CC BY NC ND

Zeste de savoir

Comment DOOM et Wolfenstein affichaient leurs graphismes

12 août 2019

Table des matières

Les tout premiers *First Person Shooters*, comme DOOM, Wolfenstein 3D, et autres jeux des années 1990 avaient un rendu relativement simpliste et anguleux.

FIGURE 0. – Screenshot de FreeDoom

Le rendu n'était pas totalement en 3D, le meilleur moyen pour s'en rendre compte étant de tourner autour d'un objet : la forme de l'objet ne change pas du tout, ce qui indique que les objets sont de simples images, placées au-dessus du décor : ce sont des ***sprites***.

Mais on n'observe pas la même chose pour les murs, parce que la 3D des murs est simulée par un mécanisme différent de celui utilisé pour les objets et ennemis. Plus précisément, les moteurs de DOOM et autres jeux du même genre utilisent du ***ray-casting*** pour les murs, et des sprites pour les items et objets.

1. La map et le joueur : de la 2D pure

Avec cette méthode, la map doit respecter quelques contraintes :

- la map est un labyrinthe, avec des murs impossibles à traverser ;
- tout mur est composé de polygones, généralement des carrés de taille fixe ;
- la map n'a qu'un seul niveau : pas d'escaliers, d'ascenseurs, ni de différences de hauteurs (du moins, sans améliorations notables du moteur graphique).

Si elle respecte ces contraintes, on peut la représenter en 2D, avec un tableau à deux dimensions. Chaque case du tableau indique la présence d'un mur avec un bit (qui vaut 1 si le carré est occupé par un mur, et 0 sinon).

FIGURE 1. – Une map fictive d'un jeu avec ray-casting - couleurs = zones visitables et items

Le joueur est un vecteur dont l'origine est la position du joueur et la direction est celle du regard. Ce que voit le joueur est défini par :

— un angle (le champ de vision, ou FOV) ;
— la distance de l'écran par rapport au joueur.

2. Les murs : le ray-casting

Pour simuler la 3D à partir de l'image 2D de la *map*, le *ray-casting* a besoin de quelques contraintes :

— le sol et le plafond sont plats ;
— les murs font un angle de 90° avec le sol et le plafond ;
— les murs ont tous la même hauteur ;
— le regard du joueur est à une hauteur fixe au-dessus du sol, généralement la moitié de la hauteur d'un mur.

La dernière contrainte implique l'impossibilité de sauter, s'accroupir, lever ou baisser le regard. Les autres contraintes font que chaque mur est composé de cubes ou de pavés juxtaposés les uns à coté des autres.

A partir de ces contraintes et de la *map* en 2D, le moteur graphique peut afficher des graphismes de ce genre :

FIGURE 2. – Principe du ray-casting

2.1. Calcul de la hauteur perçue

Avec ce rendu, on colorie une colonne de pixels à la fois sur l'écran. Il faut pour cela connaitre la hauteur du mur vue depuis l'écran : cette hauteur sera appelée la **hauteur perçue**.

FIGURE 2. – Hauteur perçue et regard

La hauteur du mur perçue sur l'écran dépend de sa distance, par effet de perspective : plus un mur est proche, plus il paraitra "grand".

Dans le monde réel (ainsi que dans un jeu vidéo), si on multiplie la distance d'un objet par deux, trois ou quatre, celui-ci devient respectivement deux, trois ou quatre fois plus petit. Dit autrement, un objet de hauteur h_1 situé à une distance d_1 aura une hauteur perçue identique à celle d'un objet de hauteur double/triple/quadruple situé deux/trois/quatre fois plus loin. En clair, pour un objet de hauteur h_1, situé à une distance d_1, et un autre objet de hauteur h_2 et de distance d_2, les deux ayant la même hauteur perçue :

$$\frac{h_1}{d_1} = \frac{h_2}{d_2}$$

.

Dans un jeu qui utilise le ray-casting, la hauteur perçue est la hauteur du mur sur l'écran h_e, écran qui est situé dans le jeu a une distance d_e.

FIGURE 2. – Hauteur du mur sur l'écran

On sait donc que :

$$\frac{h_e}{d_e} = \frac{h_m}{d_m}$$

On en déduit la formule suivante, qui donne la hauteur perçue :

$$h_p = d_e \times \frac{h_m}{d_m}$$

Vu qu'on a supposé plus haut que la hauteur du regard est égale à la moitié de la hauteur d'un mur, on sait que le mur sera centré sur l'écran : il suffit de colorier avec la couleur du mur les pixels situés dans l'intervalle suivant, avec h_r est la hauteur du regard :

$$[h_r - \frac{h_p}{2}, h_r + \frac{h_p}{2}]$$

Les pixels situés au-dessus de cet intervalle correspondent au plafond : ils sont coloriés avec la couleur du plafond, souvent du bleu pour simuler le ciel. Les pixels dont les cordonnées verticales sont en-dessous de cet intervalle sont ceux du sol : ils sont coloriés avec la couleur du sol.

FIGURE 2. – Calcul de la couleur d'une colonne

2.2. Calcul des distances avec le mur

Dans l'équation vue plus haut, h_m et d_e sont des constantes connues à la compilation, et il ne manque que d_m pour faire le calcul. Or, d_m n'est pas la même pour chaque colonne de pixels : il faudra recalculer cette distance pour chaque colonne de pixel dans le champ de vision.

FIGURE 2. – Distance en fonction de la position dans le champ de vision

Pour cela, il faut déterminer une ligne (un rayon) qui passe par le joueur pour chaque colonne de pixel. Pour faire ce lancer de rayons, le moteur graphique doit connaitre la direction du regard, l'angle du champ de vision, et la résolution horizontale de l'écran (le nombre de colonnes de pixels).

FIGURE 2. – Lancer de rayon

Ensuite, il faut déterminer les coordonnées de deux points :

— la position du joueur ;
— l'intersection entre la ligne et le mur le plus proche.

On peut alors calculer la distance voulue à partir des coordonnées, avec l'aide du théorème de Pythagore. Si le joueur est à la position de cordonnées (x_1 , y_1), et l'intersection aux coordonnées (x_2 , y_2), la distance d respecte cette équation :

$d^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2$

La position du joueur est connue : elle est initialisée par défaut à une valeur bien précise au chargement de la map (on ne réapparait pas n'importe où), et est mise à jour à chaque appui sur une touche de déplacement. Ce n'est pas le cas de l'intersection, qui est calculée à l'aide d'un algorithme nommé *Digital Differential Analyser*.

2.3. Correction de perspective

En faisant ainsi, on obtient un rendu en œil de poisson (fish-eye), assez désagréable à regarder. Si ce rendu porte ce nom, c'est parce que les poissons voient leur environnement ainsi.

FIGURE 2. – Effet de rendu en œil de poisson

En fait, les rayons du bord du regard parcourent une distance plus grande que les rayons situés au centre du regard. Si on regarde un mur à la perpendiculaire, les bords seront situées plus loin que le centre : ils paraîtront plus plus petits.

FIGURE 2. – Origine de l'effet de vision en fish-eye

Les humains ont une lentille dans l'œil (le cristallin) pour corriger cet effet d'optique, lentille qu'il faut simuler pour obtenir un rendu adéquat.

Pour comprendre quel calcul effectuer, il faut faire un peu de trigonométrie. Prenons un joueur qui regarde un mur à la perpendiculaire (pour simplifier le raisonnement), tel qu'illustré ci-dessous : le rayon situé au centre du regard sera le rayon rouge, et les autres rayons du champ de vision seront en bleu.

FIGURE 2. – Situation de la démonstration

Pour éliminer le rendu en œil de poisson, les rayons bleus doivent donner l'impression d'avoir la même longueur que le rayon rouge. Or, vous remarquerez que le rayon bleu et le rayon rouge forment un triangle rectangle avec un pan de mur.

FIGURE 2. – Triangle formé par le champ de vision et le mur

Dans un triangle rectangle, le cosinus de l'angle a est égal au rapport entre le coté adjacent et l'hypoténuse, qui correspondent respectivement au rayon rouge et au rayon bleu. On en déduit que : $l_{rouge} = l_{bleu} \times \cos a$. On peut donc corriger la hauteur perçue en la multipliant par le cosinus de l'angle a.

FIGURE 2. – Rendu correct, sans effet d'œil de poisson

2.4. Application des textures

Le *ray-casting* permet aussi d'ajouter des textures sur les murs, le sol, et le plafond. Comme dit précédemment, les murs sont composés de pavés ou de cubes juxtaposés les uns à coté des autres. Une face d'un mur a donc une hauteur et une largeur.

Pour se simplifier la vie, les moteurs de ray-casting utilisent des textures dont la hauteur est égale à la hauteur d'un mur, et la largeur est égale à la largeur d'une face de pavé/cube.

FIGURE 2. – Application des textures

En faisant cela, chaque colonne de pixel d'une texture correspond à une colonne de pixel du mur sur l'écran (et donc à un rayon lancé dans les étapes du-dessus). Reste à trouver à quelle colonne de texture correspond l'intersection avec le rayon, et la mettre à l'échelle (pour la faire tenir dans la hauteur perçue).

L'intersection a comme cordonnées x et y, et est située soit sur un bord horizontal, soit sur un bord vertical d'un cube/pavé. On sait que les murs, et donc les textures, se répètent en vertical et en horizontal toutes les l_{mur} (largeur/longueur d'un mur).

FIGURE 2. – Détermination de la colonne de texture à afficher sur un rayon avec intersection verticale

En conséquence, on peut déterminer la colonne de pixel à afficher en calculant :

- le modulo de x avec la longueur du mur si l'intersection coupe un mur horizontal ;
- le modulo de y avec la largeur d'un mur si l'intersection coupe un mur vertical.

2.5. Résumé

Pour résumer, le moteur graphique doit :

- déterminer les équations des lignes à partir de la direction du regard ;
- détecter les intersections de ces lignes avec les murs ;
- en déduire les distances entre joueur et murs ;
- appliquer une correction de perspective ;
- appliquer Thalès pour calculer la hauteur perçue du mur ;
- déterminer les couleurs des murs, du plafond, et du sol (avec ou sans usage de textures) ;
- et potentiellement d'autres choses si on utilise un moteur qui gère les murs de hauteur variable, ou d'autres fonctionnalités.

3. Sprites

Le rendu des ennemis et items du jeu est basé sur des ***sprites***, des images d'item ou ennemi superposées sur le décor. Mais ces sprites donnent de mauvais résultats quand on tourne autour d'un item ou ennemi : la forme de l'objet ne change pas du tout. Pour gérer les effets de la distance, la taille des sprites est mise à l'échelle en fonction de leur distance, en suivant les mêmes méthodes que pour les murs. En posant la taille perçue d'un sprite t_p et t_s la taille réelle d'un sprite (aussi bien en vertical et horizontal) déterminée lors de la conception du jeu, on a une équation qui vaut non seulement pour la hauteur, mais aussi pour la largeur du sprite à l'écran :

$$t_p = t_s \times \frac{d_e}{d_m}$$

Cependant, certains sprites peuvent se recouvrir : il faut impérativement que le sprite le plus proche soit affiché au-dessus de l'autre. Pour cela, les sprites sont ajoutés suivant l'algorithme du peintre : on commence par intégrer les sprites des objets les plus lointains dans l'image, et on ajoute des sprites de plus en plus proches. Faire cela demande évidemment de trier les sprites à rendre en fonction de la profondeur des objets/ennemis dans le champ de vision (qu'il faut calculer).

Si cela vous intéresse, sachez qu'il existe de nombreux tutoriels sur le net, qui expliquent comment programmer un moteur de ray-casting, dont certains sont accessibles via les liens ci-dessous :

— S3DE : Simple 3D Engine ;
— A first-person engine in 265 lines ;
— Lode's Computer Graphics Tutorial : Raycasting .