8INF935 - Mathématiques et physique pour le jeu vidéo

Phase 3 - Journal de bord

Jules RAMOS (RAMJ27020100) - Mikhael CHOURA (CHOM23060100) - Gaëlle THIBAUDAT (THIG24539900) - Béatrice GARCIA CEGARRA (GARB19510105)

Table des matières

Introduction

L'objectif de cette troisième phase de développement était de poursuivre le développement du moteur physique en ajoutant une implémentation de matrices, de quaternions et de corps rigides avec leur intégrateur propre gérant leur orientation afin de réaliser un jeu de tir balistique avec des objets de formes plus complexes que des sphères.

I. Les matrices

Afin de mettre en place les matrices pour ce projet, des classes de matrices 3x3 et 4x4 ont été créées. Comme la dimension de ces matrices est fixe, des fonctions d'inversion et de calcul de déterminant spécifiques à la taille considérée ont été implémentées, plutôt que des fonctions génériques pour toutes les dimensions. Un array statique de float stocke les coefficients de la matrice, permettant un gain de mémoire comparé à un vector de la STL dynamique ou à un tableau en deux dimensions. Cela signifie cependant que la matrice est stockée dans une structure unidimensionnelle, ce qui a fortement influencé le code des méthodes.

Pour les matrices 3x3, en plus des constructeurs et opérateurs standards nécessaires à des calculs matriciels et des getters et setters standards, des getters et setters utilisant la logique ligne x colonne pour accéder aux éléments ont été implémentés pour faciliter l'utilisation de la classe. Des méthodes d'inversion de matrice, de transposée, de calcul de déterminant et de matrice adjacente (utile pour l'inversion) ont également été implémentées.

Le calcul du déterminant a pu être "hard codé" en utilisant la règle de Sarrus: det = aei + bfg + cdh - ceg - bdi - afh. La matrice adjacente a également pu être hard codée en utilisant la formule de calcul de déterminant d'un matrice 2x2 pour obtenir directement les coefficients. Des recherches sur le web ont permis d'obtenir ces formules et de les placer en commentaire dans le code, à la fois pour faciliter l'écriture et la lecture des méthodes. Pour l'inversion, la formule $A^{-1} = (1 / det(A)) * adj(A)$ utilisant les deux méthodes précédentes est triviale. Une vérification que le déterminant n'est pas nul est réalisée afin d'éviter des erreurs en runtime, et renvoient la matrice identité par défaut. On notera cependant que ce n'est qu'une sécurité et que la vérification de la valeur du déterminant devrait être réalisée avant l'appel de la méthode dans tout code l'utilisant.

Dans le cas des matrices 4x4, une grande proportion du code est similaire à celui des matrices 3x3 mais en considérant 16 réels au lieu de 9. Les principales différences viennent des méthodes de calcul du déterminant et de la matrice adjacente, qui ont requis l'utilisation de la classe de matrices 3x3. En effet, dans les deux cas, un calcul de déterminant de sous-matrices 3x3 était nécessaire. Afin de faciliter à nouveau l'écriture et la vérification du code, la sous-matrice 3x3 est créée et chacun de ses coefficient calculé séparément plutôt que de réaliser tous les calculs en un seul set de coefficients. Ces calculs représentent en conséquence un grand nombre de lignes de

code et ont demandé une certaine concentration à l'écriture et à la vérification, mais permettent d'obtenir les résultats recherchés.

II. Les quaternions et les tests unitaires

Les quaternions ont été programmés de manière classique avec une structure contenant un entier et un vecteur. De même les opérations sur ces derniers ont été codées suivant la théorie et en ayant recours aux opérations sur les vecteurs déjà implémentées.

La vérification de notre implémentation de quaternions, tout comme celle des matrices, s'est faite grâce à nos tests unitaires. Stockés dans un second fichier sln, ils comparent la valeur obtenue avec nos fonctions sur un quaternion ou une matrice d'exemple avec les valeurs théoriques correctes. Ils ont notamment permis de repérer des erreurs de calcul d'inversion de matrices tôt dans la phase et de les corriger.

III. Un second registre des forces

Afin de gérer les forces appliquées aux corps rigides et de pouvoir appliquer des forces avec des timings différents, ce qui nous permet notamment d'anticiper la phase 4 et la gestion de collisions, nous avons mis en place un second registre des forces pour les corps rigides. Ce dernier fonctionne comme le registre des forces pour les particules et le complète en calculant spécifiquement des forces qui s'appliquent à des corps rigides et pour lesquelles un objet de la classe Rigid plutôt que Particule doit être stocké.

Ce second registre permet de définir une nouvelle interface, *RigidBodyForceGenerator*, qui est utilisée pour définir nos forces d'entrées pour la phase 3 et pourra être utilisée en phase 4 pour définir différentes forces de façon analogue à *ParticuleForceGenerator* en phase 2.

IV. Physique des corps physiques

La classe Rigid représentant les corps rigides possède les attributs nécessaires aux calculs d'orientation, notamment: une particule pour représenter le centre de rotation, un vecteur pour représenter le décalage du centre de masse par rapport à ce centre de rotation, des vecteurs alpha et omega qui représente l'accélération et la vitesse angulaire, un accumulateur de torque, une matrice de tenseur d'inertie, un quaternion et une matrice de rotation. Un vecteur d'échelle, utilisé pour les représentations graphiques, est également inclus mais n'intervient pas dans les calculs physiques.

Le tenseur d'inertie est calculé dans le constructeur de la classe. Il est mis à jour tant que la position du centre de masse peut être modifiée, c'est-à-dire jusqu'au lancement de la boîte lorsque l'utilisateur a fini de donner les forces d'input. La formule générale pour un parallélépipède rectangle a été utilisée, et le théorème des axes parallèles est utilisé pour calculer ensuite le tenseur d'inertie avec décalage du centre de masse. Le tenseur d'inertie d'une boîte de dimensions a*b*c est obtenu comme suit:

$$\begin{pmatrix} \frac{1}{12}m\left(b^2+c^2\right) & 0 & 0 \\ 0 & \frac{1}{12}m\left(a^2+c^2\right) & 0 \\ 0 & 0 & \frac{1}{12}m\left(a^2+b^2\right) \end{pmatrix}$$ (source).

Étant donné que le seul calcul demandant la position du centre de masse est le théorème des axes parallèles, qui demande sa position par rapport au centre de l'objet, il a été choisi de représenter le centre de masse par un simple vecteur et de donner les données inhérentes au corps rigide, telles que sa masse, à la particule représentant son centre d'orientation. C'est par conséquent à cette particule que la gravité et la friction de l'air (friction dynamique) sont appliquées dans le gameworld.

L'accumulateur de torque a un rôle analogue à celui de l'accumulateur de forces d'une particule: il permet de calculer le torque total du corps pour une frame. Une fois ce torque calculé, il est multiplié avec le tenseur d'inertie pour obtenir l'accélération angulaire de la frame, qui est ensuite intégrée pour donner omega puis le quaternion rotation du corps rigide pour la frame. Une fonction de conversion permet d'obtenir la matrice de rotation avec ce quaternion.

Ces calculs répétés à chaque frame sont appelés par le second registre de forces au même moment que ceux du premier registre, à nouveau de manière analogue à la phase 2.

V. Affichage Graphique

Dans chaque classe héritant de la classe *Rigid,* nous implémentons une fonction *draw*, permettant de dessiner le solide sur une frame. Cette méthode permet de tirer parti des aspects de polymorphisme, et ainsi de pouvoir facilement créer d'autres corps rigides, seulement en changeant cette fonction, ainsi que la définition de la variable J. Dans notre cas, nous avons créé un cube, avec une teinte différente de couleur sur chaque face, permettant de mieux visualiser la rotation.

Un des principaux soucis de cette partie fut la compréhension de comment faire tourner un objet sur OpenFrameworks. En effet, si nous prenions simplement la matrice de rotation de notre solide pour une frame donnée, nous avions une rotation dans le sens inverse au mouvement souhaité. Nous avons donc procédé de manière itérative pour découvrir d'où venait le problème, et nous avons compris que ce n'était pas un problème de calcul mathématique, mais un problème de traduction entre notre calcul et l'affichage d'openFrameworks. En prenant le conjugué du quaternion pour faire cette traduction, nous obtenons le visuel souhaité.

VI. Input de l'utilisateur

Pour la réalisation des Inputs, nous avons réutilisé ce que nous avions fait lors des précédentes phases, et nous les avons améliorés en ajoutant des sliders, permettant de contrôler les valeurs de position et de direction. Ces sliders sont des outils de la librairie openFrameworks qui ne sont pas intégrés par défaut au projet, ce qui nous a fait perdre un peu de temps pour la compréhension de ce problème, et pour le partage via gitHub. Nous avons donc rajouté un dossier nommé *ofGui*, qui se charge de faire le lien entre notre projet et les fichiers de cet addon.

VII. Répartition des tâches

Béatrice	Quaternions, intégrateurs physiques des corps rigides, calcul de force, tenseur d'inertie
Jules	Matrices, registre des forces et interface des forces, gameworld, nettoyage de code et debug
Mikhael	Tests unitaires, affichage des boîtes, gestion de la caméra, des inputs et du GUI
Gaëlle	Création du document du journal de bord