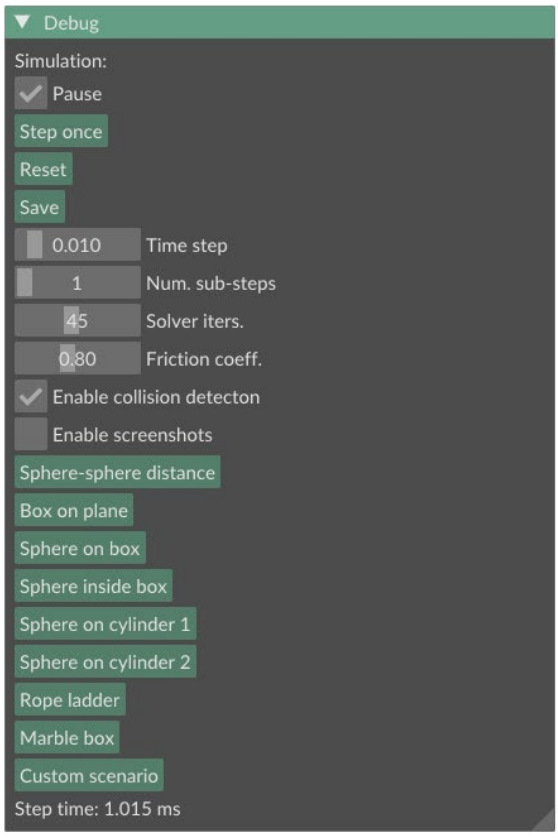
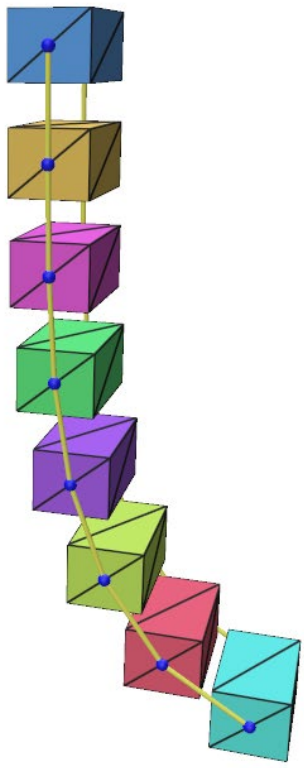
Devoir 2

Simulation des corps rigides





Rédigé par Sheldon Andrews
Version 1.2 (19 juin 2024)

Introduction

Lors des dernières semaines, nous avons vu plusieurs aspects de la simulation de corps rigides: l'intégration numérique, les contraintes, la modélisation de contact et de frottement, et la détection des collisions. Dans ce devoir, vous allez mettre en œuvre le code d'un simulateur des corps rigides qui comprend toutes ces fonctionnalités.

Les endroits où vous devez implémenter du code sont indiqués par un commentaire `// TODO` dans le code de départ.

Code de départ et dépendances

Compilateur C++. Vous devrez installer un compilateur C++. Cela dépend de votre plate-forme et de votre système d'exploitation. Sous Windows, je recommande *Visual Studio 2019* ou *2022*, que vous pouvez télécharger et installer gratuitement. Sous Linux, *gcc 7.x* est supporté. Sous macOS, *Xcode 11* est supporté. Vous pouvez trouver plus d'informations sur des compilateurs et des plateformes spécifiques dans les liens ci-dessous.

des compilateurs et des plateformes spécifiques dans les liens ci-dessous.

Eigen. Une copie de la bibliothèque d'algèbre linéaire Eigen (http://eigen.tuxfamily.org/index.php) est fournie avec le projet. Plusieurs des structures de données du code utilisent Eigen pour stocker les matrices et les vecteurs. Un guide sur l'utilisation d'Eigen est disponible ici:
https://eigen.tuxfamily.org/dox/GettingStarted.html
… et voici le guide de référence rapide :
https://eigen.tuxfamily.org/dox/group__QuickRefPage.html

Polyscope. Une copie de Polyscope (https://polyscope.run/) est également incluse, qui est utilisé pour visualiser et pour rendre les simulations. Note : Polyscope utilise ImGUI pour l'interface d'utilisateur. Vous êtes libre (et encouragé) de créer votre propre interface pour tester votre code. Polyscope est capable de visualiser des nuages de points, des maillages polygonaux et des champs vectoriels. Il est fortement recommandé d'utiliser ce cadriciel pour vous aider à déboguer votre code.

CMake. Les fichiers de projet pour compiler le code en C++ sont produit par CMake. Si vous travaillez sous *Windows*, avec *Visual Studio* :

1. Exécuter l'interface graphique de CMake.
2. Mettre le chemin du code source au dossier `MTI855-Devoir02` et mettre le chemin pour compiler les binaries au dossier `MTI855-Devoir01/build`
3. Appuyer sur le bouton *Configurer* et spécifier le compilateur et la plateforme du projet (par ex. *Visual Studio 16 2019* et *x64*)
4. Appuyer sur le bouton *Générer* pour créer les fichiers du projet.

Rédigé par Sheldon Andrews
Version 1.2 (19 juin 2024)

5. Dans le dossier `MTI855-Devoir02/build`, identifier le fichier `MTI855-devoir02.sln` et l'ouvrir avec *Visual Studio.*
6. Dans Visual Studio, faire CTRL-B pour compiler et F5 pour exécuter le programme.

Interaction avec la souris

Roulette de la souris : mode zoom
Bouton gauche : mode de navigation
Bouton droit : mode de mouvement dans le plan de vue

Il n'y a pas de ressort de souris :(

Objectifs

Intégration numérique des corps rigides (20%)

Dans la fonction `RigidBodySystem::step()`, bouclez sur chaque `RigidBody` dans le tableau `m_bodies` et utilisez une intégration semi-implicite pour mettre à jour leur configuration (position et orientation). Tout d'abord, calculez les vélocités mises à jour en utilisant les forces et les couples (forces angulaires), puis les positions et orientations en utilisant les nouvelles vélocités.

Notez que contrairement aux notes de cours de Baraff, vous devrez directement intégrer les vélocités plutôt que d'utiliser les quantités de mouvements **`L`** et **`P`**:
Ex.

```
body->xdot  += dt * (1/body->m) * body->f
body->omega += dt * (body->Iinv * body->tau)
```

Faire attention aux corps qui où `fixed == true` ! Ils doivent être traités pour ne pas avoir de mises à jour de leur position ou de leurs vélocités.

Calculer le jacobien de la contrainte de distance (10%)

Mettez en œuvre le code pour calculer le jacobien et l'erreur de la contrainte dans la fonction `Distance::computeJacobian()` qui est trouvée dans le fichier `Distance.cpp`.

L'erreur de contrainte peut être stockée dans `Joint::phi[0]` (de la classe parent). La matrice jacobienne de cette contrainte est une matrice 1x12, divisée en deux parties : un bloc pour `body0` et un autre bloc pour `body1`. Ceux-ci sont stockés respectivement dans `J0` et `J1`.

Précalculez les termes `J0Minv` et `J1Minv`. Ce sont simplement les blocs jacobiens multipliés par l'inverse de la masse et de l'inertie de chaque corps. Une astuce pour calculer

les termes `J * inv(M) * J^T` : les divisez en calcul des termes linéaires et angulaires. Par ex., pour calculer le terme `J * inv(M) * J^T`, commencez par body0 du contact :

```
J0Minv(1..3) = J0(1..3) * (1/body0->mass)
J0Minv(4..6) = J0(4..6) * body0->Iinv
```

Les étapes sont identiques pour `J1Minv`, mais en utilisant la masse et le jacobien de `body1`.

Notez que chaque bloc du jacobien est une matrice 1x6, et la fonction `block()` de la classe `Eigen::Matrix<>` peut être utilisé pour accéder aux sous-blocs 3x3,
par ex. `J0.block(0, 0, 1, 3)` accède à la première ligne et trois premiers colonnes de la matrice `J0` .

Détection de collision sphère-sphère, sphère-boite et cylindre-sphère (20%)

Implémentez le code d'un test de collision sphère-sphère dans la fonction `collisionDetectSphereSphere()` et de collision sphère-boite dans la fonction `collisionDetectSphereBox()` (voir le fichier `CollisionDetect.cpp`)

Les fonctions doivent calculer le point de contact `p`, la normale de contact `n` (qui est un vecteur unitaire) et la pénétration `pene`, qui sont fournis comme paramètres de passage par référence. Si une collision est détectée, ces fonctions doivent ajouter un nouveau `Contact` au tableau `m_contacts`.

Calculer le jacobien de contact et de frottement (15%)

En utilisant la normale et la position de contact, calculez le jacobien de la contrainte pour chaque contact. Implémentez le code dans la fonction `Contact::computeJacobian()` (voir le fichier `Contact.cpp`)

Comme la contrainte `Distance`, le jacobien de `Contact` est stocké en deux blocs - `J0` et `J1` - qui correspondent au vecteur 3 x 6 pour le premier et le deuxième corps, respectivement. Note : La première ligne correspond à la contrainte de non-interpénétration, et la deuxième et troisième ligne dans la matrice jacobienne sont les contraintes de frottement.

Solveur sans matrice (20%)

Dans le fichier `SolverPGS.cpp`, implémentez l'algorithme de *Projected Gauss-Seidel (PGS)* **sans matrice**. Le solveur utilise le complément de Schur de les équations de mouvement contraintes, mais la matrice complète ne doit jamais être assemblée. Le nombre

d'itérations de Gauss-Seidel est par defaut `maxIter` = 10, mais cette valeur peut être ajusté dans le GUI.

Utilisez la pénétration, qui est stockée dans `Contact::phi[0]`, pour stabiliser la non-interpénétration en utilisant la technique de Baumgarte. Supposez que le paramètre de réduction d'erreur (ERP) gamma = 0.7, bien qu c'est un paramètre que vous souhaiterez peut-être ajuster

Lorsque la fonction `SolverPGS::solve()` retourne, les forces de contrainte correctes seront stockées dans la variable membre `lambda` de chaque `Contact` ou joint `Distance`.

IMPORTANT lors de la résolution des impulsions d'une contrainte ou d'un contact, vous devez tenir compte des impulsions des joints et des couplées en bouclant les contraintes et les contacts qui partagent `body0` ou `body1`

(Astuce: utilisez `RigidBody::contacts[]` et `RigidBody::joints[]` pour vous aider)
(Astuce : pour les corps fixes, leur masse est effectivement infinie.)

Créer un scénario de test intéressant (10%)

Ajoutez votre propre scénario en implémentant la fonction `Scenarios::createCustomScenario()` dans le fichier `Scenarios.h`. La copie du code à partir des autres fonctions est correcte… mais soyez créatif ! Tentez de démontrer toutes les fonctionnalités de votre simulateur.

Style de programmation (5%)

N'oubliez pas d'ajouter votre nom, code permanent et votre adresse e-mail en haut des fichiers que vous modifiez.

Évaluation

Grille d'évaluation

Évaluation	Pondération
Implémenter le code manquant dans la fonction `RigidBodySystem::step()` pour avancer l'état des corps rigides	10
Calculer le jacobien de la contraint de distance dans la fonction `Distance::computeJacobian()`	10
Implémenter le test de collision sphère-sphère dans les fonctions `CollisionDetect::collisionDetectSphereSphere()` et `CollisionDetect::collisionSphereBox()` qui retourne le point de contact, normal et pénétration	10
Implémenter le test de collision sphère-sphère dans la fonction `CollisionDetect::collisionCylinderSphere()`	10
Calculer le jacobien de la contraint de contact avec frottement dans la fonction `Contact::computeJacobian()`	20
Implémenter l'algorithme de PGS sans matrice dans la fonction `SolverPGS::solve()`	25
Créer un scénario de test intéressant	10
Style de programmation (commentaires, clarté du code, etc.)	5
Total	**100**