Les principes d'un moteur 3D

Par Plouf

www.mandragor.org

Archive PDF du cours intitulé: Les principes d'un moteur 3D.
Auteur: Philippe Detournay (theplouf@yahoo.com)

PDF généré automatiquement le 4/12/2003, à 18:50.
Copyright (c) 2003 Philippe Detournay.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the file 'LICENSE' distributed with this archive. "GNU Free Documentation License".

Groupe de programmation Mandragor, www.mandragor.org.

Sommaire

1. Support mathématique

1.1. Matrices et opérations matricielles
1.2. Les coordonnées normalisées
1.3. Notion de base
1.4. Notion d'espace

2. Le calcul des lumières

3. Les objets

4. Un premier pipeline

5. La projection

6. Le clipping 3D

7. Le backface culling

8. Un pipeline un peu plus complet

9. Le tri des faces et le Z-Buffer

10. Considérations sur la transparence

11. Les triangles 2D

11.1. Introduction
11.2. La correction de perspective

12. Les portals

13. Annexes

13.1. Matrices de transformation 3D 4x4

Support mathématique

1.1. Matrices et opérations matricielles

Introduction

Il doit être facile de trouver sur le net des tas et des tas de textes traitant de ces sujets, mais ils sont tous en anglais et puis bon, autant partir du bon pied et commencer par le début. Hé oui, mauvaise nouvelle, la 3D c'est avant tout un problème mathématique.

Nous allons commencer en 2D, car c'est quelque chose qui est en général beaucoup plus instinctif car on peut le dessiner facilement, Paint ayant encore de beaux jours devant lui.

Mais avant tout cela, je vais vous dire ce que c'est qu'une matrice et une opération matricielle, d'un point de vue totalement abstrait. La seule chose dont vous devez toujours vous souvenir, c'est que, comme pour tout, on peut toujours s'en passer, même dans la 3D, il ne s'agit juste que d'une manière plus simple pour représenter les choses.

Car, après tout, le jour où le mec qui a inventé les maths (Monsieur Mathématique?) s'est dit que l'opération << + >> serait une bonne idée, il n'a peut-être pas songé tout de suite à l'opération << * >>. Pour faire 4*3, il s'est dit qu'il était tout aussi simple de faire 4+4+4, ce dont tout le monde en conviendra.

Bon, pour faire 4*48 ça commence à devenir un peu chiant, mais c'est toujours faisable, non ? Par contre pour faire 3.14159*78.9, c'est plus compliqué. Bien sûr, on peut s'en sortir, en multipliant tout par un facteur un million pour éviter les virgules, et puis tout diviser après. Après tout, n'est-ce pas là justement le principe dit de << la virgule fixe >> ? Mais je m'égare...

Vous aurez compris, il semblait utile de définir un nouveau concept, la multiplication, avec des propriétés à elle, des opérateurs à elles, des inversions, des trucs et des machins qui n'existaient pas avec l'addition, même si la multiplication n'est jamais basée que sur l'addition. Vous devez avoir à l'esprit que la multiplication est inutile, on peut toujours s'en passer, comme on peut toujours se passer des calculs matriciels, c'est juste que c'est diablement plus simple.

Matrice

Une matrice, c'est un tableau de réels, on parlera de la taille d'une matrice comme on parle de la

taille d'un tableau. C'est un tableau bi-dimensionnel, il est donc constitué de lignes et de colonnes. Une matrice 4x3, c'est un tableau contenant 4 lignes et 3 colonnes.

On dit d'une matrice qu'elle est carrée quand son nombre de lignes est égal à son nombre de colonnes. Une matrice dite identité est une matrice carrée, dont tous les éléments sont nuls sauf les éléments de la première diagonale qui valent tous un.

En général, la notation préférée pour les matrices ce sont les lettres majuscules. C'est ainsi que l'on parlera de la matrice A, ou B, etc. Quand on dit l'élément A(i,j) il s'agit de l'élément du tableau se trouvant à la ligne i et à la colonne j. Les lignes et les colonnes sont numérotées à partir de 1.

Bon, jusque là, rien d'infaisable.

On peut additionner deux matrices, à condition qu'elles soient de mêmes tailles (même nombre de lignes et même nombre de colonnes), le résultat étant une troisième matrice où chaque élément est la somme des éléments qui se trouvaient à la même position dans les matrices de départ.

n peut surtout multiplier deux matrices. Malgré ce que l'on pourrait supposer, ça n'est pas du tout la même chose que l'addition (c'était trop simple, non ?). On peut multiplier une matrice de taille (i,j) avec une matrice de taille (j,k), et ainsi former une matrice de taille (i,k) selon la règle suivante :

```
Si C=A*B, alors C(i,k)=Somme(Pour m allant de 1 à j)A(i,m)*B(m,k)
```

Oula.

Bon, un petit exemple :

ca nous donne

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

$$\begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{pmatrix}$$

Attention ! Contrairement à l'addition et contrairement à la multiplication << classique >>, la multiplication matricielle n'est pas commutative. En d'autres termes :

```
A*B <> B*A
```

On parle de la transposée d'une matrice lorsque l'on a inversé les lignes et les colonnes. On parle alors de A' ou At

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

$$\begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

La transposée de

On dit d'une matrice qu'elle est une matrice ligne si elle n'a qu'une seule ligne. De même, on dira qu'une matrice est une matrice colonne si elle n'a qu'une seule colonne. La transposée d'une matrice ligne donne une matrice colonne et vice-versa.

L'inverse d'une matrice A, notée A-1, est telle que A*A-1=I, où I est la matrice identité. En général on ne parle de matrice inverse que pour les matrices carrées. Une propriété dit que si A est une matrice carrée, alors A*A-1= A-1*A=I, ce qui n'était pas évident, je vous rappelle que le produit matriciel n'est pas commutatif.

Bon, normalement ici vous devez commencer à en avoir marre de ces cochonneries de matrices. On va donc passer à légèrement autre chose.

Matrices et transformations géométriques

Peut-être savez-vous que pour faire tourner un point (x,y) autour de l'origine, dans un plan, la formule à utiliser est :

```
x'=x*Cos(a)-y*Sin(a)
y'=x*Sin(a)+y*Cos(a)
```

Mais si vous regardez cette forume avec ce que l'on vient de voir un peu au dessus, vous vous rendrez compte qu'il s'agit de la multiplication matricielle suivante :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} Cos(a) & -Sin(a) \\ Sin(a) & Cos(a) \end{pmatrix} * \begin{pmatrix} x \\ y \end{pmatrix}$$

Ainsi donc, une rotation peut se représenter par une multiplication matricielle, pour autant que les points soient eux-mêmes représentés par des matrice colonnes.

A présent, si vous voulez faire deux rotations successives, vous allez devoir utiliser deux fois cette formule. Et si vous voulez faire deux rotations à 2000 points, vous allez devoir, finalement, vous taper 4000 opérations matricielles.

Et bien non (et c'est là qu'on va piger à quoi vont nous servir les matrices), car la multiplication de deux matrices de transformation (comme la matrice de multiplication vue plus haut) est une nouvelle matrice de transformation, qui correspond à la composition des deux transformations. Si vous ne comprenez pas ce que je viens de raconter, ça se résume par :

Si A est une matrice de rotation d'un angle a, et B est une matrice de rotation d'un angle b, alors A*B est une matrice de rotation d'angle a+b.

Bref, pour vos 2000 points, vous faites d'abord le calcul du produit matriciel et ensuite vous faites uniquement 2000 opérations matricielles sur vos points. En pratique ici, c'est totalement inutile car tout le monde sait bien qu'il aurait suffit de créer une autre matrice de rotation dont l'angle est la somme des deux angles, mais ça n'est pas toujours aussi simple que ça.

Figurez-vous qu'il existe des matrices de translation, de rotation, de scaling, de projection, etc, etc. Si vous devez faire une suite d'opérations de ce genre à un certain nombre de points, vous commencer par calculer la matrice de transformation qui est la composition des opérations élémentaires et vous appliquez cette matrice à vos points. Le gain est énorme.

1.2. Les coordonnées normalisées

Si vous vous demander à quoi peut bien correspondre une matrice de translation en 2D, et que vous chercher une matrice 2x2 comme pour la matrice de rotation, vous allez chercher pour rien, elle n'existe pas. Pour la trouver, il va falloir introduire un nouveau concept : les coordonées normalisées.

Vous savez que dans un plan, un point se représente par deux valeurs, un x et un y. Voici donc un nouveau concept : moi je dis qu'à présent, un point se représentera par trois valeurs : un x, un y et un t.

Là vous vous dites, ça y est, le Plouf a perdu les pédales, dans un plan il n'y a que deux dimensions et là il nous tape trois variables, comme dans l'espace, c'est complètement louf.

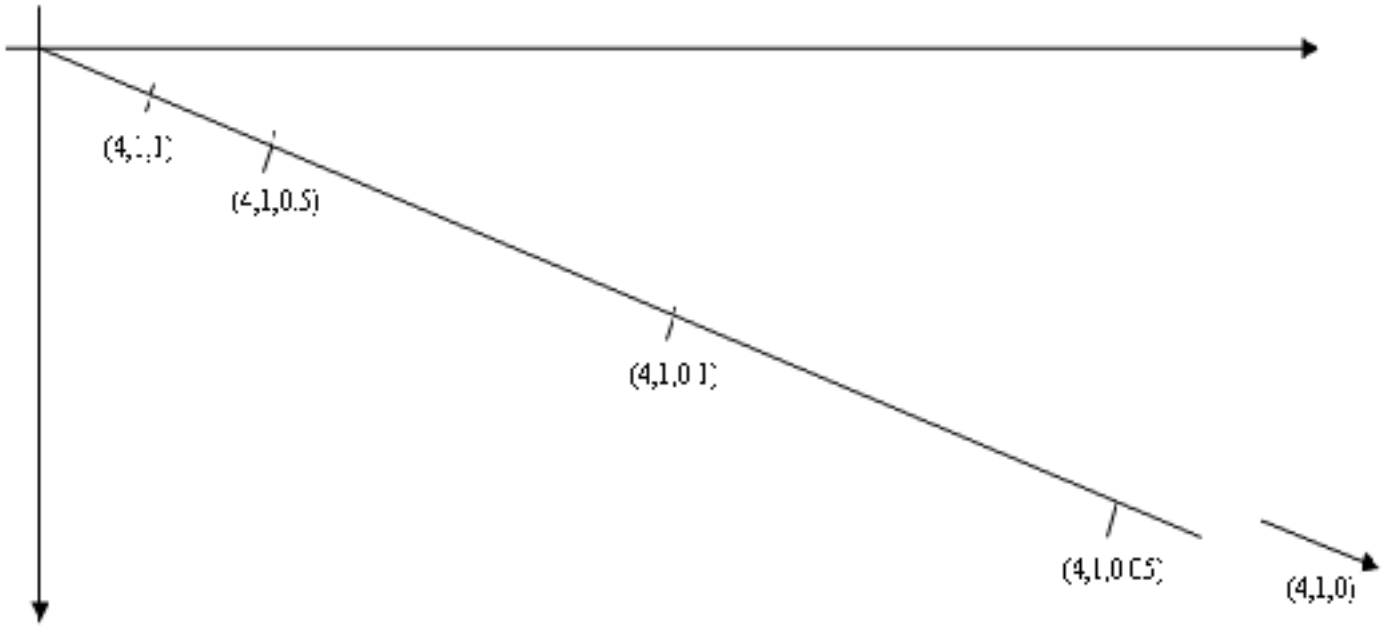
Et vous aurez raisons, sauf si je vous dit que le point (1,2,1) correspond au point (2,4,2). En fait, la formule pour passer d'une représentation à une autre est :

```
old_x=x/t
old_y=y/t
```

Bref, si on multiplie tout par 2, on est toujours au même endroit. Jusque là, cette nouvelle invention semble donc totalement inutile. Sauf si je vous demande (pour rigoler) où se trouve le point (1,4,0).

Heu, en (infini,infini) ? Wais, mais c'est pas très précis, ça, non ? Parce que (4,1,0) et (1,4,0) ça doit sûrement être deux choses différentes alors que dans le « vieux » système c'est tout les deux le « point » (infini,infini). Me voilà capable de distinguer des choses qui étaient indistinguables avant. Bien joué mais ça sert à quoi ? C'est quoi finalement, ce point (1,4,0) ?

Vous allez comprendre, imaginez le point (1,4,1), bon c'est facile, non ? Ensuite prenez le point (1,4,0.1). Ha, il est un peu plus loin (en 10,40). Ensuite le point (1,4,0.001), il est loin, lui, en (1000,4000), etc... Et bien tous ces points sont alignés sur une droite de pente 4. Le point (1,4,0), c'est juste le « dernier » point de la droite. Un point qui est à l'infini, certes, mais pas n'importe où à l'infini : à l'infini sur une droite donnée. Bref, ce point, c'est une direction, une direction qui est la direction de la droite de pente 4. De même que le point (4,1,0) c'est le point qui représente la direction de la droite de pente un quart.



L'avantage du système ? On ne fait plus de différence entre un point et une direction, c'est devenu deux concepts identiques, qui s'expriment exactement de la même façon. Mais à quoi ça sert ? Vous ne voyez pas ?

Si je suis en un point et que je regarde dans la direction de pente 4 (wé, en fait je regarde le point 1,4,0), quelle direction je regarde si je tourne mon regarde d'un angle a ?

Il suffit de faire subir une rotation d'angle -a au point (1,4,0), grâce à quoi ? A une matrice de rotation, bingo ! Voilà comment on peut faire tourner des lumières qui sont à l'infini dans les moteurs 3D...

La matrice de rotation dans ces nouvelles coordonées s'exprime comme ceci :

$$\begin{pmatrix} \mathrm{Cos}(a) & -\mathrm{Sin}(a) & 0 \\ \mathrm{Sin}(a) & \mathrm{Cos}(a) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Ben wé, c'est une matrice 3x3, puisque nos points sont à présent des matrices colonnes de trois lignes. (3x1)

Et, je vous le donne dans le mille, la matrice de translation, c'est

$$\begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & dt \end{pmatrix}$$

Essayez, vous verrez que ça marche. A présent savoir pourquoi ça se met à fonctionner en 3x3 alors que ça ne marchait pas en 2x2, j'ai pas envie ici de faire des tats de calculs existentiels, je vous demande d'admettre que c'est comme ça et pas autrement.

A présent, comment on fait pour faire tourner un point autour d'un autre point quelconque ? Et bien on commence par translater le centre de rotation à l'origine, ensuite on applique la rotation, et puis on translate le résultat pour replacer le centre au bon endroit. Il s'agit de trois opérations consécutives, qui peuvent s'exprimer en une seule, grâce à la combinaison des opérations avec le produit matriciel. C'est-y pas beau ? Sans compter qu'on peut même translater un point « à l'infini » d'une distance infinie pour le ramener à l'origine, et vice-versa, il suffit de mettre dt=0. (ce qui arrive rarement, j'en conviens, en général on laisse dt à 1).

Dans toutes ces matrices, il y en a une qui est amusante, c'est justement la matrice Identité, c'est la matrice qui ne fait rien :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Si vous transformez un point avec cette matrice, vous allez obtenir le point lui-même.

La notion d'inverse de matrice prend tout son sens, vu que la composition d'une matrice avec son inverse donne l'identité, on comprend que si une matrice fait une transformation, la matrice inverse fait l'opération inverse. Finalement, l'inverse d'une matrice de rotation d'angle a, c'est une matrice de rotation d'angle -a. De même pour la translation vous pouvez vérifier vous-même que

$$\begin{pmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & dt \end{pmatrix} * \begin{pmatrix} 1 & 0 & -dx \\ 0 & 1 & -dy \\ 0 & 0 & -dt \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Le fait que le produit matriciel n'est pas commutatif prend également tout son sens, il n'est en effet pas du tout équivalent de faire la rotation avant la translation ou la translation avant la rotation. Faites un dessin, vous comprendrez tout de suite pourquoi. Il faut donc toujours bien se méfier et vérifier que l'on compose les matrices dans le bon ordre.

On peut donc construire la matrice qui tourne un point x (x,y,t) autour d'un centre (cx,cy,ct) d'un angle a :

$$R = \begin{pmatrix} 1 & 0 & cx \\ 0 & 1 & cy \\ 0 & 0 & ct \end{pmatrix} * \begin{pmatrix} Cos(a) & -Sin(a) & 0 \\ Sin(a) & Cos(a) & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -cx \\ 0 & 1 & -cy \\ 0 & 0 & -ct \end{pmatrix}$$

Avec x'=R*x

Remarquez que le produit se fait en sens inverse, on commence par les dernières opérations.

Tout ce que je viens de raconter est immédiatement adaptable en 3D. Au lieu de jouer avec des (x,y,z), on joue avec des (x,y,z,t), les matrices de transformation sont des matrices (4x4) et vous pouvez facilement trouver dans la littérature spécialisée (la doc d'opengl par exemple) quelles sont les matrices de transformation intéressantes.

Bien, à présent que je vous ai convaincu de l'utilité des matrices, il ne vous reste plus qu'à trouver un tutoriel sérieux là dessus, qui ira surement beaucoup plus en profondeur que moi. Les matrices, c'est un sujet presque inépuisable, et qui ne sert pas qu'en géométrie (en fait ça sert partout).

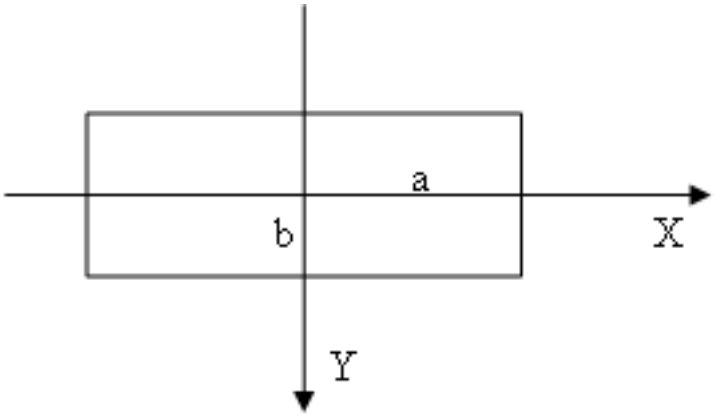
Finalement, la première chose à faire quand on se met à programmer un moteur 3D, c'est de programmer une classe Matrix...

1.3. Notion de base

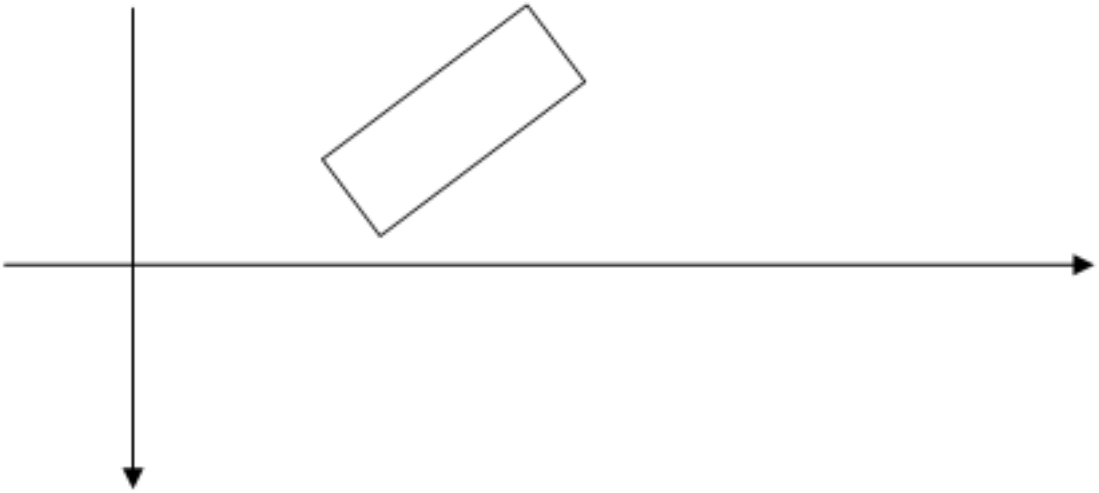
Pas à prendre dans le sens « notions basiques » mais bien dans le sens « qu'est-ce qu'une base ? »

Commençons comme toujours en 2D avec un petit problème assez simple. Nous définissons un

rectangle comme ceci :



Si je vous demande à quel endroit se trouve le coin supérieur gauche, vous pouvez tout de suite me répondre, il se trouve en (-a,-b). Maintenant plaçons ce rectangle comme ceci :



Même si je vous donne la position du nouveau centre et de l'angle de rotation, vous ne pouvez pas immédiatement me donner la position du coin supérieur gauche du rectangle. Il faut faire un calcul.

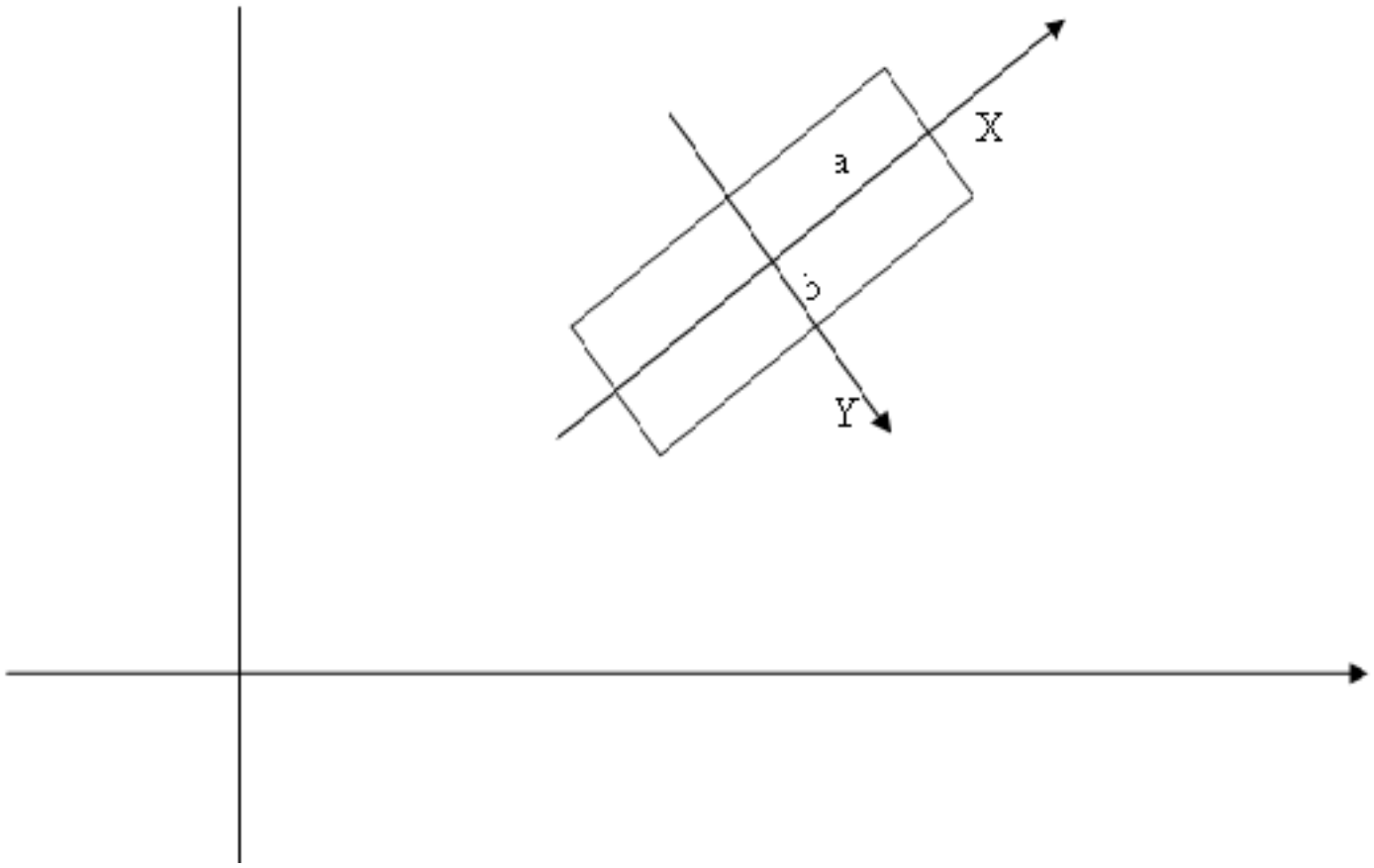
En fait qu'allez-vous faire ? Une méthode classique, c'est de prendre ce point (-a,-b), de lui appliquer la même rotation suivie de la même translation que celles appliquées au rectangle. Vous obtiendrez en effet la position du point.

Mais il existe une autre manière de voir les choses (qui revient, bien sûr, au même), c'est de dire

que la position du point (-a,-b) original se trouve au point... (-a,-b), après tout, c'est toujous le coin supérieur gauche et il n'y a pas de raison que le coin supérieur gauche se trouve ailleurs qu'en (-a,-b). Et vous aurez parfaitement raison, de la même manière que quand l'on vous demande à quelle hauteur vous vous trouvez, vous donnerez sûrement la hauteur de la chaise sur laquelle vous êtes assis(e), sans prendre en compte la hauteur du bâtiment, ni la hauteur de la colline sur laquelle se trouve ce bâtiment, etc...

L'important, c'est de savoir qu'il s'agit de la hauteur par rapport au plancher. Si on est capable de donner la hauteur du plancher par rapport au sol, et la hauteur du sol par rapport au niveau de la mer, on saura bien donner votre hauteur par rapport au niveau de la mer. Et bien ici c'est la même chose, le point se trouve en (-a,-b) par rapport... au premier schéma, pas du tout par rapport au second. La seule question à se poser, c'est : « comment passer du premier schémas au second ? ».

La réponse est évidente, il faut faire une translation, suivie d'une rotation, ça revient au même. En réalité, la bonne manière de représenter la seconde figure est celle-ci :



Et donc, si on vous demande la position du coin supérieur gauche, vous répondrez deux choses : (-a,-b), et surtout comment passer du premier repère d'axe au second. Et comme pour passer de

l'un à l'autre, il s'agit d'une combinaison d'opération, devinez ce qu'on va donner pour décrire le passage d'un repère (appelé base) à une autre ? Une matrice, bien entendu...

Finalement, une coordonée ne représente rien si elle n'est pas accompagnée d'un renseignement qui dit de quelle base on parle.

A quoi ça sert me direz-vous ? C'est de toutes façons évident, tout ça... Oui ?

Voyons si vous avez compris alors : moi je vous donne un point quelconque, (x,y), se trouvant dans la « grande » base, pas celle de travers qui est celle du rectangle. Ce que je vous demande, c'est de me dire si ce point (x,y) se trouve dans le rectangle ou pas. Facile ?

Ce problème n'est à prioris pas facile, il faut regarder les équations des droites des côtés du rectangle, etc, etc, etc, bonne chance...

Alors qu'il suffirait de se demander : « quelles sont les coordonées de ce point dans la base du rectangle ? » Si on connaît ça, le problème devient très bête, car ça revient à savoir si un point se trouve dans un rectangle qui est parrallelle aux axes. Ce qui se code facilement par « si x est compris entre -a et a et y compris entre -b et b ».

Et comment connaître la coordonée de (x,y) dans la base du rectangle ? Ben pardis, vous pouvez faire l'autre opération inverse grâce à cette fameuse matrice qui représente la position du rectangle dans la grande base, faites donc l'opération inverse ! Et hop, un petit calcul d'inversion de matrice, un petit produit matriciel, et quelques petits tests idiots, et le tour est joué !

Moralité : quand un problème se pose, demandez-vous toujours si il n'est pas plus simple de le résoudre dans une base appropriée. Vous verrez un peu plus tard qu'il s'agit d'une des bases (c'est le cas de le dire, hihi) de tout moteur 3D. Vous verrez que tout objet possède une base « locale », dont la position est exprimée par rapport à une base « globale », commune à tous les objets. Mais je m'avance...

1.4. Notion d'espace

Un espace, ça n'est jamais qu'une base, en gros c'est équivalant. Quand on parle d'un point, on doit préciser sa base, et donc finalement dans quel espace il se situe. Votre hauteur, celle finalement de la chaise, se trouve dans l'espace « pièce », tandis que la hauteur du plancher se trouve dans l'espace « maison », et ainsi de suite.

Dans un moteur 3D, on peut dire qu'il y a 4 espaces, ou plutôt 4 types d'espace. J'y arrive.

Depuis le chapitre précédent, vous pouvez imaginer que chaque objet (3D ou pas) est défini dans sa base propre. Un rectangle (cube, ...) est défini par une paire (a,b), un cercle par un rayon, etc... On suppose donc que le rectangle est centré dans sa base, de même que le cercle. On peut même tout normaliser, en disant qu'un cercle est finalemement toujours de rayon 1. Il s'agit en fait de l'objet abstrait, qu'il faudra « placer » dans un espace commun, là où se trouvent rassemblés tous les objets, la scène où se déroule l'action.

Il faudra donc, pour chaque objet, connaître une matrice (une base, un repère, etc...) qui permette de passer de la base de l'objet (« l'espace objet ») à la base globale (« l'espace global ». Mais cela suffit-il ? Pour décrire la position de chaque objet dans l'espace, certainement, mais pas pour les afficher à l'écran. Il faut connaître un renseignement en plus.

Car où se trouve la caméra ? En général dans les moteurs simples, la caméra se trouve à l'origine et regarde dans la direction des Z, alors que l'axe des X est horizontal et l'axe Y vertical. Evidement, ça n'est pas très souple...

Le problème, c'est que les formules qui permettent de projeter un point à l'écran sont des formules qui supposent que la caméra se trouve à l'origine, regarde l'axe des Z etc...

Alors que faire ? Le plus simplement du monde, il faut déplacer tous les objets de sorte que ceux qui sont devant la caméra et dans l'axe de regard de cette caméra se retrouvent placés sur l'axe des Z, et ainsi de suite. Finalement, il faut faire un nouveau changement de base, pour se déplacer dans « l'espace caméra ». Quelle est la matrice qui correspond à ce changement de base ? Tout simplement la matrice qui explique comment la caméra est située par rapport à sa position « de base », et donc quelle est la position relative de la caméra par rapport à l'origine de l'espace global, quelle est son orientation, etc... Vous voulez faire tourner la caméra? Multipliez sa matrice par une matrice de rotation...

Le dernier espace est finalement l'écran lui-même, espace en deux dimensions, c'est l'étape ultime, ce que le joueur ou l'utilisateur verra. C'est également une base, et il existe une matrice de transformation qui permet de passer de l'espace caméra à l'espace écran. Il s'agit en fait de la représentation matricielle des formules classiques :

```
u=x/z
v=y/z
```

- transformer ce point avec la matrice de base de l'objet, pour connaître sa position dans l'espace « global »
- transformer cette position globale avec la matrice de base de la caméra, de manière à connaître sa position dans l'espace « caméra »
- transformer cette position caméra avec la matrice de projection, de manière à connaître sa position dans l'espace « écran »

Finalement, si j'ai un point d'un objet, dans l'espace objet, mettons ce fichu coin supérieur gauche, et que je veux connaître sa position à l'écran, je dois faire les opérations suivantes :

Trois transformations à la suite ? Non ! Une seule ! Je vous rappelle que l'on peut une bonne fois pour toute combiner toutes ces transformations au moyen du produit matriciel. Cette combinaison doit être faite à chaque image (car d'une image à une autre, la position de l'objet ou de la caméra aura sûrement changé), mais pas à chaque point. Pour chaque point, on fera UN et UN SEUL produit matriciel. Le bonheur, ça trace...

Enfin, tout ça c'est la théorie...

Mais ça vous permet à présent de comprendre l'origine des trois matrices que l'on peut modifier dans OpenGl, la matrice de l'objet « model matrix », la matrice caméra « camera matrix » et la matrice de projection « projection matrix ». La théorie n'est donc pas si éloignée que cela de la pratique...

Le calcul des lumières

Tant qu'à faire des maths, autant régler une bonne fois pour toute le problème du calcul des lumières.

En général on considère que l'intensité résultant de la rencontre d'un point avec un faisceau lumineux est calculée en fonction de l'angle que fait la direction du faisceau avec la normale au point.

Je vais aborder ici deux types de lampes : les ambiances et les lampes ponctuelles. Avec ça vous avez déjà largement de quoi vous amuser et en général les moteurs classiques n'utilisent pas autre chose.

La lampe de type ambiance est la plus simple que l'on puisse imaginer : quelle que soit la position de la lampe, quelle que soit la position du point, et quelle que soit la direction entre la lampe et le point, le résultat est toujours le même : on augmente ou on diminue la luminosité du point, c'est tout. Bref cette lampe porte bien son nom, c'est l'ambiance lumineuse de la scene.

On peut faire des ambiances colorées, préciser que l'ambiance est rouge par exemple, ce qui reviendra à augmenter la valeur de rouge de la couleur du point sans toucher ni au vert, ni au bleu.

Un calcul classique c'est:

```
Rouge=Rouge+AmbianceRouge;
Vert=Vert+AmbianceVert;
Bleu=Bleu+AmbianceBleu;
```

Et puis on fait des tests pour vérifier que l'on a pas dépassé la valeur maximale (ou minimale, les ambiances négatives ça existe).

La lampe de type ponctuel est un peu plus intéressante. En fait elle aussi possède une couleur, mais la couleur n'est pas rajoutée telle quelle au point, comme pour l'ambiance, on la rajoute, mais

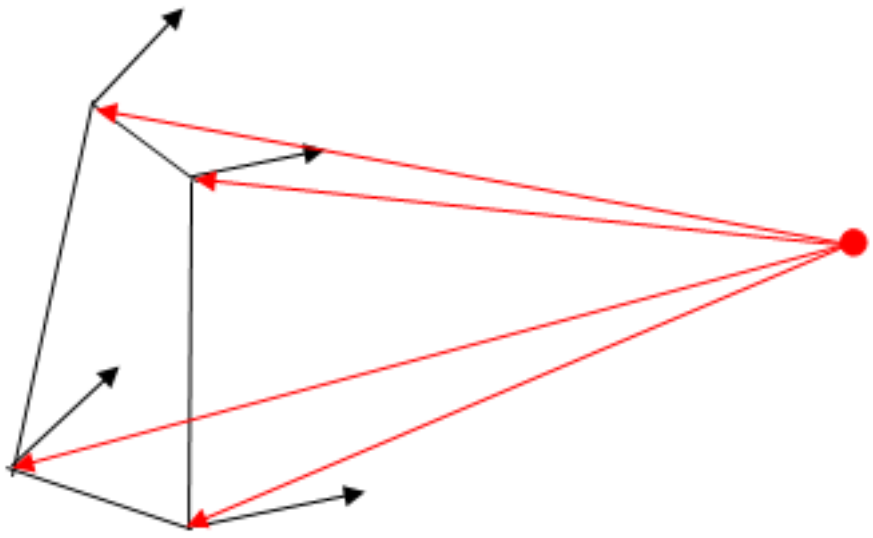
multipliée par un facteur k, inférieur à 1 :

```
Rouge=Rouge+k*LampeRouge;
Vert=Vert+k*LampeVert;
Bleu=Bleu+k*LampeBleu;
```

Comment calculer k ? C'est simple, il suffit de prendre pour valeur le produit scalaire de la direction de la lampe avec la normale du point, et de changer le signe.

```
k= -(normalx*dirx+normaly*diry+normalz*dirz) ;
```

k est calculé en faisant le produit scalaire entre la normale au point (la flèche noire) et la direction de la lumière (la flèche rouge).



normalx,y et z sont des informations que le point doit connaître, tandis que dirx,y et z doit être calculé en fonction du point et de la position de la lampe. C'est un calcul qu'on peut faire comme suit :

```
distance=racine((pointx-lampex)²+ (pointy-lampey)²+ (pointz-lampez)²);
dirx=(pointx-lampex)/distance;
diry=(pointy-lampey)/distance;
dirz=(pointz-lampez)/distance;
```

Vous constaterez que ce calcul est un rien lent, il demande une racine carrée et des tats de calculs souvents en flottants, c'est pourquoi bien souvent on ne considère que des lampes se trouvant à distance infinie. L'avantage ? La direction ne dépend plus de la position du point, il s'agit d'un renseignement intrinsèque à la lampe.

A ces resultats on peut éventuellement rajouter une valeur si le matériau possède une « couleur d'ambiance », et des trucs du genre, à vous de voir comment vous programmez vos matériaux.

Il existe bien sûr des tas de « trucs » pour augmenter la vitesse de ces calculs, et il existe également des tas d'autres type de lampes, à vous de les découvrir =).

Au final, chaque sommet de la face pourra avoir une intensité de couleur différente, ce qui nous permettra d'avoir la possibilité de faire un beau dégradé. Un moteur plus simple, lui, se contentera d'un calcul de normale par face, ce qui rendra la face unie. Une sphère éclairée donnera alors un effet « boule à facettes », alors que dans notre cas le dégradé sera continu.

Les objets

Ok, bon pour ce qui est math, on va momentanément en rester là, passons à des choses un peu plus concrètes. Une question qui revient souvent c'est : comment représenter un objet 3D en mémoire, de quoi est-il constitué (de faces ? de vectrices ? de trucs ou de machins ?), comment est-il stocké, dans quels type de structures etc...

La question à se poser est de savoir quelles informations doit véhiculer un objet. Il ne doit pas dire où il est (ce problème étant celui de sa matrice de base), mais bien ce qu'il est. Il s'agit donc, pour tout ce qui sera renseignements géométriques, d'informations définies dans l'espace objet. (Comme pour notre rectangle, défini par a et b)

Un objet est en général défini par un ensemble de faces, elles-mêmes constituées d'un ensemble de triangles. On peut sans trop de perte de généralité résumer qu'un objet est un ensemble de triangles. Bien sûr il n'y a pas que des objets « triangularisés » en 3D, je veux parler de tout ce qui est courbes (Quake3 rulez J) et objets que le moteur « comprend », mais bon, ça sort un peu du cadre de ce texte alors on va gentiement laisser tomber.

Vu qu'on sait que notre objet est un ensemble de triangle, il faut à présent savoir ce qu'est un triangle. Vous savez sûrement qu'un triangle est défini par trois points. On parlera de trois vectrices (Vertexes, Meshes). Il s'agit de points dans l'espace (objet) sur lesquels sont « posés » les triangles.

Il est conseillé d'avoir un ensemble de vectrices séparé de l'ensemble des faces (c'est à dire que chaque face pointe vers trois vectrice, sans les contenir vraiment), car la plupart du temps une vectrice est utilisée par plus d'un triangle. Ca serait idiot d'avoir deux vectrices en mémoire qui représentent exactement le même point.

Une vectrice, finalement, c'est un quadruplet (x,y,z,t), point final.

- - La normale en chaque sommet du triangle, pour des calculs de lumières.
- - Le matériau de cette face.
- - Si le matériau de cette face possède une texture, alors il faut que chaque sommet possède une coordonée de texture.

Mais ça n'est pas tout, pour définir totalement une face (un triangle), on a besoin de quelques renseignements en plus :

Avec ça en mémoire, vous avez de quoi afficher votre objet, si du moins vous savez comment exploiter ces renseignements.

Un premier pipeline

Un pipeline c'est la suite des opérations qui sont faites pour obtenir le résultat à l'écran. C'est un terme souvent utilisé.

1 - Trouver quelle est la nouvelle position de l'objet dans l'espace global. Egalement trouver la position de la caméra dans ce même espace.
2 - Calculer la matrice qui permet de passer de l'espace objet à l'espace écran.
3 - Pour chaque vectrice de l'objet, faire l'opération matricielle pour trouver sa place sur l'écran.
4 - Passer en revue les faces et dessiner les triangles à l'écran grâce aux positions des vectrices dans l'espace écran calculées au point 3.

Plutôt simple non ? Et on recommence ça pour chaque objet.

Mais nulle part on ne parle de lampes ici, d'ailleurs. C'est curieux non ?

Aie aie aie, c'est la première fois que la belle théorie va devoir être bousillée J

Il se fait que ce qu'on fait au point 3 est très rapide, mais il est parfois utile de connaître la position des vectrices dans l'espace global, alors le système « toutes les transformations en une fois » ne nous donne justement que le résultat final!

Ici, on aurait bien besoin de connaître la position des vectrices dans l'espace global ainsi que de la direction de chacune des normales dans ce même espace global pour pouvoir calculer correctement les lumières (vraiment ? Exercice donné au lecteur : trouver une autre solution).

De plus, beaucoups d'algorithmes de traçage de triangles à l'écran ont besoin de connaître la coordonée Z du point dans l'espace caméra (Z-Buffer, correction de perspective, on verra ça plus tard), et d'autres algos ont besoin de connaître la coordonée complète (Clipping 3D, encore pour plus tard) ! Bien joué, on va quand même devoir se taper plusieurs opérations matricielles.

Que voulez-vous, c'est la vie, et c'est à vous de trouver comment on peut s'en passer au maximum (je vais pas tout vous dire non plus, et puis quoi encore =))

Un pileline un peut plus complet serait donc :

1 - Trouver quelle est la nouvelle position de l’objet dans l’espace global. Egalement trouver la position de la caméra dans ce même espace.

2 - Calculer la position des vectrices ainsi que la direction des normales dans l’espace global grâce à la matrice de base de l’objet.

3 - Procéder au calcul des lumières pour trouver la couleur des points.

4 - Utiliser la matrice caméra pour trouver la position de chacun des points dans l’espace caméra.

5 - Utiliser la matrice de projection pour trouver la position de chacun des points dans l’espace écran.

6 - Passer en revue les faces et dessiner les triangles à l’écran.

La projection

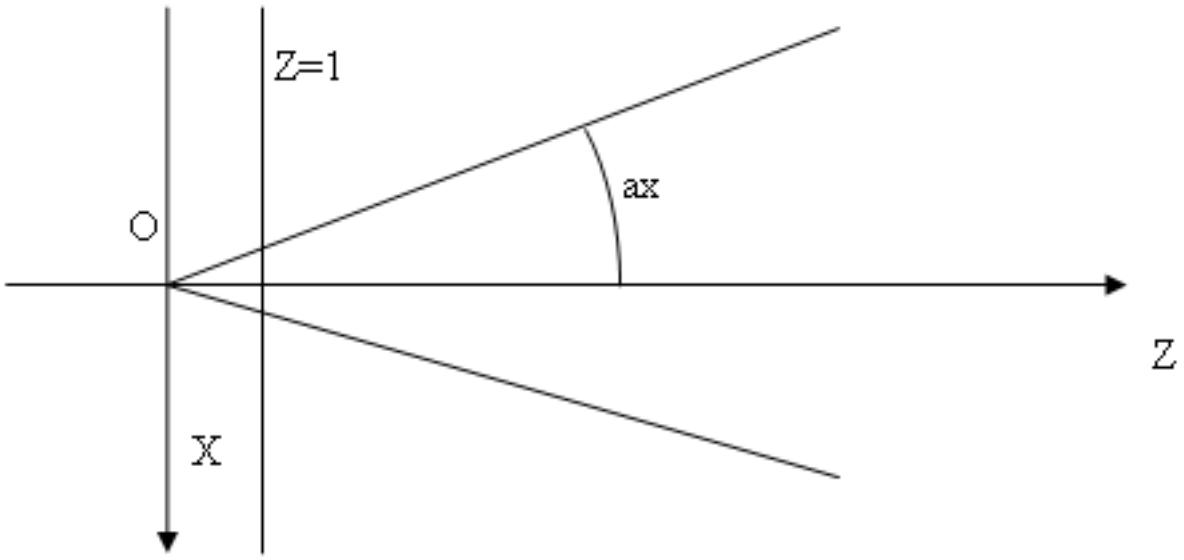
Ceux qui ne s'intéressent pas aux maths peuvent très bien pomper le résultat final.

La question est : nous avons des points dans l’espace, quelle est leur position à l’écran ? Question pertinente, on affiche sur un écran, et pas dans l’espace. Nous devons passer d’un espace à trois dimensions vers un espace à deux dimensions. Une formule souvent utilisée (il en existe d’autres) est la formule suivante :

```
u=factor_x*x/z +w/2
v=factor_y*y/z +h/2
```

Avec (u,v) la coordonnée sur l’écran, et (x,y,z) la coordonnée dans l’espace. On a également w et h qui sont la longueur et la largeur en pixel de l’image projetée (souvent tout l’écran, mais ça pourrait n’être qu’une fenêtre).

Cette formule suppose que l’observateur se trouve en (0,0,0), que le plan de projection (celui où se trouve l’écran, finalement) est Z=1, que c’est écran est centré autour de l’axe des Z, et que l’observateur regarde dans la direction de l’axe des Z. Vu du haut, ça donne :



Le but ici, c'est de trouver ce que valent ces factor_x et factor_y. On va travailler avec une méthode appelée « l'identification » : on sait à quel résultat on doit arriver, et on va faire en sorte que factor_x et factor_y nous amènent effectivement à ce résultat.

Quel est le résultat auquel nous voulons arriver ? Nous voulons qu'un point de l'espace qui se trouve sur un plan qui correspond aux angles d'ouverture de la caméra (et donc aux limites de visibilité) se trouve sur le bord de l'écran. Précisons donc nos hypothèses :

Il y a quatre plans qui permettent de délimiter la zone visible, deux plans verticaux et deux plans horizontaux, on suppose que les plans verticaux forment un angle 2ax entre eux et font un angle ax avec l'axe des Z.

Les deux plans horizontaux font un angle 2ay entre eux, et font un angle ay avec l'axe des Z.

Le plan de projection est Z=1. On pourrait tout refaire en considérant cette valeur comme paramétrique, mais ça n'apporte fondamentalement pas grand chose si ce n'est des problèmes dans les algorithmes de correction de perspective...

On appelle souvent « ratio » le rapport tan(ax)/tan(ay). Il est fréquent qu'au lieu de configurer le viewport avec un angle horizontal et un angle vertical, on demande un angle horizontal et un ratio. Un bon ratio est évidemment un ratio qui fait en sorte que l'écran sur lequel vous projetez à le même rapport de dimensions que la taille (en mètres, sisi) de l'image finale que vous, utilisateur, verrez sur l'écran. Bref, si vous affichez en plein écran avec une résolution de 640x480 et que vous supposez qu'un pixel est carré (ce qui est vrai en 640x480, mais pas en 640x400), le ratio est de 640/480=4/3. En pratique si on suppose toujours qu'un pixel est à peu près carré, le ratio se calcule comme le rapport de la longueur par la largeur de l'image affichée. Une résolution d'écran a des pixels carrés si les proportions de l'image sont égales aux proportions du nombre de pixel. Un écran est normalement en proportions 4/3, donc il faut travailler dans des résolutions qui respectent ce rapport : (320x240), (400x300), (512x384), (640x480), (800x600), (1024x768), (1152x864), (1600x1200), mais PAS (320x200), (640x400) , (1280x1024), et autres bizarreries.

Bref : ratio=w/h (sinon on va encore dire que je complique tout)

Un point se trouvant à l'intersection du plan vertical de droite avec le plan horizontal du bas est (tan(ax),tan(ay),1). On veut que sa projection donne (w,h), c'est à dire le coin inférieur droit de l'écran.

Il faut donc que:

```
w=factor_x*tan(ax)+w/2
h=factor_y*tan(ay)+h/2
```

Donc que:

```
w/(2*tan(ax))=factor_x
h/(2*tan(ay))=factor_y
```

Et c’est gagné. Comme on voit que factor_x et factor_y ne dépendent que de la résolution et des angles du viewport, il suffit de les calculer une seule fois au moment où ces valeurs changent, et basta.

En résumé :

```
u=factor_x*x/z +w/2
v=factor_y*y/z +h/2
```

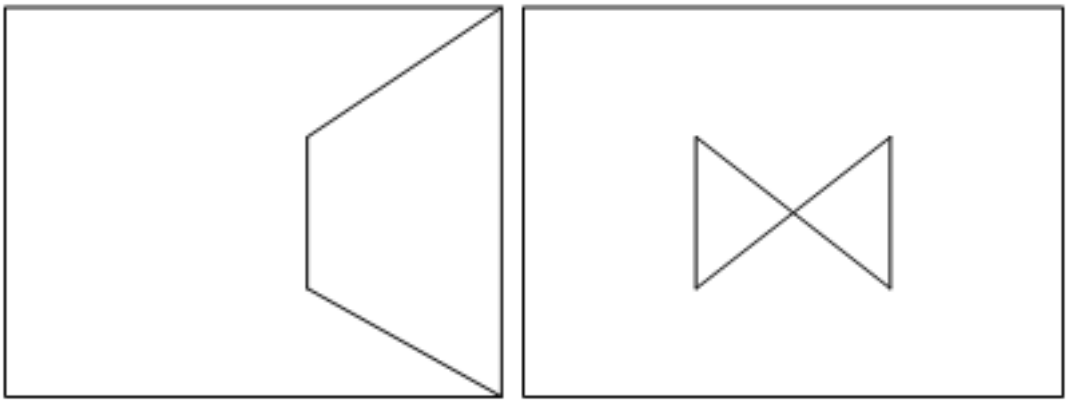
avec

```
factor_x=w/(2*tan(ax))
factor_y=h/(2*tan(ay))
```

On voit que plus ax et ay sont petits, et plus le facteur est grand, et donc plus l’image projetée sera grande. C’est normal, vu que diminuer l’angle d’ouverture d’une caméra, c’est justement la définition du zoom.

Le clipping 3D

Ceux qui ont déjà tenté de faire un moteur 3D sont sans aucun doute tous tombés sur le même problème : si une face commence devant nous et termine derrière nous (par exemple il s'agit d'un mur dans une pièce ou n'importe quoi du genre), on observe le résultat amusant suivant :



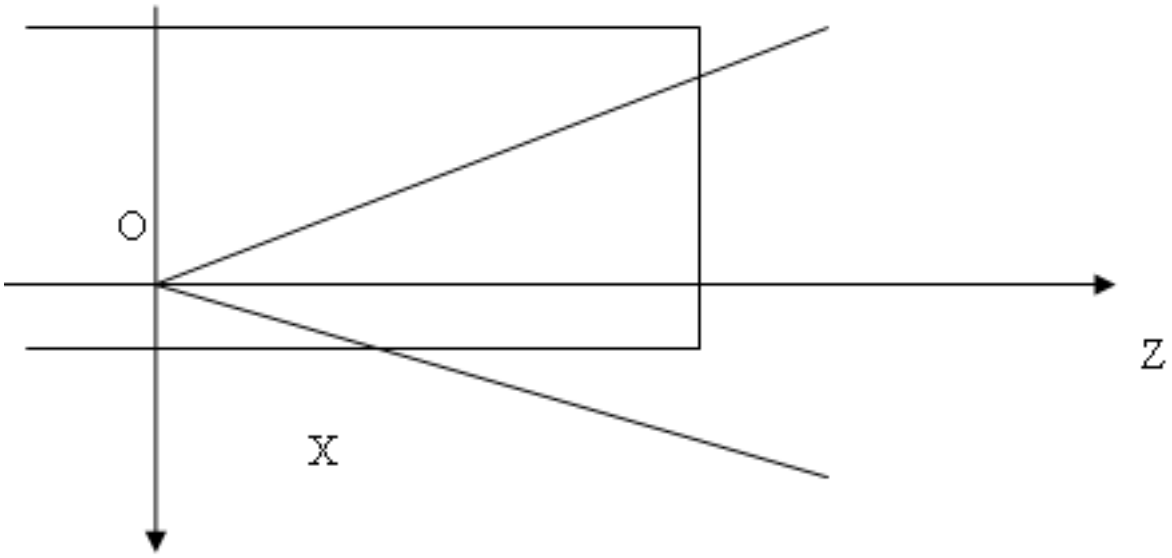
A gauche vous avez le résultat naïvement espéré, à droit vous avez le résultat obtenu (quand ce n'est pas pire...). Que s'est-il passé ? C'est simple. La formule de projection ressemble à celle-ci (voir la projection) :

```
u=factor_x*x/z
v=factor_y*y/z
```

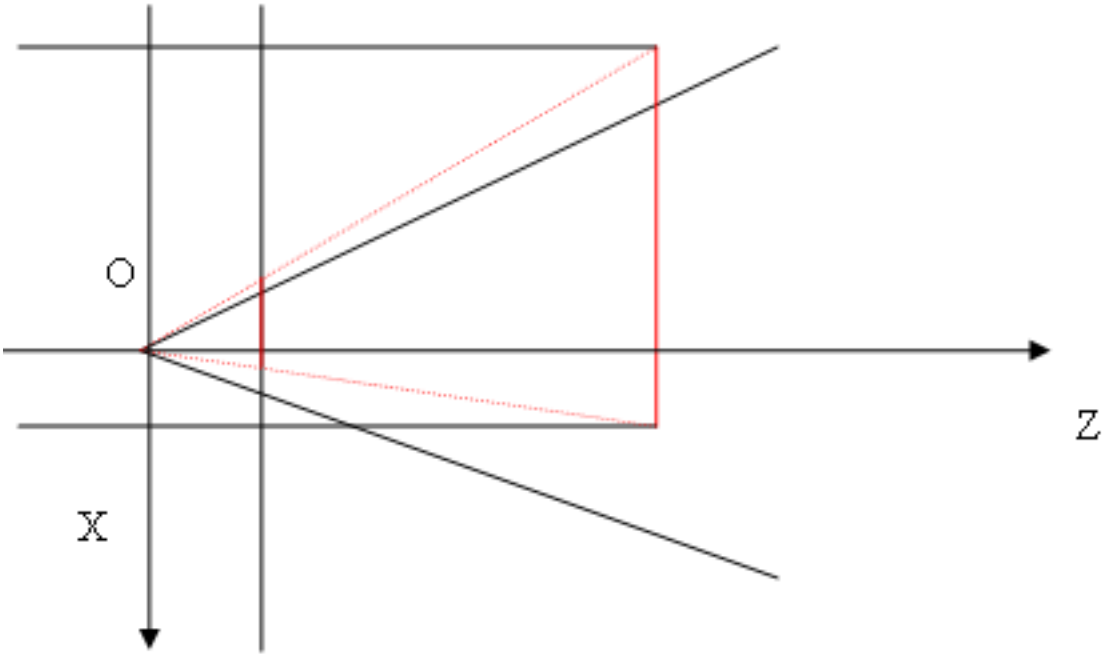
Vous remarquez que si z est négatif, la formule ne se tracasse pas et on obtient un u négatif, quand bien même x était positif. On a une sorte de « retournement » qui provient du fait que la formule ne fait pas la différence entre un point « devant » et un point derrière.

Il n'existe pas de solution simple contre ce problème. Tous les raisonements que j'ai tenté en 2D pour éviter d'avoir ce résultat ont été inutiles. La seule solution élégante provient du clipping 3D.

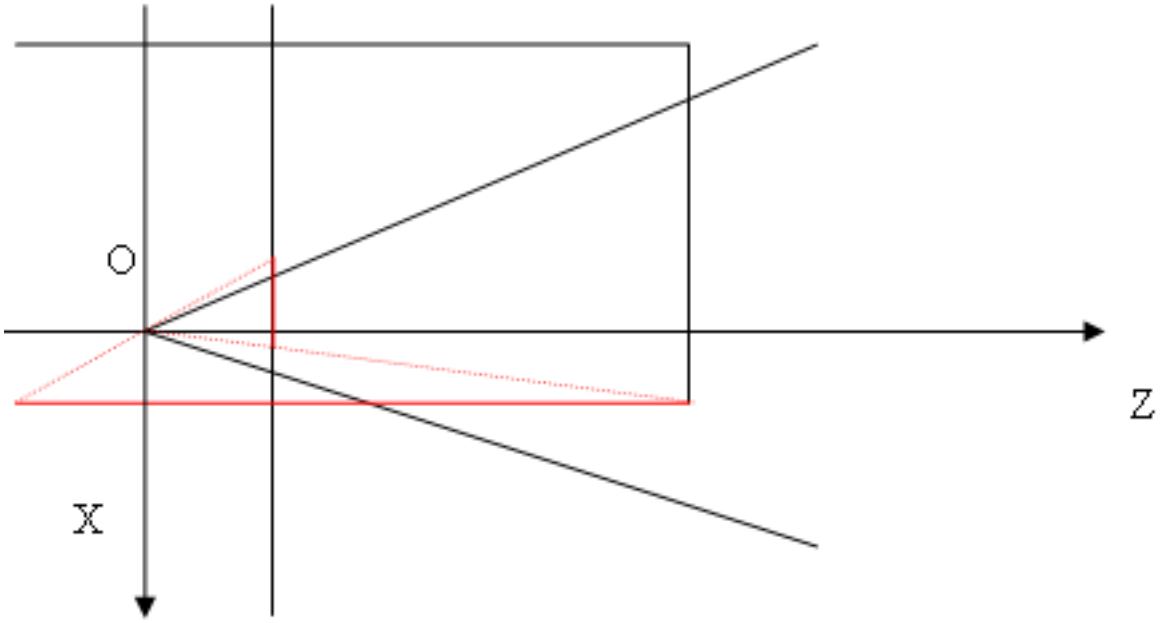
Tout d'abord examinon un peu ce qu'en général on appelle le « viewport » ou un truc du genre : vu du haut, voici ce que donne l'espace caméra quand on se balade dans un couloir :



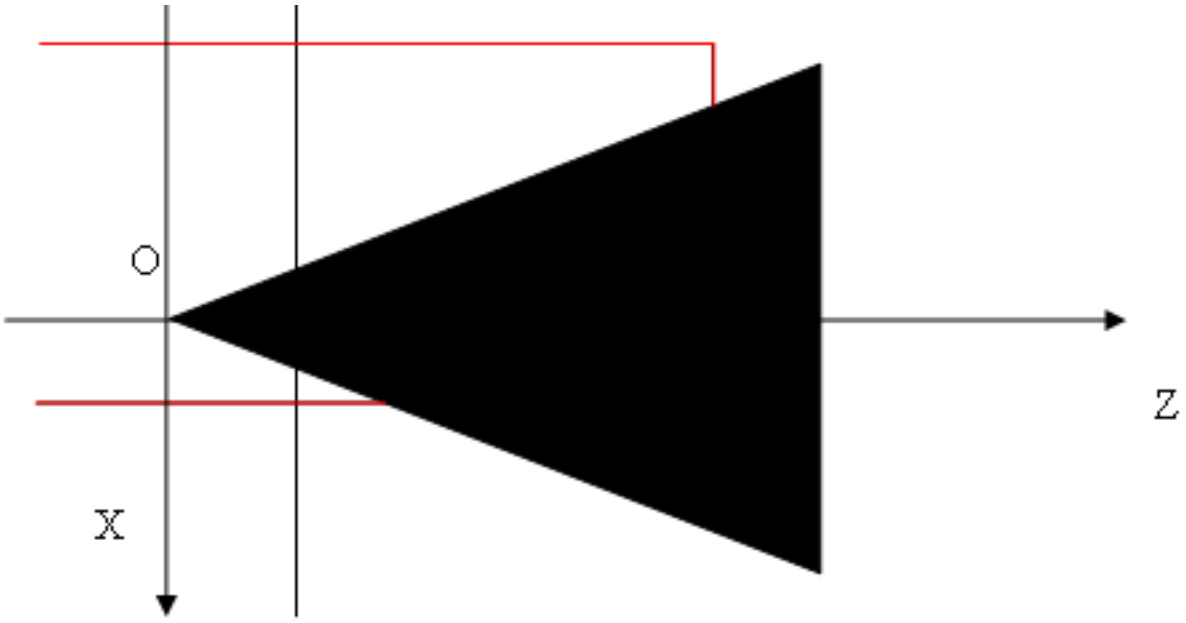
Par exemple, essayons de voir ce que va donner à l'écran la projection du mur de fond :



Regardons à présent ce que donne la projection du mur de droite :



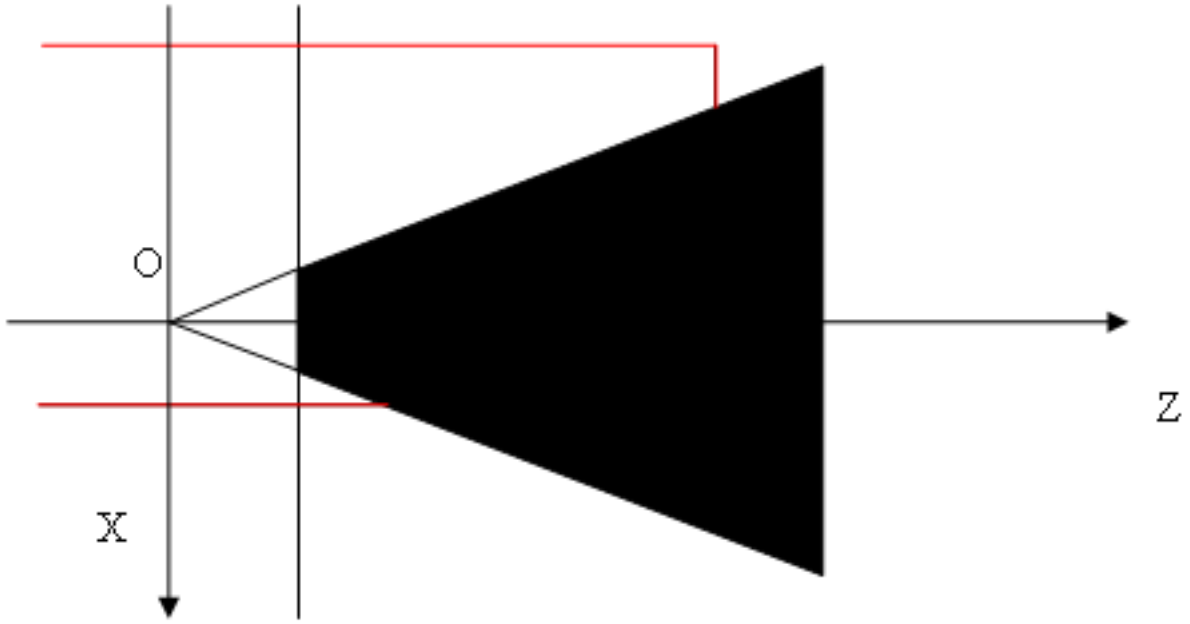
Une solution existe néanmoins : c'est de couper tout ce qui sort du « viewport », c'est à dire de la zone hachurée sur le schémas qui suit, et donc la zone dans laquelle se trouve tout ce qui est visible après la projection :



Regardons un peu ça de plus près. Tout ce qu'on a fait jusqu'à présent supposait qu'on regardait tout d'en haut. Ce qui nous fait dire que les droites par rapports auxquelles ont doit couper les faces

sont en fait des plans verticaux. Il en va de même si on regardait de côté, on aurait les plans qui correspondent aux limites supérieure et inférieure de l'écran.

Par sécurité, on rajoute souvent un dernier plan, un plan qui se trouve en Z=1, et qui transforme finalement notre viewport de la manière suivante :



Vous vous souvenez de l'équation d'un plan dans l'espace ? Pour ceux que ça ne dit pas grand chose, je vous le rappelle :

```
P=ax+by+cx+d=0
```

Le vecteur (a,b,c) correspond à la direction du plan, et pour être précis, il s'agit de la direction de la normale au plan. Les cinq plans possèdent l'équation très simple de plans passant par l'origine, et de « pente » correspondant aux angles d'ouverture de la caméra (ax et ay):

```
x=tan(ax)*z
x=-tan(ax)*z
y=tan(ay)*z
y=-tan(ay)*z
z-1=0
```

Comment savoir si un point est d'un côté ou de l'autre d'un plan ? Facile, il suffit de remplacer les x,y et z de l'équation du plan par la position du point. Si le résultat est négatif, alors le point se trouve d'un côté, si il est positif, alors c'est que c'est de l'autre côté. Si le résultat est nul, ben c'est que le point est sur le plan vu... qu'il vérifie alors l'équation du plan !

On peut donc savoir (après un petit test pour savoir si le côté qui nous interesse est positif ou négatif, j'en suis jamais bien sûr, fo essayer J) si un point se trouve dans le viewport ou pas.

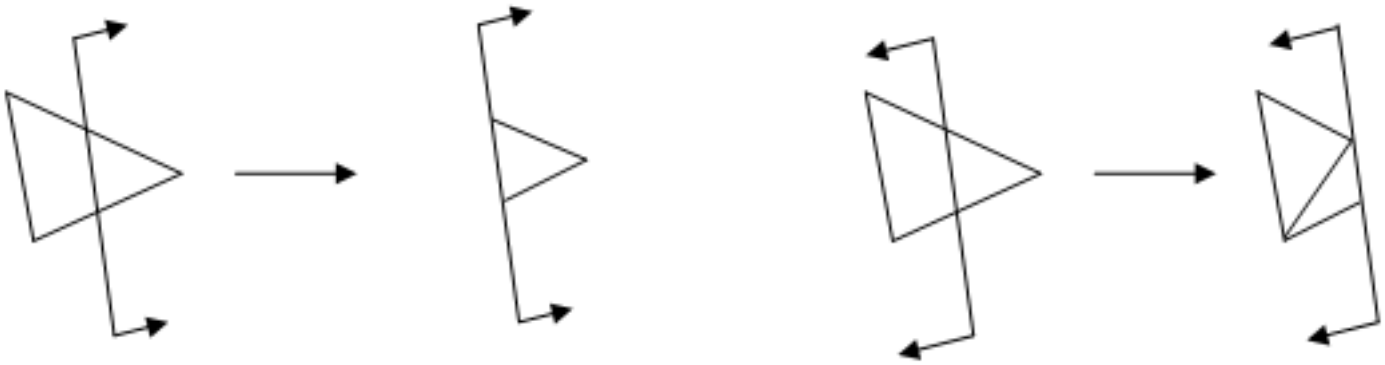
Dans l'histoire, nous on a une série de triangle. Prenons un triangle. On a également une série de plans. Prenons un plan. Il y a trois possibilités :

- Soit les trois points du triangle sont du « bon » côté du plan, et le triangle est alors testé avec le plan suivant. Si nous étions le dernier plan, alors le triangle peut être affiché tel quel.
- Soit les trois points du triangle sont du « mauvais » côté du plan, auquel cas il termine sa vie ici, il est éliminé et on en parle plus.
- Soit, évidemment, une partie des points est du bon côté, et les autres sont du mauvais, ce qui fait qu'il va falloir... couper notre triangle en deux.

C'est cette dernière situation qui est vraiment la plus interessante. Elle se décompose elle-même en deux cas :

- Soit deux points sont mauvais, et donc un seul point est bon.
- Soit le contraire, deux points sont bons, et un seul point est mauvais.

Illustration des deux cas:



Dans le premier cas, on voit qu'il faut remplacer le triangle par un autre triangle, dont les deux points qui étaient en dehors sont à présent à l'intersection des côtés du triangle et du plan.

Le second cas est un peu plus amusant : vu qu'on a dit qu'on ne dessinait que des triangles, il faut remplacer le triangle par... deux triangles, hé wais, qu'est-ce qu'on rigole...

Sans compter que ces deux triangles, il faudra ensuite, pour chacun, les tester avec les autres plans, et chacun de ces triangles risque donc fort de se retrouver à nouveau coupé en deux nouveaux triangles. C'est ainsi qu'un petit triangle de rien du tout peut se retrouver, après être passé par les cinq plans, transformé en... 32 triangles ! Rassurez-vous, ça n'arrive jamais, le cas défavorable le plus fréquent c'est qu'un triangle « rencontre » deux plans, et il est rare que chaque plan ne transforme le triangle en deux triangles, donc rassurez-vous, c'est pas aussi dingue qu'il n'y parait...

Par contre, pour le programmer, c'est une autre histoire, mais pour ça je vous fais confiance, moi ici j'explique les principes =).

Sans compter que c'est encore plus drôle que ça, car bien souvent la face possède une texture, et chaque point possède une couleur. Il faut donc trouver la valeur de la position de la texture au nouveau point ainsi créé, et la couleur de ce nouveau point. Une règle de trois y suffit (on fait une interpolation linéaire), mais ça complique d'autant le programme...

Il faut également vérifier que l'ordre des points est le même après avoir coupé le triangle. Si les points étaient définis dans l'ordre horlogique, ils doivent le rester, et inversément. Pourquoi ? Hop, point suivant...

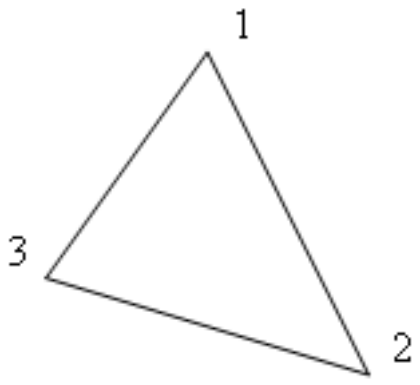
Le backface culling

Et un terme barbare, un...

Cette fois, il s'agit de quelque chose qui se fait après la projection, et donc sur base des coordonées écran des points des triangles. Le principe est assez simple, mais il n'est pas toujours d'application. Il consiste à repérer les faces qui nous tournent le dos, et à ne pas les dessiner. Pour des objets comme un cube ou une sphère, c'est évident que les faces qui nous tournent le dos ne sont pas visibles, mais pour des objets plus complexes, ça n'est pas forcément le cas. En général, il faut créer l'objet en se souvenant que seul un côté de la face est visible. Si on veut avoir une face visible des deux côtés, il faut... créer deux faces, l'une sur l'autre, chacune « pointant » dans une direction opposée.

Ou bien explicitement signaler qu'on ne veut pas utiliser le backface culling (ou backface removal, ou supression des faces cachées)

Comment savoir si une face nous tourne le dos ? Il suffit d'avoir dit au mec qui a créé l'objet (ou à l'éditeur, 3dsmax le faisant automatiquement) que toute face doit être définie dans l'ordre horlogique (ou anti-horlogique, l'important c'est que ça soit la même convention tout le temps, bien sûr...). C'est à dire que quand on nous donne les points du triangle, ces points doivent être donnés dans l'ordre suivant :



Et comment qu'on sait si les points sont dans l'ordre horlogique ou pas ? Et bien c'est simple : connaissez-vous un truc qui s'appelle le produit vectoriel ? Je ne vais pas vous retaper la formule parce qu'elle est chiante. La seule astuce ici est de voir que, après tout, les trois points sont dans un

même plan (forcément, ils sont sur l'écran...), et que la formule du produit vectoriel, chiante à ses heures, se simplifie fort bien dans cette situation particulière.

Et c'est quoi le produit vectoriel ? Ca permet de trouver un axe dans l'espace, qui est perpendiculaire à deux autres axes, et qui représente par sa norme et sa direction la rotation qu'il faut effectuer pour passer d'un axe à un autre.

Par exemple, si on faisait le produit vectoriel de l'axe 1-2 avec l'axe 1-3 de notre triangle, on obtiendrais un axe qui est perpendiculaire à l'écran, et qui y rentrerait ou en sortirait en fonction du fait... que la rotation à effectuer est positive ou négative. Et donc finalement, la valeur Z du produit vectoriel est positive ou négative en fonction du fait que l'ordre de définition est points est horlogique ou pas. Bingo ! On a notre test pour savoir si une face est face à nous ou pas.

La formule :

```
va1=x1-x2
vb1=y1-y2
va2=x3-x2

vb2=y3-y2

resultat=va2*vb1-va1*vb2
```

Resultat est positif ou négatif, en fonction de l'ordre des points. N'affichez les faces qui ont un resultat positif, et si ça ne marche pas, ben n'affichez que les faces qui ont un resultat négatif, comme toujours c'est toujours dur d'arriver du premier coup au première bon résultat =). En toute logique le bon resultat doit être négatif pour être affiché.

Ce test est fortement utilisé, imaginez qu'il permet en moyenne d'éliminer la moitié (!) des faces avant des les afficher.

On voit à présent pourquoi le clipping devait respecter l'ordre des points...

Un pipeline un peu plus complet

Avec ces derniers résultats, on peut compléter notre pipeline :

1. Trouver quelle est la nouvelle position de l’objet dans l’espace global. Egalement trouver la position de la caméra dans ce même espace.
2. Calculer la position des vectrices ainsi que la direction des normales dans l’espace global grâce à la matrice de base de l’objet.
3. Procéder au calcul des lumières pour trouver la couleur des points.
4. Utiliser la matrice caméra pour trouver la position de chacun des points dans l’espace caméra.
5. Procéder au clipping des faces, en éliminant les faces en dehors du viewport, et en découpant les faces qui sont à la limite. Faire passer les faces restantes au point suivant.
6. Utiliser la matrice de projection pour trouver la position de chacun des points dans l’espace écran.
7. Pour chaque face, regarder si elle est correctement orientée avec le test du backface culling, et si c’est le cas, la dessiner.

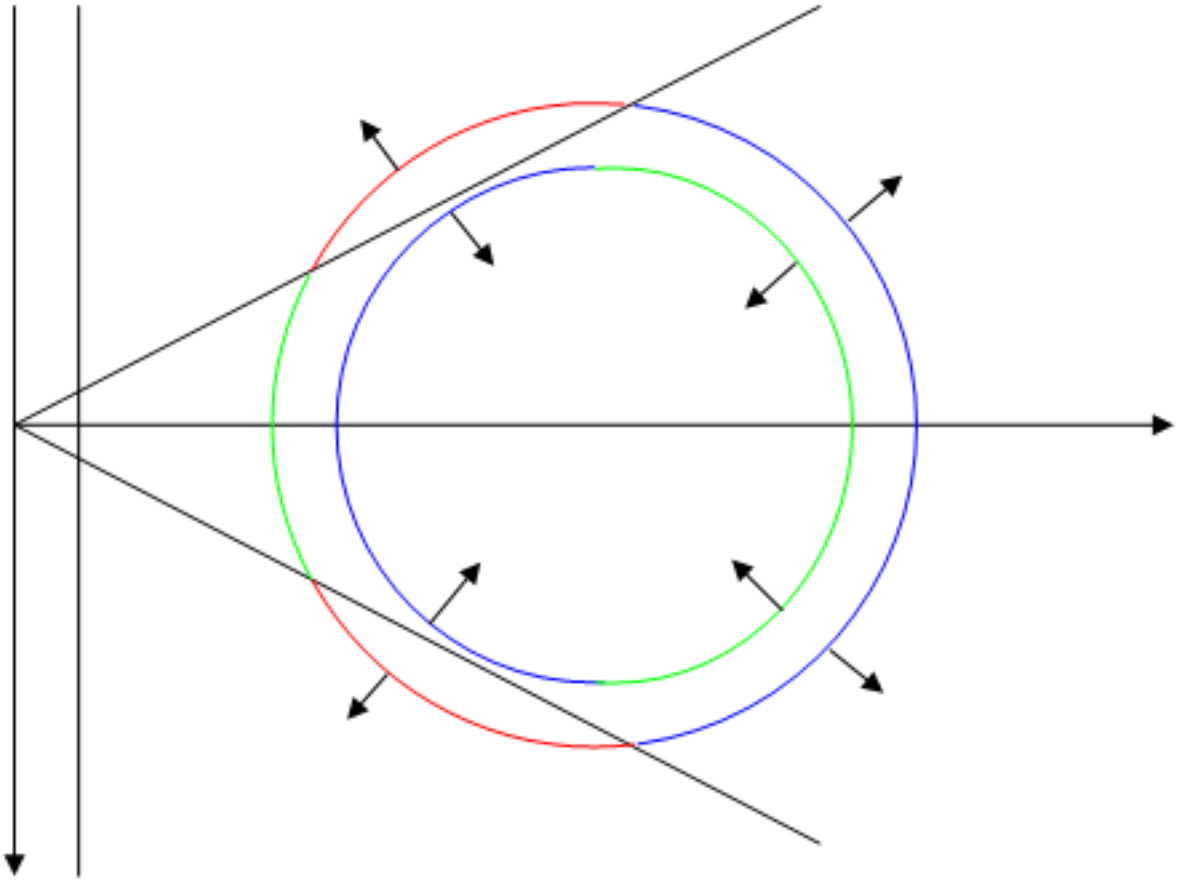
Le tri des faces et le Z-Buffer

Jusqu'à présent, le point 7 pourrait faire rire toute personne qui a déjà programmé un moteur 3D, car c'est en général par là qu'on commence (l'important n'est-il pas d'avoir quelque chose de joli à l'écran ?) et c'est également par là qu'on termine quand on se rend compte que ça n'est pas si simple que ça.

Outre le fait qu'afficher un triangle, avec une texture, des couleurs et même parfois de la transparence n'est pas quelque chose de facile en soit, il faut encore savoir dans quel ordre les afficher, les faces.

Ben wais, imaginez que vous ayez un tore. C'est joli un tore, non ? Pour ceux qui ne voient pas, un tore c'est un anneau, finalement... Bon, on veut dessiner notre tore. On fait donc un peu de backface culling, ce qui supprime toutes les faces qui nous tournent le dos.

Wé mais bon, il y a toute une série de faces qui passent à travers le test du backface culling sans pour autant être visibles, vu du haut ça donne :



En rouge, j’ai mis les parties qui sont éliminées par le clipping.

En bleu, on a ce que le backface culling a supprimé.

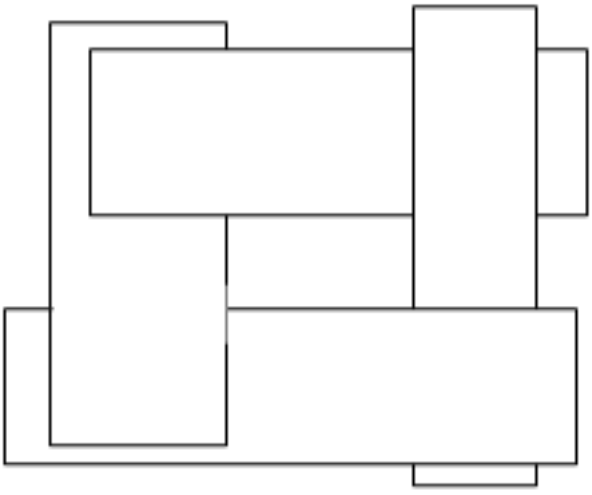
Finalement restent en vert vers parties « visibles » du tore. Visibles, vraiment ? Moi j’ai comme l’impression qu’on ne va pourtant voir que la partie la plus proche, qui recouvre la partie intérieure, plus éloignée.

Hé oui, mauvaise nouvelle, si le backface culling élimine une bonne moitié des faces non visibles, elle ne les élimine pas toutes non plus... (ce qui revient à dire qu’en moyenne dans un objet, beaucoup moins de la moitié des faces ne sont réellement visibles, ça laisse songeur et ça permet de trouver la motivation pour des systèmes encore plus performants, comme pour les moteurs de Quake etc...)

Et donc, si on ne fait pas attention, on va tout d’abord afficher les faces qui sont tout devant, pour ensuite afficher les faces qui sont les plus éloignées (pourquoi pas, après tout, hein ?), et le résultat va être assez moche. Ce qu’il faut donc faire avec les faces qui restent, c’est les trier selon leur distance à l’observateur (ou selon leur coordonée Z, ce qui ne revient pas exactement au même mais qui suffit malgré tout, ça évite de calculer une distance), et commencer à afficher d’abord les faces les plus éloignées, pour poursuivre et terminer avec les faces les plus proches.

L’ennui, c’est que si des faces, il y en a beaucoup, trier ça va prendre un sacré temps (et puis c’est fatigant). Remarquez, des algorithmes de tri, il y en a de très performants (je pense bien sur au quicksort en nlog(n) pour les amateurs), mais c’est pas la panacée non plus (quoique parfois on ne peut pas s’en passer, pour la transparence par exemple).

D’autant plus que le tri ne permet pas de régler le problème suivant, vu de l’avant :



Bon, vous me direz que ce genre de dessins, on ne le rencontre pas souvent et vous aurez raison, mais c'est juste un exemple, des situations qui font foirer le tri en Z, il y en a plus que vous ne le pensez. Sans compter qu'un tri ne vous permettra jamais de gérer l'intersection de deux triangles...

Il faut donc autre chose, et c'est autre chose, c'est le Z-Buffer.

L'idée est assez simple. Si, au moment de tracer le triangle à l'écran, on connaît la position en Z de l'espace caméra de chaque sommet (ce qui est le cas, on en a eu besoin pour le clipping et d'ailleurs pour le tri si on fait du tri), on peut connaître, monayant quelques petits calculs savants qu'on appelle une règle de trois (sisi) ou plus précis quand on fait de la correction de perspective (voir plus loin, toujours plus loin...) la position en Z de chaque points du triangle sur l'écran.

A quoi ça sert ? Et bien c'est évident, on garde, dans un buffer (d'où le nom, subtil), la position Z de chaque point dessiné sur l'écran (ou une valeur « très loin » si aucun point n'est dessiné à cet endroit sur l'écran). Ca demande un peu de place en mémoire, comme en général on stocke les valeurs sur 32bits et que les écrans sont en 1024x768, ça prend quelque chose comme 4mo pour ce z-buffer. Mais bon, il faut ce qu'il faut.

Et on en fait quoi de ce buffer ? C'est simple voyons, avant de tracer un pixel sur l'écran, on regarde si la position Z (la distance, finalement) du point que l'on va dessiner est plus éloignée que celle du pixel déjà dessiné à cet endroit. Si c'est le cas, alors on ne dessine rien du tout et on passe au pixel suivant. Sinon, et bien on trace le pixel et on mes à jour le Z-Buffer.

Grâce à ce système, non seulement on a plus besoin de faire attention à l'ordre d'affichage de nos faces (plus de tri), mais en plus on peut gérer les situations comiques vues un peu plus haut, et, comble du rafinement, on peut gérer au pixel près les intersection de faces de deux objets qui se rentrent dedans. Que demander de plus ?

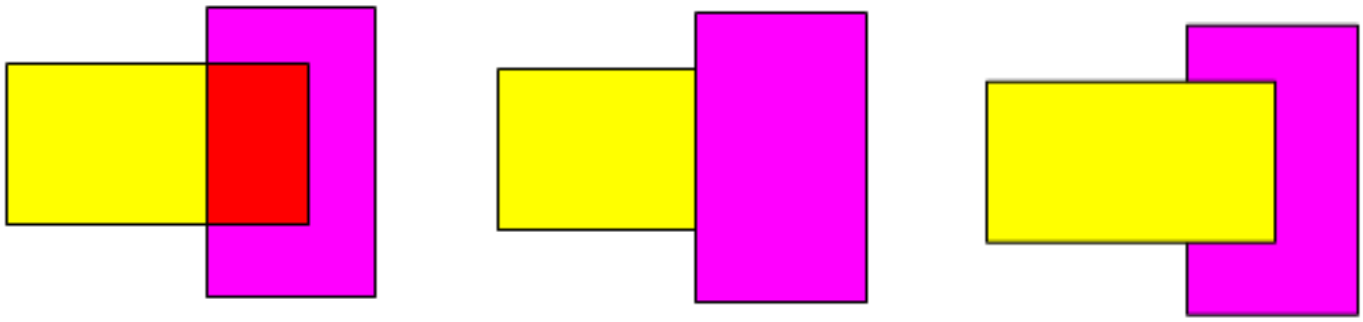
Considérations sur la transparence

En général, la transparence ne fait pas bon ménage avec le Z-Buffer, mais on peut tenter de les faire cohabiter sans trops de dégats.

Le problème, avec la transparence, c'est que ce qui se trouve derrière une face peut être visible (c'est justement pour ça qu'elle est transparente). On voit tout de suite que si on affiche la face en Z-Buffer, toute face qui serait dessinée après la face transparente, et se situant derrière elle, ne serait pas vue.

On peut corriger partiellement ce problème en décidant qu'au moment de tracer la face, on se contente de lire le Z-Buffer, sans jamais écrire dedans. Mais ça inverse le problème : si une face non-transparente est affichée derrière une face transparente, elle va recouvrir tout ou une partie de cette face transparente, mais sans être affectée par la transformation venant de la transparence (texture transparente, ...).

Exemple : la face rose est transparente, la face jaune ne l'est pas, se trouve derrière la face rose et est affichée en dernier. A gauche, le résultat correct, au centre le résultat avec Z-Buffer, et à droite le résultat avec Z-Buffer uniquement en lecture pour la transparence :



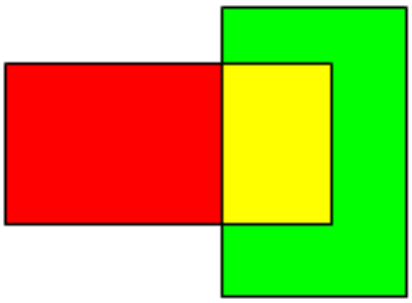
Par contre, si on se débrouille pour que la face rose (transparente) soit toujours affichée après la face jaune, alors il n'y a aucun problème, c'est correct dans tous les cas. Première leçon : toujours afficher les faces transparentes après les faces qui ne le sont pas.

Le second problème qui va se poser, c'est que, bien souvent, la formule de transparence utilisée n'est pas associative ni commutative. Je m'explique : si on a deux faces transparentes, afficher la premier au dessus de l'autre ou vice-versa ne va pas donner le même résultat (malgré ce que l'on

pourrait croire à prioris). Ca vient du fait qu'en général on utilise plutôt une formule de superposition à la place d'une formule de filtrage.

Le filtrage, c'est ce qui se passe quand vous placez effectivement une face rose transparente au dessus d'une face jaune : vous voyez du rouge. Pourquoi ? Parce que la couleur jaune est une combinaison de rouge et de vert, tandis que la couleur rose est une combinaison de rouge et de bleu. Le point commun entre les deux, c'est bien le rouge. Il s'agit du filtrage (le rose ne laisse passer que le rouge et le bleu, le vert du jaune est donc arrêté).

La superposition, c'est l'inverse, on fait un mix des couleurs. C'est à dire que c'est exactement le contraire : du rouge sur du vert, ca va donner du jaune, comme ceci :



On utilise cette formule car elle est plus simple à programmer. Mais si vous permuttez l'ordre des faces, bien souvent le résultat sera différent (pas ici avec ces exemples simples). Pour me convaincre, en son temps, j'avais fait la preuve mathématique, mais je suis sûr que ça ne vous intéresse pas, contentez-vous de me croire et puis basta.

Bref qu'au final, il faut faire quoi ? Il faut trier les faces transparentes pour être sûr de toujours les afficher dans le même « bon » ordre. Les trier selon Z par exemple, comme à l'époque de l'avant Z-Buffer. Un vrai plaisir, hein ?

Moralité, un pipeline devrait faire les choses suivantes :

- Afficher, avec Z-Buffer, toutes les faces non transparentes.
- Trier selon Z toutes les faces transparentes.
- Afficher, avec Z-Buffer uniquement en lecture, toutes les faces transparentes dans l'ordre Z.

Mais en général on ne possède pas toutes les faces à afficher d’un seul coup comme ça. En pratique il est normal de passer les objets un à un en revue, et de relancer le pipeline dessus chaque fois. Ben ici ça ne marche plus, car avec ça, même si les faces transparentes sont affichées en dernier dans un objet particulier, il est fort probable que l’objet suivant affichera des faces non transparentes.

Une solution, c’est de séparer les objets en deux catégories : les objets transparents et les objets qui ne le sont pas. Les objets non transparents sont affichés avant les transparents, et les faces des objets transparents sont tirées. De plus, les objets transparents sont tirés en Z entre eux, de sorte d’afficher en premier les objets transparents les plus éloignés. Il faut bien comprendre qu’un objet « transparent » sera considéré tel quel si il possède au moins une face transparente. C’est pourquoi il est conseillé de faire des objets soit totalement transparents, soit pas du tout. Si vous faites un avion avec le cockpitt transparent, il vaudrait mieux décomposer cet avion en deux objets : le cockpitt et le reste de l’avion. Et le cockpitt sera affiché après l’avion.

Ce système fonctionera « souvent », mais dès que vous ferez des intersections entre des objets transparents, tout va tomber en morceau. Moralité : évitez à tout prix de faire se rentrer dedans des objets transparents.

La gestion de la transparence demande un peu plus d’organisation que l’affichage de « bêtes » objets non transparents.

Les triangles 2D

11.1. Introduction

Voir mon tutoriel sur le traçage de rectangle flat, gouraud et texturés, je commence en supposant que c'est lu, na =)

Bon, avec ces fichus tuts, vous en savez déjà un peu plus, on va donc aller voir un peu plus loin, mais pas tellement.

En réalité, il arrive rarement que l'on se contente d'afficher un triangle avec une texture, ou bien uniquement un ombrage gouraud. En général on aime fusionner les deux de manière à pouvoir modifier la luminosité de la texture en fonction des lumières et autres contrariétés habituelles.

Il faut donc juste être capable de mixer les deux algos, et de fonctionner en 32bits. Au lieu d'interpoler une valeur de couleur, on en interpolle trois (un rouge, un vert et un bleu), on interpole aussi les coordonées de texture, et si on fait du Z-Buffer, on interpole la valeur Z. On peut même interpoller une valeur Alpha pour la transparence.

- Je trouve la couleur R,G,B qui provient du gouraud
- Je trouve la couleur R,G,B qui provient de la texture
- Je mixe les deux couleurs avec une simple moyenne géométrique, c'est à dire en faisant (en gros) le produit de chaque composante. Rtotal=Rgouraud*Rtexture etc...
- Si l'algo doit faire de la transparence, je vais rechercher la couleur qui se trouve déjà à l'écran au même endroit et j'utilise la formule d'alpha classique : couleur=couleur_originale*alpha+nouvelle_couleur*(1-alpha)
- Et j'affiche le résultat, sous réserve que le pixel soit plus proche si je fais du Z-Buffer.

Pour mixer le tout, voici comment je fais moi :

Pour faire plus complet encore on peut utiliser comme valeur alpha un mix entre la valeur alpha calculée par gouraud, une valeur alpha propre à la texture (alors la texture est en 32bit rgba) et une valeur alpha disponible à l'écran dans les 8 derniers bits. Il faut alors modifier en plus cette valeur alpha à l'écran.

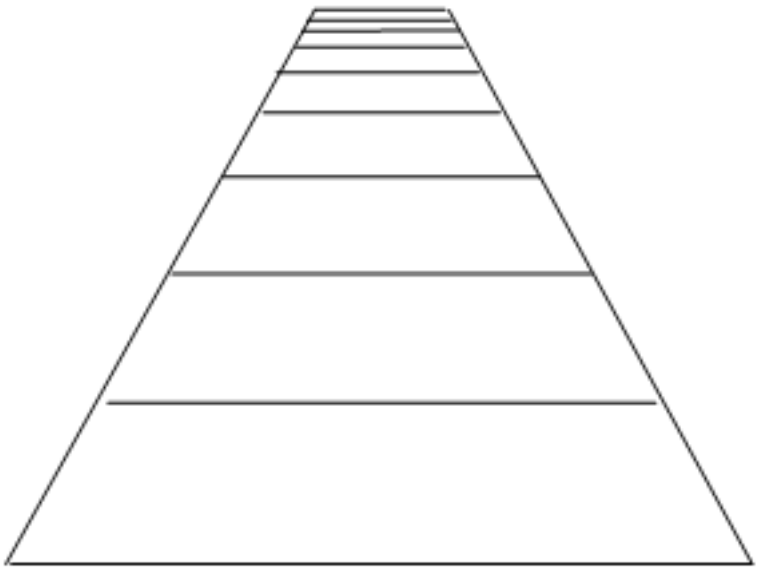
Mais bon...

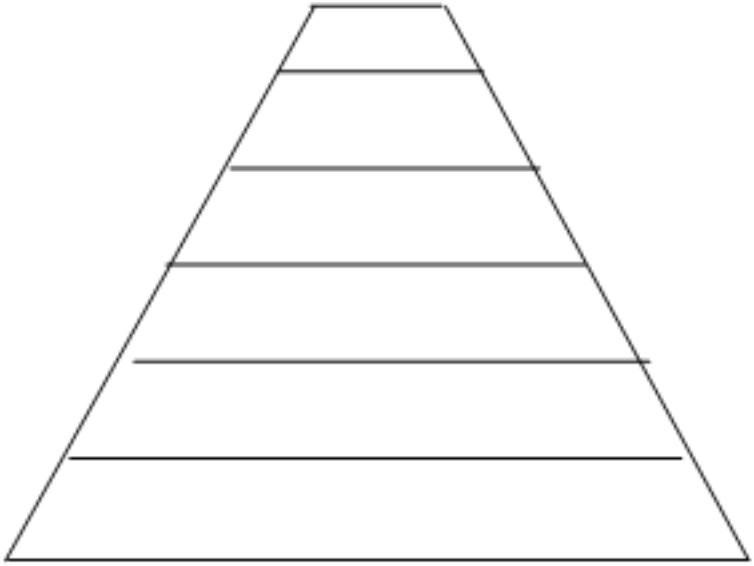
11.2. La correction de perspective

Le système de traçage de triangle « classique » comme proposé dans les tuts en annexe par sur un postulat faux (c'est pas de bol) : la coordonée de la texture au centre du triangle n'est pas la moyenne des coordonées au sommet. Il en va de même pour le gouraud et pour la valeur Z, en fait tout est faux, du début à la fin. (Pas beaucoup, mais quand même...)

Pour comprendre pourquoi, prenons une texture « chemin de fer » :

Si nous nous mettons sur les rails et que nous regardons dans la direction de la voie, nous avons à gauche la vue correcte, et à droite ce que nous donnera notre programme incorrect :





C’est logique, notre algo subdivise la hauteur en sous-hauteurs égales, sans comprendre que plus on est haut, plus on est loin, et que les distances doivent diminuer.

Et encore, comme c’est là ça ne saute pas toujours aux yeux, mais quand ça bouge, je vous jure qu’on ne s’y laisse pas tromper plus d’un dixième de seconde...

Heureusement, ya un jour un mec qui a sortit une feuille et un bic et qui a trouvé (c’était pas dur, remarquez, fallait juste y penser) que si u,v ou n’importe quoi ne pouvait pas s’interpoller linéairement comme ça sur l’écran, n’importe quelle valeur divisée initialement par z pouvait, elle, s’interpoler de cette manière, à condition qu’avant de l’utiliser, on remultiplie la valeur par le bon Z local.

Et comment on le trouve le bon Z local ? Ben c’est jusque 1/Z lui assi, forcément, il s’interpole correctement linéairement.

Alors quoi qu’on fait ? C’est simple :

1. On commence par diviser les valeurs par leur Z respectif. inverse_u1=u1/z1, inverse_v1=v1/z1, inverse_r1=r1/z1 etc... et finalement inverse_z1=1/z1 ;
2. On interpole ces valeurs comme précédement, ca c’est facile
3. Une fois qu’on a besoin de ces valeurs pour dessiner quelque chose d’interessant à l’écran, on remultiplie tout par z, ce qui revient, remarquez, à diviser par 1/z, et ça tombe bien, on l’a justement sous la main : vrai_u= inverse_u/ inverse_z
4. C’est prêt, régalez-vous...

Evidement, avec des calculs pareils, il faut passer par virgule flottante et ça trace pas beaucoup. Petit exercice : trouver comment faire ça rapidement =)

Les portals

Avec tout ce qu'on a vu jusqu'à présent, on est en théorie capable d'afficher à peu près n'importe quoi. Le problème c'est qu'en général, le moteur ne « comprend » pas l'objet. Pour lui, il n'est qu'une suite de triangles. C'est normal me direz-vous, que voulez-vous qu'il sache d'autre ?

Ce genre de moteur « qui affiche les objets sans les comprendre », c'est ce qu'en général j'appelle des moteurs « externes », par opposition aux moteurs « internes », qui affichent en général les objets dans lesquels on se déplace. Dans des couloirs etc, donc tout ce qui est moteur à la doom-like.

Parce que c'est vrai, pour afficher un niveau de doom (ou de quake), que feriez-vous ? Vous allez passer en revue toutes les faces du niveau et les afficher ? Bonne chance, il y en a un certain nombre, et même un nombre certain. D'autant plus qu'il s'agit d'un gaspillage énorme. En général, on ne voit pas sur un écran le centième du niveau dans lequel on évolue.

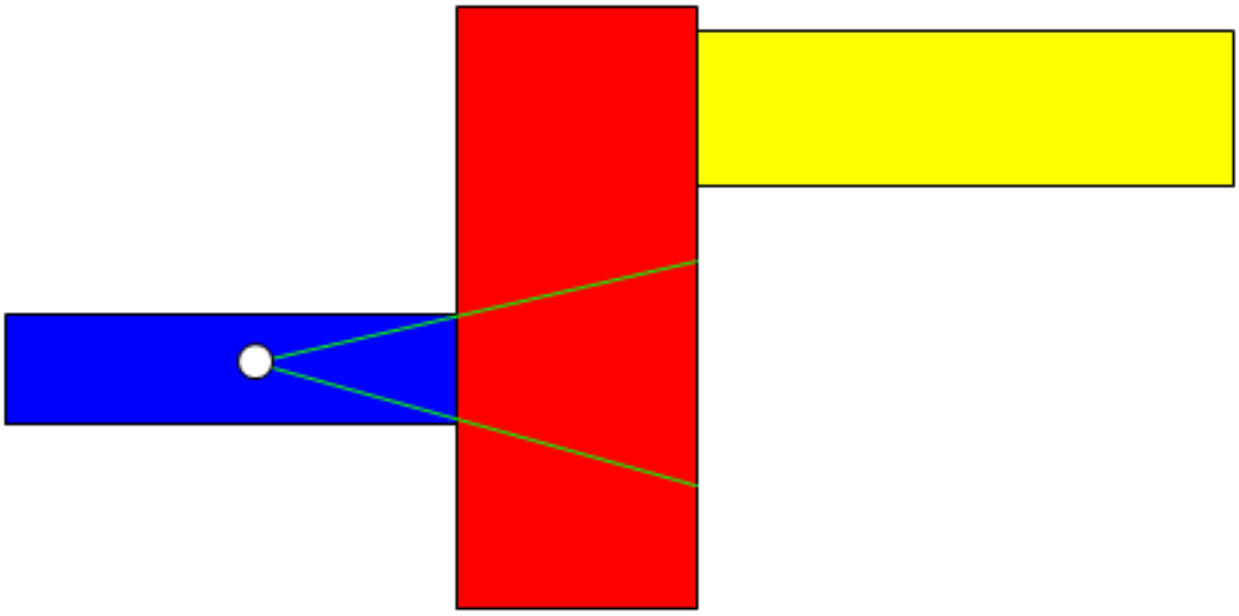
L'idée de base, pour afficher ce genre de niveaux, c'est de le décomposer en pièces, que l'on appellera « secteurs ». Ces secteurs sont fermés, dans le sens où il n'y a pas de « trous » dedans. Quelle que soit la direction dans laquelle on regarde, on va forcément tomber sur un mur du secteur. Ca peut sembler idiot, et bien pas tant que ça

Bon, vous allez me faire remarquer que si le secteur est à ce point fermé, cela veut dire qu'il n'y a tout simplement jamais la moindre porte à mes pièces, ça va être dur de se balader là dedans. Effectivement... Mais après tout, une porte, même ouverte (disons le « trou » de la porte), ne peut-on pas considérer ça comme étant une face ? Bon, une face un peu spéciale, mais une face quand même... En fait, une face qui dit : « moi, il faut pas chercher ma texture ou des conneries du genre, il faut juste afficher le secteur machin à la place ». Lequel secteur machin est également un secteur fermé, mais qui possède également des faces spéciales, lesquelles vont demander l'afficher d'autres secteurs, et ainsi de suite. Ces faces spéciales sont appelées de « portals ».

On se rend bien compte que, de proche en proche, avec une logique pareille, on va dessiner tout le niveau. Pire que ça, on risque même de boucler, donc de revenir au point de départ, de ne pas s'en rendre compte et de tout réafficher une fois de plus, et ainsi à l'infini. Plantage assuré.

C'est là qu'est la nouvelle idée : supposons que la pièce que nous regardons au travers d'un portal possède elle-même des portals. Est-il utile d'aller afficher le secteur qui se trouve derrière ce

nouveau portal si le portal lui-même n’est pas visible car caché par un des murs du secteur dans lequel on se trouve ? Explication en image vu du haut :



Mais c’est là qu’on réfléchit et qu’on se dit : si, au moment de tracer le secteur rouge, je resteignais mon champ visuel à l’ouverture de la porte (je sais le faire très facilement grâce à tout mon système du clipping 3D), ce qui serait effectivement dessiné du secteur rouge serait... uniquement ce qu’on est sencé voir du secteur rouge, et rien de plus ! Et donc, si avant de tracer le secteur jaune, on fait passer le portal rougeà jaune dans le « clippeur », comme toutes les autres faces, on se rendra compte ici que ce portal est invisible et doit donc être ignoré.

Si vous réfléchissez un instant, vous vous rendrez compte que ce système est totalement général et fonctionne très bien : quand on dessine un secteur et que l’on tombe sur un portal, on « clippe » ce portal de manière à n’en garder que sa partie effectivement visible, on restraint le champ de vision (on reconfigure le clippeur) à cette partie visible du portal, et on dessine le secteur qui se trouve « derrière » ce portal. Lequel secteur va peut-être lui-même restreinte son champ de vision etc, ça fonctionne d’une manière récursive. Ce mouvement s’arrêtera de lui-même quand on arrivera dans des secteurs n’ayant aucun portals de visible, ce qui arrivera rapidement.

Si vous continuez à réfléchir, vous comprendrez que si vous partez du secteur dans lequel se trouve l’observateur avec comme clipping initial le clipping « écran » normal, vous afficherez EXACTEMENT ce qu’il faut afficher. Pas un pixel de plus, et pas un pixel de moins. Si votre écran fait 640x480 pixels, vous allez dessiner 640x480 pixels. Plus besoin de trier quoi que ce soit, plus besoin de faire de Z-Buffer, plus besoin de rien du tout, et vous ne prenez en compte que les

secteurs visibles, c’est à dire une toute petite partie du niveau. Génial, non ?

La seule difficulté, c’est d’être capable de correctement placer les plans de clipping 3D pour correspondre à « l’ouverture de la porte », ce qui est un petit travail de géométrique analytique pas trop compliqué. Il faut également se souvenir que modifier la visibilité pour afficher un portal, c’est bien, mais restaurer le clippeur comme il était avant, c’est encore mieux. Il faut donc penser à sauvegarder l’état du clippeur avant de passer aux autres portals.

Comme les niveaux de Doom et de Duke3D étaient en fait des niveaux « plats », le calcul du clipping était beaucoup plus simple, mais ça n’est jamais qu’un niveau de généralisation en plus. Si, à l’époque, ils n’ont pas fait des niveaux un peu plus génériques, je pense que c’est plus un problème d’affichage de faces sans trop de correction de perspective qu’autre chose. Car dans Doom, les murs sont toujours verticaux et le sol est toujours horizontal, ça permet de simplifier de beaucoup le traçage des faces avec une bonne perspective.

Annexes

13.1. Matrices de transformation 3D 4x4

Identité

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation

$$\begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation autour de x

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & -\sin(a) & 0 \\ 0 & \sin(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation autour de y

$$\begin{pmatrix} \cos(a) & 0 & \sin(a) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(a) & 0 & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation autour de z

$$\begin{pmatrix} \cos(a) & -\sin(a) & 0 & 0 \\ \sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation autour d'un axe (x,y,z) (priez pour qu'il n'y ait pas de faute =))

$$\begin{pmatrix} \cos+a^2(1-\cos) & -c\sin+ab(1-\cos) & b\sin+ac(1-\cos) & 0 \\ c\sin+ab(1-\cos) & \cos+b^2(1-\cos) & -a\sin+bc(1-\cos) & 0 \\ -b\sin+ac(1-\cos) & a\sin+bc(1-\cos) & \cos+c^2(1-\cos) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Projection centrale via (0,0,-d)

$$\begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & d \end{pmatrix}$$

Changement d’échelle (a,b,c)

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$