Détection des collisions dans un univers 3D

Table des matières

1 – Notions de géométrie vectorielle

Pour bien comprendre le principe des calculs de collisions dans un univers 3D, il est nécessaire de connaître quelques notions de géométrie vectorielle. Les mathématiciens me pardonneront, j'espère, de préférer un langage simplifié aux termes scientifiques, mais le présent document s'adresse aussi – en fait surtout – à des débutants, et ceux-ci ont besoin d'explications simples et d'images claires, plutôt que d'un document scientifiquement correct, mais qu'ils ne seront jamais en mesure de comprendre.

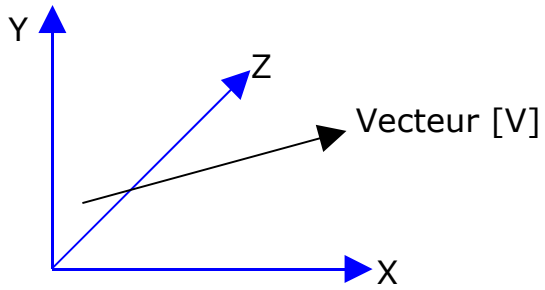
Le développement 3D requiert de bonnes connaissances en algèbre, en trigonométrie et en géométrie, car dans ce domaine, tout est vectoriel. je vous conseille donc de dépoussiérer vos vieux bouquins de mathématiques si vous souhaitez devenir expert en ce domaine.

1.1 – Les vecteurs

1.1.1 – Le vecteur, du point de vue mathématique

Selon Wikipédia, en mathématiques, le vecteur est un objet véhiculant plus d'informations que les nombres usuels et sur lequel il est possible d'effectuer des opérations. À l'origine, un vecteur est un objet de la géométrie euclidienne. À deux points, Euclide associe leur distance. Or un couple de points porte une charge d'information plus grande. Ils définissent aussi une direction et un sens. Le vecteur synthétise ces informations. La notion de vecteur peut être définie en dimension deux (le plan) ou trois (l'espace euclidien usuel). Elle se généralise à des espaces de dimension quelconque. Cette notion, devenue abstraite et introduite par un système d'axiomes, est le fondement de la branche des mathématiques appelée algèbre linéaire.

Donc, on peut aussi imaginer un vecteur géométrique comme l'expression d'une droite, ou d'un segment de droite dans un espace euclidien.



1.1.2 – Le vecteur, du point de vue de Direct3D

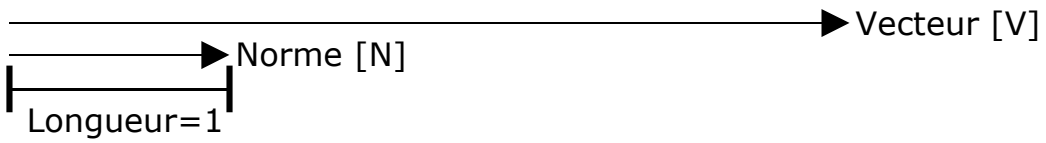
Direct3D met à disposition une classe, nommée D3DXVECTOR3, qui représente un vecteur, toutefois ce terme prend ici un sens plus vague que dans le monde scientifique. Dans Direct3D, un vecteur peut par exemple être utilisé pour représenter la coordonnée d'un point, ou pour exprimer une direction. La classe D3DXVECTOR3 contient 3 variables, x, y et z, qui représentent bien sûr les coordonnées d'un point ou d'une direction sur l'axe X, Y et Z. Cette classe

assume également diverses opérations, entre autres mathématiques, sur les vecteurs, telles que l'addition, la soustraction, la multiplication et la division de deux vecteurs, ou la comparaison entre deux vecteurs.

1.1.3 – Mathématiques vectorielles

Certaines formules de géométrie vectorielle vont être utile pour calculer une collision. Nous aurons besoin de **normaliser** des vecteurs, puis de connaître le **produit scalaire** (nommé Dot Product, selon le terme anglais, dans beaucoup de documents) de deux vecteurs. Nous devrons aussi connaître le **produit vectoriel** (nommé Cross Product, selon le terme anglais, dans beaucoup de documents) de deux vecteurs, pour calculer un **plan**.

La **normalisation** d'un vecteur est une opération qui consiste à ramener la norme d'un vecteur à 1. Pour faire cela, on divise chacune des coordonnées du vecteur par sa norme.Après résultat, les valeurs sont comprises entre 0 et 1. D'une certaine façon, on peut considérer qu'en normalisant un vecteur, on supprime la notion de distance, pour ne garder que la direction du vecteur.



Voici à quoi ressemble la formule géométrique d'une normalisation:

N = ||V||

Voici la traduction en algèbre:

$N.x=V.x/\sqrt{V.x^2 + V.y^2 + V.z^2}$; $N.y=V.y/\sqrt{V.x^2 + V.y^2 + V.z^2}$; $N.z=V.z/\sqrt{V.x^2 + V.y^2 + V.z^2}$

Enfin, sous forme d'algorythme, dans un code en C++, ça donne ceci :

```
D3DXVECTOR3 Normalize( D3DXVECTOR3 Vector )

{
    float Len = sqrt( ( Vector.x * Vector.x ) +
                      ( Vector.y * Vector.y ) +
                      ( Vector.z * Vector.z ) );

    return D3DXVECTOR3( ( Vector.x / Len ),
                        ( Vector.y / Len ),
                        ( Vector.z / Len ) );
}
```

Le **produit vectoriel**, aussi appelé **produit croisé**, permet de calculer la normale à un polygone planaire à partir de 3 de ses sommets non alignés. Ceci nous permettra par exemple de calculer une normale pour un plan, comme nous le verrons plus loin. Attention lors du calcul d'un produit vectoriel : Si vous croisez les vecteurs, vous inversez aussi le sens du vecteur résultant.

Voici à quoi ressemble la formule géométrique d'un produit vectoriel:

V = V1 x V2 ou V = V1 ^ V2

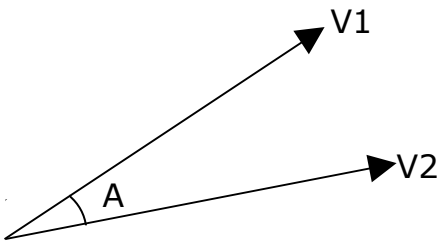
Voici la traduction en algèbre:

V.x = V1.y * V2.z – V2.y * V1.z
V.y = V1.z * V2.x – V2.z * V1.x
V.z = V1.x * V2.y – V2.x * V1.y

Enfin, sous forme d'algorythme, dans un code en C++, ça donne ceci :

```
D3DXVECTOR3 GetCrossProduct( D3DXVECTOR3 V1, D3DXVECTOR3 V2 )
{
    return D3DXVECTOR3( ( V1.y * V2.z ) - ( V2.y * V1.z ),
                        ( V1.z * V2.x ) - ( V2.z * V1.x ),
                        ( V1.x * V2.y ) - ( V2.x * V1.y ) );
}
```

Le **produit scalaire** permet de calculer l'angle entre deux vecteurs. Pour cela, il faut normaliser les vecteurs, puis multiplier chacune des valeurs du premier vecteur par celles du deuxième. L'addition des résultats donne le cosinus de l'angle entre les deux vecteurs.



Voici à quoi ressemble la formule géométrique d'un produit scalaire :

A = V1.V2

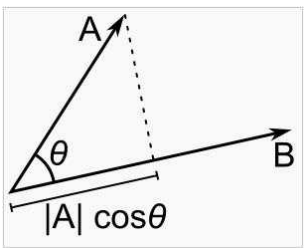
Voici la traduction en algèbre :

cos(A) = (V1.x * V2.x) + (V1.y * V2.y) + (V1.z * V2.z)

Enfin, sous forme d'algorythme, dans un code en C++, ça donne ceci :

```
float DotProduct( D3DXVECTOR3 V1, D3DXVECTOR3 V2 )
{
    return ( ( V1.x * V2.x ) + ( V1.y * V2.y ) + ( V1.z * V2.z ) );
}
```

Une autre propriété intéressante du produit scalaire est qu'il permet de calculer la projection d'un vecteur sur un autre vecteur. En effet, la normale de V1 multiplié par le cosinus de l'angle permet d'obtenir la projection du vecteur V1 sur V2, comme vous pouvez le voir sur l'illustration ci-dessous.



Ne vous inquiétez pas si vous ne comprenez pas tout, c'est surtout le but des calculs qu'il faut comprendre. Gardez en tête que la **normalisation** d'un vecteur ramène le vecteur à des valeurs comprises entre 0 et 1, et que le **produit scalaire** de deux vecteurs permet de calculer l'angle entre ceux-ci.

1.1.4 - Le calcul vectoriel avec Direct3D

Direct3D mets à disposition un ensemble de fonctions qui permettent de simplifier grandement les calculs de géométrie vectorielle. Les fonctions suivantes nous intéressent particulièrement :

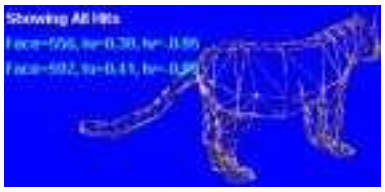
- D3DXVec3Dot
- D3DXVec3Cross
- D3DXVec3Normalize

Je ne pense pas nécessaire de donner de plus amples explications ici, les noms des fonctions me paraîssent assez clairs. Vous n'avez donc pas à vous préoccuper de mettre en place les équations vectorielles décrites ci-dessus lors de votre développement en C++, Direct3D s'en charge pour vous.

1.2 – Les polygones

1.2.1 – C'est quoi ?

Un polygone est un ensemble de plusieurs points, appelés sommets, représentant des coordonnées dans l'espace 3D. Comme pour les briques dans une construction, les polygones assemblés forment les objets visibles dans le monde 3D. Un polygone est un modèle mathématique simple, c'est pour cela qu'il est utilisé pour les rendus 3D.



1.2.2 – Le polygone dans Direct3D

Direct3D utilise exclusivement des listes de polygones pour le rendu d'une scène. Chaque objet 3D est composé d'un ensemble de polygones, et comme chaque polygone est composé de 3 sommets, donc de 3 coordonnées spaciales, nous avons ici une ressource intéressante à exploiter pour calculer une collision.

Toutefois, Direct3D place les données des polygones dans une classe spéciale, appelée VertexBuffer. Cette classe contient non seulement les données spaciales de l'objet 3D, mais elle peut aussi contenir différentes données d'affichage, comme par exemple les coordonnées des textures, les couleurs des sommets, ou encore la normale du polygone, tout ceci en fonction des choix du développeur. Ceci complique quelque peu l'accès aux données nécessaires pour le calcul d'une collision.

1.2.3 – La classe E_Polygon

L'accès en lecture/écriture d'un VertexBuffer est un procédé qui dépasse le

cadre du présent document, et de la démo qui l'accompagne, je ne vais donc pas entrer dans les détails ici. Toutefois, dans le but de faciliter la compréhension, j'ai crée une classe qui contient exclusivement les données dont nous avons besoin pour calculer une collision. Cette classe se nomme E_Polygon. On y trouve les coordonnées des 3 sommets formant un polygone, et quelques outils pour simplifier les calculs des collisions.

Pour faciliter encore un peu plus la compréhension du code, j'ai crée une liste chaînée pouvant contenir un nombre indéfini de polygones. Pour y accéder, nous avons juste besoin du pointeur sur le premier noeud de la liste, ainsi nous pouvons lire tous les noeuds les uns après les autres grâce à une fonction nommée GetNext. Le code de cette liste de polygones se trouve dans la classe E_PolygonList.

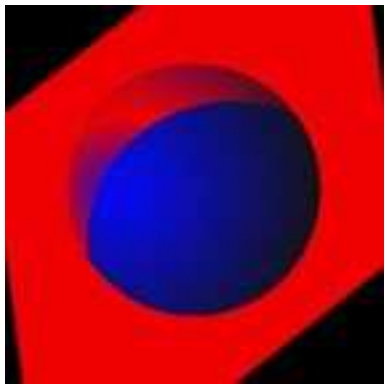
1.3 – Les plans

1.3.1 – Le plan, du point de vue mathématique

Selon Wikipédia, en mathématiques, un plan est un objet fondamental à deux dimensions. Intuitivement il peut être visualisé comme une feuille d'épaisseur nulle qui s'étend à l'infini.

Les plans sont donc des objets imaginaires qui séparent l'espace en deux parties. A cela, on peut ajouter qu'un point appartient à un plan si l'équation suivante peut être résolue : A*x+B*y+C*z+D = 0

Gardez en tête cette équation, c'est en quelque sorte une formule magique qui permet de résoudre une énorme quantité de problèmes mathématiques liés aux plans.



Analysons un peu la formule ci-dessus: [x, y, z] représentent les coordonnées d'un point dans l'espace. Le plan est représenté par les lettre [A, B, C, D], où [A, B, C] est la normale du plan, c'est-à-dire la direction perpendiculaire au plan dans l'espace, et [D] est la composante de décalage du plan. Cette dernière propriété est un peu difficile à saisir, c'est une notion mathématique complexe. Comme elle n'est pas nécessaire pour comprendre le mécanisme de calcul d'une collision, je ne m'étendrai pas plus sur ce point ici.

Pour construire un plan, il nous faut soit 1 point de l'espace et une normale (c'est un vecteur représentant une direction), soit 3 points. Le calcul de construction est relativement simple : Avec 1 point et une normale, vous avez

déjà la normale [A;B;C] du plan. Vous avez également les coordonnées spaciales [X;Y;Z] d'un point sur le plan. Vous pouvez donc calculer D en transformant la formule exposée ci-dessus, soit :

A*x + B*y +C*z = -D, donc D = -(A*x + B*y +C*z)

Si vous avez 3 points de l'espace, vous pouvez choisir P1 comme point d'origine du plan. Vous calculez ensuite deux vecteurs à partir des 3 points, de la manière suivante : V1 = P2 – P1 et V2 = P3 – P1. En effectuant un produit vectoriel sur V1 et V2, vous obtenez la normale du plan. Depuis ici, vous avez un point et une normale, vous pouvez appliquer la méthode décrite ci-avant. Attention toutefois au sens, si vous croisez V1 et V2 lors du produit vectoriel, vous inversez aussi le sens du plan.

1.3.2 – Les plans dans Direct3D

Direct3D met à disposition une classe, nommée D3DXPLANE, qui représente un plan. Cette classe contient les 4 variables, a, b, c et d, qui représentent la normale [A, B, C] du plan, ainsi que la composante de décalage [D].

1.3.3 – Les fonctions de calcul des plans avec Direct3D

Direct3D met à disposition un ensemble de fonctions qui permettent de simplifier grandement les calculs avec les plans. Voici celles qui nous intéressent:

- D3DXPlaneFromPoints

- D3DXPlaneIntersectLine

La première fonction permet de construire une classe D3DXPLANE à partir de 3 coordonnées, représentés par des objets D3DXVECTOR3. A ce titre, les polygones, avec leur 3 sommets, feront parfaitement l'affaire. La deuxième fonction permet de calculer à quelle coordonnée de l'espace un plan et un segment de droite se croisent. Nous reviendront plus tard sur ce point.

Nous aurons également besoin de connaître la distance qui sépare un point d'un plan. L'intérêt d'une telle fonction est de savoir si le point se trouve avant ou après le plan. Malheureusement, Direct3D ne met pas à disposition une telle fonction. Mais la solution est simple, il suffit d'appliquer la formule du plan décrite ci-dessus, et nous avons la réponse :

Distance = (A * P.x) + (B * P.y) + (C * P.z) + D

Vous avez remarqué que la première partie de la formule est un produit scalaire ? Nous pouvons donc écrire l'algorythme comme suit :

```
float GetDistanceToPlane( D3DXPLANE thePlane, D3DXVECTOR3 thePoint )
{
   // On récupère la normale du plan.
   D3DXVECTOR3 N = D3DXVECTOR3( thePlane.a, thePlane.b, thePlane.c );
   // Puis, on calcule la distance entre le plan et le point.
   return D3DXVec3Dot( &N, &thePoint ) + thePlane.d;
}
```

1.4 – Conclusions

Voilà, vous n'avez pas besoin d'en savoir plus pour comprendre la suite des explications. Je vous invite toutefois à réviser votre géométrie, votre algèbre et votre trigonométrie, vous en aurez besoin pour devenir un excellent développeur 3D.

2 – Principe de fonctionnement du moteur de collisions

Maintenant que vous possédez quelques connaissances en géomértie vectorielle, nous pouvons étudier la manière d'utiliser tout cela pour calculer un point de collision.

2.1 – Méthode de calcul d'une collision

2.1.1 – La sphère de collision

Nous aurons besoin de définir mathématiquement un joueur de la manière la plus simple possible. Je suppose que tous verront l'intérêt à ne pas calculer une collision entre un mur et un objet représentant un corps humain en mouvement.

Nous allons utiliser un objet mathématique simple, qui englobera notre personnage. Pour ma démo, j'ai choisi la sphère, car elle convient assez bien à l'environnement dans lequel le joueur évolue : Une pièce fermée par quelques murs.

2.1.2 – Le plan du polygone

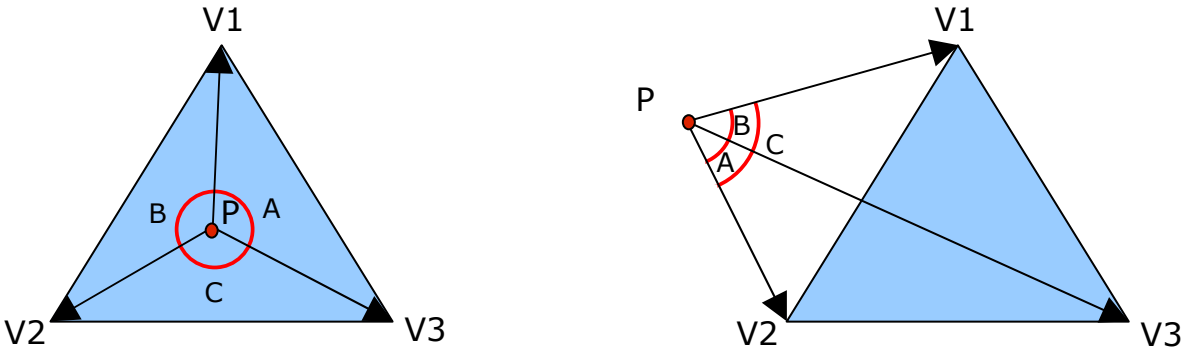
La première chose à faire, pour rechercher une collision, est de créer une frontière virtuelle. Si le segment de droite représentant le déplacement croise cette frontière, alors on est potentiellement en collision avec l'objet propriétaire celle-ci. Pour cela, quoi de plus pratique que de créer un plan à partir des 3 sommets du polygone ?

Lorsque le joueur se déplace, il est aisé de calculer où le vecteur partant de la position actuelle dans le sens du déplacement va croiser le plan du polygone, si croisement il y a. Ainsi, on peut déterminer un point de test P sur le plan du polygone.

2.1.3 – Le polygone et le point

Nous allons commencer par étudier une méthode simple et efficace, utilisée au coeur de beaucoup de calculs de collisions. Imaginez un polygone quelconque. Vous voudriez savoir si un point P se trouve à l'intérieur ou à l'extérieur du

polygone. Pour calculer cela, nous allons imaginer 3 segments de droites, reliant le point P et chacun des sommets du polygone. Voici deux illustrations :

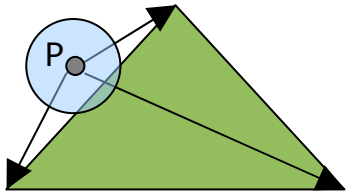


Dans la première illustration, le point P se trouve à l'intérieur du polygone, alors que dans la deuxième, il est à l'extérieur. Essayez d'additionner les angles A, B, C, formés par les segments de droites [P-V1/P-V2], [P-V2/P-V3] et [P-V3/P-V1]. Vous remarquerez que la somme des angles fait 360° dans la première illustration. Essayez à présent d'additionner les angles de la deuxième illustration. Vous ne pouvez pas arriver à une somme de 360°. Quel est la conclusion ? Si la somme des angles fait 360°, alors le point est à l'intérieur du triangle.

Bien. Mais comment on calcule les angles ? Eh bien, nous avons à disposition toutes les coordonnées pour calculer les vecteurs [P-V1], [P-V2] et [P-V3], puisque nous sommes censés connaître le polygone et le point P. Après, il suffit de calculer le produit scalaire entre chaque paire de vecteurs, et on peut obtenir tous les angles. Le problème est résolu.

2.1.4 – Le bord du triangle

L'algorythme ci-dessus fonctionne bien dans la plupart des cas. Mais connaître si le point P se trouve à l'intérieur du polygone n'est pas suffisant. Dans certains cas, le joueur se trouve tout près du polygone, la sphère de collision touche même celui-ci, mais le point P se trouve juste à l'extérieur du triangle. C'est le cas par exemple si le joueur arrive contre un angle convexe, un angle de mur, par exemple. Dans cet exemple, bien qu'il y ait une collision, le résultat de l'algorythme ci-dessus nous indiquera le contraire.



Donc, à ce stade, si le test de détection échoue, nous devons faire un deuxième test pour nous assurer qu'il n'y a vraiment pas de collisions. Nous devons tester le bord du triangle.

Voici comment nous allons calculer cela. Soit S1, S2 et S3 les 3 sommets du polygone, et P la projection de la sphère sur le plan du polygone. Pour chaque sommet, on calcule le côté adjacent, ainsi que le vecteur reliant P a ce

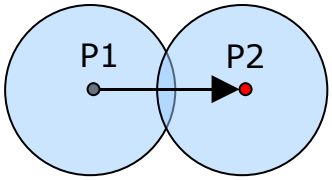
sommet. Rappelez-vous que le produit scalaire de deux vecteurs permet aussi de calculer la projection du premier vecteur sur le deuxième.

Donc, il faut calculer, puis normaliser les 3 vecteurs S1-S2, S2-S3 et S3-S1. Ensuite, on calcule les vecteurs P-S1, P-S2 et P-S3, mais on ne les normalise pas. On calcule le produit scalaire entre S1-S2 et P-S1, S2-S3 et P-S2, puis S3-S1 et P-S3. Le résultat donne la projection des vecteurs P-Sx sur chacun des côtés du polygone. De cette façon, on peut calculer le point de projection sur chaque côté du polygone. Ensuite, on calcule la longeur entre chaque point projeté et le point P. On retient la projection la plus courte, et on teste si la sphère de collision englobe le point de projection. Si c'est le cas, le polygone est en collision avec la sphère.

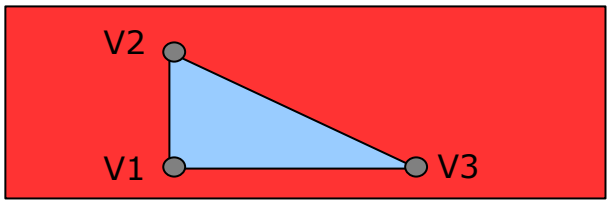
2.1.5 – Le procédé complet

Maintenant que vous connaissez les différentes techniques, nous pouvons analyser l'ensemble du scénario d'une collision. En même temps, je vous indiquerai où regarder dans le code source pour faciliter la compréhension du programme.

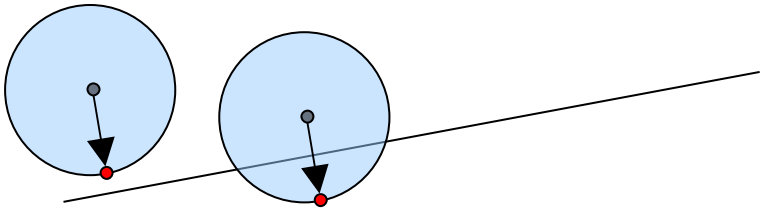
Phase 1 : Le joueur commande un déplacement, en avant ou en arrière, c'est égal, dans l'environnement 3D. Dans le code source, la fonction ReadKeyboard, à la ligne 788 du fichier application.cpp a été appelée, et DirectInput vient de signaler qu'une touche de déplacement a été pressée, aux lignes 850 ou 866. On calcule donc une nouvelle position, puis, on appelle la fonction de test CheckNextPosition, aux lignes 862 ou 878. Rendez-vous donc au début de la fonction de test, en ligne 650. On y crée une boucle qui teste tous les murs du terrain de jeu, en ligne 675, puis on obtient les données des différents polygones aux lignes 678 à 683. En ligne 693, on applique la matrice de transformation locale au polygone, pour que l'on teste tous les objets dans le même système d'axes. Enfin, en ligne 701, on démarre le test de collisions à proprement parler.



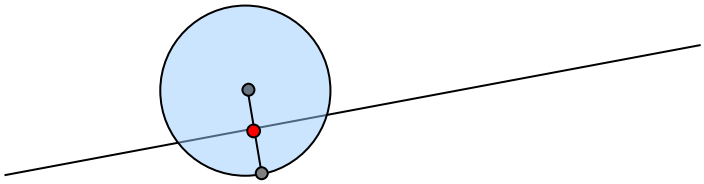
Phase 2 : On crée un plan à partir du polygone à tester. Dans le code, la fonction GetTriSphereCollision, du fichier e_collisions.cpp, vient d'être appelée. A la ligne 23, on commande à la fonction GetPlane, de l'objet E_Polygon, de calculer, puis de retourner le plan. L'objet E_Polygon va demander à Direct3D de calculer le plan, à la ligne 96 du fichier E_Polygon.cpp.



Phase 3 : On calcule si la sphère de collision du joueur touche le plan. Le centre de la sphère correspond à la prochaine coordonnée du joueur, celle que nous devons tester. On commence par calculer de quel côté du plan se trouve le joueur, à la ligne 26 du fichier e_collisions.cpp, puis, en fontion de la réponse, on calcule la direction du plan par rapport au joueur, à la ligne 33. Depuis le centre de la sphère de collision, on trace une droite en direction du plan, à l'aide de la direction obtenue, et on regarde à quel coordonnée de l'espace la droite croise le bord de la sphère, en ligne 48. On calcule ensuite la distance au plan de ce point, ainsi que la distance au plan du centre de la sphère, aux lignes 26 et 54, puis on teste de quel côté du plan se trouvent les points, en ligne 60. Ici a lieu la première élimination : Si les points sont du même côté du plan, alors la sphère ne peut pas traverser celui-ci. Si c'est le cas, le polygone à tester ne peut pas être en collision avec le joueur.



Phase 4 : On calcule où la sphère de collision touche le plan. Maintenant, on sait que la sphère de collision traverse le plan du polygone. Nous avons un segment de droite, qui part du centre de la sphère en direction du plan, et nous savons que ce segment croise le plan. A la ligne 79 du fichier e_collisions.cpp, nous demandons à Direct3D de calculer la coordonnée de l'espace où ce croisement a lieu. Puisque ce point est forcément sur le plan du polygone, on peut tester si celui-ci se trouve à l'intérieur des limites du triangle dessiné par le polygone sur le plan.



Phase 5 : On applique l'algorythme de calcul des collision décrit au chapitre 2.1.3. Celui-ci est écrit dans la fonction IsPointIntoTriangle, à la ligne 241 du fichier e_collisions.cpp. On commence par calculer et normaliser les 3 segments de droite reliant le point à tester aux sommets du polygone, de la ligne 231 à la ligne 247. Puis, de la ligne 254 à la ligne 256, on demande à Direct3D de calculer le produit scalaire de chaque couple de droites. On additionne les angles à la ligne 261, puis on teste le résultat à la ligne 313. Si la somme des angles fait 6,28 (on travaille en radians, pas en degrés, c'est pour cela que la somme à tester fait 6,28 et non pas 360), alors le joueur est en collision avec le polygone.

Phase 6 : Si la collision précédente est négative, **on teste la collision avec le bord du triangle, comme décrit au chapitre 2.1.4**. On commence par appeller la fonction ClosestPointOnTriangle, à la ligne 102 du fichier Collisions.cpp. Celle-ci va calculer la projection du point de test sur chacun des côtés du polygone, puis retourner le point le plus proche. Si l'appel à la fonction IsPointIntoSphere, à la ligne 106, est positif, alors la sphère est en

collision avec le polygone.

2.2 – Le sliding

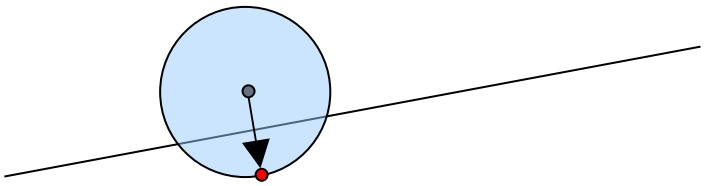
2.2.1 – Qu'est ce que le sliding

Dans la démo, si vous être en mode « collision et Sliding » vous pouvez remarquer que lorsque le personnage virtuel entre en collision avec un mur, celui-ci ne le bloque pas complètement, mais on a plutôt l'impression que le personnage glisse le long du mur. Cet effet est appelé Sliding.

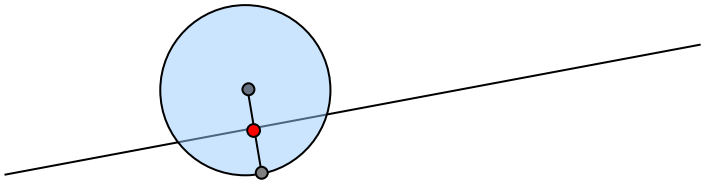
2.2.2 – Comment calculer un sliding

Un sliding ne devrait être calculé que si le joueur est en collision avec le polygone, sinon cela n'a pas de sens. Pour le calcul, nous avons besoin de la coordonnée du joueur, du plan du polygone pour lequel le sliding doit être calculé, et du rayon de la sphère de collision. La fonction qui calcule le Sliding dans la démo s'appelle GetSlidingPoint, et elle se trouve à la ligne 127 du fichier e_collisions.cpp.

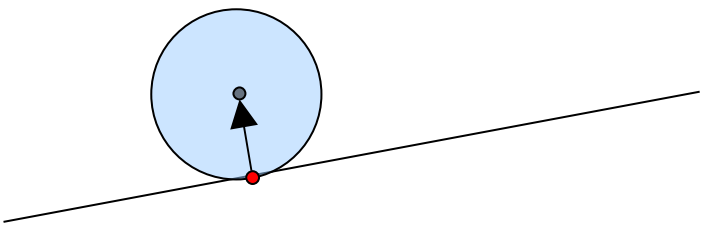
On commence par calculer la distance entre la coordonnée du joueur et le plan. Si cette valeur est négative, on inverse le plan. C'est ce que nous faisons aux lignes 133 et 136. Ensuite, à la ligne 146, on calcule la longeur du rayon de la sphère de collisions, en direction du plan. Puis, en additionnant cette longeur à la coordonnée du joueur, on calcule le point qui se trouve derrière le plan, sur la sphère de collisions, comme vous pouvez le voir à la ligne 153 du code source.



On demande ensuite à Direct3D de calculer la coordonnée ou le segment de droite coordonnée du joueur – point après le plan croise le plan, à la ligne 161.



Puis, à la ligne 168, on additionne le rayon de la sphère de collision au point de croisement du segment de droite et du plan, et on retourne la réponse.



2.3 – Conclusions

2.3.1 – Pourquoi cet algorythme n'est pas parfait

Dans la démo, tout semble fonctionner à merveille. Il n'y a pas de bugs avec les collisions, aucun obstacle ne laisse passer le joueur, on peut glisser le long des murs, tout semble parfait.

Pourtant, cet algorythme n'est pas utilisable tel quel pour un grand projet. Pourquoi ?

Parce-que l'algorythme est dépendant du fait que le déplacement du joueur ne doit pas être plus grand que le diamètre de la sphère de collision. C'est une astuce pour simplifier les calculs, mais il y a beaucoup d'inconvénients. En voici quelque-uns :

- On ne peut pas calculer un déplacement en fonction du temps écoulé, puisque ceci implique des distances variables. Donc, si le processeur n'est pas immédiatement disponible pour les calculs, la vitesse de déplacement peut varier, parfois de façon très gênante.
- On ne peut pas utiliser de vecteurs de force. Pas de gravité, par exemple.
- On est obligé de calculer le point suivant avant de savoir s'il est valide.
- Le Sliding ne peut que corriger approximativement un point. Pas de gestion de plans multiples, pas de correction de la vitesse en fonction du plan, donc pas de possibilité d'utiliser des lois phisiques, comme la gravité, ou plus simplement pas de possibilité de monter sur un objet, ou de monter un escalier, par exemple.

Ce qui est intéressant ici, c'est le principe du moteur de collisions. Deux fonctions, en particulier : IsPointIntoTriangle et ClosestPointOnTriangle, qui sont des fonctions utilisables telles quelles pour un moteur de collisions de plus grande envergure. Ce sont des algorythmes standards, d'une efficacité reconnue.

Pour le reste, il faudrait prendre le problème sous un autre angle. On ne devrait pas tester uniquement le point final, mais plutôt calculer la trajectoire du joueur, et tester s'il y a une collision possible sur le trajet. Donc partir d'un point et d'une vitesse pour le calcul des collisions, et non pas uniquement du point final et de la sphère de collision. Pour ma part, j'espère que cette démo donnera envie à d'autres d'aller plus loin.

2.3.2 – Remarques sur le moteur

Malgré ses défauts, le code source de cette démo peut constituer un point de départ pour un projet plus sérieux. Comme je l'ai cité précédemment, certains algorythmes utilisés ici se retrouvent dans d'autre moteurs de collisions, de plus grande envergure. La détection de collisions fonctionne bien, c'est la technique pour y arriver qui est à corriger. Vous devrez également apporter un simulateur de lois physiques à votre moteur, pour le rendre vraiment complet. A ce moment, vous devrez explorer une nouvelle utilisation du mot vecteur. Amateurs de gros projets, à vos claviers...

Pour ma part, je mets le code source ci-joint à disposition de qui souhaite l'utiliser. N'oubliez pas qu'il n'est pas utile de réinventer la roue. Si une partie de ce code convient à l'un de vos projets, n'hésitez pas à l'utiliser. Toutefois, je vous saurais reconnaissant de bien vouloir citer vos références, si vous créez un projet de grande envergure, et que vous vous aidez de ce code source pour avancer. Déposer un vote et/ou un commentaire serait également bienvenu, si cette démo vous a permis d'avancer.

2.3.3 – Références de l'auteur

Jean-Milost Reymond

Développeur pour la société ProcessSoft.

Jean_Milost@hotmail.com