Compte rendu projet Portal 0.0

BIDAULT Matthieu, BADSTÜBER Elian, FOCHEUX Vital

March 17, 2024

Remerciements

Nous tenons à remercier notre enseignant de projet, M. Bernard qui a su nous guider et nous conseiller tout au long de ce projet.

Table des matières

1 Introduction **4**

2 Besoins et objectifs du projet **4**
- 2.1 Contexte . . . 4
- 2.2 Motivations . . . 7
- 2.3 Objectif et contraintes . . . 7

3 Développement **8**
- 3.1 Des technologies nouvelles ou presque . . . 8
- 3.2 Différentes stratégie pour le rendu 3D . . . 8
  - 3.2.1 Raycasting, l'approche historique . . . 8
  - 3.2.2 Line Of Sight, l'approche moderne . . . 10
- 3.3 Détails du développement . . . 11
  - 3.3.1 Les cartes: . . . 12
  - 3.3.2 Les murs: . . . 12
  - 3.3.3 Les collisions: . . . 14
  - 3.3.4 La boucle de jeu: . . . 17
  - 3.3.5 Le rendu des murs: . . . 17

4 Spécifications **19**

5 Gestion du projet **19**
- 5.1 L'équipe . . . 19
- 5.2 Planification et outils de gestion . . . 19
- 5.3 Répartition des tâches . . . 19

6 Conclusion technique & personnelle **19**
- 6.1 Bilan technique . . . 19
  - 6.1.1 Résultats obtenus . . . 19
- 6.2 Bilan personnel . . . 19
- 6.3 Perspectives . . . 19
  - 6.3.1 Améliorations possibles . . . 19
  - 6.3.2 Nouvelles fonctionnalités . . . 20
  - 6.3.3 Un plus dans nos CV . . . 20

Figure 1: Knights of the Sky

Figure 2: Hovertank 3D

1 Introduction

Portal 0.0 est un jeu vidéo de type puzzle à la première personne. Il est une version très simplifiée du jeu Portal[13] qui utilise un moteur de type Raycaster à la façon du jeu Wolfenstein 3D[8] pour afficher les éléments du jeu.

L'objectif de ce projet est de réaliser un jeu vidéo en C++ avec la bibliothèque Gamedev Framework[1] (GF) afin d'occuper M. Bernard lorsqu'il ne veux pas corriger les copies de ses élèves.

Lors de ce rapport, nous allons détailler les différents aspects conceptuels et techniques de ce projet. Nous commencerons par présenter les besoins, les contraintes et les objectifs du projet. Ensuite, nous détaillerons différentes phases de développement du projet, comme les stratégies envisageables pour le rendu 3D. Nous continuerons dans les spécifications techniques du projet en détaillant la façon dont les rendus 3D on été réalisés. Nous parlerons ensuite rapidement de notre politique de gestion de projet. Enfin, nous terminerons par une conclusion sur le travail réalisé et les perspectives d'avenir.

2 Besoins et objectifs du projet

2.1 Contexte

Les graphismes : Au début des années 90, la société Id Software[19] a entrepris des recherches pionnières dans le domaine des graphismes 3D, alors principalement réservés aux simulateurs de vols tels que Wing Commander[12] ou Knights of the Sky[3] (Knights of the Sky), deux titres parus en 1990. Face aux contraintes de performance des ordinateurs de l'époque, le développement de jeux d'action en 3D rapide représentait un défi de taille.

Figure 3: Ultima Underworld

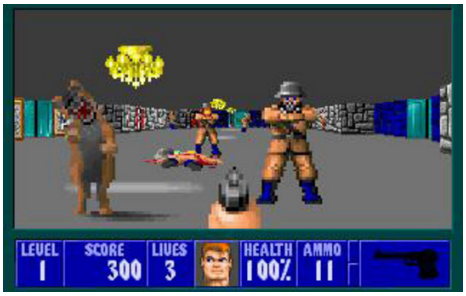

Figure 4: Wolfenstein 3D

Figure 5: Doom (1993)

Figure 6: Quake (1996)

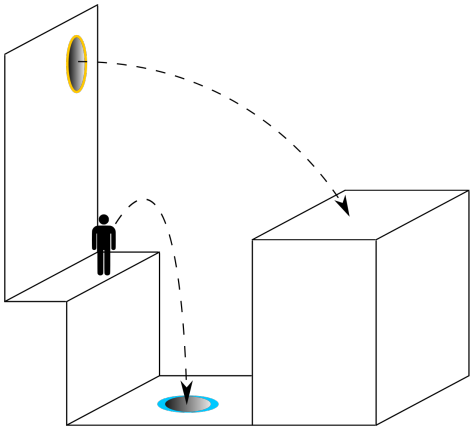

Figure 7: Schéma de fonctionnement du système de portail

C'est dans ce contexte que John Carmack[14] a proposé l'utilisation de la technique du raycasting[16], permettant de calculer uniquement les surfaces visibles par le joueur. En six semaines, Carmack développe un moteur 3D innovant utilisant des sprites 2D pour représenter les entités du jeu. Ce moteur a été utilisé dans le jeu Hovertank 3D[7] (Hovertank 3D), publié en avril 1991. À l'automne 1991, alors que John Carmack[14] et John Romero[15] finalisaient le moteur de Commander Keen in Goodbye, Galaxy[6], Carmack découvre Ultima Underworld[4] (Ultima Underworld), un jeu développé par Blue Sky Productions[18] (qui deviendra plus tard Looking Glass Studios), doté d'un moteur capable de rendre des graphismes 3D texturés sans subir les limitations de Hovertank 3D. Inspiré, Carmack décide d'améliorer son propre moteur pour intégrer le mapping de textures tout en conservant de hautes performances. Après un intense travail de six semaines, le nouveau moteur 3D est achevé et utilisé pour le jeu Catacomb 3D[5], publié en novembre 1991. La révélation de Catacomb 3D a poussé Scott Miller[17] d'Apogee à convaincre l'équipe de développer un jeu d'action en 3D sous forme de shareware. Cela a conduit au lancement du projet Wolfenstein 3D[8] (Wolfenstein 3D), un remake en 3D de Castle Wolfenstein[11]. Sorti le 5 mai 1992 sur PC, ce jeu a non seulement été un succès commercial mais a également posé les bases du genre du jeu de tir à la première personne, préfigurant ainsi des titres légendaires tels que Doom[9] (Doom (1993)) et Quake[10] (Quake (1996)).

Le système de jeu : En 2007, Valve Corporation[20] a révolutionné le genre des jeux de réflexion à la première personne avec la sortie de Portal[13]. Ce jeu introduit un mécanisme unique permettant au joueur de générer deux portails, l'un orange et l'autre bleu, sur des surfaces planes et interconnectées. Ces portails offrent la possibilité de traverser instantanément l'espace d'un point à un autre, tout en conservant l'inertie. L'objectif est de résoudre divers puzzles en se servant de cette capacité à manipuler l'espace pour atteindre la sortie des différents niveaux proposés.

2.2 Motivations

L'une des motivations principales de ce projet est de réaliser un jeu vidéo avec graphique comme à l'époque de Wolfenstein 3D[8] mais avec des concepts de jeux plus récentes et qui plus est pourrai être un préquel de Portal[13].

2.3 Objectif et contraintes

Les principaux objectifs de ce projet sont :

- Réaliser un jeu vidéo en C++ avec la bibliothèque GF
- Implémenter un moteur de type Raycaster
- Implémenter un système de portail
- Implémenter un système de collision
- Offir une expérience de jeu simple et agréable à jouer.

Mais avec des contraintes :

- L'apprentissage d'un langage de programmation nouveau pour nous
- L'apprentissage d'une bibliothèque de programmation nouvelle pour nous
- L'apprentissage de la programmation d'un moteur de type Raycaster
- L'apprentissage de la programmation d'un système de portail
- L'apprentissage de la programmation d'un système de collision
- L'apprentissage de la programmation d'un système de jeu

3 Développement

3.1 Des technologies nouvelles ou presque

Programmer en C++ a pu s'avérer assez techniques car cela était un tout nouveau langage de programmation à apprendre pour nous car il n'avait jamais été abordé auparavant lors de notre parcours universitaire. Une des plus grandes difficultés de ce projet était de développé avec la bibliothèque GF. En effet, quand bien même c'est un bibliothèque possédant un bon nombre de fonctions pouvant permettre le développement graphique 2D. Elle reste néanmoins parfois limité pour les besoins techniques du projet, mais grâce à cela, cela permet de donner des idées de méthodes à rajouter dans GF voire dans GF2.[2]

3.2 Différentes stratégie pour le rendu 3D

Pour implémenter notre jeux, l'un des points les plus importants était de pouvoir afficher des murs en 3D. Pour cela, nous partions d'une grille de cellules représentant les emplacements des murs donc d'un environnement 2D pour aller vers un rendu 3D.

3.2.1 Raycasting, l'approche historique

La première approche envisagée était celle du raycasting historique.

Historiquement, le raycasting est une méthode de rendu graphique utilisée pour créer une perspective 3D à partir d'un environnement 2D. Cet effet est réalisé en projetant des rayons depuis la position du joueur à travers la scène pour déterminer les intersections avec les murs. Pour optimiser ce processus, l'algorithme Digital Differential Analyzer (DDA) a été utilisé.

Le raycasting génère une illusion de profondeur en projetant des rayons depuis le point de vue du joueur jusqu'à ce qu'ils rencontrent un objet dans l'espace de jeu. Le processus est répété pour chaque colonne de l'écran, permettant ainsi de dessiner une image 3D à partir d'une carte 2D. Les distances calculées entre le joueur et les points d'intersection sont utilisées pour déterminer la hauteur des objets rendus, créant ainsi une perspective en fausse 3D.

Expliquons plus en détail cet algorithme DDA : L'algorithme DDA est historiquement conçu pour la rasterisation de lignes, c'est-à-dire la conversion de lignes géométriques en ligne visuelle composées de pixels alignés. Dans le contexte du raycasting, DDA est employé pour déterminer où les

rayons projetés à travers la scène intersectent avec les objets de l'environnement, typiquement et dans notre cas représenté par une grille de cellules.

Plutôt que de calculer chaque point le long d'une demie droite en utilisant des formules de géométrie directe, ce qui pourrait être coûteux en termes de performances, DDA avance par petits incréments. Cela signifie que pour chaque pas sur l'axe le plus dominant (x ou y), DDA calcule l'emplacement correspondant sur l'autre axe en ajoutant un incrément constant. Cela se traduit par un parcours régulier et efficace le long de la ligne en faisant des sauts à chaque intersection entre la demie droite et la grille 2D.

Dans le raycasting, DDA est utilisé pour trouver rapidement et précisément les intersections entre les rayons et les murs.

Les avantages de DDA dans ce contexte :

- **Efficacité :** DDA réduit la quantité de calculs nécessaires pour trouver les intersections. En progressant par incréments réguliers, l'algorithme évite les calculs répétés ou complexes typiques des méthodes géométriques standards.
- **Précision :** Bien que simple, DDA maintient une précision élevée dans le calcul des intersections. Chaque étape avance de manière incrémentielle, garantissant que le rayon suit précisément sa trajectoire prévue sans sauter de pixels ou de cellules.
- **Adaptabilité :** L'algorithme s'adapte bien aux grilles de toutes tailles, ce qui le rend idéal pour les jeux avec différentes résolutions et des espaces de jeu variés.
- **Simplicité et Robustesse :** DDA est relativement simple à implémenter et à comprendre, ce qui réduit le risque d'erreurs. Sa robustesse en fait une méthode fiable pour le raycasting dans des environnements de jeu variés.

Dans l'application concrète DDA au raycasting, l'algorithme calcule les pas nécessaires pour que le rayon traverse une cellule de la grille, soit en horizontal, soit en vertical. Ces pas sont basés sur la direction du rayon et sont ajustés à chaque étape pour correspondre à la grille du jeu. Cela permet au rayon de "marcher" à travers la grille, cellule par cellule, en vérifiant à chaque pas si un mur a été rencontré.
Cette approche est assez simple à implémenter, promettant par ailleurs des performances élevées et une implémentation de la vue à travers les portails

plus aisée. Cependant, cette simplicité est peut être ce qui nous a fait choisir une autre approche, plus adaptée à un projet d'étudiants en 3ème année de licence informatique et plus moderne.

3.2.2 Line Of Sight, l'approche moderne

Cette seconde approche s'approche des algorithme *Line Of Sight* et consiste à calculer la distance entre le joueur et chaque sommet de chaque mur. Pour ce faire on à pu récupérer l'algorithme DDA précédemment utilisé pour le rendu des murs mais cette fois-ci en visant chaque sommets au lieu de viser chaque colonne de pixel. Cela permet théoriquement de réduire considérablement le nombre de calculs à effectuer.

Cependant, des problèmes ont été rencontrés, les voici avec leur solution :

- **Viser un sommet avec DDA :** Si DDA est efficace et précis cela peut être la cause de certain problèmes. En effet, si l'on vise un sommet de mur avec un rayon, la moindre approximation dans le calcule de l'angle avec lequel on lance le rayon ferais rater le sommet en moyenne une fois sur deux.

  Solution : Pour remédier à cela, nous avons ajuster DDA pour qu'à chaque fois qu'un pas passe assez proche d'un sommet, si l'une des quatre cellules adjacentes est un mur alors on considère que le rayon à toucher le sommet.

- **Continuer le rayon après avoir touché un sommet :** Si l'on touche un sommet avec un rayon, il faut déterminer si le rayon doit continuer ou non. Cela est important dans le cas ou l'on est au bord d'un mur est qu'il faut faire le rendu du mur se trouvant derrière

  Solution : Pour déterminer si le rayon doit continuer ou non, il faut regarder si dans les cellules adjacentes au sommet touché, il y a exactement un mur et que ce mur ne se trouve pas dans la direction du rayon. Si c'est le cas alors on lance un nouveau rayon partant du sommet touché dans la même direction que le rayon précédent.

- **Récupérer les sommets touchés dans le bon ordre :** Pour afficher tous les murs correctement, nous avons rangé les sommets dans l'ordre croissant de leur angle par rapport au joueur. Cela permet d'afficher les murs par segments en prenant les sommets deux par

deux. Cependant, dans certains cas, le premier sommet dois être relié au dernier sommet.

Solution : Pour identifier ces cas, nous regardons si dans la liste des sommets, il y a au moins un segment qui en contient un autre. Si c'est le cas alors on lie le premier sommet au dernier.

- **Impossibilité de générer les murs à plus de 180° :** Lorsque l'on a envoyé tous les rayons, afin de générer tous les murs visibles par le joueur, si l'on essaie de projeter un mur se trouvant derrière le joueur sur le plan de projection, alors la projection se retrouve de l'autre coté du plan en symétrie par rapport au joueur. Concrètement, cela signifie que la carte graphique ne peux pas gérer ces mur les affichant alors devant le joueur.

  Solution : Pour régler ce problèmes, il faut créer une droite imaginaire passant par le joueur et orthogonale à la direction dans laquelle le joueur regarde. Ensuite, il faut retirer tous les segments qui se trouvent de l'autre coté de cette droite. Puis, il faut ajouter un sommet à l'intersection de cette droite avec chaque segment l'intersectant, en plaçant ces points, légèrement décalé de l'intersection entre la droite et le segment, car sinon il serait impossible de projeter ce point sur le plan de projection, le plan étant lui même parallèle à la droite. Enfin, on peut générer les murs comme précédemment.

- **Afficher des textures sur les murs :**Pour afficher des murs texturés, il faut pouvoir mapper une texture sur un trapèze en deux dimensions. Cependant il ne suffit pas d'afficher une texture coupé en deux triangles. En effet, il faut que la déformation soit correcte du point de vue de la perspective et ce problème est loin d'être trivial.

  Solution : Ce problème étant très complexe à régler, nous avons décider de choisir une autre méthode qui consiste à couper le trapèze en six triangles choisis de telle sorte que la déformation ne soit pas trop flagrante. Cela dit, cette méthode empêche l'utilisation de textures avec des motifs réguliers et principalement horizontaux.

3.3 Détails du développement

Dans cette partie nous allons détailler les différentes étapes de développement de notre projet avec la seconde approche. En commençant par la partie la moins technique pour finir par la partie la plus technique. Il faut noté que détails du développement dans les parties qui vont suivre sont pour un rendu

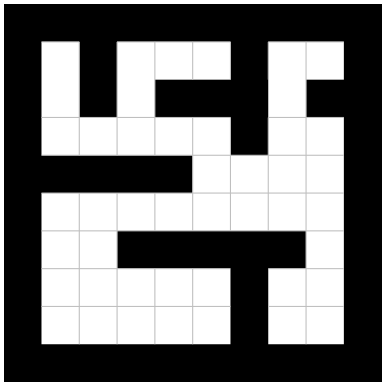

Figure 8: PNG d'une carte

3D, mais tout au long de ce projet les différentes étapes ont été tester sur un rendu 2D pour faciliter le développement.

3.3.1 Les cartes:

Afin de pouvoir afficher des cartes, nous avons dû créer un fichier PNG[1] où nous dessinons les murs ainsi que la case de départ et d'arrivée (PNG d'une carte). Lors du lancement du jeu, le programme lit le fichier PNG et enregistre les différentes cases dans un tableau à deux dimensions. En même temps, le programme va regarder si les coordonnées correspondent à une cellule de mur, la case de départ, la case d'arrivée ou aucune des deux. Si c'est une cellule de mur, alors on va effectuer un parcours en profondeur des coordonnées adjacentes pour savoir si ceux sont des cellules de murs ou non. Grâce à cela nous pouvons récuperer toutes les cellules d'un mur directement et ainsi compter le nombre de mur assez facilement.

3.3.2 Les murs:

Dans cette partie nous allons détailler les différentes étapes pour la création des murs.

Récupération des sommets utiles: Pour la création des murs, on effectue un boucle sur la liste de coordonnées des cellules occupées pour un mur précédemment récupérée. Sur chacune des coordonnées, on va regarder quatre de ses coordonnées adjacentes qui sont celles en haut, en bas, à gauche

[1] Portable Network Graphics

et à droite. Si l'une d'entre elles fait partie de la liste des coordonnées des cellules alors on va incrémenter un compteur. Si ce compteur est égal à 1 ou 3 alors on va l'ajouter dans une liste de coordonnées dites utiles, c'est-à-dire que ceux sont les coordonnées des sommets des murs.

Triage des sommets: Afin de faciliter le rendu des murs, nous avons besoin de trier les sommets. Pour cela, nous effectuons un boucle tant que la taille de la liste des sommets utiles est supérieur à la taille de la liste des sommets triés. Dans cette boucle, nous allons effectuer une boucle sur la liste des sommets utiles si le sommet est dans la liste des sommets triés ou non. Si il ne l'est pas alors on va récupérer les coordonnées de ce sommet et faire appel à une fonction pour trier à partir de ce sommet, la liste des sommets utiles et la liste des sommets triés. Dans cette fonction, on va effectuer une boucle sur la liste des sommets utiles à partir du sommets donné en paramètre. On va regarder pour chaque sommet si il existe un sommet dans la direction choisie ainsi que dans sa direction opposée.

- Si il existe un sommet dans la direction choisie et aucun sommet dans la direction opposée alors on va ajouter ce sommet dans la liste des sommets triés.
- Sinon si il existe un sommet dans la direction opposée et aucun sommet dans la direction choisie alors on va ajouter le sommet de la direction opposée dans la liste des sommets triés et changer la direction courante par la direction opposée.
- Sinon
  - Si il existe un sommet dans la direction choisie et un sommet dans la direction opposée alors on regarder si le sommet de la direction opposée n'est pas dans la liste des sommets triés et qu'un changement de direction doit être effectué ou bien que le sommet de la direction choisie est dans la liste des sommets triés et qu'un chanegement de direction ne doit pas être effectué alors on va ajouter le sommet de la direction opposée dans la liste des sommets triés et changer la direction courante par la direction opposée.
  - Sinon on va ajouter le sommet de la direction choisie dans la liste des sommets triés.

On va changer de direction en fonction de la direction courante, si le changement de direction reste le même que l'ancien alors on va changer de direction.

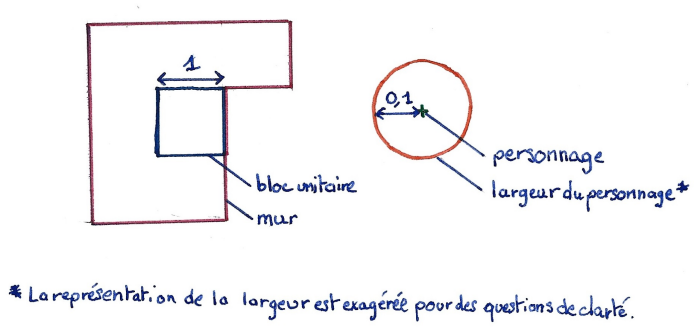

Figure 9: Collision cercle/rectangle

Si n'existe aucune sommet dans les directions choisies et opposées alors on va incrémenter un compteur. Si ce compteur est supérieur ou égal à 4 alors on arrête la fonction de triage.

Contruction des murs: Une fois les sommets utiles triés nous allons pouvoir construire les murs et les ajouter dans une liste de murs. Chaque murs de cette liste a pour attributs les coordonnées de ses sommets triés, les coordonnées de ses cellules occupées et la taille de son périmètre.

3.3.3 Les collisions:

Dans ce point nous allons traiter des collisions. Notre objectif est d'empêcher que notre personnage puisse passe dans ainsi qu'à travers les murs.

Aspect conceptuel: Le personnage est représenté par un point auquel nous accordons une largeur et donc un cercle centré autour de lui. Chaque mur est composé de blocs carrés unitaires. Ainsi, nous employons un système de collisions entre un cercle et un rectangle (cf: Collision cercle/rectangle). Ce système s'appuie sur une méthode dite du "clampage", qui permet de connaître le point du mur le plus proche et ainsi de savoir si notre cercle est en collision avec notre rectangle. Alors deux cas s'ouvre à nous.

Premier cas: Dans ce cas là, le cercle n'est pas en collision avec le rectangle, alors tout va bien et aucun traitement spécifique n'est à prévoir (cf: Pas de collision).

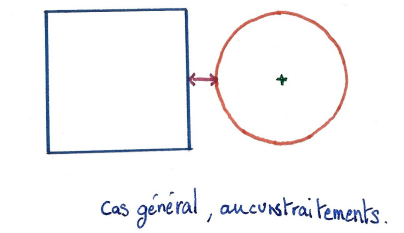

Figure 10: Pas de collision

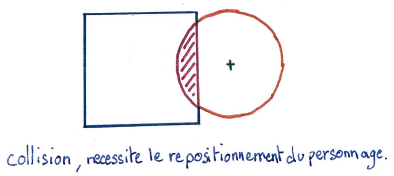

Figure 11: Collision

Second cas: Dans ce cas ci, le cercle est en collision avec le rectangle (cf: 'Collision' on page 15) et nous devons alors procéder à un repositionnement du personnage de manière à ce qu'il ne soit plus imbriqué avec le rectangle. Ce qui donne un effet visuel de glissement contre le rectangle.

Aspect technique: Nous définissons notre personnage, noté P, par des coordonnées représentant sa position ainsi qu'un nombre représentant sa largeur, noté L Ainsi :

$$P = ((x; y); l), x, y, l \epsilon R$$

$$L = 0, 1^{2}$$

La méthode de "clampage" consiste à prendre la valeur la plus proche d'une valeur donnée dans un intervalle donnée. Ce qui nous donne l'application suivante :

A partir de cette application, nous pouvons trouver le point le plus proche d'un mur. Nos murs sont composés de blocs unitaires, c'est-à-dire que ce sont des carrées de longueur de côté 1. Pour expliquer le procéder, nous prenons un de ces blocs, que l'on notera B, et qui est représenté par l'ensemble des coordonnées des ses sommets. Pour se repérer dans l'espace, il suffit d'une coordonnée pour avoir toutes les autres. Nous choisissons arbitrairement de prendre la coordonnée à la plus basse abscisse et plus basse ordonnée du bloc que nous noterons (a ; b). Nous avons alors :

$$B = \{(a; b); (a + 1; b); (a; b + 1); (a + 1; b + 1)\}$$

Que l'on simplifiera donc par

[2]Nous avons choisi cette valeur de manière empirique suite à une salve d'essais et de réflexions.

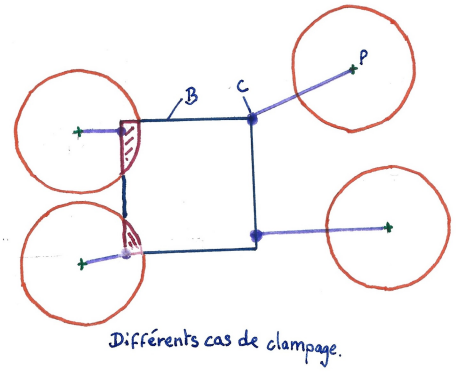

Figure 12: Clampage

$$B = (a; b)$$

Pour avoir le point le plus proche de B, noté C, nous devons alors "clamper" l'abscisse, resp. l'ordonnée, du personnage sur l'intervalle d'abscisse, resp. d'ordonnée, de B (cf: Clampage). Alors :

$$C = (clamp(P.x, B.a, B.a + 1); clamp(P.y, B.b, B.b + 1))$$

Grâce à C, nous pouvons maintenant vérifier si P est à moins de 0,1 de B, et donc en collision. Pour cela, calculons la distance entre P et C :

$$dist = \sqrt{((P.x - C.x)^2 + (P.y - C.y)^2)xxx}$$

Maintenant, nous pouvons généraliser notre réflexion à un mur pour nous offrir une légère optimisation. Nos murs sont composés de plusieurs blocs. En appliquant le traitement précédent à chaque bloc et en gardant la distance minimale, nous pouvons déterminer le point le plus proche de chaque mur cette fois ci. Ceci nous évite des comparaisons inutile par la suite. On peut alors définir DistMur :

$$DistMur = min(dist)$$

Ensuite, si distMur > P.l, c'est-à-dire si dist > 0,1, alors P n'entre pas en collision avec B (fig.2). Sinon, nous devons replacer P.

Dans ce second cas, nous devons connaître la nouvelle position de P. Il suffit alors d'ajouter la différence entre les coordonnées de P et de C :

$$P = (P.x + (P.x - C.x); P.y + (P.y - C.y))$$

Nous avons donc efficacement mis en place un système de gestion de collisions entre un cercle et un rectangle.

3.3.4 La boucle de jeu:

Dans cette partie nous allons détailler les différentes étapes de la boucle de jeu.

Events: Nous allons tous d'abord gérer la gestion des évènements. Pour cela nous allons effectuer une boucle sur les évènements et pour chaque bouton sur lesquels nous avons ajouté une action, nous allons effectuer cette action. Nous allons aussi gérer les fonctions de déplacement de la vision grâce à la souris avec la position relative de la souris. Ensuite, nous allons gérer la fonction d'envoie des portails en fonction du clic de la souris effectuer.

Update: Nous allons effectuer une update du jeu par rapport au temps écoulé depuis le dernier update. Cette update va permettre de mettre à jour (les attributs du joueur). De calculer la nouvelle position du joueur si il est en colision avec un mur. Et enfin calculer les nouveaux attributs du joueur si il est proche d'un portail lorsque les deux portails sont activés.

Render: Avant de faire le nouveau rendu des entités, nous allons d'abord effacer tous les rendus précédents. Ensuite nous allons effectuer le rendu de toutes les entités que nous allons détaillés dans la parties suivante.

3.3.5 Le rendu des murs:

Pour le rendu des murs, tous les sommets de chaque mur sont récupérés et triés selon leur angle par rapport au joueur. Ensuite, ces sommets sont parcourus dans ce nouvel ordre pour y envoyer un rayon grâce à DDA. Connaissant alors l'angle avec lequel le rayon est envoyé, la distance entre le joueur et le sommet est calculée en prenant le projeter orthogonale du sommet sur la droite de direction de vue du joueur (cf : Lancé d'un rayon). Avec ces deux informations, nous pouvons alors calculer le projeté du sommet sur le plan de projection. Chaque sommet de mur est, sur la carte en 2D, un point. Sur le plan de projection, chaque sommet est représenté par un segment vertical. Il faut donc, pour projeter un sommet sur le plan de projection, connaître deux points. Ces points étant verticalement alignés, il faut déterminer l'abscisse commun à ces deux points et leur ordonnée respectives. Les deux points étant une symétrie l'un de l'autre par rapport à la droite horizontale passant par le centre du plan, il nous faut connaître deux informations seulement : l'abscisse du sommet sur le plan et la distance entre le joueur et le sommet afin d'en déduire la taille du sommet (cf:

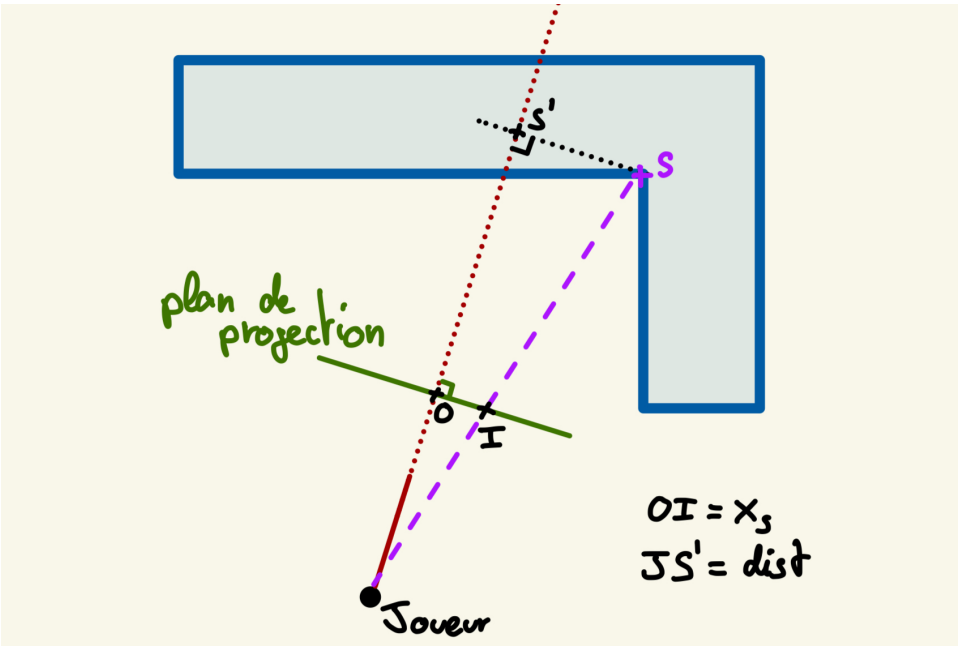

Figure 13: Lancé d'un rayon

Lancé d'un rayon) :

On pose (x_j, y_j) les coordonnées du joueur, (x_s, y_s) celles du sommet, W la largeur de la vue et H sa hauteur. On a alors :

Pour l'angle du sommet :

$$\theta = atan2(y_s - y_j, x_s - x_j)$$

la fonction $atan2(y, x)$ permet de calculer l'angle entre l'origine du plan et le point (x, y).

Pour la distance du sommet :

$$dist = \cos(\theta - \theta_{joueur})\sqrt{(x_s - x_j)(x_s - x_j) + (y_s - y_j)(y_s - y_j)}$$

Pour l'abscisse des deux points :

$$x = \frac{W \tan(\theta - \theta_{joueur})}{2}$$

Pour la hauteur du segment vertical :

$$h = \begin{cases} \frac{H}{2dist} & \text{si } dist \neq 0 \\ 2H & \text{sinon} \end{cases}$$

On obtient alors les deux points du segment représentant le sommet sur le plan comme suit :

$$P_1 = (x + \frac{W}{2}, \frac{H}{2} - \frac{h}{2})$$
$$P_2 = (x + \frac{W}{2}, \frac{H}{2} + \frac{h}{2})$$

4 Spécifications

5 Gestion du projet

5.1 L'équipe

5.2 Planification et outils de gestion

Pour la gestion du projet nous avons utilisé le versionneur github[21] qui est un outil de gestion de projet en ligne.

5.3 Répartition des tâches

6 Conclusion technique & personnelle

6.1 Bilan technique

6.1.1 Résultats obtenus

6.2 Bilan personnel

6.3 Perspectives

6.3.1 Améliorations possibles

La plus grande amélioration possible serait de pouvoir voir à travers les portails lorsque les deux sont activés. (Bref explication de comment cela serai possible)

6.3.2 Nouvelles fonctionnalités

Parmi de nouvelles fonctionnalités, que nous pourrions rajouter en voici quelques unes:

- Permettre à l'utilisateur de créer ses propres cartes
- Avoir des murs où il est impossible de tirer un portail
- Avoir des endroits au sol que lorsqu'on marche dessus, on est téléporter à la case départ
- Pouvoir mettre plusieurs textures différentes sur les murs
- Avoir des murs qui possèdent des trous permettant de voir ainsi que tirer des portails à travers

6.3.3 Un plus dans nos CV

References

[1] Julien BERNARD. Gamedev framework, 2024. https://gamedevframework.github.io/.

[2] Julien BERNARD. Gamedev framework 2, 2024. https://github.com/GamedevFramework/gf2.

[3] MicroProse. Knights of the sky, 1990. https://en.wikipedia.org/wiki/Knights_of_the_Sky.

[4] Blue Sky Productions. Ultima underworld, 1992. https://fr.wikipedia.org/wiki/Ultima_Underworld.

[5] ID Software. Catacomb3d, 1991. https://fr.wikipedia.org/wiki/Catacomb_3D.

[6] ID Software. Commander keen in goodbye, galaxy, 1991. https://en.wikipedia.org/wiki/Commander_Keen_in_Goodbye,_Galaxy.

[7] ID Software. Hovertank3d, 1991. https://fr.wikipedia.org/wiki/Hovertank_3D.

[8] ID Software. Wolfenstein3d, 1992. https://fr.wikipedia.org/wiki/Wolfenstein_3D.

[9] ID Software. Doom 1993, 1993. https://fr.wikipedia.org/wiki/Doom_(jeu_vid%C3%A9o,_1993).

[10] ID Software. Quake, 1996. https://fr.wikipedia.org/wiki/Quake.

[11] Muse Software. Castle wolfenstein, 1981. https://fr.wikipedia.org/wiki/Castle_Wolfenstein.

[12] Origin Systems. Wing commander, 1990. https://fr.wikipedia.org/wiki/Wing_Commander_(jeu_vid%C3%A9o).

[13] Valve. Portal, 2007. https://fr.wikipedia.org/wiki/Portal_(jeu_vid%C3%A9o).

[14] Wikipédia. John carmack. https://fr.wikipedia.org/wiki/John_Carmack.

[15] Wikipédia. John romero. https://fr.wikipedia.org/wiki/John_Romero.

[16] Wikipédia. Raycasting. https://fr.wikipedia.org/wiki/Raycasting.

[17] Wikipédia. Scott miller. https://fr.wikipedia.org/wiki/Scott_Miller_(programmeur).

[18] Wikipédia. Looking glass studios, 1990. https://fr.wikipedia.org/wiki/Looking_Glass_Studios.

[19] Wikipédia. Id software, 1991. https://fr.wikipedia.org/wiki/Id_Software.

[20] Wikipédia. Valve corporation, 1996. https://fr.wikipedia.org/wiki/Valve_Corporation.

[21] Wikipédia. Github, 2008. https://fr.wikipedia.org/wiki/GitHub.

4ème de couverture