Conception des jeux 3D

Moteurs graphiques

Christian Nguyen

Département d'informatique
Université de Toulon

Moteur de jeu (*game engine*)

Ensemble de composants logiciels permettant le développement jeu sur PC, consoles et mobiles et comprenant :

- un moteur graphique 2D et 3D,
- un moteur physique (détection de collision, etc.),
- un moteur d'animation,
- le graphe de scènes,
- la gestion du son (mixage musique et bruits, etc.),
- la possibilité de scriptage,
- l'intelligence artificielle,
- le réseau et le streaming,
- le support de la localisation,
- ...

Source Engine (Valve Software)

Moteur complet (Half-Life, Counter-Strike, Portal, Titanfall) qui prend en charge graphisme, son, réseau et physique, écrit en C++ (2004) :

- DirectX9 (OpenGL pour PlayStation 3 et Mac OS X),
- jusqu'à 32767 sommets par modèle,
- bump, cube, environment, displacement mapping,
- système de particules,
- éclairage et ombrage dynamique,
- flou de mouvement (*blur motion*),
- reflets dans l'eau, effets climatiques, génération de ciel,
- rendu dynamique des organismes, expressions faciales,
- ...

OGRE (Object-Oriented Graphics Rendering Engine)

Moteur de rendu de scènes 3D (Torchlight, Ankh, Garshasp, . . .) en temps réel écrit en C++ (2005) :

- support de Linux, Windows, OS X, NaCl, WinRT, Windows Phone 8, iOS et Android,
- gestion de scènes 3D (octree, BSP tree, . . .),
- multi-plateformes avec support d'OpenGL et DirectX,
- support des vertex et fragment shaders (GLSL, HLSL, Cg),
- MIP mapping (Progressive LOD - Level Of Detail),
- moteur d'animation (squelette, . . .),
- langage de scripts,
- système de particules,
- . . .

Unity

Moteur 2D/3D et physique (Cities Skylines, ...), licence gratuite.

- pas de modélisation (mais importation de modèles Maya, Cinema 4D, ...),
- création de scènes intégrant éclairages, terrains, caméras et textures,
- éditeur de script compatible Mono (C#), UnityScript (langage proche du JavaScript) et Boo (au lieu de Lua),
- applications compatibles Windows, Mac OS X, Linux, iOS, Android, Wii, PlayStation, XBox,
- compatible avec les API graphiques Direct3D, OpenGL et Wii,
- ...

Plan

1 Raycasting

2 Binary space partitioning (BSP)

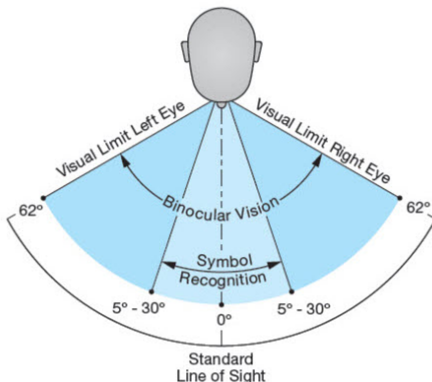
3 Potentially Visible Set (PVS)

4 Constructive Solid Geometry

A l'origine de produits ludiques en 3D (1992 : *Wolfenstein 3D* - John Carmack). Porté de la SNES (1993) à l'iPad (2010).

Le but est d'afficher à l'écran ce que voit un avatar qui progresse dans un labyrinthe.

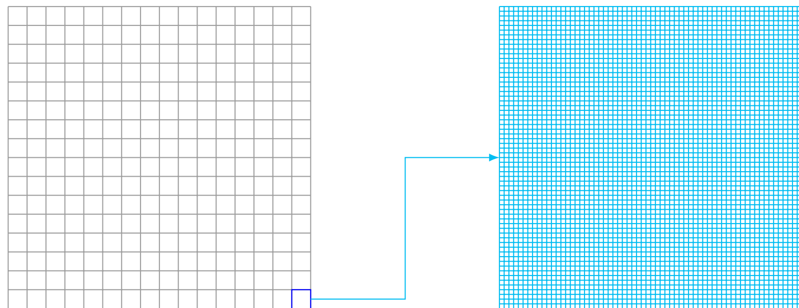
L'observateur possède un “champ de vision” (*field of vision* ou FOV) de 60°, ce qui permet d'appréhender l'environnement autour de l'axe de vision.



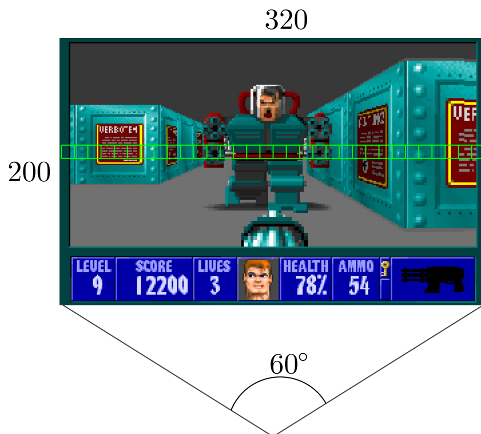
Les mouvements sont limités à la translation et la rotation dans le plan.

Le labyrinthe est représenté en vue aérienne, en 2D. Cette carte est subdivisée en cellules de taille fixe, dont l'unité de mesure est le pixel.

L'exemple de Wolfenstein : le labyrinthe se compose de 16×16 cellules, chaque cellule fait 64×64 unité.

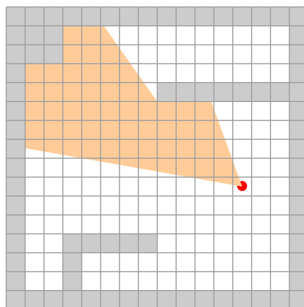


Si l'affichage se fait dans une résolution 320×200 pixels, sachant que l'angle de vision fait 60°, chaque colonne de pixels à l'écran équivaut à $60/320 = 0{,}1875°$.



Le joueur est représenté par trois nombres : sa position en (x, y) et son orientation (un angle du cercle trigonométrique) qui donne la direction de vue (le FOV se répartie donc de part et d'autre de cette direction sur $2 \times 30°$).

Depuis la position du joueur, 320 rayons sont lancés (résolution horizontale), chacun séparés de 0,1875°. Ils déterminent la distance de l'observateur aux murs.



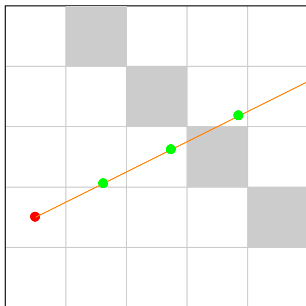
Si un rayon ne rencontre aucun mur alors la colonne de pixels correspondante de l'image à l'écran est affichée.

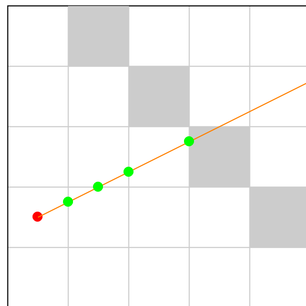
A chaque colonne est associée son équivalent dans le buffer de profondeur, ce qui va permettre d'afficher les sprites à la bonne dimension en utilisant la technique du MIP [1] mapping.

L'accroissement de la place requise pour tous les échantillons des MIP maps est un tiers de celle de la texture d'origine, car la somme des nombres $1/4 + 1/16 + 1/64 + 1/256 + \ldots$ converge vers $1/3$.

1. *Multum In Parvo* (beaucoup de choses dans un petit endroit)

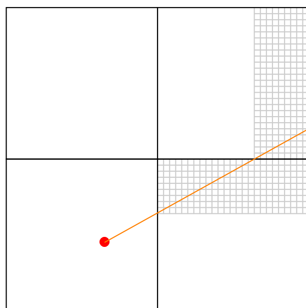
Le calcul de l'intersection d'un rayon avec un mur dans l'espace discret nécessite un algorithme adapté (DDA ou *Digital Differential Analysis*).





L'intersection entre un rayon et un mur peut s'effectuer soit sur une horizontale soit sur une verticale.

Si elle s'opère avec une horiz. (resp. vert.), on cherche la colonne (resp. la ligne) où elle a lieu puisque l'intersection se produit toujours à la frontière d'une cellule.

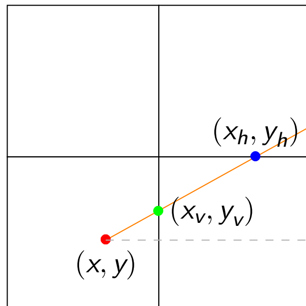


Calcul de la première intersection verticale :

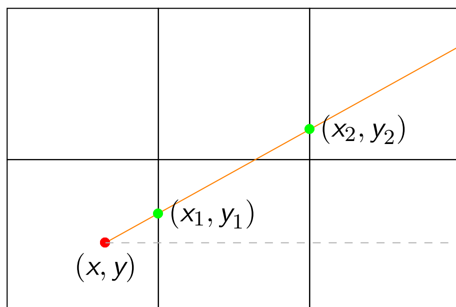
$$x_v = \lfloor \frac{x}{64} \rfloor \cdot 64 \quad (+64 \text{ si } -90 \leqslant \alpha \leqslant 90)$$
$$y_v = y + (x - x_v) \tan \alpha$$

Calcul de la première intersection horizontale :

$$x_h = x + (y - y_h) \tan \alpha$$
$$y_h = \lfloor \frac{y}{64} \rfloor \cdot 64 \quad (+64 \text{ si } 0 \leqslant \alpha \leqslant 180)$$



Calcul de la prochaine intersection verticale (x_2, y_2).



La distance $(x_2 - x_1)$ est constante (égale à la taille de la cellule, ici 64), d'où :

$$y_2 = y_1 + 64 \tan \alpha$$

Remarque : dans les quadrants 2 et 4 la tangente doit être multipliée par -1.

Calcul de la distance

On parcourt la carte d'intersection en intersection, en vérifiant pour chacune d'elles si le rayon touche un mur.

Lorsqu'un mur est détecté, on calcule sa distance à l'observateur en utilisant le calcul de l'hypoténuse (car la racine carrée est croissante indéfiniment alors que les fonctions trigonométriques sont bornées).

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

L'affichage s'effectue en dessinant successivement les 320 bandes de pixels verticales qui constituent la largeur de l'écran.

L'angle α correspond à la bande verticale qu'il faut dessiner et la distance *d* permet de savoir à quel moment les pixels du mur doivent apparaître.

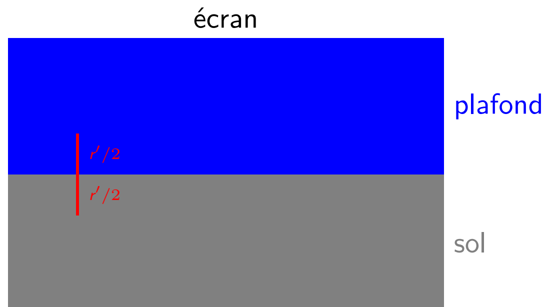
Pour adapter l'affichage du mur à la résolution choisie, on utilise la notion de plan de projection, ici l'écran. Connaissant la largeur de l'écran (320 pixels) et l'angle de vision (60°), on en déduit la distance virtuelle δ :

$$\delta = \frac{160}{\tan \frac{\alpha}{2}}$$

Soit h la hauteur d'un mur alors la taille sur l'écran du tronçon de mur observé r est donnée par :

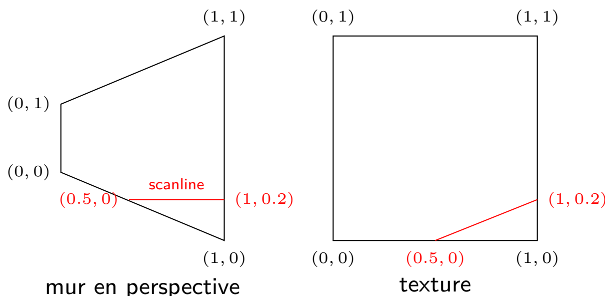
$$r = h \cdot \frac{\delta}{d}$$

Sans correction, on obtient un effet de distorsion “grand angle” (*fishbowl effect*) dû aux erreurs induites par l'utilisation des fonctions trigonométriques (solution : $r' = r \cos \alpha$).



Un mot sur les textures ...

Les coordonnées des éléments de textures pour chaque pixel sont calculées grâce à un algorithme de type *scanline*.



Les fonctions de génération de textures sont définies au lancement du jeu, en fonction de la taille de l'affichage, à des emplacements fixes en mémoire.

Plan

1 Raycasting

2 Binary space partitioning (BSP)

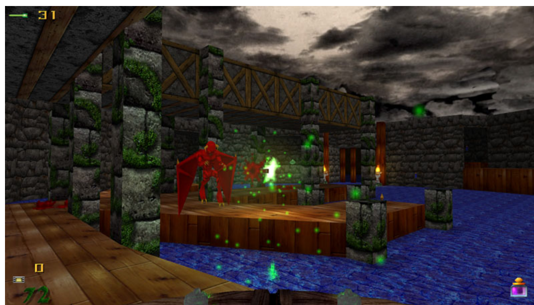
3 Potentially Visible Set (PVS)

4 Constructive Solid Geometry

Utilisé dans le moteur graphique id Tech 1 (Doom 1995).

3D et nettement supérieur au moteur utilisé dans *Wolfenstein* mais limité par une modélisation faite dans le plan (lignes de vue parallèles au sol, murs perpendiculaires au sol, ...).

Le code source de ce moteur a été mis à la disposition de la communauté en 1999 sous licence publique générale GNU.

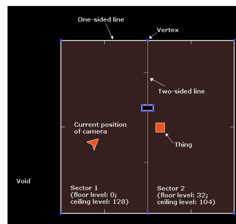


Structure

L'objet de base est le sommet. Celui-ci peut être relié à un autre sommet par un segment (*linedef*). Chaque *linedef* peut avoir un ou deux côtés (*sidedefs*). Les *sidedefs* forment des polygones (*sectors*).

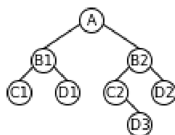
Chaque *sector* est défini par : hauteur et texture du sol, hauteur et texture du plafond, niveau d'illumination. Les *linedefs* à un côté (resp. deux côtés) représentent les murs (resp. des passages).

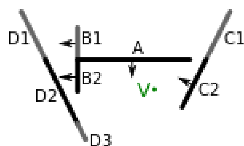
Chaque *sidedef* peut avoir jusqu'à trois textures : centre, haut et bas.



Chaque niveau est représenté par un arbre binaire dont les nœuds représentent des lieux et les feuilles des polygones convexes (*subsectors*).

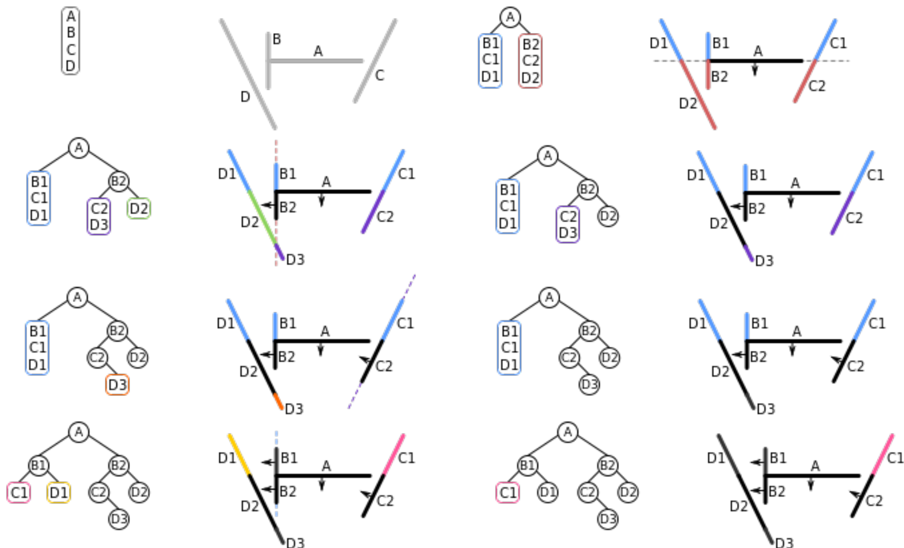
L'algorithme de rendu repose sur l'arbre BSP pour sélectionner chaque *subsector* dans le bon ordre (voir transp. suivant).





Dans cet exemple, les polygones sont traités dans l'ordre (D1, B1, C1, A, D2, B2, C2, D3).

Algorithme BSP



Le rendu

Les murs sont rendus avec la technique du raycasting (tranches verticales de texture de hauteur variable).

Les sols et plafonds sont rendus avec une méthode similaire : le Z-Mapping (Mario Kart) et qui fonctionne avec des tranches de texture horizontales.

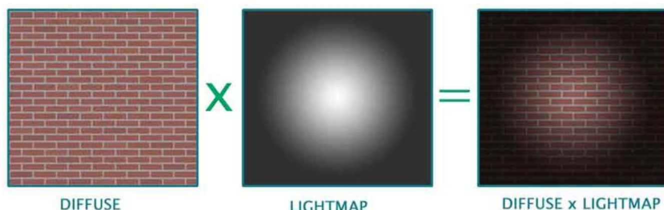
Les sprites sont stockés non triés mais sous la même forme que les murs pour accélerer leur affichage. Une liste des sprites visibles (*vissprites*) est maintenue et triée juste avant l'affichage.

Plan

1. Raycasting
2. Binary space partitioning (BSP)
3. Potentially Visible Set (PVS)
4. Constructive Solid Geometry

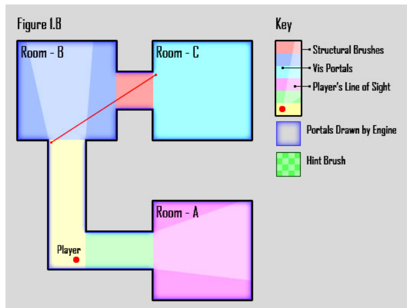
Le id Tech 2 est un moteur de jeu vidéo développé par id Software (Quake 1996). Il propose deux modes de rendu de la 3D : via la carte vidéo (hardware) ou calculée par le processeur (software).

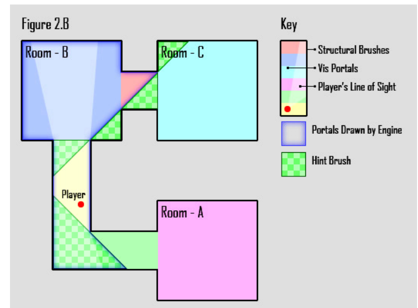
Il utilise les arbres BSP pour accélérer le rendu des scènes, l'ombrage Gouraud pour les objets dynamiques et des *lightmaps* pour les objets statiques.



Adaptation du code de la partie réseau à Internet (plus lent qu'un réseau local). Réalisation d'un support OpenGL générique (GLQuake) en 1997.

L'occlusion des feuilles de l'arbre BSP est précalculée à l'aide de "portails" (*portals*) lors du traitement de chaque map.





- ⊕ choix d'un ensemble pré-calculé en fonction du point de vue (encore réduit par le *frustum culling*),
  le pré-calcul libère du temps pour le reste (visibilité, rendu, IA, physique, ...).
- ⊖ espace mémoire supplémentaire nécessaire,
  le pré-calcul peut être très long,
  inutilisable pour les éléments de scènes dynamiques.

Fonctionnement du pipeline de rendu :

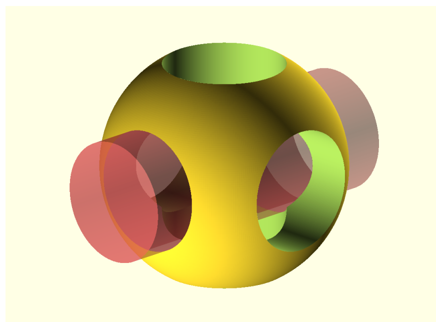
- le frustum de la caméra parcourt le BSP avec un z-ordering afin d'éliminer les feuilles en dehors du champ de vision.
- on indique au moteur de rendre tous les polygones contenus dans ces feuilles visibles.
- en rendu "software" les polygones sont clippés entre eux avant d'être rendus (Global Edge List - GEL).
- les polygones sont envoyés à la carte vidéo (en mode hardware) ou tracés par le processeur (en mode software).
- les personnages sont rendus plus simplement, directement en z-buffering (en software comme en hardware)

Plan

1. Raycasting
2. Binary space partitioning (BSP)
3. Potentially Visible Set (PVS)
4. Constructive Solid Geometry

L'Unreal Engine est un moteur de jeu (écrit en C++) développé par Epic Games (1998 - Unreal, Deus Ex, Rune, ...).

Utilise une géométrie constructive et l'opérateur de soustraction (on part d'une scène pleine que l'on creuse).



On peut créer des *portals* pour cloisoner le monde en zones fermées et n'afficher que la partie visible grâce aux PVS (*Potential Visibility Sets*).

To be continued ...