Université de Lille

MASTER 2 INFORMATIQUE
Premier semestre

FACULTÉ DES SCIENCES ET TECHNOLOGIES
Département Informatique

Modélisation 3D Avancée

Rapport SOFA

BARCHID Sami

Année académique 2019-2020

Introduction

Ce rapport présente les travaux réalisés dans le cadre de notre cours avec Monsieur Christian Duriez. Le rapport est divisé en trois parties, correspondant aux trois travaux à réaliser :

- **Modèle du pendule explicite :** travail sur l'implémentation d'un modèle de pendule par intégration explicite et analyse de son comportement suivant les valeurs de ses paramètres.

- **D'un pendule implicite à la simulation d'un drap :** utilisation d'un pendule implicite, analyse de son comportement suivant les valeurs de ses paramètres, comparaison avec le pendule explicite précédent et modélisation d'un drap grâce à ce modèle de pendules implicites.

- **Simulation d'un robot mou avec SOFA** : réalisation d'un workshop SOFA avec l'utilisation d'un robot mou.

Note : le rapport est plus long que ce qui a été prévu en cours (1,5 pages). Cependant j'ai décidé d'être un peu plus exhaustif afin de réviser cette partie pour les partiels en parallèle du rapport et parce que j'ai trouvé cette partie du cours intéressante. Je présente mes excuses d'avance pour la longueur réglementaire du rapport dépassée.

Modèle du pendule explicite

Dans ce travail, nous avons dû implémenter, à l'aide d'un code Python, la simulation d'un pendule à l'aide d'un schéma d'intégration explicite. Nous commençons par expliquer de manière théorique les schéma d'intégration, ensuite on décrit l'implémentation de l'intégration explicite du pendule, puis nous analysons son comportement suivant la modification de ses paramètres.

Schémas d'intégration

Pour la réalisation d'une simulation dynamique, il faut être capable de discrétiser l'évolution d'un système dans le temps selon un pas de temps dt défini à l'avance (c'est un paramètre à fixer lors de la construction de la simulation). Un schéma d'intégration est la méthode numérique permettant d'approximer la solution d'un système d'équations différentielles à l'instant $t+dt$ à partir de la situation courante (instant t). Concrètement, le schéma d'intégration va servir à estimer la vitesse v et la position x à l'instant $t+dt$.

Il existe deux types de schéma d'intégration :

- **Schéma d'intégration explicite :** la position x et la vitesse v à l'instant $t+dt$ sont calculées grâce à la situation sur le pas de temps précédent (instant t). L'intégration explicite est plus rapide car les calculs sont plus simples à résoudre. Cependant, le système est conditionnellement stable. C'est-à-dire que le pas de temps dt doit être plutôt petit sans quoi on observe des problèmes de stabilités. Ainsi, les calculs à réaliser sont plus nombreux mais aussi plus simples.

- **Schéma d'intégration implicite :** la position x et la vitesse v à l'instant $t+dt$ dépendent de l'accélération a à ce même instant $t+dt$. On n'utilise donc pas l'état du système à l'instant précédent. L'intégration implicite est généralement plus lente car le système d'équations à résoudre est plus complexe (il est non-linéaire). Cependant, ce schéma est beaucoup plus stable donc on peut se permettre un pas de temps dt plus grand.

Implémentation de l'intégration explicite

Le modèle de pendule que nous simulons se présente sous la forme d'un maillage de plusieurs nœuds (que l'on peut adapter) et d'un *MechanicalObject* qui représente chaque nœud comme un objet rigide avec 3 DOFs (**D**egree **O**f **F**reedom = « degré de liberté »). Ainsi, le modèle déformable, dont le nombre de DOF est théoriquement infini, est discrétisé avec un ensemble de nœud « rigides ».

De plus, chaque « edge » (= liaison entre deux nœuds) est représenté comme un ressort qui exerce une force sur les nœuds qu'il relie. Pour rappel, on peut modéliser les forces qu'exerce un ressort sur les deux nœuds i et j qu'il relie de la manière suivante :

$$f_i = k\left(\|x_j - x_i\| - l_0\right) \cdot \frac{x_j - x_i}{\|x_j - x_i\|} \quad \text{et} \quad f_j = -f_i$$

où :

- f_i et f_j sont respectivement la force appliquée au nœud i et j .
- x_i et x_j sont respectivement la position courante du nœud i et j .
- l_0 est la longueur du ressort au repos.
- k est le coefficient de raideur défini au départ de la simulation.
- *Note* : $\|x_j - x_i\|$ donne la longueur actuelle du ressort. Ainsi, $\|x_j - x_i\| - l_0$ donne le ΔP vu en cours (= la différence de longueur de ressort avec sa longueur au repos).

Ainsi, pour résumer grossièrement, la somme des forces à l'instant t qui s'exercent sur un nœud du mesh est :

$$\sum F = \text{gravité} + \text{la somme des forces des ressorts attachés au noeud}$$

Et cette somme des forces permet de retrouver la valeur de a , l'accélération à l'instant t grâce à la relation :

$$m \cdot a = \sum F \Leftrightarrow a = \frac{\sum F}{m}$$

avec m qui est la masse d'un nœud (définie au départ de la simulation).

Maintenant que l'on connaît la valeur de a pour l'instant t , il est aisé de calculer les valeurs de v et x pour l'instant suivant $t+dt$:

$$v_{t+dt} = v_t + (a_t \cdot dt) \text{ et } x_{t+dt} = x_t + (v_{t+dt} \cdot dt)$$

où

- v_t est la vitesse à l'instant t .
- a_t est l'accélération à l'instant t .
- x_t est la position à l'instant t .

Résumé de l'algorithme

On résume les étapes à réaliser décrites précédemmentpour donner l'algorithme à implémenter pour réaliser l'intégration explicite pour chaque noeud :

```
TANT QUE [simulation en cours] :
| CALCULER  ΔF
| CALCULER  a_t = ΣF / m
| CALCULER  v_{t+dt} = v_t + (a_t · dt)
| CALCULER  x_{t+dt} = x_t + (v_{t+dt} · dt)
```

Variation des paramètres

Comme décrit dans la partie « implémentation », on peut faire varier plusieurs paramètres :

- La masse m d'un nœud.
- Le coefficient de rigidité k des ressorts.
- Le pas de temps dt utilisé pour la simulation.
- La force de gravité g utilisée pour la simulation.

Lors du test, on fixe un ou plusieurs points pour que le maillage « pende », comme un pendule.

Les observations faites sont les suivantes :

Variation du pas de temps

- Comme dit plus haut, un pas de temps très haut (dt = 0.4) rend la simulation instable, ce qui amène à un comportement du pendule faux. Concrètement, on voit que les nœuds du pendule tombent quasi instantanément.
- Un pas de temps très réduit (dt = 0.005) crée une simulation stable et une animation assez rapide.
- Un pas de temps très (voire trop) réduit (dt = 0.00001) crée une simulation stable mais le rendu est trop lent (il ne se passe quasiment rien à l'écran).
- En résumé, pour le pas de temps, il faut trouver un dt optimal pour le rendu. **Pour la suite des analyses, je choisis de garder un dt = 0.005, pour un rendu assez rapide et stable**.

Variation du coefficient de rigidité

- Un coefficient trop grand (k = 5000) entraîne une instabilité du système.

- Un coefficient nul (k = 0), comme on pourrait s'y attendre, annule l'effet de ressort. À éviter donc.

- Un coefficient très faible (k = 0.1) montre un ressort très peu rigide ce qui donne un très grande amplitude de mouvement aux nœuds.

- **On retient un « juste milieu » avec k = 1500**, qui donne un pendule avec des ressorts assez rigide pour que les nœuds bougent sans qu'ils sortent de notre champ de vision.

Variation de la masse

- On observe, avec une masse importante (m = 5000) que les ressorts agissent comme lorsque le coefficient de rigidité était faible, c'est à dire que les ressorts s'étirent plus car les nœuds sont plus lourds.

- Une masse trop légère (m = 0.001) entraîne un problème de stabilité, comme lorsque le coefficient de rigidité était trop haut.

- **On définit une masse « juste milieu » avec m = 1**, qui donne un pendule dont les mouvements restent visibles sans devoir dézoomer.

Variation de la gravité

- La variation terrestre (g = 9.81, environ) donne un comportement normal de pendule.

- J'ai testé avec une très grande valeur de gravité (g = 1000). On observe que le pendule bouge moins et tend à rester en bas. Ce comportement est logique puisque la force de gravité (dans l'axe Y vers le bas, donc) est plus forte.

D’un pendule implicite à la simulation d’un drap

Dans ce travail, nous avons tout d’abord modifié le schéma d’intégration de notre pendule pour utiliser l’intégration implicite offerte par SOFA (« *EulerImplicitSolver* »). Ensuite, nous avons utilisé un maillage complexe organisant des nœuds en sorte de « drap ». En dernière étape, nous avons ajouté un modèle de collision au drap et un obstacle, de sorte que le drap puisse « envelopper » l’obstacle. Chaque étape de cette partie a été testée en faisant varier les paramètres pris en compte.

Pendule implicite

Pour créer ce pendule implicite à l’aide des composants disponibles dans SOFA, on modifie le pendule explicite de la manière suivante :

- Suppression du controller python custom qui permettait de faire l’intégration explicite.

- Ajout des composants suivants :

  - **EulerImplicitSolver** : décrit le schéma implicite pour solutionner le système d’équation $Ax=b$. Ce solver permet aussi de définir un amortissement de rayleigh grâce aux paramètres *« rayleighStiffness »* et « *rayleighMass* ».

  - **SparseLDLSolver** : après que le schéma d’intégration EulerImplicit ait été défini, on définit le solver permettant de résoudre le système $Ax=b$. Il existe deux catégories d’algorithmes pour régler cela : les solvers directs et les solvers itératifs. Le **SparseLDLSolver** est un solver direct, c’est-à-dire qu’il approxime la solution du système en une étape, contrairement aux solvers itératifs qui réalisent une approximation en améliorant la précision en plusieurs itérations. L’avantage d’utiliser un direct solver ici est qu’il est efficace pour des systèmes linéaires de petites tailles, comparable à celui du pendule.

Note :le SparseLDLSolver utilise la méthode de la décomposition LDL pour la résolution.

Ces composants permettent de définir l'algorithme final pour notre pendule implicite. Les autres composants définis dans le code permettent de décider des points fixes ou des contraintes telles que le coefficient de rigidité des ressorts du mesh.

Les variations de paramètres testées font observer plusieurs choses :

- Par rapport au schéma d'intégration explicite, le pas de temps peut être plus grand sans qu'il y ait de problème de stabilité, comme il a été dit précédemment. Ainsi, un pas de temps $dt = 0.4$ qui entraînait des instabilité pour l'explicite (voir partie précédente) ne crée pas d'instabilité en implcite.

- En ce qui concerne les variations de rigidité de ressort et de masse, le comportement est similaire à celui du modèle d'intégration implicite (c'est-à-dire que les mouvements du pendule sont plus ou moins amples suivant le rapport poids des nœuds et rigidité des ressorts).

Drap implicite

Le drap conserve toujours le modèle du pendule implicite. Ce qui change est simplement l'organisation du maillage de notre simulation : on organise une série de nœuds avec des edges entre les nœuds voisins. Le maillage est réalisé avec un **MeshObjLoader**, qui prend un fichier obj en paramètre pour organiser le maillage. Le maillage chargé est organisé comme un carré de plusieurs nœuds, ce qui donne l'apparence et l'effet d'une couverture.

Les variations de paramètres font observer plusieurs choses :

- Pour créer le drap, il a fallu un très grand assemblage de nœuds, ce qui entraîne une masse de calculs très importante. De ce fait, lorsque l'on utilise un pas de temps très court ($dt = 0.001$), le nombre de FPS chute par rapport à avant. Ceci montre que le schéma d'intégration implicite est plus lourd, ce qui entraîne un ralentissement du modèle.

- Les autres observations (sur la masse, la rigidité des ressorts, etc) seront développée dans la sous-partie suivante.

Drap implicite et modèle de collision

On ajoute un système de collision au drap implicite ainsi qu'un « obstacle » fixe (grâce à un OBJ loader, etc). Plusieurs composants SOFA sont requis pour ajouter ce système de collisions. Ces composants sont détaillés ici.

Modéliser les contacts

L'important ici est l'ajout des composants SOFA qui permettent de modéliser les contacts :

- **FreeMotionLoop** : composant définissant la boucle d'animation de l'animation. Ce composant est obligatoire dans toute simulation dans SOFA. Un composant de boucle d'animation a pour rôle définir l'ordre dans lequel les étapes de la simulation seront résolues (ex : résolution des systèmes d'équation, gestion des collisions, etc). La « FreeMotionLoop » sépare la boucle en deux étapes principales :
  - L'étape « free motion » en premier, qui construit et résout tous les systèmes linéaires sans prendre les contraintes en compte.
  - L'étape « correction step » ensuite, qui résout les contraintes et collisions grâce aux multiplicateurs de Lagrange.
- **GenericConstraintSolver** : composant chargé de résoudre le problème des contraintes (algorithme de Gauss-Seidel). Dans notre simulation, ce solver sera appelé lors de l'étape de « correction step » de notre FreeMotionLoop.
- **GenericConstraintCorrection** : composant de « ConstraintCorrection », requis pour l'utilisation d'un **FreeMotionLoop**. Sans entrer dans les détails, les composants de ConstraintCorrection sont requis dans la simulation pour pouvoir définir la manière dont une matrice utilisée dans les multiplicateurs de Lagrange est calculée.

Pipeline de collision

Dans SOFA, la phase de collision est gérée séparément de la simulation physique. Cette phase de collision est gérée par une série de composants SOFA qui définissent ensemble le « collision pipeline ». Ce « collision pipeline » est défini avec par deux étapes successives : la détection de collision et la réponse à la collision. Les composants qui gèrent le pipeline de collision dans notre simulation sont :

- **CollisionPipeline :** composant qui déclare le pipeline de collision
- **BruteForceDetection** : composant qui définit la détection de collision (avec la broad phase et la narrow phase, etc).
- **LocalMinDistance :** composant qui définit les distances minimales au bout desquelles on détecte une détection avec un autre mechanical object. Il faut ainsi faire attention à définir une distance assez grande pour ne pas qu'un objet trop rapide soit.
- Pour définir la réponse à la collision, on a le choix entre deux composants :
    - **CollisionResponse** (par défaut) : composant par défaut pour la collision. Modélise une réponse de collision en pénalité (ajout de ressorts pour la collision).
    - **RuleBasedContactManager** : composant qui modélise la réponse de collision basé sur des contraintes avec multiplicateurs de Lagrange, qui seront gérés en même temps que les autres forces et contraintes par les solvers définis précédemment.

Expérimentations

La variation des paramètres (et du composant de réponse de collision) donnent les observations suivantes :

- On observe une première limitation au niveau de la détection de collision. En effet, en prenant une distance minimale très courte, on remarque que le drap se déplace trop vite pour le détecteur, ce qui fait qu'il traverse l'obstacle sans que la collision soit détectée.

- Lorsque l’on utilise une masse très importante, on remarque que le drap a tendance à échapper plus simplement à la détection de collision.

- Il y a une grosse limitation du modèle de construction de drap avec les ressorts :

    - Lorsque les ressorts ne sont pas assez rigides par rapport au poids des nœuds, ils s’étendent énormément (comme on a pu voir dans les parties précédentes). Cependant, cela pose un gros problème dans ce qu’on cherche à modéliser étant donné qu’un drap ne s’étend pas de cette manière.

    - On en conclut qu’il faut créer des ressorts assez rigides pour que le modèle paraît cohérent.

- Une autre limitation est celle du rebond infini : on remarque que la couverture va rebondir sur l’obstacle à l’infini (des tout petits rebonds), ce qui ne donne pas un effet visuel satisfaisant. Il aura fallu donner un critère au bout duquel on arrête ce comportement de rebonds pour nous donner une simulation qui paraît convaincante.

- **Note importante** : il n’a pas été possible de tester l’autre mode de réponse de collision (à savoir, **RuleBasedContactManager**) car celui-ci entraînait un bug de SoFA. En résumé, il faisait planter le logiciel.

Simulation d’un robot mou avec SOFA

Le travail dans cette partie consiste à participer au Workshop présenté en exemple dans SOFA : « worshop.pyscn ». Ce workshop permet d’appliquer les bases apprises dans les travaux précédents pour utiliser un robot déformable : le « Tripod ».

Figure – Image de Tripod

Le Tripod possède une pièce déformable en silicone de trois bandes qui chacune tenue par un bras actionneur motorisé. Il est possible de manipuler ces moteurs grâce à la simulation. De par l’activation de ces trois moteurs, on peut déformer le composé de trois bandes afin de déplacer l’objet placé au sommet des bandes. L’objectif final est de faire bouger une bille dans un labyrinthe placé sur le Tripod grâce à une interaction avec la simulation (qui fait directement déplacer le robot).

La simulation du Tripod est fournie avec trois prefabs qui sont utilisés pour modéliser le Tripod et son comportement :

- Un « ServoMotor » S90 qui exercie une force grâce à une rotation sur un seul axe.

- Un « ActuatedArm » qui modélise un ServorMotor (donc, qui utilise le prefab du ServorMotor) qui active un bras d'actionneur.

- Le « Tripod » qui est le prefab final. Il est composé de trois « ActuatedArm ». De cette manière, on peut actionner les bandes et déplacer le labyrinthe.

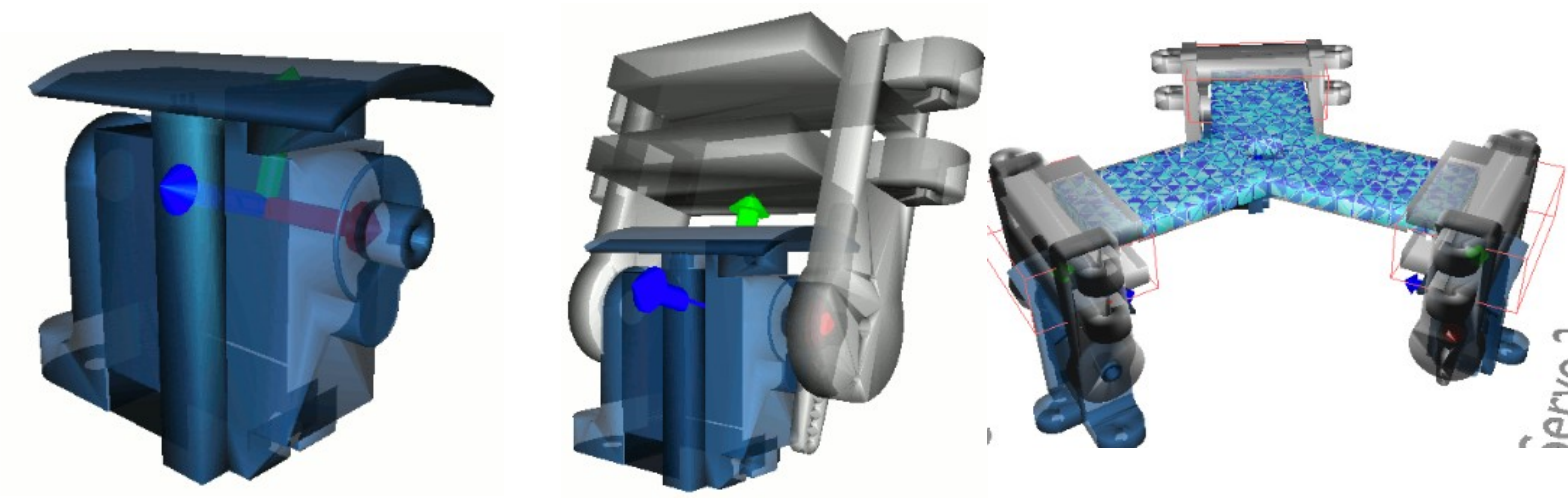

Figure – De gauche à droite : ServorMotor, ActuatedArm et Tripod

En résumé, le workshop s'organise en plusieurs étapes successives :

1. **Mise en place de la pièce de silicone déformable** : modèle visuel, chargement du mesh.

2. **Ajout du modèle mécanique sur la pièce de silicone :** modèle mécanique, utilisation du EulerImplicitSolver + SparseLDLSolver, utilisation d'un modèle élastique.

3. **Ajout d'une « fixing boxe » pour tester le modèle mécanique** : définir une zone en forme de boite sur le corps élastique qui fixe son contenu, afin de pouvoir visualiser l'elasticité produite par le modèle.

4. **Assemblage du Tripod** : ajout du prefab « Tripod », rigidification du bout de chacune des trois bandes afin de pouvoir être fixées sur les bras actionneur du Tripod, liaisons des nouvelles parties rigides du silicone aux bras articulés.

5. **Interaction dans la simulation** : ajout de la prise en compte de touches du clavier pour piloter les bras actionneurs.

6. **Ajout de la gestion des contacts (collisions)**

7. **Connexion avec le Tripod physique** : mise en place de la communication entre la simulation et le vrai robot à travers un port USB auquel est branché le Tripod.

8. **Cinématique inverse** : mise en place de la cinématique inverse.

Conclusion

Dans ces sessions de travail sur SOFA, nous avons pu voir le fonctionnement interne d’une simulation 3D avec un outil de haut niveau. Nous avons pu aussi appréhender les problématiques de modéliser un système physique et d’interagir avec lui (human in the loop, etc).

Avec le recul, je trouve qu’il aurait été apprécié pour la plupart des élèves de bénéficier de cours plus approfondis sur les thématiques introduites par ces sessions. Je pense même qu’un cours à part entière serait le bienvenu, au vu de l’intérêt porté par la classe.