorithme complet pour les 4 possibilités d'angle (les cadrans nord-est, sud-est, sud-ouest et nord-ouest). **Important** : il y a 4 cas particuliers à traiter lorsque l'angle du rayon arrive directement sur l'axe des X ou des Y. Votre algorithme doit également fonctionner pour le point de départ (la position du joueur) qui a de fortes chances de ne pas être situé directement sur le bord d'une case.

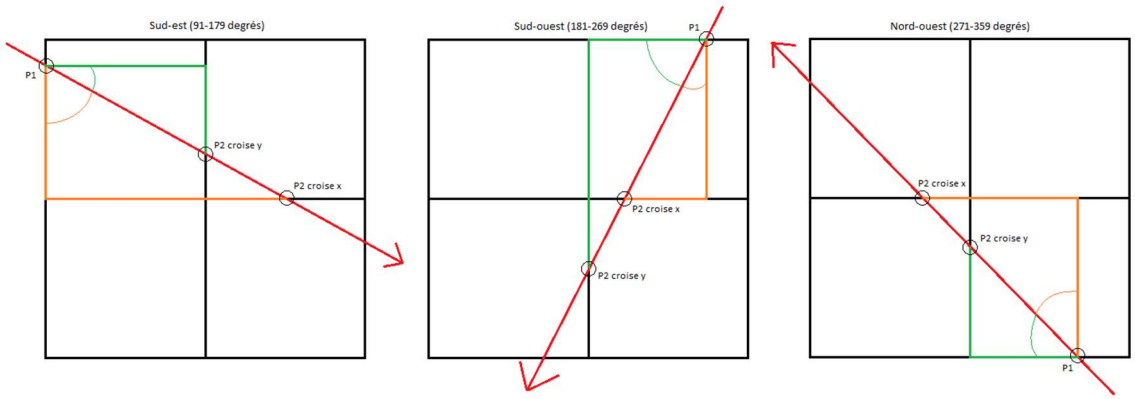

Figure 7 : Évolution d'un rayon dans les cadrans sud-est, sud-ouest et nord-ouest

Réponse : voir page suivante

```
P2.x = P1.x
P2.y = P1.y
si angleRayon  == 0:
        P2.y = hautGauche.y
        caseAtester.rangée -= 1
sinon si angleRayon == 180 :
        P2.y = basDroit.y
        caseAtester.rangée +=  1
sinon si angleRayon == 90 :
        P2.x = basDroid.x
        caseAtester.colonne += 1
sinon si angleRayon == 270 :
        P2.x = hautGauche.x
        caseAtester -= 1
sinon si angleRayon > 0 et angleRayon < 90 :
        adjacent = P1.y – hautGauche.y
        opposé = opposé * tan(angleRayon)
        P2.x = P1.x + opposé
        si P2.x < basDroit.x :
                P2.y = hautGauche.y
                caseAtester.rangée -=1
        sinon :
                angle = 90 – angleRayon
                adjacent = basDroit.x – P1.x
                opposé = adjacent * tan(angle)
                P2.y = P1.y – opposé
                P2.x = basDroit.x
                caseAtester.colonne += 1
sinon si angleRayon > 90 et angleRayon < 180 :
        angle = 180 - angleRayon
        adjacent = basGauche.y – P1.y
        oppose = adjacent * tan(angle)
        P2.x = P1.x  + opposé
        si P2.x < basDroit.x :
                P2.y = basDroit.y
                caseAtester.rangée +=1
```

```
        sinon :
                angle = angleRayon – 90
                adjacent = basDroit.x – P1.x
                opposé = adjacent * tan(angle)
                P2.y = P1.y + opposé
                P2.x = basDroit.x
                caseAtester.colonne+=1
sinon si angleRayon > 180 et angleRayon < 270 :
        angle = angleRayon-180
        adjacent = basDroit.y – P1.y
        opposé = adjacent * tan(angle)
        P2.x = P1.x – opposé
        si P2.x > hautGauche.x :
                P2.y = basDroit.y
                CaseAtester.rangée += 1
        sinon
                Angle = 270 * angleRayon
                Adjacent = P1.x – hautGauche.x
                Opposé = adjacent * tan(angle)
                P2.y – P1.y + opposé
                P2.x = hautGauche.x
                caseAtester.colonne -= 1
sinon :
        angle = 360 – angleRayon
        adjacent = P1.y – hautGauche.y
        opposé = adjacent * tan(angle)
        P2.x = P1.x – opposé
        si P2.x > hautGauche.x :
                P2.y = hautGauche.y
                caseAtester.rangée -= 1
        sinon :
                angle = angleRayon – 270
                adjacent = P1.x  - hautGauche.x
                opposé = adjacent * tan(angle)
                P2.y = P1.y – opposé
                P2.x = hautGauche.x
                caseAtester.colonne -= 1
```

Distorsion due à l'écran

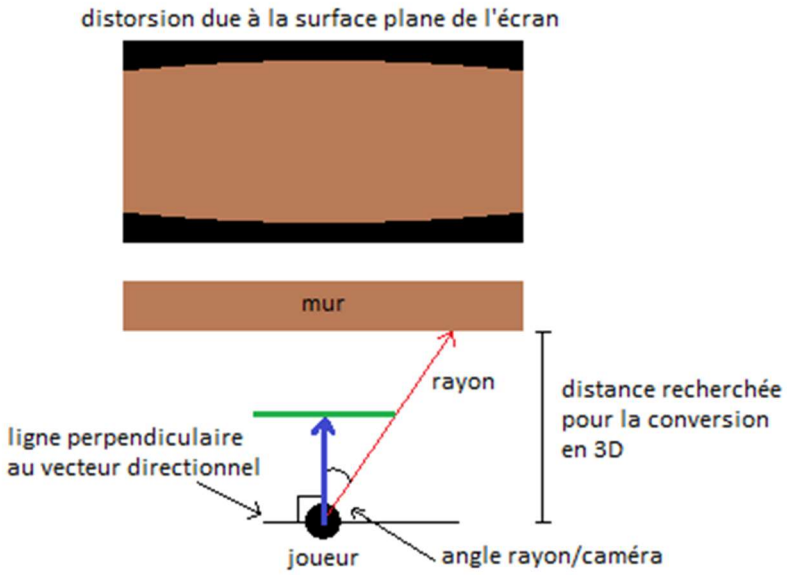

Figure 8 : Distorsion de l'image 3D causée par la différence entre la surface plane d'un écran et la nature sphérique de l'œil humain.

Maintenant que nous avons la longueur de notre rayon, il reste une dernière étape avant de pouvoir effectuer notre conversion 3D. La surface de l'œil humain est ronde. Un rayon lumineux partant du mur et se rendant à notre œil serait perçu correctement. Cependant, les écrans de nos ordinateurs sont plats. Si nous n'effectuons pas de correction, nos murs seront distordus comme dans l'image ci-dessus. C'est ce qu'on nomme l'effet « œil de poisson ». Pour corriger cette déformation, il faut imaginer une ligne perpendiculaire au vecteur directionnel et croisant le joueur. Nous recherchons la distance entre cette ligne et le point d'intersection du rayon et du mur.

Question : Comment calculer cette distance ? Indice : nous avons la longueur de l'hypoténuse (notre rayon) et l'angle que fait le rayon avec notre ligne perpendiculaire (90 – angle entre le rayon et le vecteur directionnel).

Distance = longueur du rayon * cos(angle rayon/caméra)

Augmentation de l'effet 3D par variations des couleurs

Pour accentuer l'effet 3D de l'engin il faut faire varier légèrement l'intensité de la couleur des murs. Si un rayon frappe une case sur l'axe X la couleur originale sera conservée. S'il frappe sur l'axe Y, l'intensité de la couleur sera augmentée. Pour cette raison, toutes les couleurs des murs devront être codée en RVB. Voir la figure 1 de ce document pour un visuel de l'effet.

Conversion en colonne de pixels

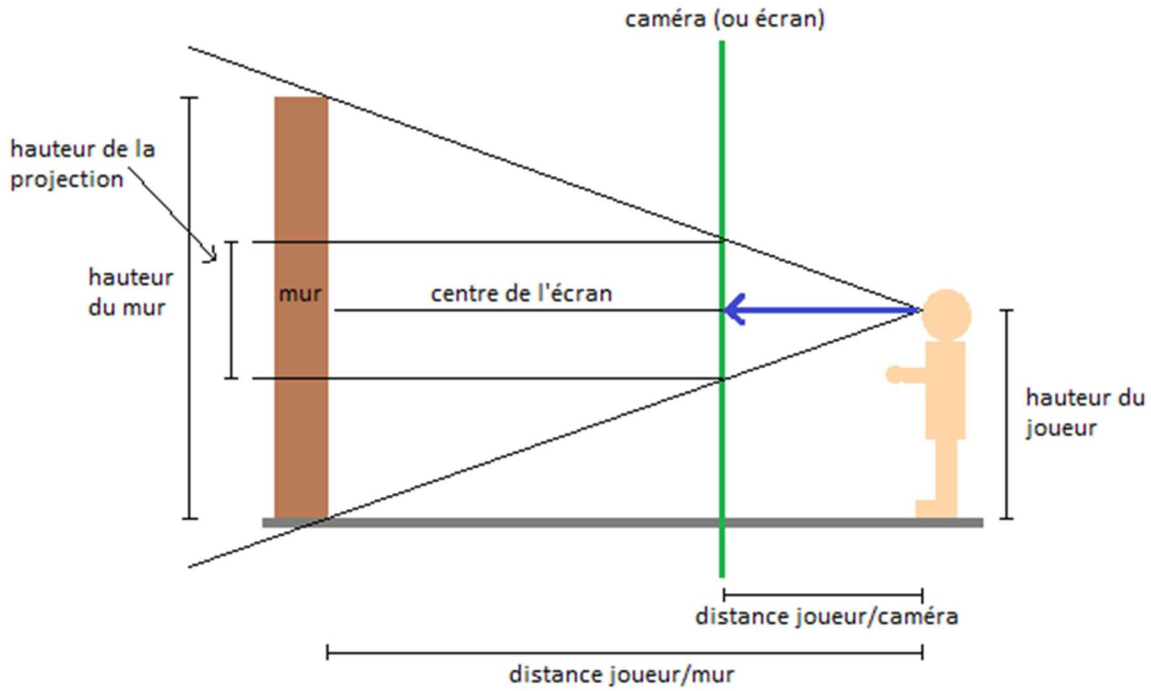

Figure 9 : Vue de profil des éléments entrant dans la conversion d'un rayon en colonne de pixels.

Quelques observations sur le dessin ci-haut :

- Le centre du mur arrive (et arrivera toujours) au centre de l'écran.
- Nous connaitrons la hauteur de la projection et sa position dans notre colonne de pixels (au centre) nous pourrons facilement déterminer quelle portion de la colonne sera de la couleur du plancher et laquelle sera de celle du plafond.
- Si on prend les yeux du joueur comme sommet, le triangle formé par la hauteur du mur et celui formé par la hauteur de la projection sont semblables (leurs angles sont égaux).

Une règle mathématique dit que si deux triangles sont semblables alors les ratios entre les longueurs de leurs côtés correspondants sont égaux :

Hauteur mur / distance joueur et mur = hauteur projection / distance joueur et caméra

Par transformation :

Hauteur projection = (hauteur mur / distance joueur et mur) * distance joueur et caméra

- Hauteur des murs : nous débuterons par une hauteur de 2*largeur d'une case
- Distance joueur/caméra : la longueur de notre vecteur directionnel
- Distance joueur/mur : la longueur de nos rayons après correction de la distorsion

Évolution de l'ensemble des rayons

Maintenant que nous savons comment calculer une colonne de pixels. Il nous reste à déterminer en combien de colonne nous diviserons notre affichage 3D afin de former l'ensemble de notre image. Rappel de quelques éléments connus ainsi que quelques déductions :

- La largeur de la vue 2D est de 1280px.
- La grille aura 32 colonnes.
- Chaque case fait donc 40px * 40px.
- La surface de notre caméra est de quatre fois celle d'une case, donc 160 pixels.
- La largeur de notre fenêtre 3D est de 800px.
- Si nous voulons tracer un rayon par pixel de notre caméra les colonnes de la vue 3D feront 800/160 = 5 px de largeur.
- Notre champs visuel (FOV) est de 60^0
- L'angle d'écart entre chacun de nos rayons sera de 60^0 / 160px.

La distance entre les points de croisement rayon/caméra et caméra/vecteur de direction sera conservée dans une variable. Au lieu d'avoir à la recalculer par trigonométrie pour chaque nouveau rayon nous pourrons l'ajuster par addition (de 1px). Il en va de même pour l'angle des rayons

Le premier rayon sera tracé à l'extrémité gauche de l'objectif de la caméra. Son angle sera donc de 325^0. La distance entre le point de croisement rayon/caméra et le point caméra/vecteur directionnel sera de 2 cases. À chaque évolution cette distance diminuera de 1 et l'angle augmentera de 60/160 degré. Lorsque la distance atteindra 0 elle augmentera de 1 jusqu'à atteindre à nouveau 2 cases. L'angle du rayon sera remis à 0^0 et se rendra jusqu'à 60^0.

Déplacements verticaux et horizontaux du joueur.

Maintenant que la rotation de la caméra et la conversion 2D/3D sont implantées nous pouvons nous occuper des déplacements du joueur. **Important** : Avant de bouger la grille ne pas oublier de vérifier si le mouvement du joueur est possible en procédant à la détection des collisions avec les murs.

Si le déplacement est possible, cela se fait très simplement en effectuant une translation de la carte 2D. Tous les éléments de celle-ci, soit grille et cases, devront avoir une fonction permettant de changer leurs positions. Pour bien imager les translations voyons ce qui se produit avec la grille si l'angle de la caméra est de 0^0 (le joueur fait face au nord) :

- Déplacements vers l'avant = augmentations des positions y
- Déplacements vers l'arrière = diminution des positions y
- Déplacement vers la gauche = augmentation des positions x
- Déplacement vers la droite = diminution des positions x

On peut observer que la grille bouge dans la direction inverse du déplacement du joueur. Voyons maintenant comment calculer les mouvements si on change l'orientation de la caméra.

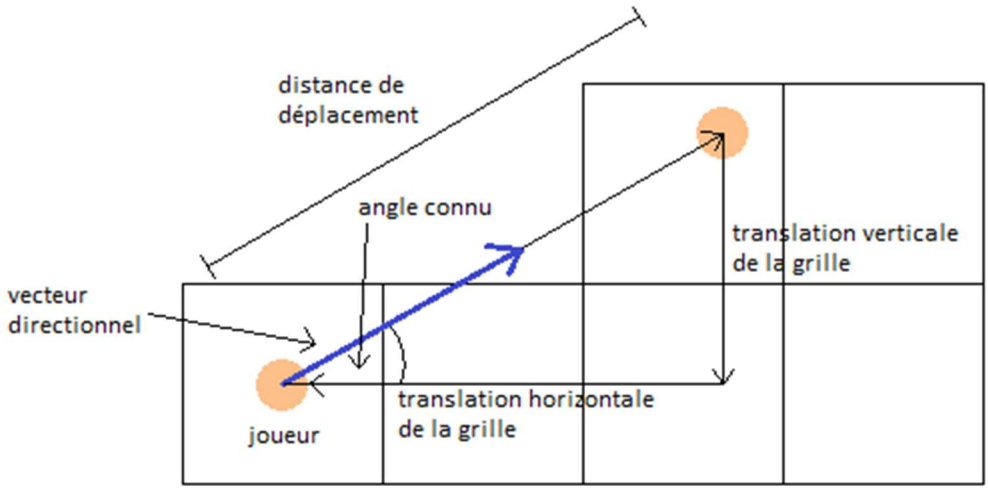

Figure 10 : Mouvements du joueur

Question : Nous avons un angle connu et la valeur de l'hypoténuse qui correspond à la distance de déplacement. Quel serait l'algorithme pour calculer les mouvements verticaux et horizontaux de la grille pour un déplacement vers l'avant?

```
Déplacement.x = 0
Déplacement.y = 0
Si angleCaméra == 0 :
        Déplacement.y = distanceDéplacement
Sinon si angleCaméra == 90
        Déplacement.x = - distanceDéplacement
Sinon si angleCaméra == 180 :
        Déplacement.y = - distanceDéplacement
Sinon si angleCaméra == 270 :
        Déplacement.x = distanceDéplacement
Sinon si angleCaméra > 0 et angleCaméra < 90 :
        Angle = 90 – angleCaméra
        Déplacement.x = - distanceDéplacement * cos(angle)
        Déplacement.y = - distanceDéplacement * sin(angle)
Sinon si angleCaméra > 90 et angleCaméra < 180 :
        Angle = 180 – angleCaméra
        Déplacement.x = - distanceDéplacement * sin(angle)
        Déplacement.y = - distanceDéplacement * cos(angle)
Sinon si angleCaméra > 180 et angleCaméra < 270 :
        Angle = 270 – angleCaméra
        Déplacement.x = distanceDéplacement * cos(angle)
        Déplacement.y = - distanceDéplacement * sin(angle)
Sinon :
        Angle = 360 – angleCaméra
        Déplacement.x = distanceDéplacement * sin(angle)
        Déplacement.y = distanceDéplacement * cos(angle)
```

Les déplacements vers l'arrière et les glissements de côté se font sur le même principe. Il suffit de modifier l'angle du vecteur de mouvement par tranche de 90^0 à partir de l'angle de la caméra. Par exemple on se déplace vers l'arrière en ajoutant 180^0 à note angle de caméra.

Conception des cartes

Maintenant que notre démo fonctionne pour une carte ayant la largeur et la hauteur de la fenêtre 2D, nous pouvons en concevoir de plus grandes et également les transférer dans des fichiers externes au code. Pour ce faire nous utiliseront Excel. Il est important que nos grilles soient rectangulaires. Elles doivent avoir au minimum 32x18 cases mais n'ont pas de maximum. Le logiciel fera la lecture des rangées et des colonnes tant qu'il y aura des valeurs à l'intérieur. Idem pour le nombre de couleurs qui peuvent être utiliser pour les murs. Notre classeur aura 3 feuilles :

- La première sera utilisée pour concevoir la carte à l'aide de cases dont les couleurs seront celles de nos murs ou du plancher.
- La deuxième contiendra les informations nécessaires à la conversion de notre carte colorée en grille numérotée. La colonne 1 contiendra toutes les couleurs que nos murs pourraient avoir. La colonne 2 contiendra l'équivalent en valeur numérique. La colonne 3 contiendra le code RGB de la couleur de la première colonne afin que notre logiciel puisse la reproduire. Les nombres 0 et 100 devront être définis pour les couleurs des planchers et du plafond
- La troisième contiendra une grille semblable à celle de la première feuille mais les cases n'auront pas de couleur et contiendront uniquement les conversions en valeurs numériques.

Cartes plus grandes que la fenêtre 2D

Avant d'afficher la carte de jeu il faudra maintenant tenir compte du fait qu'une certaine partie de celle-ci pourrait se trouver à l'extérieur de l'écran. Les méthodes qui s'occupent de dessiner cette carte devront donc aller chercher uniquement les éléments de la grille qui seront affichés.

Question : En sachant que chaque case est composée de points haut-gauche et bas-droit comment pouvons-nous identifier les éléments à afficher?

Première colonne : Il faut itérer au travers les cases d'une rangée de la grille et trouver la première dont la valeur hautGauche.x est supérieure ou égale à 0. Si celle-ci n'est pas égale mais supérieure, il faut trouver notre point x = 0 dans la précédente. Toutes les premières cases de chaque rangée seront alors partiellement dessinées à partir de ce point x.

Dernière colonne : Il faut itérer au travers les cases d'une rangée pour trouver celle dont la valeur basDroit.x est supérieure ou égale à la largeur de la fenêtre 2D.

Première rangée : Il faut itérer au travers les cases d'une colonne pour trouver celle dont la valeur hautGauche.y est supérieure ou égale à 0.

Dernière rangée : Il faut itérer au travers les cases d'une colonne pour trouver celle dont la valeur basDroit.y est supérieure ou égale à la hauteur de la fenêtre 2D.

Cas particuliers : Il faut valider si les premières/dernières rangées/colonnes sont déjà supérieures à 0 ou inférieures à la largeur/hauteur de la fenêtre. Si c'est le cas cela signifie qu'une partie de la grille débute ou se termine à distance du bord correspondant de la fenêtre.

Bonus #2 (optionnel)

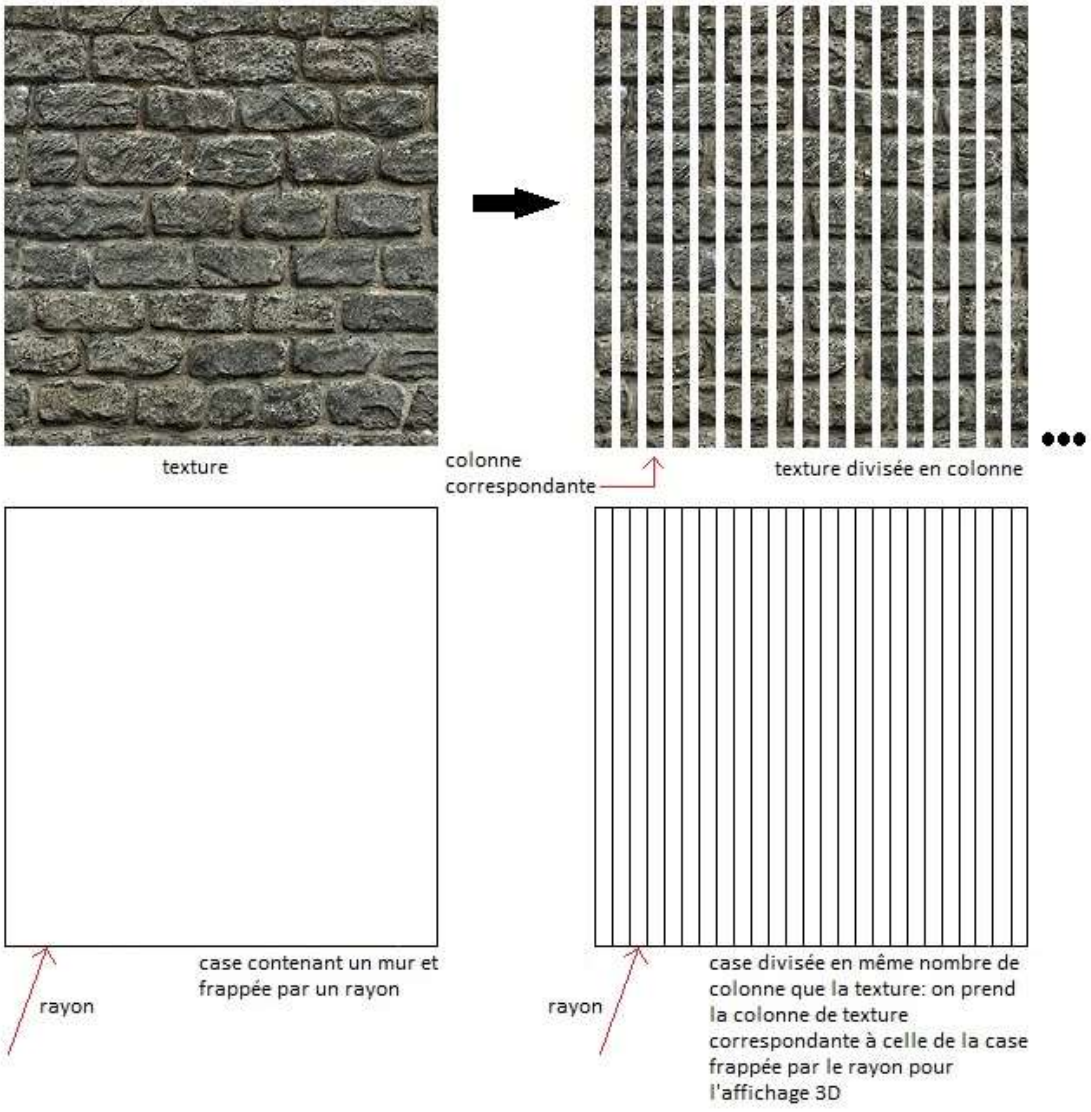

Figure 11 : Ajout de textures aux murs

Dans le jeu *Wolfenstein* 3D qui a inspiré ce projet, les murs sont dessinés à l'aide de textures. Connaissant la largeur et la hauteur (en pixels) d'un mur « pleine grandeur », il est possible de concevoir des textures ayant exactement les mêmes dimensions. Si vous désirez faire ce bonus ne pas oublier de modifier les documents Excel des cartes de jeu pour que les valeurs numériques de la grille correspondent à des fichiers de textures.

Pour dessiner la partie des murs dans les colonnes de pixels de l'affichage 3D il faudra diviser la texture en sections verticales de la même largeur que nos colonnes de pixels. Chacune d'elles deviendra une image en mémoire et pourra être étirée ou rapetissée pour correspondre à la hauteur du mur projetée à l'écran. Il est préférable de charger les images de chaque division de texture en mémoire dès le début d'un niveau afin d'évider d'avoir à faire des traitements d'images chaque fois qu'on affiche une colonne de pixel.

Références

Note : La recherche a été effectuée entièrement sur Internet. Voici tous les liens pertinents consultés :

https://lodev.org/cgtutor/raycasting.html

https://permadi.com/1996/05/ray-casting-tutorial-8/

https://en.wikipedia.org/wiki/Ray_casting

https://www.playfuljs.com/a-first-person-engine-in-265-lines/

https://dev.opera.com/articles/3d-games-with-canvas-and-raycasting-part-1/

https://gamedevhowto.blogspot.com/2017/09/a-diversion-raycast-engine-built-in-c.html

https://forum.unity.com/threads/raycasting-engine-experiment.525347/

https://www.youtube.com/watch?v=eOCQfxRQ2pY&t=122s

Figure 1 :

https://github.com/aruvham/javascript-ray-casting

Figure 11 – la texture :

http://mirror.splatterladder.com/wolfplayer.nl/www.wolfplayer.nl/tutorials/custom_text/custom_text.html