

# 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<sup>1</sup> 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,
- ...

---

1. impasse volontaire sur les moteurs spécialisés (tel que l'Unreal Engine)

Moteur complet (Half-Life, Counter-Strike, Portal, Titanfall, ...), issu du moteur id Tech 2 ([John Carmack](#)) 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 **libre** 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,
- ...

Moteur 2D/3D et physique (Cities Skylines, Hearthstone, ...),  
licence gratuite (2005) :

- pas de modélisation avant v. 2018.1 (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,
- ...

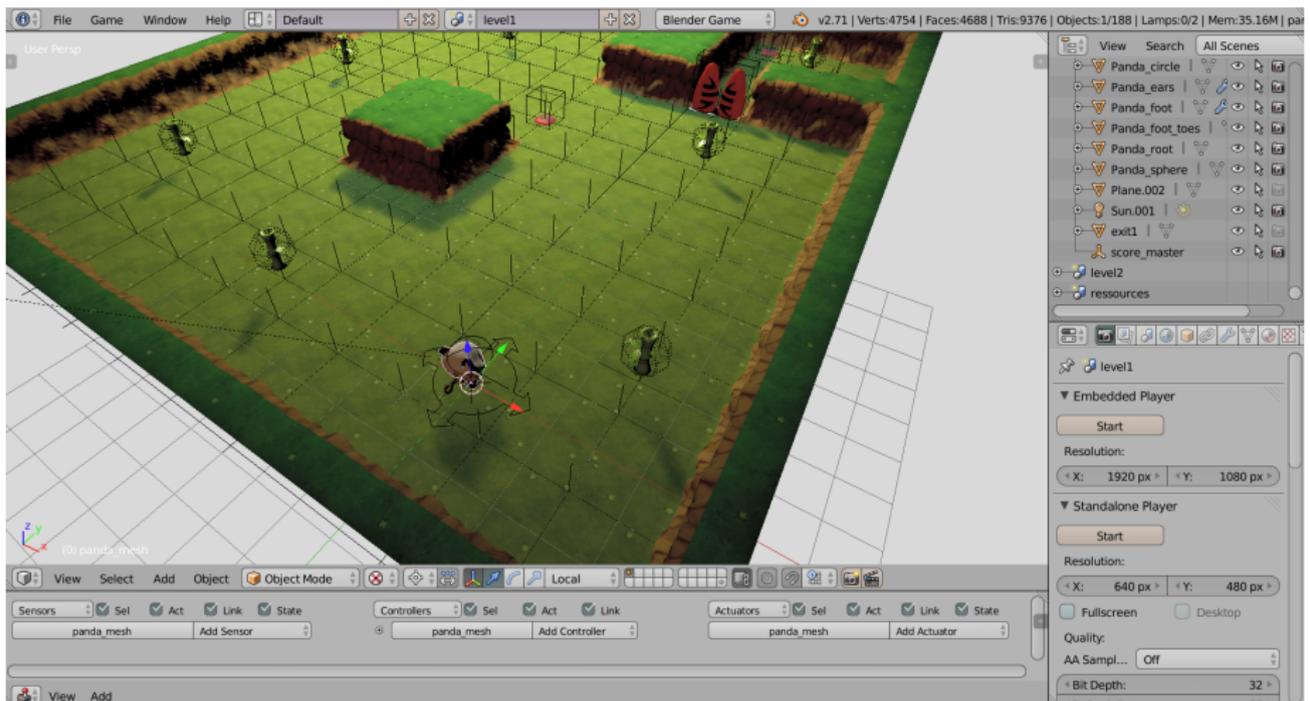
[Wikipédia] Moteur de jeu intégré dans Blender (jusqu'à avril 2018), gratuit et open source, affichage en temps-réel de scènes 3D interactives<sup>2</sup>.

- multiplateformes, plugin web (Burster),
- affichage GLSL (OpenGL 4 avec Vertex et Pixel shaders),
- prog. logique : capteurs, contrôleurs et actionneurs,
- langage de script Python,
- moteur physique intégré Bullet,
- sons 3D,
- IA : recherche du plus court chemin (pathfinding),
- gestion de scènes complexes grâce à une gestion intégrée du Level of Detail, d'occlusion culling, etc.

---

2. <https://fr.flossmanuals.net/blender-pour-le-jeu-video/introduction/>

# Blender (Game Engine)



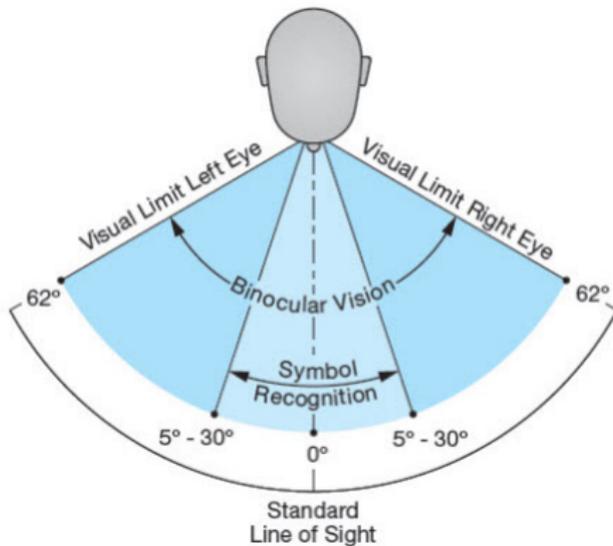
- 1 Raycasting
- 2 Binary space partitioning (BSP)
- 3 Potentially Visible Set (PVS)
- 4 Constructive Solid Geometry

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.



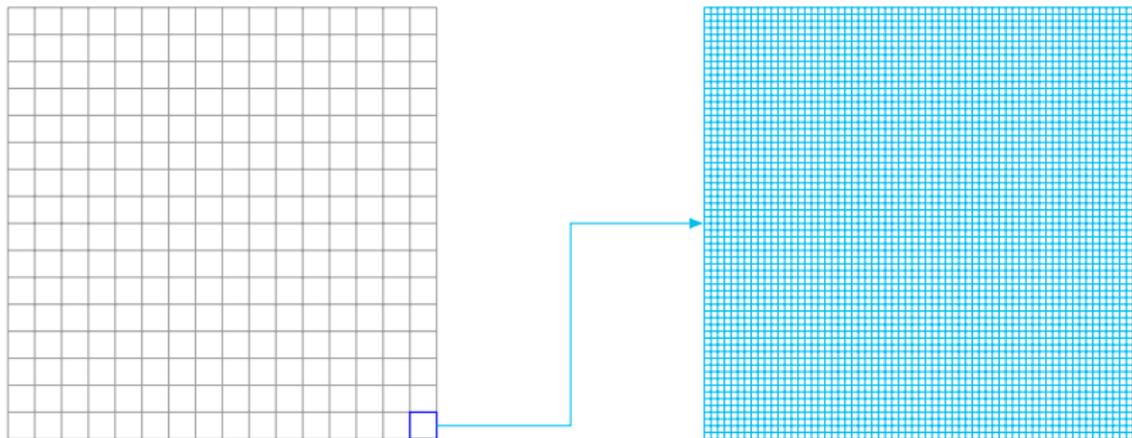
L'observateur possède un "champ de vision" (*field of vision* ou FOV) de  $60^\circ$ , ce qui permet d'appréhender l'environnement autour de l'axe de vision.



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.

Exemple de Wolfenstein : le labyrinthe se compose de  $16 \times 16$  cellules, chaque cellule fait  $64 \times 64$  unité.



Dans cette représentation 2D, le labyrinthe doit respecter quelques contraintes :

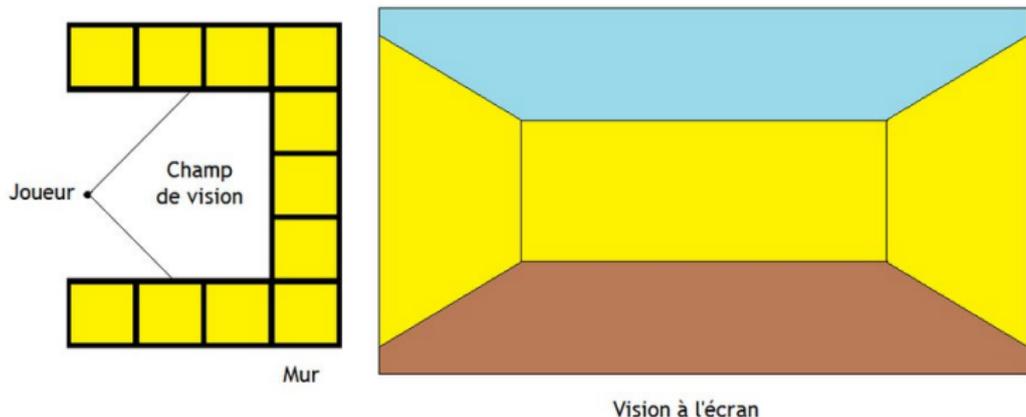
- tout mur est composé de polygones, généralement des carrés de taille fixe,
- chaque carte définit un seul niveau, il n'y a aucune différence de hauteur.

Si la « map » 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, ou en son absence ce qu'elle contient.

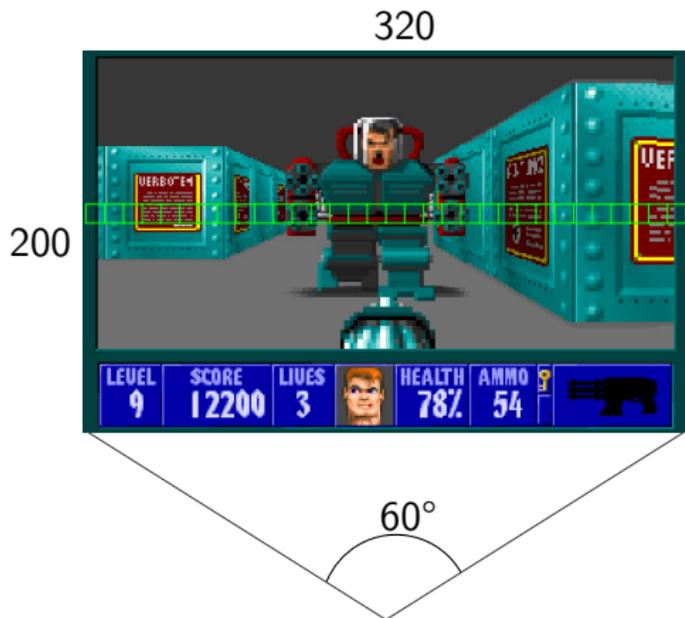


À partir du plan 2D du labyrinthe, on peut afficher une scène 3D à condition d'imposer un certain nombre de restrictions :

- le sol et le plafond sont plats,
- les murs font un angle droit avec le sol et le plafond, et sont tous de la même hauteur,
- le regard du joueur est à une hauteur fixe au-dessus du sol, généralement la moitié de la hauteur des murs (impossibilité de sauter, s'accroupir, lever ou baisser le regard).

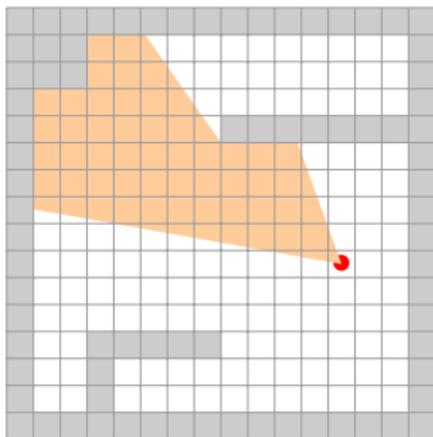


Si l'affichage se fait dans la résolution (originelle)  $320 \times 200$  pixels, avec un angle de vision de  $60^\circ$ , alors chaque colonne de pixels à l'écran équivaut à  $60/320 = 0,1875^\circ$ .

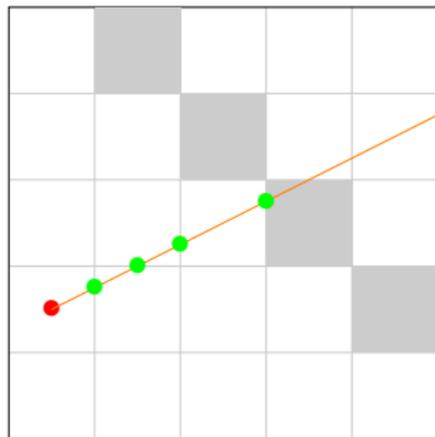
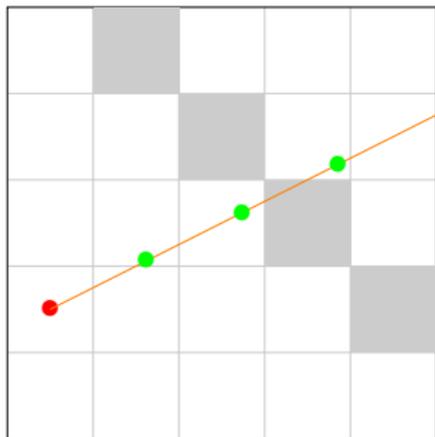


Le joueur est représenté par trois nombres : sa position en  $(x, y)$  et son orientation (un angle du cercle trigonométrique) qui donne la direction de vue (le FOV se répartie donc de part et d'autre de cette direction sur  $2 \times 30^\circ$ ).

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



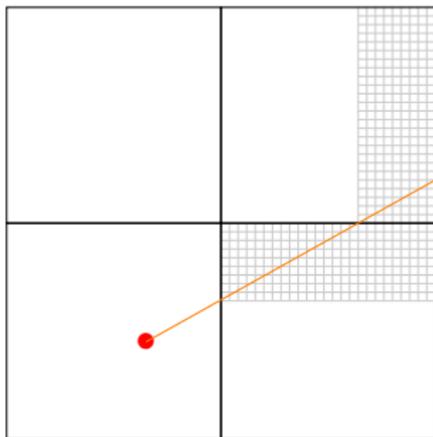
Le calcul de l'**intersection** d'un rayon avec un mur dans l'espace discret nécessite un algorithme adapté (DDA ou *Digital Differential Analysis*).



# Intersections

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 :

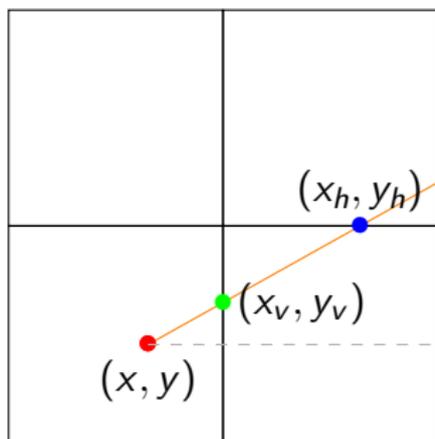
$$x_v = \lfloor \frac{x}{64} \rfloor \cdot 64 \quad (+64 \text{ si } -90 \leq \alpha \leq 90)$$

$$y_v = y + |x - x_v| \tan \alpha$$

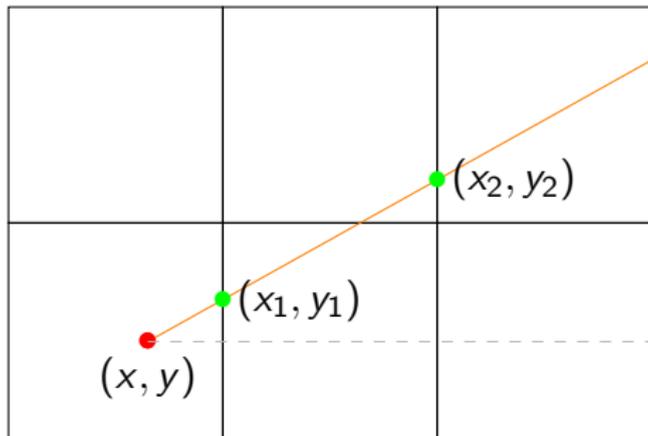
Calcul de la première intersection horizontale :

$$x_h = x + \frac{|y - y_h|}{\tan \alpha}$$

$$y_h = \lfloor \frac{y}{64} \rfloor \cdot 64 \quad (+64 \text{ si } 0 \leq \alpha \leq 180)$$



# 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  $-1$ .

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).

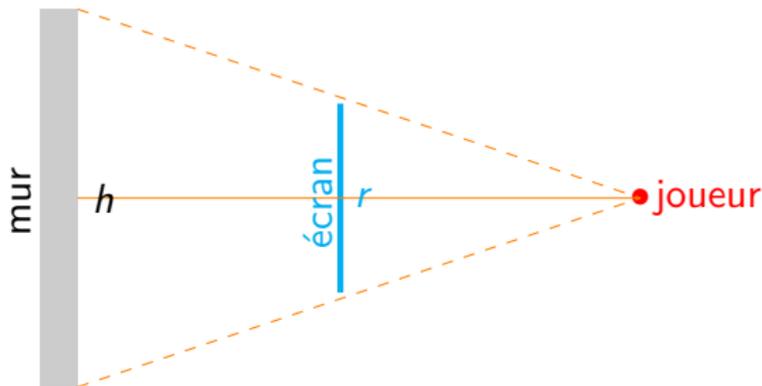
$$d = \sqrt{(x_m - x)^2 + (y_m - y)^2}$$

# Affichage par bande verticale

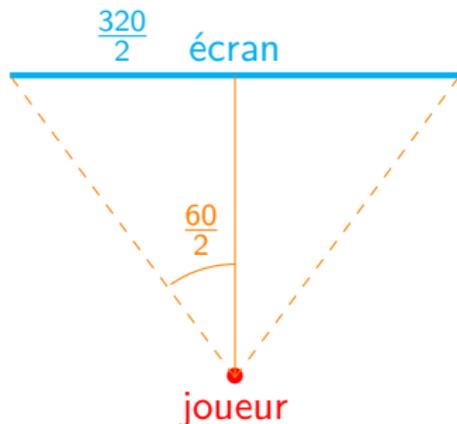
L'affichage s'effectue en dessinant successivement les 320 bandes de pixels verticales qui constituent la largeur de l'écran.

L'angle  $\alpha$  correspond à la bande verticale qu'il faut dessiner et la distance  $d$  permet de savoir à quel moment les pixels d'un mur doivent apparaître.

Pour adapter l'affichage du mur à la résolution choisie, on utilise la notion de plan de projection (ici l'écran).



# Distance à l'écran de projection



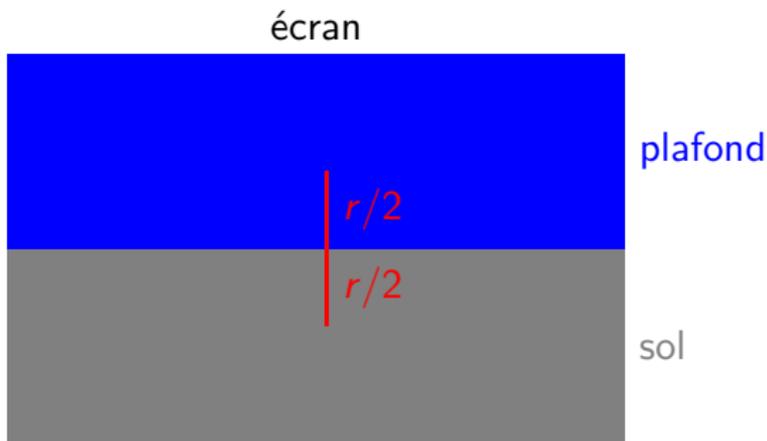
Connaissant la largeur de l'écran (320 pixels) et l'angle de vision ( $60^\circ$ ), on en déduit la distance virtuelle  $\delta$  :

$$\delta = \frac{320/2}{\tan(60/2)} \approx 277$$

# Hauteur d'un mur

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

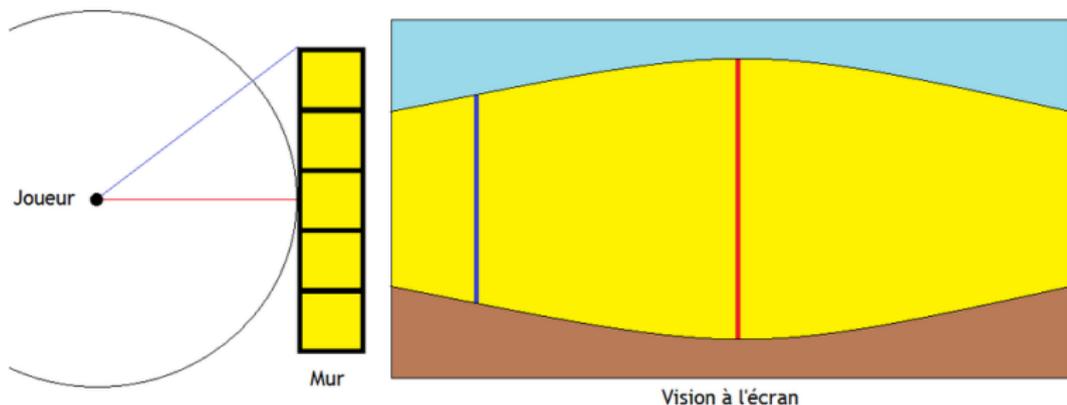
$$\frac{r}{\delta} = \frac{h}{d} \iff r = h \cdot \frac{\delta}{d}$$



# Hauteur d'un mur

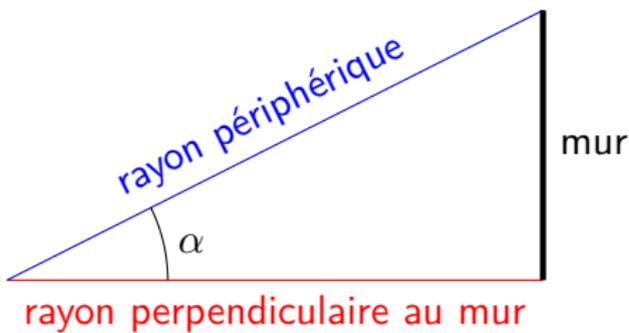
Sans correction, on obtient un effet de distorsion « grand angle » (*fishbowl effect*) conséquence de l'utilisation des fonctions trigonométriques.

En effet, les rayons de vue périphériques parcourent une distance plus grande que ceux situés au centre. Si on regarde un mur à la perpendiculaire, les bords seront situés plus loin que le centre : ils paraîtront plus petits.



# Hauteur d'un mur

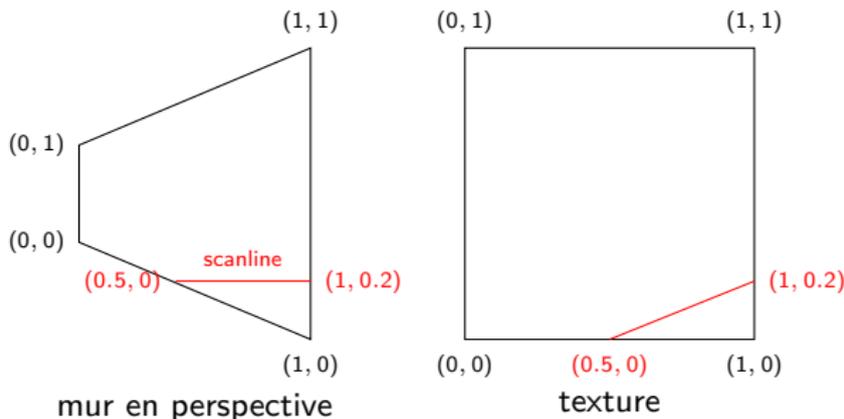
Pour éliminer le rendu « grand angle », les rayons bleus doivent avoir la même longueur que le rayon rouge.



Soit  $r$  (resp.  $r'$ ) le tronçon de mur observé par le rayon rouge (resp. bleu), on a la relation :  $r = r' \cos \alpha$ . On peut donc corriger la hauteur perçue en la divisant par  $\cos \alpha$ .

# Un mot sur les textures

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

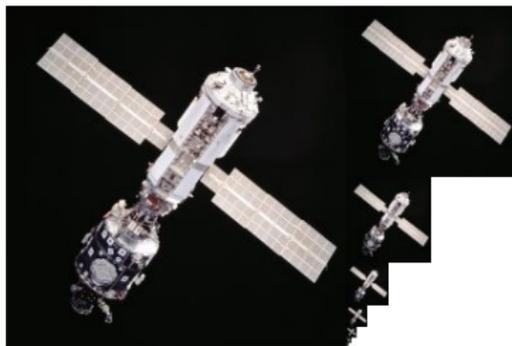


Les fonctions de génération de textures sont définies au lancement du jeu, en fonction de la taille de l'affichage, à des emplacements fixes en mémoire.

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

**Z-buffer** : à chaque colonne est associée son équivalent dans le buffer de profondeur, ce qui va permettre d'afficher des *sprites* à la bonne dimension en utilisant la technique du **MIP<sup>3</sup> 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  $1/4 + 1/16 + 1/64 + 1/256 + \dots$  converge vers  $1/3$ .



---

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



Le rendu des ennemis et des objets du jeu est basé sur des *sprites*, des images superposées sur le décor.

L'usage de sprites donne de mauvais résultats quand on tourne autour d'un objet ou d'un personnage car ceux-ci ne changent pas.

Seule leur taille  $h$  est mise à l'échelle en fonction de leur distance  $d$  au joueur, en suivant les mêmes méthodes que pour les murs :

$$r = h \cdot \delta / d.$$

Certains sprites peuvent se recouvrir : il faut impérativement que le sprite le plus proche soit affiché au-dessus des autres.

⇒ on utilise pour cela l'*algorithme du peintre* : on intègre les sprites des objets des plus lointains aux plus proches.

Pour résumer, le moteur graphique doit :

- déterminer les équations des rayons de vue à partir de la direction du regard,
- détecter les intersections de ces demi-droites avec les murs,
- en déduire les distances entre joueur et murs,
- appliquer une correction de perspective,
- calculer la hauteur perçue des murs,
- déterminer les couleurs (ou les textures) des murs, du plafond, et du sol,
- déterminer les sprites dans le champs de vision du joueur, leur taille relative et les afficher du plus éloigné au plus proche.

- 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).

En 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.

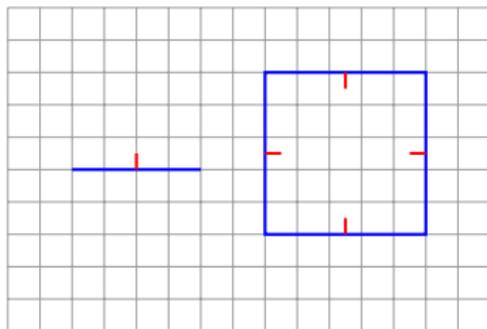


# Structure

L'objet de base est le sommet. Celui-ci peut être relié à un autre sommet par un segment (*linedef*).

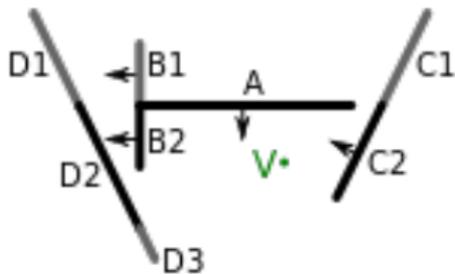
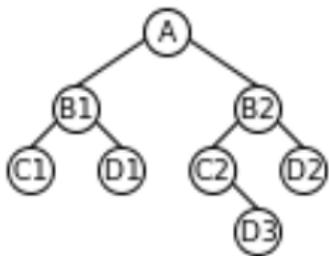
Chaque *linedef* peut avoir un ou deux côtés (*sidedefs*). Les *linedefs* à un côté (resp. deux côtés) représentent les murs (resp. des passages). Chaque *sidedef* peut avoir jusqu'à trois textures : centre, haut et bas.

Les *sidedefs* forment des polygones (*sectors*). Chaque *sector* est défini par : hauteur et texture du sol, hauteur et texture du plafond, niveau d'illumination.



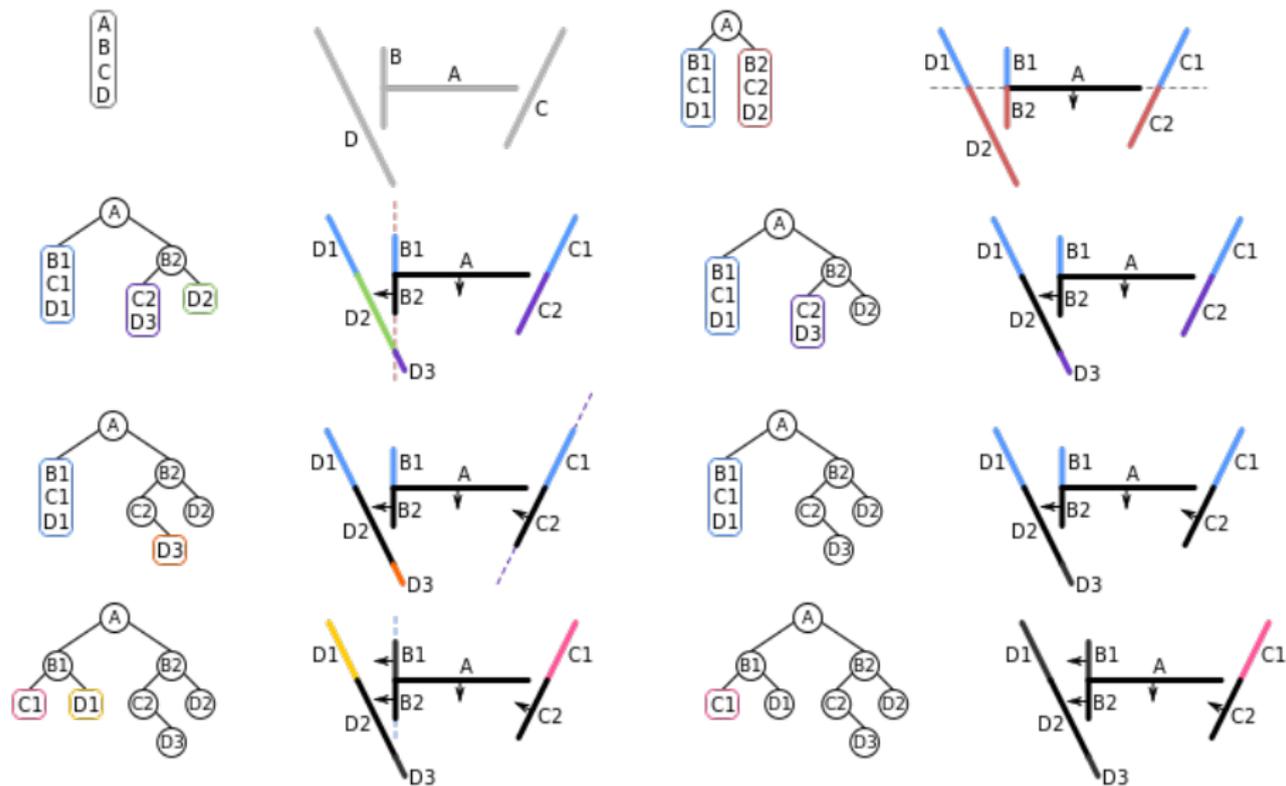
Chaque niveau est représenté par un arbre binaire dont les nœuds représentent des lieux et les feuilles des polygones convexes (*subsectors*).

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



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

Les sols et plafonds sont rendus avec une méthode similaire : le **Z-Mapping** (ou mode 7 sur Super NES) et qui permet d'appliquer des rotations et des dilatations ligne par ligne à l'arrière-plan (effet de perspective, etc.).

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

# Z-Mapping

Permet de transformer le fond d'écran ligne par ligne par rotation, dilatation, etc. L'effet le plus utilisé est la perspective (la hauteur est prise comme paramètre de profondeur).



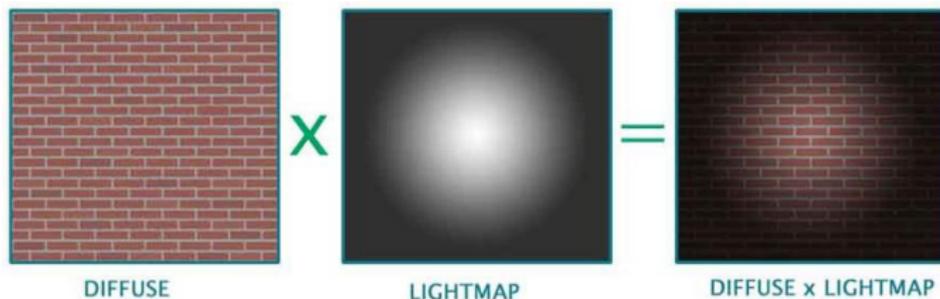
Un pixel est projeté à l'écran en utilisant une transformation affine (qui peut être spécifique à chaque ligne) comportant 6 paramètres :  $a$ ,  $b$ ,  $c$  et  $d$  pour la transformation et  $x_0$  et  $y_0$  pour l'origine de la transformation.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \left( \begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \right) + \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \quad (1)$$

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

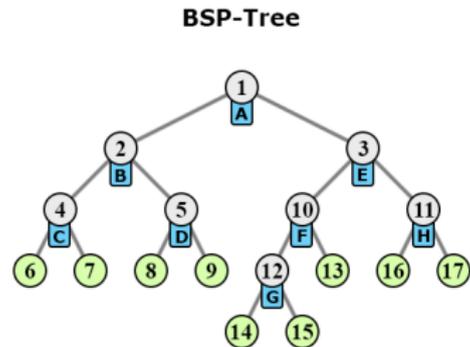
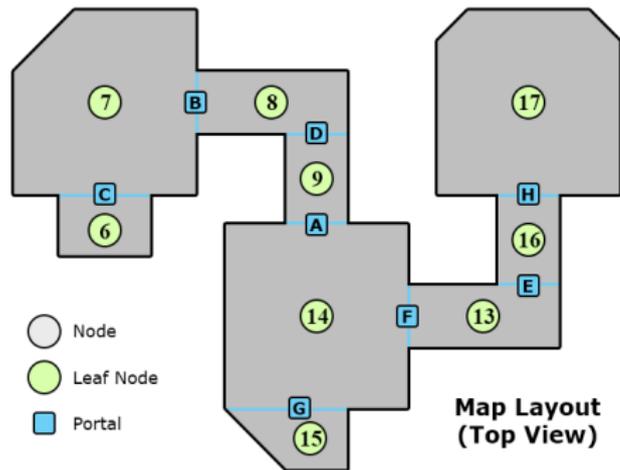
Le id Tech 2 est un moteur de jeu vidéo développé par id Software (Quake 1996). Il propose deux modes de rendu de la 3D : via la carte vidéo (hardware) ou calculée par le processeur (software).

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.



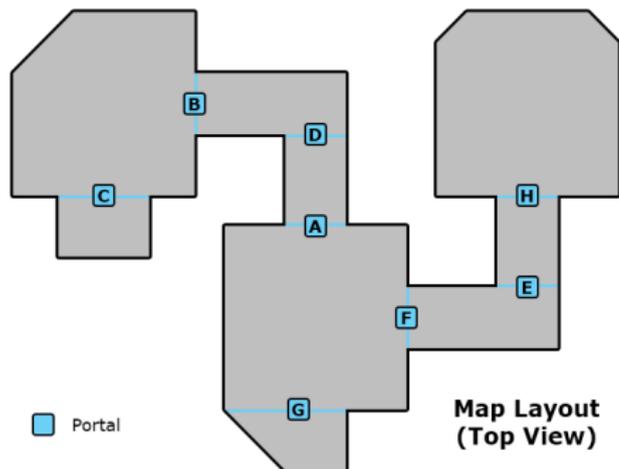
Adaptation du code de la partie réseau à Internet (plus lent qu'un réseau local). Réalisation d'un support OpenGL générique (GLQuake) en 1997.

L'occlusion des feuilles<sup>4</sup> de l'arbre BSP est précalculée à l'aide de "portails" (*portals*) lors du traitement de chaque map.



4. constituées de polygones convexes

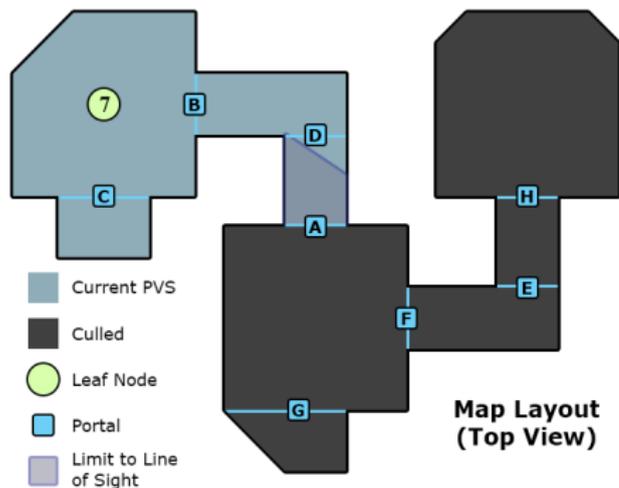
Si un portail est partiellement visible depuis un autre portail, il est coché ✓ sinon il est coché ✗.



PVS-Table

	B	C	D	E	F	G	H
A	✗	✗	✓	✗	✓	✓	✗
B		✓	✓	✗	✗	✗	✗
C			✗	✗	✗	✗	✗
D				✗	✓	✓	✗
E					✓	✓	✓
F						✓	✗
G							✗

Si le joueur est en 7, les portails associés sont B et C. Dans la table PVS, seul D s'ajoute pour le rendu.



PVS-Table

	B	C	D	E	F	G	H
A	✗	✗	✓	✗	✓	✓	✗
B		✓	✓	✗	✗	✗	✗
C			✗	✗	✗	✗	✗
D				✗	✓	✓	✗
E					✓	✓	✓
F						✓	✗
G							✗

- ⊕ 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.

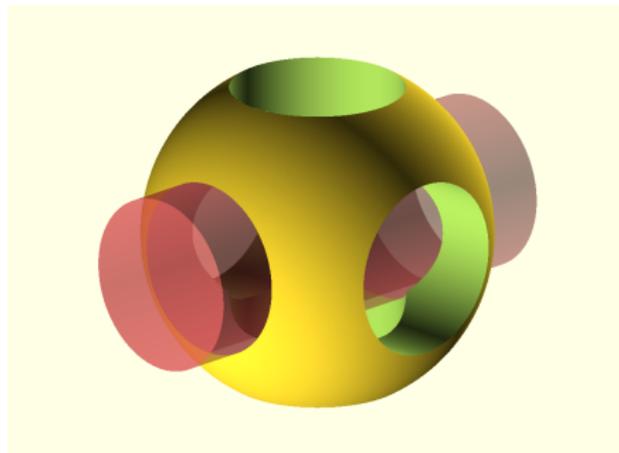
## Fonctionnement du pipeline de rendu :

- 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)

- 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, ...).

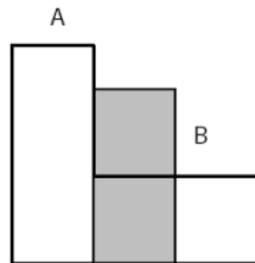
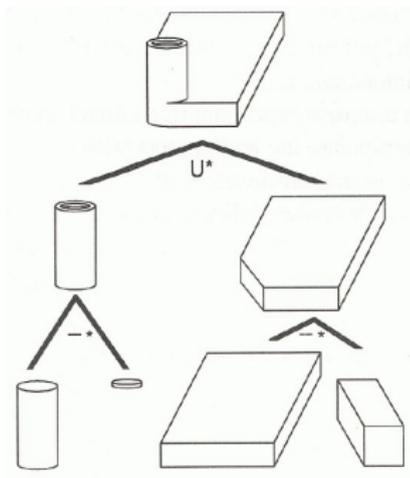
Utilise une géométrie constructive et l'opérateur de soustraction (on part d'une scène pleine que l'on creuse).



Comme dans Quake, des *portals* cloisonnent le monde et permettent de n'afficher que la partie visible grâce aux PVS.

Hiérarchie d'opérateurs ensemblistes ( $\cup$ ,  $\cap$ ,  $-$ ) sur des volumes élémentaires (sphère, cylindre, ...).

Représentation non unique et ambiguë si les opérateurs ne sont pas réguliers.



Création de modèle :

- primitives bornées (feuilles),
- transformations géométrique (nœuds),
- opérations ensemblistes (nœuds).

Traitement sur le modèle

- paradigme diviser pour régner (*Divide & Conquer*),
- récursivité sur un arbre binaire (traitement géométrique et booléen).

Intérêt de la récursivité pour la synthèse d'images :

- classification d'appartenance (modélisation géométrique 3D),
- suivi de rayon (ray-tracing en rendu réaliste).

# CSG - Classification d'appartenance

Traitement géométrique sur les primitives (calculs d'intersection segment/surface).

Traitement booléen : déterminer les parties du segment  $S$  par rapport au fils gauche (L :  $S_{onL}$ ,  $S_{inL}$ ,  $S_{outL}$ ) et au fils droit (R :  $S_{onR}$ ,  $S_{inR}$ ,  $S_{outR}$ )

