Université Toulouse III
Paul Sabatier

Chef d'œuvre - M2IM

Simulation Physique Basée Sur Les Positions

Méthodes et Algorithmes

Auteurs :
Fabien Boco,
Agathe Duyck,
Anselme François,
Dimitri Raguet

Superviseurs :
Charly Mourglia
Valentin Rousselet

6 novembre 2015

Résumé

Ce document présente les méthodes et algorithmes entrant dans la réalisation du chef d'œuvre de master 2 Image et multimédia. Ce premier rapport à pour objectif de démontrer notre compréhension du sujet qui nous est proposé et des outils algorithmiques et mathématiques impliqués.

Table des matières

Chapitre 1

Contexte

Dans le domaine de l'informatique graphique, un problème majeur est la simulation d'objets dynamiques et leurs interactions. Afin d'obtenir un rendu esthétiquement réaliste, les algorithmes calculant ces interactions doivent prendre en compte une certaine représentation des règles physiques du monde réel.

Par ailleurs, afin d'avoir un rendu en temps réel, ces règles physiques ne peuvent être directement implémentées.

Les moteurs physiques actuels savent manipuler sans problèmes les objets rigides, cependant dans le cas des objets déformables tels que les tissus, les fluides et autres objets mous, ces moteurs ne peuvent garantir des performances temps réel. En effet, ces méthodes sont généralement basées sur les éléments finis ou des systèmes masses-ressort et sont très peu performantes.

La méthode proposée par Matthias Müller et ses collègues dans l'article Position Based Dynamics (PBD) [Müller et al., 2007] propose une une simulation basée sur les positions qui parvient à manipuler tout type d'objet en respectant les contraintes de temps réel.

Contrairement aux méthodes classiques, basées sur la simulation de Newton, cette méthode possède comme principal avantage d'être assez facile à implémenter. En effet, les objets sont tous ramenés à des primitives de base : des particules élémentaires et des triangles. Ce choix engendre certes un nombre très important de primitives mais ce traitement en devient extrêmement rapide et aisé. En effet toutes les règles caractérisant les objets modélisés et leurs comportement sont représentées par des contraintes ne prenant en paramètre que les positions des primitives. Malgré cette simplification, le modèle reste stable et possède de bonnes propriétés de conservation d'énergie (moment linéaire et moment angulaire).

Chapitre 2

Algorithme général

L'algorithme PBD [Müller et al., 2007] est composé d'une boucle principale exécutée à un pas de temps régulier. Il modifie la position des points à chaque itération et se décompose en cinq parties :

Intégration (1) La première consiste à intégrer sur le temps, les accélérations afin de mettre à jour la vitesse des points puis à intégrer ces vitesses pour obtenir leurs positions selon une version modifiée de la méthode semi-implicite d'EULER

Génération des contraintes (2) A partir de ces positions mises à jour, on détecte des collisions qui permettent de générer des contraintes au temps courant.

Solveur (3) La troisième partie consiste à résoudre les contraintes : elle prend en entrée les contraintes et met à jour les positions des points en conséquence.

Mise à jour des positions et vitesses (4) La scène est mise à jour en fonction des résultats apportés par le solveur.

VelocityUpdate (5) Cette dernière partie sert à modifier les vitesses en fonction du comportement particulier des particules lors d'une collision. Cela permet de modéliser des effets spécifiques tels que la friction, la restitution d'énergie, qui seraient plus difficiles à modéliser au travers des contraintes.

```
loop
    (1)
    for all vertices i do v_i ← v_i + Δt w_i f_ext(x_i)
    for all vertices i do p_i ← x_i + Δt v_i
    (2)
    for all vertices i do genCollConstraints(x_i → p_i)
    (3)
    loop solverIteration
        projectConstraints(C_1 ,... ,C_M , p_1 ,... ,p_N)
    endloop
    (4)
    for all vertices i do
        v_i ← (p_i − x_i)
        x_i ← p_i
    endfor
    (5)
    velocityUpdate(v_i ,... ,v_N)
```

```
end loop
```

Chapitre 3

Intégration

L'intégration est une réécriture des formules de la dérivée qui nous permet d'estimer le déplacement des points, en effet on a :

$$v(t) = \frac{\delta x(t)}{\delta t} \Leftrightarrow \delta x(t) = v(t) * \delta t$$
$$a(t) = \frac{\delta v(t)}{\delta t} \Leftrightarrow \delta v(t) = a(t) * \delta t$$

Or lors de la discrétisation il y a plusieurs méthodes pour estimer ce déplacement :

- La méthode explicite consiste à mettre à jour la position du point en fonction de son ancienne vitesse, puis à mettre à jour sa vitesse en fonction des forces appliquées à son ancienne position. Cette méthode est relativement peu stable.
- La méthode implicite consiste à résoudre un système linéaire : la nouvelle position du point est calculée en fonction de la vitesse du point à cette nouvelle position ($v_2(t) \leftarrow v(t) + \Delta t\ w\ f_{ext}(p)$ et $p \leftarrow x + \Delta t\ v_2$). Elle est de ce fait extrêmement lourde à résoudre.
- La méthode semi-explicite consiste à calculer d'abord la vitesse du point en fonction des forces qui s'appliquent en ce point, puis de calculer le déplacement de ce point en fonction de sa nouvelle vitesse. C'est la méthode utilisée ici. Elle possède comme avantage d'être suffisamment stable et d'être rapide à évaluer.

Chapitre 4

Détection des collision

Les collisions peuvent être détectées de différentes manières :

- Projection du point sur le plan : An efficient ray-polygon intersection [Badouel, 1990] et Segment-triangle intersection [O'Rourke, 1998]
- Test de l'inclusion du point dans le triangle : Ray tracing complex models containing surface tessellations [Snyder and Barr, 1987]
- Fast, minimum storage ray/triangle intersection [Möller and Trumbore, 2005]

Les deux premières méthodes ont l'avantage d'être plus intuitives tandis que la dernière à été conçue pour le temps réel et devrait être plus adaptée à nos contraintes.

Structure adaptée Nous manipulons une scène composée d'un grand nombre de triangles dynamiques et indépendants, il n'est donc pas possible de les organiser dans un Kd-tree ou un Octree, sous peine de devoir les reconstruire à chaque itération.

Nous utiliserons le hachage spatial tel qu'il est décrit dans cet autre article de Matthias Müller [Teschner et al., 2003]. Cette structure de données propose de diviser la scène en voxels, chaque voxel étant une cellule de table de hachage, les triangles sont rangés dans toutes les cellules intersectant avec les boîtes englobantes de ces triangles. Comme la fonction de hachage lie une cellule aux coordonnées des points qu'elle contient, cette méthode est très rapide pour trouver les triangles proches du segment d'intersection lors du calcul de collisions.

Chapitre 5

Contraintes

A cette étape, nous sommes en présence d'un moteur pouvant contenir des objets animés. Cependant, aucune interaction entre eux ne peut être simulée. Par exemple, si deux objets se dirigent l'un vers l'autre, ils passeront l'un au travers de l'autre. On souhaite donc par exemple interdire qu'un point d'un objet pénètre un autre objet. si on prend le cas de deux sphères de centre P_1 et P_2 et de rayons r_1 et r_2, cette contrainte peut être modélisée par l'équation suivante :

$$(P_1.x - P_2.x)^2 + (P_1.y - P_2.y)^2 + (P_1.z - P_2.z)^2 \geq (r_1 + r_2)^2$$

De la même manière, toutes les règles physiques (telle que la pression, la conservation de volume, les degrés de liberté autour d'un point, la viscosité, la rigidité, etc.) qui s'appliquent aux primitives de la scène peuvent être modélisées par des égalités ou inégalités.

Concrètement, ces contraintes sont des fonctions qui prennent en entrée les positions des particules et renvoient en sortie un scalaire que l'on cherchera à minimiser.

À un instant donné, la totalité des contraintes doivent être respectées et forment donc un système d'équations à résoudre. La méthode classique de résolution d'un tel système est d'effectuer une analyse en composantes principales.

Cependant, au vu des dimensions qu'a la matrice de contraintes, il n'est pas possible d'allouer en mémoire la totalité de ses éléments. Comme cette matrice contient un nombre important de valeurs nulles, il est possible d'utiliser une matrice creuse, notamment l'implémentation fournie par la bibliothèque Eigen. Ceci dit, il est également possible de s'affranchir de la représentation mathématique sous forme de matrice. En effet, la résolution par la méthode de Gauss-Seidel permet de résoudre chaque contrainte indépendamment. Une structure de données telle qu'une matrice contenant simultanément toutes les contraintes sous le même format n'est pas nécessaire. Une simple liste de contraintes suffit, en utilisant le polymorphisme pour les modéliser, quelle que soit la complexité de chaque contrainte.

Chapitre 6

Résolution de contraintes

Gauss-Seidel non-linéaire

Dans notre article nous allons utiliser une version non linéaire de la méthode de résolution de système linéaire Gauss Seidel. Mais dans un premier temps nous allons parler le la résolutions classique.

Soit un système linéaire avec n inconnues $x = [x_1, x_2, ..., x_n]$ et n équations. Le système est d'abord initialisé avec des valeurs arbitraires pour chaque x_i, puis pour chaque équation la valeur de x_i est résolue en fonction des autres valeurs. Puis le résultat est injecté dans les équations suivantes :

Exemple en dimension 2 :

$$a_1x_1 + a_2x_2 + b_1 = 0$$
$$a_3x_1 + a_4x_2 + b_2 = 0$$

Ces équations peuvent être réécrites en réinjectant le x à chaque fois dans l'équation suivante, comme montré dans la figure 6.1 :

$$x_1^1 = (-a_2x_2 - b_1)/a_1 \tag{6.1}$$
$$x_2^1 = (-a_3x_1^1 - b_2)/a_4 \tag{6.2}$$

Projection des contraintes : Lors de l'étape de la projection des contraintes, nous avons un système de N équations $C = [C_1, C_2, ..., C_N]$, qui retournent un scalaire et $3 * M$ inconnues $x = [x_1, x_2, ..., x_M]$ (chaque point étant composé de trois composantes)

$$\begin{gathered} C_1(x + \Delta x) = 0 \\ C_2(x + \Delta x) > 0 \\ \vdots \\ C_n(x + \Delta x) = 0 \end{gathered}$$

Pour linéariser les équations nous appliquons donc la formule de Lagrange au premier niveau :

$$\begin{gathered} \nabla C_1(x).\Delta x = -C_1(x) \\ \nabla C_2(x).\Delta x > -C_2(x) \\ \vdots \\ \nabla C_n(x).\Delta x = -C_n(x) \end{gathered}$$

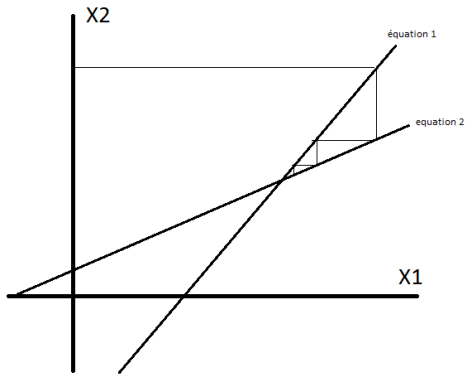

FIGURE 6.1 – Représentation de GAUSS SEIDEL en dimension 2

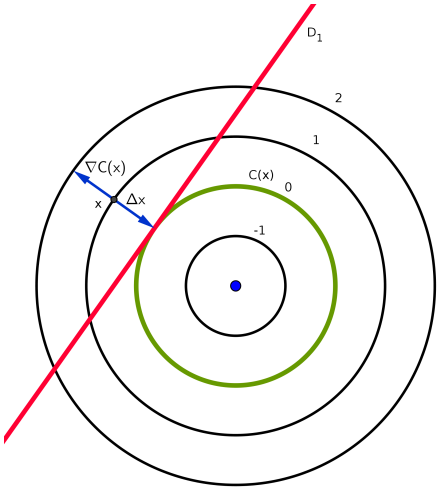

FIGURE 6.2 – Représentation d'une contrainte

Or dans ce cas nous avons un système fortement sous contraint. En effet nous partons d'une équation non linéaire (dont la solution $C(x) = 0$ est représenté par le cercle vert dans la figure 6.2), que nous approximons par un produit scalaire avec LAGRANGE. Or ce produit possède une infinité de solutions (représenté par D_1 dans la figure 6.2) . Nous ajoutons donc une contrainte qui a la bonne propriété de conserver les quantités de mouvement : $\Delta x = -\lambda \nabla C(x)$, afin d'obtenir un système bien contraint avec une unique solution (représentée par le vecteur Δx dans la figure 6.2).

Le système n'étant pas forcément bien contraint, c'est-à-dire N pouvant être différent de $3 * M$, nous allons donc utiliser une version modifiée de la méthode de GAUSS SEIDEL non linéaire qui applique les déplacements calculés à chaque pas aux système de manière itérative.

```
loop nbloop times
    forall contrainte i do
        forall particule j do
            if isInequality or C_i(particule_j)< 0
                particule.Delta x = contrainte.calculerDeltaX(particule_j);
            endif
        endfor
        forall particules j do
            particules_j.position += particule_j.Delta x;
        endfor
    endfor
endloop
```

Par rapport à la méthode de Jacobi, cette méthode a pour principal avantage d'être stable (c'est-à-dire que l'énergie du système est conservée (ou au moins diminue), et permet de rendre un résultat plus doux. De plus cette, cette méthode convergeant plus rapidement que celle de Jacobi, et les données en entrées étant relativement proche de la solution, on peut obtenir un résultat cohérent avec un petit nombre d'itérations. Cette méthode est en revanche difficilement parallélisable.

Chapitre 7

Visualisation

L'implémentation de cette méthode sera intégrée en tant que plugin au sein d'un moteur de rendu existant. Notre environnement 3D sera géré par Radium Engine, un moteur graphique proposé et développé par nos encadrants.

Nous souhaitons développer un plugin le plus indépendant possible du moteur de rendu afin de faciliter la réutilisation du code dans un autre contexte.

Chapitre 8

Améliorations

8.1 Parallélisation de Gauss-Seidel

La méthode de Gauss-Seidel est un algorithme itératif difficilement parallélisable. Cependant, chaque itération n'agit que sur un nombre limité de primitives.

Il est possible de construire un graphe de contraintes et déterminer quelles contraintes sont indépendantes des autres. Un algorithme de coloration de graphe pourra être appliqué, permettant de mettre en évidence ces indépendances et de traiter en parallèle plusieurs contraintes. Par ailleurs, on pourra déterminer le nombre de passes nécessaires grâce au nombre de couleurs minimales nécessaires pour colorer le graphe.

Un algorithme de partition de graphe pouvant être appliqué dans ce cas est décrit dans l'article de [Fratarcangeli and Pellacini, 2015]

8.2 Fluides

La simulation des fluides [Macklin and Müller, 2013] repose essentiellement sur le PBD mais va se différencier à deux niveaux. La totalité des contraintes qui s'appliquent à une particule d'un fluide sont locales, et donc dépendantes uniquement des particules voisines. Il faut ajouter à la boucle principale une étape de détection de voisinage. Afin de modéliser le comportement d'un fluide, il est nécessaire de prendre en compte de nouvelles contraintes telles que la pression, l'incompressibilité et la tension de surface.

Il est également nécessaire d'implémenter un algorithme spécifique pour la visualisation des particules, car les fluides ne contiennent aucune surface. Il faut donc générer une surface à la volée à chaque pas de temps, en utilisant par exemple les algorithmes décrits dans les articles [Yu and Turk, 2013] [van der Laan et al., 2009].

La simulation de fluides en temps réel implique que l'implémentation de cet algorithme soit presque entièrement réalisée sur GPU. L'ajout des fluides dans le moteur physique PBD est une modification majeure qui demande un temps de travail considérable, il est peu probable que nous ayons le temps de l'implémenter.

Bibliographie

[Badouel, 1990] Badouel, D. (1990). An efficient ray-polygon intersection. In Graphics gems, pages 390–393. Academic Press Professional, Inc.

[Bender et al.,] Bender, J., Müller, M., and Macklin, M. Position-based simulation methods in computer graphics.

[Fratarcangeli and Pellacini, 2015] Fratarcangeli, M. and Pellacini, F. (2015). Scalable partitioning for parallel position based dynamics. In EUROGRAPHICS, volume 34, page 2015.

[Macklin and Müller, 2013] Macklin, M. and Müller, M. (2013). Position based fluids. ACM Transactions on Graphics (TOG), 32(4) :104.

[Möller and Trumbore, 2005] Möller, T. and Trumbore, B. (2005). Fast, minimum storage ray/triangle intersection. In ACM SIGGRAPH 2005 Courses, page 7. ACM.

[Müller, 2008] Müller, M. (2008). Hierarchical position based dynamics.

[Müller et al., 2007] Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. (2007). Position based dynamics. Journal of Visual Communication and Image Representation, 18(2) :109–118.

[O'Rourke, 1998] O'Rourke, J. (1998). Segment-triangle intersection. Computational Geometry in C, 2nd edn. Section, 7.

[Snyder and Barr, 1987] Snyder, J. M. and Barr, A. H. (1987). Ray tracing complex models containing surface tessellations, volume 21. ACM.

[Teschner et al., 2003] Teschner, M., Heidelberger, B., Müller, M., Pomerantes, D., and Gross, M. H. (2003). Optimized spatial hashing for collision detection of deformable objects. In VMV, volume 3, pages 47–54.

[van der Laan et al., 2009] van der Laan, W. J., Green, S., and Sainz, M. (2009). Screen space fluid rendering with curvature flow. In Proceedings of the 2009 symposium on Interactive 3D graphics and games, pages 91–98. ACM.

[Yu and Turk, 2013] Yu, J. and Turk, G. (2013). Reconstructing surfaces of particle-based fluids using anisotropic kernels. ACM Transactions on Graphics (TOG), 32(1) :5.