HAL
open science

Application de SOFA à la simulation des mouvements du moteur MCE-5 VCRi

Jérémy Jaussaud

To cite this version:

Jérémy Jaussaud. Application de SOFA à la simulation des mouvements du moteur MCE-5 VCRi. [Stage] 2010, pp.35. inria-00514435

HAL Id: inria-00514435

https://inria.hal.science/inria-00514435v1

Submitted on 2 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

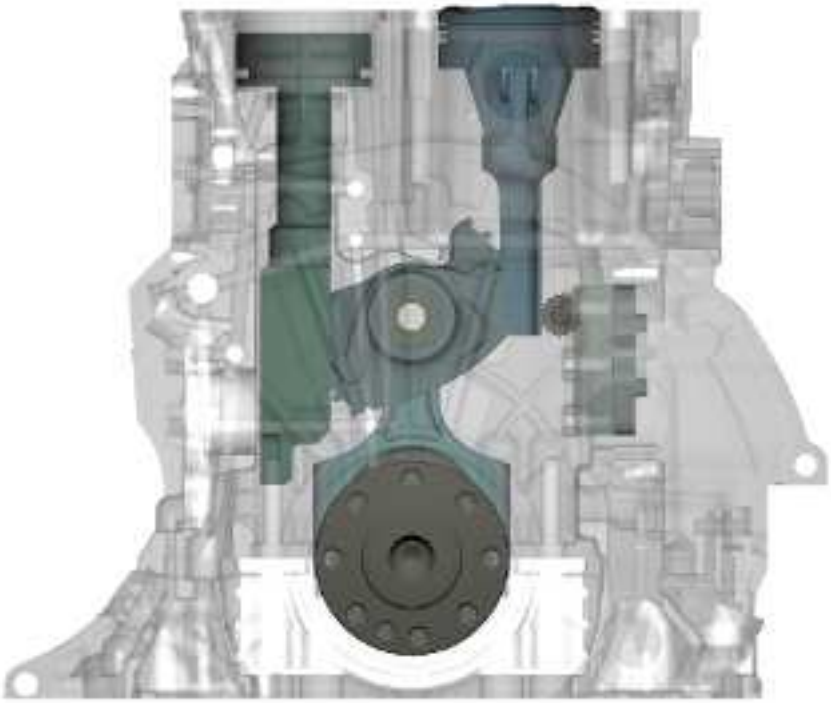
L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Application de SOFA à la simulation des mouvements du moteur MCE-5 VCRi

Jérémy JAUSSAUD

sous la direction de François FAURE



INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Application de SOFA à la simulation des mouvements du moteur MCE-5 VCRi

Jérémy JAUSSAUD * † ‡
sous la direction de François FAURE * † ‡

Équipe Evasion

Septembre 2010 — 35 pages

* INRIA Rhône-Alpes
† Université Joseph Fourier
‡ Laboratoire Jean Kuntzmann

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Table des matières

Page de garde **1**

Table des matières **2**

1 Introduction **3**
1.1 Contexte . . . 3
1.2 Résultats attendus . . . 4
1.3 Problématique . . . 4
1.4 Présentation de SOFA . . . 5

2 PairwiseCudaRasterizer **8**
2.1 CudaRasterizer . . . 8
2.2 Problématique . . . 8
2.3 Principe de fonctionnement . . . 10
2.4 Mise en œuvre . . . 11
2.5 Paramètres . . . 14
2.6 Limitations et développements futurs . . . 17

3 Entrées / Sorties **19**
3.1 CSV Loader . . . 19
3.2 GenerateRigidMass . . . 19
3.3 Loader Switch . . . 22
3.4 LMConstraint Monitor . . . 24
3.5 Config . . . 25

4 Problèmes Numériques **26**
4.1 CudaLDIContactLMConstraint . . . 26
4.2 Discrétisation . . . 26
4.3 Interpénétration . . . 27
4.4 Objets déformables . . . 27

5 Synthèse **28**
5.1 PairwiseCudaRasterizer . . . 28
5.2 Échange de données . . . 28
5.3 Simulation d'une partie du moteur . . . 29
5.4 Simulation du moteur complet . . . 29

6 Annexes **31**
A Export de données . . . 31
B Paramètres clefs . . . 32
C Réinstallation des *drivers* CUDA . . . 33

Références **35**

1 Introduction

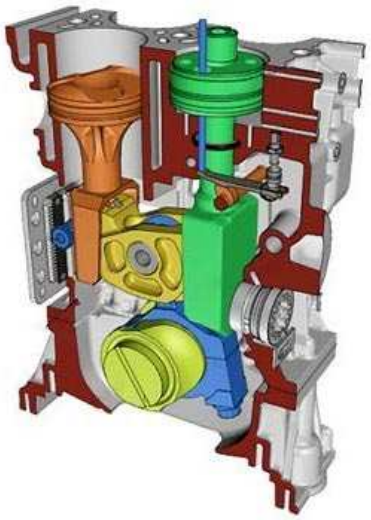
1.1 Contexte

Organisme d'accueil

Mon organisme d'acceuil est l'INRIA Rhônes-Alpes. Ce stage se fait officiellement dans le cadre d'un CDD de 5 mois au sein de l'équipe de recherche EVASION du laboratoire LJK. Mon contrat a commencé le 6 Avril et j'ai dès ce jour intégré l'équipe de recherche dans ses locaux de l'INRIA. Mon maître de stage est François Faure, enseignant-chercheur à l'Université Joseph Fourier (UJF) Grenoble 1, et dans l'équipe de recherche EVASION.

Cette équipe de recherche participe entre autres choses au développement du *framework* de simulation SOFA[1] avec deux autres équipes équipes de l'INRIA.

MCE-5 Development



La société MCE-5 Development[2], le client, est une entreprise dont le siège se situe à Lyon. Celle-ci développe depuis maintenant plus de 10 ans un moteur à compression variable et ont, pour ce faire, un grand besoin d'outils de simulation numérique. Toutefois, les logiciels de simulation actuels ont de très grandes difficultés à simuler les collisions et les contacts au sein de certains mécanismes, notament les engrenages. Or justement, leur moteur comporte plus d'engrenages qu'un moteur classique. Il en résulte qu'ils doivent passer du temps à préparer les modèles de pièce en vu des simulations voire à simplifier certaines interaction.

Par ailleurs, SOFA offre une nouvelle méthode de détection de collision[1] qui a l'avantage de pouvoir être calculée rapidement et à une précision arbitraire, sans que l'on ait à préparer les modèles. De plus, SOFA dispose, comme réponse aux collisions ainsi détectée, d'une résolution des contacts par contraintes[2] avec frottements. C'est pour MCE-5 Development l'opportunité d'obtenir de nouveaux outils de simulation qui pourront les aider à perfectionner leur moteur.

[1] http://www.sofa-framework.org/
[2] http://www.mce-5.com/

1.2 Résultats attendus

MCE-5 veut pouvoir faire différentes simulations avec SOFA pour en obtenir les données voulues, notament les positions des pièces et les forces qu'elles subissent. Ils fournissent les modèles des pièces, la vitesse du vilebrequin et la force appliquée au piston. L'objectif à terme a été de permettre non seulement la simulation du moteur dans son ensemble, mais aussi la simulation de certaines parties du moteurs seulement. Bien que moins ambitieuses, ces dernières sont d'ores et déjà susceptible de fournir des informations utiles tout en permettant de mettre au point les échanges de données entre MCE-5 et SOFA.

Pour toutes ces simulations, MCE-5 voulait pouvoir récupérer les trajectoires des pièces et les forces de contact. Enfin, ils voulaient pouvoir changer le plus facilement possible les modèles de pièces afin de pouvoir constater les conséquences de la présence de défauts sur le déroulement de la simulation. Un autre objectif était de pouvoir faire ces simulation autant avec des objet rigides qu'avec des objets déformables.

1.3 Problématique

Comme cela a été dit précédemment, SOFA possède un outil de détection de collisions très performant fonctionnant sur GPU[1] : il s'agit du CudaRasterizer. Toutefois, celui-ci a été fait pour répondre à la problèmatique première de SOFA qui est de faire des simulations réalistes intéractives, et non des simulations physiques très précises. Les informations temporaires sont générées et interprétées sur la carte graphique pour limiter le transit d'information entre le GPU et le CPU[1]. Cependant, lorsque l'on a essayé de simuler le moteur d'MCE-5 Development, il est tout de suite apparu que la mémoire disponible sur la carte graphique était limitante. On ne pouvait pas être suffisament précis pour simuler la partie fine du moteur.

L'objectif premier du stage était donc de développer un nouveau composant SOFA qui permette de réaliser la détection de collision en utilisant moins de mémoire tout en permettant de changer la finesse de la détection selon les contacts, et ce quitte à perdre en vitesse d'exécution.

Il fallait ensuite créer des composants et des scènes SOFA permettant à MCE-5 Development de faire leurs simulations sans qu'ils aient à apprendre le fonctionnement de SOFA. Il fallait simplifier le changements des paramètres de la simulation et offrir aux informations recueillies un format facilitant leur traitement.

1.4 Présentation de SOFA

SOFA est un *framework* libre (publié sous licence LGPL[3]) écrit en C++, conçu et développé pour permettre de créer des simulations réalistes et intéractives, donc en temps réel. SOFA est développé depuis 2005, le premier objectif étant de permettre de faire des simulations médicales. À cette fin SOFA offre un cadre cohérent permettant de développer et de mettre en relation divers outils de simulation numérique, entre eux, mais aussi avec différents moteurs de rendu visuel. Voir [3], [4] et [5] pour de plus amples détails.

Modèles

De manière générale, la simulation numérique demande de résoudre de nombreux problèmes. SOFA commence par les séparer en trois grandes catégories : les problèmes de détection de collision, les problèmes de la simulation du comportement intrinsèque d'un objet et de ses réactions à des forces extérieures, et enfin le rendu graphique de la simulation effectuée.

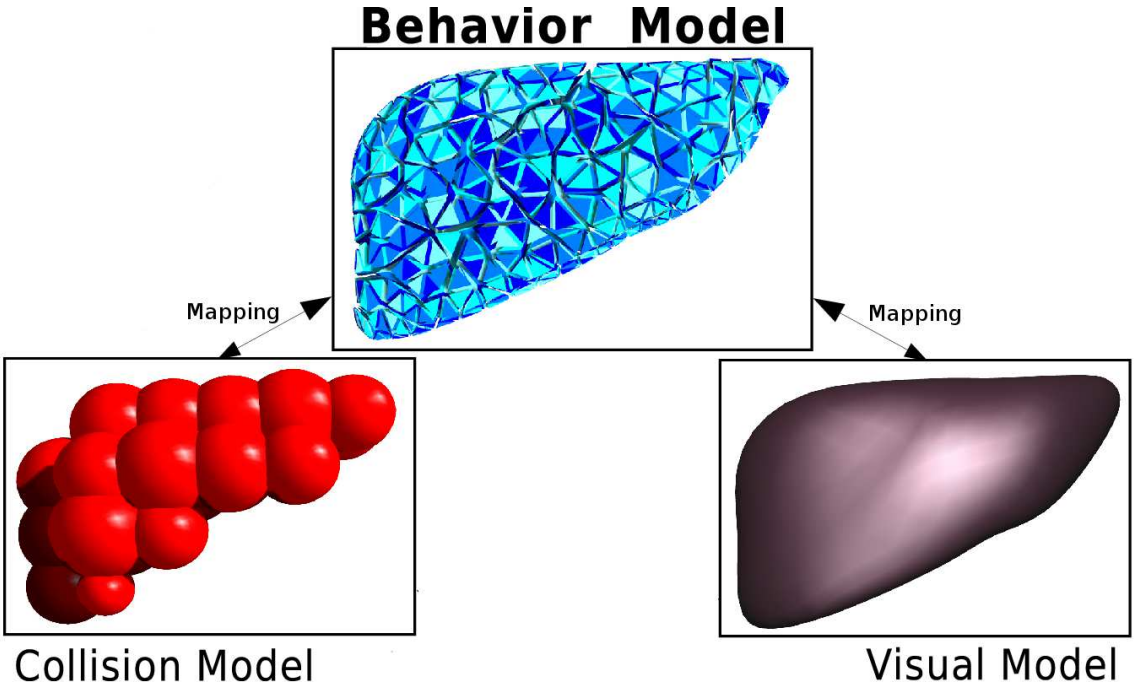


FIG. 1 – Exemple de séparation d'un objet (un foi) en plusieurs modèles.

Pour séparer ces problèmes, on sépare donc chaque un objet en plusieurs modèles. Tout d'abord un modèle maître (le « *behavior model* »), puis des modèles esclaves. Le lien qui relie chaque modèle esclave à son modèle maître s'appelle un *Mapping*, son rôle est de coordonner le modèle esclave avec son modèle maître.

[3] `http://www.gnu.org/licenses/lgpl.html`

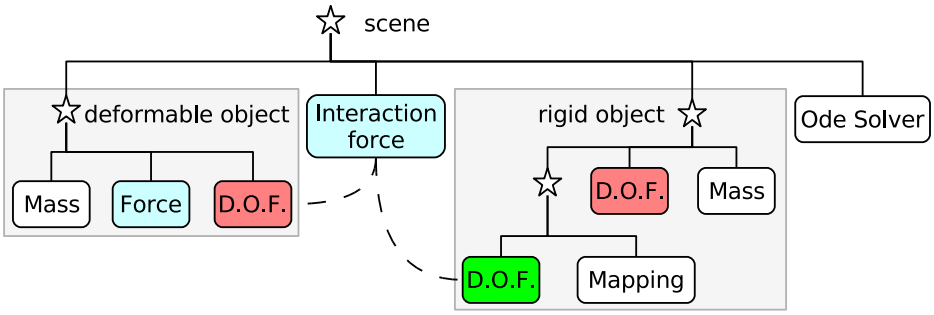
Behavior Model **:** C'est le modèle maître qui régit le comportement intrinsèque de l'objet. Est-ce un objet rigide ou déformable ? À quels forces ou contraintes est-il soumis ?

Collision Model **:** Quel modèle de collision est utilisé ? Cela peut-être le même maillage que pour le *Behavior Model*, mais s'il s'agit d'un objet rigide, ce dernier n'est représenté que par une seule particule, d'où le besoin d'un *Mapping*. Il peut enfin être une version simplifiée du maillage original, de sphères « accrochées » au behavior model, etc...

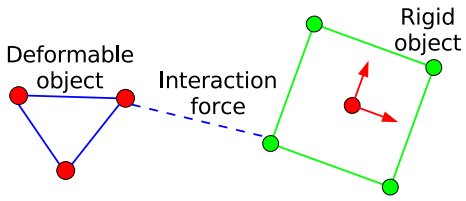
Visual Model **:** Permet d'afficher par exemple un maillage plus détaillé que celui utilisé pour la simulation, d'utiliser des *shaders*, etc...

Graphes de scène

Une simulation SOFA est représentée par un graphe de scène (exemple : Fig. 2). Ce graphe, qui peut être entièrement écrit en XML[3], est constitué d'une hiérarchie de nœuds, chaque nœud contenant une liste ordonnée de composants. Généralement, un objet est décrit par un nœud contenant les composants formant le *Behavior Model* ainsi qu'un sous-nœud pour chaque modèle esclave ; les modèles esclaves étant eux-mêmes constitués de différents composants. On peut alors aisément changer ces composants ou leurs paramètres pour immédiatement relancer la simulation et constater les conséquences qu'ont les modifications effectuées sur les résultats obtenus.



SOFA permet ainsi de séparer à nouveaux les problèmes :
- résolution de systèmes d'EDO,
- lecture de fichiers de maillage,
- le types de déformation des objets,
- les types d'intéraction entre les objets,
- la méthode de détection de collision,
- la réponse à la détection de collision...



On peut ainsi développer séparément des outils performants pour chaque tâche afin qu'ils soient ensuite « assemblés » à l'exécution.

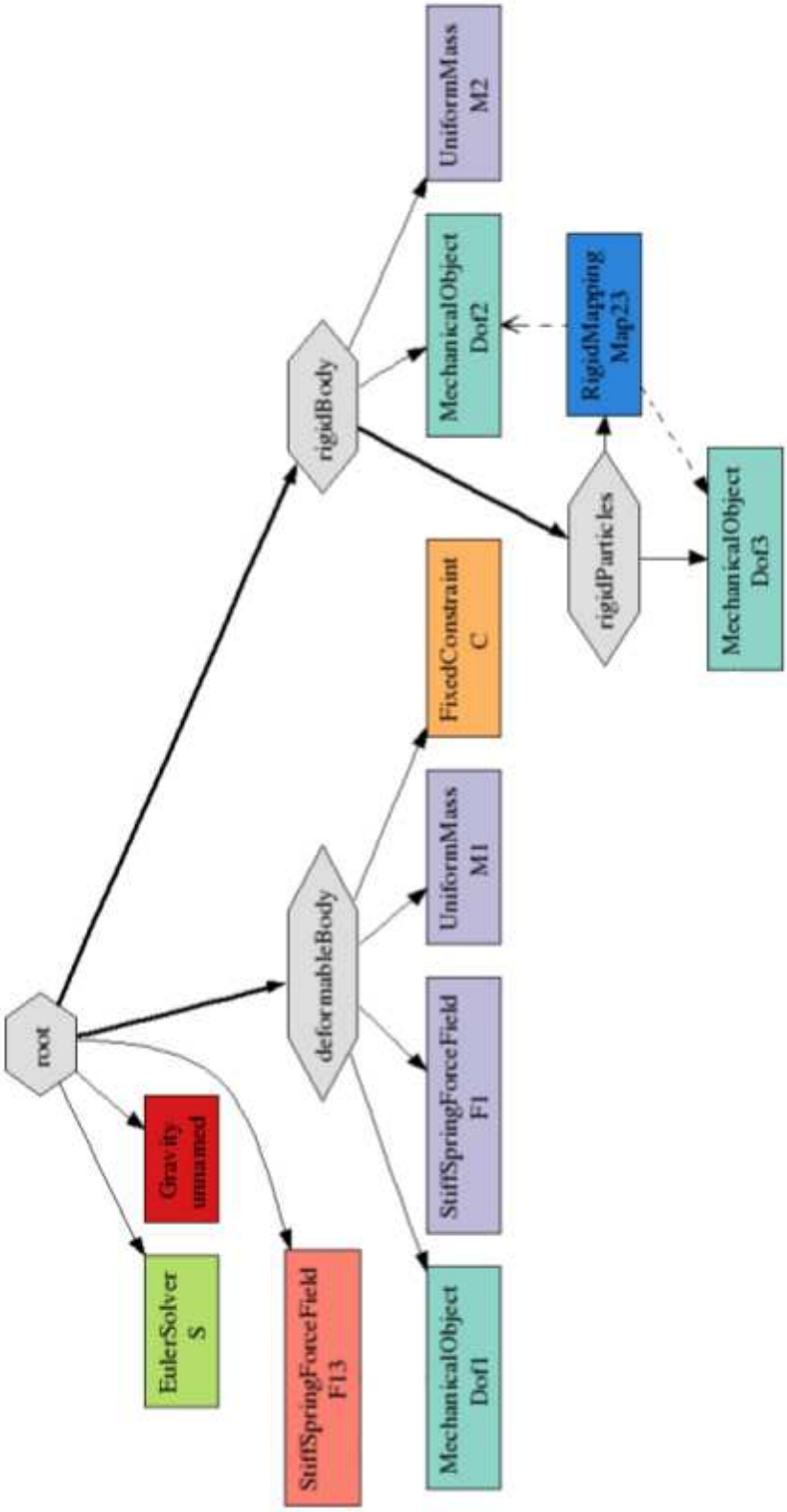


FIG. 2 – Exemple de graphe de scène complet. Les flèches pleines représentent les liens d'appartenance des composants. Les flèches en tirets représentent quant à elles les liens créés par les *Mappings*.

2 PairwiseCudaRasterizer

2.1 CudaRasterizer

Le *CudaRasterizer* est une méthode de détection de collision implémentée en CUDA. Celle-ci se fait en deux grandes étapes (voir Fig. 3) :

1. La première consiste à *rastériser*[4] les maillages des objets dans trois directions orthogonales : celles des axes du repère. Dans chaque direction, et pour chaque pixel de la *rastérisation*, on retient à quelles profondeurs l'on est entré et sorti de chacun des objets considérés.
2. Ensuite, à partir de ces données brutes, des *Layered Depth Images*[6] (LDI), on calcule les volumes de collision mais aussi, et surtout, leurs gradients en les vertex des maillages[1].

2.2 Problématique

CUDA[6] est une API développée par NVidia permettant de créer des programmes pour qu'il soient executés directement sur une carte graphique NVidia, et ce en ayant plus de libertés qu'avec les *Pixel Shaders* et les *Vertex Shaders*.

Cela permet d'exploiter la puissance des cartes graphiques non seulement pour la rastérisation en elle-même mais aussi pour calculer les volumes et leurs gradients (l'étape 2) directement sur GPU. On évite ainsi d'avoir à transferer les LDI du GPU au CPU, ce qui constituait le point faible d'une précédente mise en œuvre de cet algorithme.

Cependant, toutes les informations intermédiaires étant interprétées sur la carte graphique, elles doivent également être stockées sur cette même carte graphique. La conséquence est que, lorsque l'on essaye de trop augmenter la précision de détection en diminuant la taille des pixels lors de la rastérisation, on se retrouve limité par la taille de la mémoire.

Dans le cas du moteur MCE-5 VCR, on a besoin d'être très précis au niveau du rouleau de synchonisation, mais une telle précision, demandée sur la totalité du moteur est impossible avec le *CudaRasterizer*. C'est pour répondre à ce problème qu'a été créé ce nouveau composant, le *PairwiseCudaRasterizer*, pour permettre de faire des simulations qui ne pourraient pas être faites avec le composant d'origine.

[4]De manière générale, on parle de *rastérisation* lorsque l'on transforme un objet continu en un objet discret. Par exemple, on parle de *rastérisation* lorsque l'on transforme une image vectorielle (comme une image SVG[5]) en une image bitmap.

[5]`http://www.w3.org/Graphics/SVG/`

[6]`http://www.nvidia.com/object/what_is_cuda_new.html`

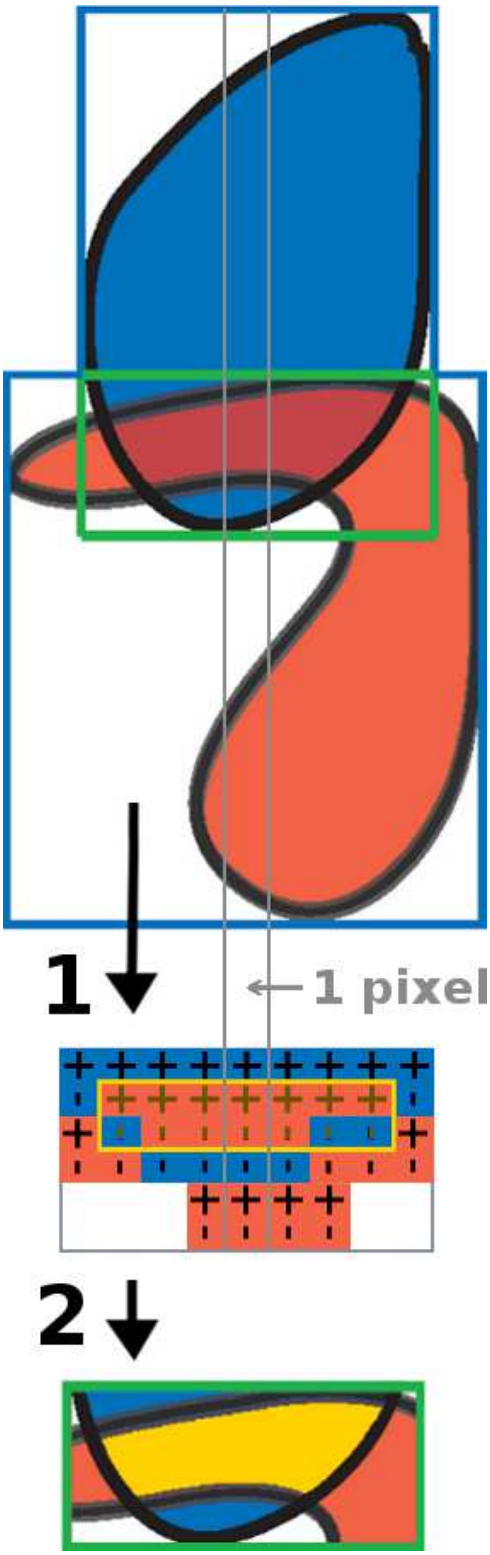


FIG. 3 – Illustration du fonctionnement du *CudaRasterizer* selon une direction : le maillage (en haut) est rastérisé de haut en bas pour produire les LDI (au milieu). Celles-ci sont ensuite interprétées pour calculer le volume et son gradient en les vertex des maillages.

2.3 Principe de fonctionnement

Le *CudaRasterizer* n'effectue la coûteuse rastérisation dans les intersections des boîtes englobantes des objets, mais pour tous les objets en même temps.

L'idée est de mener à bien séparément, par paire d'objets, l'ensemble de la détection de collision : la rastérisation et l'interprétation des LDI. On ne doit alors conserver les informations intermédiaires, les LDI, que pour une seule de ces paires à la fois.

On n'est donc plus limité par la quantité totale de mémoire que l'on aurait eu à utiliser avec le *CudaRasterizer*, mais seulement par le contact qui nécessite le plus de mémoire.

On peut alors utiliser, pour un seul contact, autant de mémoire que ce qu'on en utilisait pour l'ensemble de la scène. On peut donc également augmenter la précision de la détection pour une paire d'objets si l'on sait à l'avance que leur zone de contact va être limitée.

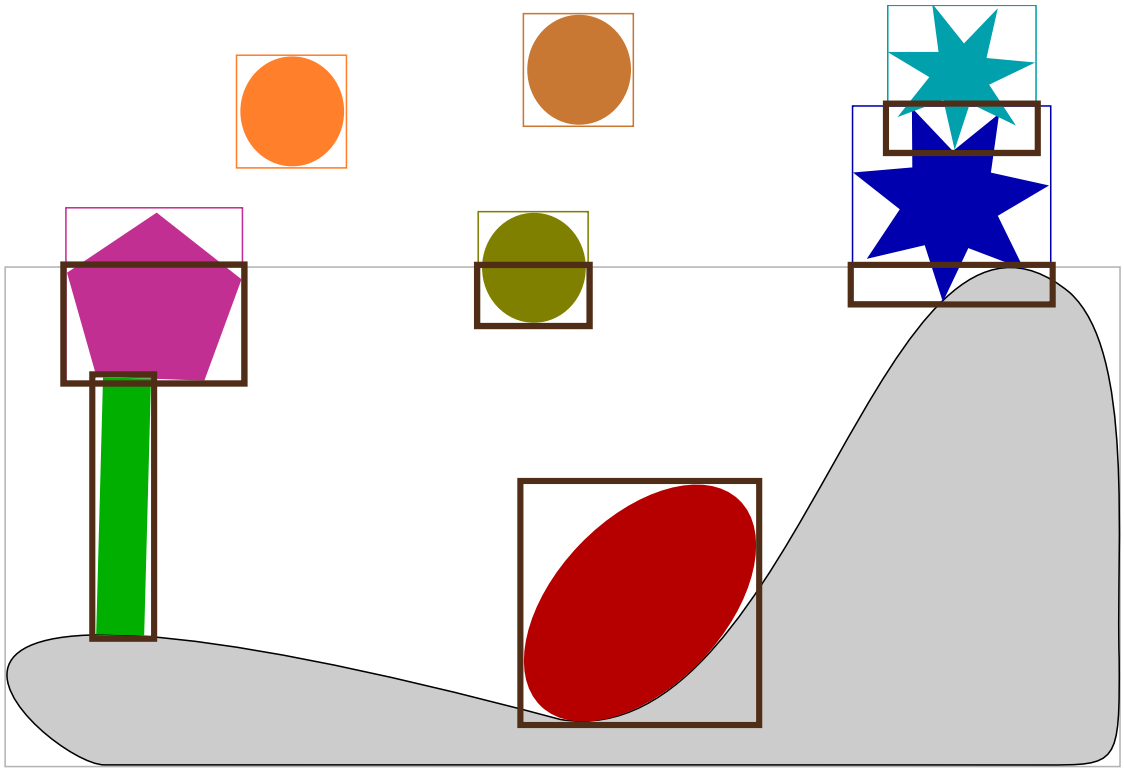


FIG. 4 – Schématisation d'une simulation avec la représentation des intersections de boîtes englobantes à *rastériser* (en marron).

2.4 Mise en œuvre

Dans la pratique, il faut déjà déterminer quelles sont les détections à faire :
- Quelles paires d'objets ?
- Avec quelle précision ?
- Dans quelle zone de l'espace ?

On répond à ces question en fonction des paramètres du composant qui sont décris dans la section suivante. Mais, en supposant que l'on a ces informations, comment faire pour « découper » le processus de détection de collisions ? Cela s'est fait progressivement, par étapes, sur une durée totale d'environ 2 mois pour en arriver à une version pleinnement fonctionnelle du composant.

Il a fallut tout d'abord mettre en œuvre le principe de base dans un *MasterSolver*, ce qui permettait de commencer de développement sans avoir encore appris tout ce qu'il y aurait à apprendre sur le fonctionnement de SOFA. Le code ainsi produit a ensuite été utilisé pour créer un composant héritant du *CudaRasterizer*. Enfin il a fallu conformer complétement ce dernier à l'API de SOFA puis à celle du *CudaRasterizer*.

MasterSolver

La première étape a été de réaliser un composant SOFA de type *MasterSolver*. Les *MasterSolvers* court-circuitent le fonctionnement du pipeline de collision. Il peut s'agir de ne faire la détection de collision qu'un pas de temps sur deux, de mettre en place un pas de temps variable, ou encore d'inverser l'ordre dans lequel sont effectuées la détection de collision et l'intégration du pas de temps.

Dans le cas présent, on commençait par récupérer la liste des objets en collision de la même manière que le *CudaRasterizer*, pour en calculer les boîtes englobantes. On calculait ensuite les intersections de boîtes englobantes puis, pour chaque intersection, lançait la détection de collision et la création de la réponse.

CudaRasterizer	
Détection	1, 2 et 3
Réponse	1, 2 et 3

MasterSolver	
Détection	1
Réponse	1
Détection	2
Réponse	2
Détection	3
Réponse	3

FIG. 5 – Le *CudaRasterizer* effectue d'une traite la détection d'une part puis la réponse d'autre part. Le *MasterSolver* entremêle complètement ces deux étapes.

Avant chaque couple détection-réponse, on reconfigurait les paramètres déterminant la finesse de la détection puis on réinitialisait le *CudaRasterizer* de sorte qu'il ne prenne en compte que ces deux objets de la scène. On répondait donc à la volonté de pouvoir changer la résolution selon les paires d'objets, tout en limitant la consomation de mémoire à celle nécessaire pour le contact en nécessitant le plus.

Héritage du CudaRasterizer

Cette seconde étape a été bien plus simple. Il s'agissait seulement de créer un nouveau composant SOFA qui hérite cette fois-ci, au sens de la Programation Orientée Objet, du *CudaRasterizer*. Le changement est que, plutôt que d'avoir un *MasterSolver* configurant et reconfigurant un seul autre composant, on n'a qu'un seul composant se reconfigurant lui-même.

Ce composant, comme le précédent, présentait un bug qui apparaissait plus où moins systématiquement mais seulement après plusieurs centaines de pas de temps, provoquant alors le plantage complet du programme. Ce bug n'apparaissait pas dans des scènes simples, en fait, il n'a été observé que dans la scène du moteur complet. Il a fini par s'avérer que ce bug était dû au fait que la réponse à la collision était créée en plusieurs fois, chose incompatible avec le fonctionnement interne d'un des éléments du pipeline de collision.

Ce bug, pour le moins génant n'a put finalement être débusqué qu'après le lancement de la simulation incriminée avec valgrind. Cela prit une bonne journée mais les information ainsi obtenue permirent de mettre en place une modification temporaire du *CudaRasterizer* évitant ce bug en attendant de séparer détection et réponse.

Séparation de la détection et de la réponse

Lorsqu'il est utilisé, le *CudaRasterizer* fait partie intégrante du pipeline de collision. Il a de fait une interface précise qu'il doit respecter pour pouvoir être utilisé correctement. Mon composant, alors, ne respectait absolument pas cette interface parcequ'il mélait détection et réponse.

Pour séparer les deux, il fallait sauvegarder les informations finales de chaque détection. Toutefois trouver et comprendre quelles informations sont utilisées pour la création de la réponse n'est pas vraiment évident. Le fait est que le *CudaRasterizer* souffre d'un certain manque de documentation et que, même au sein du code, il est difficile au premier abord de comprendre quelles sont les structures de données clefs et leurs rôles. La coopération des personnes ayant mis en œuvre le *CudaRasterizer* a de fait été vraiment très précieuse.

CudaRasterizer	
Détection	1, 2 et 3
Réponse	1, 2 et 3

PairwiseCudaRasterizer 1ère version	
Détection	1
Réponse	1
Détection	2
Réponse	2
Détection	3
Réponse	3

PairwiseCudaRasterizer 2ème version	
Détection	1
	2
	3
Réponse	1
	2
	3

FIG. 6 – La deuxième version du *PairwiseCudaRasterizer* sépare la détection et la réponse, mais ni l'une ni l'autre n'est effectuée d'une seule traite.

À cela s'est ajouté une autre difficulté. Sauvegarder les informations à chaque passe, et utiliser à la volée ces informations pour la création de la réponse ne posait aucun problème. Mais malgré cela, lorsque l'on séparait la détection et la réponse, les simulations ne fonctionnaient plus du tout correctement, sans que valgrind n'offre cette fois-ci la moindre piste. Différents essais ont indiqué que ce nouveau problème avait la même origine que le premier.

Le fait que la réponse soit créée en plusieurs fois provoquait la corruption de la mémoire en des endroits à présent critiques, là où ils n'avait pas d'incidence visible jusqu'alors. La précédente modification du *CudaRasterizer* ne faisait en fait que cacher le problème sans le résoudre. Il fallait que la réponse ne soit créé qu'en une seule fois pour qu'il n'y ait plus de comportement indéterminé et, à moins de mutiler ou réécrire un des composants du pipeline de collision, il n'y avait qu'une solution.

Recollement des informations de détection

Le *CudaRasterizer* est avant tout un outil de détection de collisions. À ces collisions, plusieurs réponses sont possibles : cela va de la création d'un champs de force à la destuction de l'un des deux objets. Mon composant ne devant être qu'une modification de la détection, l'idéal (et finalement la nécessité) était de recoller les blocs d'information des différentes passes pour ne former qu'une seule structure de données cohérente, compréhensible par tous les types de réponses possibles.

Il fallait, pour cela, non seulement savoir exactement quelles structures de données étaient utilisées, mais aussi connaître précisément leur signification. Là, encore, le manque de documentation a été une difficulté. Il a fallu chercher la réponse dans le *CudaRasterizer*, et dans les composants créant les types de réponses.

On ne peut en effet se contenter de mettre les informations produites bout à bout. Chaque passe du *CudaRasterizer* doit être modifiée pour que toutes puissent être cohérentes entre elles et avec la liste complète des objets. Au final, les résulats

CudaRasterizer	
Détection	1, 2 et 3
Réponse	1, 2 et 3

PairwiseCudaRasterizer 2ème version	
Détection	1
	2
	3
Réponse	1
	2
	3

PairwiseCudaRasterizer 3ème version	
Détection	1
	2
	3
Réponse	1, 2 et 3

FIG. 7 – La troisième version du *PairwiseCudaRasterizer*, en plus de séparer la détection et la réponse, et effectue la réponse d'une seule traite.

obtenus ont été comparés à ceux obtenus avec le *CudaRasterizer* sur des scènes complexe et sur plusieurs pas de temps pour s'assurer de la validité du composant.

2.5 Paramètres

Il y a, par rapport au *CudaRasterizer*, trois nouveaux paramètres qui permettent de configurer ce composant. Tous nécessitent de pouvoir identifier les objets de la simulation.

Le *CudaRasterizer* utilise les *tags* pour identifier les objets qui vont être impliqués dans la détection de collision : il n'effectue la détection de collision qu'avec les objets dont le composant *topology* possède les ses *tags*. C'est donc tout naturellement que ce même moyen a été utilisé pour le *PairwiseCudaRasterizer* qui recherchera certains autres *tags* dans les mêmes composants.

Un *tag* est une « étiquette », une chaîne de caractères que l'on associe à un ou plusieurs objets. C'est un moyen de les regrouper virtuellement, où bien de les distinguer les uns des autres. Il existe un *tag* particulier, le *tag* nul correspondant à la chaîne de caractères « 0 ».

Objets Pairs

Ce nouveau paramètre permet de sélectionner à l'avance pour quelles paires d'objets il va y avoir détection de collision. En effet, dans le cas de la simulation d'un mécanisme dont on connait le fonctionnement, il y a de nombreuses paires d'objets dont on sait qu'ils ne sont pas suceptibles d'entrer en contact. On économise autant de temps d'exécution que l'on évite le long processus de rastérisation pour chacune de ces paires, même si leurs boîtes englobantes s'intersectent.

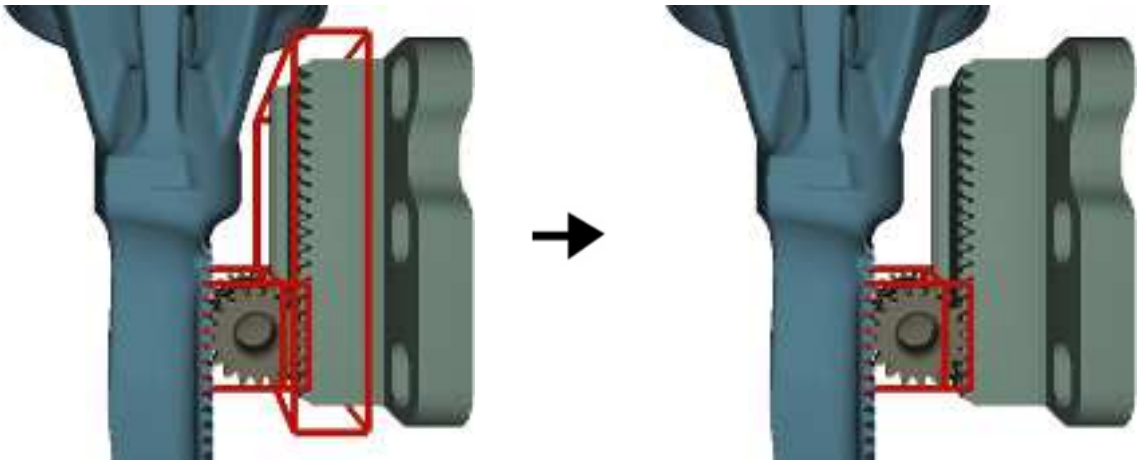


FIG. 8 – Dans cet exemple, on aura donné en paramètre : « Piston Rouleau Rouleau platine », évitant ainsi qu'une rastérisation inutile soit faite entre le piston et la platine de synchronisation.

Ce paramètre est une liste de paires de *tags*. Un *tag* signifie ici « tous les objets possédant ce *tag* », le *tag* nul signifiant alors « tous les objets ».

On ne cherche de collision qu'entre les paires d'objets correspondant à au moins l'une de ces paires. On peut donc aussi, à l'aide du *tag* nul, spécifier la recherche de collision entre les objets possédant un certain *tag* et tous les autres. Enfin, si la liste donnée en paramètre est vide, on fait la détection de collision entre tous les objets.

Resolution Rules

Ce paramètre a été le premier à être mis en place, il s'agit d'une liste ordonnée de paires de *tags* auxquelles on associe des informations de configuration du *CudaRasterizer*. Les paires d'objets sont identifiées par ces paires de *tags* de la même manière que pour le précédent paramètre. Pour un contact donné, le *CudaRasterizer* est configuré selon la première règle correspondant à la paire d'objets impliquée et, si aucune règle n'est trouvée, on utilise la configuration d'origine du composant.

Les paramètres reconfigurés sont la taille de pixel utilisée pour la rastérisation et la taille des sous-volumes (*cells* [2]). Les sous-volumes correspondent à une grille régulière définie par la taille de pixel et un paramètre *resolution* qui n'est autre que la taille des sous-volumes en terme de pixels.

On peut donc changer, selon les paires d'objets considérées, la finesse avec laquelle est effectuée la détection de collisions et, si le type de réponse le permet, la finesse avec laquelle est effectuée la réponse à la détection de collision.

Bounding Trees Depth

Ce paramètre est le dernier à avoir été mis en place. Avant cela, toutes les boîtes englobantes des objets étaient calculée pour élaguer la liste de paires d'objets à rastériser, alors même que cela pouvait être rendu inutile par l'ajout du paramètre « *Objets Pairs* ». Mais le plus gros avantage vient du fait que l'on peut maintenant calculer des arbres de boîtes englobantes au lieu de simples boîtes englobantes.

Il se présente sous la forme d'une liste ordonnée de *tags* associés à un entier positif. Pour chaque objet, on calcule l'arbre de boîtes englobantes avec la profondeur associée au premier *tag* de la liste qu'il possède. Une profondeur de 0 indique le calcul d'une simple boîte englobante.

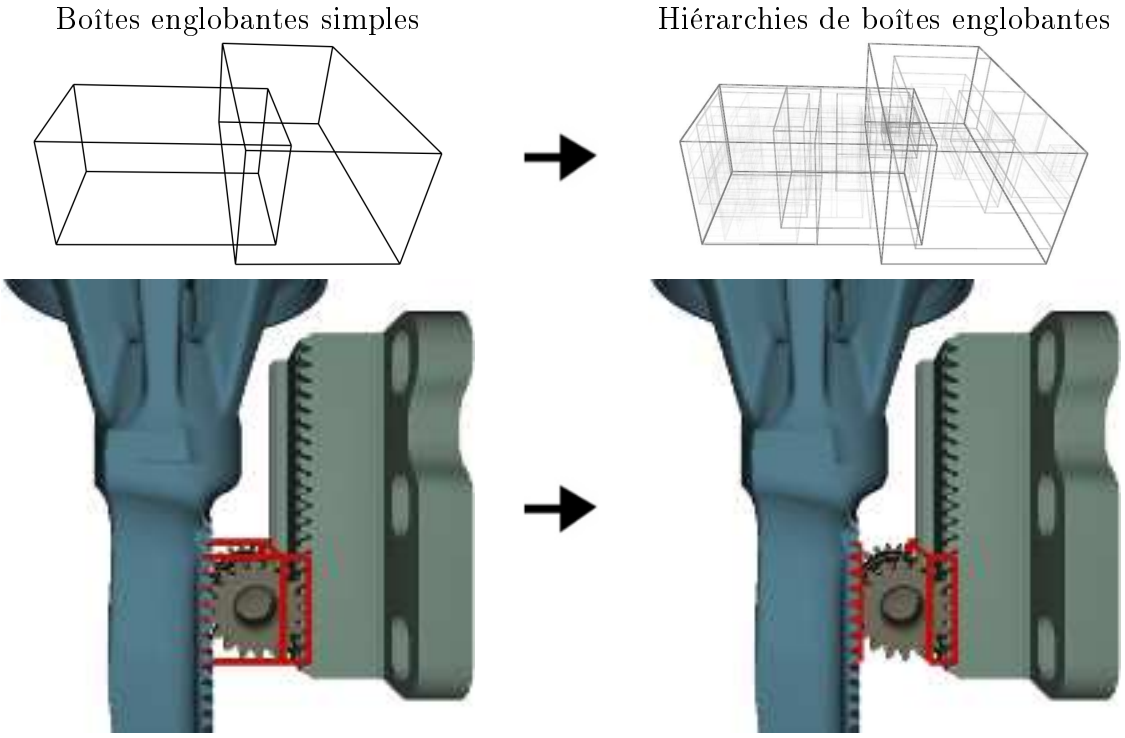


FIG. 9 – Dans cet exemple, on aura donné en paramètre : « Piston 3 Rouleau 0 », permettant ainsi de limiter la zone de rastérisation entre le Piston et le Rouleau. Au final, on a ici le même temps d'exécution mais un gain de mémoire GPU.

Cela permet, comme avant la mise en place de ce paramètre, d'élaguer la liste des contacts en fonction de l'intersection des boîtes englobantes des objets impliqués. Toutefois, et c'est ce pourquoi ce paramètre à été mis en place, calculer un arbre de boîtes englobantes pour certains objets permet de faire la détection de collisions, et donc le long processus de rastérisation (coûteux en temps de calcul et en mémoire), dans des volumes plus petits qu'avec les boîtes englobantes simples.

2.6 Limitations et développements futurs

Sur-utilisation du BUS

Comme cela a été évoqué, ce composant est, dans le cas général, plus lent que le composant original, cela principalement pour deux raisons. La première est une sur-utilisation du BUS de communication entre GPU et CPU.

Là où le *CudaRasterizer* ne fait transiter les modèles de collision qu'une fois sur le BUS, ce nouveau composant les fait transiter une fois par contact. De plus, les mêmes pré-calculs sont effectués à chaque fois que l'on envoie un même objet sur GPU pour qu'il y soit rastérisé.

Une solution envisageable, et qui serait intéressante, serait d'envoyer l'ensemble des objets sur la carte graphique pour y effectuer les pré-calculs, puis de conserver les informations obtenues. Il ne resterait alors plus qu'à copier ces informations dans les structures de données adéquates du *CudaRasterizer* pour pouvoir éviter d'avoir à déclencher à chaque fois ces transits d'information entre CPU et GPU.

Cela demanderait cependant une bonne compréhension de ces structures de données et de leur utilité précise, comme il a fallu le faire pour les données de sortie du *CudaRasterizer*.

Redondance de certains calculs

Une autre cause de ralentissement est une certaine redondance de la rastérisation : si l'on a plusieurs collisions potentielles en une même zone de l'espace, alors les objets qui s'y trouvent seront inmanquablement rastérisés plusieurs fois. Il y a là aussi une perte de temps mais c'est ainsi que l'on s'assure d'utiliser le moins de mémoire possible. Cette façon de faire a été également choisie car c'est par paire d'objets que se présentent les informations en sortie du *CudaRasterizer*

Il pourrait être intéressant de pouvoir effectuer la détection de collision par groupes d'objets plutôt que par paires, mais ça n'est vraiment pas la direction prise lors du développement de ce composant. La transition n'est pas aussi évidente à faire puisque les paramètres de configuration et les structures de données internes au composant ne sont pas adaptées à cela.

Il faut aussi garantir l'unicité de la détection concernant une paire d'objets, tout en étant sûr que la détection est bien faite pour chaque paire d'objets. Cela est moins évident à mettre en place en raisonnant par groupes. Ces problèmes, qui pourraient alors incomber à l'utilisateur, sont évités par la solution choisie.

Auto-collisions et gestion des fluides

Une grosse limitation de ce nouveau composant était qu'il ne gérait pas les auto-collisions, les topologies dynamiques, ni les fluides. Tout au long du développement du composant, cette problématique avait été laissée de côté car y répondre n'était pas nécessaire aux simulations demandées par MCE-5.

Cependant, une fois le recolement des informations mis en place, une modification simple permettait de s'attaquer à ce problème. En effet, le fait de faire des détections par paires d'objets uniquement est ici d'une grande aide. À chaque fois que l'on fait une détection, on ne peut se retrouver qu'avec deux cas possibles : soit on fait une détection entre un objet et lui-même, soit entre de objets distincts. Dans le premier cas, les informations obtenues ne peuvent concerner que des auto-collisions, elles sont donc conservées. Dans le deuxième cas, il suffit de supprimer les informations d'auto-collision pour ne conserver que les informtions concernant les deux objets considérés.

Par ailleurs, une fois le recollement des informations mis en œuvre, le fonctionnement des structures étant compris, cette nouvelle précaution n'a pris que quelques heures à être mise en place.

Toutefois, s'il n'a fallu que peu de temps pour la mettre en place, il aurait fallu plus de temps pour la mettre à l'épreuve. Ayant été décidé précédemment que ces fonctionnalités n'étaient pas une priorité au regard des engagements pris envers MCE-5, seuls quelques tests ont étés effectués. Il en ressort que le fonctionnement concernant les auto-collisions semble correct puisque des résultats obtenus semblent identiques à ceux du *CudaRastizer*, mais ils faudrait le tester rigoureusement sur des scènes complexes pour pouvoir s'en assurer. Enfin, le composant ne fonctionne absolument pas tel quel avec les fluides.

3 Entrées / Sorties

Cette section présente les composants SOFA qui ont été développés exprès pour mettre en place les simulations demandés par MCE-5. La plupart répondent à des besoins très précis et seul GenerateRigidMass a été déjàintégré à SOFA.

3.1 CSV Loader

Ce composant SOFA permet à MCE-5 de paramétrer la vitesse du vilbrequin et les différentes forces à appliquer au sein du moteur lors des simulations, sur le piston par exemple. C'est un *loader* : il lit un fichier à la création de la scène pour que d'autres composants SOFA puissent utiliser les données extraites pendant la simulation.

Les différentes informations (fonction du temps) sont placées dans différentes colonnes, selon le format CSV. Les fichiers de ce format de fichier sont encodés en ASCII, ce qui les rend faciles à *parser*. Les informations doivent être stockées dans l'ordre pour être correctement lues : que ce soit l'ordre chronologique où l'ordre des colonnes. L'ordre attendu est le suivant :

1. angle du vilbrequin (degrés),
2. temps (s),
3. la moitié de la position verticale du piston (mm),
4. force à appliquée sur le piston (N),
5. force appliquée sur le rouleau de synchronisation (N),

3.2 GenerateRigidMass

Problématique initiale

Face à l'objectif de faciliter le changement des modèles des pièces, les premiers problèmes à résoudre furent les suivants :

- Un premier problème inhérent aux objets rigides est qu'il faut calculer leurs matrices d'inertie. Par ailleurs, et que les objets soient rigides ou non, il faut que leur masse soit correcte.

- Enfin, dans SOFA, les objets rigides ont, lorsqu'ils sont créés, leur centre de gravité placé arbitrairement à l'origine de l'espace. Ceci a été fait ainsi pour faciliter certains calculs, mais impose de fait que les maillages utilisés soient centrés sur leur centre de gravité si l'ont veut que le comportement soit correct.

Ces problèmes ont été dans un tout premier temps résolus « à la main ». Le logiciel Blender a été utilisé pour traduire les premiers modèles, au format VRML, qui ont été fournis par MCE-5. Les matrices d'inertie ont été calculées avec une petite application en ligne de commande annexe à SOFA : *generateRigid*, qui calcule aussi la masse et le centre de gravité. Ils ont ensuite (et enfin) été centrés à l'aide de *meshconv*, une autre application du même genre.

Tout cela était fastidieux et introduisait des erreurs d'arrondis puisque tous les calculs réalisés par *generateRigid* était effectués en simple précision et que l'interface graphique du *Modeler* limitait le nombre de chiffres significatifs.

Nouveau besoin

Par ailleurs, un membre de l'équipe Evasion a réalisé, dans le cadre du projet ROMMA[7], un composant SOFA permettant de charger des modèles STEP. Les fichiers STEP décrivent les objets par des surfaces analytiques plutôt que par un maillage, ce composant les triangule à l'aide d'OpenCascade[8], utilisé en tant que bibliothèque externe.

Toutefois, les applications utilisées précédemment pour modifier les maillages et calculer leurs propriétés étaient incapables d'utiliser ce nouveau composant pour lire des fichiers STEP. Il fallait donc automatiser ces calculs, qu'ils soient faits directement dans SOFA, pour pouvoir profiter de l'opportunité de permettre à MCE-5 d'utiliser directement leurs modèles STEP dans SOFA.

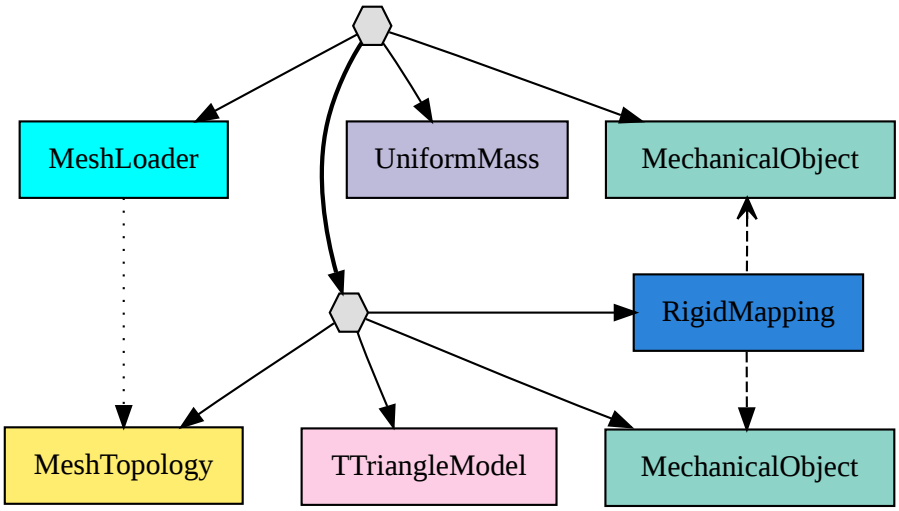


FIG. 10 – Graphe du nœud d'un objet rigide, sans GenerateRigidMass. La flèche pointillée représente un transfert de données.

[7] `http://www-ljk.imag.fr/MGMI/projets.html`
[8] `http://www.opencascade.org/`

Automatisation

Ainsi, le code de l'application *generateRigid* a été récupéré et adapté pour créer un nouvel *engine* : *GenerateRigidMass*. Des tests ont ensuite été effectués avec différents maillages pour vérifier la conformité des résultats obtenus avec ce nouveau composant à ceux obtenus avec l'application originale.

Dans SOFA, un *engine* est un composant permettant de transformer l'information : il prend des paramètres en entrée (dans notre cas, un maillage) et fait des calculs à partir de ces données pour en laisser le résultat à la disposition d'autres composants. Volumes, masses, centres de gravité et matrices d'inertie sont donc calculés automatiquement au début de la simulation et récupérés par le composant *UniformMass*.

Par ailleurs, le maillage est centré sur son centre de gravité par un autre *engine* : *transformPosition*. Le composant *MeshTopology* du modèle de collision récupère donc les positions du maillage dans ce composant-là, puis les indices des triangles dans le composant contenant le maillage chargé en mémoire : le *MeshLoader*.

Le centre de gravité est donc bien placé par rapport à l'objet. Il reste à translater ce dernier vers le centre de gravité du maillage original, ce qui est fait directement dans le *MechanicalObject*.

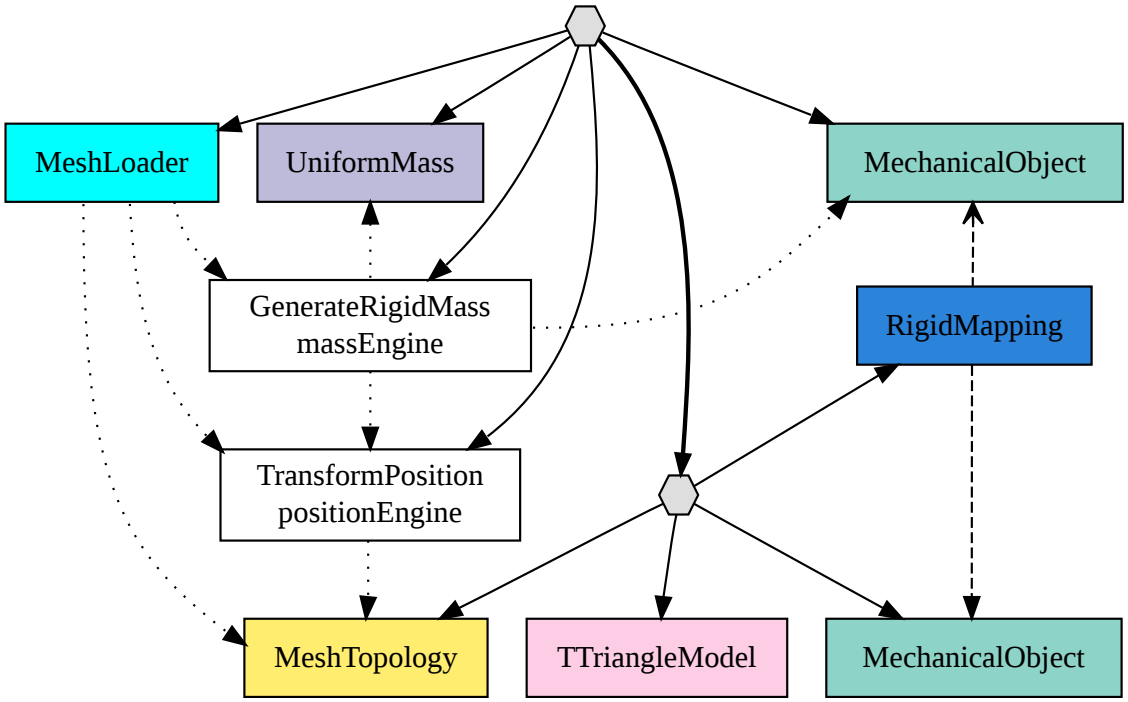


FIG. 11 – Graphe du nœud d'un objet rigide utilisant GenerateRigidMass

3.3 Loader Switch

Problématique

Il s'agit d'un problème arrivé de paire avec le composant permettant de lire les modèles STEP. En effet, parce que ces derniers sont constitués de surfaces analytiques, ils peuvent être très longs à charger : cela peut aller de quelques secondes à plusieurs minutes. On se retrouvait ainsi avec des durées d'initialisation des simulations très longs : jusqu'à 20 minutes pour la scène du moteur complet. Un tel temps de chargement rend bien trop compliqué l'ajustement des paramètres de simulation, il fallait donc trouver une solution.

Principe

Le principe est très simple : lorsque le fichier STEP est chargé pour la première fois, on sauvegarde la triangulation effectuée pour ne pas avoir à la calculer chaque fois que l'on relance la simulation. Il faut pour cela :
- vérifier automatiquement la présence d'une triangulation d'un fichier STEP.
- charger soit le fichier STEP, soit sa triangulation.
- n'avoir qu'un composant où aller récupérer le maillage obtenu.
- sauvegarder le maillage, uniquement si le fichier STEP venait d'être lu.

Mise en œuvre

Il y a tout d'abord un premier *engine* qui prend en paramètre le nom du fichier STEP dont on veut la triangulation. Il possède trois sorties, la première contient le nom du fichier STEP, la deuxième le nom du fichier contenant la triangulation, la troisième un booléen indiquant le choix effectué : ce composant vérifie si la triangulation associée existe ou non et selon le cas, vide l'une de ses deux premières sorties. Si la triangulation n'est pas faite, on lit le fichier STEP, sinon, on lit la triangulation.

Ensuite, un deuxième *engine* vérifie la valeur du booléen évoqué ci-dessus et récupère le maillage dans le bon *loader*. Ainsi, partout où l'on récupérait des informations dans le *loader*, on les récupère dans ce *engine*.

Enfin, on crée un nouveau nœud contenant un composant appelé *MeshTopology*, récupérant le maillage non centré et un composant *VTKExporter*. Ce dernier était le composant le plus simple à utiliser pour sauvegarder un maillage. Il permet de sauvegarder les maillages des objets tout au long de la simulation, ou à la fin de celle-ci. Il n'y a eu qu'à lui ajouter un paramètre booléen permettant de demander

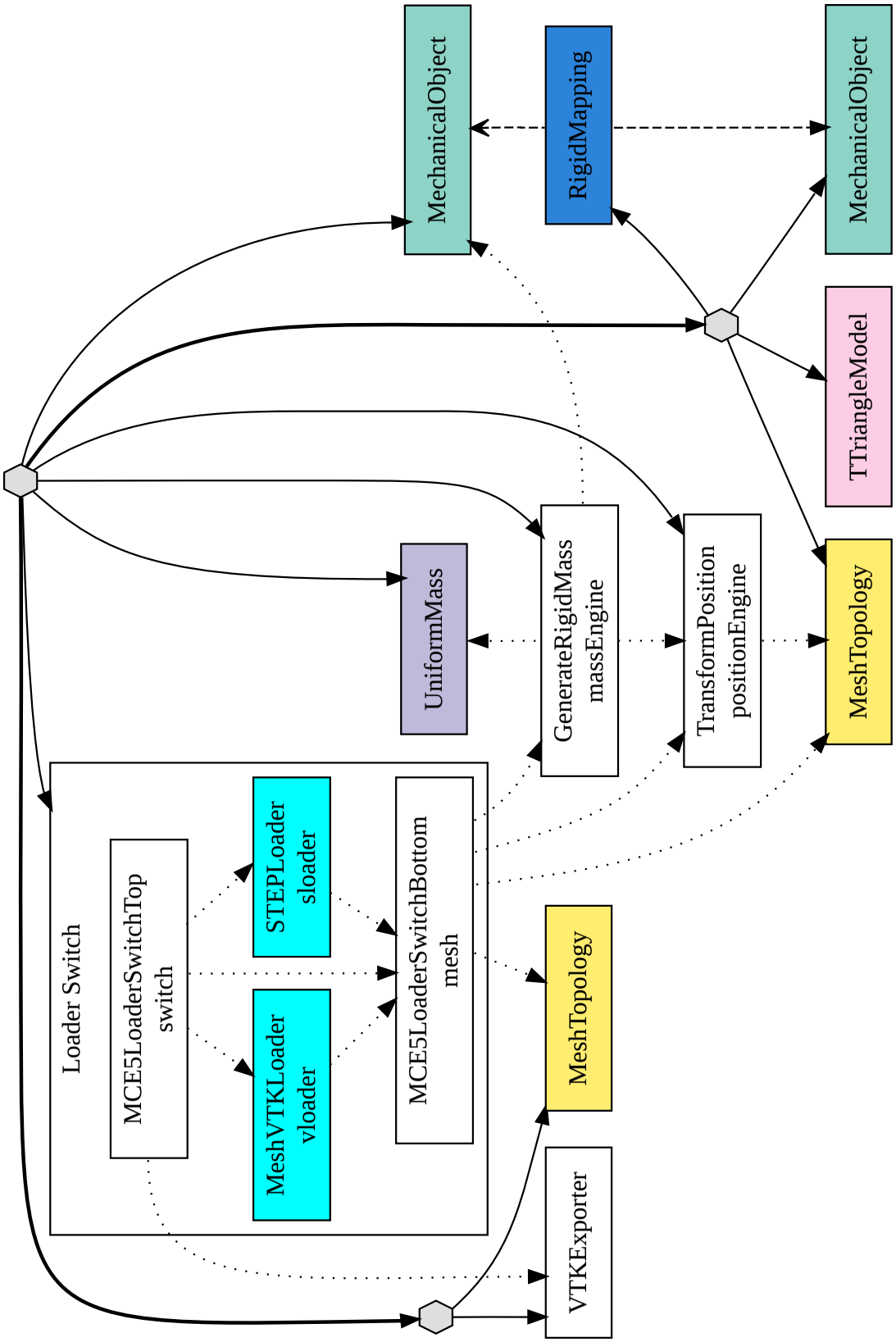


FIG. 12 – Graphe du nœud complet d'un objet rigide

à exporter après l'initialisation de la scène, paramètre qui vient récupérer sa valeur dans le premier composant décrit ci-dessus.

Au final on se retrouve à nouveau avec des temps d'initialisation qui se mesurent à nouveau en secondes, et non en minutes. Si l'on souhaite forcer la retriangulation d'un modèle STEP « *model.step* » alors il est nécessaire de supprimer le fichier « *model0.vtu* ».

3.4 LMConstraint Monitor

Ce composant a été créé afin d'exporter les forces de contact. Il récupère les forces calculées par les composants créés en réponse aux détections de collision, les *CudaLDIContactLMConstraint*, à l'aide d'une méthode créée à cet effet, puis les exporte dans différents fichiers.

Utilisation

Ce composant se place dans le nœud racine de la simulation. On lui donne en paramètre des couples de chaînes de caratères. Ne sont exportées que les forces concernant les couples de *MechanicalObject* dont les noms correspondent à ces couples de chaînes de caractères.

Les forces récupérées sont enregistrées dans plusieurs fichiers de sorte que l'on puisse récupérer séparément celles appliquées sur chaque objet par chaque objet. Le nom du fichier est en deux partie, la première correspond au nom de l'objet subissant les forces, la deuxième partie correspond aux deuxième objet impliqué. Est exportée également, à chaque pas de temps, la somme des forces appliquées à chaque objet, dans un fichier identique mais dont le nom se termine par « _sum ».

Enfin, le format utilisé est simpliste : les informations sont parsées par de simples tabulations. Cela permet de facilement les récupérer dans un programme externe ainsi que de les visualiser directement à l'aide de gnuplot[9].

Problèmes rencontrés

Un des problèmes a été que la réponse utilisée pour la détection de collision ne calculait pas explicitement des forces. En fait, les données récupérées à la fin de chaque pas de temps n'étaient pas homogènes à une forces vu la façon dont elles dépendaient du pas de temps ou de la précision de la détection.

[9] `http://www.gnuplot.info`

Une fois celui-ci détecté, une relecture attentionnée de [2] a permis de cerner le problème : malgrès un certain abus de language, les « forces » calculées ne sont pas des forces mais des pressions surfaciques. La difficulté restante était de trouver un moyen économique de calculer la force équivalente à la correction effectuée.

La publication [2] décrit un moyen de calculer cette force à partir des données disponibles au début du pas de temps, mais pas à partir des seules données disponibles une fois la correction effectuée. Elle décrit toutefois la façon dont est évaluée la surface de contact. Le problème qui est apparu alors, était que les « *contactForces* » récupérées à la fin de chaque pas de temps au sein des *CudaLDIContactLMConstraint* n'étaient pas non plus des pressions surfaciques, mais des pressions surfaciques multipliées par un temps.

Un nouveau membre a donc été ajouté pour stocker la force, dorénavant calculée, correspondant à la correction effectuée. Des tests ont ensuite été effectués sur des scènes simples. Ils confirment que les données finalement obtenues sont bel et bien des forces puisque l'on retrouve les résultats prévus, et ce « indépendement » du pas de temps ou de la précision de la détection et de la réponse.

3.5 Config

Ce composant a été créé pour deux raisons. La première est de permètre de régler un temps à partir duquel la simulation s'arrête automatiquement. La seconde est de pouvoir facilement et simplement changer les paramètres principaux de la simulation :
- densité des pièces
- coefficients de friction
- coefficients de poisson des objets déformables
- modules de Young pour les objets défomrables

Ce composant ne fait que stocker ces valeurs pour qu'elles soient ensuite récupérées par les composants concernés, à l'aide de « *Data link*s ».

Data Link

Ces liens sont un moyen souple permettant de transférer des données d'un composant SOFA à un autre. Ces liens ne peuvent aller que de haut en bas dans le graphe de scène. Le composant du bas, lorsqu'il en a besoin, copie la valeur du composant du haut après s'être assuré qu'elle soit à jour (comme cela peut ne pas être le cas si ce composant est un *engine*).

4 Problèmes Numériques

Cette section présente les principaux problèmes numérique rencontrés durant le stage, en-dehors du développement des composant présentés précédemment, durant la mise au point des simulations. Certains ont pu être corrigés, d'autres, inhérents aux outils informatiques utilisés, se sont avérés être des contraintes avec lesquelles il a fallu composer.

4.1 CudaLDIContactLMConstraint

Il a fallu très vite faire face à des bugs du composant « *CudaLDIContactLMConstraint* » dont la création était relativement récente très récent : certains problèmes ou artéfacts, passés inaperçus sur des scènes moins exigentes, sautaient aux yeux alors que l'on cherchaient à simuler non seulement un mécanisme, mais aussi à des vitesses très élevées. Certains ont été d'autant plus durs à corriger qu'ils ne se produisaient pas sur n'importe quelle machine, alors même que je ne connaissais pas encore assez le fonctionnement de SOFA et des contraintes pour pouvoir débugguer efficacement.

Ainsi certains bugs, en plus de ne pas se produire sur n'importe quelle machine ne se produisaient plus si l'on changeait l'échelle de la scène. Il a finalement été choisi d'utiliser un système d'unités homogène au système d'unités S.I., les distances étant alors considérées en millimètres plutôt qu'en mètres, et les masses en tonnes plutôt qu'en kilogrammes.

4.2 Discrétisation

Un premier problème était l'apparition de chocs au sein du moteur complet. On a tout d'abord suspecté un problème dans le composant résolvant les contraintes mais, alors qu'aucun problème n'a été isolé au sein de la résolution des contraintes mécaniques, il s'est avéré que le problème avait pour origine première des approximations au niveau de la détection de collisions au niveau de l'axe de la roue et du piston.

Ces approximations étaient parfois trop importantes : il en résultait alors de brusques mouvements des corps rigides mis en jeu. Il a fallut donc augmenter la précision de la détection de collision plus qu'on ne pensait avoir à le faire au dépard, particulièrement au niveau de l'axe.

4.3 Interpénétration

Un autre problème numérique à prendre en compte est inhérent à la méthode de résolution des contacts : c'est le fait qu'il s'agisse en fait d'une détection de collisions. On n'effectue en réponse aux collisions détectée qu'une correction des vitesses des objets et non de leurs positions. Cela introduit une certaine tolérance à leur interpénétration. La correction des positions est possible mais très instable dans le cas ou les objets contraints n'ont que peu de libertés de mouvements. Elle est inutilisable en l'état dans la simulation du moteur complet.

Cela n'a posé aucun problème pour la simulation de la partie du moteur, mais s'est avéré critique dans le cas du moteur complet, au niveau de l'axe permettant la liaison pivot entre la bielle et la roue. Il a fallu régler la finesse de la réponse à ce niveau. Trop peu de contraintes ne permettaient pas de contrôler l'interpénétration tout au long de la simulation, mais trop de contraintes ne permettent pas de répondre convenablement à la détection.

Il a fallu donc trouver un juste milieu. MCE-5 souhaitait en effet éviter de résoudre ce problème par des contraintes « dures » (fixer l'axe à la roue par exemple) pour que le maximum de choses soient véritablement simulées. Le but était de pouvoir en apprécier le potentiel et les limites de SOFA.

4.4 Objets déformables

Comme cela a déjà été précédemment évoqué, SOFA a été développé dans le but de créer des simulation « réalistes », et non pour faire des simulation précises d'objets quasi-rigides. Cela s'est ressenti lorsque l'on est passé à des objets déformables. Il s'est alors avéré problématique de calibrer les déformations des objets pour qu'ils correspondent à de l'acier. Il a été très facile d'obtenir des petites billes d'acier rebondissant de façon réaliste sur un sol rigide, mais difficile de calibrer en même temps l'ampleur de la déformation, son amortissement, et sa fréquence de résonnance, même sur le cas simple d'une bille posée sur un sol rigide.

5 Synthèse

5.1 PairwiseCudaRasterizer

Ce composant est le plus gros apport fait à SOFA durant ce stage. Les objectifs visés, l'économie de mémoire GPU et la possibilité d'adapter la finesse de la détection selon les paires d'objets sont parfaitement atteints.

Par ailleurs, la solution mise en œuvre est telle que l'on n'est plus limité par la quantité totale de mémoire que l'on aurait eu à utiliser avec le *CudaRasterizer*, mais uniquement par le contact qui nécessite le plus de mémoire. Cela apporte deux énormes bénéfices :

1. On peut alors utiliser, pour un seul contact, autant de mémoire que ce qu'on en utilisait pour l'ensemble de la scène. On peut donc également augmenter la précision de la détection pour une paire d'objets si l'on sait à l'avance que leur zone de contact va être limitée.
2. On peut augmenter la taille de la scène et le nombre d'objets, virtuellement sans limite du point de vue mémoire GPU, dès l'instant où l'on n'introduit pas la possibilité d'un contact top important. Quand bien même ce serait le cas, on pourrait y remédier grâce à la résolution variable.

Ce composant ne marche pas encore avec les fluides, et son fonctionnement avec les auto-collisions doit encore être mis à l'épreuve, mais on a déjà un composant qui aggrandit les possibilités d'utilisation du *CudaRasterizer*, et donc les champs d'applications de SOFA.

5.2 Échange de données

Là aussi, il était dans les objectifs initiaux de faciliter autant que possible le changement des modèles des pièces, et de faciliter les échanges de données entre SOFA et MCE-5.

Les paramètres de la simulation, notament la vitesse du vilbrequin et la force appliquée sur le piston dans le cas du moteur complet, sont lus dans un fichier. Ils peuvent changer ces paramètres d'entrée en changeant ce fichier. La tâche est facilitée par l'utilisation d'un format de fichier connu des tableurs utilisés par MCE-5.

Les scènes créées permettent de changer les modèles de pièces en changeant tout simplement de fichier STEP, le format de fichier utilisé par l'entreprise en interne comme avec leurs partenaires industriels. Tous les pré-calculs nécessaires à la simulation, particulièrement dans le cas où l'on créé un objet rigide (masse,

centre de masse et matrice d'inertie) sont effectués automatiquement. D'ailleurs, le composant *GenerateRigidMass*, créé pour l'occasion, a été d'ores et déjà intégré à SOFA.

L'export des forces de contact est effectif et les forces exportées ont été testées et validées sur des cas simples.

5.3 Simulation d'une partie du moteur

La simulation de la sous-partie du moteur a pris beaucoup plus de temps à être mise en place que prévu. Elle a permis de mettre en évidence des problèmes pré-existant dans SOFA qui ont du être réglés avant d'obtenir les premiers résultats de simulation.

Cette scène est exécutée très vite : environ 30% plus rapidement qu'avec le *CudaRasterizer*, ce qui mène à des temps inférieurs à 4 minutes sur un « DELL PRECISION 5500 » équipé d'une carte graphique « NVidia GTX 285 ». On arrive même en-dessous de 3 minutes si l'on se passe de toute sortie graphique.

Toutefois, les forces obtenues posent deux types de problèmes :

1. En rigide, elles sont extrèmement bruitées et les courbes obtenues n'ont pas encore pu être suffisament étudiées pour pouvoir conclure quant à leur pertinence.
2. En déformable, elles ne sont plus du tout bruitées, mais là non plus, elles n'ont pas encore été suffisament étudiées pour pouvoir conclure quant à leur pertinence.

5.4 Simulation du moteur complet

La scène du moteur complet a réservé elle aussi son lot de surprises. Au final, avec les modèles originaux, sans défaut, la scène se déroule bien mais plus lentement que prévu. Tout comme la précision de rastérisation a du être augmentée, le temps calcul l'a été tout autant. Au final, la simulation est effectuée en environ 1h15 sur le même « DELL PRECISION 5500 » équipé d'une carte graphique « NVidia GTX 285 ».

Le point critique était au dépard le rouleau de synchronisation, il s'agit à présent de l'axe de la roue. Il aurait été sans doute plus simple d'utiliser une contrainte pour fixer l'axe à la roue ou à la bielle, mais c'est une simplification dont MCE-5 voulait se passer. Du jeu au niveau de l'axe peut avoir des répercussion profondes sur le fonctionnement de tout le moteur. Poser une contrainte à cet endroit supprimerait

de fait une partie du jeu en cet endroit, et donc une partie des phénomènes qu'ils espèrent pouvoir simuler grâce à SOFA.

De plus, si l'application des forces ne posait aucun problème sur la partie du moteur, ici, l'application des forces sur le piston gène le déroulement de la simulation qui n'arrive plus à gérer l'interpénétration des pièces.

6 Annexes

A Export de données

Export des forces

Les forces sont tout le temps exportées automatiquement mais, si le but d'une simulation n'est véritablement que d'exporter les forces et que l'affichage est inutile, alors la désactivation des modèles visuels permet de gagner un temps précieux. Pour cela, il suffit, dans SOFA, d'aller dans l'onglet « Visual » du menu de gauche, et de sélectionner « désactiver le nœud » dans le menu contextuel du nœud racine.

Export des positions

La position d'une pièce peut être très facilement exportée à l'aide du composant « *Monitor* », tel qu'il est utilisé pour le rouleau, dans la scène rigide de la partie du moteur. Pour récupérer ces données pour d'autres objets rigides, dans la scène du moteur complet par exemple, il suffit de copier ce composant d'un objet à l'autre puis d'en changer le nom. Le nom du fichier créer pour exporter les données dépend du nom du composant « *Monitor* ».

Export d'une video

Pour faire une vidéo de la simulation, il faut tapper sur la touche « V » après avoir cliqué dans la fenêtre d'affichage. Une capture d'écran est alors sauvegardée à chaque pas de temps dans le répertoire « `~/Sofa/share/screenshots` » sous la forme : « `temp2.scn_00000565.png` ».

Pour transformer cette suite d'images en une vidéo, une solution est d'utiliser le logiciel ffmpeg[10]. Pour ce faire, ouvrez un terminal et tapez les commandes suivantes :

```
cd ~/Sofa/share/
ffmpeg -r 100 -sameq -i screenshots/temp2.scn_%08d.png ./video.mp4
```

Le nombre 100 détermine ne nombre d'images utilisées pour une seconde de vidéo. changer ce chiffre change donc la vitesse de la vidéo, mais ffmpeg ne pourra pas créer la vidéo si ce chiffre est trop important. Si l'on veut créer une vidéo plus rapide, on pourra également n'utiliser qu'une image sur dix afin d'éviter ce problème :

```
ffmpeg -r 100 -sameq -i screenshots/temp2.scn_%07d0.png ./video.mp4
```

[10] `http://www.ffmpeg.org/`

B Paramètres clefs

Voici une liste des quelques paramètres clefs contrôlant les scènes livrées.

Dans le nœud racine

MCE5CSVLoader
- filename : le nom du fichier où l'on ira chercher les données décrites plus tôt.

MCE5Config
density : densité des pièces
coeffFriction : coefficients de friction
coeffPoisson : coefficients de poisson des objets déformables
youngModulus : modules de Young pour les objets déformables

EulerImplicitSolver
rayleighStiffness : coefficient de « rayleigh damping » associé à la raideur
rayleighMass : coefficient de « rayleigh damping » associé à la masse

PairwiseCudaRasterizer
pixelSize : taille des pixels par défaut lors de la rastérisation
subVolutmeResolution : taille des sous-volumes par défaut, en pixels
resolutionRules (onglet Pairwise) : liste des configurations (pixel, résolution)

Dans les objets

MCE5LoaderSwitchTop
filename : fichier STEP original

GenerateRigidMass
density : densité de l'objet

MechanicalObject (behavior model)
rotation2 : rotation initale à effectuer sur l'objet
translation2 : translation initale à effectuer sur l'objet

HexahedronFEMForcefield (behavior model)
poissonRatio : coefficient de poisson
youngModulus : module de Young

TTriangleModel (collision Model)
contactFriction : coefficient de friction de l'objet

C Réinstallation des *drivers* CUDA

Afin de pouvoir utiliser l'API CUDA, il est nécessaire d'avoir installé des *drivers* spécifiques. Or, il peut être nécessaire de réinstaller ces derniers après une mise à jour du système, notament si cela concerne une mise à jour du noyau. Ainsi, si SOFA cesse de fonctionner normalement après une mise à jour (fenêtre d'affichage vide, crash au lancement du programme, ...), il est alors probablement nécessaire, et sûrement suffisant, de réinstaller les *drivers* CUDA en suivant la démarche suivante.

Arrêt de *GNOME*

Il faut d'abord arrêter l'environnement de bureau *GNOME* : l'interface graphique fenêtrée. Ouvrez une console et tapez cette commande.

```
sudo service gdm stop
```

Il vous sera demandé le même mot de passe que celui utilisé pour ouvrir la session au démarage de l'ordinateur. Une fois la commande exécutée, toute l'interface graphique habituelle est arrêtée et l'on se retrouve face à un terminal en ligne de commande. Connectez-vous en utilisant le même nom d'utilisateur et le même mot de passe que ceux utilisés habituellement au démarrage de l'ordinateur.

Édition d'un lien symbolique

À l'heure actuelle, les parties de SOFA écrites en CUDA doivent encore être compilés avec gcc-4.3. Mais Tout le reste est compilé avec gcc-4.4. Pour permettre de faire cela automatiquement, gcc-4.3 est utilisé comme compilateur par défaut. Or, pour réinstaler les drivers, il faut utiliser le même compilateur que celui utilisé pour pré-compiler le noyau : gcc-4.4. Il faut donc changer la version de gcc par défaut. Tapez les trois commandes suivantes :

```
cd /usr/bin/
sudo rm gcc
sudo ln -s gcc-4.3 gcc
```

Réinstalation des *drivers*

On en vient enfin à la réinstalation des drivers proprement dite :

```
cd  /Downloads
sudo ./NVIDIA-Linux-x86-190.53-pkg1.run
```

Rétablissement du lien symbolique

Il faut ensuite rétablir le lien symbolique précédemment modifié :

```
cd /usr/bin/
sudo rm gcc
sudo ln -s gcc-4.4 gcc
```

Démarage de *GNOME*

Il ne reste plus qu'à relancer l'interface graphique habituelle. La commande suivante permet de le faire tout en fermant la session créée pour la réinstallation des drivers. Son exécution donne à nouveau accès à l'écran de connexion habituel du démarage de l'ordinateur.

```
sudo service gdm start && exit &
```

Références

[1] François Faure, Sébastien Barbier, Jérémie Allard, Florent Falipou : Image-based collision detection and response between arbitrary volumetric objects. In *ACM Siggraph/Eurographics Symposium on Computer Animation (SCA)*, Dublin, Irlande, July 2008.
http://hal.inria.fr/inria-00319399
http://hal.inria.fr/inria-00319399/PDF/PDF/ldi08.pdf

[2] Jérémie Allard, François Faure, Hadrien Courtecuisse, Florent Falipou, Christian Duriez, Paul G. Kry : Volume contact constraints at arbitrary resolution. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)*, 29(3), August 2010.
http://www.sofa-framework.org/projects/ldi/
http://www.sofa-framework.org/projects/ldi/sig10ldi-preprint.pdf

[3] Jérémie Allard, Stéphane Cotin, François Faure, Pierre-Jean Bensoussan, François Poyer, Christian Duriez, Hervé Delingette, Laurent Grisoni : SOFA – an open source framework for medical simulation. In *Medicine Meets Virtual Reality (MMVR)*, pages 13–18, 2007.
http://hal.inria.fr/inria-00319416
http://hal.inria.fr/inria-00319416/PDF/MMVR07.pdf

[4] François Faure, Jérémie Allard, Stéphane Cotin, Paul Neumann, Pierre-Jean Bensoussan, Christian Duriez, Hervé Delingette, Laurent Grisoni : SOFA : a modular yet efficient simulation framework. In *Surgetica*, September 2007.
http://hal.inria.fr/inria-00319407
http://hal.inria.fr/inria-00319407/PDF/sofa-surgetica07.pdf

[5] Jérémie Allard, François Faure, Erwin Coumans, Kenny Erleben, Richard Tonge : IEEE VR 2009 tutorial on interactive physics simulation. In *IEEE International Conference on Virtual Reality*, March 2009.
http://www.sofa-framework.org/tutorial-vr09
http://www.sofa-framework.org/docs/vr09-sofa.pdf

[6] Bruno Heidelberger, Matthias Teschner, Markus Gross : Real-time volumetric intersections of deforming objects. In *Proc. of Vision, Modeling, Visualization (VMV)*, pages 461–468, 2003.
http://www.beosil.com/publications.html
http://www.beosil.com/download/VolumetricIntersections_VMV03.pdf

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)