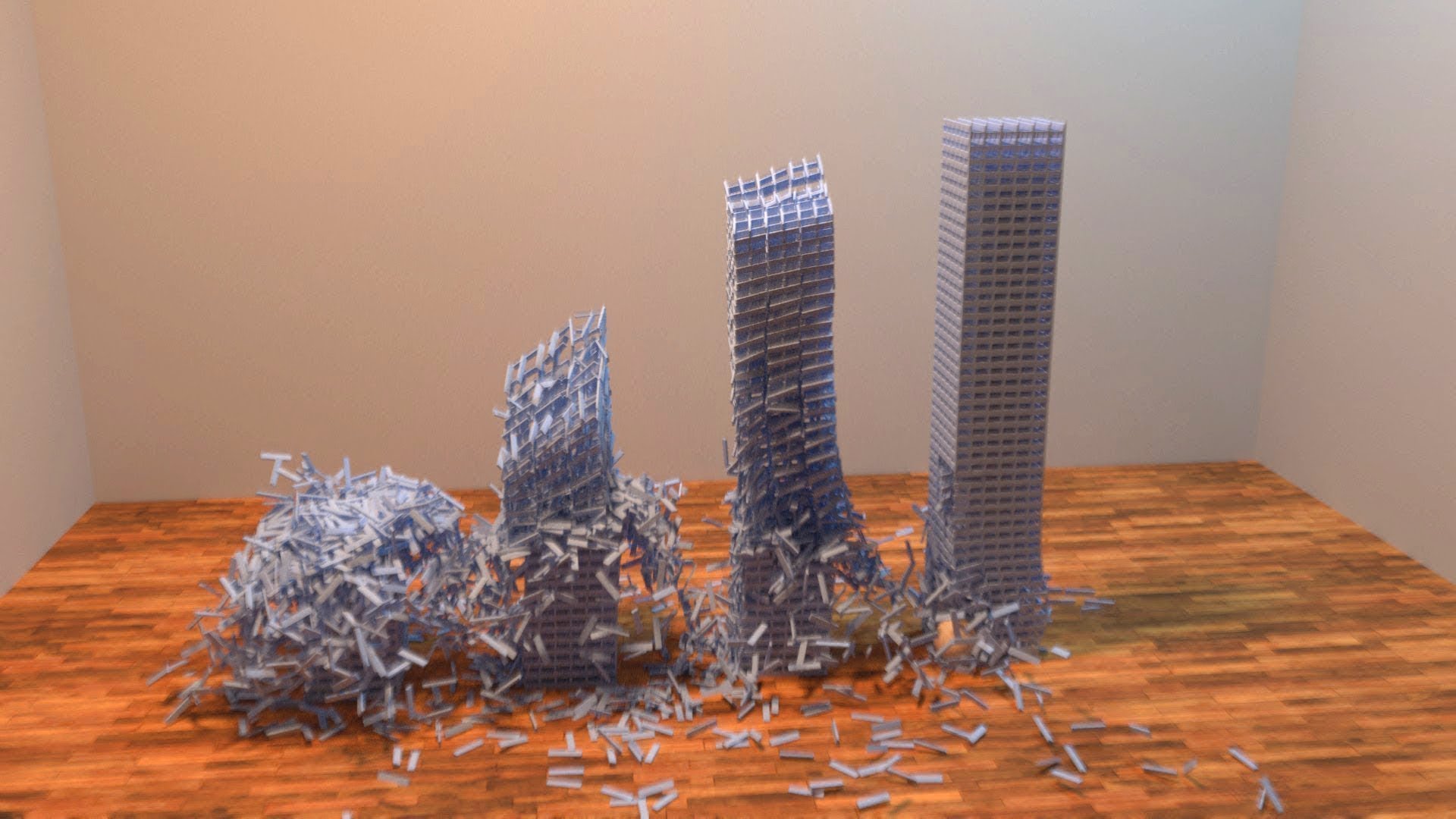
ISEP P1B   2015-2016

## Travail d’initiative personnelle encadré :

## Moteur physique



Adrien Grouard de Tocqueville

Grégoire Fessard

Charles Boissy

**Sommaire**

Introduction

1. Détection d’intersection être polyèdres convexes
   1. Mise en place de la structure
      1. Solides étudiés
      2. Solides indéformables
   2. Implémentation de GJK
      1. Différence de Minkowski
      2. Fonction de support
      3. Définition du simplex
      4. Fonction containsOrigin()
   3. Implémentation de l’EPA
      1. Agrandissement du polytope
      2. Génération des informations de contact
2. Dynamique des corps solides sous l’effet de forces de contrainte
   1. Définition d’une contrainte
      1. Les contraintes d’égalité
      2. Les contraintes d’inégalité
   2. Détermination du multiplicateur de Lagrange
   3. Résolution des systèmes de contraintes
3. Rotations
   1. En deux dimensions
   2. En trois dimensions
      1. Les angles d’Euler et le blocage de cardan
      2. Les matrices de rotation
      3. Les quaternions
4. Caractéristiques des solides utilisés
   1. Sphère
   2. Cube
   3. Cylindre
   4. Cône

Conclusion

# Introduction

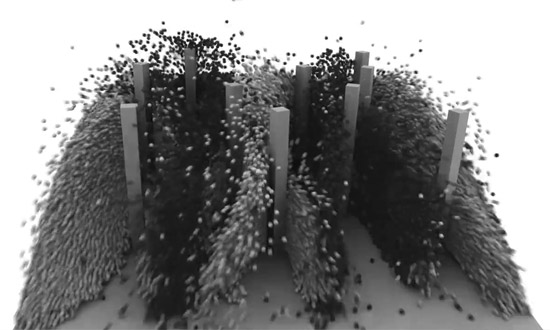
Invisibles, les phénomènes physiques qui régissent le fonctionnement de notre univers sont pourtant omniprésents et perceptibles. Etudiées depuis longtemps, les lois les décrivant ont été à maintes reprises redéfinies et affinées et il est de nos jours indispensable de pouvoir les modéliser de façon fidèle ou tout du moins d’être capable de les approximer de façon réaliste.

Le modèle informatique simulant la plupart des facteurs de la mécanique classique (gravité, fluide, contraintes,…) est appelé moteur physique. C’est une bibliothèque intégrable à un programme qui permet la résolution des équations physiques. Typiquement, il permet de détecter les collisions entre plusieurs solides et de calculer leurs réponses.

Suite à l'essor des jeux vidéo en tant que loisirs de masse, les développeurs sont à la recherche d'univers toujours plus immersifs. Pour cela, ils souhaitent modéliser le plus fidèlement possible les phénomènes naturels. La course à la performance passe avant tout par la technique et donc par le moteur physique (PhysX, Havoc pour ne citer que les plus populaires) qui est d’ailleurs devenu récemment un commerce à part entière. Le moteur physique peut également trouver des utilités dans la robotique ou dans la simulation physique plus généralement.  
Le projet que nous avons réalisé pendant la fin de l’année est un moteur physique simple programmé en C++. La partie graphique utilise l’API OpenGL.

Le moteur complet est disponible à l’adresse : <https://github.com/AdrienDeTocqueville/TIPE>  
Une sélection des fichiers les plus importants ainsi qu’un fichier exécutable est disponible ici : <https://github.com/AdrienDeTocqueville/Simple>

Pour la présentation orale, nous avons utilisé une version simplifiée du moteur pour réaliser une simulation de fluide en l’assimilant à un nombre très élevé de billes de petite taille. Cela permettra de créer un sablier virtuel et de montrer, en calculant le débit, qu’un sablier réel fonctionne.

  
Image tirée d’un autre programme

I. Détection d'intersection entre polyèdres convexes

L'objectif de cette partie va être l’implémentation en C++ d'un algorithme de détection de collisions pour permettre une réponse réaliste à celles-ci par le moteur physique. Cet algorithme doit être capable d’obtenir les informations relatives à l'intersection telles que la normale au plan de collision, les points de contacts et l'interpénétration.

Dans un souci de compréhension des méthodes et techniques employées, nous procéderons d’abord à une explication théorique puis nous parlerons des détails d'implémentation.

L’enjeu d'un algorithme de détection de collision est de fournir une réponse précise le plus rapidement possible. En effet, la boucle de mise à jour ayant beaucoup de tâches à accomplir, il est crucial d'économiser du temps de calcul dès que possible.

Nous présenterons quelques solutions efficaces à ce problème:

- L’utilisation d'AABB (Axis Aligned Bounding Boxes) pour éliminer rapidement certaines paires.

- L'exploitation de la cohérence temporelle. Utiliser les résultats précédents permet souvent d'accélérer le processus.

Cependant, il faut trouver un équilibre entre la précision de la détection et le temps imparti à celle-ci. Des résultats erronés donneront lieu à des défauts de simulation.

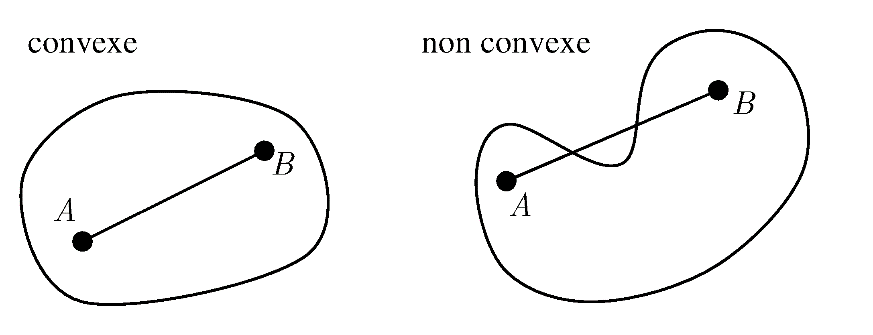
De manière évidente, le type de formes utilisées par le moteur aura un impact direct sur la quantité de calculs. On utilisera ici des formes géométriques simples.

1. Mise en place de la structure

1. **Solides étudiés**

Afin de garder l'algorithme performant et robuste nous ne considèrerons que des formes géométriques convexes telles que des boules, des boites, des cônes et des cylindres. On pourra toutefois décomposer un polyèdre concave en union de parties convexes.

Définition : Un solide convexe désigne un solide qui contient tout segment reliant deux de ses points ou bien de manière plus grossière un solide qui n’a aucun « trou » ou « creux ».



Ci-dessus un solide convexe (à gauche) et un solide concave (à droite)

1. **Solides indéformables**

Le moteur physique ne gérera que des solides indéformables (pas de solides mous ni de fluides). Plus précisément, un solide indéformable est tel que la distance entre toute paire de points du solide est constante en fonction du temps.

Les chocs seront donc élastiques la masse et l'énergie cinétique seront conservés. Les phénomènes de déformations seront donc négligés.

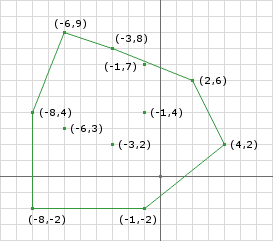
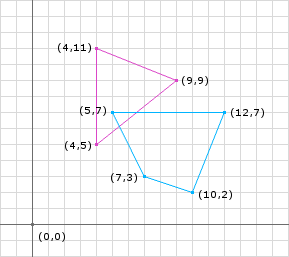
2. Implémentation de GJK (Gilbert-Johnson-Keerthy)

1. **Différence de Minkowski**

L’algorithme GJK se base au départ sur la différence de Minkowski. L’idée est de calculer la différence entre deux polyèdres pour obtenir un nouveau polyèdre (qui sera également convexe) afin de vérifier si l’origine du repère (vecteur nul) est à l’intérieur de ce solide, c’est-à-dire être sûr qu’il y ait au moins un point commun. Si cette condition est vérifiée, alors les deux solides sont en collision et dans le cas contraire, il n’y a pas de collision.

Soient A et B deux formes convexes en 3D définies par des ensembles de points. C est la différence de Minkowski de A et B.

et



*Dans cet exemple en* 2D, on peut voir deux polygones convexes à gauche qui s’intersectent et le polygone représentant la différen*ce de Minkowski à droite (on note qu’il contient l’origine).*

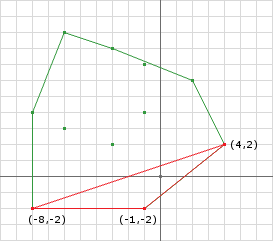
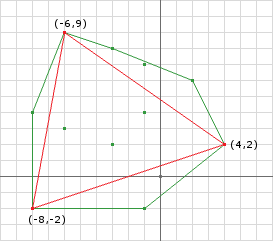
1. **Fonction de support**

Une fonction de support renvoie le point d’une forme convexe le plus éloigné selon un axe. Autrement dit, elle donne le point dont le produit scalaire avec l’axe est maximum.  
Celle-ci à des propriétés remarquables sur la différence de Minkowski et va nous permettre de ne pas la calculer mais de pouvoir obtenir la position de certains de ses points. En effet

Il suffit donc de définir une fonction de support pour chacune des formes que l’on veut utiliser (boite, boule, cône, …).

1. **Définition du simplex**

On désire maintenant établir un polygone encerclant l’origine à l’intérieur de la différence de Minkowski, le simplex. En effet, comme on ne veut pas calculer quel point de cette différence est égal à l’origine (bien trop couteux en opérations et calculs), on va procéder itérativement à la construction du simplex pour englober l’origine, si celui-ci contient ce point, alors il y a intersection des solides donc collision.



A gauche un simplex de 3 points (triangle) et à droite un autre simplex pour le même polygone mais cette fois-ci encerclant l’origine*.*

L’algorithme va débuter avec n’importe quel axe de recherche. La première étape sera de calculer le point de support selon cet axe. On vérifie ensuite que, sinon on peut en déduire que, puisque le point le plus loin selon l’axe de recherche ne dépasse pas l’origine, elle ne peut pas se trouver dans la différence. On ajoute ce point au simplex avant de passer à l’itération suivante. L’algorithme s’arrêtera soit quand il trouvera un point de support qui ne dépasse pas l’origine (renverra), soit quand il détectera que l’origine se trouve dans le simplex (renverra. Ceci est réalisé par la fonction. Cette fonction déterminera également l’axe de recherche pour l’itération suivante.

1. unsigned steps = 0;
3. **while** (steps++ < 50)  // Evite les boucles infinies
4. {
5. axis = normalize(axis);
7. supportPoint = support(a, b, axis);
8. **if** (dot(supportPoint.p, axis) <= 0) // Le point ne dÃ©passe pas lâ€™origine
9. **return** **false**;
11. \_simplex.push\_back(supportPoint);
13. **if** (containsOrigin(\_simplex, axis))
14. **return** **true**;   // il y a collision
15. }
17. **return** **false**;
18. **La fonction containsOrigin**

Comme son nom l’indique, la fonction containsOrigin va permettre de déterminer si le simplex formé contient l’origine et donc de conclure si il y collision ou non. Si les tests n’aboutissent pas alors la boucle principale de GJK continuera de tourner.

A chaque étape, le point A représentera le point le plus récemment ajouté au simplex.

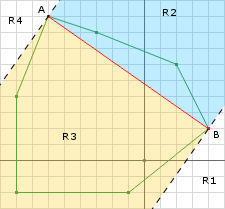
1. Point

A la première itération, le simplex ne contiendra forcément pas l’origine (à cause de la précision limitée de l’ordinateur et des arrondis). On prend pour nouvel axe l’opposé du point de support.

2. Ligne

De la même manière, l’origine ne peut pas se situer sur une ligne.

Pour déterminer la nouvelle direction, on va utiliser les régions de Voronoi, c’est-à-dire un découpage du plan en plusieurs zones comme représenté ci-dessous :



Régions de Voronoi d’une ligne

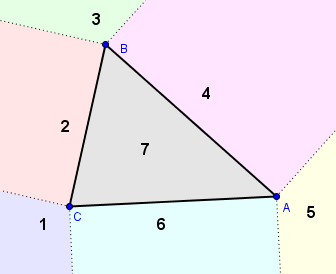
On veut savoir dans quelle région se trouve l’origine. Déjà, les régions R1 et R4 peuvent être éliminées immédiatement car on sait que A et B sont aux bords de la différence de Minkowski et que d’après les tests précédents (au premier et deuxième tour), l’origine ne peut pas se trouver dans ces régions.

On prend alors comme nouvelle direction la normale au segment dans le plan qui point vers l’origine. La nouvelle direction s’écrit alors grâce au triple produit vectoriel :

3. Triangle

Encore une fois, l’origine ne peut se situer dans le triangle. Il n’y a donc qua déterminer l’axe de recherche.

Le triangle ABC formé peut être représenté par sept régions de Voronoi de la manière suivante :



Régions de Voronoi d’un triangle

Par la suite, on notera le vecteur normal au triangle.

On sait que l’origine ne se trouve pas dans la région 5 car le point de support a dépassé l’origine. De plus, les itérations précédentes permettent d’affirmer que l’origine ne se trouve pas dans les régions 1, 2 et 3.

Pour les régions 6 et 4, on procède respectivement comme suit pour savoir de quel côté se trouve l’origine : si ( (6) ou si (4)

On supprime donc le point qui ne sert à rien au simplex, c’est-à-dire B pour la région 6 ou le point C pour la région 4. On a plus qu’une ligne donc on peut définir le nouvel axe de la manière que dans le cas précédent :

( (6) ou (4)

Si l’origine ne se trouve dans aucune de ces 2 régions, alors on regarde si elle se situe au-dessus du triangle en vérifiant si le produit scalaire suivant est strictement positif : puis on prend comme nouvelle direction.

Sinon, cela veut dire que l’origine se situe en-dessous du triangle. On inverse donc les points B et C et on prend pour nouvel axe – .

4. Tétraèdre

Lorsque le simplex possède quatre points, on obtient un 3-simplex appelé plus communément tétraèdre. Il est composé de quatre faces triangulaires.

On définit cette fois-ci quatre vecteurs :

, , , et ainsi que les normales des triangles ABC, ADB, et ACD respectifs :

= (ABC) (ADB) et = (ACD)

Afin de déterminer si l’origine est dans le tétraèdre, on va projeter celle-ci sur les différents plans formés par les faces. Si l’origine se trouve derrière chacune des faces, elle est dans le tétraèdre. Si elle est devant une face, on retire le point qui ne fait pas partie de cette face et on procède au test sur le triangle pour déterminer la direction.

On va donc effectuer des tests sur chaque face du tétraèdre pour savoir devant laquelle se trouve l’origine. Pour y parvenir, on regarde si le produit scalaire de la normale au triangle considérée avec le vecteur partant de A jusqu’à l’origine est supérieur ou égale à 0, c’est-à-dire si la normale et le vecteur sont dans la même direction. cross() calcule le produit vectoriel et dot() le produit scalaire.

1. vec3 ABC = cross(AB, AC); // Vecteur normal au triangle ABC
3. **if** (dot(ABC, AO) >= 0.0f)   // Devant ABC
4. {
5. \_simplex.erase(\_simplex.begin() + 0); // On enleve D
6. **return** checkTriangle(ABC, AO, AB, AC, \_simplex, \_axis);
7. }

10. vec3 ADB = cross(AD, AB); // Vecteur normal au triangle ADB
12. **if** (dot(ADB, AO) >= 0.0f)   // Devant ADB
13. {
14. \_simplex.erase(\_simplex.begin() + 1); // On enleve C
15. **return** checkTriangle (ADB, AO, AD, AB, \_simplex, \_axis);
16. }

19. vec3 ACD = cross(AC, AD); // Vecteur normal au triangle ACD
21. **if** (dot(ACD, AO) >= 0.0f)   // Devant ACD
22. {
23. \_simplex.erase(\_simplex.begin() + 2); // On enleve B
24. **return** checkTriangle (ACD, AO, AC, AD, \_simplex, \_axis);
25. }

28. **return** **true**;    // L'origine est a l'interieur du tetraedre
29. }

Si tous les tests sont négatifs, alors on peut en déduire que l’origine se trouve dans le tétraèdre et donc qu’il y a intersection.

3. Implémentation de EPA (Expanding Polytope Algorithm)

L’algorithme EPA est complémentaire de GJK, en effet, il permet d’obtenir les informations essentielles lors de la détection de collision pour déterminer la réponse adaptée telles que le vecteur d’interpénétration ou encore le point de contact.

Pour cela, on va réutiliser le simplex formé par GJK et l’agrandir (il n’y a plus de limite de points, on aura donc un polytope) pour trouver le point de la différence de Minkowski le plus proche de l’origine.

1. **Agrandissement du polytope**

Tout d’abord, on va créer un polytope à partie des points du simplex. On gardera les faces dans une liste triant les éléments en fonction de leur distance à l’origine. L’ordre des points est important puisqu’il déterminera la direction de la normale.

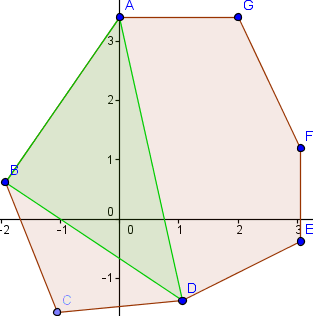
1. Face::simplex = &\_simplex;
3. std::multiset<Face, Closer> faces;
4. std::multiset<Face, Closer>::iterator closest;
5. faces.insert(Face(3, 2, 1)); // ABC
6. faces.insert(Face(3, 1, 0)); // ACD
7. faces.insert(Face(3, 0, 2)); // ADB
8. faces.insert(Face(2, 0, 1)); // BDC

Notre but va être de trouver la face du polytope la plus proche de l’origine afin d’ajouter un nouveau point à notre polytope ce qui le fera évoluer puis de réitérer cette opération jusqu’à ce qu’il ne soit plus possible de le faire, a ce stade, on obtiendra le polytope final.

On commence donc par une boucle while en définissant préalablement les points supports a et b qui vont nous servir par la suite pour trouver le point support (dans la différence de Minkowski) dans la direction de la normale au triangle et ainsi peut être agrandir notre polytope.

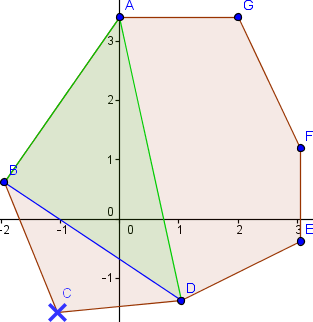
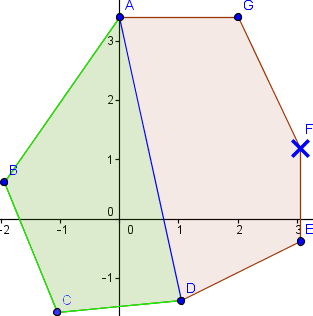
1. **static** **double** maxSteps = 20;   // Evite les boucles infinies
2. unsigned steps = 0;
3. **float** distance;
5. **while** (steps++ < maxSteps)
6. {
7. closest = faces.begin();   // Donne la face la plus proche
9. /// Recherche du point de support
10. Point supportPoint = support(\_a, \_b, closest->normal);
11. distance = dot(supportPoint.p, closest->normal);

Pour visualiser de manière intuitive la démarche suivie, prenons l’exemple d’un simplex en 2D :



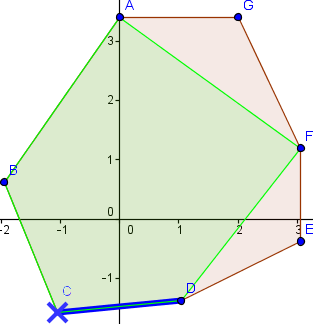
Le triangle vert est notre simplex et le polyèdre rouge représente la différence de Minkowski.

On veut trouver le segment le plus proche de l’origine qui est ici BD et appeler la fonction support afin de trouver le point support dans la direction de la normale à BD, on obtient :

*C est le point support trouvé On supprime donc le segment BD et on trace les segments BC et DC dans la direction de la normale puis on répète le processus en trouvant F comme nouveau point support*

Enfin, après une dernière opération, on trouve que notre prochain point de support est C, or celui fait déjà parti du polytope, par conséquent, on ne peut plus agrandir le polytope.



*Le polytope n’a plus besoin d’être agrandi, on met fin à la boucle*

Après chaque appel de la fonction de support, on regarde si la distance de la face la plus proche est égale à la distance entre le point de support et l’origine à epsilon près. Si le test s’avère positif on sort de la boucle et on génère les informations de contact que l’on décrira dans la prochaine sous-partie.

1. /// Test de la condition de terminaison
2. **if** (epsilonEqual(distance, closest->distance, EPSILON))
3. **break**;

Si ce n’est pas le cas, on va essayer d’agrandir notre polytope actuel en ajoutant et supprimant des faces. Cette opération est plus difficile qu’en deux dimensions puisqu’il faudra supprimer toutes les faces pour lesquelles le point de support est situé devant.

Cependant, lorsqu’on va rajouter les nouvelles faces, il faudra faire attention à ne pas rendre le polytope concave. On va donc conserver une liste des arrêtes retirées dans laquelle chaque arrête n’apparait qu’une seule fois.

1. **if** (dot(faceIt->normal, supportPoint.p - \_simplex[faceIt->a].p) >= 0.0f)
2. {
3. // On garde en mÃ©moire les arrÃªtes retirÃ©es
4. // Si il y en a une en double, on la retire de la liste
5. addEdge(edges, Edge(faceIt->a, faceIt->b));
6. addEdge(edges, Edge(faceIt->b, faceIt->c));
7. addEdge(edges, Edge(faceIt->c, faceIt->a));
9. // On supprime cette face de la liste
10. faceIt = faces.erase(faceIt);
11. }
12. **void** addEdge(std::vector<Edge>& edges, Edge b)
13. {
14. **for** (unsigned i(0) ; i < edges.size() ; i++)
15. {
16. **if** (edges[i].first == b.second && edges[i].second == b.first) // L’ordre des points sera inversé
17. {
18. edges.erase(edges.begin() + i);    //supprime l’arrête en double
19. **return**;
20. }
21. }
23. edges.push\_back(b);    // Ajoute l’arrête seulement si on ne l’a pas trouvée
24. }

On peut maintenant ajouter le nouveau point au simplex et les faces au polytope.

1. // Construction des nouvelles faces
2. \_simplex.push\_back(supportPoint);   // Ajout du point support
3. unsigned spIndex = \_simplex.size() -1;
4. **for** (auto& edge: edges)
5. faces.insert(Face(spIndex, edge.first, edge.second));
6. **Génération des informations de contact**

Une fois que l’on a trouvé la face du polytope la plus proche de l’origine, on peut chercher le point le plus proche. Ce point est le projeté de l’origine sur la face. Cependant, ce point appartiendra à la différence de Minkowski, on va donc déterminer les coordonnées barycentrique du projeté de l’origine sur la face de la différence. Lorsque l’on calculait le point de support, on enregistrait en plus dans la variable supportPoint le point de support sur A et celui sur B. Grace à ceci, on va pouvoir retrouver les points de la face sur A et celle sur B qui ont permis d’obtenir les points de la face la plus proche de l’origine sur la différence de Minkowski.

1. **struct** Point
2. {
3. Point(vec3 \_supA, vec3 \_supB):
4. p(\_supA - \_supB), supA(\_supA), supB(\_supB)
5. { }
7. vec3 p; // point sur la différence de Minkowski
8. // On garde en mémoire les points de support sur A et B
9. vec3 supA, supB;
10. };
11. vec3 lambdas; // CoordonnÃ©es barycentriques du projeté de l'origine sur la face la plus proche de l’origine
12. getBarycentricCoordinates(distance \* closest->normal, \*closest, lambdas);    // Fonction dont on ne détaillera pas le fonctionnement
14. vec3 A1 = \_simplex[closest->a].supA, A2 = \_simplex[closest->b].supA, A3 = \_simplex[closest->c].supA;
15. vec3 B1 = \_simplex[closest->a].supB, B2 = \_simplex[closest->b].supB, B3 = \_simplex[closest->c].supB;
17. Manifold\* man = **new** Manifold();
18. // On utilise les coordonnés pour trouver un point sur A et B
19. man->points[0] = lambdas.x\*A1 + lambdas.y\*A2 + lambdas.z\*A3;
20. man->points[1] = lambdas.x\*B1 + lambdas.y\*B2 + lambdas.z\*B3;
22. man->normal = closest->normal;   //normale au plan de collision
23. man->penetration = distance;     //distance de pénétration
24. computeBasis(man->normal, man->u, man->v); // Pour les frottements

Ainsi, on regroupe les informations clés liées à la collision dans une instance de la classe Manifold qui sera donné à l’algorithme de résolution des collisions.

La fonction computeBasis prenant en argument trois vecteurs a, b, c d détermine les vecteurs b et c en fonction de a de telle sorte que (a, b, c) soit une base de.

II- Dynamiques des corps solides sous l’effet de forces de contraintes

L’objectif de cette partie est de réussir à trouver un modèle physique réaliste qui sera capable de gérer les interactions entre les solides. Pour cela, nous utiliserons des contraintes. Elles auront pour effet de créer des restrictions sur le mouvement des solides. Des exemples de contraintes sont les joints, les frottements ou les contraintes de non pénétration. Ces dernières seront celles utilisées pour résoudre les collisions, en forçant les objets à modifier leur trajectoire de manière réaliste.

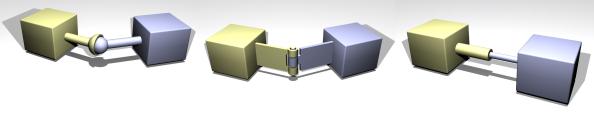
Nous présenterons d'abord les contraintes d'égalité puis celles d'inégalités dans le cadre d'une simulation en temps réel. Nous verrons ensuite comment utiliser les contraintes pour déterminer les positions des solides à chaque pas de la simulation.

# 1. Définition d’une contrainte

Les contraintes sont des règles qui doivent être satisfaites pendant la simulation. En d'autres termes, une contrainte supprime un ou plusieurs degrés de liberté au mouvement d'un solide (jusqu'à 6 en 3 dimensions).

À chaque tour de la boucle de simulation, on établit une liste des contraintes non satisfaites et on applique des impulsions pour corriger les erreurs.

En pratique, une contrainte est définie par une fonctionprenant en argument l'état des solides impliqués (i.e. la position et l'orientation) et renvoyant un réel. En fonction de cette valeur, la contrainte sera satisfaite ou non. On appliquera aux solides des impulsions dans le but de garder la valeur de C dans un certain intervalle.



*Trois différents types de contraintes*

1. **Les contraintes d’égalité**

Une partie des contraintes est appelée contrainte d’égalité. Ce sont celles qui ne sont satisfaites que lorsque C vaut zéro. Par conséquent, le moteur physique devra garder C le plus proche possible de 0. Ce type de contraintes est utilisé lorsque la position ou la vitesse d’un point doit être exactement égale à une valeur définie.

La contrainte de distance

On considère une particule en deux dimensions placée au point et ayant pour vitesse.

On définit une contrainte d’égalité C sur la position de la particule par :   
 renvoie la distance entre l’origine et la particule lorsque la particule est à une distance de l’origine. Par conséquent, cette contrainte va garder la particule à une distance constante de l’origine, tel un pendule rigide.

Détermination de la force corrective

La solution la plus simple serait de modifier la position de sorte que la contrainte soit satisfaite. Cependant, cela résulterait en un comportement irréaliste. Pour conserver le plus petit possible, une approche intuitive est la dynamique des forces. En effet, on appliquera aux solides des forces pour satisfaire les contraintes dans le respect des lois du mouvement de Newton.  
Pour calculer ces forces, on doit savoir quelles sont les accélérations et vitesses autorisées par la contrainte. On dérivera donc par rapport au temps et on s’assurera quesoit égal à zéro également.

Il n’est pas nécessaire de dériver une deuxième fois car l’utilisation des impulsions permet l’application des lois de Newton sans passer par l’accélération.

On obtient

On définit alors la matrice Jacobienne de la contrainte par

On a

Donc le vecteur représenté par la matrice, noté, est perpendiculaire à la vitesse.

Or d’après le principe de d’Alembert, l’ensemble des forces de contrainte appliquées à un système ne travaille pas lors d’un déplacement virtuel.

Donc, on a donc que la force de contrainte est parallèle à.

On note le scalaire, appelé multiplicateur de Lagrange, tel que

Cependant, dans le cas général la contrainte de vitesse ressemblera à ceci :

Oùest un facteur de redressement. Si ce terme est non nul, la force de contrainte travaillera. Cela est parfois nécessaire, par exemple pour des contraintes motrices, mais surtout pour corriger les éventuelles imprécisions. En effet, quelques erreurs peuvent s’introduire lors de la mise à jour des positions des solides. Ce problème est appelé la dérive de position. On peut donc ajourer un termedans la contrainte de vitesse pour corriger le problème. L’erreur de la contrainte de position est mesurée grâce à la fonction. Si on veut réduire cette erreur à zéro pendant l’intervalle de temps, la vitesse nécessaire à ajouter est. Cependant, nous ne voulons pas que celle-ci soit éliminée en une seule fois, la vitesse nécessaire à ajouter est :

Où est le facteur de stabilisation. Ce terme définit le montant de l’erreur qui sera corrigé à chaque pas de la simulation. Ce type de correction est appelé stabilisation de Baumgarte. Dans le programme, on utilise

La dérivation de la contrainte permet d’obtenir sa matrice Jacobienne. Ceci est fait à la main pour chaque contrainte. Le but du moteur sera de déterminer le multiplicateur de Lagrange pour chaque contrainte, à chaque mise à jour.

Le calcul du multiplicateur de Lagrange se fera dans la seconde sous partie.

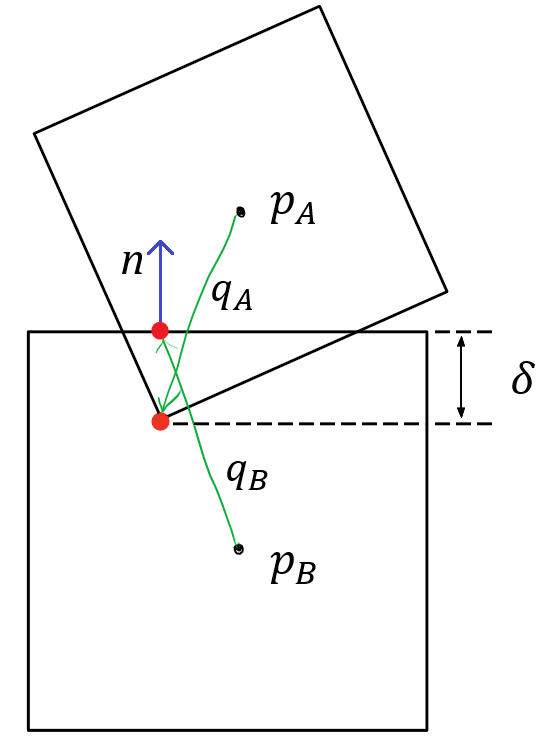
1. **Les contraintes d’inégalité**

Un autre type de contraintes est connu sous le nom de contraintes d’inégalités. En effet, il est parfois nécessaire d’avoir plus de flexibilité et de bénéficier d’un intervalle de valeur pour satisfaire la contrainte. On considère que la contrainte est satisfaite lorsque est supérieur ou égal à. Sinon, on cherchera à rendre sa dérivée positive.

La contrainte de non-pénétration

L’exemple le plus typique est celui de la contrainte de non pénétration, également la plus importante dans beaucoup de simulations physiques, car c’est elle qui est responsable de la résolution des collisions en empêchant deux solides de se pénétrer et en leur donnant des trajectoires réalistes après un choc.

Comme présenté dans la première partie, l’algorithme EPA nous donne un certain nombre d’informations cruciales pour la résolution des collisions, c’est-à-dire la normale au plan de collision, l’interpénétration et un point de contact sur chaque solide.



*Schéma des informations de collisions utilisées par la contrainte de non-pénétration*

On pose

Si aucune force n’a besoin d’être appliquée puisque les solides ne sont pas en contact.

En dérivant par rapport au temps :

Or

Donc

Dans le cas d’un choc, le facteur de redressement est utilisé pour deux raisons. Premièrement, pour corriger la dérive de position présentée dans la partie a), c’est à dire.

Cependant, pour des raisons de stabilité, on autorisera aux objets une légère interpénétration. On a donc :

Car puisque la contrainte n’est pas satisfaite.

Deuxièmement, on utilise un second facteur pour introduire la restitution de la vitesse. Lorsqu’un objet tombe sur le sol, il rebondira plus ou moins, selon la valeur des coefficients de restitution de chacun des matériaux. La vitesse relativeentre deux solides selon la normale est donnée par :

Après le contact, on a :

Oùest le coefficient de restitution compris entre 0 et 1.

On peut donc introduire le terme suivant :

Le facteur de redressement total est donc

On peut finalement remarquer que l’on doit limiter les valeurs de pour chaque contrainte. Pour les contraintes d’égalité, on peut avoir car la force résultante peut être soit positive, soit négative selon l’axe d’application pour ramener la valeur de la contrainte à 0. Cependant, pour une contrainte d’inégalité, on doit avoir car la force ne peut que être positive pour que soit croissante et devienne supérieure à. Cela sera discuté lors de la résolution des contraintes.

# 2. Détermination du multiplicateur de Lagrange

On considère une contrainte s’appliquant sur un solide à laquelle on associe la matrice Jacobienne de dimension.

On définit l’impulsionpar :

En supposant que la force soit constante pendant le temps, on obtient :

D’après la seconde loi de Newton :

Cette formule peut être étendue aux mouvements de rotation en utilisant le moment des forces. On a alors

Avec la masse du solide et la matrice d’inertie du solide à son centre de gravité.

On définit la matrice où est la matrice identité de taille 3 et la matrice nulle.

On peut alors réécrire les deux équations en une seule :

et correspondent respectivement aux impulsions linéaires et angulaires à appliquer au solide.  
On notera la matrice.

La dernière équation dont nous disposons est celle obtenue grâce à la dérivation de la fonction de contrainte.

On a donc un système de trois équations à deux inconnues : et

En injectant dans puis dans

Or est une matrice ligne donc est une matrice de taille, soit un réel que l’on appellera masse effective, notée On peut donc écrire

La contrainte s’appliquait dans cet exemple sur un seul solide, la matrice était donc de taille et les matrices etde taille. Cependant, on trouve le même résultat avec des contraintes mettant en jeu deux solides, tel que la contrainte de non pénétration.

# 3. Résolution des systèmes de contraintes

La détermination du multiplicateur de Lagrange dans la partie précédente est valable lorsqu’il n’y a qu’une seule contrainte à la fois (c’est-à-dire une contrainte non satisfaite à chaque mise à jour du moteur). Toutefois, dans la plupart des simulations physique, on souhaite pouvoir créer plusieurs contraintes en même temps.

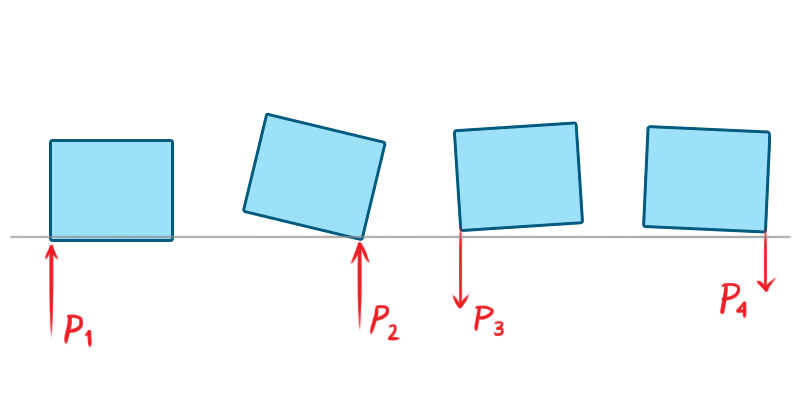
Dans un tel cas, la matrice Jacobienne ne sera plus une matrice ligne mais possèdera autant de lignes qu’il y a de contraintes simultanées. En effet, lorsqu’on empile des boites, la position d’une boite dépendra de celle du dessous mais également de celle du dessus. Par conséquent, le produit matricielsera une matrice carrée de taille, avecle nombre de contraintes non satisfaites. Le calcul du multiplicateur nécessitera donc l’inversion de cette matrice, une opération extrêmement couteuse compte tenu du nombre de contraintes que doit être capable de gérer le moteur physique.

On utilisera donc un solveur itératif appelé solveur par impulsions séquentielles. Il existe d’autres méthodes pour résoudre des systèmes linéaires (comme la méthode de Gauss-Seidel) mais l’algorithme choisi offre de bons résultats tout en restant simple à mettre en place.

L’idée est d’appliquer successivement à chaque contrainte une impulsion en résolvant le système comme si cette contrainte était seule. Au bout d’un certain nombre d’itération, on se rapprochera de la solution réelle du système. Ce nombre est fixé à 15 dans le programme.

Comme indiqué à la fin de la partie 1/, le multiplicateur de Lagrange ne peut pas prendre n’importe quelle valeur. En effet, dans le cas d’une contrainte d’inégalité, on a. Cependant, lorsque l’on utilise les impulsions séquentielles, on calcule plusieurs fois ce multiplicateur, on pourrait donc penser qu’il faut s’assurer qu’à chaque itération, le calculé soit positif. Toutefois, cela donnerait lieu à des erreurs puisque chaque impulsion à pour but de corriger les éventuelles erreurs introduites par l’impulsion précédente. Elle peut donc être négative. Il faut alors garder une variable se chargeant d’accumuler les multiplicateurs de Lagrange et appliquer l’impulsion corrective seulement si le total est positif. Dans le cas de d’une contrainte d’inégalité, on a :

1. **float** oldLambda = accumulatedLambda;
2. accumulatedLambda = max(accumulatedLambda + lambda, 0.0f);
4. lambda = accumulatedLambda - oldLambda;



4 impulsions appliquées successivement. Les impulsions et seront négative

On peut maintenant écrire les étapes principales de la boucle physique :

1. Détermination de la liste des contraintes non satisfaites.
2. Intégration des forces en utilisant la méthode d’Euler, donnant des contraintes ne respectant pas les contraintes.
3. Application des impulsions pour chaque contrainte pour corriger les vitesses. Cette étape est répétée plusieurs fois.
4. Intégration des vitesses pour obtenir les positions et orientations de chaque solide.

*Schéma de l’organisation du moteur*

Voici l’implémentation dans le programme :

1. **void** PhysicEngine::update()
2. {
3. // Generation des informations de collision
4. **for**(auto i(colliders.begin()) ; i != colliders.end() ; ++i)
5. **for**(auto j(std::next(i)) ; j != colliders.end() ; ++j)
6. detectCollision(\*i, \*j);
8. // Determination de la liste des contraintes non satisfaites
9. **for**(Constraint\* constraint: constraints)
10. {
11. **if** (constraint->positionConstraint())
12. activeConstraints.push\_back(constraint);
13. }
15. // Integration des forces
16. **for**(RigidBody\* body: bodies)
17. {
18. body->applyForceToCOM(body->mass \* gravity);
20. body->integrateForces(dt);
21. }
23. // Resolution des contraintes
24. **for**(unsigned j = 0; j < maxIterations; ++j)
25. {
26. **for**(ContactConstraint\* collision: collisions)
27. collision->velocityConstraint(dt);
29. **for**(Constraint\* constraint: activeConstraints)
30. constraint->velocityConstraint(dt);
31. }
33. // Integration des vitesses
34. **for**(RigidBody\* body: bodies)
35. {
36. body->integrateVelocities(dt);
37. body->updateAABBs();
38. }
39. }

III. Rotations

1. En deux dimensions

En deux dimension afin de calculer les rotations d’un objet il suffit, en connaissant l'angle θ d'une rotation et le les coordonnées du point avant la rotation de multiplier ces coordonnées par une matrice de rotation afin d’obtenir les coordonnées du point après la rotation :



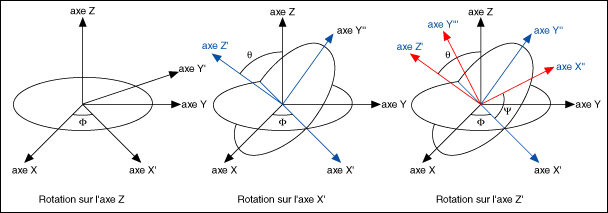
Il est également possible d’utiliser les

2. En trois dimensions

En trois dimensions, Il y a trois manières principales de calculer les rotations d'un objet: Les angles d’Euler, les matrices de rotations ou les quaternions.

1. **Les angles d’Euler et le blocage de cardan**

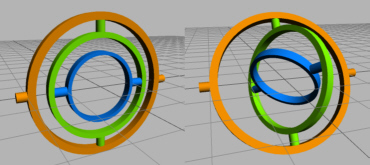
Les angles d’Euler sont trois angles qui permettent de définir l’orientation d’un solide par rapport à un repère, chaque rotations de l’objet est alors décomposée en trois rotations qui s’effectuent autour de trois axes, ces rotations entrainent le déplacement des deux autres axes, sur le schéma ci-dessous, la première rotation autour de l’axe Z entraine le déplacement de l’axe X et de l’axe Y qui devienne respectivement X’ et Y’, puis une rotation autour de l’axe X’ créé les axes Z’ et Y’’, puis enfin la dernière rotation autour de l’axe Z’ entraine la création de X’’ et de Y’’’.



Cependant ces rotations peuvent poser des problèmes de blocage de cardan.

Les rotations avec les angles d’Euler peuvent être comparées à un système contenant trois cardans

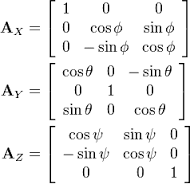
Un cardan est un objet permettant de visualiser les rotations, un cardan est un cercle qui tournent autour d’un axe, le cardan le plus grand influe sur les rotations du cardan plus petit. Sur l’image de droite l’on peut voir trois cardans avec trois axes de rotations orthogonaux, cependant après avoir effectué une rotation de 90 degré selon l’axe du cardan vert, on peut remarquer que les axes des cardans jaune et bleus se superposent (image de gauche): c’est le blocage de cardan.



Lorsque un blocage de cardan apparait, le système perd un degré de liberté, il ne peut alors plus tourner que selon deux axes puisque les rotations de deux axes différents sont les mêmes, pour récupérer un degré de liberté il faut que l’axe non superposé tourne.

1. **Les matrices de rotations**

En 3 dimensions, les coordonnées de chaque point peuvent être représentées par un vecteur donnant la distance du point vers l'origine, il est ensuite possible de multiplier ces coordonnées par une matrice pour chacun des axes, comme montré dans la formule ci-dessous, représentant respectivement les rotations selon l'axe x, y et z selon les angles ϕ, θ et ψ.

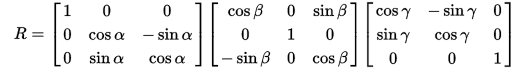


On peut remarquer que la matrice de rotation en 3 dimension ressemble à celle en 2 dimensions, c’est parce chaque rotation autour d’un axe équivaut à une rotation sur un plan, et donc en 2 dimensions.

Une fois la matrice de rotation trouvée, les coordonnées finales des points sont calculé en multipliant la matrice de rotation de taille 3x3 aux coordonnées actuelles des points, de taille 3x1. On obtient alors une matrice de taille 3x1.

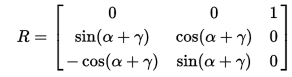


La matrice de rotation d’un point peut donc s’écrire :



Avec α l’angle de rotation selon x, β selon y et γ selon z.

Cependant l’utilisation des matrices est soumise au même problème que les angles d’Euler avec le blocage de cardan, en prenant l’angle et en simplifiant R, on trouve une rotation valant :



Ici, intervertir les deux angles ne changerait pas la rotation, ce qui veut dire que l’objet a perdu un degré de liberté.

1. **Les quaternions**

Finalement les quaternions, le quaternion est un nombre hypercomplexe, un quaternion q s'écrit:

avec, et.

Avec les quaternions, la multiplication n'est pas commutative, par conséquent on peut retrouver d'autres égalités telles que ou.

Pour utiliser les quaternions dans une rotation, il faut connaitre l’angle ainsi que l’axe autour duquel s’effectuera la rotation, on pourra alors créer un vecteur unitaire à partir de cet axe qui se décomposera en une somme de i , j et k les trois axes orthogonaux définissant la base du système. On aura donc v = ai + bj + ck, puis avec une extension de la formule d’Euler :

Une fois le quaternion trouvé, il est possible de calculer la rotation d’un vecteur en utilisant la formule :

Avec et

On a donc, pour = 1,

Tous les calculs avec les quaternion se feront donc avec des quaternions normalisés, c’est-à-dire :

Les quaternions ont l’avantage de ne pas avoir de problème de blocage de cardan. Nous les avons choisi principalement pour cette raison mais aussi car ils sont plus faciles à utiliser dans les équations que des matrices. Par exemple, pour intégrer la vitesse angulaire et obtenir le quaternion à chaque étape de la simulation, on utilise :

Avec W(t) un quaternion tel que W(t) = (0,) , représentant le vecteur de la vitesse angulaire.

On a donc

On peut donc calculer le quaternion représentant la rotation à l’instant d’après en connaissant la vitesse angulaire actuelle ainsi que le quaternion représentant la rotation actuelle.

Ce quaternion doit alors être ré-normalisé, car à cause des imprécisions, sa norme s’éloignera de plus en plus de 1.

Cependant, lorsque l’on voudra afficher un objet à l’écran, il faudra transformer tous ses points grâce à la position, la rotation et l’échelle. Cela peut être fait séparément mais les matrices nous permettent de regrouper ces trois opérations en une seule. De plus, il faudra encore la multiplier par une matrice dépendant du point de vue (elle transformera chaque point en fonction de la position et l’orientation de la camera en prenant en compte la perspective), et la matrice de projection (projetant le monde en 3D sur un image en 2D).

La matrice équivalente d’un quaternion est :

1. **void** Transform::toMatrix()
2. {
3. toWorldSpace = glm::translate(mat4(1.0f), position);
4. toWorldSpace \*= toMat4(rotation);
5. toWorldSpace \*= glm::scale(scale);
7. toLocalSpace = inverse(toWorldSpace);
8. }

Ces matrices de changement de base seront aussi utilisées pour convertir un point de l’espace local du solide à l’espace global et inversement.

IV. Caractéristiques des solides utilisés

Pour chaque nouvel objet ajouté il est nécessaire de calculer plusieurs variables à partir des mesures basiques de l'objet tel que la hauteur, la largeur, la longueur ou le rayon, et la masse volumique. On peut citer :

-la masse, avec masse = masse volumique \* volume.

-La fonction de support de l'objet, qui diffère en fonction de la forme de l’objet.

-L’AABB (Axis Aligned Bounding Box).

- la matrice d’inertie, qui permettra de calculer des déplacements et qui varie pour chaque forme d’objets.

1. Sphère

La matrice d’inertie d’une sphère est:

Avec M la masse de la sphère et R son rayon.

Pour la sphère :

1. **void** Sphere::computeMass()
2. {
3. **float** squareRadius = getRadius();    squareRadius \*= squareRadius;
5. **float** mass = rigidBody->density \* (4.0f/3.0f) \* PI \* squareRadius;
7. rigidBody->mass += mass;
8. rigidBody->inertia += diagonal3x3(vec3((2.0f/5.0f) \* mass \* squareRadius));
10. rigidBody->COM += mass \* center \* tr->scale;
11. }

La fonction computeMass permet donc à partir du rayon du cercle de calculer sa masse, sa matrice d’inertie ainsi que le centre de gravité (COM).

La fonction de support d’une sphère de rayon r selon l’axe v est :

2. Cube

Matrice d’inertie du cube :

Avec M la masse du cube, h la hauteur, l la longueur et p la profondeur.

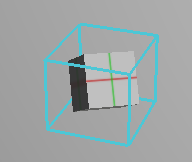
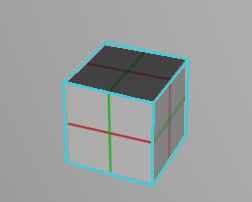
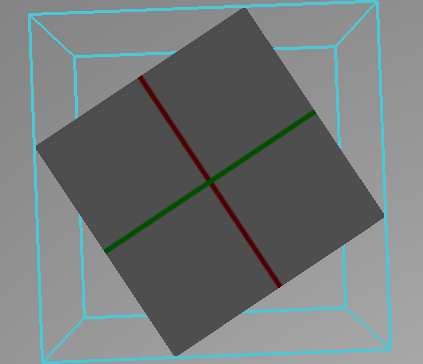
1. **void** Box::computeMass()
2. {
3. vec3 d = getDimensions();
5. **float** mass = rigidBody->density \* d.x\*d.y\*d.z;
7. **float** x2 = d.x\*d.x;
8. **float** y2 = d.y\*d.y;
9. **float** z2 = d.z\*d.z;
11. rigidBody->mass += mass;
12. rigidBody->inertia += diagonal3x3((1.0f/12.0f)\*mass \* vec3(y2 + z2, x2 + z2, x2 + y2));
14. rigidBody->COM += mass \* center \* tr->scale;
15. }

Pour déterminer l’AABB d’une boite :

1. **void** Box::computeAABB()
2. {
3. aabb.center = min(tr->getToWorldSpace(-halfExtent), tr->getToWorldSpace(vec3(-halfExtent.x, -halfExtent.y, halfExtent.z)));
5. **for** (**int** i(1) ; i < 4 ; i++)    // On cherche le point le plus bas
6. {
7. vec3 point(halfExtent.x \*(2.0f\*(i>1) -1.0f), halfExtent.y \*(2.0f\*(i%2) -1.0f), -halfExtent.z);
9. aabb.center = min(aabb.center, tr->getToWorldSpace(point));
10. point.z \*= -1.0f;
12. aabb.center = min(aabb.center, tr->getToWorldSpace(point));
13. }
15. aabb.dim = (tr->getToWorldSpace(vec3(0.0f)) - aabb.center);
16. aabb.center = tr->getToWorldSpace(center);
17. }

La fonction convertit un vecteur depuis la base de l’objet vers la base canonique de l’espace.

Le programme créé une nouvelle boite dont la taille selon chaque axe sera celle de la plus grande distance selon cet axe de tous les points du cube. On peut voir ci-dessous le cube en blanc, avec le cube bleu représentant la zone de détection de l’objet qui s’adapte en fonction des rotations et translations de l’objet, si le cube est aligné avec les axes le carré bleu se superpose à l’objet.

La fonction de support du carré est :

Ce qui donne, une fois écrit dans le programme :

1. vec3 Box::getSupport(vec3 \_axis)
2. {
3. \_axis = tr->getVectorToLocalSpace(\_axis);
5. vec3 support(halfExtent);
7. **for** (unsigned i(0) ; i < 3; i++)
8. **if** (\_axis[i] < 0.0f)
9. support[i] \*= -1.0f;
11. **return** tr->getToWorldSpace(support + center);
12. }

Sgn(x) représente le signe de x, par conséquent il sera toujours égal à 1 ou à -1

3. Cylindre

La matrice d’inertie du cylindre est :

Avec M la masse h la hauteur et r le rayon du cylindre.

Fonction de support du cylindre:

Avec et

4. Cône

Matrice d’inertie du cône :

Avec M la masse, h la hauteur et r le rayon de la base du cône.

Fonction de support du cône :

Avec , , r est le rayon et est l’angle au sommet. Donc

1. vec3 Cone::getSupport(vec3 \_axis)
2. {
3. \_axis = tr->getVectorToLocalSpace(\_axis);
5. **float** axisDistanceSquared = \_axis.x \* \_axis.x + \_axis.y \* \_axis.y;
7. vec3 support(0.0f, 0.0f, -height \* 0.25f);
9. **if** (\_axis.z > sinusAngle)
10. support.z = height \* 0.75f;
12. **else** **if** (axisDistanceSquared > EPSILON\*EPSILON)
13. {
14. **float** rOnDist = radius / sqrt(axisDistanceSquared);
15. support = vec3(rOnDist \* \_axis.x, rOnDist \* \_axis.y, support.z);
16. }
18. **return** tr->getToWorldSpace(support + center);
19. }

# Conclusion :

Nous conclurons ce TIPE en présentant les limites de notre programme :

-Comme nous avons pu le voir dans la partie sur la détection de collision, tous les solides convexes ou pouvant être décomposés en plusieurs solides convexes sont gérés par le moteur physique. Cependant, seulement les sphères, boites, cônes, et cylindres sont implémentés. Une amélioration intéressante serait de rendre possible l’utilisation de solides générés à partir d’une liste de point et de normales.

-De plus, uniquement les chocs élastiques sont simulés. Cette simplification permet d’utiliser des méthodes simples pour la réponse aux collisions. Il est possible de simuler des objets mous en temps réel mais cela demande des algorithmes très performants ainsi que des machines relativement puissantes.

-Notre implémentation de GJK+EPA, l’algorithme de détection de collision, ne nous permet pas de trouver plus d’un point de contact au cours d’une même collision, ce qui s’avère problématique quand une boite est posée sur un sol plat. On pourra s’apercevoir sur le programme que l’objet oscillera autour d’une position sans jamais se stabiliser. Ceci est dû au changement permanent du point de contact détecté par l’algorithme. De plus, EPA étant un algorithme itératif, on se rapprochera de plus en plus de la solution à chaque étape. Dans certains cas, comme quand on pose une boule sur le sol, la normale au plan de collision trouvée sera très proche du vecteur dirigé vers le haut mais pas tout à fait égale. Cela a pour résultat de faire rouler la boule même lorsque sa vitesse est nulle initialement.

-Finalement, la détection de collision est discrète (par opposition à continu). En effet, si un objet petit à une vitesse suffisamment élevée, il se peut qu’il puisse traverser complètement un objet fin en un intervalle de temps. Dans ce cas, il n’y aura pas de détection de collision.

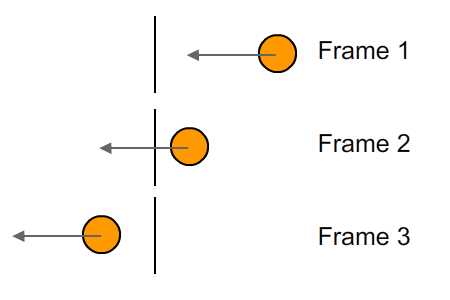


Illustration de l’effet « tunnel »

# Sources

* <http://box2d.org/downloads/>
* <http://danielchappuis.ch/download/ConstraintsDerivationRigidBody3D.pdf>
* <https://www.toptal.com/game/video-game-physics-part-iii-constrained-rigid-body-simulation>
* <https://mollyrocket.com/849>
* <http://www.dyn4j.org/2010/05/epa-expanding-polytope-algorithm/>
* <http://www.dyn4j.org/2010/04/gjk-distance-closest-points/>
* <http://hacktank.net/blog/?p=119>
* <https://github.com/bulletphysics>
* <http://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/index.htm>
* <http://scienc.industrielles.free.fr/dynamique/matrice%20d'inertie.htm>
* <http://mathworld.wolfram.com/RotationMatrix.html>
* <http://solid.sourceforge.net/jgt98convex.pdf>