MÉMOIRE
Présenté en vue d'obtenir
LE DIPLÔME D'INGÉNIEUR I.I.E

Rapport Final

Aurélien Pocheville

Modèles comportementaux pour le prototypage virtuel d'objets rigides avec retour haptique

Etude, benchmark et mise en place de différentes solutions

Directeur du stage : Mr Abderrahmane Kheddar, Maître de Conférences
Laboratoire des **S**ystèmes **C**omplexes.

Soutenu le 01/07/2003 devant le jury

M. A. FAYE	Maître de conférences	Examinateur
M. D. ROUSSEL	Maître de conférences	Examinateur

Remerciements

Je tiens à remercier en particulier Mr Kheddar, pour m’avoir accueilli et conseillé lors de ce stage. Ses conseils, son aide et son soutien m’ont été très précieux lors de ce stage.

Je tiens aussi à remercier mes co-stagiaires Guillaume Bouyer ainsi que Crispthone Devine, pour leur aide. Je remercie aussi Yann CHENU pour son aide et son support avec la librairie SWIFT++, dans le cadre de son stage de 2eme année.

Conservatoire National des Arts et Métiers

INSTITUT D'INFORMATIQUE D'ENTREPRISE

FICHE SIGNALÉTIQUE

Mémoire d'Ingénieur I.I.E.

Modèles comportementaux pour le prototypage virtuel d'objets rigides avec retour haptique

Auteur : Aurélien POCHEVILLE.

Directeur de recherche : M. Abderrahmane Kheddar, Maître de Conférences - HdR, Laboratoire des Systèmes Complexes, CNRS FRE 2494, Université d'Evry Val d'Essonne.

Descriptif : Mise en place d'un framework OpenGL et haptique afin de tester l'influence des modèles comportementaux sur le retour haptique. Implémentation de calculs de forces de contacts et de retour d'efforts. Etudes des différents modèles d'interactions entre la boucle haptique et la boucle comportementale.

Résumé

Le prototypage virtuel et ses dérivés (opérations en milieu virtuel) suscitent un grand intérêt parmi les industriels et les groupes de recherches. Ce besoin n'est pas, à l'heure actuelle, comblé. De plus, aucun standard n'existe quant aux différents composants d'une simulation virtuelle. Le projet Touch-Hapsys, dans lequel s'inscrit ce stage, plaide pour des avancées dans ce domaine. Les deux composantes principales d'un prototypage virtuel sont les suivantes :

- Un modèle comportemental, qui simule un comportement physique dans un monde virtuel,
- un retour haptique, permettant à l'utilisateur d'entrer en interaction avec cet environnement virtuel.

De réelles avancées sont encore possibles, tant au niveau des modèles comportementaux qu'au niveau du retour haptique. A l'heure actuelle, trois tendances principales existent parmi les modèles comportementaux :

- Les approches par pénalités, qui, de par leurs défauts, sont déconsidérées ;
- les approches par contraintes, les plus proches du monde réel. Toutefois ces méthodes sont aussi les plus lentes ;
- les approches impulsionnelles, qui gèrent, non plus les forces, mais des impulsions (changement instantané de vitesse).

Une quatrième voie émerge, qui consiste à sacrifier du réalisme physique pour se concentrer sur le « vraisemblale » . Cette voie, si elle est idéale pour le secteur du multimédia, n'est malheureusement que peu applicable dans le prototypage virtuel. Toutefois de nombreuses connexités existent.

Une simulation physiquement réaliste et un retour haptique de qualité sont antagonistes : là où l'un requiert un temps considérable en calculs, l'autre désire des temps de réponses très courts, et, bien entendu, du temps réel. La plupart des publications et des recherches en cours se concentrent sur l'un *ou* l'autre de ces aspects.

Les travaux engagés lors de ce stage de fin d'étude tentent de réconcilier ces deux domaines.

Afin de pouvoir tester des modèles, et d'avoir un rendu visuel des scènes, un framework OpenGL/haptique a été mis en place. Basé sur une architecture objet flexible et extensible, il est la base des travaux. Il intègre les éléments suivants :

- Un moteur de rendu graphique, ce moteur inclut la possibilité de gérer simplement des objets graphiques et d'afficher des informations,
- un moteur de configuration,
- un moteur de simulation, hautement configurable et extensible,
- une suite d'utilitaires venant avec le framework, (compilateur de fichiers 3d, créateur de fontes, convertisseurs...).

Un modèle comportemental a été implémenté avec succès, et des variations sur ce modèle ont été effectuées. Deux types de dispositifs haptiques ont été mis en place, l'architecture permet facilement l'intégration de futurs périphériques.

Des réflexions ont été engagées sur les bases théoriques des modèles comportementaux, et des lignes de conduite, quant à ces modèles, ont pu être déduites. Ces réflexions ne sont pas restées au stage académique, mais une implémentation a bien été réalisée.

Enfin, une profonde réflexion sur le retour haptique a été engagée. Ce domaine étant vaste et inexploré, il reste beaucoup à découvrir. Néanmoins, des constatations très intéressantes ont été réalisées, constatations accompagnées de tests physiques et d'implémentations.

Les travaux réalisés lors de ce stage constituent une avancée dans le domaine, en laissant une architecture ouverte afin d'améliorer le retour haptique couplé à un modèle comportemental de qualité.

Table des matières

1 Cadre du stage **13**
- 1.1 Coordonées . . . 13
- 1.2 Matériel . . . 14
- 1.3 Touch Hapsys : vers une meilleure présence haptique . . . 15
- 1.4 Planning . . . 15

2 Enjeux de la réalité virtuelle et de l'infohaptie **17**
- 2.1 Buts et objectifs de la réalité virtuelle . . . 17
  - 2.1.1 Présentation générale . . . 17
  - 2.1.2 Rendu visuel . . . 17
  - 2.1.3 Rendu sonore . . . 18
  - 2.1.4 Rendu haptique . . . 18
  - 2.1.5 Rendu olfactif . . . 19
  - 2.1.6 Modèle comportementaux, physique du modèle . . . 19
  - 2.1.7 Conclusion . . . 19
- 2.2 Enjeux de la simulation physique . . . 19
  - 2.2.1 Limitations de la simulation physique . . . 19
  - 2.2.2 Notations . . . 20
  - 2.2.3 Base physique de la simulation . . . 21
  - 2.2.4 Composantes de translations des lois physiques . . . 21
  - 2.2.5 Composante rotationnelles des lois physiques . . . 23
  - 2.2.6 Avantages des équations réelles . . . 25
  - 2.2.7 Réponses aux collisions . . . 25
  - 2.2.8 Étapes d'une simulation physique . . . 29

3 Framework de simulation et de retour haptique **31**
- 3.1 Framework OpenGL . . . 31
  - 3.1.1 Structure générale du framework OpenGL . . . 31
  - 3.1.2 Section système (`system`) . . . 32
  - 3.1.3 Section graphismes (`graphics`) . . . 32
  - 3.1.4 Section utilitaires (`utils`) . . . 35
  - 3.1.5 Section mathématiques (`maths`) . . . 36
  - 3.1.6 Section scène basique (`base`) . . . 36

3.1.7 Section entrées (`input`) . . . 36
3.1.8 Section entrées 6d (`6dinput`) . . . 37
3.1.9 Section interface utilisateur (`gui`) . . . 41
3.1.10 Section simulation (`simul`) . . . 41
3.1.11 Module `CSimulationManager` . . . 41
3.1.12 Module `CSimulationObject` . . . 42
3.1.13 Conclusion . . . 46
3.2 Approche envisagée au problème . . . 47
3.2.1 Une approche globale par contraintes . . . 47
3.2.2 Approche envisagée par Sauer et Schömer . . . 47
3.2.3 Résolution du LCP . . . 51
3.2.4 Problèmes inhérents au modèle choisi . . . 52
3.3 Retour haptique . . . 54
3.3.1 Pourquoi un retour haptique ? . . . 54
3.3.2 La boucle classique en haptique . . . 54
3.3.3 Retour en force . . . 55
3.3.4 Retour en position . . . 55
3.3.5 Contrôle en position (seule) . . . 57
3.3.6 Contrôle en vitesse (seule) . . . 58
3.3.7 Contrôle en position et en vitesse . . . 59
3.3.8 Un nouveau type de contrôle : le contrôle par requête . . . 60
3.3.9 La nouvelle boucle de simulation . . . 61

4 Conclusion **62**
4.1 Modèle comportemental . . . 62
4.2 Framework . . . 62
4.3 Retour haptique . . . 62
4.4 Bilan . . . 62

A Fichiers de configuration **66**
A.1 Fichier de configuration graphique (`graphics.opt`) . . . 66
A.2 Fichier de configuration de souris (`mouse.opt`) . . . 67
A.3 Fichier de configuration d'un objet (`obj_xx.opt`) . . . 67
A.4 Fichier de configuration du retour d'effort . . . 69

B Fichier de debug de l'application **70**

C L'utilitaire de création de fonte **72**

D Photos d'écran **73**
D.1 GUI en action . . . 73
D.2 Un simple contact . . . 74
D.3 Une cuvette : de nombreux contacts . . . 75

Table des figures

1.1 Le bras à retour d'effort `Phantom Desktop`,de Sensable 14
1.2 La SpaceBall de 3DConnexion 15

2.1 Trajet du point pendant Δt 22
2.2 Angles de rotations classiques 23
2.3 Deux solides en interpénétration 25
2.4 Méthode de pénalité - Le ressort repousse les deux solides 26
2.5 Méthodes de correction de la réalité - Retour dans le passé 27
2.6 Méthodes de correction de la réalité - Collisions entre **S1**, **S2**, **S3** 27
2.7 Méthodes de correction de la réalité - Position de **S2** corrigée 28
2.8 Méthode d'approximation de contact 29

3.1 Architecture du framework OpenGL 33
3.2 Temps pour le projet Haptics 36
3.3 Différents moyens d'entrées utilisateurs 37
3.4 Un délai est inévitable entre deux boucles tournant à des fréquences différentes . 40
3.5 Exemple en deux dimensions, de solides en contact 48
3.6 Une force est générée pour empêcher l'interpénétration 52
3.7 Resting contact - Les forces ne génèrent pas de rotations 52
3.8 Interpénétration - Le solide « s'enfonce » dans le sol 53
3.9 La boucle haptique 55
3.10 Un système masse ressort relie la position réelle du phantom à sa position virtuelle 56
3.11 Couplage des sphères par ressort 56
3.12 Une nouvelle sorte de rampe de ressort 58
3.13 Contrôle en position - Pas de contact 58
3.14 Contrôle en position - Interpénétration 59
3.15 Contrôle en vitesse - Accumulation des erreurs 59
3.16 Contrôle en position et en vitesse - En contact, les vitesses sont nulles 60
3.17 Contrôle en position et en vitesse - Les deux solides évoluent à la même vitesse, la différence de position ne change pas 60

Chapitre 1

Cadre du stage

1.1 Coordonées

Ce stage s'effectue sous l'égide de Mr Abderrahmane Kheddar, Maître de Conférences. Mr Kheddar peut être joint aux coordonnées suivantes :

- Adresse postale :

  ```
  Laboratoire Systèmes Complexes - Fre 2494 CNRS
  Université d'Evry Val-d'Essonne
  40, rue du Pelvoux CE1455
  Courcouronnes
  91020 Evry Cedex
  ```

- Adresse email :

  ```
  kheddar@iup.univ-evry.fr
  ```

- Téléphone :

  ```
  01 69477564
  06 81746722
  01 69477599 (fax)
  ```

Le stagiaire, quant à lui, peut être joint aux coordonnées suivantes :

- Adresse email :

  ```
  slurdge@slurdge.org
  pocheville@iie.cnam.fr
  ```

- Téléphone :

  ```
  06 60077529
  01 42370688 (domicile)
  ```

Le stage est réalisé au **LSC** (**L**aboratoire des **S**ystèmes **C**omplexes). Le **LSC** a pour but d'effectuer des recherches dans divers domaines de la robotique et de la réalité virtuelle. Il est décomposé en trois pôles principaux de recherche :

- Un pôle traitement des données et modélisation.
- Un pôle modélisation et contrôle des machines complexes.
- Un pôle réalité virtuelle et vision artificielle, dans lequel s'effectue ce stage.

1.2 Matériel

Le stage a été réalisé par un seul stagiaire, en collaboration avec le maître de stage Mr Kheddar. Les moyens mis en œuvre lors de ce stage sont les suivants (ces mêmes moyens ont servi lors des tests, si une échelle de performance est requise).

1. Hardware :
   - Pc de type x86 2.0 Ghz équipé avec 1 gigaoctet de mémoire.
   - Carte graphique dernière génération, ATI Radeon 9700.
   - Bras à retour d'effort de type PHANTOM (Figure 1.1).

FIG. 1.1 – Le bras à retour d'effort `Phantom Desktop`,de Sensable

   - Souris à six degrés de libertés (souris 6d) SPACEBALL et SPACEMOUSE de chez Logitech. Ces souris ne disposent pas de retour haptique, toutefois elle présentent l'avantage de donner les forces sur les trois axes de translation ainsi que de rotation (soit en tout 6 axes) (Figure 1.2).

   - Extensions possibles vers le bras à retour d'effort VIRTUOSE.
2. Software :

FIG. 1.2 – La SpaceBall de 3DConnexion

- Microsoft Visual C++ .NET version 2003, qui présente de nombreux avantages quand au respect de la norme ISO.
- 3DSMax version 5.
- Librairie OpenGL.
- Librairie SWIFT++[1].
- API de programmation de 3DConnexion.
- API de programmation de Sensable, GHOST.

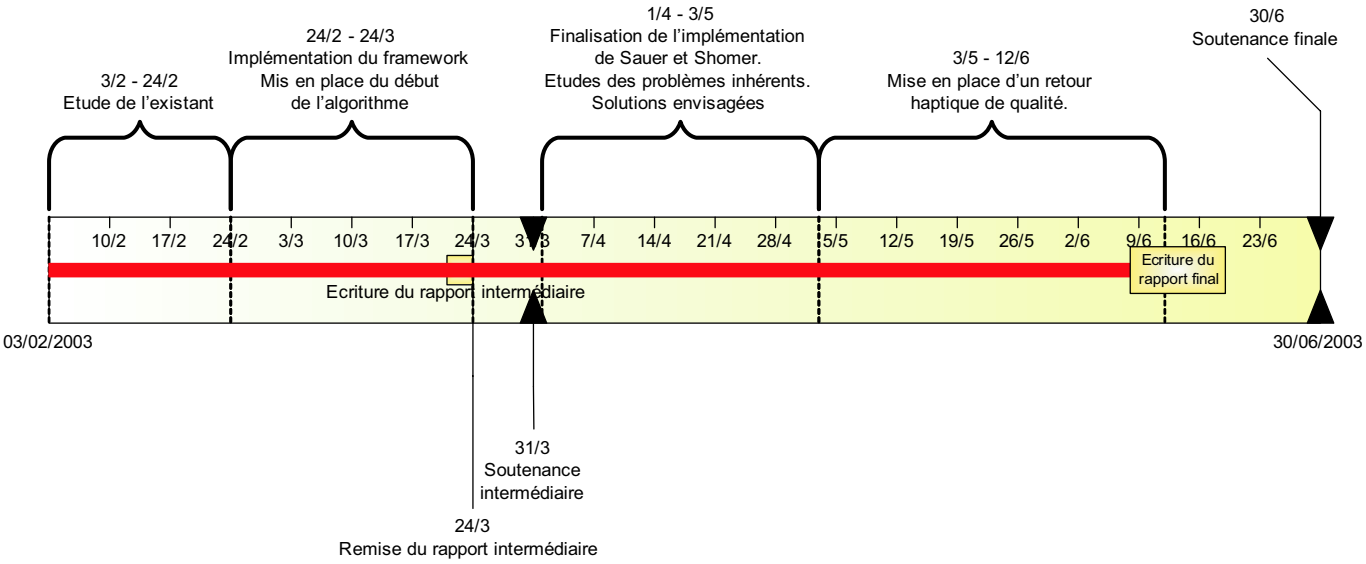
1.3 Touch Hapsys : vers une meilleure présence haptique

Ce projet fait partie de la recherche dans le cadre de Touch Hapsys, programme européen de recherche IST du 5eme PCRD. Le but de Touch Hapsys est de réaliser une recherche dans les techniques permettant de donner l'illusion d'un toucher. Pour cela, il est nécessaire d'investiguer les mécanismes neurologiques et psychologiques du toucher chez l'être humain, mais aussi les méthodes pour donner l'illusion d'un toucher. Dans cette optique, retour d'effort et l'exactitude des calculs physiques joue une importance non négligeable. Le prototypage virtuel, en ce sens, doit implémenter des techniques d'illusions haptiques réutilisables. De plus, la combinaison des sens visuel et du toucher peut provoquer un impact bien plus grand que chacun pris séparément. La page internet de Touch Hapsys peut être trouvée à l'url suivante : `http://www.touch-hapsys.org/`

1.4 Planning

Par rapport au planning présenté dans le rapport intermédiaire a été quelque peu modifié. En effet, il est apparu que la méthode envisagée posait des problèmes conceptuels et théorique, il a donc été décidé d'étudier ces problèmes afin de mieux cerner les centres d'intérêt dans les modèles comportementaux.

[1]SWIFT++ est un package de détection de collisions et de contacts, développé par le UNC Gamma Research Group. Plus d'informations sont disponibles sur `http://www.cs.unc.edu/~geom/`



Ensuite, l'implémentation d'un retour haptique de qualité, indépendant de la méthode de simulation choisie, a lui aussi posé de nombreux problèmes théoriques. Malgré cela, des résultats probants ont été obtenus vers la fin du stage.

Chapitre 2

Enjeux de la réalité virtuelle et de l'infohaptie

2.1 Buts et objectifs de la réalité virtuelle

2.1.1 Présentation générale

La réalité virtuelle consiste à « immerger » un être humain dans un monde virtuel, afin de lui donner l'impression d'avoir une expérience réelle. Les applications et les avantages d'une telle technique sont multiples : jeux, simulations, entraînement. La réalité virtuelle, contrairement à une idée répandue, ne consiste pas uniquement à avoir un rendu visuel, mais a pour but d'émuler tous les aspects de la réalité telle qu'elle est perçue par les sens humains : vue, ouïe, toucher, odorat et goût (ces deux derniers ne consistant du point de vue *biologique* quasiment que d'un seul sens). D'autre part, un autre aspect important de la réalité non virtuelle est l'application des lois physique, l'être humain étant capable très rapidement de différencier ce qui à l'air « naturel » de ce qui n'est « pas naturel ». Une brève description des différents aspects est donnée dans les section suivantes.

2.1.2 Rendu visuel

Actuellement, le rendu visuel progresse très vite. D'une part, les rendus off-line[1] sont très proche de la réalité en ce qui concerne le rendu d'être humains de synthèse, le rendu des décors ayant atteint la perfection (un œil humain ne peut pas percevoir la différence entre un décor de synthèse et un décor réel). Il suffit de regarder les images des derniers salons (Imagina, Virtual Laval, Siggraph...) pour s'en convaincre. D'autre part, avec l'explosion des cartes accélératrices 3D, le rendu on-line (en temps réel) rattrape le rendu pré-calculé. Les dernière cartes sont capables de calculer les projections et les transformations de plusieurs millions de pixels (nouvelles Radeon 9800 et GeForce FX), et cette tendance d'accélère au fil des années. Il est donc certain

[1] off-line peut être traduit par hors ligne, c'est-à-dire qui n'est pas calculé en temps réel, mais à l'avance pour une utilisation ultérieure.

que d'ici quelques années, le rendu visuel en temps réel sera très proche de la réalité, au point que l'opérateur humain aura beaucoup de mal à faire la différence.

2.1.3 Rendu sonore

Le rendu sonore a longtemps été délaissé au profit du rendu visuel dans les simulations. Toutefois, aujourd'hui, le matériel sonore est capable de rendre toute sorte d'effets en 3 dimensions et en temps réel. Il reste à combler le vide entre les résultats de calculs dans la réalité virtuelle et la synthèse des sons en 3 dimensions, ce qui n'est pas simple au vu de la complexité de certains sons (sons de frottements, sons de chocs ...). Les travaux de Guillaume BOUYER, un de mes co-stagiaires, sont fournis à ce sujet.

2.1.4 Rendu haptique

Le rendu haptique concerne le sens du toucher. Ce sens peut percevoir trois informations :

- la force exercé sur l'organe qui touche, c'est une information de pression.
- la texture de l'objet.
- la chaleur de l'objet.
- les informations de vitesse, encore méconnues.

Le premier rendu est souvent appelé rendu kinesthésique, les second et troisième sont regroupés en rendu tactile.

Le rendu kinesthésique, s'il est le plus avancé, souffre encore de quelques problèmes. Un de ces problèmes est la miniaturisation, certain appareils étant très gros et très lourds (du fait des moteurs). Cela limite fortement l'intérêt pratique (dans la vie courante par exemple) de ces dispositifs. Toutefois, les progrès en la matière sont notables, comme en témoigne le dispositif haptique **PHANTOM** par exemple. De plus ces dispositifs doivent souvent fonctionner à haute fréquence (de l'ordre de 1000 Hz), qui est bien supérieure au fréquences typiques de simulation (de l'ordre de 100 Hz). Il devient donc nécessaire d'effectuer une désynchronisation entre la boucle haptique et la boucle de simulation. Cette désynchronisation pose de nombreux problèmes de stabilité, comme il est discuté en section 3.3.2.

Au niveau des informations de texture et de température, l'état de l'art est encore dans une phase d'expérimentation. Les informations de textures nécessitent un dispositif ayant une résolution très fine (de l'ordre du 10eme de millimètre), ce qui n'est pas aisé à obtenir. De plus, le système doit avoir une stabilité optimale, car une quelconque vibration nuit gravement aux informations de textures perçues par l'opérateur. Le rendu de température est souvent obtenu par effet Peltier[2]. Toutefois, d'autres composantes interviennent dans les informations de températures : l'absorption de chaleur par exemple, n'est pas la même entre le bois (peu conducteur) et le fer (très conducteur). Le rendu tactile en est encore à ses débuts, de nombreuses avancées sont possibles.

Le rendu haptique kinesthésique est une des composantes importante du stage.

[2]Cet effet permet de transformer du courant électrique en différence de température

2.1.5 Rendu olfactif

Dû à la difficulté de synthétiser n'importe quelle odeur (ou n'importe quel goût) à partir d'un nombre fini d'odeurs de bases et d'opérations, ce domaine de la réalité virtuelle en est encore à ses débuts. Quelques expérimentations sont en cours, mais de nombreuses difficultés subsistent. Toutefois, il est à l'heure actuelle possible, assez facilement, d'avoir des *ambiances*, qui permettent de plonger l'opérateur dans une jungle, ou dans une boulangerie par exemple[3].

2.1.6 Modèle comportementaux, physique du modèle

L'opérateur trouvera assez peu d'intérêt à se trouver immergé dans un monde virtuel statique, i.e. où rien ne bouge. Il faut donc animer les objets. Cela est fait par l'intervention des modèles comportementaux. Certains modèles sont dérivés à partir de lois ou de dispositifs physiques (par exemple, un dispositif masse-ressort). Si ces modèles peuvent convenir dans les cas où l'on ne souhaite qu'une impression convenable, ils sont souvent refusés par les industriels dans le cas du prototypage virtuel. Le prototypage virtuel intervient très tôt dans la conception, et est destiné à remplacer tout ou partie des maquettes. Il est donc nécessaire que le modèle comportemental employé ait un sens physique exact au vu de la réalité (i.e. que la même manipulation dans la réalité produise le même résultat). Deux cas peuvent se produire :

- les objets ne sont pas en interaction avec l'utilisateur. Dans ce cas, l'ordinateur doit calculer le résultat des lois comportementales et mettre à jour la simulation ;
- l'opérateur est en interaction avec les objets, dans ce cas, la simulation devient plus compliquée, car il faut prendre en compte les forces entrantes et sortantes vers l'opérateur.

Le rendu physique, notamment à cause de sa relation étroite avec le rendu haptique, ansi que son importance capitale dans le prototypage virtuel, est une autre composante principale de ce stage.

2.1.7 Conclusion

Différentes techniques sont donc à mettre en œuvre afin de réaliser une immersion totale de l'utilisateur. Certains domaines sont plus avancés que d'autres, toutefois il reste encore beaucoup de progrès à faire dans tous les aspects de la réalité virtuelle.

2.2 Enjeux de la simulation physique

2.2.1 Limitations de la simulation physique

La simulation envisagée possède plusieurs limitations afin d'obtenir quelque chose de réalisable, car il n'est pas encore possible, à l'heure actuelle, d'implémenter tous les aspects phy-

[3]Cette technique est très au point, elle est souvent utilisée dans les supermarchés, pour donner envie au consommateur d'acheter, ainsi que pour lui donner l'illusion de passer d'un univers à un autre

siques du monde réel dans une seule et même simulation. Les limitations envisagées sont les suivantes :

- Les objets considérés sont des solides indéformables, dont la densité est homognène. A ce titre, ils possèdent certaines propriétés géométriques intéressantes et simplificatrices, parmi lesquelles :
  - Non déformabilité : cela permet de savoir que lors d'un choc ou d'un contact, aucun des deux solides ne se déformera. Donc les solides doivent se mouvoir pour répondre à ce choc.
  - La distance entre deux points quelconque du solide ne varie pas au cours de la simulation, cela est particulièrement pratique pour les algorithmes de détection de collision/-contact.
  - Les solides peuvent être modélisés comme des surfaces triangulées en lecture seule, très rapides au niveau du matériel pour le rendu.
- Le modèle ne prends en compte les objets que sous forme d'assemblage de polygones, même si ces assemblages peuvent être quelconques[4].
- Les forces envisagées sont les forces globales et les forces de frottement, dans un premier temps les rebonds et forces impulsionnelles ne sont pas prises en compte.

2.2.2 Notations

Dans la suite, les notations suivantes seront utilisées :

- Les scalaires sont noté normalement, comme ceci : γ
- Les vecteurs et assimilés sont notés en gras minuscule : $\mathbf{v}$
- Les matrices sont notées en gras majuscule : $\mathbf{M}$
- La transposée d'une matrice $\mathbf{M}$ est notée : $\mathbf{M}^T$
- Le produit scalaire n'a pas de notation, ainsi que le produit de quaternions. C'est le produit par défaut.
- Le produit vectoriel est noté $\times$
- La dérivation par rapport à t est parfois notée avec un point, $\dot{\mathbf{v}}$
- Le symbole de Krönecker est noté δ_{ij}, où

$$\delta_{ij} = \begin{cases} 1, & \text{si } i = j\,; \\ 0, & \text{si } i \neq j. \end{cases}$$

- Un petit intervalle de temps est noté Δt, dans la plupart des simulations (voire dans toutes), cet intervalle doit rester raisonnablement petit.
- Pour ne pas confondre l'identité avec une matrice d'inertie, la matrice identité d'ordre n est notée $\mathbf{E}_n$
- Lorsque l'on écrit $a \geq 0 \perp b \geq 0$, cela signifie que $a \geq 0$ et $b \geq 0$ et $a^T b = 0$.

[4] Ces assemblages sont parfois référencés dans la littérature comme « soupes de polygones » ou « polygon soup »

2.2.3 Base physique de la simulation

Toute simulation « physiquement réaliste » doit se baser sur les équations mathématiques dérivées des lois de la physique classique. Les simulations ne se basant pas sur les équations physiques (notamment les simulations à base de système masse-ressort) rencontrent de gros problèmes d'adéquation avec le monde réel dans le cadre du prototypage virtuel, et doivent mettre en jeu des systèmes de corrections souvent inexacts dans de nombreux cas.

Voici un résumé des résultats utilisés dans la majorité des simulations respectant les équation physiques classiques.
Les objets sont supposés solides (i.e. non déformables). Un objet est caractérisé par sa masse M, sa densité, $\rho(x,y,z)$ ainsi que par sa surface $S(x,y,z)$. Si la densité est constante, $\rho(x,y,z) = \rho_0 \ \forall (x,y,z) \in \mathbb{R}^3$, elle notée ρ. Dans ce cas, la surface définit l'enveloppe de l'objet. Sinon, on peut considérer que la surface est définie implicitement par la limite entre $\rho(x,y,z) \neq 0$ et $\rho(x,y,z) = 0$. Un tel objet peu être ramené à un point G de masse M, et de position G_x, G_y, G_z. La position de ce point est notée $\mathbf{p}$. La matrice d'inertie du solide est $\mathbf{I}$. Cette matrice est l'analogie de la masse pour les mouvements rotationnels, elle est calculée comme suit :

$$I = \int_V \rho(x,y,z) \begin{bmatrix} y^2+z^2 & -xy & -xz \\ -xy & z^2+x^2 & -yz \\ -xz & -yz & x^2+y^2 \end{bmatrix} dxdydz$$

2.2.4 Composantes de translations des lois physiques

La vitesse de ce point est sa dérivée par rapport au temps,

$$\mathbf{v} = \frac{d\mathbf{p}}{dt}$$

L'accélération est la dérivée de la vitesse, soit,

$$\mathbf{a} = \frac{d\mathbf{v}}{dt}$$

La somme des forces s'exerçant sur le solide est notée $\mathbf{f}$. L'équivalence entre force et accélération s'écrit alors

$$\mathbf{f} = M.\mathbf{a}$$

De ces équations, il est très facile d'écrire une version discrète entre l'instant t et l'instant $t + \Delta t$.

$$\mathbf{a}(t+\Delta t) = \frac{1}{M}\mathbf{f}(t+\Delta t)$$

A partir de ce point, divers schémas d'intégration numérique peuvent être envisagés. Un des plus classiques est, à l'ordre 1, pour déduire les vitesses puis les positions,

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \mathbf{a}(t+\Delta t)\Delta t$$

De même pour **p**

$$\mathbf{p}(t+\Delta t) = \mathbf{p}(t) + \mathbf{v}(t+\Delta t)\Delta t$$

On notera l'inexactitude de l'approche. Si $\Delta t \to 0$, alors les formules ne divergent pas. Toutefois, si Δt devient grand, des problèmes de divergences surgissent. A l'ordre 2, si l'on souhaite conserver l'énergie, on a directement l'équation suivante

$$\mathbf{p}(t+\Delta t) = \mathbf{p}(t) + \mathbf{v}(t+\Delta t)\Delta t - \frac{1}{2}\mathbf{a}(t+\Delta t)\Delta t^2$$

Cet ordre 2 permet d'avoir une conservation de l'énergie dans le système. Preuve, pour les mouvements de translation (cette preuve peut être étendue aux mouvements rotationnels) :

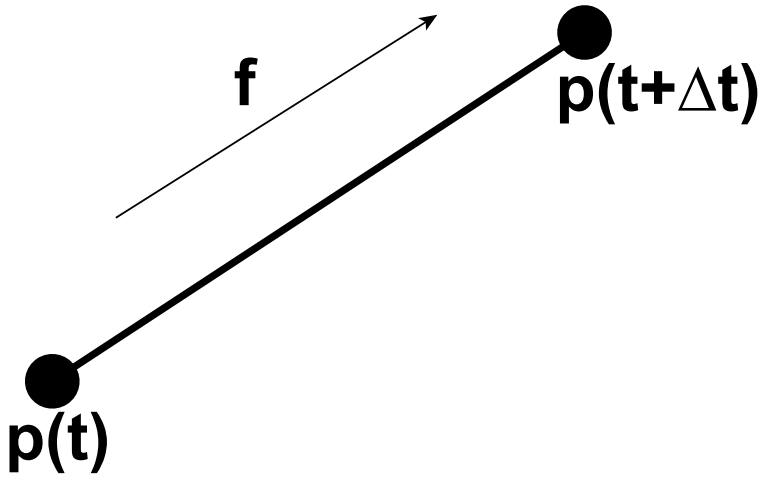


FIG. 2.1 – Trajet du point pendant Δt

Pendant Δt, le solide parcourt une ligne droite (Figure 2.1). Soient ses paramètres de départ $\mathbf{p}(t)$, $\mathbf{v}(t)$. Son énergie vaut $E(t)$. A $t+\Delta t$, son énergie vaut

$$E(t+\Delta t) = E(t) + \frac{1}{2}M\left(\mathbf{v}(t+\Delta t)^2 - \mathbf{v}(t)^2\right)$$

avec

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{\mathbf{f}}{M}\Delta t$$

La différence d'énergie vaut

$$E(t+\Delta t) - E(t) = \frac{1}{2}M\left(\mathbf{v}(t)^2 + \left(\frac{\mathbf{f}\Delta t}{M}\right)^2 + 2\mathbf{v}(t)\frac{\mathbf{f}}{M}\Delta t - \mathbf{v}(t)^2\right)$$

$$E(t+\Delta t) - E(t) = \frac{1}{2}\frac{(\mathbf{f}\Delta t)^2}{M} + \mathbf{v}(t)\mathbf{f}\Delta t$$

Le travail de la force vaut

$$W = \int_{Chemin} \mathbf{f} d\mathbf{p}$$

$$W = \mathbf{f}\left(\mathbf{p}(t+\Delta t) - \mathbf{p}(t)\right)$$

$$W = \mathbf{f}\left(\mathbf{v}(t)\Delta t + \frac{\mathbf{f}}{M}\Delta t^2 - \frac{1}{2}\frac{\mathbf{f}}{M}\Delta t^2\right)$$

$$W = \frac{1}{2}\frac{(\mathbf{f}\Delta t)^2}{M} + \mathbf{v}(t)\mathbf{f}\Delta t$$

D'autres schémas existent, comme celui-ci, qui approxime les courbes par des trapèzes (le précédent l'approximant par des rectangles).

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{1}{2}\left(\mathbf{a}(t) + \mathbf{a}(t+\Delta t)\right)$$

2.2.5 Composante rotationnelles des lois physiques

La représentation des translations par des vecteurs est simple et efficace, de plus elle permet de faire des calculs d'interpolation et de composition facilement et rapidement. Malheureusement, il n'existe pas d'équivalent pour les rotations et les torseurs.

De nombreuses simulations physiques utilisent trois angles pour représenter leurs rotations : x, y, z ou bien α, β, γ (Figure 2.2). Ces angles sont parfois appelés angle d'Euler, parfois roulis,

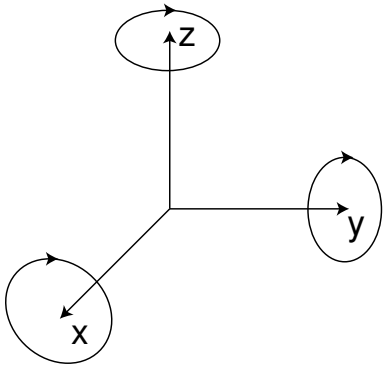


FIG. 2.2 – Angles de rotations classiques

tangage, lacet[SN01]. Toutefois, cette représentation, si elle facile à se représenter mentalement facilement, possède de nombreux défauts. Tout d'abord, il existe un problème, appelé « gimbal lock » ou singularité, qui intervient à cause du fait que plusieurs combinaisons d'angles peuvent exprimer la même rotation. Cette redondance empêche d'effectuer correctement les rotations autour de certains axes lorsque les rotations autour des autres axes sont proches ou égales à 90°.

De plus, il est très difficile de calculer l'interpolation entre deux rotations sous cette forme. Une alternative existe sous la forme axe-angle, cette représentation, si elle est toujours facile à se représenter mentalement, se prête bien à l'interpolation. Toutefois, il devient ardu de composer les rotations, car on est obligé de passer par des représentation intermédiaires (matrices,...). Il existe une solution qui pallie à tous ces problèmes : les *quaternions*. Les *quaternions* sont une généralisation des nombres complexes et permettent, entre autre, de représenter une rotation. Toutefois il n'est pas possible de se les représenter facilement mentalement. Il existe des formules de composition (multiplication), de dérivation et d'interpolation simples. De plus, il est assez aisé de les transformer en matrices de rotation et vice versa. Un quaternion est noté sous la forme suivante.

$$q = (q_x, q_y, q_z, q_w)$$

Les composantes (q_x, q_y, q_z) sont les composantes imaginaires du quaternion, et q_w est la composante réelle du quaternion. Les quaternions sont à mi-chemin entre les vecteurs et les nombres imaginaires. Une rotation d'angle α autour de l'axe (x,y,z) avec (x,y,z) vecteur unitaire, s'écrit sous forme de quaternion :

$$q_{\alpha,x,y,z} = (sin(\alpha)x, sin(\alpha)y, sin(\alpha)z, cos(\alpha))$$

Dans certaines publications, q_w est placé en premier [HFK94], mais dans ce cas les formules sont changées de manière adéquates. La multiplication de deux quaternions s'écrit de la manière suivante.

$$\begin{aligned}\mathbf{q1}(q1_x, q1_y, q1_z, q1_w).\mathbf{q2}(q2_x, q2_y, q2_z, q2_w) \quad = (\quad & q1_x q2_w + q1_w q2_x + q1_y q2_z - q1_z q2_y, \\ & q1_w q2_y - q1_x q2_z + q1_y q2_w + q1_z q2_x, \\ & q1_w q2_z + q1_x q2_y - q1_y q2_x + q1_z q2_w, \\ & q1_w q2_w - q1_x q2_x - q1_y q2_y - q1_z q2_z)\end{aligned}$$

On notera que la multiplication n'est pas commutative (comme il est d'habitude avec les rotations).

Il semble donc très judicieux d'utiliser les quaternions pour les rotations et les vitesses angulaires, car ils représentent toutes les caractéristiques des rotations, tout en étant extrêmement facile à interpoler et à concaténer. Une vitesse angulaire est notée $\omega(\omega_x, \omega_y, \omega_z, 0)$. Il est à noter que le terme en w n'a pas de sens pour une vitesse angulaire. Dans ce cas, la différenciation d'un quaternion de rotation s'écrit par une multiplication de quaternions.

$$\dot{\mathbf{q}} = \frac{1}{2}\omega\mathbf{q}$$

Si une force $\mathbf{f}_i$ est exercée sur le solide au point $\mathbf{p}_{f_i}$, alors elle engendre un couple de valeur :

$$\dot{\omega} = \mathbf{I}^{-1}(\sum(\mathbf{p}_{f_i} - \mathbf{p}) \times \mathbf{f}_i)$$

où $\mathbf{I}$ est la matrice d'inertie du solide considéré, et $\mathbf{p}$ la position du centre de gravité.
On peut alors utiliser les mêmes formules d'intégration numériques que précédemment.

2.2.6 Avantages des équations réelles

Mise à part l'intégration numérique, le modèle utilisé est le même que celui en physique classique, on est donc assuré de l'exactitude (aux imprecisions de l'intégration numérique près) des résultats. Le fait de vouloir une réponse « physiquement » exacte introduit la nécessité d'une contrainte : dans le monde réel, les objets ne disposent pas d'une marge pour s'interpénétrer. Il faut noter que le fait d'utiliser l'intégration numérique, si elle introduit des erreurs, permet de simuler des systèmes pour lesquels on ne possède pas de solution analytique.

2.2.7 Réponses aux collisions

Un des principaux problèmes des simulations, et surtout des simulations temps réel, est que le pas de temps est discret avec une précision médiocre (car la simulation prend du temps à calculer et à afficher). Cela entraîne ce que l'on pourrait appeler des « sauts » , et ne permet pas d'envisager de détecter le contact « au moment où il se produit » sans modifications. La plupart du temps, les solides, à un moment ou à un autre, s'interpénètrent, ce qui n'est pas possible physiquement (les objets sont supposés indéformables). Il se produit alors une situation comme celle présentée en Figure 2.3

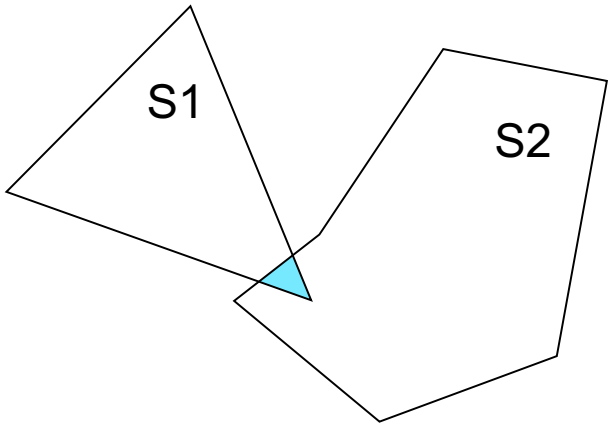


FIG. 2.3 – Deux solides en interpénétration

Cette situation est une des choses que les simulations cherchent à tout prix à éviter. Diverses solutions ont été envisagées.

• ***Méthodes de pénalités***

Ces méthodes reposent toutes sur le même principe : si les solides sont l'un dans l'autre, il faut rétablir une situation normale. De plus, plus les solides sont en interpénétration, plus il est nécessaire d'effectuer une correction. En général, cela est obtenu grâce à un système de ressort qui est introduit lors de la collision [Gal02], comme illustré par la Figure 2.4.

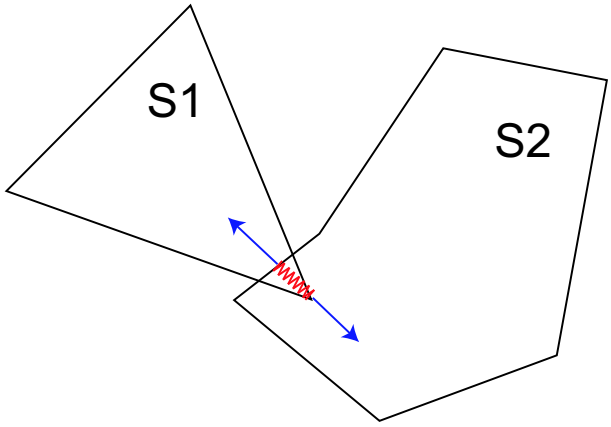


FIG. 2.4 – Méthode de pénalité - Le ressort repousse les deux solides

La constante de raideur du ressort définie la répulsion qu'auront les deux objets l'un pour l'autre. La formule utilisée, pour calculer la force, est

$$\mathbf{f} = \lambda \mathbf{P}_1 - \mathbf{p}_2$$

Toutefois, les méthodes de pénalités possèdent, de part leur structure, de nombreux défauts. Premièrement, elles sont locales, et ne tiennent compte que de l'interpénétration à un endroit. Le fait est que le solide est certainement en interpénétration à de nombreux endroits, et que les calculs n'en tiennent pas compte. Deuxièmement, les ressorts, outre leur totale inexactitude physique, posent des problèmes d'intégration numérique. Afin que la répulsion soit rapide, les ressorts doivent avoir une constante de raideur forte, ce qui induit des divergences assez rapidement si le pas de temps est trop grand. Cela oblige à avoir un pas de temps court, ce qui n'est pas forcément possible lorsque l'on fait du temps réel (notamment à cause du non-déterminisme des algorithmes de détection de collision). Les méthodes de pénalité sont toutefois très faciles à implémenter, ce qui explique leur fréquente utilisation.

• ***Méthodes de « backtracking*[5] *»***

Ces méthodes sont proches des méthodes de pénalités, en ce sens qu'elles laissent les solides s'interpénétrer et qu'elles corrigent cette interpénétration. Toutefois, elles sont physiquement plus exactes, car elles cherchent la configuration *dans le passé* qui n'a pas violé la contrainte. Ensuite, une force est appliquée afin d'éviter l'interpénétration. On se retrouve alors dans la configuration de contact, comme montré par la Figure 2.5. Il reste à faire tourner la simulation pour le temps restant, et recommencer l'opération si une nouvelle collision est détectée pendant.

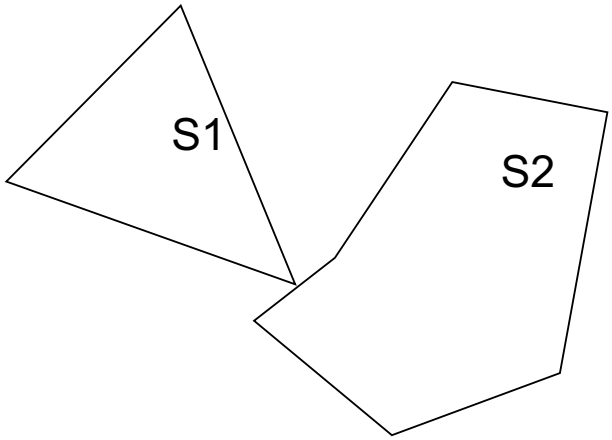


FIG. 2.5 – Méthodes de correction de la réalité - Retour dans le passé

Ces méthodes se rapprochent aussi en cela des méthodes à base contraintes. Toutefois, elle restent à un niveau local, et donc posent des problèmes lorsque de nombreux solides s'interpénètrent à proximité (par exemple dans le cas d'empilements). Ce type de problème s'étend à toutes les méthodes de gestion locale.

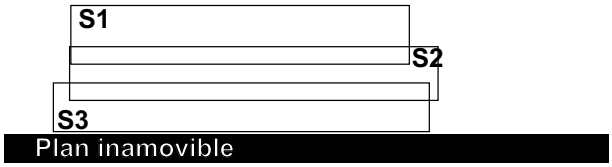


FIG. 2.6 – Méthodes de correction de la réalité - Collisions entre **S1**, **S2**, **S3**

La Figure 2.6 montre deux intersections entre les solides **S1**, **S2**, **S3**. La simulation choisi de résoudre le problème **S1**-**S2** en premier. Le solide **S2** se retrouve alors dans la situation montrée

[5]Cela pourrait être traduit par retourner dans le passé

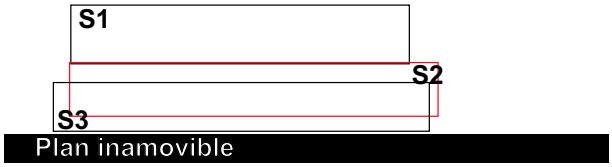


FIG. 2.7 – Méthodes de correction de la réalité - Position de **S2** corrigée

en Figure 2.7. Il devient alors impossible de corriger la position de **S3** sans changer de nouveau **S2**, ce qui entraîne de nombreux problèmes de propagation. Certains algorithmes de détection de collisions continus permettent de lever cette indétermination, en résolvant les contacts dans l'ordre (mais cela fait autant de calcul en plus, car on est obligé de recalculer les positions des objets après chaque résolution).

• ***Méthodes de gestion globale***

Les méthodes de gestion globale utilisent la totalité des informations présentes dans la simulation afin de résoudre le problème des collisions. On sait que, pour chaque **contact**, certaines conditions doivent être vérifiées sur la force et l'accélération. Cette technique a été envisagée en premier par Baraff dans [Bar89]. Elle a été ensuite étendue et reprise par Baraff lui-même, et par d'autre auteurs, dont Sauer et Schömer, dont l'article sert de base à l'implémentation réalisée. L'expression de ces conditions sous forme matricielle permet de ramener le problème à un LCP[6]. Toutefois, cette méthode pose le problème de connaître les contacts exacts, ce qui implique, si il y a eu interpénétration, de « remonter dans le temps » pour arriver à la configuration où il y a contact. De plus, comme les contacts sont rarement simultanés, cela oblige à résoudre les équations une fois par contact, ce qui peut se révéler coûteux en temps de calcul. Ensuite, il faut résoudre le LCP, et cela peut aussi se révéler coûteux en temps de calcul (n'oublions pas l'objectif est d'avoir une simulation temps réel). Toutefois, cette méthode est beaucoup plus exacte physiquement, c'est donc la méthode qui est envisagée lorsque l'exactitude physique est désirée.

• ***Méthode d'approximation de contact***

Afin de pouvoir évaluer rapidement les contacts en une fois, et ne pas à avoir à évaluer plusieurs fois les forces, une approche nouvelle a été envisagée. C'est l'approche qui est actuellement implémentée dans la simulation. Au lieu d'attendre que les solides s'interpénètrent, on considère que les solides sont en contacts si leur distance est inférieure à un seuil ε. Si ε est suffisamment petit, cela ne nuit pas à l'exactitude de la simulation, et l'opérateur ne perçoit pas de différence. Le principe de la méthode est illustré par la Figure 2.8.

Dans les premiers tests effectués, l'opérateur à effectivement le sentiment que la "collision" a eu lieu lorsque les objets sont très proches (avec ε de l'ordre du centième de la taille carac-

[6]Linear Complementary Problem, une excellente source sur les problèmes de complémentarité linéaire peut être trouvée dans [Mur97]

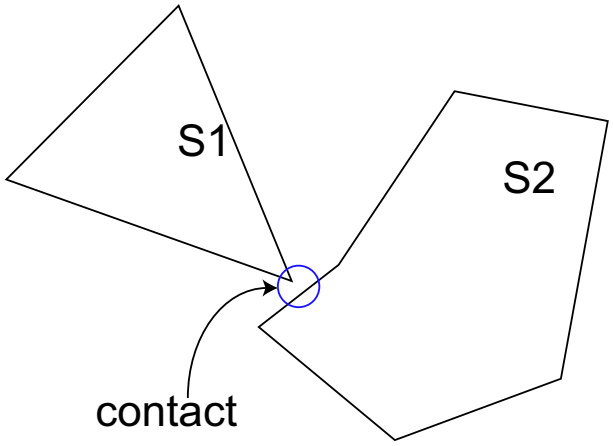


FIG. 2.8 – Méthode d'approximation de contact

téristique des objets). De plus, on peut jouer sur le nombre de contacts détectés en jouant sur ε. Le but est qu'à tout instant du calcul, il n'y ait **jamais** interpénétration, ceci permet de faire des suppositions utiles par la suite. De plus, si un contact exact est nécessaire par la suite, cette approche ne remet pas en cause la suite des calculs, on peut donc sans crainte implémenter un calcul de contact exact par la suite et être assuré qu'il fonctionne avec le reste de la simulation. Cette méthode semble très prometteuse, et ne semble pas avoir été envisagée sérieusement dans d'autres simulations.

2.2.8 Étapes d'une simulation physique

Les différentes étapes d'une simulation physique classique sont les suivantes :

- Initialisation des positions, des vitesses et des accélérations des différents solides.
- Mise à jour des différentes positions.
- Calcul des collisions.
- Calcul de la correction.
- Correction des positions.
- Affichage des positions.

Il est à noter que si le pas de temps est petit, le moment où les positions sont affichés importe peu, car l'œil humain de percevra pas la différence.

L'approche suivie diffère un peu de cette approche classique, car il n'y a pas de correction. L'étapes de détection de collision est remplacée par une étape de détection de contact, ensuite les forces dues à ces contacts sont calculés. Ces forces interviendrons lors du calcul des positions grâce au schéma d'intégration numérique envisagé.

Les étapes deviennent :

- Initialisation des positions,des vitesses et des accélérations des différents solides.
- Mise à jour des différentes positions (les forces dues aux contacts sont prises en compte).
- Calcul des contacts.

– Calcul des forces dues aux contacts.
– Affichage des positions.

Bien entendu, cette boucle doit être changée dans le cas où il y a un retour de force, cela est expliqué en 3.3.9.

Chapitre 3

Framework de simulation et de retour haptique

3.1 Framework OpenGL

Afin de pouvoir tester les différentes approches haptiques et les différents modèles, un framework OpenGL encadre le rendu visuel ainsi que les entrées diverses. Ce framework se doit d'être flexible afin de pouvoir intégrer de nouveaux modèles, performant, et simple d'utilisation. Dans les section suivantes, le détail de ce framework est donné. Le framework mis en place à pris appui sur des travaux réalisés en deuxième et troisième année d'étude à l'IIE, certaines parties, très portables, ont ainsi put être reprises (par exemple certains parsers).

3.1.1 Structure générale du framework OpenGL

La framework est composé de plusieurs modules, eux-mêmes regroupés en section traitant d'un aspect particulier du programme. Ces sections correspondent à un répertoire dans le répertoire source. Les diverses sections sont décrites brièvement ci-dessous, une description plus détaillée est donnée dans les sections suivantes :

- `system` : Cette section traite du système. Elle opère avec le système pour créer le programme et enregistrer les différentes composantes de celui-ci (système de rendu graphique, système de rendu physique, etc).
- `graphics` : Cette section est assez importante (environ 100 kilo-octets de code), et contient tout ce qui concerne l'affichage, de manière « brute » . Cette section contient aussi la gestion des scènes et les objets de scène basique.
- `utils` : Deuxième section importante, elle comporte de nombreuses classes utilitaires qui permettent au programme d'être écrit et utilisé plus facilement. Elle contient, entre autre, des parsers, un système de deboguage et d'erreur, ainsi qu'un timer.
- `maths` : Cette section contient tout ce qui a trait aux mathématiques, notamment les calculs matriciels. Une classe est dédiée aux opérations et les calculs sur les quaternions.
- `input` : Cette section traite des inputs divers, qu'ils viennent du clavier, de la souris.

– `6dinput` : Cette section traite des dispositifs haptiques ou non, qui possèdent 6 degrés de libertés. Elle est chargée de périphériques tels le bras PHANTOM Desktop ou bien une SPACE BALL (souris 6D).
– `base` : Cette section contient une implémentation de scène de base, ce qui permet de developer et de tester des solutions très rapidement.
– `gui` : Cette section contient une GUI[1] basique, écrite pour être indépendante du rendu,
– `simul` : Cette section contient les fichiers propres à la simulation. Entre autre modules, il existe un module qui définit un objet solide et un module réalisant la liaison entre les différents algorithmes.

Les différentes section du framework ont une consistence dans le codage et dans la structure. L'initialisation des différentes classes est non pas réalisée par le constructeur, mais par la fonction `Init()`. Cela permet, pour les objets statiques, de contrôler le moment de l'initialisation (car pour les objets statiques ou les variables membres, l'ordre d'instanciation est laissé au compilateur). Or l'ordre d'initialisation est souvent très important. De plus, les constructeurs n'ont pas de valeur de retour, il est donc difficile de gérer les problèmes d'initialisation. C'est pourquoi il a été choisie de faire appel à des fonction `Init()` et `Shutdown()`.

D'autre part, l'héritage C++ a beaucoup été utilisé afin d'avoir des fonctionnalités de bases partagées.

Un schéma de l'architecture globale du Framework est donnée en Figure 3.1.

3.1.2 Section système (`system`)

Cette section est chargée de l'initialisation et de la terminaison de l'application. Une classe principale est chargée de cela, il s'agit de la classe `CApplication`. Afin de préserver une portabilité du Framework, la `GLUT` est utilisée pour la créations des fenêtres et la gestion événementielle. La classe `CApplication` contient deux fonctions principales, `Init` et `Shutdown`. Pour implémenter une application, il suffit de créer un objet `CApplication` et de l'initialiser. La fonction `Shutdown` sera appelée automatiquement, et la libération des ressources sera faite. Voici un exemple d'usage de la classe `CApplication` :

```
void main(int _iArgc, char *_pcArgv[])
{
    CApplication::Init(_iArgc,_pcArgv);
}
```

3.1.3 Section graphismes (`graphics`)

Cette section est chargée de la gestion de l'affichage et des modes graphiques. La gestion de l'affichage est basé sur des scènes et les objets de scènes.

[1] **G**raphics **U**ser **I**nterface

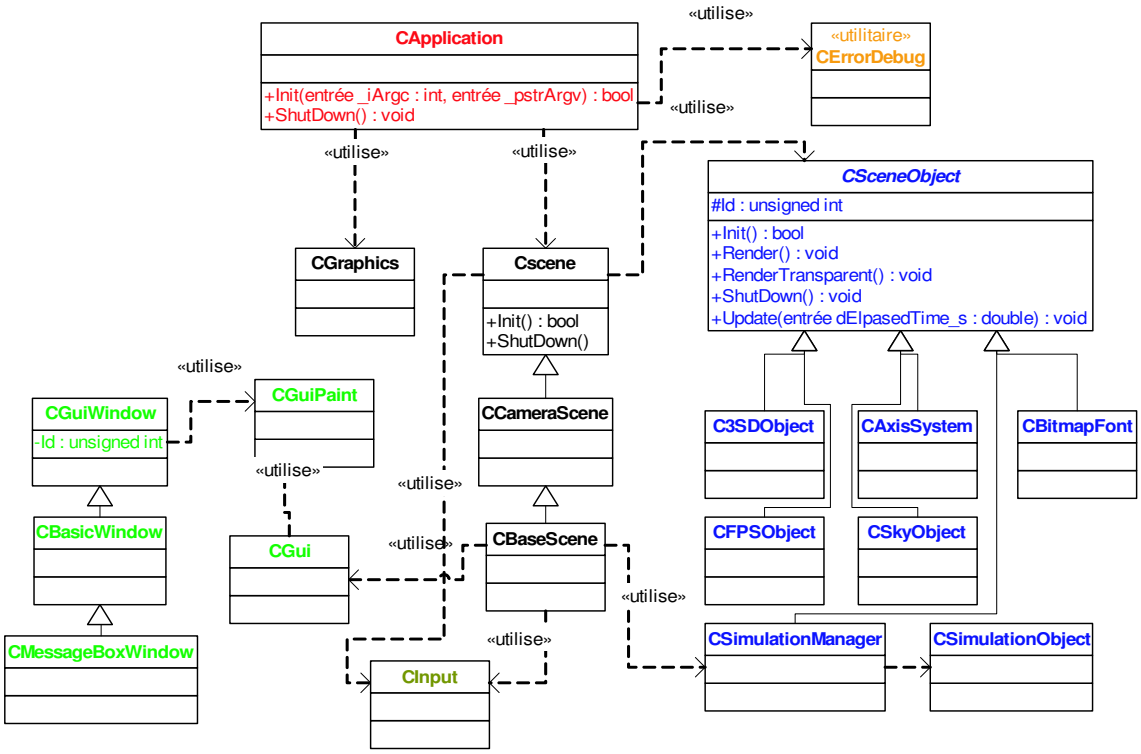


FIG. 3.1 – Architecture du framework OpenGL

La classe `CGraphics` est une classe qui gère l'affichage à un niveau global. Elle initialise la partie graphique de la `GLUT` ainsi que OpenGL. Elle permet aussi d'énumérer les modes graphiques, de gérer la projection 2D/3D OpenGL, et de libérer les ressources quand elles ne sont plus utilisées.

Afin de gérer simplement l'affichage, le programme est basé sur une gestion par scène/objet. Chaque scène représente un aspect particulier du programme, par exemple une scène peut être dédiée aux écrans de configuration, une autre scène peut être dédiée à la simulation. Chaque scène dérive de la classe mère (`CScene`), et implémente un certain nombre de méthodes obligatoires. Elle peut de plus, implémenter un certain nombre de méthodes optionnelles. Un mécanisme « d'extensions » existe pour savoir si une scène prends en charge certaines extensions. Si elle les prends en charge (par exemple la gestion du clavier est une extension), on fait appel à ces fonctions, sinon le programme ignore ces fonctionnalités mais garde tout de même un comportement cohérent.

Les objets sont tous dérivés de la classe `CSceneObject`, qui définit l'interface des objets. Un apercu de cette classe est donnée ci-dessous. Pour des raisons de concisions, la gestion de l'enregistrement dans un conteneur C++ a été omise, ainsi que les commentaires.

```
class CSceneObject {
```

```
public:
    virtual void    Render() = 0;
    virtual void    RenderTransparent();
    virtual bool    Init(){return false;}
    virtual void    ShutDown(){}
    virtual void    Update(double _dElapsedTime_s){}
    };
```

Chaque object se doit d'implémenter `Render()`, afin de pouvoir être rendu à l'écran. Deux approches peuvent être utilisées pour gérer les objets dans la scène courante :

- Une gestion avec conteneurs C++, utile lorsque la majorité des objets est créée dynamiquement.
- Une gestion en utilisant les objets comme variables membres de la classe de scène utilisée. Cette gestion sert surtout lorsqu'il y a peu d'objets de scène, et qu'ils sont tous connus à l'avance.

Lors de l'implémentation d'une simulation, la deuxième solution convient le mieux, car c'est l'objet simulation qui va gérer les objets de la simulation. La scène, quand à elle, n'aura à gérer que peu d'objets graphiques supplémentaires.

Des objets de base sont déjà implémentés afin de permettre leur intégration rapide dans une scène.

- `CAxisObject` : Cet objet permet de représenter le système de coordonnées actuel grâce à des primitives OpenGL simple. On peut le voir dans les captures d'écran en annexe XXX.
- `CBitmapFont` : Cet objet est une classe de fonte bitmapée puissante. Elle permet d'afficher rapidement du texte anti-crénelé[2]. De plus, elle est complètement dirigée par un fichier de configuration, ce qui permet d'avoir plusieurs comportement de fontes très facilement. Un utilitaire Windows a été développé en parallèle, qui permet d'exporter les fontes Windows en fontes bitmapées. Une photo d'écran de cet utilitaire est donnée en Annexe C.
- `CFPSObject` : Cet objet permet d'avoir rapidement et facilement un compteur de frames[3] par seconde. Ce compteur peut se positionner dans les 4 coins de l'écran, et possède un rafraîchissement variable. La seule contrainte est qu'il doit être mis à jour à chaque frame, mais en général tous les objets sont mis à jour donc ce n'est pas un problème.
- `CMouseCursorObject` : Cet objet implémente un curseur de souris bitmapé, géré par un fichier de configuration avancé (cf Annexe A.2).
- `CSkyObject` : Cet objet permet d'avoir un fond non uni, donnant l'illusion d'un décor. La classe n'a besoin que d'une texture pour arriver à cet effet.
- `3DSObject` : Cet objet ouvre la possibilité de charger n'importe quel modèle tri-dimensionnel crée à partir de 3DSMax. Il travaille en collaboration avec un programme qui compile les fichiers exportés à partir de 3DSMax. Cela permet d'avoir une indépendance à propos du logiciel de modélisation, et permet aussi d'envisager deux modèles de représentation des

[2] anti-crénelé signifie que la figure est très légèrement flouée pour éviter les effets de pixellisation du aux écrans d'ordinateurs, surtout visibles sur les écrans plats ou les écrans ayant une très bonne définition des pixels.

[3] Une frame correspond à une image affichée entièrement.

objets, un avec une normale par arrête (plus simple pour les objets lisses), un avec une normale par face.

En plus des objets de bases, la section graphismes contient des fichiers utilitaires tels que le chargement de texture, la lecture l'écriture de fichiers aux format tga, eps...

Enfin, elle contient aussi une classe qui dérive de la classe de scène générique, et qui implémente une gestion de la caméra. Cela est utile pour pouvoir se "promener" dans une scène, ou observer un phénomène de plus près (par exemple il est très intéréssant de "zoomer" sur une collision probable pour voir si l'algorithme de collision se comporte normalement).

3.1.4 Section utilitaires (`utils`)

Cette section contient de nombreux utilitaires aidant à un développement plus rapide, plus efficace et plus sûr. Voici les principaux modules qui sont présents.

- `CErrorDebug` Cette classe permet d'avoir un mécanisme simple et efficace de debug et d'erreur. Les messages de debug sont centralisés mais peuvent être séparés en section (afin que certains modules, comme les parseurs, qui produisent beaucoup de données, ne polluent pas les informations importantes). Elles peuvent être dirigées vers un fichier, vers l'écran, ou vers un mécanisme de sortie quelconque. Un exemple de résultat de ces utilitaires est en Annexe B.
- `COptionsParser & CCommandlineParser` : Ces deux parseurs permettent d'avoir des fichiers de configurations extrêmement simple d'utilisation, ainsi qu'un ligne de commande facile à utiliser. Des exemple de fichiers de configuration sont donnés en Annexe A. Comme on peut le voir, le typage des données et automatique, ce qui permet de ne pas à avoir à s'en soucier, et ouvre la configuration à des personnes néophytes. Cela permet d'experimenter facilement de nouveaux paramètres, et de gérer d'autres modèles aisément. Cela est d'importance car il est nécessaire, afin de pouvoir « benchmarker » différents modèles, d'avoir une architecture souple et flexible.
- `CTimer` : Cette classe importante, puisqu'elle implémente un timer haute résolution multiplateforme. Dans le cas de benchmark ou de mesure de pas de temps, une précision suffisante est nécessaire. Cette classe donne un moyen simple et efficace d'y parvenir. Voici un exemple d'utilisation de la classe.

```
void func()
{
    CTimer  myTimer;
    myTimer.Start();
    ...
    ...
    ...
    while()
    {
    double myTime = myTimer.GetElapsedTime();
    ....
```

```
        }
    }
```

Ce timer permet d'effectuer des mesures haute précisions sur les performances des différents algorithmes utilisés, et donc de discriminer entre les algorithmes performants et ceux qui ne le sont pas.
Pour l'application Haptics (le projet), les résultats sont les suivants :

Tâche	Temps pour 300 frames	Temps en %
Calculs de matrices	0.62	78.1 %
Calculs de SWIFT++	0.11	14.0 %
Résolution de LCP	0.01	1.2 %
Résolution des contacts	0.05	6.7%

FIG. 3.2 – Temps pour le projet Haptics

On voit que les calculs de matrices, non optimisés, prennent 80 % du temps. Il est donc inutile d'essayer d'optimiser les autres calculs tant que ces calculs de matrices n'ont pas été optimisés.

3.1.5 Section mathématiques (`maths`)

Cette section contient des fichiers permettant de manipuler simplement en C++ des matrices, des vecteurs et des quaternions. Une partie de ces fichiers ont été repris de sources internet libres (GPL). Il est prévu de changer cette section par la compilation de Blitz++, et par l'ajout de fonctionnalités à cette distribution. Cela résultera à n'en pas douter en un gain de performance non négligeable, car de nombreuses opérations matricielles sont nécessaires dans les algorithmes envisagés.

3.1.6 Section scène basique (`base`)

Cette section à pour but de mettre en place une scène basique, et la classe principale dérive de la classe `CCameraScene` afin de gérer la caméra. En s'appuyant sur cette classe, il est très facile d'implémenter des tests en personnalisant quelques fonctions bien définies, ou bien d'essayer des preuves de concept. C'est cette classe qui est actuellement utilisée pour la mise en place de l'algorithme de Sauer et Schömer, et sera transformée en une autre classe cette implémentation finalisée.

3.1.7 Section entrées (`input`)

Cette section contient plusieurs modules, qui interfacent les différentes moyens d'entrées. Compte tenu de la disparité des moyens d'entrée, il a été choisi de ne pas utiliser l'héritage dans

FIG. 3.3 – Différents moyens d'entrées utilisateurs

cette section spécifique. En effet, chaque moyen d'entrée (clavier, souris, PHANTOM, souris 6D) possède des caractéristiques propres que l'on ne retrouvent pas chez les autres. Si l'on envisage un moyen unifié d'accéder à ces entrées utilisateurs, il est nécessaire de sacrifier des fonctionnalités avancées, ce qui n'est pas souhaitable. Chaque classe (ou module) contient donc une interface pour un moyen d'entrée. Pour l'instant, le clavier et la souris sont complètement gérés grâce à une encapsulation de la `GLUT`. Il est aussi possible d'utiliser la `GLUI`, même si l'intérêt est limité au vu de la section interface utilisateur.

3.1.8 Section entrées 6d (`6dinput`)

Cette section est nécessaire à l'intégration de l'opérateur dans le simulation. Elle suit les mêmes principes de design C++ que le reste du framework. Deux points importants pour cette section sont l'interopérabilité (ie on doit pouvoir utiliser de nombreux périphériques aux diverses fonctions, sans changer le reste du code) ainsi que la résistance aux erreurs (le fait de ne pas trouver le périphérique ne doit pas être fatal). Une classe mère regroupe les fonctionnalités communes à tous les périphériques ainsi que proposant un mécanisme d'options de fonctionnalités. Voici la structure de la classe `C6DInput` (les commentaires de cette classe ont été retirés par soucis de concision) :

```
class C6DInput
{
public:
    virtual bool           InitDevice(){return true;}
    virtual void           ShutdownDevice(){}
    virtual bool           IsDeviceReady(){return true;}
    virtual unsigned int   GetCaps(){return INPUT_6D_NO_CAPS;}
    virtual const char  *  GetIdentifier() const {return "not_defined";}
    virtual int            GetNumberOfButton(){return 0;}
    virtual unsigned int   GetButtonState(const int _iButtonNumber)
```

```
                                                {return INPUT_6D_BUTTON_UP;}
    virtual S6DInputVector  GetSpeed(){return S6DInputVector(0,0,0,0,0,0);}
    virtual S6DInputVector  GetAcceleration(){return S6DInputVector(0,0,0,0,0,0);}
    virtual void            SetForces(S6DInputVector    _s6divForces){}
    virtual void            SetPoint(S6DInputVector     _s6divPoint){}
    virtual void            StartFeedBack(){}
    virtual void            StopFeedBack(){}
};
```

Les possibles extensions sont les suivantes :

```
    ///The device has no caps
    const unsigned int  INPUT_6D_NO_CAPS                = 0;
    ///The device supports position
    const unsigned int  INPUT_6D_SUPPORT_POSITION       = 1;
    ///The device supports speed
    const unsigned int  INPUT_6D_SUPPORT_SPEED          = 2;
    ///The device supports acceleration
    const unsigned int  INPUT_6D_SUPPORT_ACCELERATION   = 4;
    ///The device supports force feedback
    const unsigned int  INPUT_6D_SUPPORT_FEEDBACK       = 8;
    ///The device supports point feedback
    const unsigned int  INPUT_6D_SUPPORT_POINT_FEEDBACK = 16;
```

Enfin, voici le format des informations renvoyées par les différentes fonctions d'état des dispositifs :

```
template<typename Typ>
struct  S6DInputVector_Typ
{
    S6DInputVector_Typ(){}
    S6DInputVector_Typ(Typ _x,Typ _y, Typ _z, Typ _k, Typ _l, Typ _m)
                        {x=_x;y=_y;z=_z;k=_k;l=_l;m=_m;}
    Typ x,y,z;  /// Holds a position
    Typ k,l,m;  /// Holds a rotation
};

typedef S6DInputVector_Typ<double> S6DInputVector;
```

Les trois premiers champs de cette structure contiennent une position, et les trois suivants contiennent une rotation grâce aux angles (cette rotation correspond à une rotation physique de l'objet, les problèmes de Gimbal lock et autres ne se posent pas).

Chaque dispositif ne dispose pas forcément de toutes les capacités ou de toutes les fonctionnalités offertes par `C6DInput`. C'est pour cela que la méthode `GetCaps()` existe. Elle permet aux fonctions qui utilisent cette classe, de savoir si telle ou telle fonctionnalité est disponible, et le cas échéant, l'utiliser. Par exemple, la majorité des dispositifs permettent d'avoir la position, mais pas forcément la vitesse. Si la vitesse est disponible, elle est utilisée, sinon le programme appelant peut tenter de déduire la vitesse de la différence de position (cela donnera un résultat moins précis) afin d'avoir un comportement consistant. Outre la méthode `GetCaps()`, on notera dans l'interface de `C6DInput` la méthode `GetIdentifier()`. Cette méthode permet de faire de la RTTI[4] basique. Si le programme appelant sait gérer de manière spécifique un certain type de dispositif, il peut vérifier que c'est bien ce dispositif qui est relié à l'ordinateur. Cela peut être utile pour les tests, le calibrage, ou pour effectuer une action spécifique au dispositif d'entrée mais néanmoins primordiale à son fonctionnement (étant entendu que cette action ne peut pas être encapsulée dans une des fonctions existante, la fonction d'initialisation par exemple).

Cette classe est dérivée en trois classes dans l'implémentation actuelle :

- `C6DKeyboardInput` : Cette classe émule un dispositif 6D au clavier. Au moins 13 touches libres sont nécessaires afin d'obtenir 6 degrés de libertés et un bouton. Cela permet d'effectuer des tests, même lorsqu'aucun autre dispositif n'est pas présent sur le système. Le gros inconvénient de cette émulation est que le contrôle d'un objet en 6 dimensions est complètement contre-intuitif au clavier. Son utilisation est donc réservée pour des cas sporadiques. Il est à noter qu'avec 12 boutons, on n'obtient qu'un contrôle en position (ou en vitesse), et que 12 boutons supplémentaires sont nécessaires afin d'avoir la position et la vitesse simultanément.
- `CSpaceBallInput` : Cette classe implémente un contrôleur basée sur la spaceball de 3DConnexion. Le modèle utilisé pour les tests est le 4000 FLX, mais n'importe quel modèle de 3DConnexion est compatible avec cette interface. Comme 3DConnexion ne propose pas d'API en mode « poll » , c'est-à-dire une interface dont on peut demander l'état, mais ne propose qu'une solution basée sur le « push » , c'est à dire qu'à chaque fois qu'un événement est généré par la spaceball, il est transmis à l'application. De plus, la spaceball requiert une fenêtre Windows® pour envoyer ses messages. Toutes ces contraintes ont conduit à l'écriture d'un *wrapper* pour la spaceball. Ce wrapper ne sera pas visible dans les headers de CSpaceBallInput, mais uniquement dans son implémentation, il ne viendra donc pas polluer le code. Son rôle est double : premièrement, initialiser la spaceball et une fenêtre en relation avec celle-ci. Cette fenêtre, du au mécanismes inhérents de Windows®, tournera dans un thread séparé (mais dans le même processus). Afin de ne pas polluer l'interface utilisateur, et vu que cette fenêtre ne sert que de récepteur, elle est masqué et rendu invisible dans les tâches du système d'exploitation. Son deuxième rôle est de recevoir les messages de la spaceball, de les filtrer afin de ne garder que l'information

[4] **R**un-**T**ime **T**ype **I**nformation : permet de connaître *au moment de l'éxécution* le type d'une variable. En général, avoir la RTTI sur tout un programme le ralenti de manière notable.

intéressante, et enfin de traduire cette information dans un format compréhensible par le reste de l'application (dans le cas présent, ce format est `S6DInputVector`). Le rôle de la classe `CSpaceBallInput` est de fournir un support de ces fonctions au travers de l'interface unifiée de `C6DInput`. Il faut aussi noter qu'il n'est pas possible d'utiliser directement le *wrapper*, car celui-ci est forcément statique (Windows® ne supporte pas les messages vers des objets non statiques).

– `CPhantomInput` : Cette classe est chargée de gérer le dispositif `Phantom Desktop`®. Ce dispositif gère le retour d'effort, et donc son implémentation est la plus complexe. En effet, une boucle haptique typique doit tourner à 1000Hz, et une boucle de programme vers 100Hz (notamment à cause des calculs de simulation). Il est donc nécessaire de découpler les deux boucles. Néanmoins, un délai existera entre la boucle à 100Hz et la boucle à 1000Hz, comme montré en Figure 3.4.

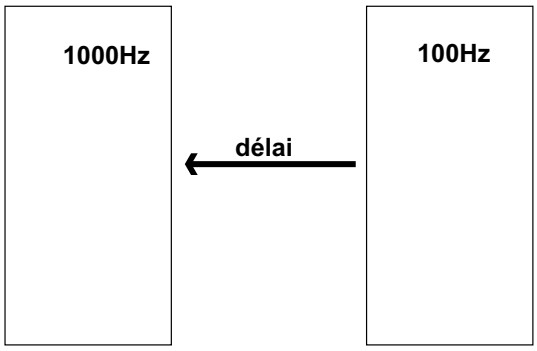


FIG. 3.4 – Un délai est inévitable entre deux boucles tournant à des fréquences différentes

Le but de la classe `CPhantomInput` est, en partie, de lutter contre ce délai (en lissant les entrées par exemple, ou en faisant de l'extrapolation). La classe sert d'entrée/sortie vers les fonctions de GHOST[5]. Elle permet d'avoir la position, la vitesse, ainsi que de faire du feedback en force et en position. Le feedback en position, si est plus complexe, est incomparable au niveau du rendu (très bon rendu) comparé au feedback en force. Une classe est chargée de gérer les données envoyées au PHANTOM, comme préconisé par GHOST. La classe `gstEffect`, qui fournie dans GHOST, est sous-classée afin de pouvoir effectuer le rendu haptique désiré. Le rendu haptique lui même est discuté dans la section suivante. La classe `gstForceEffect` surcharge la méthode `calcEffectForce`, qui est chargée de renvoyer la force actuelle au matériel. Cette classe possède de nombreux paramètres inhérents au modèle mis en place, son fichier de configuration est donné en Annexe A.4. Une instance de `gstForceEffect` est crée au départ par `CPhantomInput`, puis les requêtes lui sont suivies.

[5]GHOST est le Software Developpement Kit fourni par Sensable, les créateurs du `Phantom Desktop`®

3.1.9 Section interface utilisateur (`gui`)

Cette section est chargée de gérer une GUI qui soit directement intégrée à la fenêtre de simulation et de visualisation. Le but de cette GUI est de permettre d'avoir simplement des informations sur les variables de la simulation, d'afficher des messages, et, à terme, de modifier les paramètres en temps réel. Une des raisons pour lesquelles la `GLUI` n'a pas été utilisée est le fait qu'elle n'est pas facilement extensible, et ne propose que des moyens statiques d'afficher l'information (pas de MessageBox[6]...). De plus, l'implémentation de la `GLUI` repose sur `GLUT`, et donc oblige à utiliser `GLUT`, même lorsque cela n'est pas désiré. Une photo d'écran de la gui en action est donnée en Annexe D.1.

3.1.10 Section simulation (`simul`)

Cette section est chargée de gérer la simulation. Elle est composée de deux modules principaux.

3.1.11 Module `CSimulationManager`

Ce module est chargé de gérer, coordonner et diversifier les différentes parties d'une simulation physique. Il gère les objets de la simulation dans un conteneur de type arbre binaire balancé, pour pouvoir accéder très rapidement aux objets grâce à un numéro d'identification qui leur est donnée à l'initialisation.

Au départ, ce module charge les différents paramètres de la simulation dans un fichier de configuration. Ce fichier de configuration contient, entre autres, des paramètres physiques communs à tous les objets (gravité, forces globales...), ainsi que la liste des objets de la simulation. Ces objets sont ensuite chargés et mis en mémoire dans le conteneur. Une fois cette opération réalisée, il est virtuellement possible d'effectuer n'importe quelle action dans n'importe quel ordre sur les objets, et de manière très simple. Ainsi, l'interfaçage avec la détection de collision fournie dans la distribution de SWIFT++ a été presque immédiat, et l'exploitation des résultats est très facile. Comme ce module collabore étroitement avec le module `CSimulationObject`, il lui est donné accès aux variables membres protégées de celui-ci, afin de ne pas pénaliser l'application avec des accesseurs inutiles dans ce cas précis.

Le but final est d'obtenir un module suffisamment générique et complet, pour qu'il soit possible d'implémenter facilement et rapidement différentes approches pour le prototypage virtuel. Une approche classique au problème du prototypage virtuel est utilisé dans un premier temps :

1. Initialisation des objets, chargement des paramètres.
2. Calcul des collisions.
3. Réponses aux collisions.
4. Mise à jour des positions.
5. Affichage de la simulation.

[6] Boîte de message

6. Retour à l'étape 2.

La classe est conçue de manière à pouvoir travailler en temps réel ou en temps virtuel. Cela permet de « ralentir » la simulation aux moments cruciaux et de voir si les équations se comportent bien (problèmes de précisions...). Cela permet aussi d'accélérer la simulation et de voir si des divergences apparaissent. Une application immédiate de cela a été d'observer que les problèmes de précisions avec les quaternions les faisaient diverger en ne les rendant plus orthogonaux[7]. Cela permet aussi de simuler un environnement aux faibles capacités de calcul, et d'émettre des avis sur le nombre minimum de frames par secondes nécessaires à une simulation « normale » .

3.1.12 Module `CSimulationObject`

Ce module implémente un objet de la simulation. Un objet de simulation est une classe qui sert surtout à contenir les attributs de l'objet et à effectuer des calculs basiques sur celui-ci. Toutefois, son rôle se limite à cela, ce n'est pas cette classe qui calculera la simulation, cela est le rôle des algorithmes utilisés pas le `CSimulationManager`. Un exemple de calcul implémenté dans cette classe est le passage d'une matrice de transformation globale (utilisable par OpenGL, par SWIFT++), à un quaternion et une translation, qui eux sont utilisables par les algorithmes de la simulation.

Voici l'interface de cette classe.

```
class CSimulationObject
{
///Our old friend, the simulation...
friend class CSimulationManager;

public:

    /**
     *  Constructor.
     *  \param _iSimulId    The simulation id.
     *  It must be unique for every object in a scene
     */
    CSimulationObject(int _iSimulId);

    /**
     *  This is an alias (for now) for the Load function.
     *  This may change in the future, moreover, the
     *  Init behaviour is more standardized.
     */
```

[7] C'est à dire qu'il ne correspondent plus à une matrice de rotation.

```
    bool            Init(const char * _strConfigFile)
    {
        return Load(_strConfigFile);
    }

    /**
     *  Loads the object from a configuration file
     */
    bool            Load(const char * _strConfigFile);

    ///Getting the simulation id
    int             GetSimulId();
    ///Getting the Swift Id
    int             GetSwiftId();

    ///Render the object with opengl
    void            Render();

    ///Calculates the matrix transformation from quaternion + translation
    void            CalculateMatrix();

    ///Calculates the inertia matrix from the transformation
    void            CalculateInertia();

    ///Sets accelerations & speeds to 0 if fixed.
    void            HandleFixed();

    ///Add a global force.
    //Global forces are always applied to the center of mass
    void            AddGlobalForce(Vector3f _v3fForce);

    /**
     *  Attach a controller.
     *  \param _pc6diController The controller to be attached.
     *  Pass NULL to detach controller
     */
    void            AttachController(C6DInput * _pc6diController);

protected:

    ///The Inertia matrix
    Matrix          m_matInertia;
```

```
///The world inertia matrix
Matrix          m_matWorldInertia;

///The inverse of the world inertia matrix
Matrix          m_matWorldInvInertia;

///Is the inertia matrix null ?
bool            m_bIsInertiaInfinite;

///The mass
double          m_dMass;

///The global tranformation matrix
Matrix          m_matTransformation;
///The matrix representing a rotation, used in some calculus
Matrix          m_matTransformationRotation;
///The quaternion representing the rotation
Quaternion      m_quatRotation;
///The translation
Vector          m_vecTranslation;

///The rotation speed quaternion
Quaternion      m_quatRotationSpeed;
///The speed vector
Vector          m_vecTranslationSpeed;

///The rotation acceleration
Quaternion      m_quatRotationAccel;
///The acceleration vector
Vector          m_vecTranslationAccel;

///The forces
Vector          m_vecForces;

///The torques
Vector          m_vecTorques;

///The 3DS object
C3DSObject      m_3cdsObject;

///The scene in which the object is contained
SWIFT_Scene     m_swsScene;
```

```
///The SWIFT id
int             m_iSwiftId;

///The Simulation Id
int             m_iSimulId;

/**
 *  Tells if the object is fixed is space.
 *  If true, accelerations & speeds are set to 0
 */
bool            m_bIsFixed;

/**
 *  Tells if the object is controlled by the user
 */
bool            m_bUserControlled;

/**
 *  Tells what type of controlled is to be used
 */
std::string     m_strControlledType;

/**
 *  Tells if the object is subject to gravity
 */
bool            m_bGravityControlled;

/**
 *  Tells if the object is in collision
 */
bool            m_bIsInContact;
bool            m_bWasInContact;
bool            m_bIsInCollision;

/**
 *  The controller
 */
C6DInput    *   m_pc6diController;

///The color (when in plain mode)
Vector3f        m_v3fColor;

///The object filename (file that contains raw vertex data)
```

```
    std::string     m_strObjFileName;

    static bool     s_bVerbose;
};
```

La classe contient, outre les variables physiques, des variables d'état (comme `bIsInContact`, qui permettent à la simulation de faire des suppositions sur cet objet. De plus, la classe permet d'attacher un contrôleur à un objet, contrôleur qui va spécifier un comportement à l'objet. La fait que le contrôleur soit identifié par une chaîne de caractère permet à la simulation de connaître le type de contrôleur attaché, elle peut de plus vérifier si cette information est concordante avec ce qui est reporté par l'objet `C6DInput`. Bien entendu, des fonctionnalités peuvent être rajoutées facilement à cette classe afin de lui permettre de gérer plus d'information. Au moment de l'écriture, une extension aux objet séparés de `SWIFT++` est en cours de réalisation par Yann CHENU.

3.1.13 Conclusion

Comme on a pu le voir lors des paragraphes précédents, l'objectif d'avoir la possibilité d'avoir de nombreux modèles de frottements et de gestion de l'opérateur, dans un optique de mesure, est pratiquement atteint par le framework mis en place. Ses points forts sont une très grande modularité, qui permet de rajouter et de sous-classer de nombreuses fonctionnalités, ainsi que des outils de mesure très précis. De plus, sa haute performance graphique permet d'espérer des bons framerates[8].

[8] framerate est un équivalent de frames par secondes.

3.2 Approche envisagée au problème

3.2.1 Une approche globale par contraintes

Comme la section 2.2.7 l'a montré, de nombreuses approches existent quant à la réponse aux interpénétrations. Les approches locales ne sont pas physiquement exactes, de plus elles posent de gros problèmes de stabilité. Le réalisme physique étant primordial, c'est une solution de résolution globale qui a été choisie. Ensuite, l'approche par contraintes possède l'avantage d'être la plus proche possible du monde réel, où ce sont réellement les contraintes des objets les uns sur les autres qui modifient le mouvement. Une approche se basant sur les contraintes mais ne respectant pas le réalisme physique, comme les nouvelles solutions basées sur l'optimisation (ces simulations considèrent que l'on peut sacrifier le réalisme à la vitesse, et sont surtout utilisées dans la génération d'images animées)[Gal02], n'est pas non plus acceptable. L'approche envisagée par Sauer et Schömer possède l'avantage de poser le problème en termes simples. Toutefois, les calculs matriciels sous entendus par cette méthode, s'ils ne sont pas optimisés, prennent beaucoup de temps de calcul.

3.2.2 Approche envisagée par Sauer et Schömer

Dans cette section, le modèle de résolution des forces au contact présenté est celui de Sauer et Schömer, tel qu'il apparaît dans [SS98]. Ce modèle présente l'avantage d'être très stable, *dans le cas où les contacts sont bien déterminés*.Ce dernier point à son importance, comme expliqué en section XXXXXXXXXXX.

Dans un premier temps, on se consacre au modèle sans frottements. Par soucis de clarté, les notations par rapport à l'article originel ont été modifiées. Afin de bien différencier les variables dans le repère du monde et les variables dans le repère de l'objet, les variables dans le repère de l'objet sont soulignées, par exemple $\underline{\omega}$. La relation entre $\underline{\omega}$ et ω est donnée par : $\omega = \mathbf{R}\underline{\omega}$, où $\mathbf{R}$ est la matrice de rotation du solide. Si la matrice d'inertie du solide dans son repère d'origine est $\mathbf{I}_0$, alors la matrice du solide dans le repère du monde vaut $\mathbf{I}(t) = \mathbf{R}\mathbf{I}_0\mathbf{R}^T$

On suppose que l'on a affaire à K solides qui sont en contacts en n points, un exemple est montré en Figure 3.5.

Les contacts sont indicés grâce aux indices des deux solides en contact, on considère toujours que la normale part de l'objet avec le plus petit index vers l'objet au plus grand index. On obtient un tableau qui est de la forme suivante :

k	i_k	j_k	$\mathbf{n}_k$	$\mathbf{c}_k$
1	1	3	.	.
2	1	3	.	.
3	2	3	.	.
4	1	4	.	.
5	2	4	.	.

$\mathbf{n}_k$ dénote la normale au contact, $\mathbf{c}_k$ est le point de contact. $\mathbf{r}_k$ est le vecteur qui part du centre de gravité de l'objet et qui arrive au point de contact (ce vecteur sert dans le calcul du moment

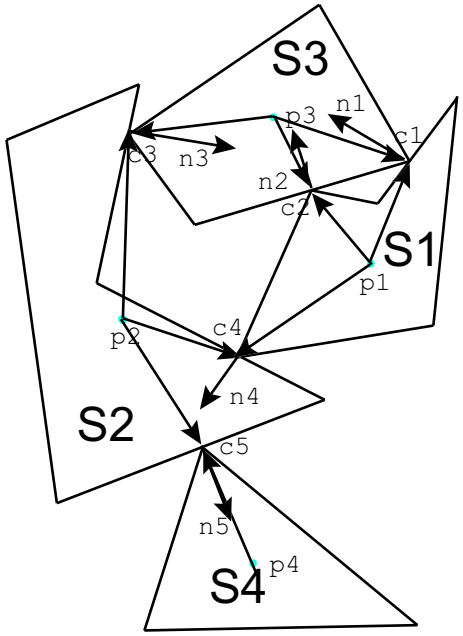


FIG. 3.5 – Exemple en deux dimensions, de solides en contact

du aux forces de contact). Les forces de contacts sont notées $\mathbf{f}_k = f_k \mathbf{n}_k$.

Les équations de Newton-Euler sont écrites de la manière suivante (on utilise le schéma d'intégration le plus simple, mais n'importe quel schéma d'intégration peut être utilisé) :

$$\begin{aligned}
\dot{\mathbf{p}}_l &= \mathbf{v}_l \\
\dot{\mathbf{q}}_l &= \frac{1}{2}\,\underline{\omega}_l \mathbf{q}_l \ (multiplication\ de\ quaternion), \\
\ddot{\mathbf{p}}_l &= m_l^{-1} \sum_{k|j_k=l} \mathbf{F}_k - m_l^{-1} \sum_{k|j_k=l} \mathbf{F}_k + \mathbf{g}, \\
\dot{\omega}_l &= \mathbf{I}_l^{-1} \sum_{k|j_k=l} \mathbf{r}_{kl} \times \mathbf{F}_k - \mathbf{I}_l^{-1} \sum_{k|j_k=l} \mathbf{r}_{kl} \times \mathbf{F}_k - \mathbf{I}_l^{-1} \omega_l \times \mathbf{I}_l \omega_l
\end{aligned}$$

Afin d'écrire les conditions de contacts sous forme matricielle, les notations suivantes sont utilisées.

Le vecteur vitesse généralisé $\mathbf{u} \in \mathbb{R}^{6n}$

$$\mathbf{u} = [\dot{\mathbf{p}}_1, \omega_1, \cdots, \dot{\mathbf{p}}_n, \omega_n]^T$$

Le vecteur de position et d'orientation généralisé $\mathbf{s} \in \mathbb{R}^{7n}$

$$\mathbf{s} = [\mathbf{p}_1, \mathbf{q}_1, \cdots, \mathbf{p}_n, \mathbf{q}_n]^T$$

Le vecteur exprimant la magnitude des forces de contact $\mathbf{f} \in \mathbb{R}^K$

$$\mathbf{f} = [f_1, f_2, \cdots, f_K]^T$$

Le vecteur des forces externes $\mathbf{f}_{ext} \in \mathbb{R}^{6n}$

$$\mathbf{f}_{ext} = [m_1\mathbf{g}, -\omega_1 \times \mathbf{I}_1\omega_1, \cdots, m_n\mathbf{g}, -\omega_n \times \mathbf{I}_n\omega_n]^T$$

La matrice $\mathbf{S} \in \mathbb{R}^{7n\times 6n}$

$$\mathbf{S} = \begin{bmatrix} \mathbf{E} & & & & \mathbf{0} \\ & \mathbf{Q}_1 & & & \\ & & \ddots & & \\ & & & \mathbf{E} & \\ \mathbf{0} & & & & \mathbf{Q}_n \end{bmatrix}$$

avec $\mathbf{q}_l := [q_{l0}, \cdots, q_{l3}]^T \in \mathbb{R}^4$ et $\mathbf{Q}_l \in \mathbb{R}^{4\times 3}$

$$\mathbf{Q}_l = \frac{1}{2}\begin{bmatrix} -q_{l1} & -q_{l2} & -q_{l3} \\ q_{l0} & q_{l3} & -q_{l1} \\ -q_{l3} & q_{l0} & q_{l2} \\ q_{l2} & -q_{l1} & -q_{l0} \end{bmatrix}$$

$\mathbf{Q}_l$ est juste un artifice mathématique simplifiant les calculs. La matrice généralisée représentant les masses et les inerties $\mathbf{M} \in \mathbb{R}^{6n\times 6n}$

$$\mathbf{M} = \begin{bmatrix} m_1\mathbf{E} & & & & \mathbf{0} \\ & \mathbf{I}_1 & & & \\ & & \ddots & & \\ & & & m_n\mathbf{E} & \\ \mathbf{0} & & & & \mathbf{I}_n \end{bmatrix}$$

La matrice des normales aux contacts $\mathbf{N} \in \mathbb{R}^{3K\times K}$

$$\mathbf{N} = \begin{bmatrix} \mathbf{n}_1 & & & \mathbf{0} \\ & \mathbf{n}_2 & & \\ & & \ddots & \\ \mathbf{0} & & & \mathbf{n}_K \end{bmatrix}$$

Enfin, les conditions de contacts s'écrivent grâce à la matrice $\mathbf{J} \in R^{6n\times 3K}$

$$\mathbf{J}_{lk} = \begin{cases} -\mathbf{E} & \textit{pour } l = 2i_k - 1 \\ -\mathbf{r}^{\times}_{ki_k} & \textit{pour } l = 2i_k \\ \mathbf{E} & \textit{pour } l = 2j_k - 1 \\ \mathbf{r}^{\times}_{ki_k} & \textit{pour } l = 2j_k \\ \mathbf{0} & \textit{dans les autres cas} \end{cases}$$

$\mathbf{r}^{\times} \in R^{3\times 3}$ est la matrice anti-symétrique

$$\mathbf{r}^{\times} = \begin{bmatrix} 0 & -r_3 & r_2 \\ r_3 & 0 & -r_1 \\ -r_2 & r_1 & 0 \end{bmatrix}$$

On utilise aussi la matrice de projection $\mathbf{Pr}_k^T$ suivante, qui permet d'extraire le k^{eme} contact

$$\mathbf{Pr}_k^T = [\mathbf{0}\cdots\mathbf{0} \quad \underset{\substack{k \\ \downarrow}}{\mathbf{E}} \quad \mathbf{0}\cdots\mathbf{0}]$$

Les équation du mouvement d'Euler deviennent alors

$$\begin{aligned} \dot{\mathbf{s}} &= \mathbf{Su} \\ \dot{\mathbf{u}} &= \mathbf{M}^{-1}(\mathbf{JNf} + \mathbf{f}_{ext}) \end{aligned}$$

$$\begin{aligned} \Rightarrow \mathbf{s}^{t+\Delta t} &= \mathbf{s}^t + \Delta t \mathbf{S}\mathbf{u}^{t+\Delta t} \\ \mathbf{u}^{t+\Delta t} &= \mathbf{u}^t + \Delta t \mathbf{M}^{-1}(\mathbf{JNf} + \mathbf{f}_{ext}) \end{aligned}$$

La composante normale de la vitesse relative des deux objets au point de contact k est exprimée par

$$\begin{aligned} \mathbf{n}_k^T \mathbf{P}_k^T \mathbf{J}^T \mathbf{u} &= \mathbf{n}_k^T (\dot{\mathbf{p}}_{jk} + \omega_{jk} \times \mathbf{r}_{kj_k}) \\ &- \mathbf{n}_k^T (\dot{\mathbf{p}}_{ik} + \omega_{ik} \times \mathbf{r}_{ki_k}) \end{aligned}$$

La condition de contact s'écrit alors

$$\mathbf{n}_k^T \mathbf{P}_k^T \mathbf{J}^T \mathbf{u}^{t+\Delta t} \geq 0 \ \perp \ f_k \geq 0$$

En faisant la somme, on en déduit

$$\mathbf{N}^T \mathbf{J}^T \mathbf{M}^{-1} \mathbf{JN} \Delta t \mathbf{f} + \mathbf{N}^T \mathbf{J}^T (\mathbf{u}^t + \Delta t \mathbf{M}^{-1} \mathbf{f}_{ext}) \geq 0$$

Ceci est un LCP de la forme

$$\mathbf{Ax} + \mathbf{b} \geq 0 \ \perp \mathbf{x} \geq 0$$

avec $\mathbf{A} \in \mathbb{R}^{K\times K}$ et $\mathbf{b}, \mathbf{x} \in \mathbb{R}^K$.
La matrice $\mathbf{A}$ s'écrit de la manière suivante,

$$\mathbf{A} = \mathbf{N}^T \mathbf{J}^T \mathbf{M}^{-1} \mathbf{JN}$$

ses éléments sont

$$\begin{aligned}
\mathbf{A}_{lk} &= \delta_{i_l i_k}\mathbf{n}_l^T(\frac{1}{m_{i_k}}\mathbf{E}-\mathbf{r}_{li_l}^{\times}\mathbf{I}i_k{}^{-1}\mathbf{r}_{ki_k}^{\times})\mathbf{n}_k \\
&- \delta_{i_l j_k}\mathbf{n}_l^T(\frac{1}{m_{j_k}}\mathbf{E}-\mathbf{r}_{li_l}^{\times}\mathbf{I}j_k{}^{-1}\mathbf{r}_{kj_k}^{\times})\mathbf{n}_k \\
&- \delta_{j_l i_k}\mathbf{n}_l^T(\frac{1}{m_{i_k}}\mathbf{E}-\mathbf{r}_{lj_l}^{\times}\mathbf{I}i_k{}^{-1}\mathbf{r}_{ki_k}^{\times})\mathbf{n}_k \\
&- \delta_{j_l j_k}\mathbf{n}_l^T(\frac{1}{m_{j_k}}\mathbf{E}-\mathbf{r}_{lj_l}^{\times}\mathbf{I}j_k{}^{-1}\mathbf{r}_{kj_k}^{\times})\mathbf{n}_k
\end{aligned}$$

Une fois le LCP écrit, il ne reste plus qu'à le résoudre.

3.2.3 Résolution du LCP

Un LCP se pose sous la forme suivante[Mur97] :
Donné $\mathbf{M} \in \mathbb{R}^{n\times n}$ et $\mathbf{q} \in \mathbb{R}^n$, trouver $\mathbf{w} = (w1, ..., w_n)^T, \mathbf{z} = (z_1, ..., z_n)^t$ tels que

$$\mathbf{w} - \mathbf{M}\mathbf{z} = \mathbf{q}$$

$$\mathbf{w} \geqslant 0, \mathbf{z} \geqslant 0 \; et \; w_i z_i = 0 \; \forall i$$

Dans le cas présent, la matrice $\mathbf{M}$ est $\mathbf{A}$.

Deux méthodes ont été envisagées afin de résoudre le LCP obtenu par la méthode précédente : une méthode itérative et la méthode de Lemke.

La méthode itérative possède l'avantage de résoudre par approximations successives le LCP. Cette méthode ne marche que si la matrice est positive définie symétrique, ce qui est le cas si il n'y a pas de frottements. Il est donc possible de ne pas avoir la solution exacte, mais néanmoins une approximation suffisante pour la simulation. De plus, l'algorithme peut être interrompu si le temps processeur vient à manquer. Toutefois, cette méthode converge de plus en plus lentement dans certains cas. On peut montrer que le LCP est équivalent au problème du point fixe de trouver $x \in \mathbb{R}^n$ tel que

$$f(\mathbf{x}) = \mathbf{x}$$

avec $f(\mathbf{x}) = \mathbf{b} + \mathbf{B}|x|, \mathbf{b} = -(\mathbf{E}+\mathbf{M})^{-1}\mathbf{q}, \mathbf{B} = (\mathbf{E}+\mathbf{M})^{-1}(\mathbf{E}-\mathbf{M})$. Typiquement, le nombre d'itérations pour arriver à la solution est en

$$\frac{log\left(\frac{(1-\rho)}{(1+\sqrt{n})(1+\rho)}\right)}{log(\rho)}$$

où $\rho = ||\mathbf{B}||$. Si $\mathbf{A}$ devient grande (ie si les masses deviennent grande), alors

$$\mathbf{B} \simeq \mathbf{M}^{-1}\mathbf{M} = \mathbf{E}$$

et $\rho \simeq 1$. On voit bien dans ce cas que le nombre d'itérations devient grand. Ce phénomène a aussi été observé dans l'expérimentation. Le problème est trop handicapant pour garder une telle solution.

La deuxième solution envisagée est celle employée par Lemke, qui est un algorithme de type pivot. Cet algorithme est en temps fini (temps en relation avec la dimension de la matrice), et s'est révélé beaucoup plus rapide que l'algorithme itératif (de 10 à 100 fois plus rapide).Tous les détails concernant cet algorithme sont donnés dans [Mur97], pages 66-80.

3.2.4 Problèmes inhérents au modèle choisi

Le problème principal vient du fait que l'on ne résout, à une étape donnée, que les contacts *existants*. Le fait est qu'une réponse à un contact peut en créer un ou plusieurs autres auquels, en théorie, il faudrait une réponse. Dans le cadre du hors ligne, les pas de temps peuvent être aussi petits que l'on veut (toutefois, il faut bien noter que même dans le cas où la simulation s'effectue hors ligne, le temps de calcul n'est pas infini). Il se produit alors la situation illustrée par les Figures 3.6 et 3.7. La première force génère une rotation, mais comme le pas de temps est petit, à la deuxième itération l'objet est en resting contact.

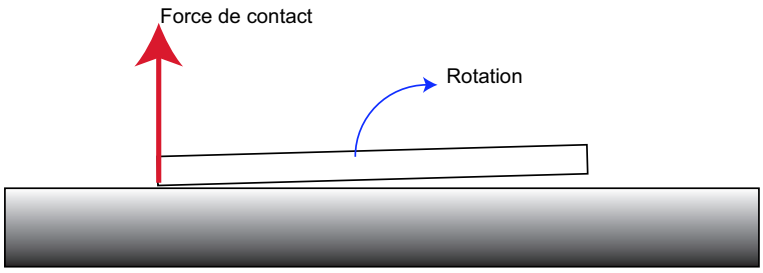


FIG. 3.6 – Une force est générée pour empêcher l'interpénétration

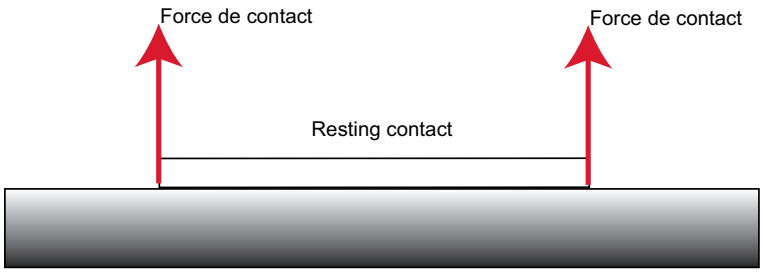


FIG. 3.7 – Resting contact - Les forces ne génèrent pas de rotations

Toutefois, en temps réel, le pas de temps peut devenir grand pour une raison quelconque. Il se produit alors la situation illustrée par la Figure 3.8. Comme le pas de temps est trop grand, le

solide à effectué une rotation telle que le solide entre en collision (et non plus en contact) avec le sol.

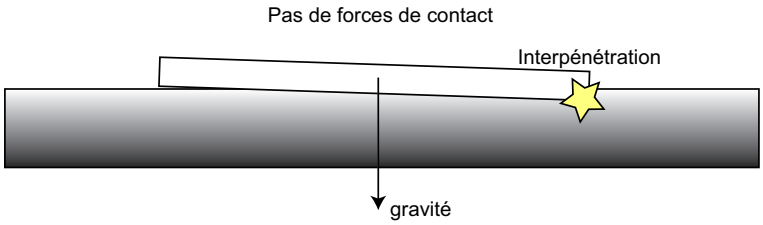


FIG. 3.8 – Interpénétration - Le solide « s'enfonce » dans le sol

La simulation a échouée dans son effort, et si les interpénétrations ne sont pas gérées (rappelons que le but est justement d'avoir un modèle physiquement exact), le solide s'enfonce dans le sol. Même si le deuxième contact est détecté, si il est le *seul* (ie le premier ne l'est plus, car le deuxième point est plus proche), alors la distance minimum du solide au plan du sol est une fonction *strictement* décroissante. Il y a donc fatalement interpénétration.

Pour palier à ce problème il faut dès le début considérer plusieurs points comme des points de contacts potentiels. C'est ce qui est réalisée avec l'approche implémentée. Malheureusement, le logiciel utilisé pour la détection de contact, `SWIFT++`, ne donne qu'un point de contact à chaque fois (même si l'interface de la fonction est censé en reporter plusieurs) pour les objets convexes. On observe donc un effet de clignotement très sensible, suivi d'une interpénétration des solides. En collaboration avec Yann CHENU, stagiaire en deuxième année à l'I.I.E., il a été tenté de résoudre ce problème, contact a aussi été pris avec Stephen A. Ehmann, dévelopeur originel du logiciel. Il n'existe, de son propre aveu, pas de solution simple, toutefois un léger succès a été obtenu en observant les plus proches voisins du point détecté, et en vérifiant que ceux-ci ne correspondaient pas aux conditions de contact.

Le problème de ne pas détecter les contacts s'étend aussi aux mouvements de translation, quand plusieurs objets sont en contacts. Le fait que le pas de temps peut devenir grand entraîne, dans certain cas, une interpénétration.

Une autre solution consiste donc à « gérer » les interpénétrations sporadiques, afin de revenir à un état normal lorsque le pas de temps devient trop grand. Cette solution demande une autre approche aux algorithmes de détection de collision, et prend un certain temps à mettre en œuvre, cela donc pas été implémenté dans le framework actuel.

3.3 Retour haptique

3.3.1 Pourquoi un retour haptique ?

L'approche originale de Sauer et Schömer n'est pas temps réel, et de plus n'intègre pas l'opérateur dans la boucle. En effet, si une simulation sans opérateur peut présenter un intérêt pour l'animation ou bien certaines simulations physiques, l'opérateur est nécessaire dans de nombreuses applications, notamment dans le prototypage virtuel. Il est connu que le fait d'intégrer un opérateur dans la boucle crée des instabilités, c'est donc pourquoi le modèle sans opérateur se doit d'être très stable. C'est pourquoi le modèle de Sauer et Schömer est très intéressant, car il travaille sur les vitesses et non pas les accélérations comme le faisait Baraff. Cela permet donc d'éviter un niveau d'intégration, et autant d'instabilité.

Le retour haptique permet à l'utilisateur de s'immerger dans le monde virtuel d'une manière beaucoup plus concrète, et lui permet de réaliser des tâches virtuelles d'assemblage, de démontage... Aujourd'hui, un retour haptique de qualité est désiré par de nombreux industriels. Le problème est qu'un retour haptique de qualité demande, en général, des pas de temps très petits en *temps réel*, mais qu'une simulation physique de qualité prends du temps de calcul.

Un retour haptique de qualité possède les propriétés suivantes :

- En mouvement libre, l'opérateur ne doit pas ressentir de forces.
- En contact, l'opérateur doit sentir de manière immédiate une résistance, plus l'opérateur « force » , plus la résistance doit être grande.
- Le dispositif ne doit pas vibrer, que cela soit à basse fréquence ou à haute fréquence.

Actuellement, un tel retour haptique « parfaitement transparent » , en relation avec des calculs physiques exacts, n'existe pas.

Plusieurs approches ont été envisagées afin d'intégrer cet opérateur dans la boucle, chacune a levée des problèmes différents. La dernière approche, innovante, corrige la plupart, voire dans certains cas tous, les problèmes posés.

Il est discuté dans les sections ci-dessous du mouvement libre et du mouvement en contact. Il peut sembler que les problèmes en mouvement libre peuvent être tous résolus en désactivant le retour d'effort quand il n'y a pas contact. Cela n'est cependant pas une une solution : le mouvement libre apparaît aussi lors du contact. En effet, dans le cas d'une sphère en contact avec un plan (un seul point de contact), il doit y avoir une contrainte suivant *la normale au plan*, mais pas suivant *le plan lui-même*. L'opérateur sentirait immédiatement un quelconque effet pernicieux. Un mouvement libre de qualité est donc le garant qu'en contact, seule la contrainte sera perçue par l'opérateur.

3.3.2 La boucle classique en haptique

En Figure 3.9 est donnée la boucle classique utilisée en haptique : la simulation reçoit des informations du contrôleur haptique, et, en échange, renvoie des demandes de forces ou de positions.

On voit tout de suite qu'il faut 10 ms à la simulation pour répondre à un changement de la part du dispositif haptique. Un délai intervient donc dans le retour des forces. Le fait que le dispositif

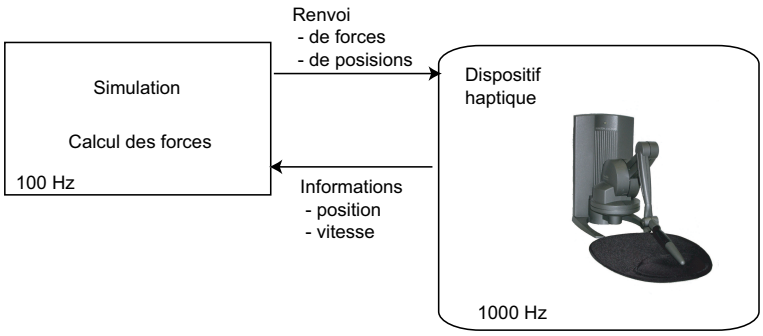


FIG. 3.9 – La boucle haptique

puisse avoir changé d'état entre temps n'est pas connu de la simulation.

Deux retours vers le dispositif haptique sont envisagés.

3.3.3 Retour en force

Le retour en force, s'il est simple à implémenter (il suffit de donner aux dispositif haptique les forces calculées par la simulation), propose une expérience utilisateur médiocre. En effet, les forces renvoyées par la simulations ne correspondent pas forcément aux dispositif, c'est à dire qu'une force peut être disproportionnée comparativement à la force exercée par l'utilisateur. Même avec un calibrage fin, les forces renvoyées ne correspondent quasiment jamais aux forces que l'utilisateur « s'attend » à recevoir. De plus, le délai existant rend les forces saccadées. Toutes les 10ms, la force change brusquement de sens, ce qui est tout à fait notable par un utilisateur.

Un lissage de forces, si il supprime le dernier problème, fait intervenir des vibrations de plus basse fréquence, et l'utilisateur à l'impression d'un « temps de retard » comparativement à la simulation. Des phénomènes de collage ou de répulsion peuvent aussi intervenir (un force est exercée alors que le contact n'existe pas).

Le retour en force est donc à éviter afin d'avoir un rendu haptique de qualité. Ce fait n'est pas surprenant, la majorité des solutions haptiques utilisant le retour en position présenté ci-dessous.

3.3.4 Retour en position

Cette approche, utilisée en autre par Baraff dans [BHB99], couple le dispositif haptique avec sa position dans la simulation grâce à une système masse-ressort. Certaines simulations rajoutent un terme de dissipation, mais dans le cas présent, cela n'est pas nécessaire car le système se régule de lui-même. Le retour en position fonctionne de la manière suivante (cf Figure 3.10) :

1. La simulation connaît la position que le dispositif haptique devrait avoir.
2. La simulation renvoie cette position à la boucle haptique.
3. La boucle haptique, à 1000 Hz, calcule une force sur le dispositif haptique par l'intermédiaire d'un système masse-ressort.

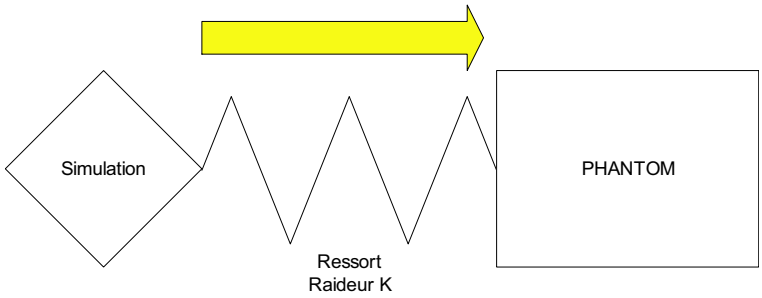


FIG. 3.10 – Un système masse ressort relie la position réelle du phantom à sa position virtuelle

Pour le dispositif haptique, c'est la position de simulation qui est *virtuelle*. La Figure 3.11 illustre ce serait le système dans la réalité : la position de la sphère bleue est fixée par un système externe (la simulation), et la sphère jaune le point final du système haptique. Une force est générée par le ressort (de longueur au repos 0) afin de ramener la sphère jaune sur la sphère bleue. C'est cette force qui est le *feedback*.

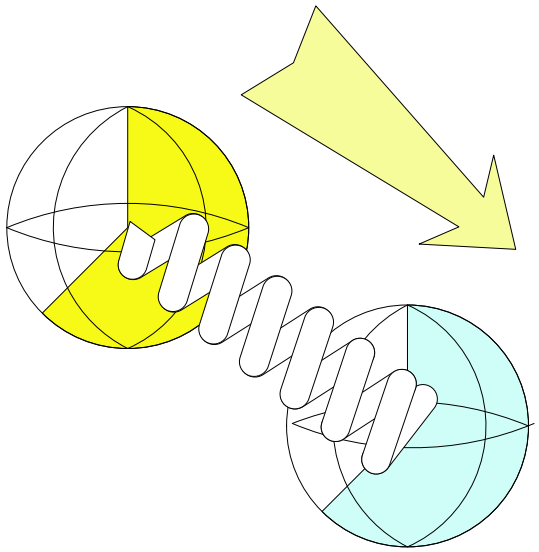


FIG. 3.11 – Couplage des sphères par ressort

Pour un ressort classique de longueur au repos 0, la formule de la force générée est :

$$\mathbf{f}_{Ressort} = \lambda(\mathbf{x}_{ref} - \mathbf{x}_{pos})$$

où $\mathbf{x}_ref$ est la position désirée et $\mathbf{x}_{pos}$ est la position actuelle. Ceci est modèle de ressort linéaire physique, toutefois il pose un problème de linéarité.

Le délai introduit par les différences de fréquences entre la boucle de simulation et la boucle haptique entraîne forcément un petite différence entre la position désirée (qui vient de la der-

nière itération de la boucle de simulation) et la position réelle, même en mouvement libre (car l'opérateur a déplacé le dispositif pendant les 10 ms de délai). Cela, avec un modèle classique de ressort, entraîne une force de « frottement » , qui ne devrait pas exister en mouvement libre.

Toutefois, si le ressort peut être utile pour l'analogie avec le monde réel, dans le monde virtuel rien ne contraint à ce que le modèle soit linéaire. Pourtant, de nombreuses publications ne remettent pas en cause ce modèle. Un nouveau modèle de ressort a donc été mis en place, qui écrase les petites différences, puis devient linéaire au bout d'un certain seuil.

La formule d'une telle fonction est la suivante (format Maple) :

```
ramp:=proc(x,th,coeflin,coefquad,alpha)
> if (abs(evalf(x))<=evalf(th)) then
> return((abs(x)/th)^alpha*coeflin*x)
> else
> return(x*coeflin)
> fi;
> end proc;
```

Les paramètres d'une telle fonction sont les suivants :

- x La variable considérée.
- th Le seuil à partir duquel la fonction doit devenir linéaire[9].
- $coeflin$ Le coefficient linéaire, celui-ci correspond à la raideur du ressort.
- α La puissance d'écrasement.

Avec $\alpha = 6, th = 6, coeflin = 0.85$, on obtient une courbe comme celle de la Figure 3.12.

Ce type de ressort virtuel donne de très bon résultats, c'est à dire qu'il est capable d'effacer (dans une certaine mesure, si le pas de temps devient vraiment trop grand, il ne reste plus grand chose à faire) les effets du délai. Son implémentation et son paramétrage sont de plus très intuitifs. Il est certain qu'à l'avenir, on gagnera à considérer de nouveaux dispositifs physiques « virtuels » afin d'augmenter l'expérience haptique de l'opérateur. Ce type de ressort est un pas vers de nouvelles formules haptiques.

3.3.5 Contrôle en position (seule)

Le contrôle en position est en général celui qui est supporté par la majorité des dispositifs. Malheureusement, dans le cas présent, il n'est pas très intéressant seul. En effet, dans les simulations par contraintes, si un objet en interpénètre un autre directement, il n'y a pas de possibilité de corriger la situation. Cette difficulté est illustrée par les Figure 3.13 et 3.14. Dans la première, il y a contact, mais aucune force n'est générée (le solide est juste présent au contact, mais n'a pas l'intention de s'enfoncer), aucune force n'est générée. Pendant le délai de 10ms, l'opérateur à eu le temps de traverser l'autre solide. Une information de vitesse est donc nécessaire, et le contrôle en position seule est donc insuffisant.

[9] Seuil se dit « threshold » en anglais.

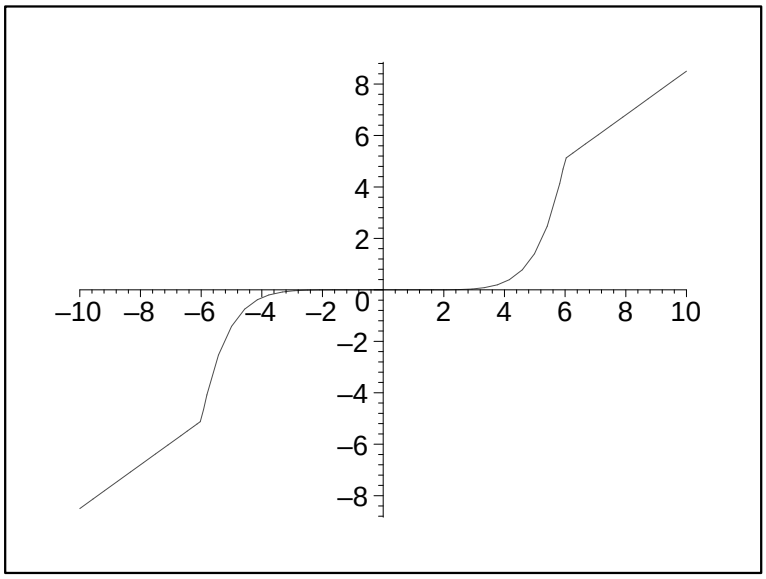


FIG. 3.12 – Une nouvelle sorte de rampe de ressort

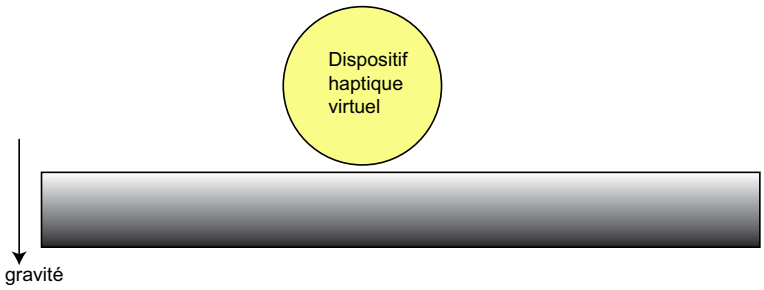


FIG. 3.13 – Contrôle en position - Pas de contact

3.3.6 Contrôle en vitesse (seule)

Afin de résoudre le problème d'interpénétration, un contrôle en vitesse a été envisagé. Cette fois ci, les positions sont synchronisées au début, puis la vitesse du dispositif haptique est appliquée au solide virtuel. Il se pose immédiatement un problème de délai et de divergence entre les deux positions. En effet, la retour d'effort va avoir tendance à corriger cette différence, mais pour ce faire, va imprimer une vitesse au dispositif haptique, et par là même l'éloigner aussi. Le figure 3.15 illustre ce propos. L'accumulation de l'erreur est très vite notée dans une implémentation, et il n'est pas possible de la corriger puisque l'on ne connaît que la vitesse à intervalles de 10ms. Il semble donc judicieux d'effecter un contrôle en position, afin d'éviter les divergences, et un contrôle en vitesse, afin d'implémenter un retour d'effort avant l'interpénétration.

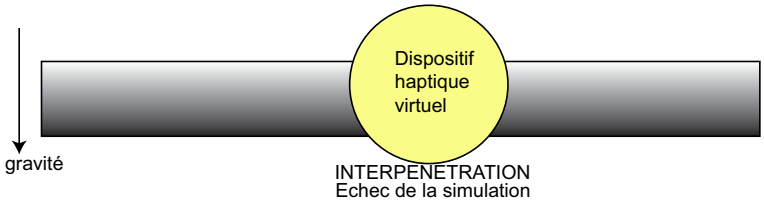


FIG. 3.14 – Contrôle en position - Interpénétration

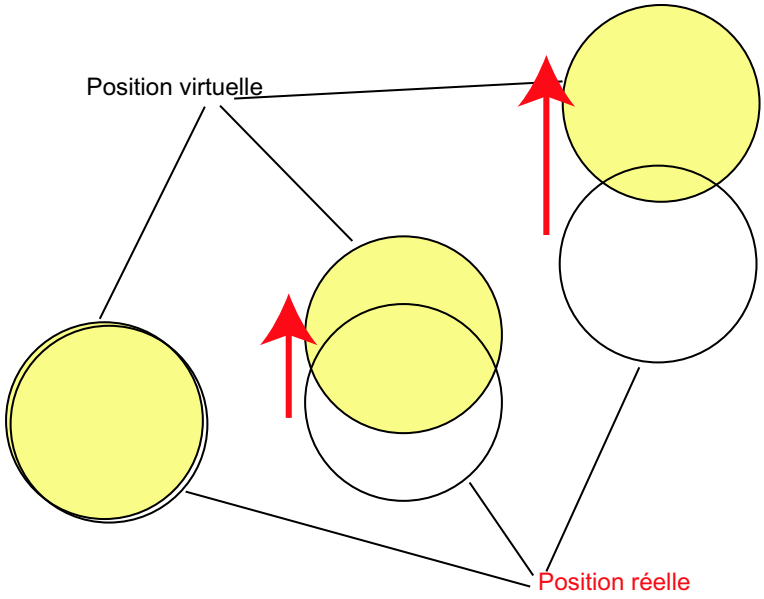


FIG. 3.15 – Contrôle en vitesse - Accumulation des erreurs

3.3.7 Contrôle en position et en vitesse

Un contrôle en position et en vitesse corrige les deux problèmes précédemment cités. En mouvement libre, l'adéquation presque totale entre la position virtuelle et la position réelle, couplée à la nouvelle rampe de ressort mise en place, permet de très bon résultats. De plus, lors du contact, l'opérateur ressent très nettement les contraintes. Toutefois, lorsque l'opérateur enlève son contrôleur du contact, il ressent un effet de répulsion qui ne cesse pas, même lorsqu'il est loin du contact. Ce phénomène, inadmissible, est expliqué par les Figures 3.16 et 3.17. Le fait est que lorsque le contact est établi, la vitesse du contrôleur s'annule du fait de l'effet des forces de contacts. Mais lorsque l'opérateur s'éloigne de la contrainte, la vitesse du contrôleur est aussi affectée, et donc la différence de position ne change pas. Il s'ensuit que le retour d'effort n'est pas stoppé.

On peut tenter de séparer les cas en contacts et en non contacts, mais comme précédemment dit, cette tentative est vouée à l'échec.

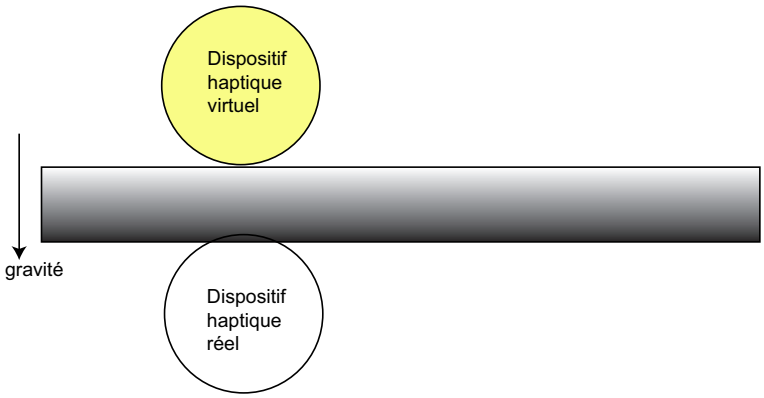


FIG. 3.16 – Contrôle en position et en vitesse - En contact, les vitesses sont nulles

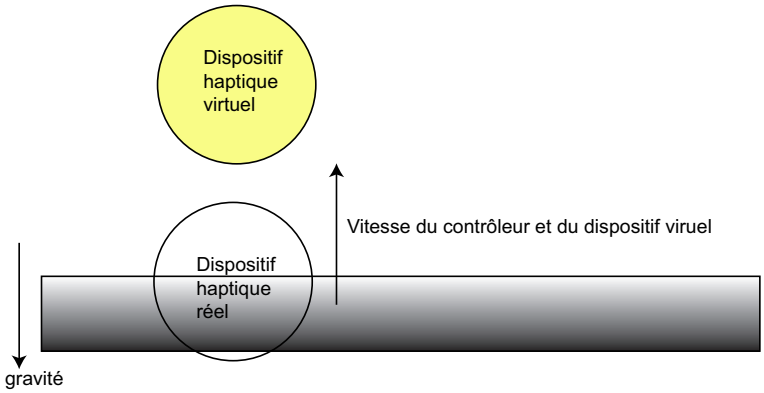


FIG. 3.17 – Contrôle en position et en vitesse - Les deux solides évoluent à la même vitesse, la différence de position ne change pas

3.3.8 Un nouveau type de contrôle : le contrôle par requête

Ces solutions, classiques et maintes fois utilisées, présentent chacune des défauts. Une nouvelle solution, basée sur le principe des requêtes, a été introduite. Le concept est non plus que le contrôleur physique va déterminer la vitesse de l'objet virtuel, où sa position, mais bien qu'il va *demander* à ce que la position soit telle. C'est l'analogie du retour haptique en sens inverse. La simulation va ensuite examiner si elle peut satisfaire à une telle requête, et si oui, dans quelle mesure et par quels moyens.

Dans la pratique, la simulation calcule la différence entre la position de l'objet virtuel et de l'objet réel, et calcule la vitesse que *devrait* avoir l'objet virtuel pour rejoindre l'objet réel. Cette vitesse vaut :

$$\mathbf{v} = \frac{\mathbf{p}_{virtuel} - \mathbf{p}_{reel}}{\Delta t}$$

En mouvement libre, rien ne vient changer cette vitesse, et donc à fin de la boucle de simulation la position virtuelle devient la position réelle. On obtient le même comportement qu'en contrôle

vitesse et position. En contact, la vitesse est contrebalancée par les forces de contact, le solide virtuel bouge moins que prévu. Il y a donc une différence de créée (comme sur la Figure 3.16), et l'utilisateur ressent un retour d'effort. De plus, plus l'utilisateur cherche à violer la contrainte, plus l'écart est grand, et plus le retour d'effort contre l'utilisateur. Quand l'utilisateur arrête d'appuyer, les efforts s'annulent tout simplement.

Ceci est exactement le comportement souhaité, et donne d'excellents résultats avec des tests physiologiques.

Ces contrôles par requêtes peuvent être étendus à d'autres applications de retour haptique, et de plus ne nécessite pas que la simulation tourne à une fréquence très élevée. Par exemple, des résultats très satisfaisants ont été obtenu avec une simulation tournant à 50Hz (et donc un calcul des forces tournant à la même vitesse), avec une dizaine d'objets de quelque centaines de polygones.

3.3.9 La nouvelle boucle de simulation

Voici la nouvelle boucle de simulation, écrite pour prendre en compte le retour haptique :

- Initialisation des positions, des vitesses et des accélérations des différents solides.
- Lecture des données du dispositif à retour d'effort.
- Calcul de la vitesse qu'aurait l'objet manipulé par l'opérateur en l'absence de contrainte.
- Mise à jour des différentes positions (les forces dues aux contacts sont prises en compte).
- Calcul des contacts.
- Calcul des forces dues aux contacts.
- La vitesse de l'objet manipulé est modifié en conséquence.
- Affichage des positions.

Chapitre 4

Conclusion

4.1 Modèle comportemental

Les modèles comportementaux sont abondants, toutefois seule une petite partie de ceux ci se concentrent sur une simulation physique en accord avec les lois classiques. Ces approches par contraintes sont celles qui ont été envisagées dans l'implémentation, en choisissant celle de Sauer et Schömer pour sa simplicité. Une étude des problèmes inhérents aux modèles utilisant les contraintes a aussi été réalisée

4.2 Framework

L'implémentation a prouvé sa modularité et son efficacité. La fait de pouvoir changer rapidement de modèle, ainsi que la modularité face aux dispositifs haptique, permettent un accroissement de la productivité dans les tests. Il reste à élargir ce framework, par exemple en proposant une interface unifiée aux algorithmes de détection de collision.

4.3 Retour haptique

Les expérimentations et les deductions dans le domaine du retour haptique ont été nombreuses. L'emergence d'un nouveau modèle de retour haptique, bi-contraint, a été l'aboutissement des travaux dans de domaine. Il reste néanmoins de nombreuses questions en suspens à résoudre.

4.4 Bilan

Ce stage a permit, outre une étude approfondie des solutions existantes, avec leurs forces et leurs faiblesses, de proposer des nouveaux modèles concernant le prototypage virtual accompagné de retour haptique. Cette proposition n'est pas restée au stade académique, et une implémentation effective de la solution proposée, accompagnée de tests, a été réalisée.

Bibliographie

[AH99] R. Adams and B. Hannaford. Stable haptic interaction with virtual environments. 1999.

[AKH00] R. Adams, D. Klowden, and B. Hannaford. Stable haptic interaction using the excalibur force display. 2000.

[Bar89] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies, siggraph. 1989.

[Bar91] David Baraff. Coping with friction for non-penetrating rigid body simulation. *Computer Graphics, Volume 23*, 1991.

[Bar94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies, siggraph. 1994.

[BB88] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constraints. *Computer Graphics, Volume 22*, 1988.

[BHB99] Peter J. Berkelman, Ralph L. Hollis, and David Baraff. Interaction with a realtime dynamic environment simulation using a magnetic levitation haptic interface device. 1999.

[BW92] David Baraff and Andrew Witkin. Dynamic simulation of non-penetrating flexible bodies. *Computer Graphics*, 26(2) :303–308, 1992.

[DM02] L. Dorst and S. Mann. Geometric algebra : A computational framework for geometrical aplications. 2002.

[DP00] Derek DiFilippo and Dinesh K. Pai. Contact interaction with integrated audio and haptics. 2000.

[Gal02] Oliver Galizzi. Animations de solides en contacts par modèle physique, 2002.

[GBF01] Eran Guendelman, Robert Bridson, and Ronald Fedkiw. Nonconvex rigid bodies with starking. 2001.

[GME+00] Arthur Gregory, Ajith Mascarenhas, Stephen Ehmann, Ming Lin, and Dinesh Manocha. *Six Degree-of-Freedom Haptic Display of Polygonal Models*, pages 139–146. 2000.

[Gos01] Andrew Gosline. Evaluation of friction models with a haptic interface. 2001.

[HFK94] John C. Hart, George K. Francis, and Louis H. Kauffman. Visualizing quaternion rotation. *ACM Transactions on Graphics*, 13(3) :256–276, 1994.

[KSW+] Olaf Körner, Markus Schill, Clemens Wagner, H.J. Bender, and Reiner Männer. Haptic volume rendering with an intermediate local representation.

[Léc01] Anatole Lécuyer. *Contribution à l'étude des retours haptique et pseudo-haptique et de leur impact sur les simulations d'opérations de montage/démontage en aréonautique*. PhD thesis, Université Paris XI, 2001.

[LSW99] C. Leenerz, E. Schömer, and T. Warken. A framework for collision detection and response. 1999.

[McD00] Kevin T. McDonnell. Dynamic subdivision-based solid modeling. 2000.

[Mur97] Katta G. Murty. *Linear Complementary Linear And Nonlinear Programming, Internet Edition*. 1997.

[Red02] Stéphane Redon. *Algorithmes de simulation dynamique interactive d'objets rigides*. PhD thesis, Université d'Evry, 2002.

[SN01] J. Schmidt and H. Niemann. Using Quaternions for Parametrizing 3–D Rotations in Unconstrained Nonlinear Optimization. In T. Ertl, B. Girod, G. Greiner, H. Niemann, and H.-P. Seidel, editors, *Vision, Modeling, and Visualization 2001*, pages 399–406, Stuttgart, Germany, 2001. AKA/IOS Press, Berlin, Amsterdam.

[SS98] Jörg Sauer and Elmar Schömer. A constraint-based approach to rigid body dynamics for virtual reality applications. *Proc. ACM Symposium on Virtual Reality Software and Technology*, 1998.

[Ste00] David E. Stewart. *Time-stepping methods and the mathematics of rigid body dynamics*, chapter 9. Birkhäuser, 2000.

[Tza98] Constantinos Tzafesta. *Synthèse de retour kinesthésique et perception haptique lors de taches de manipulation virtuelle*. PhD thesis, Université Paris VI, 1998.

Webographie

OpenGL :
`http://www.opengl.org/`

Physics :
`http://www.martinb.com/physics/`

Quaterions :
`http://mathworld.wolfram.com/Quaternion.html`

Citeseer : Une mine de publications
`http://citeseer.nj.nec.com/cs/`

Gamma research group
`http://www.cs.unc.edu/~geom/`

LSC
http://lsc.cemif.univ-evry.fr:8080/

Touch-Hapsys
http://www.touch-hapsys.com/

Sensable Technologies
http://www.sensable.com/

Annexe A

Fichiers de configuration

A.1 Fichier de configuration graphique (`graphics.opt`)

```
/**
 *  Fichier d'options graphiques.
 *  Ce fichier fait partie du projet Haptics.
 *  La distribution et/ou la copie sans l'accord des auteurs est interdite.
 *  contact: slurdge@slurdge.org
 */

//La résolution
iScreenWidth = 800; iScreenHeight = 600;
//Double buffer
bDoubleBuffer = true;
//Le FOV
dScreenFOV = 45.0;
//Game mode
//Le game mode est beaucoup plus rapide que les autres modes
//Le game mode est forcément en plein écran.
bGameMode = false;
//Résolution pour le plein écran
strGameString="1280x1024:16@85";
//La résolution se trouve dans options.opt
//Plein écran
bFullScreen = false;
//Le nom de la fenêtre
sWindowName = "Haptics v 1.0";
//Position de départ de la fenêtre.
//La position -1,-1 correpond à la position par défaut
```

```
iWindowPosX = 100; iWindowPosY = 100;
//Cette variable controle le rendu graphique
//Si elle est à true, le rendu sera plus joli mais
//plus lent aussi.
bSmoothAndNice = true;
//Si on doit effectuer le culling
bCullFaces = true;
```

A.2 Fichier de configuration de souris (`mouse.opt`)

```
/**
 *  Fichier de configuration pour le curseur souris
 *
 *  \author Aurélien Pocheville
 */

//Nom du fichier qui contient le (les) bitmaps.
strBitmapFile = "mouse.tga";
//Nombre de curseurs (1 = static)
iNumberOfCursors = 36;
//Largeur
iWidth = 60;
//Hauteur
iHeight = 60;
//Nombre de curseurs sur une ligne
iRowNumberCursor = 6;
//Nombre de curseurs en hauteur
iColumnNumberCursor = 6;
//Spot X & Y(endroit ou l'on pointe)
iSpotX = 14; iSpotY = 14;
//Temps d'animation en seconde (pour faire toute la boucle)
dAnimTime = 2.0;
```

A.3 Fichier de configuration d'un objet (`obj_xx.opt`)

```
/**
 *  Simulation object configuration file.
 *  This file contains the setting for one object in the simulation
 */

strObjFileName = "box6.obj";
```

```
//The mass
mass = 22;

//The inertia matrix
//For a box, M/12*
//  y^2+z^2     0       0
//   0      x^2+z^2     0
//   0          0   y^2+x^2
iner_m00 = 0.9 ; iner_m10 = 0.0  ; iner_m20 = 0.0;
iner_m01 = 0.0  ; iner_m11 = 0.9 ; iner_m21 = 0.0;
iner_m02 = 0.0  ; iner_m12 = 0.0  ; iner_m22 = 0.9;

///The initial transformation matrix
//The upper 3x3 matrix represents the initial rotation of the object
//The last vector on the right the translation
//pos_m03 -> pos_m23 should be 0.
//pos_m33 should be 1, or collision detection calculus are subject to error
pos_m00 = 0 ; pos_m10 = 1 ; pos_m20 = 0 ; pos_m30 = 0.0 ;
pos_m01 = 0 ; pos_m11 = 0 ; pos_m21 = 0 ; pos_m31 = 1.0 ;
pos_m02 = 1 ; pos_m12 = 0 ; pos_m22 = 1 ; pos_m32 = 1.0 ;
pos_m03 = 0 ; pos_m13 = 0 ; pos_m23 = 0 ; pos_m33 = 1 ;

///The initial speed matrix
spe_m00 = 1 ; spe_m10 = 0 ; spe_m20 = 0 ; spe_m30 = 0 ;
spe_m01 = 0 ; spe_m11 = 1 ; spe_m21 = 0 ; spe_m31 = 0 ;
spe_m02 = 0 ; spe_m12 = 0 ; spe_m22 = 1 ; spe_m32 = 0 ;
spe_m03 = 0 ; spe_m13 = 0 ; spe_m23 = 0 ; spe_m33 = 1 ;

///The color
c_r = 0; c_g = 0.5; c_b = 1;
//Is the object fixed in space?
bIsFixed = false;

//Is the object controlled by the user ?
bUserControlled = true;
strController = "phantom";
//Is the gravity affecting the object ?
bGravityControlled = true;
```

A.4 Fichier de configuration du retour d'effort

```
/**
 *  GHOST configuration file
 *
 */

//Mamixum force in Newton
dForceMaxX = 5.0;
dForceMaxY = 5.0;
dForceMaxZ = 5.0;

//Phantom multiplicator
//The less this number is, the more the relative speed of the phantom is
dMultiplicator = 20;

//Ramp coefficients
dLinearCoef         = 1.55;
dExponentialCoef    = 7.00;
dThreshold          = 10.0;

//Force smoothing coefficients
//Tells to enable force smoothing
//Force smoothing apply the following formulae to each iteration:
//NewForce = NewForce * m_dForceSmooth + OldForce * (1-m_dForceSmooth);
//Note that a dForceSmooth of 1 is equivalent to put bForceSmoothing to false.
bForceSmoothing = true;
dForceSmooth    = 0.2;
```

Annexe B

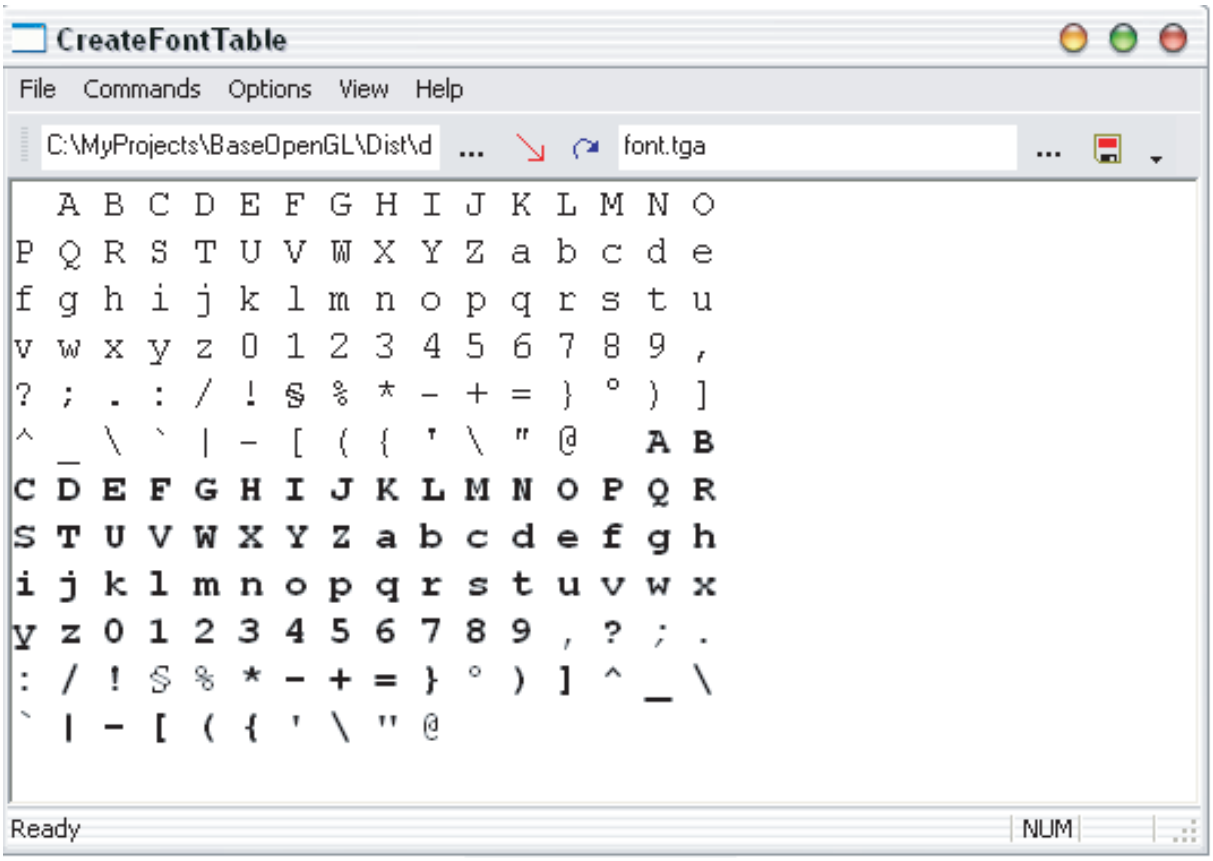
Fichier de debug de l’application

```
17:30:14:50 - -- Beginning of error file --
17:30:14:60 - -- Beginning of debug file --
17:30:14:60 - CApplication::Init -- Argument trouvé : c:\MyProjects\Haptics\Debug\Haptics.exe
17:30:14:60 - CApplication::Init -- On se place dans le répertoire default...
17:30:14:90 - CApplication::Init -- Initialisation principale de GLUT réussie
17:30:14:90 - CGraphics::Init -- Début de l’initialisation graphique
17:30:14:120 - CGraphics::Init -- Initialisation de la GLUT effectuée.
17:30:14:120 - CGraphics::Init -- Initialisation d’OpenGL effectuée
17:30:14:120 - CGraphics::Init -- Initialisation graphique réussie
17:30:14:120 - CApplication::Init -- Initialisation du module graphique réussie
17:30:14:120 - Informations système:
17:30:14:120 - ====================
17:30:14:120 - Système : Windows 5.1 build 2600 PlatformId 2 SP="Service Pack 1"
17:30:14:120 - Version de GLUT:3.000000
17:30:14:120 - Version de GLUI:2.010000
17:30:14:120 - Renderer OpenGL:GDI Generic
17:30:14:120 - Vendeur  OpenGL:Microsoft Corporation
17:30:14:120 - Version  OpenGL:1.1.0
17:30:14:120 - CApplication::Init -- Enregistrement des call-backs restants ...
17:30:14:430 - CApplication::Init -- Initialisation du système BASS terminée
17:30:14:490 - CApplication::Init -- On charge la scène principale ...
17:30:14:520 - CTgaImage::TgaImageLoad -- Début du chargement de l’image font_base.tga ...
17:30:14:520 - CTgaImage::TgaImageLoad -- On ouvre le fichier.
17:30:14:520 - CTgaImage::TgaImageLoad -- Chargement du header
17:30:14:520 - CTgaImage::TgaImageLoad -- Taille du header:0
17:30:14:520 - CTgaImage::TgaImageLoad -- Type de l’image:10
17:30:14:520 - CTgaImage::TgaImageLoad -- Caractéritiques de l’image: Width = 256, Height=256, Bpp=32
17:30:14:520 - CTgaImage::TgaImageLoad -- Taille de l’image calculée:256.000000 Ko (262144 octets)
17:30:14:540 - CTgaImage::TgaImageLoad -- Chargement de l’image font_base.tga correctement effectué
17:30:14:550 - CTexture::LoadTexture -- Chargement correctement effectué sur font_base.tga
17:30:14:560 - CTgaImage::TgaImageLoad -- Début du chargement de l’image font.tga ...
17:30:14:560 - CTgaImage::TgaImageLoad -- On ouvre le fichier.
17:30:14:560 - CTgaImage::TgaImageLoad -- Chargement du header
17:30:14:560 - CTgaImage::TgaImageLoad -- Taille du header:0
17:30:14:560 - CTgaImage::TgaImageLoad -- Type de l’image:10
17:30:14:560 - CTgaImage::TgaImageLoad -- Caractéritiques de l’image: Width = 512, Height=256, Bpp=32
17:30:14:560 - CTgaImage::TgaImageLoad -- Taille de l’image calculée:512.000000 Ko (524288 octets)
17:30:14:610 - CTgaImage::TgaImageLoad -- Chargement de l’image font.tga correctement effectué
17:30:14:650 - CTexture::LoadTexture -- Chargement correctement effectué sur font.tga
17:30:14:650 - CBaseScene::Init -- Creating SWIFT Scene ...
17:30:14:650 - CBaseScene::Init -- SWIFT Scene created.
Warning (Register_File_Reader): Replacing already registered file reader for magic number: "TRI"
Warning (Register_File_Reader): Replacing already registered file reader for magic number: "POLY"
17:30:14:680 - C3DSObject::LoadOBJ -- Ouverture du fichier (box1.obj)...
17:30:14:680 - C3DSObject::LoadOBJ -- Fichier ouvert.
17:30:14:680 - C3DSObject::LoadOBJ -- Taille du header :7
17:30:14:680 - C3DSObject::LoadOBJ -- Version du fichier: 12
17:30:14:680 - C3DSObject::LoadOBJ -- Nombre de vertices :864
17:30:14:680 - C3DSObject::LoadOBJ -- Nombre de faces : 288
17:30:14:690 - C3DSObject::LoadOBJ -- L’objet ne contient pas de coordonées de texture.
17:30:14:701 - C3DSObject::LoadOBJ -- Chargement éffectué sur le fichier box1.obj
17:30:14:701 - CSimulationObject::Load -- Initial matrix:
1 - 0 - 0 - 0 -
0 - 1 - 0 - 0 -
0 - 0 - 1 - 0 -
0 - 0 - 0 - 1 -
17:30:14:721 - CSimulationObject::Load -- Initial Quaternion:
quat : v[0]=0 v[1]=0 v[2]=0 v[3]=1
17:30:14:721 - CSimulationObject::Load -- Initial translation:
v[0] = 0 - v[1] = 0 - v[2] = 0
17:30:14:731 - CSimulationObject::Load -- Initial speed matrix:
1 - 0 - 0 - 0 -
```

```
0 - 1 - -1 - 0 -
0 - 1 - 1 - 0 -
0 - 0 - 0 - 1 -
17:30:14:741 - CSimulationObject::Load -- Initial speed Quaternion:
quat : v[0]=0.5 v[1]=0 v[2]=0 v[3]=0
17:30:14:751 - CSimulationObject::Load -- Initial speed vector:
v[0] = 0 - v[1] = 0 - v[2] = 0
17:30:14:751 - CSimulationObject::Load -- Initial quatRotationAccel:
quat : v[0]=0 v[1]=0 v[2]=0 v[3]=0
17:30:14:761 - CSimulationManager::Init -- Object number 0 initialized, ID = 1025
Warning (Register_File_Reader): Replacing already registered file reader for magic number: "TRI"
Warning (Register_File_Reader): Replacing already registered file reader for magic number: "POLY"
17:30:15:961 - C3DSObject::LoadOBJ -- Ouverture du fichier (box2.obj)...
17:30:15:961 - C3DSObject::LoadOBJ -- Fichier ouvert.
17:30:15:961 - C3DSObject::LoadOBJ -- Taille du header :7
17:30:15:961 - C3DSObject::LoadOBJ -- Version du fichier: 12
17:30:15:971 - C3DSObject::LoadOBJ -- Nombre de vertices :36
17:30:15:971 - C3DSObject::LoadOBJ -- Nombre de faces : 12
17:30:15:971 - C3DSObject::LoadOBJ -- L'objet ne contient pas de coordonées de texture.
17:30:15:971 - C3DSObject::LoadOBJ -- Chargement éffectué sur le fichier box2.obj
17:30:15:981 - CSimulationObject::Load -- Initial matrix:
1 - 0 - 0 - 0 -
0 - 1 - 0 - 0 -
0 - 0 - 1 - -1 -
0 - 0 - 0 - 1 -
17:30:15:991 - CSimulationObject::Load -- Initial Quaternion:
quat : v[0]=0 v[1]=0 v[2]=0 v[3]=1
17:30:15:1001 - CSimulationObject::Load -- Initial translation:
v[0] = 0 - v[1] = 0 - v[2] = -1
17:30:15:1001 - CSimulationObject::Load -- Initial speed matrix:
1 - 0 - 0 - 0 -
0 - 1 - 0 - 0 -
0 - 0 - 1 - 0 -
0 - 0 - 0 - 1 -
17:30:15:1021 - CSimulationObject::Load -- Initial speed Quaternion:
quat : v[0]=0 v[1]=0 v[2]=0 v[3]=0
17:30:15:1021 - CSimulationObject::Load -- Initial speed vector:
v[0] = 0 - v[1] = 0 - v[2] = 0
17:30:15:1031 - CSimulationObject::Load -- Initial quatRotationAccel:
quat : v[0]=0 v[1]=0 v[2]=0 v[3]=0
17:30:15:1041 - CSimulationManager::Init -- Object number 1 initialized, ID = 1026
17:30:15:1051 - CSimulationManager::Init -- Simulation initialized
17:30:15:1051 - CApplication::Init -- Entrée dans la boucle principale de GLUT
17:30:15:1061 - CApplication::Init -- Appel de glutMainLoop()...
17:30:15:1061 - CApplication::Init -- Enregistrement de atexit()...
17:30:17:3374 - CGui::SetTop -- Handle : 1026 Old ZOrder : 1
17:30:17:3384 - CGui::SetTop -- Handle : 1026 New ZOrder : 1
17:30:17:3384 - Call-back called : Button handle = 1027
17:30:17:3384 - Window found ! - Handle:1026
17:32:42:147802 - CApplication::ShutDown -- Fin de l'application...
17:32:42:147802 - CApplication::ShutDown -- Fin du système BASS...
17:32:42:147802 - CApplication::ShutDown -- Desactivation des scènes ...
17:32:42:147802 - CSimulationManager::ShutDown -- Shutting down simulation...
17:32:42:147802 - CSimulationManager::ShutDown -- Deleting SWIFT Scene...
17:32:42:147802 - CSimulationManager::ShutDown -- SWIFT Scene deleted.
17:32:42:147802 - CSimulationManager::ShutDown -- Releasing simulation objects...
17:32:42:147802 - CSimulationManager::ShutDown -- Objects released.
17:32:42:147802 - CSimulationManager::ShutDown -- Cleaning up...
17:32:42:147802 - CSimulationManager::ShutDown -- End of shut down
17:32:42:147802 - CApplication::ShutDown -- Desactivation des graphismes ...
17:32:42:147802 - CApplication::ShutDown --  --- Fin du programe ---
17:32:42:147802 - -- End of debug file --
17:32:42:147802 - -- End of error file --
```

Annexe C

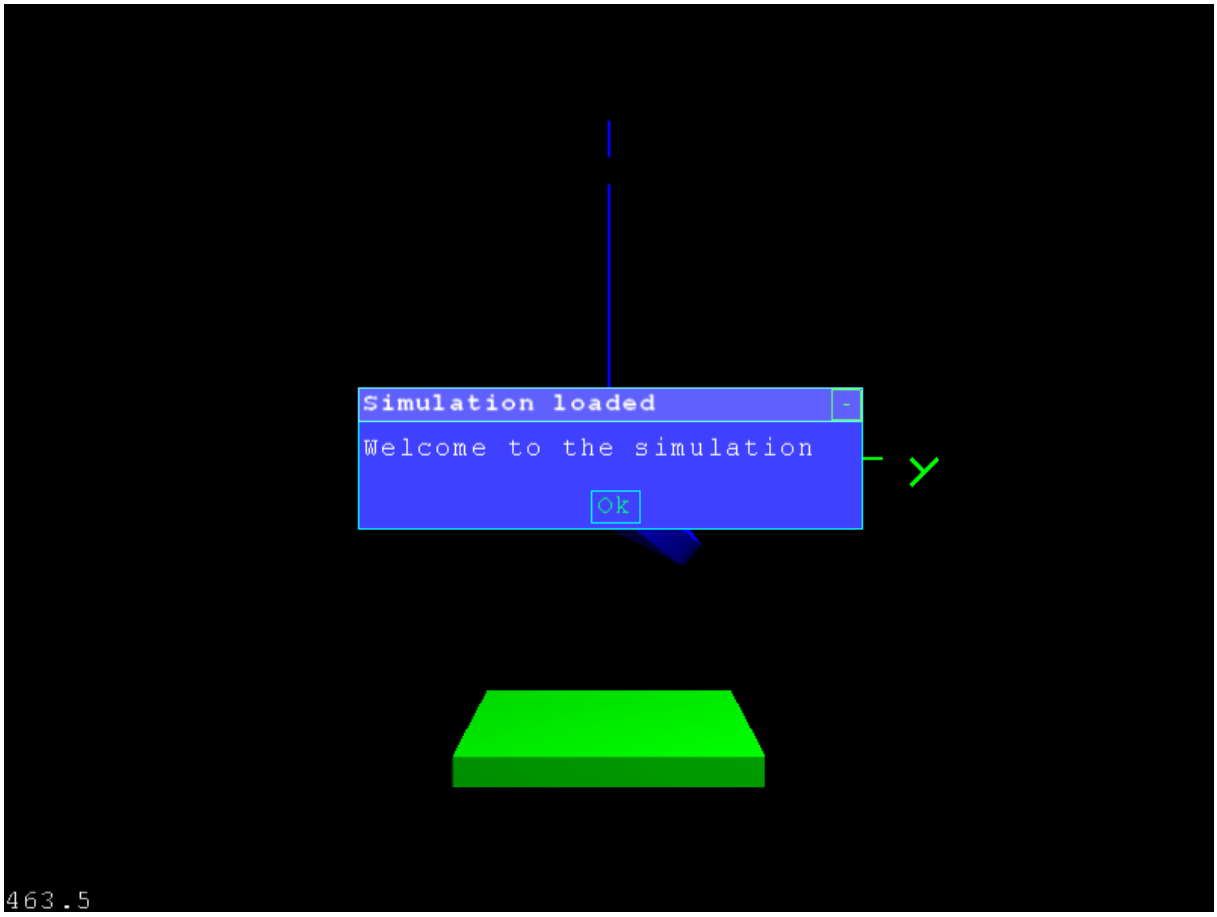
L’utilitaire de création de fonte



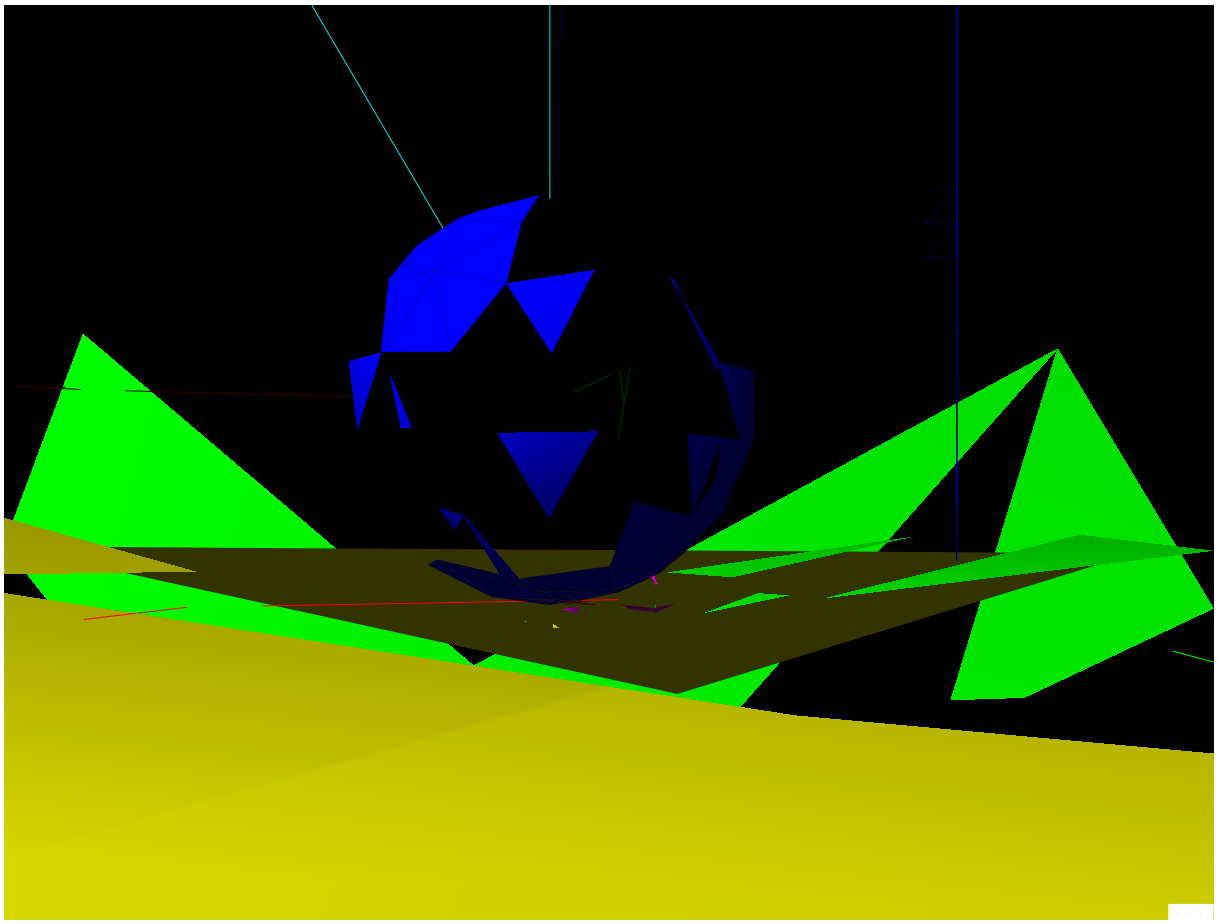
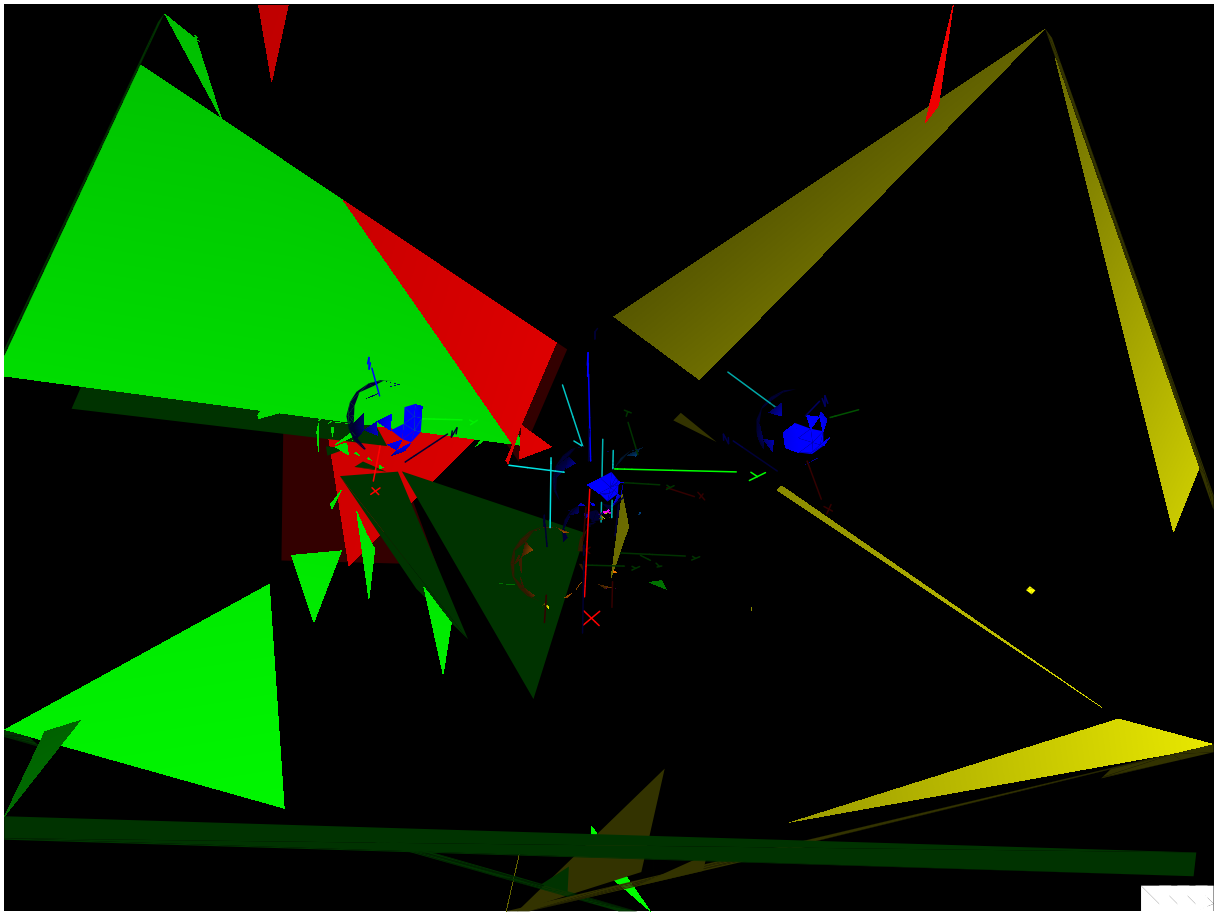
Annexe D

Photos d’écran

D.1 GUI en action



D.2 Un simple contact



D.3 Une cuvette : de nombreux contacts