Compte rendu projet Portal 0.0

BIDAULT Matthieu, BADSTÜBER Elian, FOCHEUX Vital

March 5, 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

Figure 1: Knights of the Sky

Figure 2: Hovertank 3D

I Introduction

Dans le cadre du projet semestriel de troisième année de licence informatique à l'université de Franche-Comté, nous proposons de coder une version très simplifiée du jeu Portal qui sera afficher grâce à un moteur de type Raycaster à la façon du jeu Wolfenstein 3D. Il a pour but de nous faire découvrir le monde du développement de jeux vidéo en nous faisant réaliser un jeu vidéo en C++ avec la bibliothèque Gamedev Framework (GF)[1].

II Besoins et objectifs du projet

II.i Contexte

Les graphismes : Au début des années 90, la société Id Software a entrepris des recherches pionnières dans le domaine des graphismes 3D, alors principalement réservés aux simulateurs de vols tels que Wing Commander ou Knights of the Sky (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. C'est dans ce contexte que John Carmack a proposé l'utilisation de la technique du raycasting, permettant de calculer uniquement les surfaces visibles par le joueur. En six semaines, Carmack développe un moteur 3D innovant utilisant des sprites 2D

[1]Tous au long de ce rapport, nous utiliserons abbrévation GF. Cela signifie Gamedev Framework.

Figure 3: Ultima Underworld

Figure 4: Wolfenstein 3D

Figure 5: Doom (1993)

Figure 6: Quake (1996)

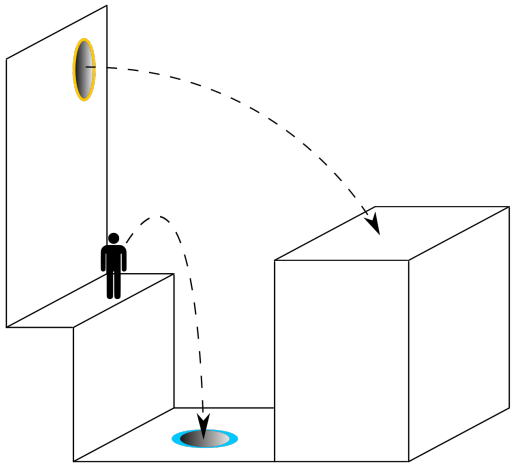

Figure 7: Schéma de fonctionnement du système de portail

pour représenter les entités du jeu. Ce moteur a été utilisé dans le jeu Hovertank 3D (Hovertank 3D), publié en avril 1991. À l'automne 1991, alors que John Carmack et John Romero finalisaient le moteur de Commander Keen in Goodbye, Galaxy, Carmack découvre Ultima Underworld (Ultima Underworld), un jeu développé par Blue Sky Productions (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, publié en novembre 1991. La révélation de Catacomb 3D a poussé Scott Miller 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 (Wolfenstein 3D), un remake en 3D de Castle Wolfenstein. 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 (Doom (1993)) et Quake (Quake (1996)).

Le système de jeu : En 2007, Valve Corporation a révolutionné le genre des jeux de réflexion à la première personne avec la sortie de Portal. 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.

II.ii Motivations

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

II.iii 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

III Gestion du projet

III.i L'équipe

III.ii Planification et outils de gestion

Pour la gestion du projet nous avons utilisé le site github qui est un outil de gestion de projet en ligne.

III.iii Répartition des tâches

IV Développement

IV.i Programmer en C++

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.

IV.ii Apprendre à utiliser la bibliothèque GF

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]

IV.iii 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.

[2]GF2 est un projet qui est la suite de GF en C++ ainsi qu'en python3.

Raycasting, l'approche historique

La première approche envisagée était celle historique : le raycasting.

Le raycasting est une méthode de rendu graphique utilisée pour créer une perspective 3D dans des environnements 2D. Cet effet est réalisé en projetant des rayons depuis la position de l'observateur à travers la scène pour déterminer les intersections avec les murs. Pour optimiser ce processus, nous avont utilisé l'algorithme Digital Differential Analyzer (DDA).

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.

Le principe fondamental de DDA repose sur l'itération linéaire. 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.

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 modifications sont importantes à faire pour que DDA fonctionne correctement dans ce contexte.

title En effet, si DDA est efficace et précis, 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 en moyenne un sommet sur deux.

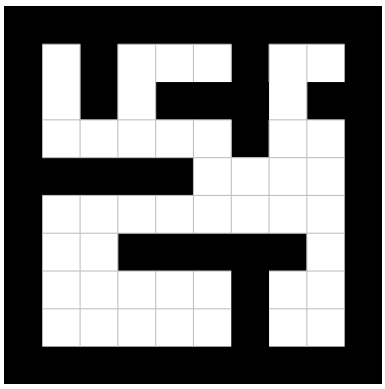

Figure 8: PNG d'une map

IV.iv Détails du développement

Dans cette partie nous allons détailler les différentes étapes de développement de notre projet avec la seconde approche. En commencençant 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 3D, mais tout au long de ce projet les différentes étapes ont été tester sur un rendu 2D pour faciliter le développement.

Les maps:

Afin de pouvoir afficher des maps[3], nous avons dû créer un fichier PNG[4] où nous dessinons les murs ainsi que la case de départ et d'arrivée (PNG d'une map). 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]En français les cartes
[4]Portable Network Graphics

Les murs:

Dans cette partie nous allons détaillés 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 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. (Explication de la fonction fctCanGo?)

- 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. 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.

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.

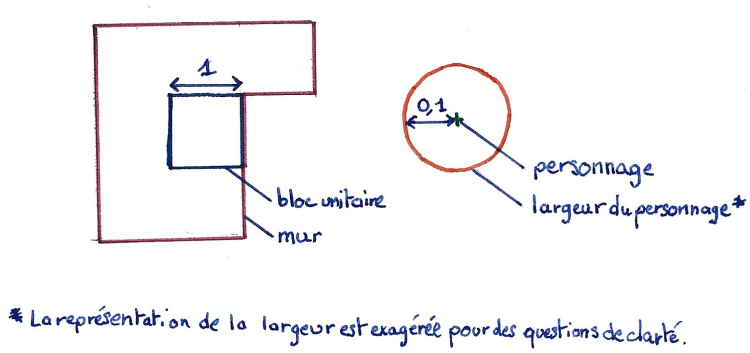

Figure 9: Collision cercle/rectangle

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

Les renders:

Ici explication de chaque rendu des différentes entités

Le rendu des murs: Pour le rendu des murs, nous allons effectuer une boucle sur la liste des sommets triés. Pour chaque sommet, nous allons dessiner une ligne blanche entre le sommet suivant dans la liste des sommets triés et le sommet courant. Si nous sommes à la fin de la liste des sommets triés alors nous allons dessiner une ligne blanche entre le sommet courant et le premier sommet de la liste des sommets triés.

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: Nous commençons avec l'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

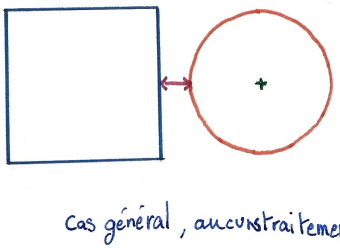

Figure 10: Pas de collision

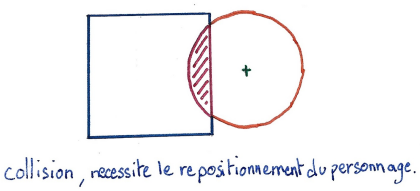

Figure 11: Collision

unitaires. Ainsi, nous employons un système de collisions entre un cercle et un rectangle (fig.1). 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 (fig.2).

Second cas: Dans ce cas ci, le cercle est en collision avec le rectangle (fig.3) 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: Maintenant, parlons de l'aspect technique. Nous définissons notre personnage, noté P, par des coordonnées représentant sa position ainsi qu'un nombre représentant sa largeur, noté L Ainsi :

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

$$L = 0,1^{5}$$

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

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 :

```
float clamp(float min, float max, float value)
{
        if (value < min)
                return min;
        if (value > max)
                return max;
        return value;
}
```

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

$$B = (a; b)$$

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 (fig.4). 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 :

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

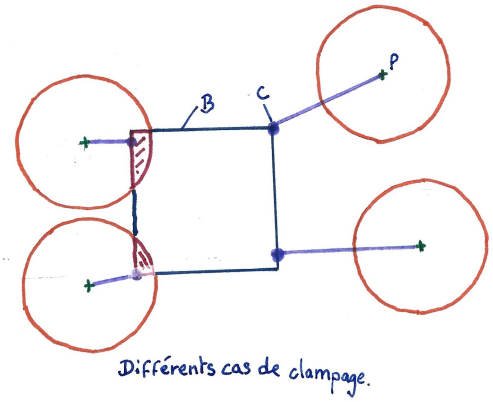

Figure 12: Clampage

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.

V Conclusion technique & personnelle

V.i Bilan du projet

Résultats obtenus

Apports technique

Apports personnels

V.ii Perspectives

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)

Nouvelles fonctionnalités

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

- Permettre à l'utilisateur de créer ses propres maps
- 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

Un plus dans nos CV

[1]

Bibliography

[1] Auteur 1 et Auteur 2. Titre de la ressource en ligne, 2023.

Gamedev Framework (GF): https://gamedevframework.github.io/