Réalisation d'un moteur graphique en pseudo-3D mappée

Version 1.0

François PESSAUX
Francois.Pessaux@inria.fr
28 ter, rue Musselburgh
94500 Champigny sur Marne - FRANCE

Janvier 1997

Table des matières

Table des figures

1 Préambule

Le présent document peut être considéré comme la suite logique d'un premier papier de vulgarisation que j'ai écrit en Novembre 1996, nommé : **BSP Trees pour la 3D Mappée - Arbres binaires de partitionnement spatial**.

Dans ce premier papier, je cherchais à décrire de façon simple, comment créer un BSP. Il ne s'agissait pas d'une discussion formelle sur le sujet, mais plutôt d'un discours devant permettre au non-spécialiste de comprendre cette notion de BSP. Intentionnellement, l'utilisation de ces BSP en vue de créer un moteur graphique à la *Doom*, n'était pas abordée (juste une ou deux petites allusions étaient présentes, mais loin d'être assez profondes pour comprendre comment fonctionne un moteur, je réservais ce sujet pour un papier ultérieur).

Je vais donc essayer ici de présenter le principe détaillé de fonctionnement d'un moteur en pseudo-3D mappée, un peu comme *Doom* l'est, avec une ou deux petites choses en moins qui seront décrites dans la section suivante. Je pense que j'aborderai la solution à ces limitations dans un addendum ultérieur au présent papier, l'histoire de me laisser le temps de prendre du recul sur la question. Néanmoins, ceci nous laisse bien des choses à voir et à comprendre.

Le but de ce document est donc de fournir des informations les plus claires possibles, comme j'aurais aimé en trouver. *Néanmoins, l'auteur du présent document ne pourra en aucun cas être tenu responsable des résultats d'une utilisation correcte ou incorrecte de ces informations, et de leurs conséquences sur qui ou quoi que ce soit.*

Ce document est volontairement très détaillé, avec des précisions pouvant paraître superflues ou évidentes au programmeur expérimenté, mais l'expérience montre qu'il vaut mieux expliquer trop que pas assez et que l'appréhension des non-dits est très variable d'une personne à une autre. J'essaierai d'illustrer mes propos par des morceaux de source C. Le portage de mon moteur en Caml-Light est toujours à l'ordre du jour, mais n'ayant pas encore été réalisé, seul du source C est actuellement disponible.

Ce document est volontairement farci de schémas illustrant mes propos. Si quelque chose n'est pas clair dans le texte, regardez s'il n'y a pas un schéma sur la page d'avant ou celle d'après qui s'y rapporte. En effet, les contraintes d'insertion d'images dans le document peuvent décaler les schémas de ± 1 page par rapport au texte...

Il est tout à fait possible que ce document contienne des erreurs ou des incorrections. Je m'en excuse par avance et les corrigerai au fur et à mesure de leurs découvertes. Toutes les remarques, critiques, etc... pourront m'être adressées par courrier électronique ou courrier postal.

Ce travail n'aurait jamais pu être réalisé sans les nombreuses références citées en fin du présent papier. Que leurs auteurs en soient remerciés. En particulier, ma principale source d'inspiration fut le source du moteur *DView* réalisé par *Jack Hill* permettant d'afficher les WADs de *Doom* (sans texturage ni objets dynamiques). Un décorticage attentif de ses fichiers, doublé d'une grosse dose de cogitation sur papier m'auront été nécessaires pour comprendre et réaliser les bases de qu'est un moteur 3D.

Je conseille fortement aux personnes ne connaissant pas le principe des BSP (structure de données représentant le monde dans lequel le joueur évolue) de lire mon précédent papier, ou un autre si celui-ci ne convient pas, afin de comprendre leur intérêt dans notre moteur graphique.

Un dernier mot avant de rentrer dans le vif du sujet : tout comme je l'ai déjà dit dans mon précédent papier, si ce document vous sert pour réaliser un programme ou un autre document, soyez sympas, mentionnez-moi dans vos sources d'inspirations, j'apprécie toujours de voir que je

n'ai pas bossé pour rien.
Et maintenant, commençons...

2 Cadre de la discussion

Commençons tout d'abord par décrire ce que nous voulons que notre moteur soit capable de faire et les points que nous allons aborder.

Je cherche ici à décrire le fonctionnement et la réalisation d'un moteur en pseudo-3D mappée à la *Doom* avec une ou deux limitations que nous allons voir à la section 2.3. Entrons donc dans le détail de cette spécification.

2.1 Un moteur en pseudo-3D

En effet, *Doom* n'est pas un moteur en 3D réelle. Comme nous le verrons un peu plus loin, lors de l'explication de la modélisation d'un monde, il s'agit en réalité de 2D à laquelle on adjoint après-coup une troisième dimension (la hauteur).

2.2 Mapping

Je parle donc d'un moteur en pseudo-3D *mappée*. Ceci signifie que l'on va appliquer des textures sur les composantes représentant notre monde afin d'en améliorer le réalisme.

2.3 Moteur à la *Doom*

Ceci signifie en particulier que nous allons chercher à gérer un monde où les murs sont orientés de façon quelconque, et non perpendiculairement comme *Wolfenstein* le faisait. Tout comme dans *Doom*, nous ne pourrons pas gérer des pièces les unes au dessus des autres.

Remarque : Notons que la plupart des livres (au moins ceux que j'ai eu l'occasion de lire) se permettent d'assimiler allègrement sur leur couverture et dans leurs introductions, les moteurs à la *Wolfenstein* aux moteurs à la *Doom*. En réalité, il existe une différence fondamentale...

2.4 Les petites choses en moins par rapport à *Doom*

Il y a principalement 3 choses que je ne décrirai pas ici et qui sont présentes dans le moteur de *Doom* :

- Les secteurs (pièces) de hauteur différente
- Le mapping (texturage) du sol
- Le mapping du plafond

Les raisons sont très simples : tout d'abord, ce sont des points que je n'ai pas encore bien explorés[1]. Ensuite, ces "petites" choses ont le don de compliquer notablement le moteur, aussi, il

[1] Maintenant que j'arrive à la fin de l'écriture de ce papier, j'ai fini par comprendre comment rajouter les murs de hauteur différente (en réalité, les secteurs de hauteur différente puisque la hauteur d'un mur est celle du secteur

n'est pas plus mal de ne pas en parler tout de suite, afin de rendre plus compréhensible un truc déjà incompréhensible $\ddot\smile$.

2.5 Le petit plus (pas par rapport à *Doom*, cette fois)

J'essaierai d'expliquer comment il est possible d'adjoindre, dans un tel moteur, l'affichage d'objets dynamiques (comme le sont les monstres et autres cibles dans *Doom*).

Ce dernier point est en fait à mon avis un autre problème, connexe certes, au rendu proprement dit de l'environnement spatial où le joueur évolue. Néanmoins, il convient de ne pas l'oublier, car *Doom* sans monstres serait un peu lassant $\ddot\smile$.

Je ne prétends certes pas que la méthode que j'exposerai dans ce point est la meilleure. Elle a le mérite de fonctionner, mais peut certainement être améliorée. D'ailleurs, toutes suggestions ou critiques seront les bienvenues.

3 Comment est représenté le monde en pseudo-3D

Je vais reprendre ici (presque) textuellement ce que j'ai déjà énoncé dans mon précédent papier, étant donné que ça n'a pas changé depuis...

Comme beaucoup d'autres l'ont déjà dit, *Doom ressemble* à de la 3D, mais *n'est pas* de la 3D... Cela ne paraîtra pas nouveau au connaisseur, mais peut épater le néophyte. Quand on crée un "tableau" (autrement dit un niveau de jeu), pour *Doom (appelé aussi un "WAD")*, on crée en fait un ensemble de lignes droites dans un plan (un peu comme si on traçait les murs d'une maison sur une feuille de papier). La troisième dimension, à savoir la hauteur, est fournie comme attribut du secteur ainsi défini (un secteur est un polygone fermé défini par un ensemble de segments orientés *dans le sens des aiguilles d'une montre*. On peut donc considérer en première approximation un secteur comme une pièce).

Le fait que les murs possèdent une orientation, particulièrement dans le sens des aiguilles d'une montre, définit pour ces derniers une notion de *côté gauche* et de *côté droit*. Du fait de l'orientation choisie, le joueur ne pourra toujours que voir le côté *droit* d'un mur. Orienter les murs revient à définir un point de *départ* et un point *d'arrivée*. Il faudra toujours garder cette idée présente à l'esprit, car nous parlerons sans arrêt de "début" et de "fin" d'un mur. Le point de départ d'un mur sera donc toujours plus à gauche dans le champ visuel du joueur que le point d'arrivée de ce même mur.

La figure 1 montre ce qui peut se faire de plus simple en matière de monde. Il s'agit d'une simple pièce rectangulaire. Pour réaliser une telle prouesse, nous avons besoin de 4 points et de 4 segments

---

auquel ce mur appartient), donc les portes et les secteurs montant et descendant. je l'ai même implanté dans mon prototype et tout semble fonctionner parfaitement. Néanmoins, comme mentionné dans la conclusion de ce document, ceci implique un peu de travail au niveau du principe du moteur, au niveau des structures représentant le monde, et donc au niveau des outils permettant de créer des tableaux de test. Je ferai certainement un addendum à ce papier plus tard afin d'exposer la nouvelle vision du monde, avec de tels murs. Rassurez-vous les $\frac{9}{10}$ de ce que nous allons voir restent tels quels dans un monde avec secteurs. Il faudra par contre rajouter une ou deux petites choses.

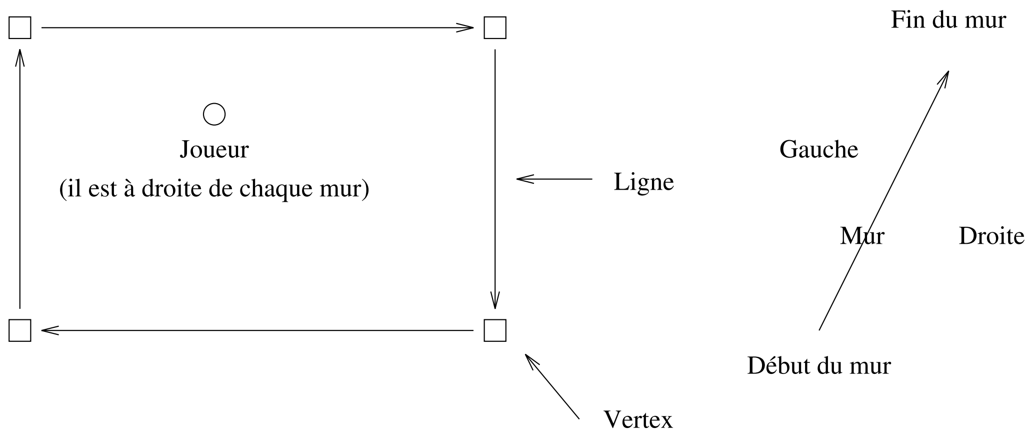

Figure 1: Un des mondes les plus simples : une pièce rectangulaire

orientés (donc des vecteurs pour être plus exact). Afin de ne pas rompre avec la littérature anglo-saxonne, nous nommerons les points des *vertices* (et au singulier un *vertex*), et les vecteurs des *linedefs*.

Plusieurs linedefs, représentant un *polygone fermé*, définissent un *secteur*. Un secteur possède alors quelques caractéristiques propres, comme la hauteur de son plafond et celle de son plancher (l'altitude devrais-je plutôt dire, car la hauteur d'un mur est en fait exprimée comme la différence entre l'altitude du plafond et celle du plancher), les textures pour chacun de ces derniers, etc...

Un objet (le joueur est aussi un objet, mais que l'on ne voit jamais) dans notre monde est donc repéré par ses 2 coordonnées (abscisse x et ordonnée y) dans le plan 2D que forme la carte du monde où notre joueur évolue. On peut éventuellement attribuer à cet objet une position verticale (une hauteur) par rapport au plancher, mais ceci n'a pas à proprement parler un lien avec la localisation dans notre "carte du monde" puisque nous avons pris comme axiome de départ que 2 pièces ne peuvent pas se superposer. Ainsi cette notion de hauteur interviendra d'une autre façon que nous verrons plus tard.

Remarque : En réalité dans *Doom*, une pièce complète peut nécessiter, plusieurs secteurs adjacents ayant des caractéristiques différentes comme la texture du sol, du plafond, la hauteur des murs formant ce secteur, la luminosité ambiante, des déclencheurs d'évènements, etc... Ainsi, la pièce pourra éventuellement être découpée en plusieurs parties.

Remarque : Au cas où vous n'en auriez jamais eu l'occasion, il serait préférable de regarder au moins un peu comment fonctionne un éditeur de WAD, comme *DEU*, pour ne citer que lui.

4 L'intérêt d'un BSP pour un moteur graphique

Comme expliqué dans mon précédent papier, les BSP sont une structure d'arbre qui permet, via un certain parcours, d'obtenir les murs (en fait les linedefs représentant les murs de notre monde) dans l'ordre d'éloignement croissant (ou inversement, par le parcours inverse).

Le parcours qui nous intéresse pour le réalisation d'un moteur graphique est celui qui nous communiquera les murs dans l'ordre d'éloignement croissant. C'est à dire que nous voullons trouver les murs de notre monde les plus proches du joueur en premier.

L'explication de l'interdiction d'avoir des pièces les unes au dessus des autres est très simple : nous utilisons des BSP en 2 dimensions. Aussi, il est impossible de partitionner la troisième dimension (la hauteur) qui pourrait alors différencier alors les 2 pièces supperposées.

5 Rappels mathématiques

Certaines notions mathématiques utilisées par le moteur se trouvent décrites dans mon précédent papier, aussi ne les réécrirai-je pas. Il s'agit principalement de la définition d'un point dans un repère, d'un vecteur et de la façon de situer un point (gauche, droite ou sur) par rapport à une droite orientée.

Il convient de préciser que le pré-requis mathématique nécessité pour la réalisation d'un tel programme est extrêmement faible : il suffit de savoir ce qu'est un *sinus*, un *cosinus* et une *tangente*.

Je tenterai d'expliquer, au fur et à mesure qu'elles apparaîtront, le pourquoi des formules mathématiques utilisées, sachant qu'elles resteront très simples.

6 Le principe du rendu

6.1 Le parcours du BSP

Comme base de départ, nous partons donc du BSP issu de la création notre monde (éventuellement par un outils spécialisé). Nous voulons connaître en premier les murs (nous utiliserons maintenant indifféremment les mots *mur* ou *linedef*) les plus proches du joueur. Nous partons bien évidemment du principe que l'on connaît la position du joueur dans le monde (en terme de coordonnées (x, y) sur la carte).

Rappelons que le BSP est un arbre dont les feuilles contiennent des murs (formant un polygone convexe vu du dessus, et appartenant au même secteur afin d'être sûr que chaque mur a la même hauteur). Un nœud de l'arbre comporte aussi un mur (la linedef de partitionnement) et 2 pointeurs vers 2 sous-arbres (1 vers le sous-arbre de droite, 1 vers le sous-arbre de gauche). Comme tout arbre, chaque branche se termine obligatoirement par une feuille éventuellement vide (pour raccourcir le traitement en C on met alors un pointeur NULL, plutôt qu'une structure de feuille ne contenant aucun mur).

Pour obtenir nos murs dans l'ordre désiré, il convient de parcourir le BSP de façon illustrée par le source en pseudo-code suivant :

```
void FrontToBackTreeParsing (Tree)
struct Tree *Tree ;
{
Si l'arbre <<Tree>> est une feuille alors
  Communiquer à la routine d'affichage chacun des murs présent à la feuille
Sinon
```

```
  {
  L'arbre représente donc un noeud.
  Chercher la position du joueur par rapport à la ligne de partition du noeud.
  Si le joueur est à droite de cette linedef alors
    {
    Récurser dans le sous-arbre de droite
    Communiquer à la routine d'affichage le mur servant de
      ligne de partitionnement pour le noeud actuel
    Récurser dans le sous-arbre de gauche
    }
  Sinon
    {
    Si le joueur est à gauche de cette linedef alors
      {
      Récurser dans le sous-arbre de gauche
      Communiquer à la routine d'affichage le mur servant de ligne de
        partitionnement pour le noeud actuel
      Récurser dans le sous-arbre de droite
      }
    Sinon
      {
      C'est que le joueur est exactement sur le mur servant de ligne de
        partitionnement à notre noeud actuel
      Communiquer à la routine d'affichage le mur servant de ligne de
        partitionnement pour le noeud actuel
      }
    }
  }
}
```

Remarque 1 : Il ne semble pas normal que le joueur puisse se trouver exactement sur un mur, car la détection de collision devrait avoir joué. Avec cet algorithme, je considère que si le joueur se trouve exactement sur (dans ?) le mur, alors il ne "verra" que ce mur puisque mathématiquement sa hauteur apparente sera infinie.

Bon, dans la réalité, le problème se pose avec les murs sans texture que l'on peut traverser. Si on autorise de tels murs, alors il convient de trouver autre chose dans ce cas. A vrai dire, je ne me suis pas encore vraiment posé la question. Je pense qu'il faut alors dessiner en premier le mur sur lequel on est, et qu'après, quelque soit l'ordre de récursion l'affichage sera correct. C'est un point à approfondir plus tard et qui ne remet, de toute façon, pas en cause le principe du moteur. C'est juste une singularité, et comme toute singularité, ça gêne affreusement pour pas grand chose $\ddot{\frown}$.

Remarque 2 : En résumé, si l'arbre est un **nœud**, on le parcourt en allant d'abord du côté où se trouve le joueur par rapport à la ligne de partitionnement, puis on s'occupe de cette ligne, et enfin, on descend dans l'arbre de l'autre côté.

Si l'arbre est une **feuille** alors on "dessine" (on va voir comment) toutes les linedefs présentes à cette feuille, le BSP nous garantissant que quelque soit l'ordre de ces murs au sein de la feuille, le résultat à l'affichage sera correct.

6.2 Optimisation du parcours du BSP

Il est possible "d'élaguer" toute une partie des branches du BSP, c'est à dire de supprimer du parcours certaines branches et donc tout le sous-arbre qu'elles peuvent représenter. Pour ce faire, nous rajoutons lors de la création du BSP, comme décrit dans mon précédent papier, des *bounding boxes*.

Une bounding box est en fait une structure de rectangle (donc, les coordonnées de l'extrémité en haut à gauche et en bas à droite de ce rectangle) rajoutée à chaque nœud du BSP. Dans notre modèle de BSP, il y en a 2 par nœud du BSP : 1 pour le fils droit, 1 pour le fils gauche. Le rectangle ainsi défini pour le fils droit se doit de contenir toutes les linedefs que peut récursivement contenir le fils droit du nœud où se trouve la bounding box (et respectivement à gauche pour le fils gauche). Ceci est illustré par la figure suivante :

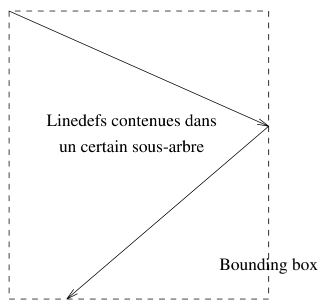

Figure 2: Bounding box et linedefs

Attention : Nous allons dorénavant beaucoup parler de l'angle d'un mur ou d'une linedef. Il m'a été plusieurs fois demandé de préciser *"angle du mur, mais par rapport à quoi ?"*. Notre mur s'inscrit dans un repère orthonormé (la carte du monde) comme nous l'avons clairement vu précédemment. Aussi l'angle d'un mur sera calculé par rapport à la droite d'angle 0 dans le repère, autrement l'axe des x. Pourquoi dis-je que c'est l'axe des x qui représente la droite d'angle 0 ? Tout simplement parce que c'est comme ça en géométrie analytique : il n'y a qu'à se souvenir du cercle trigonométrique. Pour ceux qui l'auraient oublié, j'ai inclu plus loin dans le papier une figure le représentant (il suffit de regarder dans l'index des figures à quelle page elle se trouve ☺). Cette remarque est toujours à conserver à l'esprit.

Quelle est alors la façon "d'optimiser" le parcours de notre BSP ? Eh bien, à chaque fois que nous devons descendre dans un nœud (plus exactement dans un sous-arbre, car il peut s'agir d'une feuille), nous regardons (idéalement) si le champ de vision du joueur est susceptible de traverser la bounding box associée à ce sous-arbre . Si la réponse est non, alors on sait qu'il est tout à fait inutile d'aller calculer avec tous les murs se trouvant dans ce sous-arbre puisqu'ils ne seront pas visibles.

J'ai dit juste au dessus que : "nous regardons *idéalement* si le champ de vision du joueur...". Pourquoi *idéalement* ? Eh bien parce qu'afin d'alléger les calculs, on peut être obligé de faire une approximation. Personnellement c'est mon cas, entre autre parce que je n'ai pas trouvé d'autre méthode plus efficace et pas trop gourmande en temps $\ddot{\smile}$...

Voyons comment fonctionne cette méthode. J'ai appris, plus tard, que c'était une méthode dite *"basée sur les quadrants"*. C'est certainement parce que l'on découpe notre plan en 4 parties, en 4 cas possibles...

Pour comprendre ce qu'est un quadrant, prenons notre plan en 2D. A partir d'un point de celui-ci traçons 2 droites perpendiculaires et passant par ce point. Notre plan se donc retrouve divisé en 4 parties. La première d'entre elles s'étend *angulairement* entre 0 et 90^o, la seconde entre 90 et 180^o, la troisième entre 180 et 270^o et enfin la quatrième entre 270 et 360 (ou $0)^o$. *Chaque partie ainsi construite est un quadrant.* Le point choisi pour initialiser notre "découpage" sera la position du joueur, donc comme nous le verrons plus tard, le point de coordonnées $(0, 0)$ (une transformation se chargera de re-placer le joueur au centre du repère).

Appelons x_1 l'abscisse minimum de notre bounding box, y_1 son ordonnée minimum, x_2 son abscisse maximum et y_2 son ordonnée maximum. Appelons $PlayerX$ et $PlayerY$ les coordonnées de notre joueur dans la carte. Considérons sur le schéma suivant les 4 cas possibles, qui résument en fait toutes les figures envisageables :

La figure précédente illustre les cas où l'on considère que le joueur est susceptible de voir *dans* la bounding box en fonction de sa position et du quadrant dans lequel se situe se direction de vision (représentée par les 4 vecteurs partant du joueur.

Remarque : Nous noterons que ce schéma représentent 4 cas bien séparés : chaque quadrant est concerné par le test de vivibilité si la direction de vision du joueur se trouve dans ce quadrant. Dans le cas contraire, on est sûr qu'il n'y aura pas de visibilité, étant donné que le joueur ne regarde pas dans la direction requise pour le quadrant.

Nous n'avons pas dessiné le cas où le joueur est dans la bounding box, car ce cas est en fait une combinaison des 4 cas décris par le schéma. Les bulles situées aux 4 extrémités du schéma indiquent pour chaque quadrant les relations immuables, (*quelque soit la position du joueur dans ce quadrant*) entre les coordonnées du joueur et les coordonnées de la bounding box. Je pense que le schéma illustre mieux qu'un long discours, donc examinez attentivement le schéma pour vous convaincre des relations entre coordonnées.

Maintenant, attardons-nous sur ces relations. Un problème se pose aux bornes des quadrants. Si l'on considère par exemple les 2 quadrants supérieurs (Nord-Est et Nord-Ouest). Nous remarquons qu'une des relations est commune : $PlayerY < y2$, très bien, mais que l'autre diffère selon le quadrant. Le problème se pose alors, lorsque des coordonnées de la bounding box coïncident avec un bord d'un quadrant (au moins) : laquelle des 2 relations différentes choisir, puisque moralement le bord de 2 quadrants appartient aux 2 quadrants qu'il délimite (il n'y a pas plus de raison que ce bord appartienne plus à l'un qu'à l'autre des 2 quadrants). Pour résoudre ce problème, la solution est simple : on ne testera, pour notre test de visibilité, qu'avec la partie de relation commune à

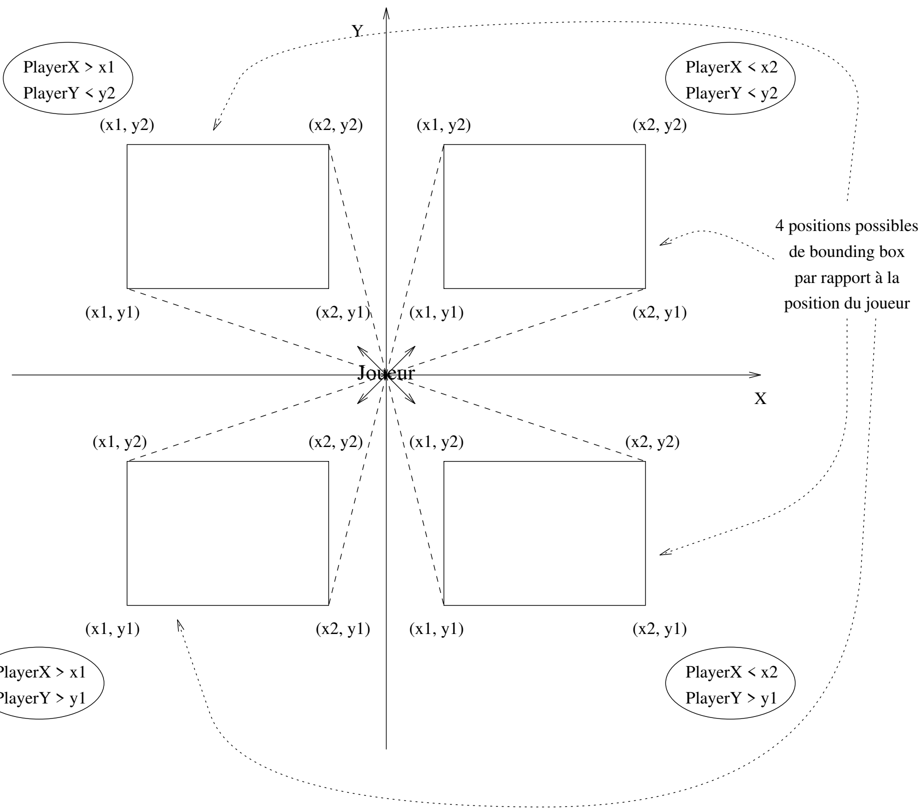

Figure 3: Quadrants et intersections avec bounding boxes

2 quadrants consécutifs dans l'ordre du sens trigonométrique. Autrement dit, nous ne vérifierons qu'une seule des 2 relations d'infériorité ou de supériorité entre les coordonnées du joueur et de la bounding box, par quadrant.

Voici en exemple une fonction C qui se charge de regarder si le cône de vision du joueur se situe dans la bounding box du sous-arbre de droite du nœud actuel (sous-arbre de droite, car nous utilisons la bounding box de droite). Afin de rendre le source plus clair, ci-dessous se trouve la définition du type `Node`.

```
struct Node {
  /* IDENTIFICATEUR DU MUR SERVANT DE LIGNE DE PARTITIONNEMENT */
  int partline ;
  /* COORDNOONES DE LA BOUNDING BOX POUR LE SOUS-ARBRE DE GAUCHE */
  double leftxmin, leftymin, leftxmax, leftymax ;
  /* POINTEUR VERS LE SOUS-ARBRE DE GAUCHE */
  struct Tree *left ;
  /* COORDNOONES DE LA BOUNDING BOX POUR LE SOUS-ARBRE DE DROITE */
  double rightxmin, rightymin, rightxmax, rightymax ;
```

```
  /* POINTEUR VERS LE SOUS-ARBRE DE DROITE */
  struct Tree *right ;
};
```

Notons que les coordonnées $PlayerX$ et $PlayerY$ du joueur sont définies globalement, en dehors de toute fonction afin d'éviter de les avoir en paramètres de toutes les fonctions.

```
extern double PlayerX, PlayerY ;

unsigned char IsRightVisible (Node)
     struct Node *Node ;
{
  double x1 = Node->rightxmin ;
  double y1 = Node->rightymin ;
  double x2 = Node->rightxmax ;
  double y2 = Node->rightymax ;

  if (PlayerLeftAngle < 90)
    {
      if (x2 > PlayerX)
        return TRUE ;
      else
        return FALSE ;
    }
  if (PlayerLeftAngle < 180)
    {
      if (y2 > PlayerY)
        return TRUE ;
      else
        return FALSE ;
    }
  if (PlayerLeftAngle < 270)
    {
      if (x1 < PlayerX)
        return TRUE ;
      else
        return FALSE ;
    }
  if (y1 < PlayerY)
    return TRUE ;
  else
    return FALSE ;
}
```

Remarque : Le test de visibilité pour la bounding box de gauche est *exactement* le même que celui vu ci-dessus, à ceci près qu'au lieu d'utiliser les coordonnées de la bounding box de droite, on utilise celles de la bounding box de gauche, ce qui change juste les 4 première lignes du corps de la fonction en :

```
double x1 = Node->rightxmin ;
double y1 = Node->rightymin ;
double x2 = Node->rightxmax ;
double y2 = Node->rightymax ;
```

Par soucis d'exhaustivité, voici le parcours complet du BSP, écrit en C. Certains points de détail ne seront abordés qu'au moment où nous étudierons la technique de rendu, aussi ne vous butez sur les traitements effectués au cours du parcours, ils seront décrits et deviendront compréhensibles un peu plus tard.

```
/* MACROS DE CASTING D'UN POINTEUR D'ARBRE EN POINTEUR SUR NOEUD OU ARBRE */
#define ASNODE(d) ((struct Node*)d)
#define ASLEAF(d) ((struct Leaf*)d)


void FrontToBackTreeParsing (Tree)
     struct Tree *Tree ;

{
struct Vertex *v1, *v2 ;
if (NbColumnsDone >= 320) return ;

switch (Tree->tag)
  {
  case TAG_LEAF:
    /* SI C'EST UNE FEUILLE, ON LA DESSINE */
    DrawLeaf (ASLEAF(Tree->data)) ;
    break ;

  case TAG_NODE:
    /* C'EST UN NOEUD. */
    v1 = &GlobalVertexTable[(GlobalLinedefTable[
                               ASNODE(Tree->data)->partline].src)] ;
    v2 = &GlobalVertexTable[(GlobalLinedefTable[
                               ASNODE(Tree->data)->partline].dest)] ;
    /* ON CLASSIFIE LE JOUEUR PAR RAPPORT A LA LIGNE */
    /* DE PARTITIONNEMENT ACTUELLE.                  */
    switch (getPointPosition (PlayerX, PlayerY ,v1->x, v1->y, v2->x, v2->y))
      {
      case LEFT:
        /* LE JOUEUR EST A GAUCHE, ON VA DONC A GAUCHE EN PREMIER */
        if ((ASNODE(Tree->data)->left) != NULL)
          if (IsLeftVisible (ASNODE(Tree->data)))
```

```
        FrontToBackTreeParsing (ASNODE(Tree->data)->left) ;
      DrawLinedef (&GlobalLinedefTable[(ASNODE(Tree->data)->partline)]) ;
      if ((ASNODE(Tree->data)->right) != NULL)
        if (IsRightVisible (ASNODE(Tree->data)))
          FrontToBackTreeParsing (ASNODE(Tree->data)->right) ;
      break ;

    case RIGHT:
      /* LE JOUEUR EST A DROITE, ON VA DONC A DROITE EN PREMIER */
      if ((ASNODE(Tree->data)->right) != NULL)
        if (IsRightVisible (ASNODE(Tree->data)))
          FrontToBackTreeParsing (ASNODE(Tree->data)->right) ;
      DrawLinedef (&GlobalLinedefTable[(ASNODE(Tree->data)->partline)]) ;
      if ((ASNODE(Tree->data)->left) != NULL)
        if (IsLeftVisible (ASNODE(Tree->data)))
          FrontToBackTreeParsing (ASNODE(Tree->data)->left) ;
      break ;

    case ON:
      /* ON EST PILE SUR LA LIGNE DE PARTITIONNEMENT */
      /* ON DESSINE ALORS JUSTE CELLE-CI.            */
      DrawLinedef (&GlobalLinedefTable[(ASNODE(Tree->data)->partline)]) ;
      break ;
    }
  }
}
```

6.3 Le dessin (affichage) des murs

Nous avons dit dans l'algorithme simplifié de parcours du BSP (c.f section 6.1) qu'il fallait *Communiquer à la routine d'affichage le mur...* Qu'est-ce que cela signifie et comment ça marche ? C'est exactement ce que nous allons esquisser ici. Contrairement à certains moteurs qui procèdent par *ray-casting* (section suivante : 6.3.1) pour construire les murs à afficher, nous allons procéder par *affichage de polygones*.

6.3.1 Rappel sur le ray-casting

Dans un tel modèle, on choisit l'angle d'ouverture représentant le champ de vision du joueur. Ensuite, on découpe l'écran en autant de colonnes qu'il y a de points horizontalement (en mode VGA 320x200, cela correspond donc à 320 colonnes).

Puis, pour chaque colonne en partant de la gauche (par exemple, mais on dessine souvent de la gauche vers la droite), en fonction du champ de vision choisi, on calcule l'angle du rayon partant de l'œil du joueur et passant par cette colonne écran[2]. On en déduit, sous une forme ou sous une autre,

[2]L'angle d'un rayon de vue est calculé exactement sur le même principe que celui d'un mur. C.f remarque précédente de la section *"6.2 Optimisation du parcours du BSP"*.

l'équation de la droite représentant ce rayon (ou bien les coordonnées de 2 points appartenant à cette droite) et on recherche avec quel mur le plus proche ce rayon intersecte (si on a aussi 2 points appartenant à chaque mur, c'est réalisé par un calcul assez simple déjà étudié dans mon précédent papier sur les BSP trees). Une fois ceci déterminé, on calcule la distance de l'œil au mur trouvé et par un petit calcul de perspective, on en déduit la hauteur de la “tranche” de mur devant être inscrite dans cette colonne écran, et il ne reste plus qu'à dessiner cette tranche (avec, en effet, le mapping, mais nous verrons ceci plus tard).

On recommence ensuite pour la colonne suivante, et ainsi de suite jusqu'à temps d'avoir rempli l'écran. Ainsi, on dessine la vue “tranche verticale” par “tranche verticale”.

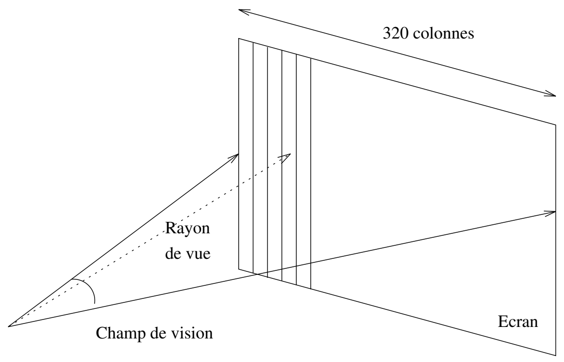

Figure 4: Principe du ray-casting

L'utilisation du BSP permet donc de trouver les murs les plus proches en premier. Ainsi, on peut réduire nos calculs d'intersection car, dès qu'on aura trouvé que notre rayon coupe un mur, le parcours du BSP décrit précédemment, nous garantit que c'est le mur le plus proche.

Tout ceci nous amène à une remarque : si notre écran a 320 colonnes, on va projeter 320 rayons, rechercher 320 intersections (au minimum, car il se peut qu'aux feuilles de notre BSP, il y ait plusieurs murs. Même si on est sûr qu'il n'y aura qu'un seul mur ayant une intersection réelle avec notre rayon parmi tous les murs présents à la feuille, on risque quand même de faire plusieurs calculs de recherche d'intersection infructueux avant de tomber sur le bon mur). Tout ceci signifie aussi que l'on va faire *320 parcours de notre arbre BSP* (pas forcément complet à chaque fois, certes). Qui plus est, comme la position du joueur n'aura pas bougé entre le dessin de chacune des colonnes, on va faire *320 fois le même parcours* ! C'est donc complètement inefficace...

Nous devons chercher à exploiter le fait que tous ces parcours sont identiques, et donc n'en faire qu'un seul. C'est ce que nous allons parvenir à réaliser avec notre méthode de rendu par polygones.

6.3.2 Rendu polygonal

Cette méthode de rendu est plus complexe que celle du ray-casting, mais va nous permettre de parcourir notre BSP au *maximum* une seule fois. Le principe est relativement simple, même si sa réalisation nécessite quand même un travail conséquent.

Nous allons toujours utiliser la méthode de parcours du BSP décrite ci-dessus. Au lieu de rechercher pour chaque colonne écran quel est le mur que l'on voit, nous allons prendre chaque mur trouvé au fur et à mesure du parcours du BSP et déterminer quelle est sa portion visible à l'écran.

Quelques calculs de perspective nous donneront ainsi les positions de début et de fin de ce mur horizontalement à l'écran, ainsi que la hauteur de ce mur pour chaque colonne écran. Ces données définiront donc un polygone à dessiner (éventuellement en le clippant).

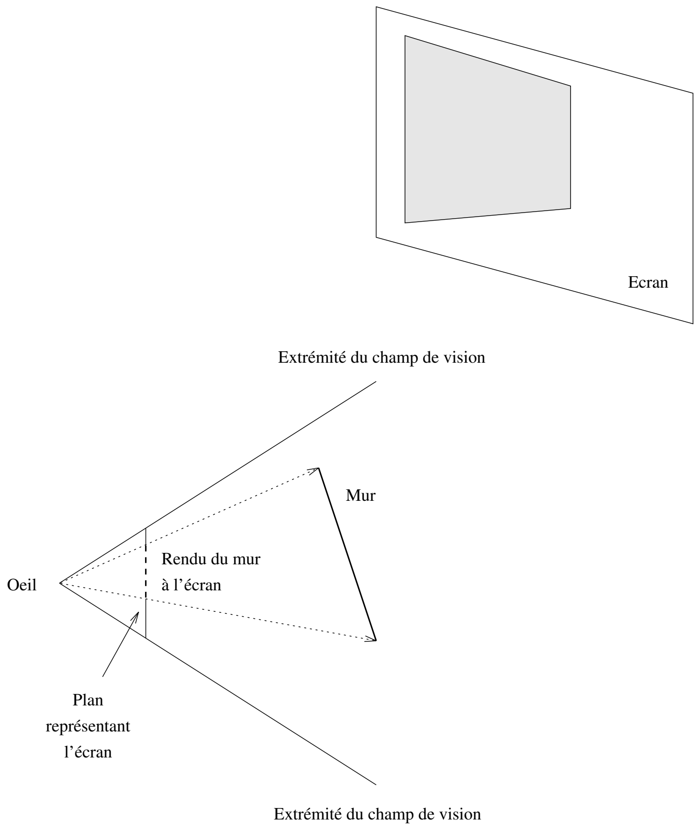

Figure 5: Principe du rendu polygonal

Comme nous recevons, du parcours du BSP, les murs les plus proches en premier, il convient de se souvenir au moment où on dessine le polygone, si on n'a pas déjà dessiné, à cet endroit, un mur plus proche. Pour y parvenir, on va dessiner nos polygones verticalement (par "tranche", un peu comme dans le ray-casting), et on va noter dans un tableau, aussi large qu'il y a de colonnes à l'écran, si on a déjà dessiné une tranche de mur dans chaque colonne. Si c'est le cas, alors on ne re-dessine rien dans cette colonne, sinon on dessine le morceau de polygone et on marque dans notre tableau que la colonne est maintenant occupée par un morceau de mur.

Afin de savoir quand s'arrêter, on va conserver un compteur incrémenté à chaque fois que l'on dessine réellement une tranche de mur. Une fois que ce compteur est égal au nombre de colonnes de l'écran, on sait que celui-ci est complètement rempli par les murs du monde (puisque l'on ne dessine pas plusieurs fois une même colonne), et on peut donc s'arrêter.

Ceci nous évite éventuellement de continuer à parcourir le BSP pour rien. On voit clairement que l'on dessine les mur en parallèle avec le parcours du BSP, ce qui correspond bien à la logique de parcours énoncée dans l'algorithme de la section 6.1 (on y disait : *Communiquer à la routine* *d'affichage le mur...*).

La figure 5 nous montre comment le polygone représentant le mur à l'écran est déterminé. Le schéma représente en quelque sorte une vue du dessus. Le plan de l'écran est matérialisé par le segment qui s'inscrit entre les 2 rayons extrémités du champ de vision. Nous verrons plus tard comment ce champ de vision (sa taille angulaire) est déterminée. Le "morceau d'écran" indiqué en ligne pointillée correspond à la partie de cet écran qui sera "peinte" par le mur. C'est donc la *projection* de ce mur sur l'écran. Nous verrons aussi plus tard comment la hauteur du polygone est calculée pour donner l'effet de hauteur et de perspective à notre mur.

Notre système de tableau pour mémoriser si l'on a déjà affiché un morceau de mur dans une colonne est une simplification drastique du principe du Z-buffer[3]. La figure suivante (figure 6), montre comment ce tableau est utilisé de façon à éviter de réécrire dans une colonne où un mur plus proche a déjà été dessiné. Par définition du parcours du BSP, le mur n^{o}1 nous sera communiqué avant le mur n^{o}2, puisque, *rappelons-le*, le parcours que nous avons choisi pour notre arbre nous donne les murs dans le sens d'éloignement croissant de l'œil du joueur, quelque soit la position de ce dernier.

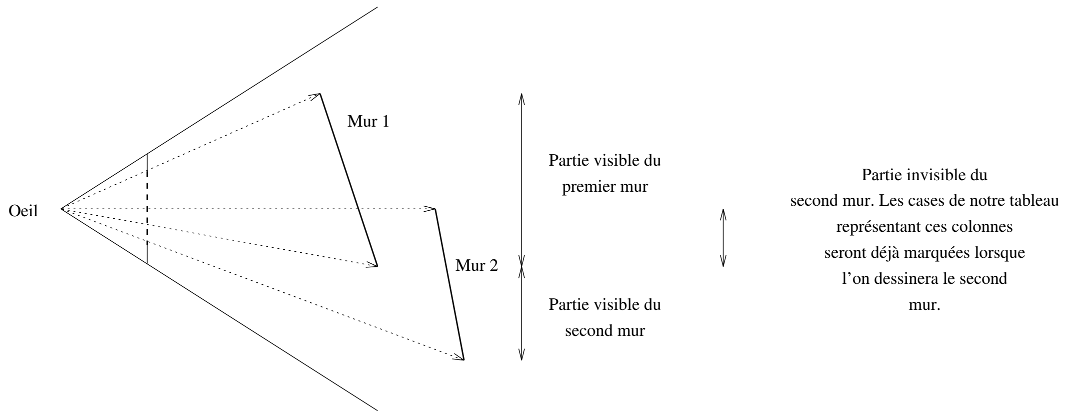

Figure 6: Superposition de murs

Nous connaissons maintenant la manière dont nous allons dessiner nos murs. La raison pour laquelle que nous ne traitons pas les secteurs de hauteur différente apparaît clairement ici. Si nous avons des murs de hauteur différente, alors dans une même colonne écran, nous pouvons devoir dessiner des morceaux de plusieurs murs.

En effet, sur la figure 7 (voir page suivante), nous voyons clairement que la colonne écran matérialisée par la ligne verticale épaisse sera formée du dessin de plusieurs murs (et de plusieurs planchers si on prenait en compte le texturage des sols et plafonds). Donc une simple structure de tableau avec 1 case représentant 1 colonne écran ne suffit plus pour savoir si cette colonne a déjà été dessinée.

[3]Le Z-buffer est une technique qui consiste à conserver un tableau aussi grand que l'écran et à y inscrire, pour chaque point écran dessiné, son éloignement de l'œil. Ainsi, si on veut écrire un point à un endroit de l'écran, on va comparer l'éloignement de ce point avec celui du point déjà dessiné à l'écran. Si le premier est plus près de l'œil que le second, alors on réécrit par dessus le second et on change la valeur d'éloignement de ce point dans la Z-buffer. Sinon, on ne dessine pas notre point car cela signifie qu'il est plus loin, donc derrière le point déjà existant dans l'image.

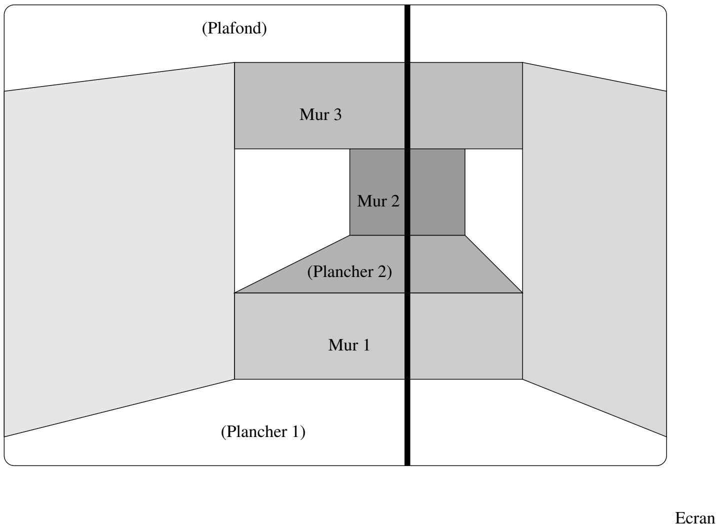

Figure 7: Secteurs de hauteur différente

7 Comment implémenter le rendu polygonal d'un mur

Nous allons maintenant examiner plus en détail comment réaliser cette méthode de rendu de la vue. Le cœur de ce mécanisme est la projection d'un mur sur l'écran afin de trouver quelle place il peut bien occuper sur notre écran, qui en fait, représente quasiment virtuellement notre œil.

Remarque importante : Nous partons donc du principe que le parcours du BSP nous a amené à un mur que nous devons dessiner. Donc notre matériel est ici *1 mur*. Le processus que nous allons décrire en détail se répétera donc pour les autres éventuels murs que l'on sera amenés à trouver lorsque l'on continuera (si nécessaire) notre parcours du BSP afin de remplir l'écran.

7.1 Précisions sur le mécanisme de vision / projection

Nous avons vu que nous cherchions donc à projeter un mur situé dans notre plan 2D (la carte, dans un jeu à la *Doom*), sur notre écran. Puisque la $3ème$ dimension (la hauteur) est rendue après, nous avons à faire à un problème en 2 dimensions. Ce problème est donc bien de *trouver les coordonnées de projection d'une ligne (le mur vu du dessus) sur une autre ligne (le plan de l'écran lui aussi vu du dessus)*. C'est exactement le problème illustré dans la figure 5 (voir plus haut).

Peut-être ai-je été un peu vite lors de la présentation de ce schéma, aussi vais-je m'y attarder un peu plus afin de préciser des trivialités pour certains et des éclaircissements pour d'autres.

Dans l'explication du mécanisme de vision couramment adopté, tous les rayons de lumière qui forment l'image que notre œil voit (discutons pour 1 œil même si nous ne sommes pas des Cyclopes. Le fait d'avoir 2 yeux nous permet en plus de ressentir l'effet de relief) convergent justement vers notre œil. La provenance de ces rayons est limitée par l'angle d'ouverture de notre champ de vision, ce qui explique que nous ne pouvons pas voir derrière nous (car notre angle d'ouverture n'est pas de 360^o !).

On peut matérialiser ceci par le schéma suivant (sous-ensemble de la figure 5 bien évidemment) :

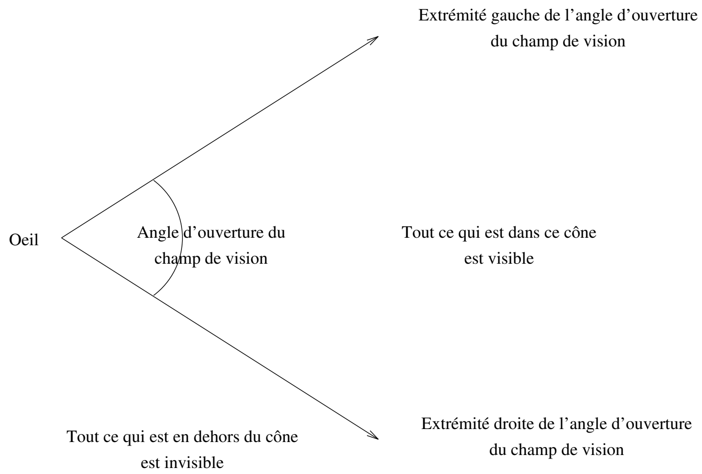

Figure 8: L'œil et la vision

Notre écran est censé nous faire parvenir ce que voit quelqu'un se trouvant dans notre monde virtuel. C'est un plan placé entre le monde virtuel et notre œil. Puisque nous représentons notre monde sur cet écran, nous formons une image avant notre œil, et c'est ce dernier (ou plutôt notre cerveau) qui va se charger de faire le reste du travail pour que nous ayons l'impression que l'image projetée sur l'écran est l'image projetée sur notre rétine. C'est donc sur cet écran (donc sur un plan) que l'on va représenter toutes les informations visuelles que l'on autorise à voir au joueur.

Je conçois bien que la différence entre l'œil du personnage virtuel (celui se trouvant réellement dans le monde) et l'œil du joueur réel (vous par exemple) soit difficile à appréhender. Aussi vais-je essayer de la dire d'une autre façon, au cas où la première explication ne vous aurait pas convaincu.

Votre œil (ou plutôt vos yeux) possèdent un certain champ de balayage. Si vous pouvez voir quelque chose dans la vie courante à un instant donné, c'est que ce quelque chose se trouve dans le cône de vision défini par votre champ visuel. Bon. Maintenant, notre moteur va se charger de former des scènes, celles qu'un être virtuel doit voir quand vous le contrôlez dans votre jeu. Pour que vous sachiez ce que vous faites, le moniteur vous transmet alors une image, donc celle formée par notre moteur graphique (comme ça vous savez s'il y a un monstre, dans quelle piece vous êtes, si vous risquez de tomber ou si un mur vous bloque). Mais il faut bien voir que par cette image, *vous voyez ce qu'un autre voit.* Aussi le champ visuel que votre écran vous fournit est exactement celui que l'on considère que le personnage virtuel possède. C'est ensuite votre cerveau qui, en se concentrant sur l'écran, vous donnera le sentiment de fusionner votre propre champ visuel avec celui que l'image à l'écran vous impose.

En résumé, le seul champ de vision qui nous intéresse est toujours celui du personnage virtuel, celui que vous contrôlez en jouant. Votre propre champ visuel ne nous intéresse pas en terme de rendu de la scène car il est en quelque sorte en dehors de l'univers que l'on modélise en fabriquant l'image (et oui ! votre œil n'appartient pas à la description du monde dans lequel vous faites évoluer votre personnage en jouant. . .) Ceci signifie que sur tous nos schémas représentant un champ de vision, avec un œil et le plan de l'écran, toute la partie se trouvant entre cet œil et ce plan de l'écran n'existant pas réellement. C'est juste une prolongation théorique permettant d'aboutir aux calculs régissant la représentation mathématique de notre monde à l'écran.

Le champ visuel du joueur virtuel (c'est à dire pas celui du gars qui est derrière son ordinateur en train de jouer, puisque cet angle d'ouverture-là, lui, ne change grosso modo pas) permet donc de déterminer la "taille" du balayage visuel que l'on s'autorise à plaquer sur l'écran. Autrement dit, dans tous les cas, les rayons-extrémités du champ de vision passeront toujours par les extrémités droite-gauche du plan que représente l'écran. Ceci explique pourquoi le segment représentant le plan de l'écran vu du dessus est clairement inscrit entre les 2 rayons extrémités. L'angle d'ouverture, c'est à dire l'angle que forment ces 2 rayons extrémités est donc fonction de la distance (distance focale) à laquelle on considère que l'œil se trouve du plan représenté par l'écran, comme le monte la figure ci dessous (figure 9).

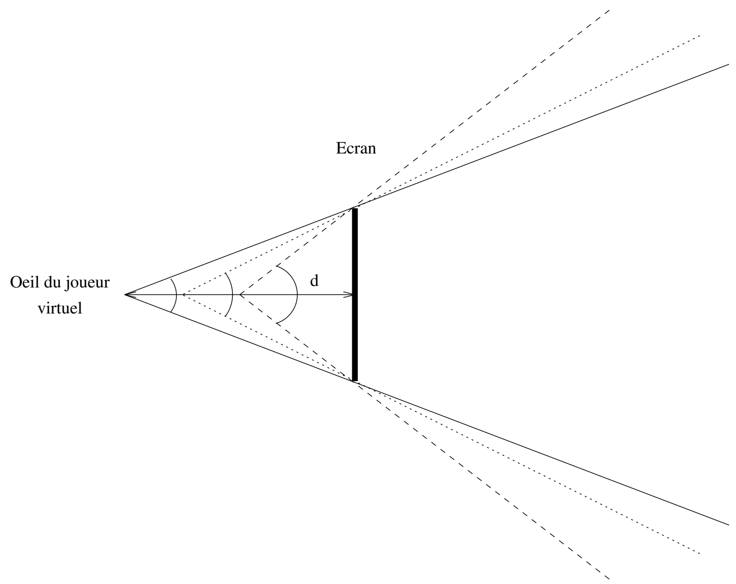

Figure 9: Relation entre l'angle d'ouverture et la distance focale

Remarque : Tout d'abord, il est bien clair que nos schémas représentent une vue du dessus de quelqu'un qui regarde devant lui. Autrement dit, le segment représentant l'écran (segment vertical sur le papier) représente la vue horizontale de notre personnage. Dit encore d'une autre façon, l'axe pieds-tête de notre personnage est perpendiculaire à la feuille de papier que vous êtes en train de lire, avec la tête vers vous, et les pieds à l'opposé. Je sais que ça peut paraître évident, mais mieux vaut 2 précisions qu'une seule...

Soyons bien d'accord que la taille de notre écran est fixe. Donc, la seule façon de faire varier notre angle d'ouverture est de modifier la distance d, distance à laquelle se trouve l'œil virtuel du personnage virtuel se trouvant dans le monde virtuel (Ouf, que de virtualité !!!)...

Nous voyons donc que plus la distance diminue, plus l'ouverture augmente, et inversement bien évidemment.

A titre indicatif, nous prendrons une ouverture de champ de vision de 90^o, c'est à dire 45^o de part et d'autre de l'axe central de vision. Pourquoi 90^o ? Parce que ça donne un bon effet visuel : on n'a pas l'impression d'avoir un champ de vision trop large ni trop étroit... Nous voyons bien ici la différence entre le champ visuel que l'on attribue à notre personnage virtuel et le vôtre. En effet, le vôtre, comme celui de la majorité des être humains $\ddot{\smile}$, est largement supérieur à 90^o (par contre, je n'ai pas réussi à trouver sa valeur dans le dico).

7.2 Premiers préliminaires à la projection d'un mur

Avant de pouvoir réaliser nos projections et nos calculs savants, nous allons faire en sorte que les coordonnées du joueur coïncident avec l'origine du repère et que son rayon central de vision soit aligné sur l'angle 0 dans le cercle trigonométrique. Rappelons les caractéristiques de ce cercle sur la figure n^o10.

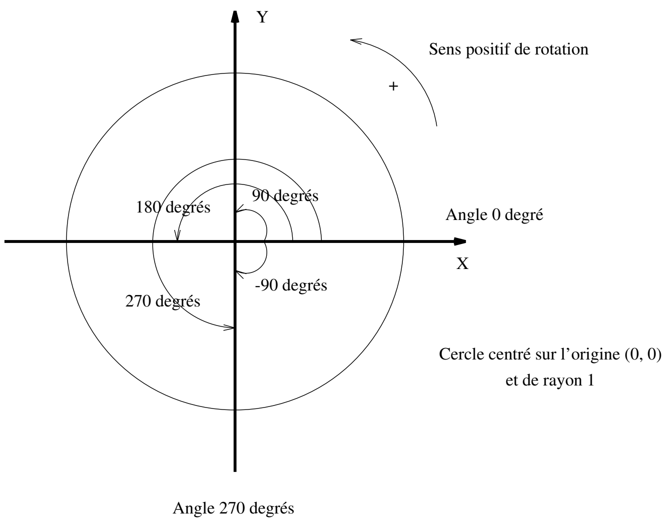

Figure 10: Le cercle trigonométrique

Par la même occasion, rappelons dans la figure 11 (page suivante), les relations trigonométriques dans ce cercle pour un vecteur d'angle θ.

La transformation que nous avons décidé de faire va nous permettre d'utiliser les relations métriques dans le triangle rectangle, les relations trigonométriques et simplifier après coup tous nos calculs. Nous commençons donc par faire un changement de repère composé de :

- 1 translation, pour faire en sorte que les coordonnées de notre personnage correspondent à l'origine du repère (2D rappelons-le) dans lequel on évolue.
- 1 rotation, pour faire en sorte que le rayon central de vision du personnage soit d'angle 0. Du fait de notre translation ci-dessus, le rayon central de vision sera donc superposé avec l'axe

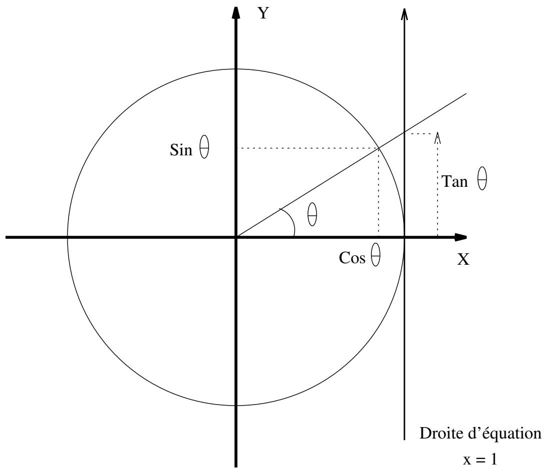

Figure 11: Relations trigonométriques dans le cercle

des abscisses (l'axe des x) de notre cercle trigonométrique.

Remarque : Dans certains ouvrages, l'axe que je représente par l'axe des abscisses (donc des x) est changé de nom et est nommé l'axe des z, puisqu'il s'agit en fait de la notion de profondeur de l'objet considéré par rapport à l'œil du joueur. Personnellement, je préfère rester avec les conventions de géométrie analytique par rapport au cercle trigonométrique. C'est juste une question de convention. Je part en fait du principe que l'on passe notre temps à raisonner avec des objets en 2D, par conséquent repérés dans un repère orthogonal conventionnel, à savoir ressemblant à :

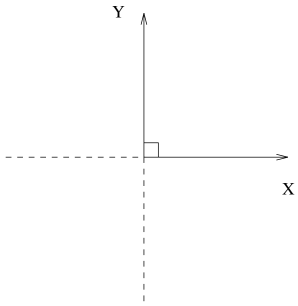

Figure 12: Repère 2D conventionnel

Il n'y a donc pas de raison de changer la place des axes. En résumé, pour moi, la composante z est et reste la *HAUTEUR*.

Les calculs à effectuer sont donc simples. Ils consistent à appliquer à chaque extrémité du mur sur lequel on travaille (donc sur 2 points de notre plan en 2D) la transformation décrite ci-dessous.

Soient :

- $PlayerX$, $PlayerY$ les coordonnées du joueur
- $PlayerAngle$ l'angle de la direction du joueur
- x_1, y_1 les coordonnées de départ du mur
- x_2, y_2 les coordonnées d'arrivée du mur

Appelons :

- x'_1, y'_1 les coordonnées de départ du mur après transformation
- x'_2, y'_2 les coordonnées d'arrivée du mur après transformation

On a alors :

$$x'_1 = (x_1 - PlayerX) * \cos(-PlayerAngle) - (y_1 - PlayerY) * \sin(-PlayerAngle) \tag{1}$$

$$x'_2 = (x_2 - PlayerX) * \cos(-PlayerAngle) - (y_2 - PlayerY) * \sin(-PlayerAngle) \tag{2}$$

$$y'_1 = (y_1 - PlayerY) * \cos(-PlayerAngle) + (x_1 - PlayerX) * \sin(-PlayerAngle) \tag{3}$$

$$y'_2 = (y_2 - PlayerY) * \cos(-PlayerAngle) + (x_2 - PlayerX) * \sin(-PlayerAngle) \tag{4}$$

Ces équations représentent une rotation d'angle $-PlayerAngle$ et une translation de $-PlayerX$ en abscisse et de $-PlayerY$ en ordonnée. Examinons le raisonnement qui aboutit à de tels calculs. Nous partons donc avec les données illustrées par le schéma suivant :

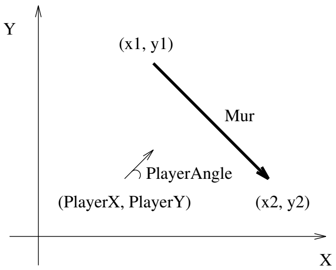

Figure 13: Repère avant transformations

Sur la figure ci-dessus, le joueur est représenté par une flèche dedans. Cette flèche représente dans quelle direction le joueur regarde. Cette flèche forme donc avec l'axe des abscisses (l'axe des x) un angle $PlayerAngle$. Le mur est également représenté par un segment fléché. Cette flèche permet de schématiser l'orientation du mur et donc de savoir quel est son point de "départ" et son point "d'arrivé".

Nous devons effectuer une translation du repère pour que le joueur se trouve à l'origine. Les nouvelles coordonnées du joueur sont donc $(0, 0)$. Reste maintenant à calculer les nouvelles coordonnées des extrémités du segment que représente le mur. Nommons-les $(x_{1recentré}, y_{1recentré})$ et $(x_{2recentré}, y_{2recentré})$. On a alors :

- $x_{1recentré} = x_1 - PlayerX$
- $y_{1recentré} = y_1 - PlayerY$
- $x_{2recentré} = x_2 - PlayerX$
- $y_{2recentré} = y_2 - PlayerY$

Le résultat de cette première partie de transformation est illustré par le schéma ci-dessous (n^{o}14).

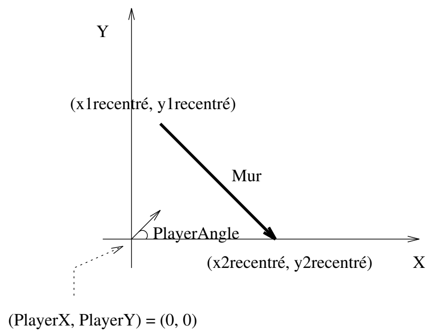

Figure 14: Repère après translation

Maintenant, nous devons faire une rotation d'angle $-PlayerAngle$ pour que la direction centrale de vue du joueur coïncide avec l'angle 0, c'est à dire avec l'axe des x. Pour ceci, nous utilisons la formule de rotation d'angle θ dans le plan, à savoir :

$$x_{rot\theta} = x\cos(\theta) - y\sin(\theta) \quad (5)$$

$$y_{rot\theta} = y\cos(\theta) + x\sin(\theta) \quad (6)$$

Puisque nous devons combiner rotation et translation, nous devons appliquer notre rotation aux coordonnées issues de la translation précédemment effectuée. Nous obtenons donc le résultat final :

- $x'_1 = x_{1recentré}\cos(-PlayerAngle) - y_{1recentré}\sin(-PlayerAngle)$
- $y'_1 = y_{1recentré}\cos(-PlayerAngle) + x_{1recentré}\sin(-PlayerAngle)$
- $x'_2 = x_{2recentré}\cos(-PlayerAngle) - y_{2recentré}\sin(-PlayerAngle)$
- $y'_2 = y_{2recentré}\cos(-PlayerAngle) + x_{2recentré}\sin(-PlayerAngle)$

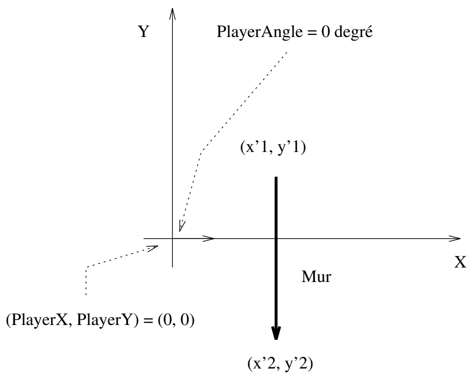

Figure 15: Repère après translation et rotation

En C (avec $NewXStart$, $NewXEnd$, $NewYStart$, $NewYEnd$ à la place de x'_1, x'_2, y'_1, y'_2), ceci aurait donc la forme :

```
NewXStart = ((GlobalVertexTable[Linedef->src]).x-PlayerX)
                                    *
                        cos (TO_RAD(-PlayerAngle))
                                    -
                     ((GlobalVertexTable[Linedef->src]).y-PlayerY)
                                    *
                        sin (TO_RAD(-PlayerAngle)) ;
NewXEnd = ((GlobalVertexTable[Linedef->dest]).x-PlayerX)
                                    *
                        cos (TO_RAD (-PlayerAngle))
                                    -
                     ((GlobalVertexTable[Linedef->dest]).y-PlayerY)
                                    *
                        sin (TO_RAD (-PlayerAngle)) ;

NewYStart = ((GlobalVertexTable[Linedef->src]).y-PlayerY)
                                    *
                        cos (TO_RAD (-PlayerAngle))
                                    +
                     ((GlobalVertexTable[Linedef->src]).x-PlayerX)
                                    *
                        sin (TO_RAD (-PlayerAngle)) ;
NewYEnd = ((GlobalVertexTable[Linedef->dest]).y-PlayerY)
                                    *
                        cos (TO_RAD (-PlayerAngle))
                                    +
                     ((GlobalVertexTable[Linedef->dest]).x-PlayerX)
```

```
                    *
                    sin (TO_RAD (-PlayerAngle)) ;
```

Quelques petites explications sur cette section de code pourront peut-être éclairer quelques interrogations. Tout d'abord, `TO_RAD` est une simple macro qui convertit les angles de degrés en radians. Elle est définie par :

```
#define TO_RAD(x) ((double)((x)*(M_PI/180.0)))
```

Ceci est dû au fait que les primitives trigonométriques $(\sin, \cos, \tan, \ldots)$ fonctionnent en radians. Il est sûr que pour s'éviter ce calcul, on pourrait directement utiliser des radians pour exprimer l'angle de vision du joueur. Mais en réalité, si l'on veut vraiment gagner de la vitesse, il ne faut plus travailler en flottants, mais rester en degrés et faire des tables de sinus, cosinus, etc... pré-calculées, indexées par l'angle voulu, lui-même mesuré en degrés et représenté par un entier. Donc, ne pinaillons pas sur un calcul de plus ou un calcul de moins !

`GlobalVertexTable` est en fait le tableau où je range tous les vertices (tous les points servant de départ ou d'arrivée aux murs). Ils sont rangés linéairement ce qui permet un accès par indexation.

Pour comprendre ce qui est des champs de structures, voici les structures d'un vertex et d'une linedef :

```
struct Vertex {
  double x ;
  double y ;
};

struct Linedef {
  int src ;         /* INDEX DU VERTEX SOURCE */
  int dest ;        /* INDEX DU VERTEX DESTINATION */
  int sector ;      /* INDEX DU SECTEUR AUQUEL APPARTIENT LA LINEDEF */
  int next ;        /* NON UTILISE DANS LE MOTEUR */
  double angle ;    /* ANGLE DU MUR */
  double len ;      /* LONGUEUR DU MUR */
};
```

Certains champs peuvent paraître étranges, mais nous y reviendrons ultérieurement.

Comme on peut le voir, après transformation (figure 15), le joueur se trouve maintenant à l'origine du repère et sa direction de vision est confondue avec l'axe des abscisses. Cette transformation a la particularité de transformer l'axe des abscisses en l'axe qui mesure la profondeur du mur par rapport au joueur, c'est à dire que notre axe des x représente la notion de *devant* le joueur. En effet, si l'on considère le rayon central de vue du joueur, la distance de l'œil au mur sur ce rayon est exactement l'abscisse du point d'intersection de l'axe des x avec le mur.

Il convient maintenant de s'attarder sur la figure obtenue. Quatre cas sont alors possibles, illustrés par la figure suivante :

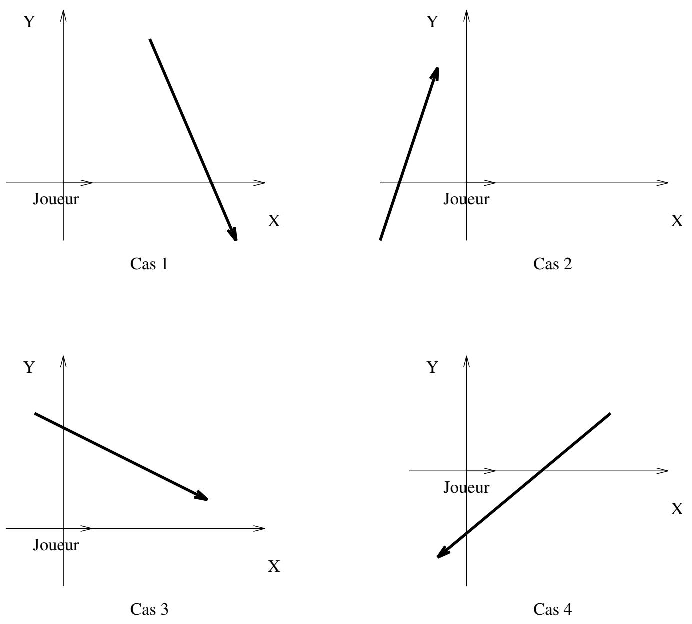

Figure 16: 4 cas possibles après transformation

1. Le mur est complètement devant le joueur
2. Le mur est totalement derrière le joueur
3. Le début du mur est derrière le joueur
4. La fin du mur est derrière le joueur

Dans le second cas, on n'a rien à faire puisque ceci nous dit clairement qu'on n'aura rien à afficher de ce mur dans notre écran puisque ce mur est totalement invisible pour le joueur. Dans les 2 derniers cas, on va devoir "clipper", c'est à dire déterminer la partie réellement visible du mur. Quant au premier cas, il est sympathique puisque nous n'avons pas besoin de clipping, le mur étant totalement visible.

La question que vous vous posez sûrement est *"mais il parle de visibilité sans qu'on ait fait intervenir la notion de champ de vision... Si ça se trouve, même si le mur est complètement devant le joueur, il y en aura une partie que ce dernier ne verra pas car son champ de vision est trop étroit... "*. L'idée de cette question est illustrée par la figure 17 (page suivante).

Bin, c'est une excellente question ...on va y venir un peu plus tard, ceci étant réalisé lors de la phase de projection proprement dite. Avant ceci, nous allons déjà nous débarrasser des cas où le mur est derrière nous et clipper le mur afin d'en retirer les parties se trouvant derrière nous. Pourquoi le faire maintenant ?

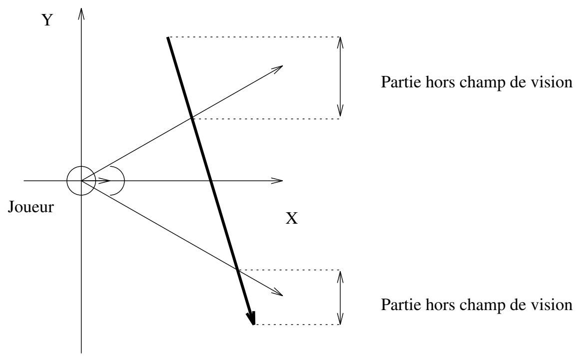

Figure 17: Et le champ de vision dans tout ça ?

Il y a à cela 2 raisons :

- Si le mur est complètement derrière nous, comme il est totalement invisible, il est inutile de se fatiguer à faire de sombres calculs dessus pour s'apercevoir en fin de compte qu'on peut jeter le mur. N'oublions pas que les applications utilisant des moteurs 3D sont généralement assez gourmandes en temps de calcul et qu'il vaut mieux économiser ce dernier afin de garder une animation fluide.
- Nos fameux calculs de projection, que nous allons bientôt voir, ont la particularité de *nécessiter* que chaque point du mur se trouve *devant* le joueur (c'est à dire avec une abscisse positive, même pas nulle car cela nous donnerait une jolie division par 0, mais ça nous le verrons après...)

7.3 Seconds préliminaires avant la projection : le clipping

Nous allons donc clipper notre mur, c'est à dire réduire ses dimensions de telle façon que ses 2 nouvelles extrémités (obtenues après clipping, donc) soient devant le joueur. Plutôt que de dire qu'un point est devant le joueur si son abscisse est supérieure ou égale à 0, nous allons utiliser une *constante* que nous nommerons (fort mal à propos mais elle est notée comme ça dans mon source) : $ZMIN$ et nous dirons qu'un point est *devant* le joueur (donc susceptible d'être visible) si son abscisse est supérieure à $ZMIN$. Pour une raison que nous verrons ultérieurement, je prendrai $ZMIN$ égale à 1.

Si on augmente $ZMIN$, alors cela signifie que l'on refuse de laisser voir des objets se trouvant d'autant plus plus loin. Autrement dit, si on met $ZMIN$ à 50, cela signifie que l'on considérera que tout point se trouvant à moins de 50 unités en avant de l'œil du joueur est invisible. Comme effet de bord, ceci crée un petit cadre autour de l'affichage, cadre correspondant à la partie écran qui aurait du représenter les objets inclus dans le champ de vision et se trouvant à une distance inférieure à 50 unités de l'œil du joueur.

Remarque : Notons que nous n'avons toujours pas fait intervenir la notion d'angle de champ de vision. $ZMIN$ est tout à fait indépendant de ce champ de vision...

Le clipping se présente alors sous la forme suivante :

```
A = Wall->angle ;

if ((NewXStart < ZMIN) && (NewXEnd < ZMIN))
    return ;

if (NewXStart < ZMIN)
  {
    NewYStart = NewYStart + (ZMIN-NewXStart) * tan(A) ;
    NewXStart = ZMIN ;
  }

if (NewXEnd < ZMIN)
  {
    NewYEnd = NewYEnd + (ZMIN-NewXEnd) * tan(A) ;
    NewXEnd = ZMIN ;
  }

if (NewXStart > 9999)
    NewXStart = 9999 ;

if (NewXEnd > 9999)
    NewXEnd = 9999 ;
```

Nous faisons 4 choses dans ce morceau de code. Tout d'abord, nous éliminons les murs se trouvant "derrière" nous (c'est à dire dont l'abscisse est inférieur à $ZMIN$) par le premier test. Ensuite, les 2 tests suivants ont rapport avec les cas où 1 seule extrémité du mur est derrière nous. Dans ces cas, nous devons ramener cette extrémité devant nous. Considérons le premier de ces cas sur un schéma (figure 18) pour comprendre les calculs, c'est à dire celui où c'est le point de départ du mur qui se trouve invisible.

Notons que nous utilisons la tangente de l'angle du mur. Celle-ci peut directement être obtenue à partir des coordonnées des points de départ et d'arrivée *originaux* du mur par :

$$\tan(Mur) = \frac{y_{end} - y_{start}}{x_{end} - x_{start}} \tag{7}$$

L'angle du mur est alors égal à l'arc tangente de ce calcul. Pour éviter de re-calculer à chaque fois l'angle du mur, celui-ci est directement inscrit dans la structure du monde par le compilateur de BSP.

Pour terminer, nous "clippons en profondeur". C'est à dire que si un objet est "trop loin devant" le joueur (être trop loin signifiant donc avoir une abscisse supérieure à une certaine valeur), alors on va restreindre son éloignement à une valeur arbitraire fixe (ici 9999). A quoi ça sert ? Eh bien à deux choses. Tout d'abord, ça va nous permettre de limiter le "nombre de distances possibles" entre un objet et l'œil, autrement dit, pour certains calculs faisant intervenir l'éloignement œil-objet, il sera possible de pré-calculer tout ou partie des résultats dans des tables. Ensuite, ceci permet de donner aux objets très éloignés une même hauteur quelque soit leur distance. Dans la réalité, c'est bien ce qu'il se passe. Au bout d'une certaine distance, tous les objets ont sensiblement la même taille.

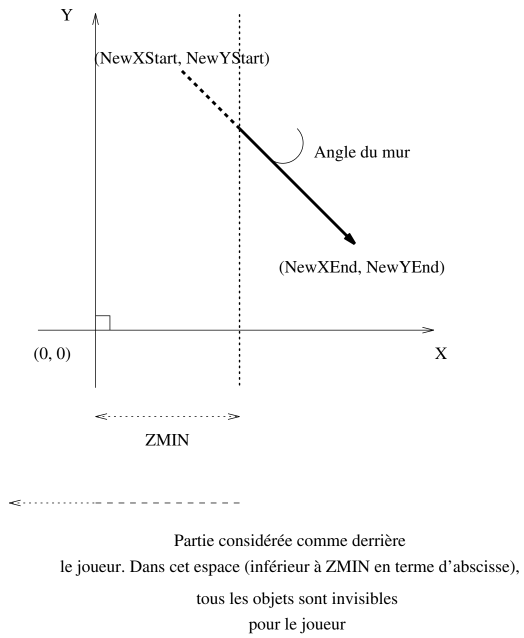

Figure 18: Clipping du mur

7.4 La projection horizontale

Nous voici enfin rendus au moment tant attendu, la projection de notre mur sur notre écran. Ceci va nous permettre de déterminer les *coordonnées des extrémités de notre mur sur l'écran* en fonction de l'effet de perspective... et du champ de vision. Ces coordonnées écran seront peut-être en dehors de l'écran physique, aussi devrons-nous éventuellement les clipper elles aussi. J'appelle *la projection horizontale*, le calcul qui va nous fournir les extrémités du mur sur l'écran, *horizontalement*, c'est à dire pas dans le sens de la hauteur. On va donc déterminer à quelle *colonne écra*n commence le mur et à quelle *colonne écran* il termine. Les calculs de hauteur, c'est à dire ceux qui nous détermineront les coordonnées verticales de début et de fin de mur, seront effectués plus tard. Ce sera la *projection verticale.*

Le principe du calcul de projection horizontale provient de la représentation de la vision que nous avions expliquée précédemment, à savoir que tous les rayons constituant l'image visible par un œil convergent vers un point unique : cet œil, en s'inscrivant dans un cône que représente l'angle d'ouverture du champ de vision (c.f figure 8). Ajoutons à cela la notion d'écran que nous avons aussi vue précédemment et on retombe sur le schéma bien connu :

Le plan de l'écran intercepte donc la vision du joueur virtuel pour que le résultat de celle-ci soit affiché sut notre *écran physique* et nous faire voir ce que l'on verrait si nous étions réellement dans le monde.

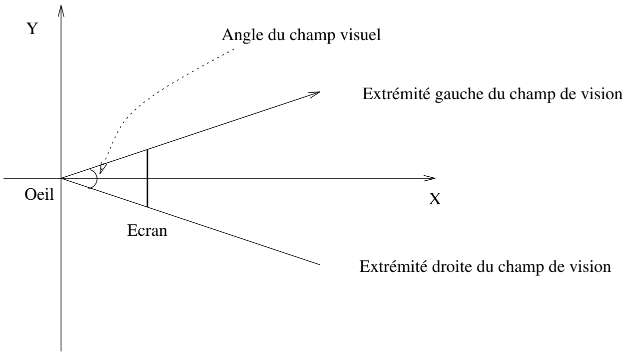

Figure 19: Matériel pour la projection horizontale

Ce qui nous intéresse, c'est de trouver la projection d'un mur sur l'écran, autrement dit de trouver la position sur l'écran (en terme de colonnes écran, rappelons-le) de chacune des extrémités du mur une fois projetée à l'écran. Pour comprendre les calculs, aidons-nous une fois de plus, de petits dessins :

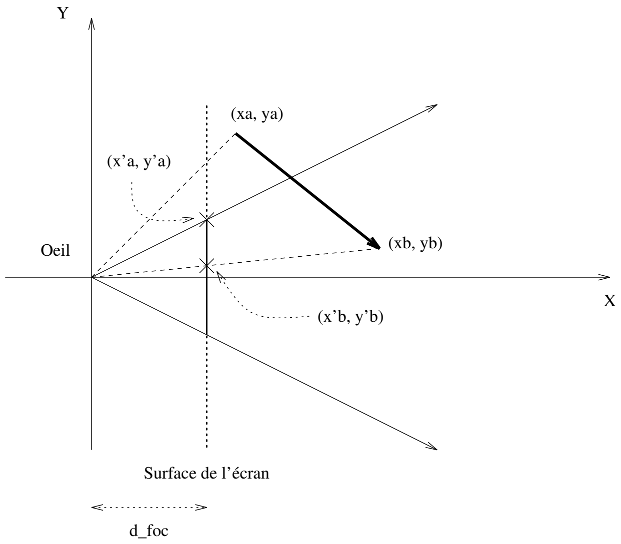

Figure 20: Le problème

La figure 20 illustre ce que nous cherchons et les données que nous avons à notre disposition. Pour simplifier le problème, nous n'allons raisonner que sur 1 seul point (comme l'illustre la figure 21) dont nous noterons les coordonnées par (x, y). Nous noterons les coordonnées de la projection de ce point (x', y'). Une fois que nous saurons les coordonnées de notre projection pour 1 point, nous pourrons appliquer ces calculs à chacune des deux extrémités du mur.

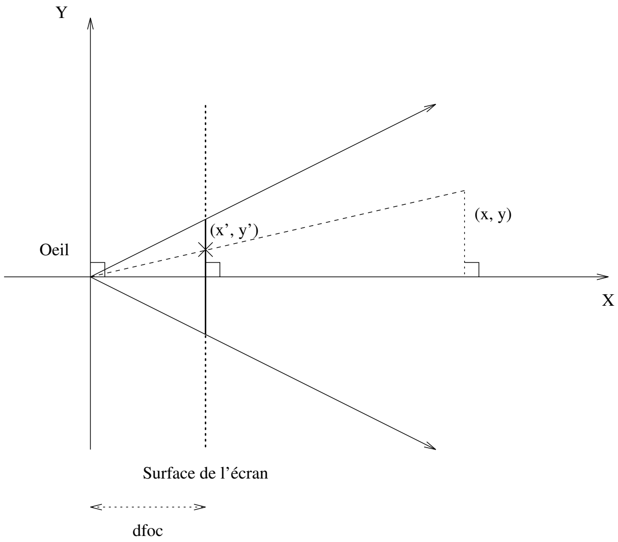

Figure 21: Le principe de le projection horizontale

Par le théorème de *Thalès*, nous avons que :

$$\frac{y'}{dfoc} = \frac{y}{x} \tag{8}$$

Donc nous en déduisons l'ordonnée du point projeté par :

$$y' = \frac{dfoc * y}{x} \tag{9}$$

Remarque 1 : L'abscisse du point projeté est forcément égale à $dfoc$ puisque l'on projette sur la droite d'équation $x = dfoc$. De toute façon, cette abscisse ne nous sert pas car la position horizontale à l'écran nous est clairement donnée par l'ordonnée (se référer à notre schéma) du point projeté.

Remarque 2 : Nous avons dit précédemment que l'angle du champ de vision était intimement lié à la distance focale. Nous n'avons toujours pas eu l'impression d'utiliser ici cette notion de champ de vision. Ceci va venir dans la partie qui suit cette remarque. Par contre, nous avons maintenant les explications suffisantes pour déterminer la valeur de l'angle du champ de vision en fonction de la distance focale que l'on a utilisée. Il suffit pour cela de considérer les rayons extrémités du champ de vision. Par construction, ils passent par les bords de l'écran et par le point origine du repère (c.f figure 22).

Nous pouvons alors utiliser les relations métriques dans le triangle rectangle pour calculer l'angle a (angle de demi-champ de vision) :

$$\frac{x'}{dfoc} = \tan(a) \tag{10}$$

donc :

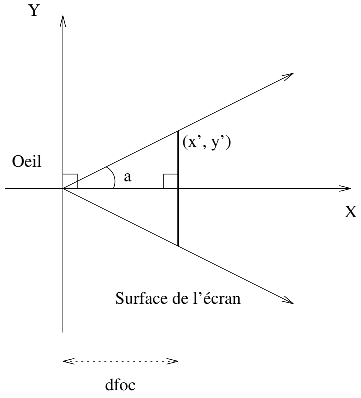

Figure 22: Détermination du champ visuel en fonction de la distance focale

$$a = \tan^{-1}(\frac{x'}{dfoc}) \tag{11}$$

ce qui se note aussi avec l'arc-tangente à la place de $\tan^{-1}$:

$$a = \arctan(\frac{x'}{dfoc}) \tag{12}$$

L'angle d'ouverture du champ de vision est donc $2 * a$. Nous avons dit précédemment que nous prenions un champ de vision de 90^o. Ceci correspond à un demi-champ de vision de 45^o. Or $\tan(45^o)$ étant égale à 1, nous en déduisons que notre distance focale doit être égale à la moitié de la largeur de l'écran (autrement dit, sur notre figure précédente, cela signifie que le côté "adjacent" est de même taille que le côté "opposé" dans notre triangle rectangle).

En résumé, la détermination des coordonnées horizontales-écran des extrémités de notre mur est régie par les 2 calculs suivants :

$$SxStart = 160 - (y'_1) * \frac{(320/2)}{x'_1} \tag{13}$$

$$SxEnd = 160 - (y'_2) * \frac{(320/2)}{x'_2} \tag{14}$$

où, rappelons-le, x'_1 et x'_2 sont les abscisses des extrémités de départ et de fin du mur obtenues après rotation-translation, puis éventuellement clipping dans le repère qui comporte notre carte du monde (celles notées respectivement `NewXStart` et `NewXEnd` dans les morceaux de source C vu jusqu'à maintenant). Pour ce qui est de y'_1 et y'_2, il s'agit bien entendu des ordonnées de ces mêmes points après les mêmes transformations (ordonnées respectivement représentées par `NewYStart` et `NewYEnd` dans nos sources C).

Remarque : Notons que nous devons re-centrer nos coordonnées horizontales-écran car le 0 de ce dernier n'est pas au milieu mais à gauche. C'est la raison de l'apparition de "$160 - \ldots$".

Nous avons dit précédemment qu'il était tout à fait possible que les coordonnées horizontales écran soient en dehors de l'écran physique. Eh bien ceci arrive dans le cas où un point est en dehors du champ de vision. Pour restreindre la vue à notre champ de vision, il ne nous reste plus qu'à clipper (ben oui, encore du clipping) nos coordonnées horizontales écran à la taille réelle de l'écran. Ainsi, la prise en compte du champ de vision sera complète. La raison pour laquelle le clipping reflète exactement la notion de limitation au champ de vision est que par construction, nous avons fait en sorte que les extrémités de l'écran nous transmettent exactement les rayons extrémités du champ de vision…

Le clipping va être réalisé en plusieurs étapes. Tout d'abord, nous allons nous assurer que le mur n'a pas une longueur nulle à l'écran. Si c'est le cas, alors on s'arrête de chercher pour ce mur et on sort de la fonction :

```
if (SxStart == SxEnd)
    return ;
```

Nous avons dit, bien avant, que du fait de l'orientation choisie (arbitrairement certes, mais il fallait bien en choisir une) pour nos murs, à savoir le sens des aiguilles d'une montre, le joueur voyait toujours la face droite d'un mur et que le début d'un mur se trouvait toujours plus à gauche dans le champ visuel du joueur que la fin de ce même mur. Nous allons donc regarder si le mur ne termine pas à gauche de la limite gauche de l'écran (je rappelle que l'on considère toujours un écran de 320x200 avec le point (0, 0) situé en haut à gauche). Si c'est le cas, c'est que tout le mur est hors du champ visuel car "trop à gauche" du joueur, et donc on laisse tomber l'affichage de ce mur, rien n'étant visible sur notre écran physique.

```
if (SxEnd < 0)
    return ;
```

On fait de même en regardant si le début du mur ne se trouverait pas à droite de la limite droite de notre écran physique. Dans ce cas, c'est que le mur se trouve hors du champ visuel parce ce mur est "trop à droite".

```
if (SxStart > 319)
    return ;
```

Conclusion : Maintenant que l'on est sûr qu'une partie, au moins, de notre mur se projette bien sur notre écran, nous devons déterminer exactement cette partie. C'est la phase de *clipping* proprement dite. Celle-ci sera effectuée un peu plus tard, car nous avons besoin indirectement de connaître la hauteur des extrémités du mur préalablement. Aussi, verrons-nous ce clipping au moment voulu. De même, il nous faudra prendre en compte le fait qu'une partie de notre mur peut se trouver derrière un autre mur, plus proche, déjà tracé.

Maintenant *SxStart* et *SxEnd* contiennent les numéros de colonne-écran de début et de fin du mur une fois projeté à l'écran, sans se soucier de savoir si ce mur "rentre" complètement dans l'écran, ni s'il est en réalité totalement visible quant à sa partie se trouvant effectivement à l'intérieur des limites de l'écran. Notons que ces deux variables sont de type entier et non flottant.

Remarque 1 : Il n'est pas ici question de chercher à s'embarrasser de considérations d'optimisations de bas niveau, comme le passage en virgule fixe ou l'écriture de morceaux "chauds" en assembleur. Ceci vient bien après avoir compris les principes qui régissent un moteur, et ont le don de compliquer considérablement l'exposé. Par contre, une fois le moteur compris, il est relativement simple de faire ce genre d'optimisation, même si cela demande un travail conséquent et pas toujours passionnant. A titre indicatif, mon prototype de moteur fonctionne actuellement avec de beaux flottants sur 64 bits sur mon Pentium 150 et il ne s'en plaint pas. Il est sûr que si l'on commence à rajouter beaucoup d'objets dynamiques (nous les verrons plus tard, ce sont entre autres les monstres), et tout ce qui fait un jeu (la gestion de l'intelligence des monstres, la musique, etc...), il risque d'y avoir besoin d'optimiser "un peu" pour conserver un taux de raffaîchissement d'image assez rapide pour avoir une animation bien fluide. Le principe que j'essaye de décrire ici est en lui-même pas mal optimisé, son implémentation (son codage), elle, peut certes bénéficier d'optimisations (par exemple remplacer des divisions entières par des décalages quand c'est possible, remplacer des indexations par des calculs sur des pointeurs, nombres en virgule fixe, assembleur, inlining, etc...), mais tout ceci n'a plus de rapport proprement dit avec le *principe du moteur*, et donc ne nous intéresse pas (du moins dans ce papier d'initiation).

Remarque 2 : Nous voyons que dans nos 2 calculs de projection, nous multiplions y'_1 (respectivement y'_2) par $\frac{(320/2)}{x'_1}$ (respectivement par $\frac{(320/2)}{x'_2}$). Nous pouvons pré-calculer une table des valeurs de ce facteur indicée par la valeur du x' (x'_1 ou x'_2 selon le calcul) afin d'économiser un peu de temps de calcul. Comme nous avons clippé l'éloignement en x à 9999 précédemment (on a dit que si $x'_1 > 9999$ alors on forçait x'_1 à être égal à 9999 ; pareil pour x'_2), nous avons besoin d'une table de dimension finie, de 10000 valeurs dirais-je même ! Voilà l'intérêt d'avoir effectué ce clipping tout à l'heure. De plus, faire une telle table possède un intérêt non négligeable car elle va nous servir à nouveau d'ici quelques instants...

7.5 La projection verticale

Il nous faut maintenant nous occuper de savoir quelle hauteur notre mur peut bien avoir du fait de la perspective. C'est donc maintenant que l'on va re-créer la $3^{ème}$ dimension à savoir la hauteur. Nous sommes bien d'accord que cette dimension n'apparaît pas directement sur la carte en 2D !

La hauteur est normalement un attribut du secteur auquel appartient le mur actuel. Pourquoi ne pas avoir rattaché la hauteur aux murs ? Tout simplement parce qu'au sein d'un même secteur, tous les murs ont obligatoirement la même hauteur, donc on fait une économie de place. Dans notre moteur, puisque nous ne gérons pas encore les secteurs de hauteur différente, la hauteur des murs est en réalité un attribut global du niveau. Plus exactement, la hauteur est décomposée en 2 parties :

- L'altitude du plancher : $CeilingH$
- L'altitude du plafond : $FloorH$

La hauteur d'un mur est donc l'altitude de son plafond *moins* (-) l'altitude de son plancher.

Nous allons en fait utiliser *exactement* le même principe que pour la projection horizontale, mais en raisonnant sur la hauteur des murs. Nous voyons donc qu'il existe aussi une notion d'angle de champ de vision vertical. Généralement on parle toujours de l'angle du champ visuel de vue horizontal qui définit le cône de vision gauche-droite, mais il existe aussi un cône de vision haut-bas. Nous allons chercher à ré-utiliser la table pré-calculée que nous avions faite pour les calculs de projection horizontale. Du fait que la hauteur de notre écrann'est plus de 320 pixels de haut, mais de 200, il est clair que l'angle visuel vertical ne sera pas de 90^o. Pour le calculer (mais comme nous ne nous en servirons pas, je ne le fait pas), il suffit d'appliquer exactement le même principe que celui utiliser pour le champ horizontal. Ceci est laissé en exercice au lecteur $\ddot\smile$... (Allez, juste le résultat : si je ne me suis pas planté, ça fait un demi-angle de 32.0053832081^o de part et d'autre de horizon, donc un angle total de champ visuel égal 2 fois ce truc, soit 64^o une fois arrondi).

Au lieu de raisonner avec un schéma plan représentant une vue du dessus comme nous l'avons fait jusqu'à présent, nous allons raisonner avec le même style de schéma, mais représentant une vue de *côté* comme l'illustre la figure 23.

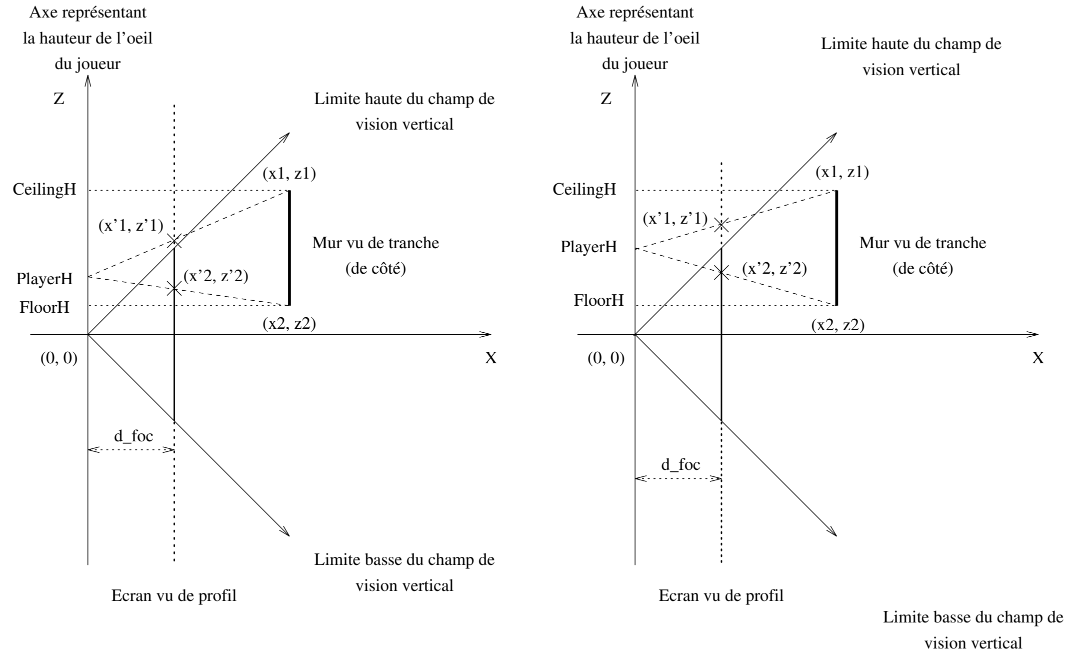

Figure 23: Différentes positions de l'œil et leurs impacts sur la projection verticale

Nous voyons apparaître maintenant un axe z qui représente la hauteur, alors que sur les schémas précédents il s'agissait de l'axe des y. Nous savons que l'œil du joueur se trouve sur l'axe des z (donc à l'abscisse $x = 0$ puisque nous nous sommes arrangés précédemment pour que le repère subisse un décalage de façon à ce que le joueur en occupe l'origine). Par contre, la hauteur du joueur n'est pas spécialement fixe et peut varier selon l'axe des z. Le joueur est donc repéré verticalement par son altitude : $PlayerH$. Selon cette altitude, la projection verticale du mur pourra changer, donnant lieu à des vue verticales différentes d'un même mur.

Pour ce qui est d'expliquer les calculs qui vont venir, nous procéderons comme précédemment, à savoir en examinant le cas de la projection d'un seul point à la fois. Pour illustrer le principe, nous nous attarderons sur le point représentant le haut du mur, point dont nous noterons l'abcisse par x et la “hauteur” (qui correspond sur notre schéma à l'ordonnée) par z (schéma ci-dessous).

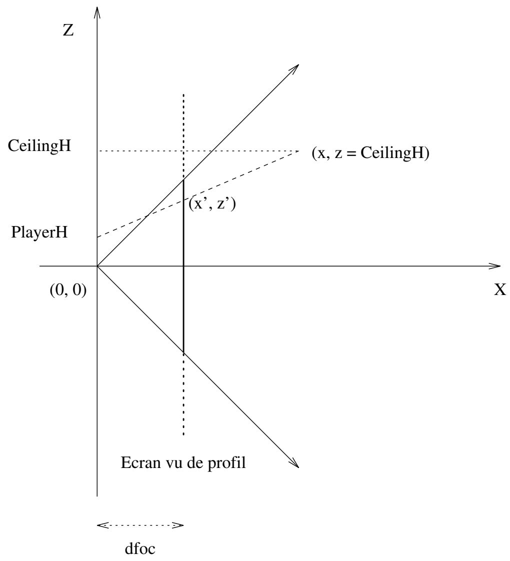

Figure 24: Le principe de le projection verticale

Par les mêmes relations métriques dans le triangle rectangle que précédemment, nous avons que :

$$\frac{CeilingH - PlayerH}{x} = \frac{z'}{dfoc} \tag{15}$$

Donc nous en déduisons que :

$$z' = \frac{(CeilingH - PlayerH) * dfoc}{x} \tag{16}$$

Un raisonnement similaire nous amène au résultat suivant pour le point déterminant le bas du mur :

$$z' = \frac{(PlayerH - FloorH) * dfoc}{x} \tag{17}$$

Nous savons donc maintenant calculer la position-verticale écran d'une tranche de mur en fonction de son éloignement en x. Il ne faut pas oublier de re-centrer tout ça par rapport au milieu vertical de notre écran (qui est d'une taille 200 lignes verticalement, rappelons-le).

Pour le re-centrage, nous allons examiner séparément le cas du point définissant l'extrémité haute du mur et celui représentant le bas de ce même mur.

Clairement, pour le re-centrage du *haut* du mur, si nous appelons α le résultat fourni par l'équation 16 (sans re-centrage, donc), nous devons faire $100 - \alpha$ pour obtenir nos coordonnées re-centrées par rapport au milieu-vertical de l'écran (100 est en fait la hauteur de l'écran divisée par 2). Personnellement, comme je préfère avoir des calculs homogènes (vous verrez cela après que l'on se soit occupé de re-centrer le point définissant le bas du mur), je préfère inverser, dans l'équation 16, la soustraction $CeilingH - PlayerH$ (qui devient alors $PlayerH - CeilingH$) ce qui me donne α' et faire $100 + \alpha'$.

Pour ce qui est de re-centrer le point représentant le *bas* du mur, nous devons directement faire $100 + \alpha$, si on appelle α le résultat donné par l'équation 17.

Mon histoire d'homogénéité se voit donc ici, dans les 2 cas, on fait une addition $(100 + \ldots)$, et on utilise toujours la soustraction $CeilingH - PlayerH$, ce qui permet de ne faire le calcul de cette soustraction qu'une seule fois si l'on veut optimiser le source.

Conclusion : Nous avons donc vu comment calculer les extrémités verticales à l'écran d'une “tranche verticale” de mur, à partir de sa hauteur dans le monde réel et de l'éloignement du joueur. Nous allons appliquer ce principe aux 2 extrémités horizontales du mur. Autrement dit, nous allons rechercher les coordonnées verticales du mur pour chacune de ses extrémités horizontales. Tout ceci nous donne donc le morceau de code suivant :

```
/* COEFFICIENTS INTERMEDIAIRES */
Rt = PlayerH - CeilingH ;
Rb = PlayerH - FloorH ;

/* ON CALCULE LES COORDONNEES VERTICALES DE DEBUT ET DE FIN */
/* DE CHACUNE DES EXTREMITES DU MUR PROJETE SUR L'ECRAN     */
STopStart = (long) (100L + (Rt * InvDistanceTable[(long) NewXStart])) ;
SBotStart = (long) (100L + (Rb * InvDistanceTable[(long) NewXStart])) ;
STopEnd = (long) (100L + (Rt * InvDistanceTable[(long) NewXEnd])) ;
SBotEnd = (long) (100L + (Rb * InvDistanceTable[(long) NewXEnd])) ;
```

Connaître les coordonnées verticales de ces 2 extrémités du mur va nous permettre de calculer incrémentalement (c'est à dire au fur et à mesure, par accumulation) les coordonnées verticales de chaque section verticale du mur, car pour dessiner le mur à l'écran, nous allons avoir besoin de ses coordonnées verticales en chaque colonne écran...

Nous allons en fait calculer de combien “varie” l'extrémité haute du mur lorsque l'on passe à la tranche verticale suivante. De même nous chercherons de combien “varie” l'extrémité basse lorsque nous passons également à la colonne (la tranche verticale écran) suivante. Ces variations peuvent bien évidemment augmenter ou réduire la taille apparente du mur en fonction de l'éloignement de la partie du mur que la colonne considérée représente.

```
DeltaTop = (STopEnd-STopStart) / (double) (SxEnd-SxStart) ;
DeltaBottom = (SBotEnd-SBotStart) / (double) (SxEnd-SxStart) ;
```

Remarque : Il est tout à fait possible que les coordonnées verticales que nous ayons calculées sortent de l'écran physique. Il nous faudra donc les clipper, mais nous verrons cela plus tard.
Voyons sur un schéma ce que le bout de code donnée juste au dessus effectue pratiquement :

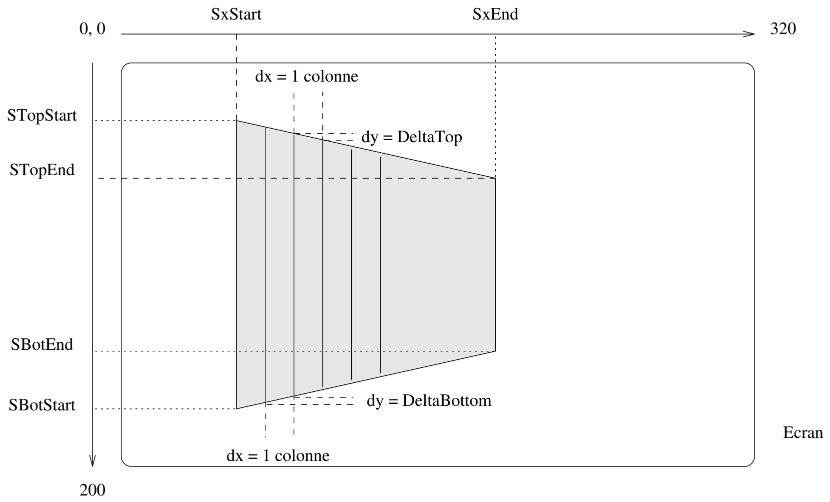

Figure 25: Le principe du calcul incrémental

Remarque : Cela signifie qu'à chaque fois que l'on progresse d'une colonne vers la droite dans le dessin du mur, il faut incrémenter la hauteur de l'extrémité haute du mur de $DeltaTop$ et incrémenter la hauteur de l'extrémité basse de ce même mur de $DeltaBottom$. Le jeu des signes donnera éventuellement des deltas négatifs, transformant nos incrémentations en soustractions.

Maintenant que nous connaissons les coordonnées verticales de l'extrémité gauche de notre mur (donc l'extrémité par laquelle on va commencer à tracer le mur, puisque nous dessinons de gauche à droite) et les deltas, nous pouvons en déduire pour chaque colonne-écran les nouvelles coordonnées-écran verticales. Rappelons qu'il faudra éventuellement aussi clipper ces coordonnées verticales car elles peuvent sortir de l'écran physique, "vers le haut" ou "vers le bas"...

7.6 Retour au clipping horizontal du mur

Nous avons dit précédemment qu'il fallait clipper horizontalement le mur, afin d'éviter que certains morceaux de ce dernier ne sortent des limites horizontales de l'écran. Ce clipping horizontal va donc restreindre, si nécessaire, l'extrémité gauche du mur à la colonne 0 et son extrémité droite à la colonne 319 (320-1 puisque les colonnes sont numérotées de 0 à 319). Nous allons donc chercher à mettre à jour les 2 variables $SxStart$ et $SxEnd$. Nous conserverons alors les valeurs des extrémités du mur une fois clippées dans 2 autres variables : $SxStartCl$ et $SxEndCl$ ("...Cl" pour "CLippées"), car nous aurons encore besoin plus tard des valeurs non clippées.

Si nous clippons les coordonnées horizontales, il est logiquement nécessaire de mettre à jour les coordonnées verticales des extrémités du mur ! Mais comme nous avons dit que nous conservions particulièrement les coordonnées verticales seulement du début du mur, les coordonnées verticales des tranches suivantes étant calculées incrémentalement grâce aux deltas que nous avons calculés ci-dessus, nous n'aurons à nous soucier que de mettre à jour les coordonnées verticales *du début du mur*. Ceci se passera donc uniquement dans le cas où l'on doit clipper effectivement l'extrémité gauche du mur. Regardons donc le morceau de source qui fait ce travail :

```
SxEndCl = SxEnd ;
if (SxEnd > 319)
    SxEndCl = 319 ;

SxStartCl = SxStart ;
if (SxStart < 0)
  {
    /* ON MET A JOUR LES EXTREMITES VERTICALES DU MUR COURANT, CAR LE  */
    /* MUR COURANT VA ETRE LE PREMIER REELLEMENT DANS LE CHAMP DE      */
    /* L'ECRAN ET NON CELUI TROUVE PAR LA PROJECTION THEORIQUE SUR LE  */
    /* PLAN VIRTUEL ET INFINI DE L'ECRAN.                              */
    Top -= (double) (SxStart * DeltaTop) ;
    Bottom -= (double) (SxStart * DeltaBottom) ;
    SxStartCl = 0 ;
  }
```

Maintenant, tout est prêt pour le dessin (et le clipping vertical si nécessaire)...

7.7 Organisation de la mémoire écran en mode chunky

Avant d'attaquer le dessin proprement dit des murs, nous allons faire un petit détour par l'examen de l'organisation de la mémoire vidéo de notre PC (Notons que d'autres machines proposent aussi un mode chunky, ceci n'est pas un privilège du PC). Ceci va nous permettre de comprendre comment afficher un point à l'écran et comment choisir sa couleur.

Pour notre moteur, nous nous intéressons au mode 320x200x256 couleurs. Dans ce mode, nous pouvons donc utiliser 256 couleurs en même temps, parmi 16 millions (16777216 très exactement, car chaque couleur est constituée d'une composante, pouvant prendre 256 valeurs, dans chacune des couleurs fondamentales - à savoir rouge, vert, bleu - ce qui fait bien 256 * 256 * 256 nuances possibles au total). La sélection des 256 couleurs choisies se fait par l'intermédiaire d'une palette de couleur, autrement dit une table d'association *(lookup table)* numérotée de 0 à 255 qui représente les couleurs disponibles pour l'image. Dans cette table, à chaque case numéro n, on stocke la valeur de chacune des composantes rouge-vert-bleu que comporte la couleur que l'on choisit de nommer n.

L'organisation de la mémoire vidéo est la suivante : un octet représente 1 point-écran. La valeur de cet octet représente sa couleur, via notre fameuse palette. Les octets sont agencés par ligne-écran : les 320 premiers octets de la mémoire vidéo représentent les 320 points du haut de l'écran, les 320 suivants octets représentent les 320 points de la ligne écran du dessous, etc... (c.f figure 26, page suivante)

Le problème restant est de trouver l'adresse dans la zone mémoire où mettre l'octet représentant la couleur voulue pour faire un point de coordonnées (x, y) à l'écran. Ceci est très simple et résolu par la formule suivante :

$$PointAdr = \text{TAILLE ECRAN HORIZONTALE} * y + x \tag{18}$$

ce qui donne dans notre cas :

$$PointAdr = 320 * y + x \tag{19}$$

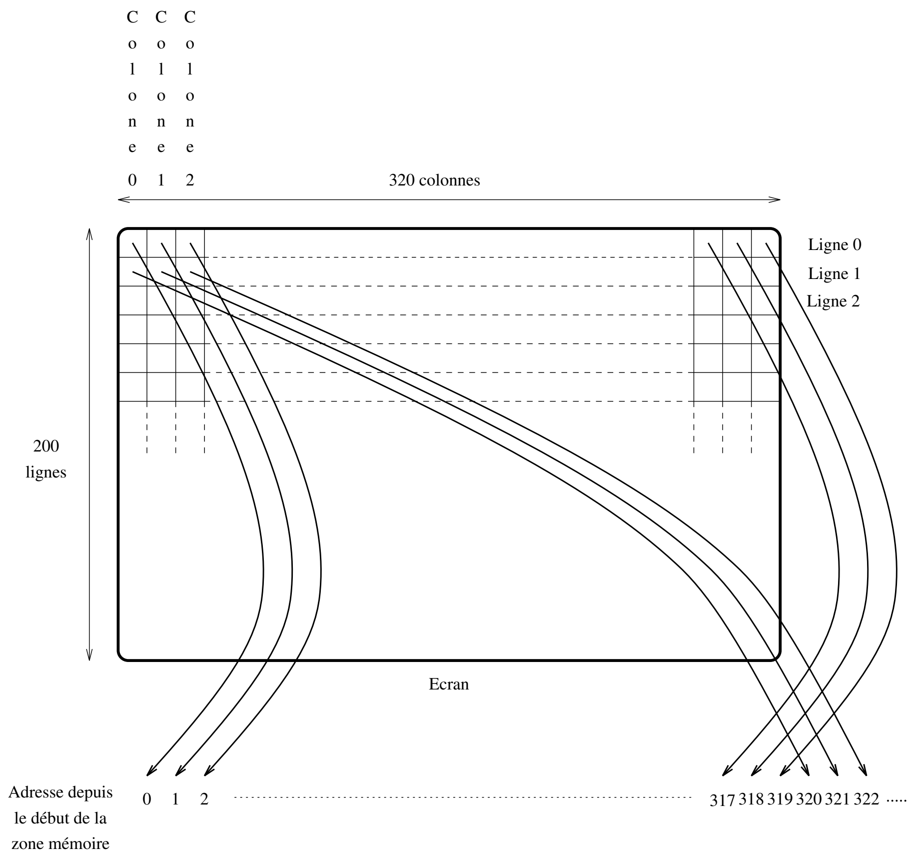

Figure 26: Organisation de la mémoire vidéo

Quelle est la taille de la zone mémoire nécessaire pour représenter un tel écran ? Nos 256 valeurs de couleurs sont codables sur 1 octet (8 bits) car $2^8 = 256$. Dans ce mode, la mémoire graphique a donc une taille de : 320 points * 200 point * 1 octet = 64000 octets.

Remarque : Si on se contente d'afficher des points en écrivant directement dans la mémoire graphique au fur et à mesure que nos calculs nous indiquent quels point allumer, il va résulter un effet très désagréable de scintillement dû à une mauvaise synchronisation entre notre envoi d'informations graphiques à afficher et le balayage du faisceau d'électrons qui forme l'image. Aussi, pour éviter cet effet, nous utilisons une zone mémoire temporaire, de même taille que notre mémoire vidéo : le *double-buffer*.

Nous écrivons tous nos pixels dedans, comme si c'était directement la mémoire vidéo. Une fois ce buffer complet, c'est à dire une fois que nous avons fini de rendre la vue, nous attendons, par une primitive spécifique, que le faisceau d'électrons atteigne la fin de l'écran courant, et nous transférons le contenu de notre double-buffer dans la véritable mémoire vidéo (par transfert rapide tant qu'à faire, 64 bits par 64 bits si possible, ou mieux encore par un processeur spécialisé si la carte graphique en possède un). A titre d'information, chez moi, pour faire simple, ce transfert est réalisé en 64 bits par le CPU. Ainsi, si tout se passe bien, le temps que le faisceau d'électrons recommence à "afficher" un nouvel écran (car celui-ci balaye le moniteur de haut en bas, donc il lui faut un certain temps pour "remonter" en haut de l'écran une fois qu'il est en bas), toute notre mémoire vidéo sera initialisée avec les informations graphiques représentant notre scène, et le scintillement disparaîtra.

7.8 Le dessin du mur sans texturage

Maintenant, nous avons toutes les informations nécessaires pour dessiner notre mur de façon unie, c'est à dire sans lui appliquer de texture. Nous verrons le texturage un peu plus tard. Pour le moment, faisons simple.

Récapitulons :

- Nous connaissons le colonne réelle de départ du mur.
- Nous connaissons le colonne réelle d'arrivée du mur.
- Nous connaissons les extrémités verticales de la colonne de mur la plus à gauche à l'écran.
- Nous connaissons les deltas pour réactualiser les coordonnées verticales de chaque colonne en fonction des coordonnées verticales de la colonne précedente (celle se trouvant immédiatement à gauche).

Une simple boucle `for` va donc nous suffir pour dessiner notre mur sans texture. Nous allons donc dessiner notre mur colonne verticale par colonne verticale. Le dessin d'une tranche de mur se révèle donc comme un simple tracé de lignes *verticales*, ce qui est extrêmement rapide.

A des fins d'accélération, plutôt que d'utiliser une fonction pour dessiner notre segment vertical, nous allons *inliner* (c'est à dire expanser pour les non-anglophones) le corps qu'aurait cette fonction directement dans notre boucle principale de dessin.

```
for (ColumnIndex=SxStartCl;ColumnIndex<=SxEndCl;ColumnIndex++)
  {
    /* ON NE DESSINE QUE SI CETTE COLONNE DE L'ECRAN N'EST */
    /* PAS DEJA OCCUPEE PAR UN MUR QUI ETAIT PLUS PROCHE.  */
    if (ColumnDone[ColumnIndex] != 1)
      {
        /* ROUTINE DE DESSIN DE LIGNE INLINEE */

        /* ON CLIPPE EN HAUT */
        RealTop = max (Top,0) ;
        /* ON CLIPPE EN BAS */
        RealBottom = min (Bottom,200) ;

        /* CALCUL DE L'ADRESSE DE DEPART */
        PointAdr = DblBuffer + (RealTop * 320 + ColumnIndex) ;
```

```
        /* ... ET DE L'ADRESSE DE FIN */
        PointAdrEnd = DblBuffer + (RealBottom * 320 + ColumnIndex) ;

       /* ET MAINTENANT, ON ECRIT DANS LE DOUBLE BUFFER */
       for (;PointAdr < PointAdrEnd; PointAdr += 320)
           (*PointAdr) = MY_COLOR ;

        /* MAINTENANT, LA COLONNE EST OCUPPEE PAR UN MUR. */
        ColumnDone[ColumnIndex] = 1 ;

        /* ON A FAIT UNE COLONNE DE PLUS */
        NbColumnsDone ++ ;
      }

    /* ON CALCULE INCREMENTALEMENT LES EXTREMITES VERTICALES */
    /* DE LA PROCHAINE TRANCHE DE MUR.                       */
    Top += DeltaTop ;
    Bottom += DeltaBottom ;
  }
```

Attardons-nous sur ce morceau de source. Nous faisons donc une boucle qui passe en revue toutes les colonnes du mur susceptibles d'être dessinées. Comme nos coordonnées verticales, elles, n'ont pas été encore clippées, on le fait ici ,ce qui nous donne 2 nouvelles variables $RealTop$ et $RealBottom$.

Ensuite, il ne nous reste plus qu'à tracer un segment vertical dans la colonne courante (repérée par l'indice de boucle $ColumnIndex$) partant de la ligne $RealTop$ et finissant à la ligne $RealBottom$. Pour y parvenir, plutôt que re-calculer pour chaque point vertical, l'adresse où il doit être écrit dans la mémoire vidéo, on calcule directement les adresses de début et de fin de la ligne, et pour obtenir une ligne verticale, il ne nous reste plus qu'à faire une seconde boucle `for` incrémentant le pointeur de la largeur de l'écran (320 ici).

Une fois que la ligne est tracée, il convient de dire que cette colonne-écran a été traitée, donc qu'il y a déjà un mur de dessiné. C'est ce que nous faisons en mettant $ColumnDone[ColumnIndex]$ à 1. Ce tableau contient donc autant de cases qu'il y a de colonnes à l'écran (320). Si $ColumnDone[i] = 1$ alors c'est que la colonne i est déjà occupée par une tranche verticale de mur. Donc il ne faudra pas réécrire dessus. C'est justement pour cette raison qu'avant de nous préparer à dessiner notre segment vertical, nous vérifions que la colonne-écran actuelle ne contient pas déjà un mur, déjà dessinée, donc plus proche.

Après avoir marqué notre colonne, nous incrémentons le nombre de colonnes contenant une tranche de mur (`NbColumnsDone ++ ;`), ainsi dès que nous aurons atteint 320, nous saurons que notre écran est complet (test réalisé dans le parcours du BSP, en dehors de notre fonction de dessin que nous sommes en train de décrire).

Notons que la constante `MY_COLOR` représente la valeur de couleur que vous désirez pour teinter vos murs. Comme nous sommes en mode 256 couleurs, cette constante peut prendre des valeurs comprises entre 0 et 255 inclus.

Nous remarquerons aussi le nom donné au pointeur de notre "mémoire vidéo" : `DblBuffer`. Il ne s'agit pas en réalité de l'adresse de la mémoire vidéo, mais comme expliqué précédemment, d'une zone de mémoire tampon : notre double-buffer.

7.9 Epilogue du rendu de scène

Notre routine d'affichage d'un mur va partir du principe que l'écran sur lequel elle travaille a été effacé entre 2 images consécutives de l'animation. De plus, une fois que l'on sait que la vue est complètement rendue, il faudra transférer le contenu du double-buffer dans la mémoire vidéo réelle. De même, entre chaque image, il faudra que le tableau $ColumnDone[]$ soit ré-initialisé à "plein de 0" et que la variable $NbColumnsDone$ soit aussi remise à 0.

Aussi, pouvons-nous imaginer que le lancement principal du rendu d'une scène aura la forme :

```
void RenderView ()
{
long i ;
/* EFFACEMENT DU DOUBLE-BUFFER */
for (i=0;i<8000;i++)
  ((double*)DblBuffer)[i] = (double) 0 ;

/* ON NOTIFIE QU'ON N'A ENCORE DESSINE AUCUNE COLONNE. */
NbColumnsDone = 0;
for (i=0;i<320;i++)
  ColumnDone[i] = (unsigned char) 0 ;

/* ET ON PARCOURT LE BSP AFIN DE DESSINER LA VUE. */
FrontToBackTreeParsing (LevelBspTree) ;

/* ATTENTE DU RETOUR DU FAISCEAU POUR EVITER LE SCINTILLEMENT. */
waitretrace () ;

/* ON BALANCE LE DOUBLE-BUFFER DANS LA MEMOIRE-ECRAN VIDEO REELLE. */
for (i=0;i<8000;i++)
  ((double*)graph_mem)[i] = ((double*)DblBuffer)[i] ;
}
```

Remarque 1 : La fonction nommée `FrontToBackTreeParsing` se charge de parcourir le BSP Tree représentant notre monde et communique, au fur et à mesure, les murs à la routine de dessin que nous avons décrite jusqu'à maintenant.

Remarque 2 : La fonction nommée `waitretrace` sert à attendre que le faisceau d'électrons atteigne le bas de l'écran, de façon à synchroniser le rafraîchissement de l'écran et le rafraîchissement de nos informations graphiques dans la mémoire vidéo.

Remarque 3: Notez ici les transferts en 64 bits par l'intermédiaire du type *double* en C. Etant donné qu'un double fait 64 bits, donc 8 octets, et que notre espace de mémoire vidéo fait 64000 octets comme nous l'avons vu précédemment, il est normal que notre boucle ne fasse que 8000 tours (64000 / 8 = 8000).

Cette méthode possède un petit inconvénient : plus le joueur est loin des murs, plus ceux-ci sont petits, donc plus ceux-ci sont dessinés rapidement. Il en résulte que lorsque l'on se rapproche, les murs devenant plus grands, il y a plus de points à dessiner et donc l'affichage se ralentit un peu. Pour réduire un peu cet effet (et en fait gagner du temps ; c'est ça qui est bien, on y gagne sur 2 aspects !), au lieu d'effacer l'écran et de lancer ensuite le rendu des murs, on peut faire en sorte que le dessin de chaque tranche de mur trace toujours une colonne de 200 points de haut, avec des points noirs pour les pixels ne représentant pas le mur (ceux qui représenteraient en fait le plafond ou le plancher). Ainsi, le nombre de points inscrits dans la mémoire est toujours le même et les calculs sont *sensiblement* (pas exactement c'est sûr) constants.

Avec une telle méthode, nous gagnons en fait du temps, car effacer l'écran revient à n'y inscrire que des octets de la couleur du fond. Or, dans beaucoup de cas, ces octets vont être réécrits par le dessin d'un mur. Donc l'effacement préalable consomme beaucoup de travail pour pas grand-chose...

Attention : Nous avons ici vu la fonction de dessin d'un mur morceau par morceau. Je récapitulerai cette fonction dans sa totalité, en donnant d'un coup tout son source, une fois que nous aurons vu le problème du mapping de textures sur les murs.

8 Mapper des textures

Dans le principe du rendu, le fait de mapper des textures sur les murs ne va pas remettre en cause tous les calculs effectués jusqu'à maintenant. Il va seulement falloir faire quelques nouveaux calculs afin de trouver comment dessiner la texture en fonction de la position du joueur par rapport au mur. Le surcroît de calculs induit par le texturage n'est pas phénoménal. Néanmoins, il est sûr que ça va ralentir un peu le moteur par rapport à la vitesse que nous pouvons obtenir en dessinant des murs unis.

8.1 Idée du mapping

L'idée du mapping est donc de plaquer une texture (un aspect) sur une surface pour lui donner un rendu plus réaliste qu'une simple couleur unie. En l'occurrence, nous allons mapper des textures seulement sur les murs.

Tout comme dans la réalité, plus le joueur sera près du mur texturé, plus le mur lui paraîtra grand et donc sa texture grosse. Plus le joueur sera loin, plus le mur lui paraîtra petit et donc cette même texture lui paraîtra d'autant plus rétrécie.

Notre mapping est donc une question de mise à l'échelle d'un dessin original. Je dis "notre" mapping, car je parle de la technique que nous allons utiliser. En effet, il existe des techniques de mapping *beaucoup* plus évoluées, donnant de bien meilleurs rendus, mais bien plus gourmandes en calculs et nettement plus branle-crâne à piger ⌣̈.

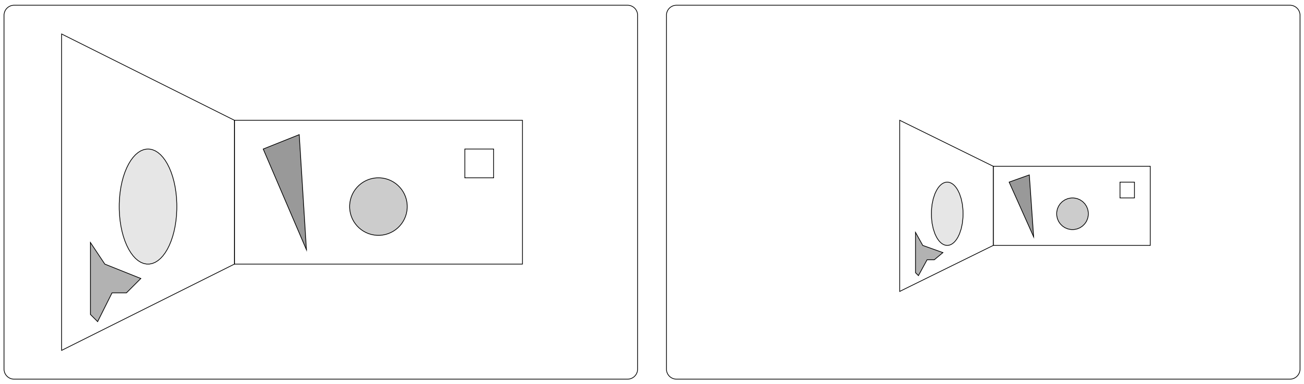

Figure 27: Mapping de texture, taille apparente et impression d'éloignement

8.2 Qu'est-ce que l'on a à faire ?

Nous devons travailler dans 2 directions :

- Le mise à l'échelle verticale
- La mise à l'échelle horizontale

Nous allons faire une petite hypothèse simplificatrice qu'il sera possible de lever une fois le principe compris : le motif de la texture est dessiné *1 seule* fois verticalement sur le mur. Autrement dit, si nous avons un mur de hauteur (dans notre monde, pas à l'écran) 128 et que notre texture de départ fait 64 points de haut, nous ne verrons pas la texture affichée 2 fois verticalement. Nous ne la verrons qu'une seule fois. Il me semble que *Doom* ne procède pas comme ça : si le mur fait 128 de haut, alors le motif de la texture sera dessiné 2 fois verticalement.

Le mapping se fait bien évidemment au moment où l'on dessine le mur, c'est à dire que nous allons avoir principalement à modifier la boucle de dessin. Chaque point du mur dessiné est associé à un point dans la bitmap [4] originale ce qui permettra de trouver de quelle couleur afficher le point. Il convient donc de trouver pour chaque point à afficher à l'écran, quel est le point correspondant dans la bitmap de texture, de prendre la même couleur et d'afficher notre point.

Pour mapper, nous avons besoin de 2 choses :

- De savoir à quelle distance du début du mur, le rayon de vue qui aboutit à la tranche que l'on dessine actuellement, frappe ce mur. Ceci nous donnera l'index vertical dans la bitmap de texture.
- De savoir, lorsque l'on dessine les pixels d'une tranche particulière, le pixel à aller chercher verticalement (on connaît déjà l'index horizontal, ceci nous donne l'index horizontal dans la bitmap).

[4] Nous appellerons bitmap le dessin à appliquer sur les murs

8.3 Mise en œuvre

Cette section expose les calculs à effectuer pour obtenir ce que nous voulons.

8.3.1 Détermination de l'index horizontal dans la bitmap

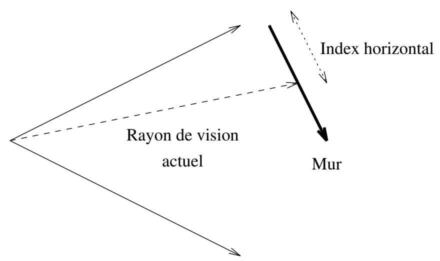

Figure 28: Index horizontal dans la texture et rayon de vision

Attention : Je tire l'idée de ce principe d'un autre papier disponible sur Internet : *"One Approach to Real Texture Mapping"* par William D. Doughty, Novembre 1994. Je me permets de re-transcrire ces informations dans mon propre document, d'une part pour faire en sorte que mon papier décrive le plus exhaustivement possible les mécanismes amenant à faire un moteur 3D, et d'autre part pour la bonne raison que j'ai implanté ce mécanisme un peu différemment (avec des divergences par rapport à ses calculs : erreur de sa part ou petites différences dues à mon implantation ?). Je ne donne pas ici l'explication théorique de la méthode. Pour plus d'informations, se reporter au papier d'origine.

Nous avons besoin des données initiales suivantes :

- **A** : l'angle du mur dans le repère après rotation-translation. Notons que la translation n'a aucune incidence sur l'angle du mur dans le nouveau repère (je parle de "nouveau repère" car le fait de faire subir aux coordonnées de notre mur notre fameuse rotation-translation, revient à faire un changement de repère...).
- **PDist** : la distance du joueur au mur. C'est donc la longueur de la projection orthogonale du point que représente le joueur, sur la ligne que supporte le mur.
- **BtoBP** : la distance entre la base du mur et le point où se projette le joueur sur le mur (point nommé BP).

Celles-ci sont illustrées par le schéma de la page suivante, (figure 29) en provenance de l'article cité ci-dessus :

Ces données doivent bien évidemment être calculées avant d'entrer dans la boucle d'affichage des points. Voici comment calculer ces données :

```
/* ANGLE DU MUR */
A = Linedef->angle + TO_RAD (-PlayerAngle) ;
l = Linedef->len ;
r = ((NewYStart-0)*(NewYStart-NewYEnd)-(NewXStart-0)*(NewXEnd-NewXStart))
  / (l*l) ;
```

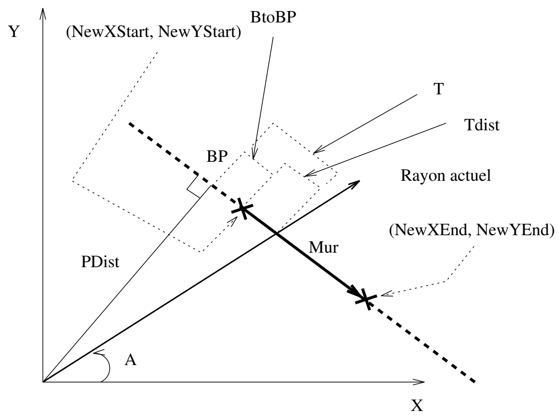

Figure 29: Schématisation des données requises

```
s = ((NewYStart-0)*(NewXEnd-NewXStart)-(NewXStart-0)*(NewYEnd-NewYStart))
  / (l*l) ;
/* DISTANCE DE L'OEIL AU MUR PAR PROJECTION ORTHOGONALE */
PDist = s * l ;
/* DISTANCE ENTRE PROJECTION ORTHOGONALE DE L'OEIL */
/* SUR LE MUR ET LE DEBUT DU MUR. */
BtoBP = r * l ;
```

Remarque 1 : Pour trouver l'angle du mur dans notre nouveau repère, il suffit de prendre l'angle de ce mur (angle qu'il est possible de pré-calculer lors de la création du niveau et de ranger directement dans la structure de linedef comme c'est le cas ici) et de lui soustraire l'angle qui définit la direction originale du joueur (ou de lui ajouter "-cet angle", puisque la rotation que nous avons appliquée au repère est de $-PlayerAngle$). Rappelons que la translation faisant partie de la transformation que nous avons appliquée initialement n'a aucun effet sur l'angle du mur.

Remarque 2 : Nous travaillons avec les coordonnées du mur *après* transformation.

Remarque 3 : La longueur du mur peut elle aussi être pré-calculée et rangée dans la structure de la linedef. Ceci nous évite ainsi quelques calculs.

Remarque 4 : Ces calculs doivent être faits le plus tard possible afin d'éviter de les faire pour rien. En effet, si on les faisait dès le début de la fonction de rendu du mur, pour peu que ce mur ne soit pas visible, alors nous aurions fait du travail pour rien. Pour ma part, ces calculs s'insèrent juste après avoir vérifié que le mur n'était pas totalement derrière nous. Il faut par contre, que ces calculs soient effectués avant le clipping des extrémités du mur chargé de ne pas laisser des morceaux de mur derrière nous ! En effet si on faisait le contraire, alors on considèrerait que le début du mur serait le début après clipping. Donc dans le cas ou ce clipping aurait effectivement réduit la taille du mur en tronquant son extrémité de départ, alors notre index ne serait calculé pas calculé depuis le début réel du mur dans le monde. Il en résulterait un décalage de texture puisque nous considèrerions que le début de la texture correspond toujours avec le début de la partie visible du mur...

Remarque 5 : Nos calculs sont basés sur la méthode permettant de calculer la distance d'un point à une droite décrite dans les *FAQ comp.graphics.algorithms* disponibles sur Internet. A titre d'exhaustivité, je récapitule cette méthode.

Soit un point C de coordonnées (x_C, y_C) et un segment d'extrémités A (x_A, y_A) et B (x_B, y_B) que l'on considère orienté de A vers B.
La longueur L du segment $[AB]$ est donnée par la formule :

$$L = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2} \tag{20}$$

Soient les 2 coefficients intermédiaires suivants :

$$r = \frac{(y_A - y_C) * (y_A - y_B) - (x_A - x_C) * (x_B - x_A)}{L^2} \tag{21}$$

$$s = \frac{(y_A - y_C) * (x_B - x_A) - (x_A - x_C) * (y_B - y_A)}{L^2} \tag{22}$$

Appelons I le point de projection orthogonale de C sur le segment $[AB]$. Ses coordonnées (x_I, y_I) sont données par :

- $x_I = x_A + r * (x_B - x_A)$
- $y_I = y_A + r * (y_B - y_A)$

La distance de A à I est égale à $r * L$, celle de C à I est égale à $s * L$. Quelques petites annotations peuvent être utiles :

- Si $r < 0$ alors I est sur l'extension "arrière" de $[AB]$.
- Si $r > 1$ alors I est sur l'extension "avant" de $[AB]$.
- Si $0 \leq r \leq 1$ alors I est sur le segment $[AB]$.
- Si $s < 0$ alors C est à *gauche* du segment $[AB]$.
- Si $s > 0$ alors C est à *droite* du segment $[AB]$.
- Si $s = 0$ alors C est *sur* le segment $[AB]$.

Après avoir vu comment calculer les pré-requis, nous arrivons enfin au calcul d'index horizontal proprement dit, c'est à dire celui qui va être effectué à chaque fois que l'on dessine une tranche verticale de mur afin d'être capable de savoir à quel morceau du mur cette tranche se rapporte, et donc quel est le motif visible à cet endroit.

Tout d'abord, considérons que notre texture a une taille horizontale de T pixels. Donc, tous les T de distance, le dessin de la texture va se répéter sur le mur. Donc, nous devons obtenir un index vertical *modulo* la taille T de la bitmap à mapper. En fait, nous allons chercher à savoir à quelle distance du début du mur, le rayon actuel de vision (déterminé par la colonne écran que nous sommes en train de tracer) frappe-t'il le mur, et nous ramènerons cette distance à la forme d'un index en effectuant un simple modulo par la taille horizontale T de notre bitmap de texture. Le source qui fait ce travail est le suivant :

```
TanIndex = A + TO_RAD(90) - AngleTable[ColumnIndex] ;
HIndex = ((PDist * tan(TanIndex)) + BtoBP) % BITMAP_WIDTH ;
/* ON COMBAT LES ERREURS D'ARRONDIS */
if (HIndex < 0) HIndex = 0 ;
```

Que fait-il ?

1. Tout d'abord, il faut trouver l'angle correspondant au rayon de vue que l'on est en train de traiter, c'est à dire l'angle du rayon qui permet de "voir" la tranche que nous sommes en train de construire. Nous savons mathématiquement qu'il n'y a pas de relation directe entre un angle et la taille de sa projection sur une droite (cf. figure n^{o}30 ci-après). Ceci signifie que nous ne pouvons pas déduire l'angle correspondant à un rayon incrémentalement. C'est à dire que l'on ne peut pas prendre l'angle du rayon le plus à gauche, se dire qu'il correspond à la colonne c, et calculer l'angle correspondant à la colonne $c+1$ en ajoutant un terme α *constant* à l'angle trouvé pour la colonne c. Nous devons obligatoirement faire le calcul à base de cosinus. Alors, plutôt que de s'envoyer ce calcul à chaque colonne, on va déterminer à l'initialisation du programme les angles correspondant aux différentes colonnes de l'écran (donc ces angles s'inscrivent dans le cône de vision s'étendant chez nous sur 90^o). Comme nous avons 320 colonnes, il nous faudra une table de 320 entrées. C'est cette table qui est appelée $AngleTable$. Nous voyons que nous l'indexons par des numéros de colonne. Nous verrons en remarque comment est effectué le calcul d'angle, donc comment est initialisée cette table.

2. Ensuite, nous rajoutons 90^o à l'angle du mur (A)

3. Nous retirons de tout ça la valeur de l'angle associé au rayon actuel. Ceci nous donne un nouvel angle.

4. Maintenant, nous multiplions $PDist$ par la tangente de l'angle obtenu ci-dessus.

5. Et nous rajoutons $BtoBP$ à tout ça.

6. Le calcul effectué jusqu'à présent nous donne donc la distance entre le point d'impact du rayon sur le mur et le début du mur, nous il ne nous reste plus qu'à effectuer notre modulo pour transformer notre distance en index dans la bitmap.

7. Pour terminer, nous corrigeons un éventuel décalage dû à des approximations de calcul en s'assurant que notre index n'est pas négatif.

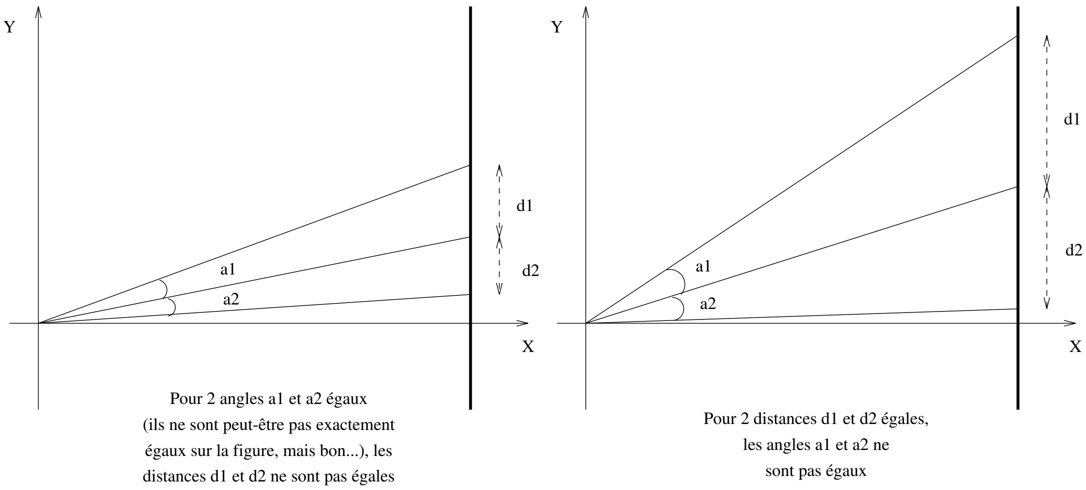

Figure 30: Non linéarité entre un angle et sa projection

Remarque 1 : Des erreurs d'arrondis peuvent se glisser dans les calculs. Ces erreurs, aussi petites soient-elles, risquent de nous faire obtenir à un moment ou à un autre (particulièrement aux extrémités des murs) des défauts de texture dus à un index négatif (très peu, c'est sûr, mais ça suffit pour aller se balader n'importe où en mémoire). Afin d'éviter ceci, nous regardons si notre calcul nous a donné un résultat inférieur à 0 et si c'est la cas, on considère que c'est 0. En fait, quand on récupère un résultat inférieur à 0, ça se joue à 10^{-2}, 10^{-3} voire encore moins (c'est à dire qu'au lieu d'avoir 0, on a -0.01 ou -0.001), donc on voit que ça n'est pas énorme.

Remarque 2 : Au lieu de faire un *modulo*, ce qui implique une division, il est possible de convertir la distance depuis le début du mur en nombre entier et de faire un masque dessus. Par exemple, pour une taille de 64 pixels, on devrait faire un masque *ET LOGIQUE* avec la valeur hexadécimale 0x3F. En effet, ceci revient à ne conserver que les 6 derniers bits du nombre, or 2^6 étant égal à 64, ceci nous donnera bien un nombre compris entre 0 et 64 : le modulo (qui est le reste de la division entière, mathématiquement parlant).

Remarque 3 : Ce calcul d'index horizontal doit être réalisé pour chaque tranche du mur que l'on calcule. Aussi, ce calcul se situera-t'il vers le début de la boucle qui balaye toutes les colonnes-écran dans lesquelles le mur s'inscrit (ou plutôt “est susceptible de s'inscrire” s'il n'y a pas déjà eu de mur plus proche dessiné dans chacune de ces colonnes). Néanmoins, par soucis d'économie de temps CPU, il ne faut réaliser ce calcul qu'après s'être assuré que nous pouvons dessiner la tranche à l'écran. En effet, si on fait le calcul et qu'on s'aperçoit au dernier moment que la colonne-écran que l'on traite est déjà occupée par un mur plus proche, donc dessiné précédemment, on aura fait ce calcul pour rien et perdu du temps.

Remarque 4 : Voyons comment est pré-calculée notre table d'angles. Donc, dans cette table, pour chaque colonne de l'écran, nous allons chercher quel est l'angle de vue qui lui correspond.

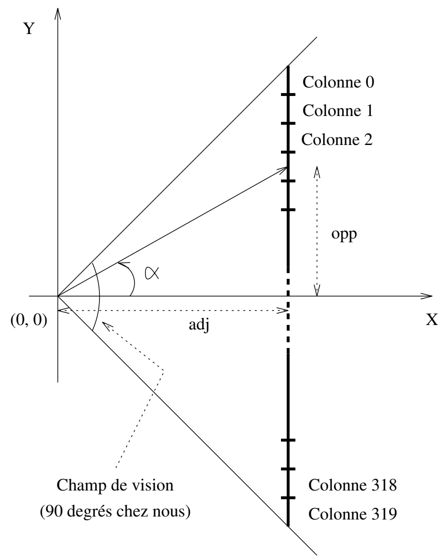

Figure 31: Angle à partir de la colonne

D'après cette figure, nous savons que :

$$\tan(\alpha) = \frac{opp}{adj} \tag{23}$$

donc il nous suffit pour trouver l'angle, de prendre l'arc tangente (tan^{-1} encore noté arctan) de ce calcul :

$$\alpha = \tan^{-1}(\frac{opp}{adj}) \tag{24}$$

Souvenons-nous que pour *adj* (c'est en fait la distance focale), nous avions pris une valeur de 160. Maintenant, il ne nous reste plus qu'à faire varier la colonne pour remplir le tableau. Donc nous ferons une boucle de 0 à 319 inclus qui calcule nos angles. Néanmoins comme l'angle 0 correspond à la colonne-écran centrale, il convient de re-centrer notre calcul. Pour se faire, nous devons transformer notre formule, en rajoutant une simple soustraction, en :

$$\alpha = \tan^{-1}(\frac{160 - opp}{adj}) \tag{25}$$

Tout ceci nous donne le code suivant :

```
AngleTable = (double*) malloc (320*sizeof(double)) ;
if (AngleTable == NULL)
  return (ANGLETABLEALLOC_FAILED) ;

for (i=0;i<320;i++)
  AngleTable[i] = atan ((160.0 - (double)i) / 160.0) ;
```

8.3.2 Détermination de l'index vertical dans la bitmap

Le calcul d'index vertical va pouvoir être réalisé entièrement incrémentalement. C'est à dire que nous allons calculer complètement la première valeur de cet index et un terme constant, et les autres valeurs seront obtenues à partir de la précédente en rajoutant ce terme constant à chaque fois.

Puisque nous cherchons à inclure verticalement la texture totalement, il suffit de faire le rapport entre la taille verticale de la bitmap et la hauteur du mur (non clippée). Ainsi, quand nous nous déplaçons d'un pixel verticalement, nous savons de combien on se déplace dans la bitmap de texture.

```
VIncr = BITMAP_HEIGHT / (Bottom - Top) ;
```

Cette valeur va être le terme constant que nous rajouterons à chaque itération. La valeur initiale de l'index vertical n'est pas forcément 0 (si l'on considère 0 comme le haut de la bitmap et $BITMAP\_HEIGHT - 1$ comme son bas). En effet, il se peut que le mur soit trop grand à l'écran et que nous l'ayons clippé. Dans ce cas, il convient de trouver de combien ce clipping nous à fait déplacer le début vertical visible à l'écran de la bitmap par rapport au début normal de cette même bitmap. Le calcul est simple ($RealTop$ et Top ont la signification que nous avons vu précédemment, à savoir que Top est la coordonnée verticale écran théorique de l'extrémité haute du mur, et $RealTop$ est Top clippé à la borne supérieure de l'écran, donc à 0).

```
VIndex = (RealTop-Top) * VIncr ;
```

Notons que l'on peut s'assurer que $VIndex$ n'est pas inférieur à 0 par un arrondi, comme nous l'avons fait précédemment pour $HIndex$. En effet, des erreurs d'arrondis peuvent encore parfois survenir entraînant une légère divergence dans les calculs (au lieu d'avoir un résultat vraiment égal à 0, il peut être égal à -0.0001 ou -0.00...1) ce qui risquerait encore de nous faire accéder en dehors de la zone mémoire de la bitmap à mapper.

```
if (VIndex < 0) VIndex = 0 ;
```

Maintenant, il ne nous reste plus, lors de la boucle de dessin des points formant la tranche (c'est à dire la boucle la plus profonde), qu'à rajouter $VIncr$ à chaque tour pour mettre à jour la valeur de $VIndex$, l'index vertical. Ainsi, pour chaque nouveau point dessiné de la tranche, on calcule le minimum pour déterminer l'index vertical qui lui correspond.

8.4 Terminons le mapping

Maintenant que nous savons calculer, pour chaque tranche son index horizontal, et pour chaque pixel de la tranche son index vertical, il est simple de retrouver le point correspondant dans la bitmap. C'est un simple calcul d'adressage dans une zone mémoire "rectangulaire" en fonction de coordonnées x et y. Sachant que notre bitmap est rectangulaire et de taille : $BITMAP\_WIDTH \times BITMAP\_HEIGHT$, notre indice dans la bitmap est donné par le calcul suivant :

$$Indice = (y * BITMAP\_WIDTH) + x \tag{26}$$

Si l'on considère que notre bitmap est rangée dans un tableau $MyBitmap[]$, alors on accède au point par : $MyBitmap[Indice]$.

Tout ceci fait qu'au lieu d'afficher un point toujours de la même couleur, comme nous l'avions fait dans les sections précédentes, nous afficherons chaque point de la couleur lui correspondant dans la texture. Comme nous l'avons vu dans le chapitre sur l'organisation de la mémoire graphique, ceci reviendra à copier l'octet considéré de la bitmap dans la mémoire graphique pour afficher le point.

Pour ce qui est de la bitmap, celle-ci peut être chargée d'un fichier à l'initialisation du programme et rangée dans un tableau. Actuellement, nous n'avons raisonné qu'avec 1 seule bitmap, il est sûr que nous pouvons aisément généraliser, en rajoutant dans la structure de linedef, un champ identifiant la texture se trouvant sur ce mur. Ce n'est qu'une modification mineure de ce que nous avons fait, et n'influence pas sur le principe du moteur. Au lieu de toujours référencer notre tableau $MyBitmap[]$ avec les index calculés, nous devrons alors référencer la tableau où se trouve la bitmap servant de texture au mur que l'on traite. C'est tout...

8.5 Conclusion sur le mapping

Nous avons vu que la mapping impose quelques petits rajouts de calculs. Ils ne sont pas énormes. Néanmoins, ceux se trouvant dans les boucles de dessin prennent quand même un peu de temps. Aussi, dans le cadre d'une optimisation, il conviendrait de soigner ces calculs de façon à les rendre les plus rapides possibles.

Pour consulter les modifications que cela a apporté au source, consultez le listing de la fonction de rendu d'un mur. Cette annexe permettra d'avoir une vue d'ensemble de la fonction et en même temps, d'illustrer exactement où insérer les calculs que nous avons faits.

9 Entr'acte

Nous venons de voir comment fonctionnait le moteur. Néanmoins, un moteur 3D se doit de pouvoir inclure des objets en plus des murs. Par exemple, on aimerait bien avoir des tables, des chaises, des flingues, et surtout des trucs à tuer ($\ddot\smile$), genre monstres, *bad guys*, etc...

Dans un tel moteur, basé sur les BSP, les objets statiques (ne se déplaçant pas) peuvent sans doute être inclus dans le BSP en cherchant un peu. Nous allons, par contre, nous intéresser à comment intégrer des *objets dynamiques*, c'est à dire des objets susceptibles de bouger dans le monde. Clairement ces objets ne peuvent pas faire partie du BSP, puisqu'il bougent, donc n'occupent pas toujours les mêmes places par rapport aux lignes de partitionnement du BSP. Clairement, il faut trouver autre chose.

Remarque : Le procédé que nous allons déterminer pour traiter et dessiner les objets dynamiques pourra très bien servir à afficher aussi des objets statiques.

10 Comment gérer graphiquement les objets dynamiques (ou sprites)

Deux problèmes se posent à nous :

- Comment trouver si et où ces objets s'affichent dans le monde.
- Comment les texturer ?

Nous allons en fait nous retrouver devant un problème similaire à celui que nous venons de traiter. Seuls quelques trucs mineurs vont changer, mais le principe est sensiblement le même sur bien des aspects.

10.1 Principe de l'affichage

Une voie possible de recherche est d'insérer les objets dans le BSP, en les considérant réduits à 1 point. Ainsi il n'y a pas de risque que l'objet puisse être coupé par un mur (c'est en ça que les sprites ne sont pas de réels objets en 3 dimensions). Ensuite, les objets faisant partie de l'arbre, sont affichés en même temps que les murs. Qui plus est, on les récupère directement triés dans le bon ordre de profondeur. Je n'ai pas décidé d'utiliser sur ce principe pour la simple raison que l'insertion d'objets dans le BSP me semble quand même coûteuse (et pas très claire au point de vue du principe), et surtout parce qu'il ne faut pas oublier de supprimer tous ces objets du BSP à chaque fin de rendu d'image. Or ceci implique pas mal de calculs, et je pense que sur un gros niveau, cela risque de devenir très lourd. Qui plus est, insérer les objets dans le BSP implique que l'on doit effectuer des calculs même pour les objets qui s'avéreront non visibles après coup dans la scène. On risque ainsi de faire pas mal de calcul pour rien.

Qui plus est, les objets n'ayant pas la même taille que les murs, il est possible de voir un mur derrière un objet. Aussi nous retrouvons-nous directement confrontés au problème des murs de hauteur différente, problème que j'ai décidé de ne pas aborder dans ce papier.

Aussi la méthode que j'ai choisie est la suivante : on affiche déjà les murs du décors et ensuite, en sur-impression, on affiche les objets dynamiques. Ceci implique que l'on réécrive éventuellement dans des colonnes où un mur a déjà été dessiné (si l'objet est en réalité plus proche que le mur). Aussi devrons-nous gérer nous-mêmes l'ordre d'affichage des objets entre eux, pour qu'un objet se situant derrière un autre soit affiché en premier, afin d'être en partie recouvert par celui qui est plus proche. Nous devons aussi gérer nous même le fait de chercher à savoir si un mur est plus près qu'un objet dynamique, de façon à ne pas afficher les objets qui, bien que se trouvant normalement dans notre champ de vision, sont invisibles car masqués par un mur plus proche. C'est ceci qui permettra le bon rendu en sur-impression.

La gestion des objets va donc être séparée en 2 parties :

- La préparation des objets, qui va déterminer quels objets se situent dans le champ de vision du joueur.
- L'affichage proprement dit de ces objets et la détermination finale de visibilité, prenant en compte la profondeur des objets entre eux et entre les murs.

Autrement dit, tous les objets vont être mis en place pour être affichés, rangés dans la *liste des objets à afficher* et ensuite, la routine d'affichage proprement dite va parcourir cette liste et afficher à l'écran tous ces objets.

10.2 La préparation des objets

Cette fonction va ressembler étrangement au début de notre fonction de gestion des murs. Nous allons tout d'abord considérer les objets comme ponctuels, c'est à dire comme ayant la largeur d'un point. Un objet est alors repéré par ses coordonnées dans la carte.

Structure d'un objet Nous donnons ici la structure d'un objet afin de pouvoir y faire réference ultérieurement. Les champs qui composent cette structure seront détaillés au moment où nous nous en servirons. Toutefois, certans d'entre eux ont une utilité assez évidente :

```
struct Object {
  double x ;              /* ABSCISSE DANS LE MONDE */
  double y ;              /* ORDONNEE */
  double heighttop ;      /* EXTREMITE HAUTE DE L'OBJET */
  double heightbottom ;   /* EXTREMITE BASSE DE L'OBJET */
  double globalheight ;   /* heighttop - heightbottom : PRE-CALCULE */
  double width ;          /* 1/2 LARGEUR DE L'OBJET */
  /* TOUT CE QUI EST EN DESSOUS EST MIS A JOUR LORS DU RENDU DES OBJETS */
  long stop, sbottom ;       /* START TOP ET BOTTOM CLIPPES DE L'OBJET */
  long sxstart, sxend ;      /* START ET END SCREEN X CLIPPES DE L'OBJET */
  long ncstop, ncsxstart ;   /* START TOP ET SCREEN X START NON CLIPPES */
  double vincr, hincr ;   /* INCREMENTS VERT/HORIZ POUR LE MAPPING */
  double prof ;           /* PROFONDEUR DE L'OBJET */
};
```

Notre fonction va répéter le traitement que nous allons décrire ci-après pour chaque objet. Les objets sont eux aussi rangés linéairement dans un tableau :

```
  for (i=0;i<NbObjects;i++)
    {
      CurrentObject = &GlobalObjectTable[i] ;
      ... TRAITEMENT ...
    }
```

Nous allons voir ce traitement sur un objet, la boucle se chargeant de le dupliquer pour chacun des objets contenus dans notre monde. Nous appliquons à cet objet la même transformation que pour les murs, à savoir nous re-centrons le repère de la carte sur le joueur en calculant les nouvelles coordonnées de l'objet :

```
      NewX = (CurrentObject->x-PlayerX) * cos (TO_RAD(-PlayerAngle))
                                  -
  (CurrentObject->y-PlayerY) * sin (TO_RAD(-PlayerAngle)) ;

      NewY = (CurrentObject->y-PlayerY) * cos (TO_RAD (-PlayerAngle))
                                  +
  (CurrentObject->x-PlayerX) * sin (TO_RAD (-PlayerAngle)) ;
```

Ensuite, nous nous assurons que l'objet n'est pas totalement derrière nous et nous clippons sa profondeur si nécessaire :

```
if (NewX < ZMIN)
  continue ;

/* ON CLIPPE SA PROFONDEUR AU CAS OU ELLE SERAIT EXAGEREE */
if (NewX > 9999)
  NewX = 9999 ;
```

Maintenant, nous devons prendre en compte la largeur de l'objet. Remarquons que notre détermination de visibilité se base sur un objet considéré comme un point (ce point étant le centre de l'objet), alors que cet objet possède en réalité une largeur. Ceci implique que si le centre d'un objet se trouve derrière un mur mais très près de celui-ci, alors l'objet est invisible, même si sa largeur devrait faire qu'une partie de l'objet sorte du mur. Il faur dire que ça n'est pas gênant car on a rarement vu des objets réels traversant les murs...

Pour prendre cette largeur en compte, nous allons calculer la projection du centre de l'objet à l'écran, ainsi que la projection des extrémités horizontales de cet objet, tout comme nous avions calculé la projection des extrémités d'un mur à l'écran.

Remarque : Dans la portion de source qui suit, le champ `width` de la structure représentant un objet dynamique représente en fait **la moitié de sa largeur** et non sa largeur totale... Je sais que ça n'est pas très logique, mais c'est un vieux reste de mon source original dans lequel je n'avais, pour ainsi dire, pas choisit un nom vraiment correct. Mais bon, c'est comme ça... Une fois qu'on le sait. Et puis de toutes façons j'ai la flemme de changer mon source pour ça, je préfère mettre un petit commentaire à la place $\ddot\smile$. De toute façon, plus loin nous devrons utiliser la largeur totale de l'objet, et là nous ferons une multiplication par 2 pour l'avoir, donc il n'est pas plus intéressant de stocker la largeur ou la demi-largeur de l'objet...

```
/* EXTREMITE GAUCHE DE L'OBJET */
SxStart = (long) (160L - (NewY+CurrentObject->width) *
                  InvDistanceTable[(long) NewX]) ;
/* POINT CENTRAL DE L'OBJET */
SxMiddle = (long) (160L - (NewY) * InvDistanceTable[(long) NewX]) ;
/* EXTREMITE DROITE DE L'OBJET */
SxEnd = (long) (160L - (NewY-CurrentObject->width) *
                InvDistanceTable[(long) NewX]) ;
```

Nous voyons que les coordonnées repérant l'objet sont considérées comme étant le centre horizontal de l'objet, puisque pour trouver les coordonnées de l'extrémité gauche, nous rajoutons la moitié de la largeur de l'objet et pour trouver l'extrémité droite, nous soustrayons cette même moitié de largeur.

Notons que ces calculs font en sorte que l'on considère toujours que l'objet fait face au joueur puisque nous expansons les coordonnées seulement dans l'axe des y du schéma (axe qui se matérialise à l'écran par la notion de coordonnées horizontales, rappelons-le). Quand je dis "expansons les coordonnées", j'entends par-là que nous donnons à notre point représentant l'objet, une largeur de façon à ce que notre objet ne soit pas réduit à néant sur l'écran.

Ceci aura pour effet à l'écran que l'objet semblera toujours être dans dans un plan parallèle au joueur (cf. figure 34). On retrouve d'ailleurs cet effet dans *Duke Nukem 3D* lorsque l'on construit un niveau de jeu et que l'on place un sprite : celui-ci semble tourner en même temps que le joueur pour toujours lui faire face. Ceci n'a aucun rapport avec le fait qu'un objet puisse être vu sous plusieurs angles (prédéfinis d'ailleurs) en changeant d'allure, mais nous verrons ce point ultérieurement.

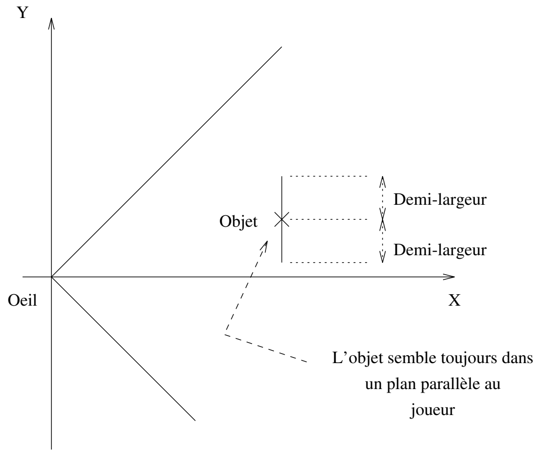

Figure 32: Objets dynamiques dans le repère

Une fois les coordonnées horizontales calculées, via notre mécanisme de projection, nous pouvons regarder, tout comme nous l'avons fait précédemment pour les murs, si notre objet une fois projeté n'est pas en dehors de l'écran :

```
/* SI L'OBJET EST DE TAILLE NULLE, ON LAISSE */
if (SxStart == SxEnd)
  continue ;

/* IDEM SI L'OBJET EST EN DEHORS DE L'ECRAN */
if (SxEnd < 0)
  continue ;

/* PAREIL, MAIS DE L'AUTRE COTE DE L'ECRAN */
if (SxStart > 319)
  continue ;
```

Maintenant vient la notion de "devant-derrière" un mur. Nous devons, avant de continuer à faire des calculs, déterminer si le centre de l'objet ne se trouve pas derrière un mur. Pour ce faire, nous allons utiliser une technique de *Z-buffer*? Nous avons expliqué cette notion de Z-buffer précédemment, lorsque nous avions déjà utilisé une simplification de ce principe pour les murs. Maintenant, nous allons utiliser ce principe dans sa forme complète, mais nous allons limiter ce Z-buffer aux colonnes.

J'entends par là que nous allons créer un réel Z-buffer (c'est à dire mémoriser des distances), mais au lieu de le faire pour chaque point-écran, nous n'allons le faire que pour chaque colonne-écran. Autrement dit, lorsque nous traçons nos murs, colonnes par colonnes, nous devons en plus mettre à jour un tableau contenant pour chaque colonne, la distance de la tranche de vue lui correspondant à l'œil du jour.

Maintenant, pour notre objet, puisque nous connaissons par projection, la colonne que son centre occupe, il ne nous reste plus qu'à comparer la valeur du Z-buffer de cette colonne avec la composante x de l'objet. Si la valeur du Z-buffer est inférieure, alors l'objet est derrière le mur, sinon, il est devant. Dans le cas d'égalité, chacun fait comme ça l'arrange. Notons que pour être complet, il faudrait gérer le fait qu'un mur puisse ne pas avoir de texture et être traversable, chose que nous ne faisons pas dans ce papier. Ceci est laissé au lecteur à titre d'exercice, comme on dit toujours dans les bouquins de maths $\ddot\smile$...

```
/* SI L'OBJET EST PLUS LOIN QUE LE MUR, ALORS IL EST CACHE */
if (NewX >= ColumnDist[SxMiddle])
  continue ;
```

Remarque : Nous voyons bien ici que le test "d'avant-arrière" n'est effectué qu'en 1 point de l'objet : son point central. Ceci prouve bien que nos objets ne sont en réalité que des approximations d'objets 3D. En toute logique, il serait plus correct de tester la visibilité en plusieurs points de l'objet, mais bon, il faut que ça ne prenne quand même pas trop de temps de calcul...

Ensuite vient le temps du calcul, comme pour les murs, des coordonnées verticales des objets. Comme nous voulons simplifier nos calculs au maximum, nous approximons encore nos objets en considérant qu'ils ont la même hauteur partout. Autrement dit, à l'écran, l'objet s'inscrira dans un rectangle. Rassurez-vous, ceci n'est pas une grosse approximation et ne gêne absolument pas à l'écran. Il semblerait que *Duke* fasse exactement la même approximation, à en voir l'effet obtenu par les sprites lors de la création d'un niveau (je ne peux rien assurer là-dessus, mais j'en ai clairement l'impression). Il nous suffit donc de calculer les coordonnées verticales du haut et du bas de l'objet. Comme ces coordonnées sont considérées identiques pour tout l'objet, nous n'avons plus besoin de calculer de deltas, et le rendu s'en trouvera bien allégé en terme de calculs.

```
/* CALCUL DES COEFFICIENTS INTERMEDIAIRES DE HAUTEUR DE L'OBJET */
Rt = PlayerH - CurrentObject->heighttop ;
Rb = PlayerH - CurrentObject->heightbottom ;

/* DEPART, ARRIVEE VERTICALE DE L'OBJET. ON APPROXIME EN FAISANT   */
/* EN SORTE QUE L'OBJET AIT LA MEME HAUTEUR PARTOUT. C'EST INEXACT */
/* CAR IL DEVRAIT RESPECTER AUSSI LA PERSPECTIVE, MAIS BON...      */
STop = (long) (100L + (Rt * InvDistanceTable[(long) NewX])) ;
SBottom = (long) (100L + (Rb * InvDistanceTable[(long) NewX])) ;
```

Ensuite, éventuellement, on peut s'assurer que l'objet n'a pas une hauteur nulle :

```
/* SI L'OBJET EST DE HAUTEUR NULLE, ON LAISSE */
/* if (STop == SBottom)
     continue ; */
```

mais ceci n'a, à mon avis, aucun sens, car les calculs sont faits de telle façon que si l'objet a une taille non nulle, alors la taille à l'écran ne le sera pas non plus (pour s'en persuader, revoir les calculs de projection et le fait que l'on clippe la profondeur d'un objet ou d'un mur à 9999). Si par contre l'objet a une taille nulle, bah, ça n'est pas vraiment normal au niveau de la construction du niveau, je pense...

Comme la routine d'affichage va nécessiter quelques informations relatives aux coordonnées non clippées de l'objet, nous inscrivons ces informations dans des champs prévus à cet effet dans la structure de l'objet, champs uniquement utilisés pour le rendu des scènes :

```
/* MISE A JOUR DES DONNEES NON CLIPPEES DE L'OBJET */
CurrentObject->ncstop = STop ;
CurrentObject->ncsxstart = SxStart ;
```

Ensuite, nous calculons et nous stockons dans l'objet les coefficients constants de calcul incrémental des index pour le mapping.

```
/* INCREMENTS DE MAPPING */
CurrentObject->hincr = (CurrentObject->width * 2) /
  ((double) (SxEnd - SxStart)) ;
CurrentObject->vincr = CurrentObject->globalheight /
  ((double) (SBottom - STop)) ;
```

Comme nous n'avons plus de notion de taille verticale qui change, le mapping s'en trouve grandement simplifié : on peut résoudre directement les recherches d'index dans les 2 directions incrémentalement. Et, qui plus est, comme la hauteur de l'objet est constante en tout point, le coefficient incrémental vertical reste le même quelque soit la colonne.

Il est maintenant temps d'effectuer le clipping des coordonnées horizontales de l'objet, tout comme nous l'avions fait précédemment pour les murs :

```
/* ON CLIPPE HORIZONTALEMENT */
if (SxEnd > 319)
  SxEnd = 319 ;

if (SxStart < 0)
  SxStart = 0 ;
```

Par contre, comme maintenant la hauteur de l'objet est constante, on peut aussi clipper directement les coordonnées verticales. On n'est plus obligé d'attendre le dessin de chaque tranche comme nous le faisions dans le dessin des murs.

```
/* ON CLIPPE VERTICALEMENT */
STop = max (STop, 0) ;
SBottom = min (SBottom, 200) ;
```

Comme le dessin se fait après-coup, par une autre routine, nous devons conserver les diverses coordonnées calculées pour note objet, ainsi, pour chacun des objets à dessiner, notre routine de dessin saura retrouver les valeurs dont elle a besoin :

```
/* MISE A JOUR DES DONNEES CLIPPEES DE L'OBJET */
CurrentObject->stop = STop ;
```

```
CurrentObject->sbottom = SBottom ;
CurrentObject->sxstart = SxStart ;
CurrentObject->sxend = SxEnd ;
CurrentObject->prof = NewX ;
```

Remarque : La notion de profondeur de l'objet (`CurrentObject->prof`) n'intervient pas pour la routine d'affichage, elle sert uniquement pour la routine de tri que nous allons voir immédiatement.

Nous avons donc jusqu'à présent, au niveau des objets dynamiques, réussi à déterminer si un objet était au moins partiellement visible, donc si au moins une partie de celui-ci devait être dessinée. Puisque l'affichage réel est retardé (il a lieu, comme nous l'avons vu) une fois que tous les objets du monde ont été préparés, il faut ranger notre objet visible dans une liste des "objets à afficher", ou liste des "objets visibles". J'ai personnellement choisi de les ranger dans un tableau dont la taille est bornée par une constante que le programmeur pourra choisir en fonction des limites qu'il désire imposer à son moteur. Avant de regarder de plus près cette insertion dans la "liste" ou plutôt le tableau, posons-nous le problème de savoir que faire s'il existe 2 objets dynamiques dont l'un est plus ou moins devant l'autre. Pour régler ce problème, la solution adoptée est de trier les objets par ordre de profondeur décroissante. Autrement dit, notre insertion d'un objet dans le tableau des objets à afficher ne se fait pas n'importe comment, mais de façon à toujours avoir les objets les plus éloignés du joueur en premier. Et c'est justement à l'établissement de cet ordre que sert le champ (`CurrentObject->prof`) mentionnée juste au-dessus.

L'algorithme de tri suivant (celui utilisé dans mon prototype de moteur) est certes trivial, mais possède le mérite de fonctionner et d'illustrer simplement mes propos.

```
for (j=0;j<NbObjectToDraw;)
  {
    if (GlobalObjectTable[ObjectsToDraw[j]].prof < NewX) break ;
    j++ ;
  }

for (k=NbObjectToDraw-1;k>=j;k--)
  ObjectsToDraw[k+1] = ObjectsToDraw[k] ;

ObjectsToDraw[j] = i ;
NbObjectToDraw++ ;
```

Son fonctionnement est très simple : on parcours le tableau des objets déjà mentionnés comme devant être dessinés. Dès qu'on en a trouvé un situé à une profondeur inférieure à la nôtre, on s'arrête de rechercher. On "décale" alors tous les éléments restants du tableau vers la "droite", et on met notre objet dans le trou laissé par le "décalage".

Un compteur *NbObjectToDraw* permet de savoir combien d'objets résident dans le tableau. Il convient de ne pas oublier de l'incrémenter après chaque insertion…

Voilà. Le traitement d'un objet est terminé. Alors, bien entendu, comme nous l'avons précisé précédemment, il faut répéter ce traitement pour tous les objets de notre monde, avant de pouvoir demander leur affichage.

Remarque : Pourquoi avoir choisi un tableau plutôt qu'une liste chaînée pour mémoriser les objets à afficher ? En effet, l'insertion dans une liste chaînée est plus facile (il suffit de modifier 2 pointeurs) que dans un tableau où il faut "décaler" une partie des éléments. En fait, le parcours de la liste chaînée est assez lourd comparativement à celui d'un tableau. Le coût du parcours devient plus intéressant que celui du décalage du tableau lorsque l'on commence à manipuler beaucoup d'objets. En règle générale, il y a assez peu d'objets dynamiques dans une vue (une dizaine de monstres, ça fait déjà pas mal), donc je pense qu'il est préférable de privilégier la rapidité dans la majorité des cas, c'est à dire ceux où il y a peu d'objets. C'est un point de vue qui peut certes être discuté, mais il fallait bien choisir quelque chose. En plus la méthode par tableau est, à mon avis, plus simple à implémenter, et donc plus simple à comprendre dans un exposé comme le nôtre où il est plus question de pédagogie que de performances extraordinaires rivalisant avec *Duke*, *Quake* et bien d'autres encore ◡̈...

10.3 L'affichage des objets dynamiques

A la sortie de la fonction de préparation des objets, nous obtenons :

- Un tableau comportant tous les objets à afficher, triés par profondeur décroissante.
- Un compteur indiquant le nombre d'objets contenus dans ce tableau.
- Des objets (ceux appartenant au tableau) avec les champs de coordonnées et autres informations diverses correctement initialisés.

Voyons maintenant comment utiliser tout ceci afin d'afficher nos objets dynamiques. Cette fonction va en fait fort logiquement parcourir le tableau des objets à afficher, et les dessiner les uns après les autres. Ces objets étant triés par ordre de profondeur décroissante, on affichera les plus lointains en premier, puis ceux plus proches viendront recouvrir les plus lointains. Ainsi, nous aurons un rendu correct des objets avec les bons recouvrements en fonction de leurs distances.

Notre fonction d'affichage sera donc une grosse boucle répétant le même travail pour chaque objet du tableau (d'où l'intérêt d'avoir conservé le compteur indiquant le nombre d'objets dans notre tableau) :

```
for (i=0;i<NbObjectToDraw;i++)
  {
    CurrentObject = &GlobalObjectTable[ObjectsToDraw[i]] ;
    ... TRAITEMENT ...
  }
```

Entrons donc dans le détail de ce traitement d'affichage... Nous allons commencer par récupérer les informations (de coordonnées) que la routine de préparation des objets a inscrites dans l'objet, et mettre ces informations dans des variables locales à la fonction afin d'y avoir un accès plus rapide :

```
    STop = CurrentObject->stop ;
    SBottom = CurrentObject->sbottom ;
```

```
SxStart = CurrentObject->sxstart ;
SxEnd = CurrentObject->sxend ;
NCSxStart = CurrentObject->ncsxstart ;
NCSTop = CurrentObject->ncstop ;
HIncr = CurrentObject->hincr ;
VIncr = CurrentObject->vincr ;
```

Nous allons, tout comme nous l'avons fait pour les murs, afficher nos objets tranche verticale par tranche verticale. Nous commençons donc par calculer l'index horizontal de départ où aller chercher nos points de dessin dans la bitmap de notre objet :

```
HIndex = ((double) (SxStart-NCSxStart)) * HIncr ;
```

Cet index sera incrémenté à chaque fois que l'on passera au dessin de la colonne suivante. Nous allons maintenant faire une boucle qui affiche toutes les tranches verticales de notre objet. Dans cette boucle, nous commençons par déterminer l'index qui nous manque pour pouvoir accéder aux points correctement dans la texture, à savoir l'index vertical.

Remarque : Notons que cet index vertical est identique à chaque début de colonne, puisque nous avons vu que notre objet est de hauteur constante, aussi une optimisation possible serait de le calculer une fois pour toute avant de passer au dessin des tranches (optimisation non négligeable car ce calcul se trouve dans une boucle, donc est effectué un nombre de fois éventuellement non négligeable).

Le reste du traitement ressemble très fort à celui que nous avons vu pour le dessin des murs. On calcul les adresses de début et de fin de la ligne verticale que l'on va dessiner point par point (travailler avec des pointeurs est plus efficace que fonctionner par indexation, c'est à dire plus efficace qu'en écrivant `DblBuffer[i] = ... ; i = i + ... ;`).

```
for (ColumnIndex=SxStart;ColumnIndex<=SxEnd;ColumnIndex++)
  {
    VIndex = ((double) (STop-NCSTop)) * VIncr ;

    PointAdr = DblBuffer + (STop * 320 + ColumnIndex) ;
    PointAdrEnd = DblBuffer + (SBottom * 320 + ColumnIndex) ;
    for (;PointAdr < PointAdrEnd; PointAdr += 320)
      {
        ColorPoint = GET_SPRITE_POINT ((short) HIndex, (short) VIndex) ;
        if (ColorPoint != 0)
          (*PointAdr) = ColorPoint ;

        VIndex += VIncr ;
      }

    HIndex += HIncr ;
  }
}
```

La seule différence réside dans le fait que lorsque nous récupérons la couleur du point à afficher, nous regardons si ce point est de couleur 0, c'est à dire de couleur ... incolore. Dans ce cas, cela signifie qu'il n'y a pas à proprement parler de point à afficher et que le fond doit être visible. En effet, les objets ne sont pas forcément des rectangles pleins, certaines de leurs parties peuvent laisser passer la lumière. Ainsi nous ne recouvront que la partie nécessaire du décors arrière, et notre rendu est correct. Par exemple, si notre objet est une roue de vielle charrette, il y aura du jour entre les rayons, et donc le fond sera visible entre ces rayons. De même, entre les jambes d'un bonhomme, nous verrons le décors, etc...

Remarque : Le fait que la couleur 0 soit considérée dans une bitmap comme la couleur transparente est une pure convention, on peut en choisir un autre sans aucun problème. Bon, généralement, c'est quand même la couleur n^{o}0 qui est choisie pour la transparence.

11 Conclusion

Voilà. Cet exposé se termine. Tout n'a bien évidemment pas été dit en matière de moteur 3D, mais j'espère que ce qui l'a été est à peu près clair. Il reste encore pas mal de points à aborder pour faire un moteur totalement fonctionnel, et en particulier pour pouvoir en faire une utilisation digne d'un jeu ou d'un logiciel de qualité :

- Il faut étendre notre prototype de façon à autoriser plusieurs textures au choix.
- On peut avoir envie de texturer aussi le sol et le plafond.
- Des effets de luminosité différente dans les secteurs, ou selon l'éloignement du joueur sont aussi sympathiques.
- Il faut aussi rajouter des attributs aux linedefs qui forment notre monde (infranchissables, transparentes, etc...) Comme actuellement, nous n'avons pas de tels attributs, en particulier pas d'attribut de transparence, s'il existe une linedef quelque part, même si elle sépare 2 secteurs, celle-ci sera texturée. Donc au premier abord, le joueur aura l'impression qu'il y a des murs partout et qu'il ne peut pas aller dans l'autre secteur...
- Il faut gérer des portes. Ceci est un point sur lequel j'ai travaillé dernièrement et qui a donné naissance à une nouvelle version de moteur. Cela peut être inclus dans une logique plus générale : la gestion des secteurs de hauteur différente. En effet, une porte n'est qu'un secteur qui bouge.

  Sans entrer dans les détails, le première solution consiste à modifier la notion de ligne entre 2 secteurs : en effet, dans le prototype défini ici, au lieu d'avoir, pour séparer 2 secteurs, 1 seule ligne avec 2 "faces", nous avons 2 lignes de sens opposé (c.f papier sur les BSP), ce qui permet au joueur de toujours avoir une face droite à voir. Rajouter les murs de hauteur différente en utilisant 2 lignes pour délimiter 2 secteurs adjacents peut compliquer notablement la construction du BSP selon la façon dont on s'y prend. A la place, il vaut mieux conserver 1 seule ligne et lui attribuer 2 faces, puis construire le BSP non plus à partir des linedefs, mais à partir de sous-linedefs Et comme par hasard, on re-tombe exactement sur un modèle isomorphe à celui de *Doom* (modèle dont les structures sont décrites dans les *Unofficial Doom Specs*), avec des Segs et des Sidedefs en plus des Linedefs...

La seconde solution, celle que j'ai réellement implantée arrive à conserver notre structure originale. Par rapport au modèle de linedef apparemment utilisé dans *Doom*, il semble que j'utilise une indirection mémoire de plus par mur dans 1 calcul ce qui n'est quand même pas hors de prix ! Par contre l'avantage est que je peux conserver sans problème mes structures de données et mon constructeur de BSP (les étendre tous deux un peu, mais pas casser tout ce qui est déjà fait), donc étendre mon moteur modulairement (ce qui est pratique sur le plan pédagogique).

Le point le plus drôle, pour rajouter à notre moteur nos fameux murs, c'est qu'il devient quasiment plus complexe de se faire un éditeur pour créer des tableaux afin de tester notre moteur (c'est bien trop long de se les taper à la main), que d'étendre le moteur en lui-même ! Enfin bref, même si la philosophie profonde du moteur ne s'en trouverait pas radicalement changée, tout ceci nécessite quand même un peu de boulot

Je pense que je ferai un addendum à ce papier, afin d'expliquer comment obtenir ces murs de hauteur différente, mais ce sera pour plus tard. Chaque chose en son temps...

- On peut élargir ce concept de secteur mobile au delà des portes et faire comme les moteurs modernes, des secteurs montant, descendant, s'étirant, tournant, etc...

- Les optimisations d'implémentation dont nous avons parlé permettraient de gagner de la vitesse en soulageant un peu le CPU.

- Les collisions avec les murs n'ont pas été traitées. Leur détection repose sur une structure : la *blockmap* fort bien décrite dans les *Unofficial Doom Specs* de *Matthew S. Fell* (cf. Références bibliographiques). En 2 mots, cette structure est basée sur une découpe de la carte par une grille de 128×128 points. Pour chaque case de la grille, on conserve une liste des linedefs qui occupent au moins en partie cette case. Ensuite, quand le joueur se déplace, on recherche dans quelle "case" de la blockmap il se trouve et on recherche s'il entre en collision avec une des linedefs se trouvant dans cette case. Ceci permet d'économiser un grand nombre de tests d'intersection puisque nous travaillons seulement avec les linedefs présentes dans la case où le joueur se trouve.

- Les sons sont un élément indispensable à rajouter au moteur. Ils *ne font pas partie* du moteur, mais rajoutent beaucoup en réalisme.

- La gestion de l'intelligence artificielle des objets dynamiques (s'ls ont une forme d'intelligence), ou au moins la gestion de leurs mouvements, est aussi une partie non négligeable d'un bon logiciel.

Les bases étant jetées, il faut continuer à rechercher dans ces directions. Ceci viendra peut-être dans un prochain papier. Je vais de mon côté regarder ces différents aspect dès que j'aurai un peu de temps. Mais tout ceci est une autre histoire...

12 Annexe 1 : source de la fonction de rendu d'un mur

Ce source est fourni tel qu'il se trouve dans mon moteur. Je n'ai même pas changé les commentaires, ni les fautes d'orthographe dans ces commentaires ☺. D'ailleurs, cette fonction est relativement bien fournie en commentaires.

```
/* ****************************************************************** */
/* FONCTION PRINCIPALE DE RENDU. C'EST ELLE QUI REND LES SEGMENTS. ELLE */
/* PREND EN PARAMETRE UNE LINEDEF, ET L'AFFICHE SI ELLE BIEN VISIBLE,   */
/* EFFECTUE LES CLIPPINGS ET LES CALCULS DE PERSPECTIVE. ELLE INITIE    */
/* AUSSI LE MAPPING DE LA TEXTURE EN PRE-CALCULANT CERTAINES VALEURS.   */
/* ****************************************************************** */
void DrawLinedef (Linedef)
     struct Linedef *Linedef ;
{
  /* NOUVELLES COORDONNEES DE LA LINEDEF APRES ROTATION ET TRANSLATION */
  double NewXStart ;
  double NewYStart ;
  double NewXEnd ;
  double NewYEnd ;
  /* COORDONNEE HORIZONTALE DE DEPART DE LA LINEDEF SUR L'ECRAN (APRES PROJ) */
  long SxStart ;
  /* COORDONNEE HORIZONTALE DE FIN DE LA LINEDEF SUR L'ECRAN (APRES PROJ) */
  long SxEnd ;
  /* COORDONNEES VERTICALES DE DEBUT ET DE FIN DE L'EXTREMITE */
  /* GAUCHE DU SEGMENT (EXTREMITE DE DEBUT DU SEGMENT).       */
  long STopStart, STopEnd ;
  /* COORDONNEES VERTICALES DE DEBUT ET DE FIN DE L'EXTREMITE */
  /* DROITE DU SEGMENT (EXTREMITE DE DEBUT DU SEGMENT).       */
  long SBotStart, SBotEnd ;
  /* INCREMENTS VERS LE HAUT VERS LE BAS, POUR CALCULER INCREMENTALEMENT */
  /* LES EXTREMITES VERTICALES DE CHAQUE TRANCHE DE MUR.                 */
  double DeltaTop, DeltaBottom ;
  long SxStartCl ;       /* SCREEN X START CLIPPED */
  long SxEndCl ;         /* SCREEN X END CLIPPED */
  /* TO ET BOTTOM COURANTS DANS LA BOUCLE DE DESSIN DU MUR */
  double Top, Bottom ;
  /* INDEX DE LA COLONNE COURANT DANS LE DESSIN DU MUR COURANT */
  short ColumnIndex ;
  /* COEFFICIENTS INTERMEDIAIRES DE HAUTEUR DU JOUEUR */
  double Rt, Rb ;
  /* DIVERS VARIABLES NECESSITEES POUR LE MAPPING DE LA TEXTURE */
  double A ;      /* ANGLE DU MUR COURANT */
  double l ;      /* LONGUEUR DU MUR COURANT, CHARGE EN LOCAL POUR ACCELERER */
  double r, s ;   /* COEFFICIENTS DE CALCUL DES PRE-REQUIS DE MAPPING */
  double PDist ;  /* DISTANCE DU PT DE VUE A LA PROJECTION ORTHO SUR LE MUR */
```

```
double BtoBP ;  /* DISTANCE DU POINT DE PROJ ORTHO AU DEBUT DU MUR */
double TanIndex ;  /* ANGLE UTILISE POUR LA DETERMINATION DU MAPPING */
/* EXTREMITES HAUT/BAS CLIPPEE A 0 - 320 POUR LE DESSIN DE CHAQUE   */
/* TRANCHE DE VUE. CORRESPOND A D'ANCIENNES VARIABLES DE LA ROUTINE */
/* DE DESSIN QUI SE TROUVE MAINTENANT INLINEE.                      */
long RealTop, RealBottom ;
/* POINTEURS DE DEPART ET D'ARRIVEE EN MEMOIRE DOUBLE-BUFFER */
/* CORRESPONDANT A LA LIGNE VERTICALE A DESSINER A CHAQUE    */
/* ETAPE. CE SONT AUSSI D'ANCIENNES VARIABLES DE LA ROUTINE  */
/* DE DESSIN QUI SE TROUVE MAINTENANT INLINEE.               */
unsigned char *PointAdr, *PointAdrEnd ;



/* ON CALCULE LES NOUVELLES COORDONNEES DU SEGMENT. POUR CELA, ON */
/* TRANSLATE LE REPERE AU NIVEAU DU JOUEUR ET ON LE FAIT TOURNER  */
/* DE FACON A CE QUE LA DIRECTION DE VUE DU JOUEUR CONCIDE AVEC   */
/* L'ANGLE 0.                                                     */
NewXStart = ((GlobalVertexTable[Linedef->src]).x-PlayerX)
                                 *
                        cos (TO_RAD(-PlayerAngle))
                                 -
                    ((GlobalVertexTable[Linedef->src]).y-PlayerY)
                                 *
                        sin (TO_RAD(-PlayerAngle)) ;
NewXEnd = ((GlobalVertexTable[Linedef->dest]).x-PlayerX)
                                 *
                        cos (TO_RAD (-PlayerAngle))
                                 -
                    ((GlobalVertexTable[Linedef->dest]).y-PlayerY)
                                 *
                        sin (TO_RAD (-PlayerAngle)) ;

NewYStart = ((GlobalVertexTable[Linedef->src]).y-PlayerY)
                                 *
                        cos (TO_RAD (-PlayerAngle))
                                 +
                    ((GlobalVertexTable[Linedef->src]).x-PlayerX)
                                 *
                        sin (TO_RAD (-PlayerAngle)) ;
NewYEnd = ((GlobalVertexTable[Linedef->dest]).y-PlayerY)
                                 *
                        cos (TO_RAD (-PlayerAngle))
                                 +
                    ((GlobalVertexTable[Linedef->dest]).x-PlayerX)
```

```
                                 *
                         sin (TO_RAD (-PlayerAngle)) ;

/* LE MUR EST TOTALEMENT DERRIERE NOUS, DONC ON LAISSE */
if ((NewXStart < ZMIN) && (NewXEnd < ZMIN))
    return ;

/* PRE-REQUIS POUR LE MAPPING ET COEFFICIENTS INTERMEDIAIRES */
A = Linedef->angle + TO_RAD (-PlayerAngle) ;
l = Linedef->len ;
r = ((NewYStart-0)*(NewYStart-NewYEnd)-(NewXStart-0)*(NewXEnd-NewXStart))
  / (l*l) ;
s = ((NewYStart-0)*(NewXEnd-NewXStart)-(NewXStart-0)*(NewYEnd-NewYStart))
  / (l*l) ;
/* DISTANCE DU POINT (0,0) (OEIL) A LA DROITE REPRESENTANT LE MUR */
PDist = s * l ;
/* DISTANCE DU LA PROJ ORTHO DE L'OEIL SUR LE MUR AU DEBUT DU MUR */
BtoBP = r * l ;

/* ON VA FAIRE LES CLIPPINGS POUR SI L'UN DES x EST DERRIERE NOUS. */
if (NewXStart < ZMIN)
  {
    /* IL N'Y A PAS DE RISQUE QUE LA TANGENTE SOIT INFINIE   */
    /* EN EFFET, CA CORRESPOND AUX CAS OU LES DEUX ABSCISSES */
    /* SONT EGALES. OR SI ELLES SONT EGALES NEGATIVES, ALORS */
    /* ON A DEJA LAISSE TOMBER LE MUR CAR IL EST DERRIERE LE */
    /* JOUEUR. SI ELLES SONT EGALES POSITIVES, ALORS ON NE   */
    /* PEUT PAS ETRE RENTRE DANS CETTE SECTION DE CODE, CE   */
    /* QUI EST NORMAL CAR DANS CA CAS,IL N'Y A PAS DE Z      */
    /* CLIPPING A FAIRE.                                     */
    NewYStart = NewYStart + (ZMIN-NewXStart) * tan(A) ;
    NewXStart = ZMIN ;
  }

if (NewXEnd < ZMIN)
  {
    /* LES DEUX TESTS SONT EXCLUSIFS CAR NewXEnd ET NewXStart  */
    /* NE PEUVENT PAS ETRE < 0 ICI EN MEME TEMPS CAR CE CAS-LA */
    /* A DEJA ETE FILTRE JUSTE AVANT. DONC LES 2 TESTS NE      */
    /* PEUVENT ETRE SATISFAITS A LA FOIS.                      */
    NewYEnd = NewYEnd + (ZMIN-NewXEnd) * tan(A) ;
    NewXEnd = ZMIN ;
  }

/* MAINTENANT, ON CLIPPE LA PROFONDEUR (CORRESPONDANT SUR NOTRE PLAN   */
```

```
/* A L'ABSCISSE) DE NOS 2 POINTS DE FACON A CE QU'AU BOUT D'UN MOMENT, */
/* TOUS LES MURS TROP ELOIGNES SOIENT QUAND MEME DE LA MEME HAUTEUR.   */
/* CA PERMET AU PASSAGE D'UTILISER UNE TABLE PRECALCULEE PLUTOT QUE DE */
/* LANCER UNE FONCTION DE CALCUL A CHAQUE FOIS.                        */
if (NewXStart > 9999)
    NewXStart = 9999 ;

if (NewXEnd > 9999)
    NewXEnd = 9999 ;

/* ON CALCULE LA PROJECTION EN X <SUR L'ECRAN> DES EXTREMITES DU MURS. */
/* AUTREMENT DIT, ON OBTIENT LES NUMEROS DE COLONNES D'OU PART ET OU   */
/* ARRIVE LE MUR.                                                      */
SxStart = (long) (160L - (NewYStart) * InvDistanceTable[(long) NewXStart]) ;
SxEnd = (long) (160L - (NewYEnd) * InvDistanceTable[(long) NewXEnd]) ;

/* ON VERIFIE QUE LE MUR N'EST PAS TOTALEMENT INVISIBLE */

/* SI LE MUR EST DE TAILLE NULLE, ALORS ON S'EN VA */
if (SxStart == SxEnd)
    return ;

/* SI LE MUR TERMINE AVANT L'ENTREE DANS LE CHAMP DE */
/* L'ECRAN, IL N'Y A RIEN A AFFICHER.                */
if (SxEnd < 0)
    return ;

/* SI LE MUR COMMENCE APRES LA SORTIE DU CHAMP */
/* DE L'ECRAN, IL N'Y A RIEN A AFFICHER.       */
if (SxStart > 319)
    return ;

/* COEFFICIENTS DE HAUTEUR DES YEUX DU JOUEUR */
Rt = PlayerH - CeilingH ;
Rb = PlayerH - FloorH ;

/* ON CALCULE LES COORDONNEES VERTICALES DE DEBUT ET DE FIN */
/* DE CHACUNE DES EXTREMITES DU MUR PROJETE SUR L'ECRAN     */
STopStart = (long) (100L + (Rt * InvDistanceTable[(long) NewXStart])) ;
SBotStart = (long) (100L + (Rb * InvDistanceTable[(long) NewXStart])) ;
STopEnd = (long) (100L + (Rt * InvDistanceTable[(long) NewXEnd])) ;
SBotEnd = (long) (100L + (Rb * InvDistanceTable[(long) NewXEnd])) ;

/* INITIALISATION DES VALEURS PAR DEFAUT DES EXTREMITES */
/* VERTICALES DU MUR COURANT.                           */
```

```
Top = STopStart ;
Bottom = SBotStart ;

/* ON CALCULE LES DELTAS POUR CALCULER  */
/* INCREMENTALEMENT LES HAUTEURS DU MUR */
DeltaTop = (STopEnd-STopStart) / (double) (SxEnd-SxStart) ;
DeltaBottom = (SBotEnd-SBotStart) / (double) (SxEnd-SxStart) ;

/* MAINTENANT, ON FAIT LES CLIPPING A LA TAILLE HORIZONTALE DE L'ECRAN */
SxEndCl = SxEnd ;
if (SxEnd > 319)
  {
    /* ON NE S'OCCUPE PAS D'EVENTUELLES EXTREMITES VERTICALES DE   */
    /* LA FIN DU MUR, CAR EN FAIT, ON N'A BESOIN QUE DE CELLES DU  */
    /* DEBUT DU MUR CAR APRES, ON CALCULERA TOUT INCREMENTALEMENT. */
    SxEndCl = 319 ;
  }

SxStartCl = SxStart ;
if (SxStart < 0)
  {
    /* ON MET A JOUR LES EXTREMITES VERTICALES DU MUR COURANT, CAR LE  */
    /* MUR COURANT VA ETRE LE PREMIER REELLEMENT DANS LE CHAMP DE      */
    /* L'ECRAN ET NON CELUI TROUVE PAR LA PROJECTION THEORIQUE SUR LE  */
    /* PLAN VIRTUEL ET INFINI DE L'ECRAN.                              */
    Top -= (double) (SxStart * DeltaTop) ;
    Bottom -= (double) (SxStart * DeltaBottom) ;
    SxStartCl = 0 ;
  }

/* POUR TOUTES LES COLONNES QUE LE MUR OCCUPE (SAUF RECOUVREMENT) */
/* A L'ECRAN, ON VA DESSINER LES "TRANCHES VERTICALES".           */
for (ColumnIndex=SxStartCl;ColumnIndex<=SxEndCl;ColumnIndex++)
  {
    if (ColumnDone[ColumnIndex] != 1)
      {
        /* ON NE DESSINE QUE SI CETTE COLONNE DE L'ECRAN N'EST */
        /* PAS DEJA OCCUPEE PAR UN MUR QUI ETAIT PLUS PROCHE.  */
        TanIndex = A + TO_RAD(90) - AngleTable[ColumnIndex] ;
        /* HIndex = ((PDist * tan(TanIndex)) + BtoBP) % 64 */
        HIndex = (short) ((PDist * tan(TanIndex)) + BtoBP) & 0x00003F ;
        /* ON CONTRE-CARRE LES ERREURS D'ARRONDIS QUI NOUS */
        /* DONNENT DES SOUCIS AUX COINS */
        if (HIndex < 0) HIndex = 0 ;
        VIncr = 64 / (Bottom - Top) ;
```

```
        /* ROUTINE DE DESSIN DE LIGNE INLINEE */

        /* ON CLIPPE EN HAUT */
        RealTop = max (Top,0) ;
        /* ON CLIPPE EN BAS */
        RealBottom = min (Bottom,200) ;
        VIndex = (RealTop-Top) * VIncr ;
        /* CALCUL DE L'ADRESSE DE DEPART */
        PointAdr = DblBuffer + (RealTop * 320 + ColumnIndex) ;
        /* ... ET DE L'ADRESSE DE FIN */
        PointAdrEnd = DblBuffer + (RealBottom * 320 + ColumnIndex) ;
        /* ET MAINTENANT, ON ECRIT LES POINTS DANS LE DOUBLE BUFFER */
        for (;PointAdr < PointAdrEnd; PointAdr += 320)
          {
            (*PointAdr) = GET_BITMAP_POINT (HIndex, (short) VIndex) ;
            /* ON CALCULE INCREMENTALEMENT LA NOUVEL INDEX VERTICAL */
            /* DANS LA TEXTURE QUE L'ON MAPPE.                      */
            VIndex += VIncr ;
          }

        /* MAINTENANT, LA COLONNE EST OCUPPEE PAR UN MUR. */
        ColumnDone[ColumnIndex] = 1 ;
        /* ON MEMORISE LA DISTANCE DE L'OEIL AU MUR (TJS cos != 0) */
        ColumnDist[ColumnIndex] = fabs (PDist / cos (TanIndex)) ;
        /* ON A FAIT UNE COLONNE DE PLUS */
        NbColumnsDone ++ ;
      }

    /* ON CALCULE INCREMENTALEMENT LES EXTREMITES VERTICALES */
    /* DE LA PROCHAINE TRANCHE DE MUR.                       */
    Top += DeltaTop ;
    Bottom += DeltaBottom ;
  }
}
```

13 Annexe 2 : Références bibliographiques

13.1 Références graphiques

- **comp.graphics.algorithms FAQ** disponibles entre autres sur les sites suivants : *ftp://rtfm.mit.edu/pub/usenet-by-group/news.answers/graphics/algorithms-faq*, *ftp://ftp.seas.gwu.edu/pub/rtfm/comp/graphics/algorithms/comp.graphics.algorithms_Fr*, *http://www.cis.ohio-state.edu/hypertext/faq/usenet/graphics/algorithms-faq.html*, *http://www.geom.umn.edu/software/cglist/*

- **The unofficial DOOM specs** by Matthew S Fell (un excellent document si vous cherchez à connaître la description des structures internes de *Doom* (Champs des WAD, format du fichier, valeurs des différents tags, où aller modifier les binaires pour hacker le jeu, quelques informations loin d'être inintéressantes sur le BSP de *Doom*, etc...)

- **L'archive dview.zip** contenant les sources et l'exécutable d'un programme sous DOS capable d'afficher les WADS de *Doom* sans texturage et réalisé par Jake Hill. Des sources qu'il fait vraiment bon avoir sous la main pour comme début d'étude des moteurs graphiques à la *Doom*. Même si, comme toutes les sources, c'est difficilement analysable (une bonne explication vaut toujours mieux que toutes les sources du monde), ceci permet de comprendre à quoi servent certaines structures des WADS, comment on peut utiliser un BSP et ce que l'on peut lui rajouter pour aller plus vite.

  Je me suis d'ailleurs inspiré de son travail pour comprendre le fonctionnement des moteurs à polygones (enfin, plutôt pour voir émerger certaines ébauches d'idées que j'ai, par la suite, revues et approfondies sur papier car la lecture du source ne m'a pas suffit à comprendre) et donc réaliser le moteur que je suis actuellement en train d'écrire. J'ai particulièrement retenu de son travail, la notion d'affichage par polygones par opposition au ray-casting, seule méthode que je connaissais à l'époque. D'ailleurs, certains points de son source restent obstinément obscurs, en particulier sa gestion des murs de hauteur différente, notion que j'ai entièrement implantée d'une autre façon (avec manifestement moins de structures de données ; mais je ne dis pas que c'est forcément une meilleure méthode, mais ça me semble plus simple à comprendre) dans la nouvelle version de mon moteur (qui sera sans doute expliquée un peu plus tard, dans un ajout à ce papier).

  En bref, une bien belle archive à mon goût, à posséder absolument et qui doit être disponible sur *le site ftp de x2ftp.oulu.fi en /pub/msdos/programming/...* Si vous lisez son source, vous pourrez remarquer une approche isomorphe avec celle décrite dans ce papier. Notons quand même que le présent papier n'est pas une simple explication textuelle du programme réalisé par Jack Hill... D'ailleurs, lui gérait la notion de murs de hauteur différente, chose que je n'ai pas gérée dans cette version de mon papier. Par contre, lui ne gérait pas le mapping ni les objets dynamiques.

- **One Approach to Real Time Texture Mapping** par *William D. Doughty*, Novembre 1994, disponible sur Internet à je ne sais plus quelle adresse.

- **The Mathematics of Three-Dimensional Manipulations and Transformations** par Trip V. Reece.

13.2 Réferences trigonométriques

- Encore **mon Papa**, car depuis mon papier sur les BSP, je n'ai toujours pas relu mes cours de collège ⌣̈...

13.3 Réference corrections (orthographe, grammaire, style, notation, oublis, erreurs diverses, ...)

- **Ma pupuce**, et **mon correcteur orthographique** que j'ai fait moi-même personnellement, car j'ai enfin eu le courage de l'utiliser ⌣̈...

Le présent document a été rédigé avec LaTeX

Le présent document est déposé à la **Bibliothèque Nationale Française** sous mon nom en tant qu'auteur. Ceci n'a pas pour but de le rendre payant d'une façon ou d'une autre. Au contraire, ceci sert uniquement à en assurer la libre distribution, en évitant que certains auteurs malhonnêtes ne se l'approprie et mettent un droit dessus, auquel cas, si je n'avais aucune preuve de ma propriété, je ne pourrais plus diffuser mon papier librement et gratuitement.