Un moteur de physique
Article 2 - Implémentation

Gabriel Peyré
nikopol0@altern.org
www.orion3d.fr.st

Le 19 septembre 2001

Table des matières

Introduction

Suite à la (brève) présentation des équations de la physique, vient le moment de mettre tout ça en pratique, et d'implémenter ces équations au sein d'un moteur de physique. Les deux choses que j'ai garder en tête en écrivant cet article sont :

1. Proposer une implémentation orientée objet au maximum, quitte à perdre un peu au niveau des performances.
2. Avoir un code extensible au maximum, de manière à ce qu'il soit facile, par la suite, de lui ajouter des fonctionnalités.

Pour présenter une implémentation typique, je vais utiliser les notations de mon moteur 3D, Orion3D, cependant, celles ci sont aisément compréhensible, et ré-utilisable pour n'importe quel contexte orienté objet.

Pour vous facilité la lecture de cet article, voici une liste des principales classes de bases utilisées :

1. `OR_Object` : objets 3D de bases manipulés par le moteur, il n'ont pas de propriétés physique particulière (il faut y remédier !).
2. `OR_Vector3D` : un vecteur 3D, avec des opérateurs mathématiques classiques (produit scalaire, vectoriel, etc.).
3. `OR_Quaternion` : un quaternion, qui représente une rotation via 4 floats, toujours avec quantité d'opérateurs ad hoc.
4. `OR_Matrix` : une matrice affine 4x4, avec les opérateurs de multiplication de matrices entre elles, et de multiplication vecteur/matrice.

Chapitre 1

Classe rigid body

La classe de base pour la gestion de la physique dans Orion3D est la class `OR_RigidBody`. Nous allons donc présenter en détails les différents éléments qui la compose, car aux vues des différentes options possible, il est important de faire les bons choix d'implémentation.

1.1 Justification des choix effectués

Tout d'abord, tous les `OR_Object` entrant pouvant être manipulé par le moteur de physique doivent posséder un `OR_RigidBody`, qui encapsule toutes les données qui vont être décrite par la suite.

Quelle sont donc ces données ? En gros, il s'agit de faire des choix sur les variables dont on se sert pour intégrer les équations de la physique, et celles qui servent pour les calculs annexes. en gros, il y a forcément 4 variables :

1. une relative à la position de l'objet
2. une relative à la dérivée de la position de l'objet
3. une relative à l'orientation de l'objet
4. une relative à dérivée de l'orientation de l'objet

1.2 Notations

Dans la suite, je noterais :

1. $q^{ToMatrix}$ l'opérateur qui transforme un quaternion en matrice de rotation.
2. M^T l'opérateur de transposition des matrices.
3. v^{ToQuat} l'opérateur qui transforme un vecteur en quaternion (ie. si on écrit un quaternion $s + v_x * i + v_y * j + v_z * k = [s,v]$, alors $v^{ToQuat} = [0,v]$).
4. v^{CrossP} l'opérateur qui transforme un vecteur v en matrice anti-symétrique de produit-vectoriel (de telle sorte qu'on ai : $v^{CrossP} * x = v \wedge x$), ie :

$$v^{CrossP} = \begin{pmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{pmatrix}$$

1.3 Variables d'état

Voici donc les 4 variables d'états retenues :

1. `OR_Vector3D Pos_` : la position du centre de gravité de l'objet, en world space.
2. `OR_Quaternion Rot_` : la rotation de l'objet, qui fait passer du repère du monde dans le repère de l'objet (ie. pour un point p_0 de l'objet, la position en world space est $p = Rot\_ * p_0 + Pos\_$). Ce n'est pas la matrice de transformation de l'`OR_Object`, puisque le centre de gravité n'est pas nécessairement le centre de l'objet !! [cf. plus loin, la matrice de transfo de l'objet est calculée 'on the fly'].

3. `OR_Vector3D P_` : la quantité de mouvement du centre de gravité, ie. $P = m * v$.
4. `OR_Vector3D L_` : le moment linéaire de l'objet, ie. $\omega = I * L$ (avec ω rotation instantanée).

1.4 Constantes

Ensuite, il y a quelques constantes :

1. `float rMass_` : la masse de l'objet.
2. `OR_Matrix IBody` : la matrice d'inertie donnée dans les coordonnées de l'objet (coordonnées).
3. `OR_Matrix IBodyInv` : ll'inverse de la matrice d'inertie donnée dans les coordonnées de l'objet (coordonnées).
4. `OR_Vector3D GravityCenter` : position du centre de gravité, donné dans les coordonnées 'Orion3d' de l'objet, cad. par rapport au centre 3DSMax de l'objet.

1.5 Variables auxiliaires

ensuite, il y a les variables auxiliaires, qui sont couplées aux variables précédente (elles sont mises à jour à chaque modification des variables précédentes) :

1. `OR_Matrix IWorld_` : matrice d'inertie dans les coordonnées du monde. Se calcule par la relation $IWorld\_ = (RotMatrix\_^T) * IBody\_ * (RotMatrix\_)$.
2. `OR_Matrix IWorldInv_` : l'inverse de la précédente.
3. `OR_Vector Speed_` : vitesse de l'objet. Se calcule par $Speed\_ = \frac{P\_}{rMass\_}$.
4. `OR_Vector Omega_` : rotation instantanée. Se calcule par $Omega\_ = IWorld\_ * L\_$.
5. `OR_Matrix RotMatrix_` : l'orientation de l'objet, mais exprimé sous forme de matrice, ie $RotMatrix\_ = Rot\_^{ToMat}$
6. `OR_Vector PosOld_` : position a la dernière update (utilisée pour la deflexion de particules).

1.6 Moments, forces et impulsions

Enfin, il y a les forces, moments et impulsions :

1. `OR_Vector3D ForceAccumulator_` : les forces agissant sur l'objet.
2. `OR_Vector3D TorqueAccumulator_` : les forces agissant sur l'objet.
3. `OR_Vector3D ForceImpulseAccumulator_` : les impulsions agissant sur la vitesse de l'objet.
4. `OR_Vector3D TorqueImpulseAccumulator_` : les impulsions agissant sur la vitesse de rotation de l'objet.

Chapitre 2

Equations de la mécanique

Nous arrivons à la partie intéressante du problème, à savoir la résolution des équations de la mécanique du solide. La première chose est de traduire en terme de variables d'états les équations vues dans le tutorial précédent. Ensuite, nous verrons comment calculer les variables annexes dont nous avons aussi besoin.

2.1 Equations du mouvement

Voici les équations différentielles à résoudre (j'ai ajouté les '(t)' pour bien montrer la dépendance vis à vis du temps ...) :

$$\frac{dPos\_(t)}{dt} = Speed\_(t)$$

$$\frac{dRotMatrix\_}{dt} = Omega_(t)^{CrossP} * RotMatrix\_(t)$$

$$\frac{dP_(t)}{dt} = ForceAccumulator\_(t)$$

$$\frac{dL\_(t)}{dt} = TorqueAccumulator\_(t)$$

2.2 Utilisation des quaternions

Remarque sur l'utilisation d'un quaternion pour `Rot_` : on n'utilise pas de matrice car :

1. a cause des erreurs d'arrondi, l'integration successive de (3) rendrait la matrice de moins en moins orthogonale, ce qui forcerait à la re-orthogonaliser (procédé de **gram-schmidt**), ce qui est beaucoup plus facile à faire avec quat (juste diviser par la norme). C'est exactement le même problème qu'avec les caméras de max qui plantent quand on regarde à la verticale : la re-orthogonalisation foire.
2. l'équation (2) est plus facile à intégrer, cf. ci-dessous.

Voici comment il faut ré-écrire l'équation (2) pour ne pas se trimbaler des conversions quat/matrice puis matrice/quat :

$$\frac{dRot\_}{dt} = \frac{1}{2} * Omega_(t)^{CrossP} * Rot\_(t)$$

Je sais, ça parait magique, mais ça marche ! (c'est juste des opérations élémentaires sur les sin et les cos)

2.3 Prise en compte des impulsions

Après chaque itération dans la résolution des équation (1), (2′), (3), et (4), il faut aussi ajouter les impulsions :

$$P\_(t) += ForceImpulseAccumulator\_(t)$$

$$L\_(t) += TorqueImpulseAccumulator\_(t)$$

Sans oublier de recalculer toutes les variables auxiliaires ...

2.4 Matrice de transformation de l'OR_Object

Voici comment on calcule la matrice de transformation "Orion3D" `AbsoluteMatrix` à partir de `RotMatrix_`, `Pos_` et `GravityCenter_` :

$$AbsoluteMatrix_{rotation} = RotMatrix\_$$

$$AbsoluteMatrix_{translation} = Pos\_ - RotMatrix\_ * GravityCente$$

Chapitre 3

Détails d'implémentation

Voilà, tout ceci résume l'implémentation la mécanique du solide dans Orion3D. Il reste bien sûr le plus dur :

1. Le calcul des accumulateurs, qui sera fait par le moteur de collisions.
2. La direction de la résolution de la physique.
3. L'intégration des équation différentielles du mouvement.

Le premier point est discuté dans les deux articles qui suivent. Je vais maintenant aborder les détails d'implémentation, et expliquer comment les deux derniers points ont été résolus dans Orion3D.

3.1 Les objets manipulés

La classe `OR_PhysicEntity` est la classe de base, de laquelle doivent hériter tous les objets qui veulent être simulés par le moteur de physique. Dans **Orion3D**, voici les trois sortes d'objets possibles :

1. `OR_RigidBody` : pas la peine d'insister, cette classe vous est déjà familière !
2. `OR_ParticuleManager` : un ensemble de particules, répondant aux équations de la mécanique du point.
3. `OR_Deflector_ABC` : un objet capable de "déflecter" [1] des particules.

Les principales méthodes que l'objet héritant de la classe `OR_PhysicEntity` doit implémenter sont celles qui résolvent les équations de la physique. Il y en a une par type d'intégrateur, et c'est le `OR_PhysicEntityManager` qui décide quel intégrateur utiliser :

1. `void UpdatePhysics_RK4( OR_U32 nStepRK4 )` : calcule l'un des 4 pas de l'intégrateur *Runge Kutta 4*.
2. `void UpdatePhysics_Verlet()` : calcule un pas d'intégration de l'intégrateur *Verlet* (utilise surtout pour les systèmes de particules).
3. `void UpdatePhysics_Euler()` : calcule un pas d'intégration de l'intégrateur *Euler explicite*.

3.2 Le moteur de physique

Le moteur de physique, dans **Orion3D**, se nomme `OR_PhysicEntityManager`, et gère un ensemble de `OR_PhysicEntity`, capable de s'entrechoquer les uns avec les autres. Il contient donc :

1. Une liste de `OR_PhysicEntity`.
2. Un manager de forces, qui regroupe les forces communes à tous les objets.
3. Un manager de collision, pour résoudre les collisions entre les objets. Voir les articles suivant pour plus de détails.
4. Un manager de contraintes, pour résoudre les contraintes entre les objets. Voir le dernier article pour plus de détails.

1. les particules rebondissent dessus

Voici comment se déroule un cycle de mise à jour du moteur de physique (appelé à chaque image) :

1. Calcul du *time step* : calcul le temps entre deux mises à jour. Voir plus loin pour les détails.
2. Ajoute les forces s'exerçant sur chaque objet aux différents accumulateurs.
3. Demande au manager de collision de résoudre les collisions.
4. Demande au manager de contraintes de résoudre les contraintes.
5. Demande à chaque objet du manager de résoudre les équations de la physique, via la fonction correspondant au solveur d'équation différentiel choisi (*RK4*, *Verlet*, ou *Euler*).

3.3 Relation scene-graph/moteur de physique

Un `OR_Object` d'**Orion3D** se situe habituellement dans un *scene graph*, qui rassemblent les objets d'une façon hiérarchisée, utile pour les animer de façon classique.

Cependant, lorsque l'on veut utiliser le moteur de physique pour calculer le mouvement de l'objet, on place un rigid body (classe `OR_RigidBody`) correspondant à l'objet dans le moteur de physique (classe `OR_PhysicEntityManager`). A partir de ce moment, c'est la moteur de physique qui va calculer la matrices de transformation de l'objet.

Il faut ainsi écrire des fonctions qui permettent de faire la transition lorsque un objet du scene graph est placé dans le moteur de physique, et réciproquement. Par exemple, il faut initialiser le rigid body en fonction de la position de l'objet, calculée à partir du scene graph.

De plus, il faut aussi savoir traduire une hiérarchie d'objets du scène graph, reliés par des liaisons particulières (pivot, glissière, etc.) en un ensemble de rigid body reliés par des contraintes dynamique.

La gestion des contraintes étant exposée à la fin de cet ensembles d'articles, nous n'entrerons pas dans les détails pour le moment.

3.4 Gestion du temps

Le dernier point sur lequel il faut insister est la gestion du temps.

Pour commencer, il convient d'implémenter un temps constant entre deux mises à jour. Ce temps, que l'on notera Δ_{update} est choisi par le moteur de physique. A chaque demande de rafraîchissement du moteur de physique, ce dernier calcule le temps écoulé depuis le dernier appel à cette fonction, notons le Δ_{time}. Il calcule alors le nombre de mise à jour par la formule :

$$\Delta_{total} = \Delta_{time} + \Delta_{rest}$$

$$NB\_UPDATE = [\Delta_{total}/\Delta_{update}]$$

$$\Delta_{rest} = \Delta_{total} - NB\_UPDATE * \Delta_{update}$$

La variable Δ_{rest} permet de reporté le temps non utilisé, d'une mise à jour sur l'autre (elle est initialisée à 0).

Ensuite, le moteur de physique doit réaliser NB_UPDATE mises à jour de façon à maintenir un temps réel constant entre deux mises à jour.

La dernière amélioration à effectuer est de choisir de façon dynamique la valeur Δ_Update. Par exemple, lorsque les objets sont au repos, il est inutile de rafraîchir le système de façon intensive. Les techniques pour calculer de façon dynamique le *time step* dans un solveur d'équations différentielles ont été présentées dans le premier article, il s'agit donc uniquement de les implémenter au sein d'un moteur de physique.

Conclusion

Bien moins complexe que la résolution complète des équations de la dynamique, les techniques présentées plus bas sont pourtant largement majoritaires dans les applications commerciales. En effet, bien que nécessitant les effort d'infographistes de talent (même en utilisant la cinématique inverse), elle est à la fois plus facile à mettre en oeuvre, et moins coûteuse en temps de calcul.

Bibliographie