ESSI 3ème année - VIMM

2001 - 2002

Marie Roque
Alexandre Reboul

Thomas Parle
Christophe Tornieri

Encadrement : Diane Lingrand

Résumé

Ce projet a pour objectif de modéliser et d'animer les vêtements en temps réel. Il s'agit de réaliser une simulation à la fois réaliste et suffisamment rapide pour une utilisation temps réel. Elle devra intégrer une gestion de l'environnement : vent, gravité, collisions… Une étude préalable des différents modèles existants sera réalisée afin de déterminer ceux qui nous paraîtrons les mieux adaptés à nos contraintes. Ce projet nous permettra de mettre en œuvre les algorithmes de résolution des équations différentielles et de simulation numérique.

TABLE DES MATIERES

1. ORGANISATION DU TRAVAIL **5**

1.1. DEROULEMENT 5
1.2. TABLEAU RECAPITULATIF 10

2. ETUDE DE L' EXISTANT **11**

2.1. DOCUMENTS 11
2.2. « PLUG-IN'S » 12

3. NOTIONS 3D, OPENGL **15**

3.1. MOUVEMENTS RIGIDES DANS L'ESPACE 3D 15
3.2. FORME MATRICIELLE DES TRANSFORMATIONS 16
3.2.1. Généralités 16
3.2.2. Principales transformations et coordonnées homogènes. 17
3.3. MODELES DE CONSERVATION DES POSITIONS ET ORIENTATIONS 19
3.3.1. Angles d'Euler et vecteur de translation 19
3.3.2. Matrice d'état 20
3.4. MODELE DE CAMERA 23

4. INTERFACE DE L'APPLICATION **24**

4.1. FONCTIONNALITES AVANCEES 25
4.1.1. Sélection interactive 25
4.1.2. Importation des fichiers 3ds 27

5. MODELES DE VETEMENTS **28**

5.1. REPRESENTATION PAR PARTICULES : PRINCIPE DE BASE 29
5.2. LES DIFFERENTS AGENCEMENTS 30
5.2.1. Agencement structurel 30
5.2.2. Agencement par rayon de connexion 33
5.2.3. Agencement par graphe 33

6. LES FORCES **35**

6.1. LES FORCES INTERNES 35
6.2. LES FORCES EXTERNES 36
6.2.1. La gravité 36
6.2.2. Force de dissipation 36
6.2.3. Force visqueuse 37

7. METHODES D'INTEGRATIONS EXPLICITES **37**

7.1. METHODE D'EULER 38
7.2. METHODE D'EULER MODIFIEE 39
7.3. METHODE DU ''MID-POINT'' 39
7.4. RUNGE-KUTTA 41

8. SURELONGATION **42**

8.1. CHOIX DE LA CONSTANTE DE RAIDEUR 43
8.2. CORRECTION DE POSITION 43
8.3. MODIFICATION DE LA VITESSE 45

9. METHODES D'INTEGRATION IMPLICITE **47**

9.1. REFORMULATION DES EQUATIONS 47
9.2. RESOLUTION INDIRECTE 48
9.3. RESOLUTION DIRECTE 49

10. GESTION DES COLLISIO NS **50**

10.1. DETECTION DES COLLISIONS 50
10.1.1. Collision avec une sphère 51

10.1.2. Collision avec un plan 51
10.1.3. Collision avec un cylindre 52
10.1.4. Collision avec des objets de forme quelconque 53
10.1.5. Détection avancée des collisions 55
10.2. REPONSE AUX COLLISIONS 57

11. THE RENDERING ENGINE 59

11.1. COMMON RENDERING ISSUES 59
11.1.1. Object Coherency 59
11.1.2. Graphic Effects and Genericity 59
11.1.3. Customization of Object Rendering 60
11.1.4. Multiplatform Robustness 60
11.1.5. API use and Specific Optimizations 61
11.2. GLOBAL RENDERING ENGINE ARCHITECTURE 61
11.2.1. Graphic Layer Abstraction 61
11.2.2. Altera tion of the Rendering Pipeline 63
11.3. OVERVIEW OF THE VERTEX PROCESSING UNIT 64
11.3.1. Per - Vertex Renderering Mechanism 64
11.3.2. Rendering Customization Possibilities 65
11.4. THE RENDERING INTERFACE 74
11.4.1. A High -Level Abstraction Layer 74
11.4.2. Optimizing the low -level API 75
11.5. SMOOTH OBJECT RENDERING 77
11.5.1. Rendering Engine Benefits 77
11.5.2. Rendered Smooth Object Examples 77
11.6. DISCUSSION ABOUT THE RENDERING ENGINE 78
11.6.1. Feature Summary 78
11.6.2. Potential Improvements 78
11.7. BRIEF CONCLUSION ON THE RENDERING ENGINE 79

Introduction

Ce projet s'est déroulé à l'ESSI, il a été proposé par Christophe Tornieri, étudiant de troisième année. Il existe actuellement de nombreux besoins dans le domaine de la simulation de vêtements, notamment dans l'industrie du cinéma et des jeux vidéos. Peu de logiciels de création d'images de synthèses intègrent en standard ce module.

Plus particulièrement, il est très difficile de trouver un logiciel permettant aux infographistes d'habiller en temps réel les personnages qu'ils créent. Nous pouvons citer par exemple le logiciel ***Poser de Metacreation*** qui intègre de puissants outils d'animation des personnages mais qui ne permet pas de gérer une dynamique réaliste des vêtements.

Nous pouvons aussi remarquer que les jeux vidéos se contentent pour l'instant d'habiller les personnages uniquement à l'aide de textures. La simulation des vêtements peut s'avérer un élément essentiel pour le réalisme dans les mondes virtuels.

Notre travail sera donc d'essayer de mettre en œuvre des techniques qui répondent le plus possible à ces besoins.

1. Organisation du travail

Ce projet s'est déroulé sur 6 mois, à raison de deux journées de travail par semaine. Pour le réaliser, nous avons formé une équipe de quatre courageux étudiants : Marie Roque, Thomas Parle, Alexandre Reboul et Christophe Tornieri. Thomas Parle ayant une situation particulière, nous avons dû nous organiser de manière sérieuse. En effet, Thomas était en entreprise pendant les heures allouées pour le projet. Il travaillait donc chez lui, le soir et le week-end, et nous nous réunissions ensuite pour faire le point. Dans cette partie, nous allons suivre l'ensemble des étapes franchies par ordre chronologique, puis nous établirons un tableau récapitulatif sur l'organisation générale de ce projet.

1.1. Déroulement

- Recherches (8 semaines + 3) :

  Afin d'aborder le sujet de la simulation d'objets souples, nous nous sommes beaucoup intéressés à l'existant dans ce domaine. Nous avons établi des recherches abondantes sur Internet. Nous avons trouvé la référence d'un livre *Cloth Modeling and Animation*[1] traitant de prés notre sujet. Diane Lingrand l'a commandé et 3 jours plus tard nous l'avions en possession. Ce livre ainsi que l'ensemble des documents trouvés sur ce sujet nous ont permis d'aborder tous les points difficiles de ce projet ambitieux. Cette étape a duré environ 1 mois. Des recherches complémentaires ont été faites lors des problèmes rencontrés et lors de l'intégration de nouvelles fonctionnalités.

- Modèle physique (5 semaines) :

  La première étape dans le développement a été de décider du modèle physique à adopter pour la modélisation du vêtement. Nous avons rencontré une première difficulté dans l'organisation du travail au début du développement puisque Christophe et Thomas ont chacun implémenté leur modèle de vêtement qui bien entendu n'était pas le même. Finalement cette mésentente a été bénéfique pour le projet car nous avions deux implémentations de modèles pour les objets souples. Nous avons donc pu

[1] ***Cloth Modeling and Animation*** de Donald H. House et David E. Breen

les comparer et garder les avantages de chacun pour ne faire plus qu'un modèle.

➢ Architecture de l'application (5 semaines) :

Le projet étant relativement complexe, nous nous sommes attelés à mettre en œuvre une architecture de code la plus générique possible. La simulation d'objets souples étant un domaine vaste, il nous fallait à tout prix une conception solide et « pensée » afin de faciliter l'intégration de nouvelles fonctionnalités à notre application de base. Nous avons choisi le langage C++ pour le développement, ce qui nous a permis de réfléchir en terme d'objets.

➢ Résolution de systèmes numériques (4 semaines) :

Pour une simulation réaliste d'un phénomène physique tel que le « tombé de vêtements », un certain nombre de méthodes numériques sont à maîtriser ainsi qu'à implémenter. Nous avons dû nous intéresser particulièrement à la résolution d'équations différentielles par les méthodes *d'Euler*, *Euler modifié*, *Runge-Kutta*, méthodes implicites… Nous avons implémenté toutes ces méthodes de résolution afin de comparer leurs performances respectives. Cette étape était très importante et incontournable à la vue du sujet !

➢ Développement d'une caméra (4 semaines) :

Pour nous permettre une meilleure visualisation de la scène (l'endroit de la simulation), nous avons développé à l'aide de la bibliothèque graphique *OpenGL*, une caméra similaire à celle que l'on peut appréhender dans le logiciel *3DSMax* de *Kinetix*. Le clavier et la souris ont des fonctionnalités particulières telles que le zoom avant, arrière ou la rotation autour de la scène… Cette implémentation rend l'application beaucoup plus intéractive et pratique de visualisation.

➢ Développement de l'interface (4 semaines) :

A ce stade de développement, nous avions une simulation de vêtements qui fonctionnait mais qui ne présentait que la scène programmée « en dur » dans l'application. Nous avons alors décidé de développer une interface graphique qui nous permettrait de créer notre scène avec des objets que nous pourrions paramétrés nous-même et ainsi rendre l'application beaucoup plus souple qu'auparavant. Pour cela, nous avons utilisé les *MFC* (*Microsoft Foundation Classes*) qui proposent des classes d'objets graphiques.

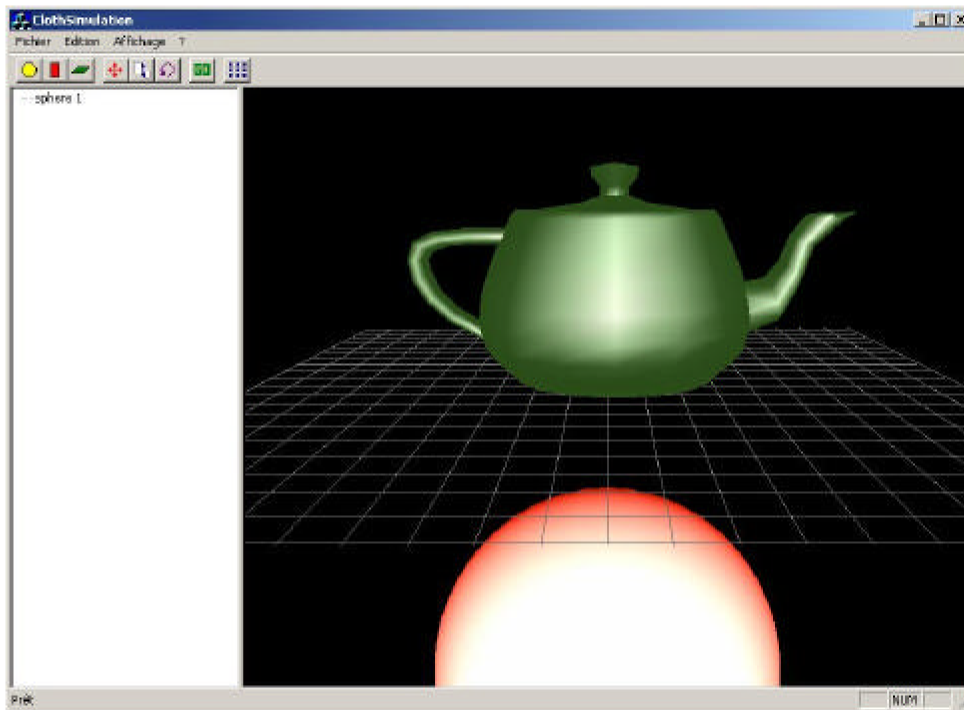


Figure 1 : Interface de l'application

- Gestion des collisions «simples » (2 semaines) :

Une fois la simulation d'objets souples implémentée, nous avons commencé à nous intéresser à la gestion des collisions. Dans un premier temps, seuls des objets simples tels que les sphères, les cylindres, les plans ont été gérés. Le test de collision avec ces primitives est simple à mettre en œuvre contrairement à la gestion des collisions avec un objet 3D quelconque. En effet, pour la sphère par exemple, il suffit de tester si les particules constituants l'objet souple (le vêtement) sont entrées dans la sphère ou pas. Grâce aux propriétés de ces primitives, le test est trivial : le rayon informe sur une collision possible avec une sphère… Afin de réduire le nombre de tests à chaque pas de temps, nous avons implémenté l'algorithme des boîtes englobantes («*BoundingBox* »). Cette intégration a permis d'accélérer la simulation sans la « dégrader ». Avec cette gestion des collisions « simples », nous avons pu vérifier le réalisme de notre simulation !

Figure 2 : Tombé d'un drap vert sur une sphère

- Intégration d'objets 3DSMax (3 semaines) :

A ce niveau de l'application, nous avons ressenti le besoin d'importer des objets 3D complexes afin de rendre la simulation plus attrayante. En effet, simuler le «tombé » d'un vêtement sur une sphère n'est pas très intéressant en soi, mieux vaut simuler le «tombé » d'un vêtement sur un mannequin ou sur une table dans le cas d'un drap. C'est pourquoi nous avons intégré au programme, l'importation des fichiers 3DSMax pour n'avoir non plus une sphère en tant qu'objet de collision mais un objet 3D

complexe tel qu'un mannequin 3D. Ces objets 3DSMax dans un premier temps représentaient l'objet souple de notre application, c'est-à-dire que si nous importions une théière, celle-ci se comportait en objet souple et se déformait donc lors du « tombé ». La *figure 1* montre une théière qui est en fait la représentation de l'objet souple de la scène. Dans un second temps, nous avons rendu l'importation des fichiers 3DSMax plus générale, en donnant la possibilité aux objets 3D importés de se comportaient en objets de collision.

- Gestion de tous types de collisions (3 semaines) :

Une fois l'intégration dans l'application de l'import d'objets 3D, il nous fallait alors implémenter la gestion des collisions avec ces objets complexes. Cette étape est loin d'être évidente, puisque tester la collision entre deux objets 3D complexes comportant des milliers de faces allait rendre la simulation très lente et donc nous allions perdre l'objectif « temps réel » que nous nous étions fixés au début. Afin de garder une simulation rapide, nous avons donc implémenté un **octree** pour chaque objet 3D. Cette astuce de programmation permet d'éliminer de nombreux tests inutiles lors de la gestion des collisions.

- Développement de différents rendus (10 semaines) :

Dans un dernier temps, nous avons intégré à l'interface la possibilité de rendre la simulation de différentes manières : rendu chromé, rendu utilisant du Phong, rendu non-réaliste (dessin animés), rendu utilisant du *Cube mapping*…

Cette dernière implémentation permet d'avoir dans notre application, la possibilité de « rejouer » la scène en utilisant un rendu différent :

Figure 3 : 3 rendus différents : chromé, Phong, Velours

1.2. Tableau récapitulatif

Dans cette partie, nous ferons un récapitulatif des tâches réalisées en fonction de leur durée et des intervenants. Nous avions tous une compétence particulière que nous pouvions apporter à l'application. En effet, Christophe avec ses capacités en mathématiques et programmation s'est concentré sur toute la partie résolution, graphe, octree. Alexandre passionné par la 3D et les logiciels de modélisation 3D, s'est intéressé à la visualisation 3D avec l'implémentation de la *Caméra*. Marie ayant des connaissances sur l'utilisation des MFC, s'est occupé de l'interface de l'application. Et Thomas, étant particulièrement attaché à tous ce qui touche le rendu final, s'est consacré au développement des différents rendus.

Le tableau qui va suivre permet de rendre compte de l'organisation du travail et de l'effectif pour ce projet de 6 mois :

semaines:	44	45	46	47	48	49	50	51	52	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Recherches			M	A	C	T																		
Modèle physique						C	T																	
Architecture de l'application					M	A	C	T																
Résolution de systèmes numériques								C			C													
Développement d'une "caméra"								A			A													
Développement d'une interface								M			M													
Gestion des collisions "simples"												M	A											
Intégration d'objets 3DSMax														C	A									
Gestion de tous types de collisions																C								
Développement de différents rendus																T								
Ecriture du rapport																					M	A	C	T

Légende :	
M=	Marie
A=	Alexandre
C=	Christophe
T=	Thomas

Ce projet développé à quatre, nous a enrichi. Nous avons dû adopter une politique d'organisation consciencieuse afin de réussir un « beau » travail d'équipe. En effet, le point le plus important à retenir au niveau de l'organisation d'un projet est le fait qu'un travail d'équipe bien dirigé est la clef de résultats optimaux en qualité et en temps de réalisation. Les compétences de chacun ont été exploitées au mieux, nous étions assez complémentaires ce qui a rendu la répartition du travail plus simple. En conclusion, la gestion du temps et des personnes s'est faite de manière naturelle, et nous sommes tous fiers aujourd'hui du résultat final !

2. Etude de l' existant

L'étude de l'existant a été une étape importante qui s'est prolongée tout au long du projet. Ces recherches nous ont permis de connaître les limites actuelles du domaine ainsi que les performances des modèles existants. Nous avons rassemblé l'ensemble des documents relatifs à la simulation d'objets souples et nous les avons étudiés. Nous allons dans cette partie, faire une synthèse des informations trouvées dans les documents, puis nous survolerons les différents *plug-in's* des plus populaires logiciels 3D traitant du sujet.

2.1. Documents

Nous avons pu remarquer l'apparition du nom de Xavier Provost à plusieurs reprises sur Internet lors de recherches sur la simulation de vêtements. Il a introduit le modèle basé sur les particules et les ressorts considéré comme le modèle élastique. Ce modèle permet de représenter le vêtement comme une structure discrète d'éléments et non comme une surface continue. Il a également donné un algorithme de correction d'élongation que nous avons par la suite étudié. Mais cet solution s'est avérée assez instable !

Nous nous sommes également beaucoup documentés sur la gestion des collisions. Pascal Volino et Nadia Magnenat Thalmann de l'université de Genève *MIRALab* sont souvent cités et à l'origine d'un grand nombre de documents sur ce sujet. Ils expliquent la difficulté de garder une rapidité d'exécution quand il faut gérer les collisions dans une scène avec des milliers de polygones. Ils ont introduit la notion d'octree afin d'accélérer la simulation.

Un autre personnage de la scène graphique dans le monde, David Barraf, s'est intéressé à la simulation de vêtements en temps réel. Il fut le premier à introduire l'utilisation d'une méthode implicite pour la résolution des équations différentielles. Son objectif était de simuler la scène plus rapidement. Il explique dans ses travaux que les méthodes explicites impliquent un pas de discrétisation petit pour obtenir une simulation stable alors que les méthodes implicites permettent un pas de discrétisation très grand sans pour autant rendre la simulation instable.

2.2. « Plug-in's »

Nous allons nous intéresser particulièrement aux *plug-in's* des logiciels de modélisation 3D les plus connus comme *Softimage, 3DSMax et Cinema 4D.*

➢ « ***SOFTIMAGE*** » :

Les caractéristiques de ce *plug-in* sont les suivantes :

- *Facilité d'utilisation* - simule une variété d'animations de vêtements comme les rideaux, mouchoirs…
- *Type de contrôles* - rigidité et flexibilité du tissu.
- *Modèle dynamique* - le modèle de vêtements réagit aux forces telles que la gravité et le vent.
- *Rapidité* - la simulation est pre-calculée afin de permettre une interaction rapide dans Softimage.
- *Détection des collisions* - la détection ne gère que les collisions du vêtement avec une sphère !
- *Collisions propres* - gestion des collisions du vêtement avec lui-même.

➢ ***« CINEMA 4D Dynamics (MAXON) » :***

Cinema 4D Dynamics est certainement le *plug-in* le plus impressionnant que nous ayons trouvé. Il gère la simulation de phénomènes naturels avec un rendu magnifique :

Il est capable de simuler un monde réaliste où des forces telles que le vent, la gravité, les ressorts, les frictions… évoluent. Les résultats proposés sur le site Internet de *Maxon* sont sous forme de vidéos. Nous ne savons pas le temps de calcul que cela demande mais il doit être équivalent aux autres *plug-in's* sur ce sujet : c'est-à-dire que plus la scène est complexe, plus la simulation sera lente. On a pu remarquer que pour une scène de complexité moyenne (un vêtement modélisé par une grille de 40 points par 40 par exemple), la simulation commence déjà à ralentir et à perdre la notion de temps réel.

➢ ***« 3DSMAX - Kelseus » :***

Kelseus est le *plug-in* que nous avons le plus testé car nous avions la chance de pouvoir le manipuler sur nos machines. Il présente à priori les mêmes possibilités que notre application : gestion du vent, des collisions simples et complexes, modélisation du vêtement… Il utilise une méthode implicite pour la résolution des équations différentielles contrairement à la méthode explicite mise en œuvre dans notre application.

Nous avons remarqué que pour les collisions, il se comporte de la même manière que notre simulation, c'est-à-dire que lorsqu'un triangle rencontre un segment aucune collision n'est détectée. La simulation semble néanmoins plus lente que la nôtre.

En conclusion de l'étude de l'existant, nous pouvons dire que la simulation d'objets souples est un domaine encore en recherche à l'heure actuelle. En effet, les résultats obtenus jusqu'à présent sont de bonne qualité mais l'exécution est rarement temps réel. Cela est plutôt un problème pour une intégration dans les jeux vidéos. Il faut donc faire des compromis entre réalisme et rapidité même si avec les nouvelles cartes graphiques de plus en plus de notions sont gérées matériellement.

3. Notions 3D, OpenGL

Afin de pouvoir tester notre modèle de vêtements et d'en simuler l'interaction avec un environnement donné, nous avons mis au point une application développée en C++/OpenGL, sous Windows.

Elle devait pouvoir mettre en œuvre:

- Des objets souples: les vêtements

- Et des objets rigides: les objets entrant en collision avec les vêtements

Nous nous proposons dans cette partie de présenter les fondements du fonctionnement de notre application.

3.1. Mouvements rigides dans l'espace 3D

Ce chapitre a pour but d'expliquer de quelle manière ont été modélisés le positionnement et le déplacement des objets dans notre scène 3D.

On peut définir le mouvement de l'ensemble d'une scène par la composition de 2 mouvements :

- Le mouvement de la camera: le mouvement 'global' de tous les objets de la scène.
- Le mouvement propre de chaque objet.

Afin de positionner la camera et les objets rigides, nous avons alors considéré les deux transformations rigides:

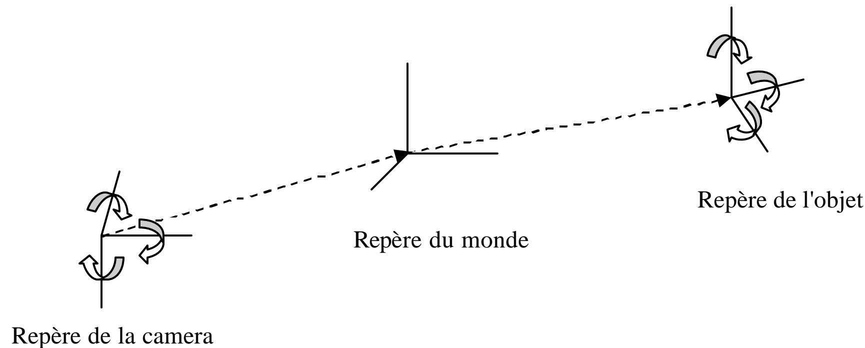
- translation
- rotation

Aussi, afin de pouvoir effectuer ces transformations, était-il nécessaire de se placer dans un repère défini.

Les différents types de repères intéressants à considérer étaient:

- Le repère absolu, ou repère du monde

- Le repère de la camera
- Ou encore le repère de chaque objet



Les parties suivantes expliquent de manière plus détaillée la façon dont ceci a été mis en œuvre.

3.2. Forme matricielle des transformations

3.2.1. Généralités

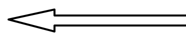
Représenter les transformations sous la forme matricielle permet d'effectuer des compositions de transformations par simple multiplication de matrices.

Ainsi soit 3 transformations : f, g et h, représentées respectivement par les matrices F,G et H,
La composition $f \circ g \circ h$ se traduit de manière matricielle par le produit H.G.F.

On se rappellera alors que l'ordre dans lequel les transformations sont effectuées, est l'ordre inverse de multiplication des matrices :

Soit P un point de l'espace, P' le point obtenu après transformation est défini par:

$$P' = (H.G.F).P$$



Ordre d'application des transformations

3.2.2. Principales transformations et coordonnées homogènes.

Etant donné les deux points suivants:

- la translation t d'un point P de l'espace 3D se traduit par l'opération :

$$P' = P + t \qquad \text{avec} \qquad t = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \quad \text{et} \quad P = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

- la rotation R du point P autour d'un axe se traduit par l'opération :

P' = R.P avec R matrice de rotation 3X3

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\boldsymbol{q}) & -\sin(\boldsymbol{q}) \\ 0 & \sin(\boldsymbol{q}) & \cos(\boldsymbol{q}) \end{bmatrix} \quad R_y = \begin{bmatrix} \cos(\boldsymbol{q}) & 0 & \sin(\boldsymbol{q}) \\ 0 & 1 & 0 \\ -\sin(\boldsymbol{q}) & 0 & \cos(\boldsymbol{q}) \end{bmatrix} \quad R_z = \begin{bmatrix} \cos(\boldsymbol{q}) & -\sin(\boldsymbol{q}) & 0 \\ \sin(\boldsymbol{q}) & \cos(\boldsymbol{q}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation autour de l'axe X — Rotation autour de l'axe Y — Rotation autour de l'axe Z

avec $\boldsymbol{q}$ angle de rotation.

Il est intéressant de pouvoir homogénéiser toutes les transformations de type rotations, translations en les repr ésentant de la même manière.

L'ajout d'une composante supplémentaire, en 3D définition d'une matrice 4X4 au lieu de 3X3, permet d'effectuer l'ensemble de ces opérations sous la forme d'un simple produit matriciel. Nous travaillons alors en système dit 'de coordonnées homogènes'.

L'expression des transformations précédentes dans ce nouveau système de coordonnée est donnée par :

$$T_h = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ pour une translation,

$$R_h = \begin{bmatrix} R_{11} & R_{12} & R_{13} & 0 \\ R_{21} & R_{22} & R_{23} & 0 \\ R_{31} & R_{32} & R_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ avec R défini précédemment pour les rotations autour des 3 axes du repère.

Et nous pouvons même définir par la même occasion la matrice de changement d'échelle

$$S_h = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ correspondant en coordonnées standards à la matrice $$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_y \end{bmatrix}$$

On s'aperçoit alors que la composée d'une rotation R puis d'une translation t, peut être définie par la transformation matricielle suivante :

$$P' = \begin{bmatrix} [R] & [t] \\ 0 & 1 \end{bmatrix} .P$$ avec cette fois-ci $$P = \begin{bmatrix} P_x \\ P_y \\ P_z \\ P_w \end{bmatrix}$$

Quelques propriétés des coordonnées homogènes utiles à savoir sont les suivantes:

- Deux quadruplets de la forme $[x, y, z, w]^T$, différents de $[0,0,0,0]^T$ représentent le même point si l'un est multiple de l'autre.
- Une représentation standard d'un point $[x, y, z, w]^T$, $w \neq 0$ est donnée par $[x/w, y/w, z/w, 1]^T$

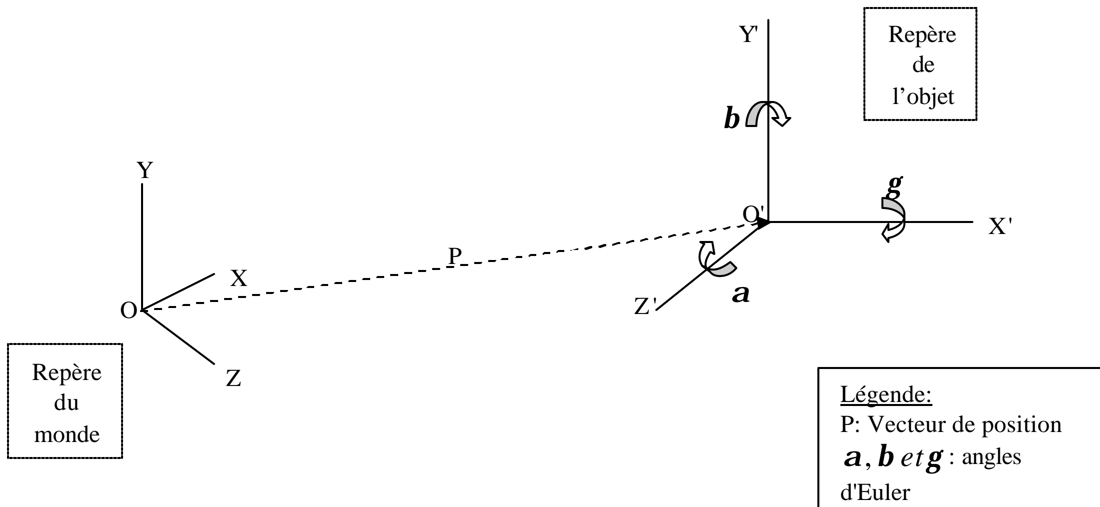
  Les points pour lesquels w = 0 sont dits 'points à l'infini'.

3.3. Modèles de conservation des positions et orientations

3.3.1. Angles d'Euler et vecteur de translation

Cette première approche n'utilisant pas le système de système de coordonnées homogènes, se propose de représenter tout positionnement d'objet ou camera dans la scène, par la donnée de:

- sa position sous la forme d'un vecteur de translation du centre du repère de l'objet par rapport au centre du repère du monde.
- son orientation exprimée par la valeur des trois angles de rotation autour de chaque axe du repère:
  - $\boldsymbol{g}$: angle de rotation autour de l'axe x
  - $\boldsymbol{b}$: angle de rotation autour de l'axe y
  - $\boldsymbol{a}$: angle de rotation autour de l'axe z



On obtient alors comme exprimé précédemment les matrices de rotation correspondantes:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\boldsymbol{g}) & -\sin(\boldsymbol{g}) \\ 0 & \sin(\boldsymbol{g}) & \cos(\boldsymbol{g}) \end{bmatrix} \quad R_y = \begin{bmatrix} \cos(\boldsymbol{b}) & 0 & \sin(\boldsymbol{b}) \\ 0 & 1 & 0 \\ -\sin(\boldsymbol{b}) & 0 & \cos(\boldsymbol{b}) \end{bmatrix} \quad R_z = \begin{bmatrix} \cos(\boldsymbol{a}) & -\sin(\boldsymbol{a}) & 0 \\ \sin(\boldsymbol{a}) & \cos(\boldsymbol{a}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation autour de l'axe X — Rotation autour de l'axe Y — Rotation autour de l'axe Z

L'expression de toute rotation R autour du point O est alors donnée par la composée de ces 3 rotations :

$$R = R_z R_y R_x$$

Attention :

Les rotations en 3D ne sont pas commutatives, contrairement au cas 2D.

Les angles d'Euler ne sont donc généralement utilisés que si on effectue des rotations que sur 2 axes à la fois : par exemple, rotation autour des axes X et Y.

Du moment que l'on fait varier la troisième composante de rotation, il apparaît des phénomènes de rotation en sens inverse ('gimble lock'), rendant le modèle inutilisable.

On notera d'autre part que la méthode des angles d'Euler ne permet pas d'effectuer des rotations aux objets autour d'un point arbitraire.

3.3.2. Matrice d'état

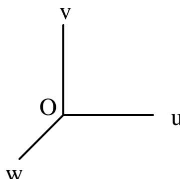
- **Généralités**

Une meilleure approche que l'utilisation des angles d'Euler , est de conserver les rotations successives dans le bon ordre :

Ainsi, au lieu de toujours effectuer $R = R_z R_y R_x$ en changeant les valeurs de α, β, γ (méthode d'Euler), on effectue cette fois-ci $R = R_n ... R_3 R_2 R_1$.

Cette méthode permet de modéliser le positionnement de l'objet dans l'espace 3D non plus à partir de son vecteur position et de son orientation, mais d'utiliser directement une matrice d''état' de l'objet, contenant toutes ces informations. Cette matrice comme expliqué précédemment, est exprimée en coordonnées homogènes.

On peut alors décrire cette matrice d'état de la manière suivante:



$$\begin{bmatrix} u_x & v_x & w_x & O_x \\ u_y & v_y & w_y & O_y \\ u_z & v_z & w_z & O_z \\ * & * & * & * \end{bmatrix}$$ avec le repère de l'objet .

Les vecteurs u, v, w et O sont alors exprimés dans le repère absolu (le repère du monde).

On comprend alors que le rôle de cette matrice est d'effectuer un changement de repère entre le repère du monde et le repère de l'objet. C'est une matrice de passage.

- **Transformation en repère local ou repère global**

Nous avons vu précédemment que de manière habituelle, lorsque nous voulons composer des transformations successives, il suffit de multiplier les matrices correspondantes à gauche.

En effet, on rappelle que pour appliquer au point P les transformations successives T1, T2, T3, on effectue :.

P' = T3.T2.T1.P

Chaque nouvelle transformation est alors appliquée à partir du repère obtenu après application de toutes les transformation précédente :

Cette transformation est donc appliquée dans le repère local à l'objet.

Il serait alors intéressant dans une application, de pouvoir spécifier les transformations par rapport au repère absolu. En effet, on aimerait bien que lorsque l'on bouge la souris vers la gauche, l'objet se déplace toujours du même côté, quelle que soit son orientation.

Pour exprimer des transformations dans le repère global, il suffit alors d'inverser l'ordre de composition des transformations, en effectuant des multiplications à droite au lieu de gauche.

Ainsi on obtient P' = T1.T2.T3.P et chaque nouvelle transformation Ti sera effectuée avant toutes les autres. On se place donc toujours dans le repère de départ pour effectuer la transformation.

➢ Implémentation en OpenGL

OpenGL représente les transformations de base : rotation, translation, mise à l'échelle, par des matrices en coordonnées homogènes.

Cependant l'implémentation des matrices est orientée colonnes et non pas orientée ligne comme précédemment.

C'est à dire qu'au lieu d'effectuer l'opération :

$$P' = M.P \quad \text{avec} \quad M = \begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_4 & C_5 & C_6 & C_7 \\ C_8 & C_9 & C_{10} & C_{11} \\ C_{12} & C_{13} & C_{14} & C_{15} \end{bmatrix}$$

OpenGL effectue l'opération : $P'^T = P^T M^T$ avec $M^T = \begin{bmatrix} C_0 & C_4 & C_8 & C_{12} \\ C_1 & C_5 & C_9 & C_{13} \\ C_2 & C_6 & C_{10} & C_{14} \\ C_3 & C_7 & C_{11} & C_{15} \end{bmatrix}$

Cette implémentation revient alors à inverser les sens de multiplication par rapport à ce qui a été exposé précédemment.

Ainsi, toutes les transformations en OpenGL sont effectuées par défaut dans le repère local (multiplication matricielle à gauche en représentation normale, et multiplication à droite en représentation OpenGL)

Astuce utile

Comment effectuer une multiplication à gauche [au sens OpenGL] en OpenGL.

Prenons par exemple la primitive OpenGL : glMultMatrix.
Soit stateMatrix la matrice à multiplier à gauche par la matrice de transformation: transformationMatrix

On veut donc effectuer: stateMatrix = transformationMatrix * stateMatrix

```
glMatrixMode(GL_MODELVIEW);                       // Passage en mode ModelView pour effectuer les opérations
glPushMatrix();                                   // Sauvegarde de la matrice courante
glLoadMatrixf(transformationMatrix);              // Chargement de la matrice de transformation
glMultMatrixf(stateMatrix);                       // Multiplication de stateMatrix à droite
glGetFloatv(GL_MODELVIEW_MATRIX, stateMatrix);    // Sauvegarde de stateMatrix
glPopMatrix();                                    // Restauration de la matrice ModelView initiale
```

Note : le procédé est le même pour les fonctions glTranslate, glRotate ou glScale qui effectuent de la même manière que glMultMatrix, une multiplication à droite [OpenGL] de matrice.

3.4. Modèle de caméra

Une fois tous les concepts précédents bien assimilés, le problème était de pouvoir trouver quels mouvements de cameras associer aux mouvement de souris, afin que l'utilisateur puisse s'orienter et se positionner du mieux possible dans la scène 3D.

Une solution simple était :

- d'effectuer une rotation *ry* autour de l'axe Y du monde pour un mouvement *dx* de la souris,
- et d'effectuer une rotation *rx* autour de l'axe X du monde pour un mouvement *dy* de la souris.

Cependant, il a été remarqué que les logiciels de graphisme comme 3D Studio Max ou encore Cinema 4DXL, utilisent un autre modèle qui permet de se déplacer dans la scène beaucoup plus facilement. Après avoir trouvé par observation comment se comportait se type de camera, nous avons pu l'implémenter dans notre application.

Le principe est le suivant :

- effectuer une rotation ry autour de l'axe Y du monde pour un mouvement dx de la souris (comme précédemment)
- effectuer une rotation rx autour de l'axe X de la camera, translaté au centre du monde.

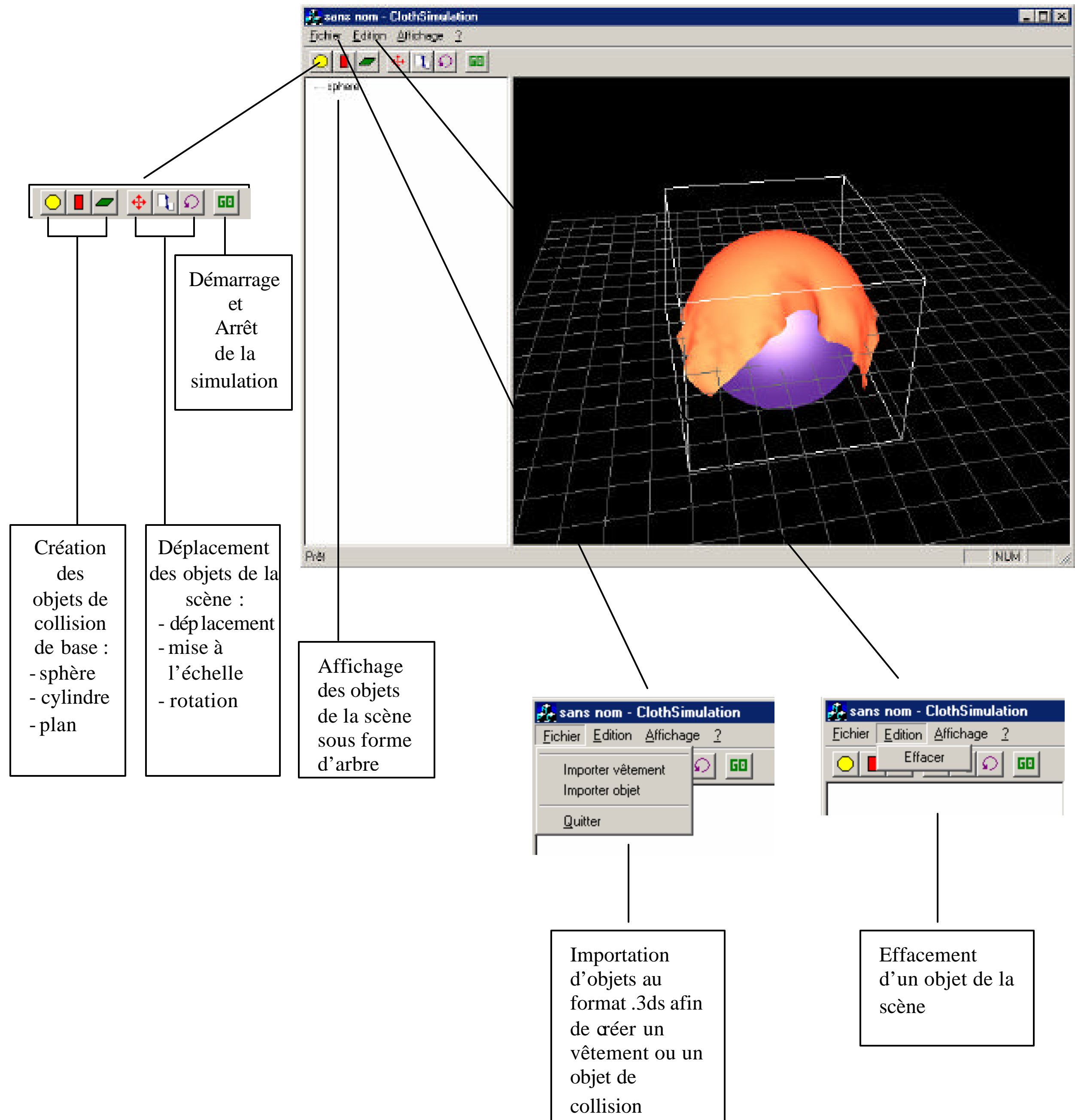
Cette transformation a notamment la bonne propriété de restaurer la position de camera initiale lorsqu'on ramène après un mouvement quelconque, la souris à son point d'origine.

D'autres solutions, comme l'utilisation de la 'boule virtuelle' (ou 'trackball'), permettant de déplacer la camera en restant sur une boule englobant la scène, ont été

envisagées, mais se sont révélées après essai, beaucoup moins maniable que la solution précédente.

4. Interface de l'application

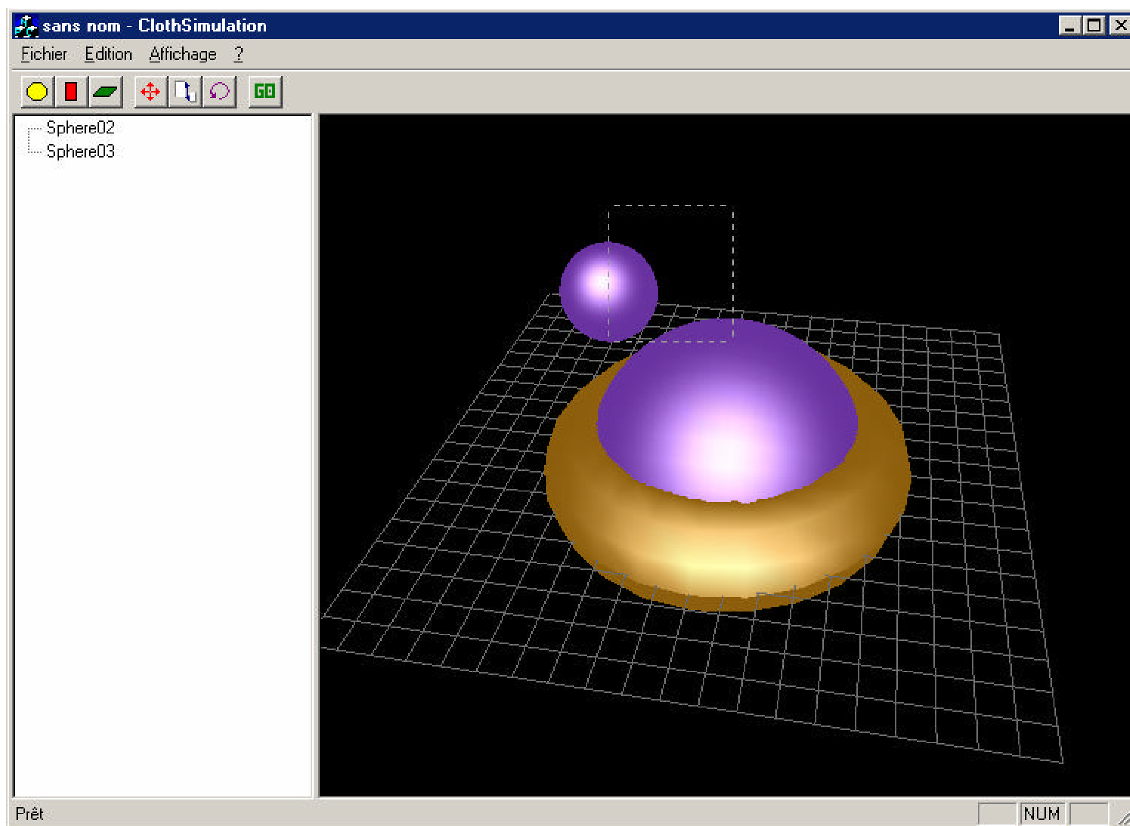
L'interface de notre application se compose de la manière suivante :



4.1. Fonctionnalités avancées

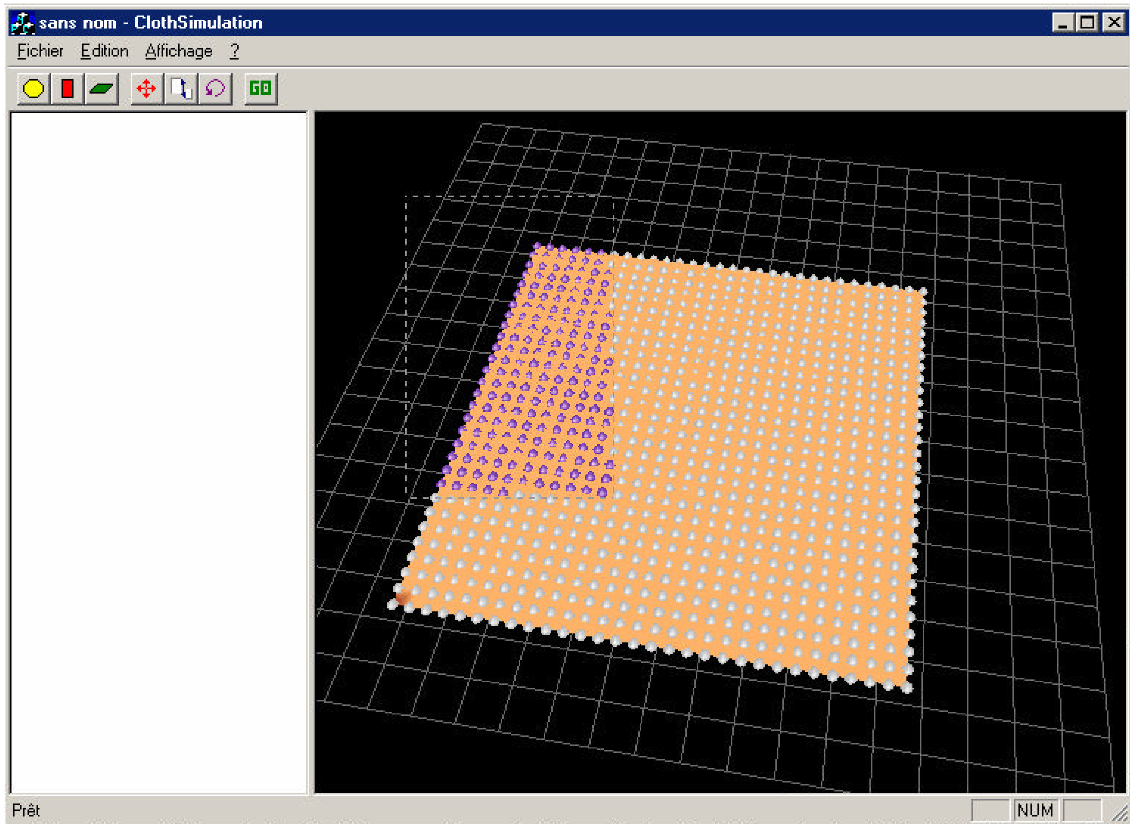
4.1.1. Sélection interactive

Afin de manipuler les objets de la scène et de leur appliquer les diverses transformations voulues, nous avons implémenté des outils de sélection en utilisant les primitives basiques OpenGL de picking.
Il a fallu cependant prendre en compte les limitations de la pile de noms (utilisée lors de l'appel des fonctions glInitNames(), glPushName(), glLoadName() …) à 64 éléments.
C'est a dire qu'il a fallu dessiner lors du rendu de sélection, les objets en nombre assez faible, pour que le nombre d'informations unitaires inscrites dans le buffer de sélection (nombre de hits, profondeurs maximales, ou noms …) soit inférieur à 64.



Sélection d'objets afin d'effectuer les transformations de base sur les objets (rotations, translations, mises à l'échelle)

On notera la possibilité de sélectionner plusieurs objets à la fois, et de pouvoir ajuster la sélection au fur et à mesure en y ajoutant ou supprimant des objets.



Sélection en mode sous objet d'un vêtement afin de déplacer et d'ancrer des particules

<u>Notes :</u>

Il semblerait que le mécanisme OpenGL de picking rencontre certains problèmes en environnement MFC.

Il a été remarqué que les informations de profondeur dans le buffer de sélection (z-min et z-max) ne sont pas initialisées.

Il est fortement recommandé, après avoir pris le recul nécessaire, d'utiliser dans une version ultérieure de notre application, un mécanisme de sélection du type marquage coloré, beaucoup plus efficace et adapté à la charge. (Voir cours de Frédéric Devernay sur le site de Michel Buffa, cf Références).

4.1.2. Importation des fichiers 3ds

Le moteur d'importation des fichiers .3ds que nous avons utilisé dans notre application, est celui de DigiBen de GameTutorials.
A la base, ce moteur était une version très allégée et codée en se rapprochant beaucoup du langage C.
Nous avons réorganisé ce moteur afin qu'il ait une architecture C++ portable, puis nous l'avons adapté à notre application, notamment au niveau des rendus, qui sont pris directement en charge par le moteur de rendu de notre application.

Ce moteur d'importation a été une base très importante pour les fonctionnalités avancées de notre projet, permettant comme décrit dans les chapitres précédents :

- de créer un objet souple
- de créer un objet de collision.

5. Modèles de vêtements

Plusieurs approches peuvent être envisagées pour simuler de manière réaliste les vêtements. Toutes ces méthodes peuvent être divisées en deux catégories : les représentations *continues* et les représentations par *systèmes de particules*.

Les représentations continues font appel à des modélisations par *éléments finis* introduites par ***EISCHEN.*** Le principal avantage de cette représentation est la précision et l'exactitude des résultats qu'elle permet d'obtenir. En revanche les calculs mis en œuvre sont encore trop lourds pour espérer une simulation temps réel ou même quasi temps réel. De plus l'introduction de dynamique dans une telle représentation n'est pas triviale. En effet, tous les modèles continus établis jusqu'à présent ne sont utilisés que pour des représentations statiques des vêtements[2]. C'est pour cette raison que notre étude s'est dirigée vers les modèles par *systèmes de particules*, plus appropriés dans le cadre d'une simulation temps réel.

Les *systèmes de particules* ont l'avantage d'être relativement faciles à implémenter. En outre, ils permettent l'ajout de dynamique et peuvent s'étendre à d'autres types de simulation (cheveux, herbes…). Cependant, d'une part, la pertinence des résultats obtenus est critiquable au niveau théorique. D'autre part, l'introduction d'équations différentielles entraîne des phénomènes d'instabilité.

[2] La simulation mène à un état d'équilibre.

5.1. Représentation par particules : Principe de base

Le principe de base de la représentation par particules est de considérer le vêtement non pas comme une surface continue mais comme un ensemble de ***nœuds***[3] reliés entre eux par des ***fibres***[4](Figure 1). Nous verrons par la suite que ces *fibres* sont virtuelles et représentent les forces qui s'exercent entre les différents *nœuds.* Avec cette représentation, les lois physiques mises en jeu sont plus simples à définir. En effet nous allons considérer que chaque *nœud* est un point possédant une masse m. Par conséquent nous utiliserons la physique du point dans la suite de notre étude.

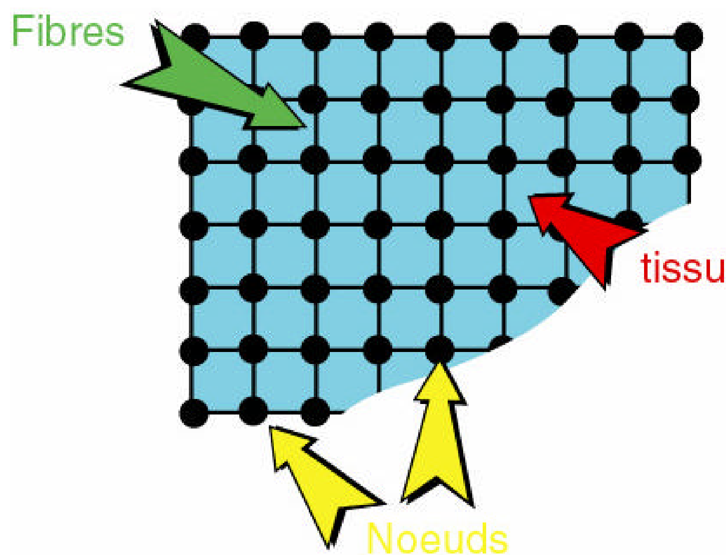


Figure 1 Discrétisation du tissu

La discrétisation du tissu peut se faire de diverses façons. Sur la Figure 1, on a simplement défini une grille représentant le tissu mais on peut effectuer des discrétisations plus complexes. Dans la suite de notre étude, nous allons présenter les différentes dispositions de *particules* et de *fibres* retenues pour la simulation.

[3] Nœud ou particule : le vêtement n'est représenté que par ces particules
[4] Fibre : définit la liaison entre deux nœuds

5.2. Les différents agencements

Dans toute la suite nous considérerons que notre tissu est un rectangle. C'est ce dernier que nous discrétiserons. Le premier agencement définit clairement trois types de fibre. Le second se base sur la complexité des connexions. Enfin nous étendrons les modèles au cas général d'un objet quelconque.

5.2.1. Agencement structurel

Cette agencement établit une structure bien définie et introduit trois types de *fibre* : les *fibres* de structure, les *fibres* de courbure dans le plan et les *fibres* de courbures dans l'espace. Ces trois *fibres* vont permettre au vêtement de résister respectivement à trois déformations différentes.

En effet, le vêtement est soumis à trois types de déformation dont deux sont coplanaires. Le comportement du vêtement va bien sûr varier suivant le type de déformation. Il sera bien évidemment beaucoup plus résistant à une déformation coplanaire. Les trois déformations dont nous tenons compte dans notre simulation sont : l'étirement, la courbure dans le plan et la courbure dans l'espace.

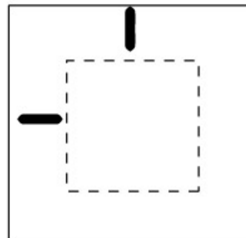


Figure 2 Etirement en hauteur et en largeur du vêtement : ''stretching''

Pour que la simulation soit réaliste, il faut que le vêtement soit très résistant à l'étirement (Figure 2) et à la courbure planaire (Figure 3).

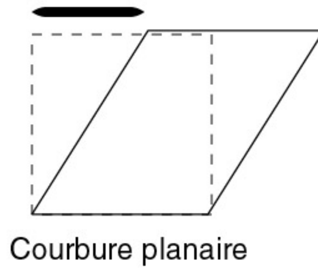


Figure 3 Courbure planaire du vêtement : ''shearing''

En revanche ce dernier sera beaucoup moins résistant aux courbures dans l'espace. En effet si un bout de tissu ne peut pas changer de taille (très peu d'étirement) en revanche il peut tout à fait se replier sur lui-même.

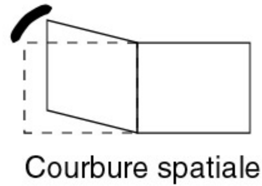


Figure 4 Courbure dans l'espace: ''bending''

Pour réagir à ces trois types de déformations, nous introduisons trois types de *fibres*. Les *fibres de structure* et de *courbure planaire* vont respectivement réagir à un étirement ou une torsion planaire du vêtement (Figure 5). Quant *aux fibres de courbure spatiale*, elles vont empêcher que le vêtement s'écrase sur lui-même (Figure 6). En effet, en ne reliant qu'un *nœud* sur deux, ces *fibres* jouent le rôle de charnière.

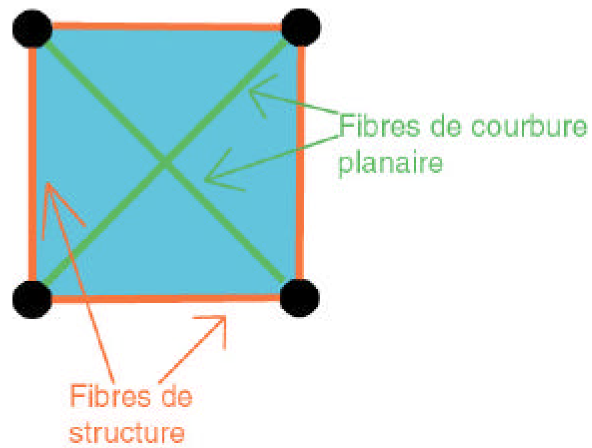


Figure 5 Fibres de structure et de courbure planaire

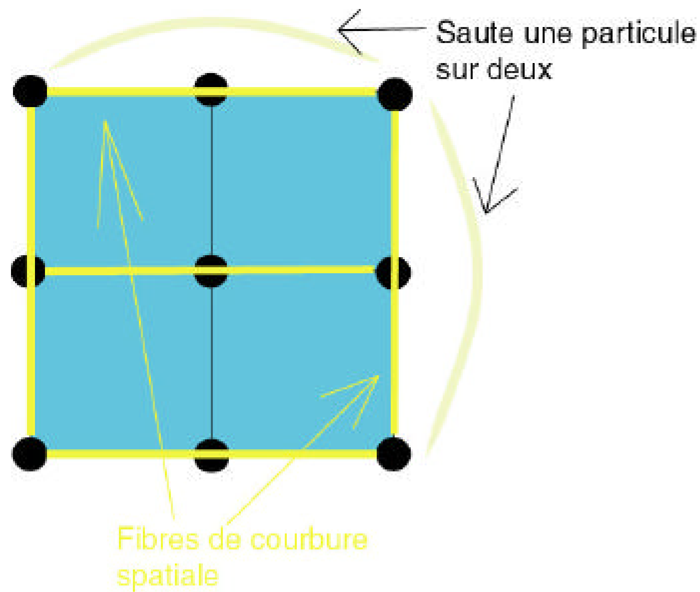


Figure 6 Fibres de courbure spatiale

Grâce à cette structure, il est possible de contrôler les réactions du vêtement aux diverses déformations et ainsi de simuler plusieurs types de tissus. Cette structure semble idéale mais elle s'adapte très mal à une utilisation plus générale dans le cadre de vêtement de forme quelconque. C'est pourquoi nous avons essayé une autre structure plus simple mais plus facilement extensible.

5.2.2. Agencement par rayon de connexion

Cette structure est bien plus simple que la précédente. Elle ne se base que sur la distance séparant un *nœud* de ses voisins. Soit n le nombre de nœuds constituant le vêtement N, i et j des nœuds quelconques appartenant à N si $d(i, j) < e$ alors il existe une fibre reliant i et j. $d(i, j)$ définit la distance entre le nœud i et le nœud j. e est la distance maximale de connexion. Si on fait varier e, on obtient des tissus plus ou moins rigides avec plus ou moins de plissures dans leur représentation. Sur Figure 7, les nœuds en rouge se trouvent à une distance 1 du nœud central, les nœuds jaunes se trouvent à une distance $\sqrt{2}$.

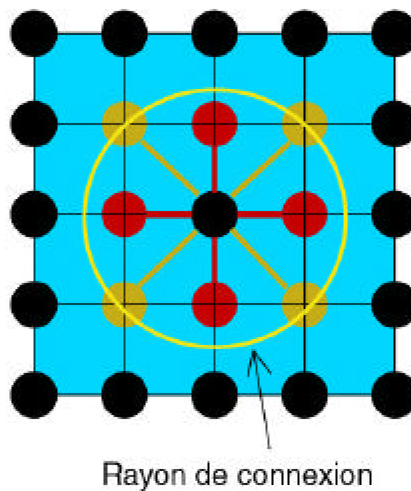


Figure 7 Connexion des nœuds suivant leu r distance

Cette modélisation peut sembler simpliste mais dans la pratique elle donne des résultats très satisfaisants. Lorsque le vêtement est une simple grille, il est facile de connaître à la construction les distances qui séparent les différents *nœuds*. En revanche, si nous souhaitons utiliser des objets fabriqués dans d'autres logiciels tels que ***3DS Max***[5], il nous faut étendre cette idée de distance.

5.2.3. Agencement par graphe

[5] 3D St udio Max de la compagnie Discreet est un logiciel de synthèse d'images largement utilisé dans l'industrie des jeux vidéos et du cinéma.

Nous allons ici présenter une modélisation qui s'adaptera à n'importe quel type d'objet. Ainsi il sera possible de fabriquer des formes souples qui pourront représenter aussi bien des vêtements que des objets ''mous''.

Supposons tout d'abord que l'objet soit représenté par ses sommets S et ses faces F. Nous ne pouvons pas comme dans le cas précédent utiliser la notion de distance entre les voisins pour obtenir l'ensemble des connexions car la répartition des *nœuds* n'est absolument pas homogène. En revanche, nous pouvons nous servir de la structure de l'objet pour essayer de calquer la modélisation précédente. En effet, tout d'abord, nous pouvons considérer que chaque sommet est un *nœud*. Ensuite la connaissance des faces et donc de tous les sommets qui appartiennent à une face va nous permettre d'utiliser un algorithme de graphe très connu : le *parcours en largeur*. En fait, il s'agit d'une version modifiée.

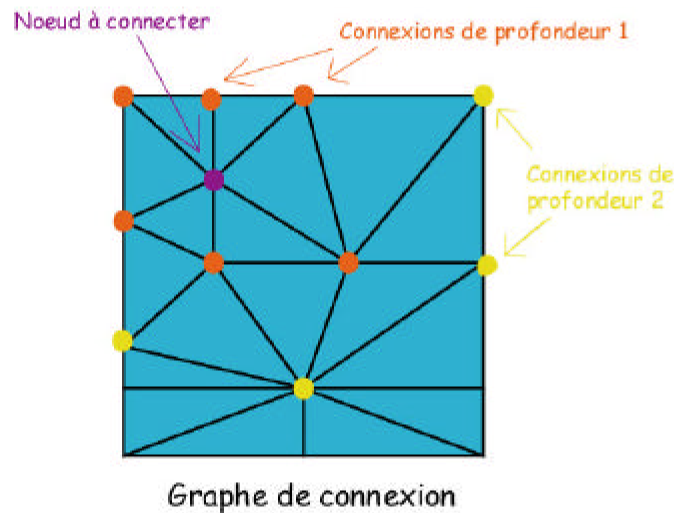


Figure 8 Graphe de connexion des nœuds

Pour chaque sommet (nœud) i de l'objet nous cherchons tous les sommets j tels qu'il existe un chemin menant de i à j et tels que la distance en nombre de sommets sur ce même chemin soit inférieure à un certain plafond (Figure 8). Sur la Figure 8, les sommets rouges sont à une distance 1 du sommet violet. Les sommets jaunes sont eux à une distance de 2.

Grâce à cet algorithme, nous possédons un moyen simple de construire des tissus à partir d'objets quelconques tout en contrôlant leur rigidité.

Maintenant que nous disposons d'une modélisation claire des vêtements, nous allons voir comment exprimer effectivement les forces que représentent les *fibres* et les autres forces prises en compte dans la dynamique de la simulation.

6. Les forces

Deux types de force vont s'exercer sur le tissu : les forces internes et les forces externes. Les forces internes sont la réalisation physique des *fibres virtuelles* définies précédemment. Les forces externes sont celles qui existent indépendamment du vêtement. Il s'agit de la gravité, du vent des forces d'amortissement dues à l'air et éventuellement des forces liées à l'action de l'utilisateur sur le vêtement (lorsque celui-ci tire sur un morceau du tissu par exemple).

6.1. Les forces internes

Les forces internes (les *fibres*) vont être modélisées par des ressorts linéaires. Les ressorts linéaires ne sont pas forcément ceux qui modélisent le mieux les fibres mais se sont ceux qui introduisent le moins d'instabilité.

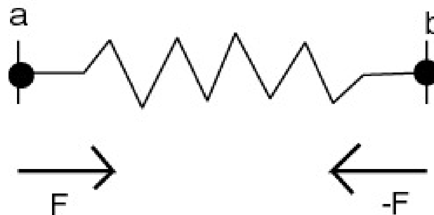


Figure 9 Schéma du ressort

La force induite par le ressort est la suivante :

$$\begin{cases} \vec{F}r_{(i,j)} = -k_{ij}(\|\vec{x}_i - \vec{x}_j\| - l_{ij}^0)\dfrac{(\vec{x}_i - \vec{x}_j)}{\|\vec{x}_i - \vec{x}_j\|} \\ \vec{F}d_{(i,j)} = kd * dt * (\vec{v}_j - \vec{v}_i) \end{cases}$$

Equation 1

Dans ces deux équations, Fr représente la force de réaction du ressort, k est la constante de raideur, l_{ij}^0 la longueur initiale du ressort, x_i et x_j le vecteur position des

nœuds i et j, Fd est la force de dissipation, v_i et v_j le vecteur vitesse des nœuds i et j, kd la constante de dissipation, dt le pas de temps. La force totale est la somme des deux forces Fr et Fd. On note aussi la relation suivante :

$$\vec{F}_{(i,j)} = -\vec{F}_{(j,i)}$$

Equation 2

Nous verrons par la suite que le principal inconvénient d'utiliser des ressorts linéaires est la surélongation qui se produit entre autres aux points d'encrage du vêtement.

6.2. Les forces externes

Pour pouvoir effectuer une simulation réaliste des vêtements, il est nécessaire de rajouter un certain nombre de forces extérieures agissant sur le vêtement telles que la gravité ou le vent.

6.2.1. La gravité

La gravité est une force simple à représenter. C'est une force dirigée vers le bas. Elle est proportionnelle à la masse. Son expression est :

$$\vec{P}_i = m_i * \vec{g}$$

Equation 3

P est le poids, g est la gravité à proprement parler et m est la masse du nœud i.

6.2.2. Force de dissipation

Cette force est souvent introduite pour donner une certaine stabilité à la simulation. Elle agit directement sur la vitesse du nœud :

$$\vec{F}diss_i = -C\vec{v}_i$$

Equation 4

C est la constante de dissipation. Cette force agit en sens opposé à la vitesse.

6.2.3. Force visqueuse

Cette force permet de simuler la résistance de l'air ou encore le vent.

$$Fvisq_i = -Cv((\vec{v}_{fluide} - \vec{v}_i).\vec{n}_i)\frac{\vec{v}_{fluide}}{\|\vec{v}_{fluide}\|}$$

Equation 5

v_{fluide} représente la vitesse du fluide, n_i est la normale au nœud i, Cv est une constante de dissipation.

La force exercée par le vent peut certes être définie comme une force visqueuse mais pour accroître le réalisme, nous avons introduit une variation sinusoïdale de son amplitude ainsi qu'une réponse plus ou moins perturbée selon la particule (il s'agit en fait d'un facteur aléatoire). Même si cette approche n'est pas physiquement correcte, elle donne des résultats visuels crédibles. De plus la simulation du vent et de l'écoulement des fluides sont des sujets très compliqués qui demanderaient une étude à eux tous seuls.

Nous ne parlerons pas ici des forces engendrées lors des collisions car elles n'ont pas été explicitement introduites comme telles. En effet, si nous avions considéré les forces de collisions, nous serions vite arrivé à des états d'instabilité.

Afin de pouvoir effectuer les calculs lors de la simulation, il nous faut une équation d'évolution qui nous permettra de connaître à chaque instant la position et la vitesse de chaque nœud.

7. Méthodes d'intégrations explicites

Nous ne traiterons ici que des méthodes d'intégrations explicites. Les méthodes d'intégrations dîtes implicites seront présentées dans une autre partie. Avant de parler des méthodes elles-mêmes, il nous faut poser les équations différentielles. Pour cela nous avons besoin de la relation fondamentale de Newton :

$$\ddot{x} = \frac{1}{m} F$$

Equation 6

où m représente la masse, F la somme des forces et $\ddot{x}$ l'accélération. C'est une équation différentielle du second ordre. Nous allons la diviser en deux équations différentielles du premier ordre.

Nous introduisons tout d'abord la variable $v = \dot{x}$ qui représente la vitesse. Ainsi nous pouvons réécrire l'équation précédente :

$$\dot{v} = \frac{1}{m} F$$

Equation 7

ce qui nous donne un système à deux équations :

$$\begin{cases} \dot{v} = \frac{1}{m} F \\ v = \dot{x} \end{cases}$$

Equation 8

De la première équation on pourra déduire v et de la seconde x. Il nous faut cependant trouver une méthode numérique qui puisse nous donner effectivement la solution.

7.1. Méthode d'Euler

La méthode d'Euler s'appuie sur le développement limité au premier ordre de $v(t+dt)$:

$$v(t+dt) = v(t) + dt * \dot{v}(t) + O(dt^2)$$

Equation 9

De ce développement limité on en déduit l'expression de $v(t+dt)$:

$$v(t+dt) = v(t) + dt * \dot{v}(t)$$
$$\Rightarrow v(t+dt) = v(t) + \frac{dt}{m} F(t)$$

Equation 10

De même pour l'expression de $\dot{x}(t)$:

$$x(t+dt) = x(t) + dt * \dot{x}(t)$$
$$\Rightarrow x(t+dt) = x(t) + dt * v(t)$$

Equation 11

Grâce à la méthode d'Euler, nous avons maintenant un moyen numérique de calculer à chaque instant en fonction de l'instant précédent la vitesse et la position de chaque *nœud*. Il faut cependant émettre quelques réserves sur cette méthode puisqu'elle devient vite instable si le pas de temps est trop grand.

7.2. Méthode d'Euler modifiée

Une variante de la méthode d'Euler permet d'obtenir une plus grande stabilité. Expérimentalement, on peut multiplier le pas de temps par un facteur 10 par rapport à la méthode d'Euler classique. La seule modification à effectuer est la suivante :

$$x(t+dt) = x(t) + dt * v(t) \rightarrow x(t+dt) = x(t) + dt * v(t+dt)$$

Equation 12

L'astuce consiste simplement à utiliser la vitesse exprimée au temps $t + dt$ et non t.

7.3. Méthode du ''Mid-Point''

Cette méthode permet d'obtenir une plus grande stabilité mais elle est plus coûteuse en temps que la précédente. En fait il s'agit de la méthode précédente à l'ordre 2 Posons tout d'abord l'équation suivante :

$$v(t+dt) = v(t) + dt * \dot{v}(t) + \frac{dt^2}{2}\ddot{v}(t) + O(dt^3)$$

Equation 13

Il nous faut trouver un moyen de calculer $\ddot{v}(t)$ puisque en général on ne connaît pas son expression.

Sachant que $\dot{v}(t) = f(v(t))$, on en déduit $\ddot{v}(t) = \frac{\partial f}{\partial v}\dot{v}(t) = ff'$. Effectuons un second développement limité sur une variation de vitesse :

$$f(v(t)+dv) = f(v(t)) + dv * f'(v(t)) + O(dv^2)$$

Equation 14

Maintenant on choisit dv tel que $dv = \frac{dt}{2}f(v(t))$. On introduit cette nouvelle expression dans l'équation précédente et on obtient :

$$f(v(t)+\frac{dt}{2}f(v(t))) = f(v(t)) + \frac{dt}{2}f(v(t))f'(v(t)) + O(dt^2)$$
$$\Rightarrow f(v(t)+\frac{dt}{2}f(v(t))) = f(v(t)) + \frac{dt}{2}\ddot{v}(t) + O(dt^2)$$

Equation 15

On multiplie par dt de chaque côté et on obtient :

$$\frac{dt^2}{2}\ddot{v}(t) + O(dt^3) = dt(f(v(t)+\frac{dt}{2}f(v(t))) - f(v(t))$$

Equation 16

On remplace l'expression de $\frac{dt^2}{2}\ddot{v}(t)$ dans l'Equation 13 par sa nouvelle valeur et on trouve :

$$v(t+dt) = v(t) + dt(f(v(t) + \frac{dt}{2} f(v(t))))$$

Equation 17

Cette méthode nous autorise une précision en $O(dt^3)$. En fait nous pouvons étendre cette méthode à un ordre supérieur.

7.4. Runge-Kutta

La méthode Runge-Kutta se base sur le même principe que la méthode précédente et permet d'obtenir une précision en $O(dt^5)$. Nous ne donnerons pas toutes les étapes nécessaires aux calculs mais seulement les résultats intéressants :

$$\begin{aligned}
&k1 = dt * f(v(t), t) \\
&k2 = dt * f(v(t) + \frac{k1}{2}, t + \frac{dt}{2}) \\
&k3 = dt * f(v(t) + \frac{k2}{2}, t + \frac{dt}{2}) \\
&k4 = dt * f(v(t) + k3, t + dt) \\
&v(t+dt) = v(t) + \frac{1}{6}k1 + \frac{1}{3}k2 + \frac{1}{3}k3 + \frac{1}{6}k4
\end{aligned}$$

Equation 18

Cette dernière requiert quatre fois plus de temps de calcul que la méthode d'Euler. Même si le gain en précision est important, elle ne s'avère pas très adaptée à une simulation temps réelle.

Avec l'ensemble de ces éléments nous disposons d'assez de moyen pour simuler des vêtements en faisant abstraction des collisions. Cependant, on note un problème du à l'utilisation de ressorts linéaires : la surélongation des fibres notamment aux points d'attaches de ce dernier. C'est pourquoi il a été nécessaire d'introduire des méthodes susceptibles de corriger ce problème.

8. Surélongation

Nous allons au cours de cette partie présenter plusieurs méthodes de correction d'élongation. Le problème de la surélongation provient de l'utilisation de ressorts linéaires. En effet, rien ne limite l'élongation maximale des *fibres*. Ce défaut donne des résultats visuels qui ne sont pas acceptables et que nous devons corriger (Figure 10). On sait que les tissus admettent des déformations qui représentent un changement d'élongation des fibres de moins de 10% voire 5% de leur longueur initiale.

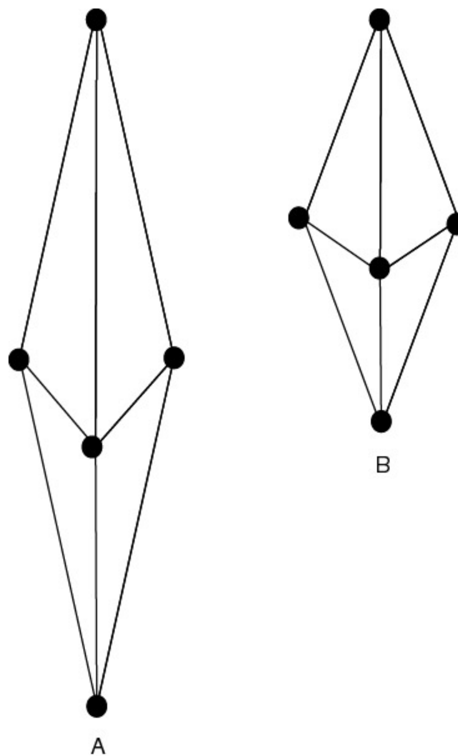


Figure 10 En A, les fibres sont trop étirées alors que l'on aimerait le comportement de B

Figure 11 Comportement idéal des fibres du vêtement en bleu ; comportement actuel en rouge

8.1. Choix de la constante de raideur

La première idée qui nous vient à l'esprit est de choisir une constante de raideur des ressorts qui soit suffisamment grande pour résister aux différentes tensions. Cette solution s'avère vite inapplicable en réalité. En effet, les équations différentielles mènent à des instabilités lorsqu'elles sont résolues par les méthodes explicites si la constante de raideur est importante. Pour palier ce problème il est alors nécessaire de réduire le pas de temps jusqu'à l'obtention d'un système stable. Or expérimentalement le pas de temps adéquate est trop petit. La simulation perd son caractère temps réel. De plus, une constante de raideur trop grande donne un comportement trop rigide au vêtement.

En conclusion, si nous souhaitons utiliser cette solution, il nous faut utiliser d'autres méthodes d'intégration : *les méthodes implicites*. Nous verrons ces méthodes dans un prochain chapitre (chapitre 9).

8.2. Correction de position

Cette méthode introduite par ***PROVOT*** consiste à modifier directement la position de chaque nœud, une fois l'intégration effectuée. La correction appliquée dépend du statut du nœud : nœud contraint ou non contraint. L'avantage de cette méthode est qu'elle ne provoque aucune instabilité puisqu'elle n'intervient pas dans le processus d'intégration.

Si aucun des nœuds n'est contraint de rester fixe (tous les déplacements sont autorisés), la correction s'effectue comme sur la Figure 12.

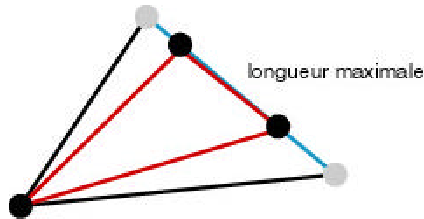


Figure 12 On déplace les deux nœuds dans la direction de leur liaison telle que la distance entre les deux soit égale à la distance maximale d'élongation

Dans le cas où l'un des deux nœuds est contraint à garder sa position, la correction s'effectue comme sur la Figure 13.

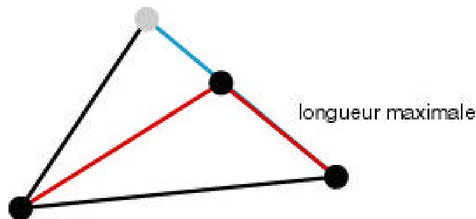


Figure 13 On déplace un seul des deux nœuds dans la direction de liaison telle que la distance soit égale à la distance maximale

Cette méthode donne de bons résultats dans certaines conditions seulement. En fait tout dépend des points d'attache. Dans la pratique, on est obligé d'effectuer plusieurs itérations en espérant que l'algorithme va converger : Aucune preuve formelle de convergence n'a été établie à ce jour. La nécessité d'itérer le processus s'explique par le caractère local de la correction appliquée. En d'autres termes lorsqu'on corrige une des distances, on risque d'introduire de nouvelles surélongations (Figure 14).
Lorsqu'on tient compte des collisions, cette méthode s'avère inefficace et devient même inutilisable (les collisions engendrent de nombreux points contraints dans leur mouvement). Il n'est donc plus possible d'utiliser cette méthode dans une simulation complète des vêtements.

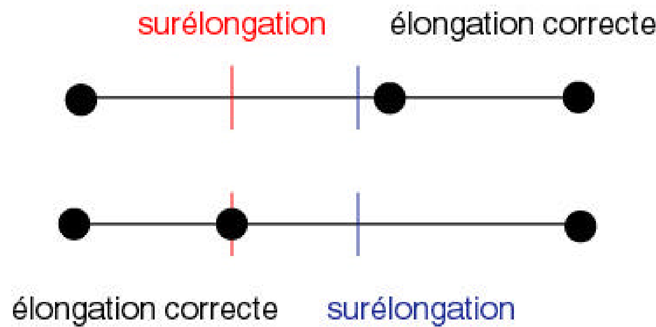


Figure 14 Problème de convergence

Il était donc nécessaire de trouver une méthode qui puisse satisfaire l'ensemble des situations.

8.3. Modification de la vitesse

Toutes les méthodes qui consistent à corriger la position des nœuds ne donnent pas les résultats escomptés. Par conséquent il a été nécessaire de trouver des méthodes travaillant sur une autre caractéristique. Quelques articles proposaient de modifier non pas la position mais la vitesse des nœuds après le processus d'intégration. Malheureusement ceux-ci restaient très vagues sur la correction réellement apportée. Nous avons donc essayé de mettre en place une méthode efficace travaillant sur les vitesses.

Si la longueur d'une fibre dépasse la longueur maximale autorisée alors la vitesse des nœuds est modifiée ainsi :

- Soit v la vitesse du nœud, on écrit $\vec{v} = \vec{v}_f + \vec{v}_r$ où $\vec{v}_f$ est la vitesse projetée dans la direction de la fibre et $\vec{v}_r$ la vitesse restante.
- Si la longueur maximale de la fibre a été dépassée alors la vitesse $\vec{v}_f$ est annulée dans le cas présenté en Figure 15 : La vitesse $\vec{v}_f$ tend à éloigner plus encore les deux nœuds. Sinon celle-ci est conservée.

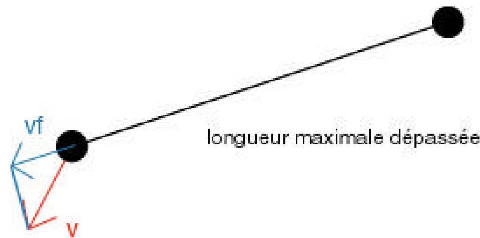


Figure 15 Correction de vitesse lorsque la longueur maximale est dépassée

- Si la longueur de la fibre est plus petite que la longueur minimale autorisée alors la vitesse $\vec{v}_f$ est annulée dans le cas présenté en Figure 16 : La vitesse $\vec{v}_f$ tend à rapprocher plus encore les deux nœuds. Sinon celle-ci est conservée.

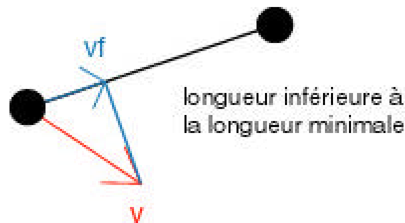


Figure 16 Correction de vitesse lorsque la longueur de la fibre est plus petite que la longueur minimale autorisée

Cette méthode se révèle très efficace et permet de limiter grandement l'élongation des ressorts. Bien évidemment, le pas de temps ne doit pas être trop important car la correction intervient après une éventuelle élongation. Un pas de temps trop important conduirait à des élongations trop importantes malgré la correction.

Il est possible et même conseillé de limiter l'élongation de certains ressorts seulement. En effet, les ressorts qui empêchent les courbures spatiales par exemple n'ont pas besoin d'être limités. L'angle de courbure peut prendre de grandes valeurs.

Toutes les méthodes présentées dans cette partie ne témoignent pas d'une réalité physique et peuvent être regroupées sous l 'appellation de méthodes post correctives. Comme expliqué précédemment, il est possible de résoudre les équations différentielles pour des constantes de raideurs grande en utilisant des méthodes d'intégration implicites.

9. Méthodes d'intégration implicite

Au cours de cette partie, nous allons reprendre les équations précédentes et en donner une formulation différente. Ceci nous permettra de trouver de nouvelles méthodes d'intégration : les méthodes implicites.

9.1. Reformulation des équations

Nous allons dans un premier temps reformuler les équations vues au cours de la présentation des méthode explicites.

$$\begin{cases} \vec{v}(t+dt) = \vec{v}(t) + \frac{dt}{m}\vec{F}(t) \\ \vec{x}(t+dt) = \vec{x}(t) + dt * \vec{v}(t) \end{cases}$$

Equation 19 Equations précédemment définies

Nous allons exprimer les forces non plus au temps t mais au temps $t+dt$. Grâce à cette modification, on peut assurer que le champ de force est cohérent avec la position.

$$\begin{cases} \vec{v}(t+dt) = \vec{v}(t) + \frac{dt}{m}\vec{F}(t+dt) \\ \vec{x}(t+dt) = \vec{x}(t) + dt * \vec{v}(t+dt) \end{cases}$$

Equation 20 Equations reformulées

Pour résoudre l'Equation 20, il faut cependant calculer $\vec{F}(t+dt)$. En général, on ne connaît pas l'expression formelle de $\vec{F}(t+dt)$. Il faut donc trouver un moyen de l'approximer.

$$F(t+dt) = F(t) + \frac{\partial F}{\partial x}\Delta x$$

Equation 21 Expression de F(t+dt) : approximation au premier ordre

Pour les ressorts, cette approximation au premier ordre est exacte.

Posons :

$$H = \frac{\partial F}{\partial x} \text{ et}$$

$$\Delta x = (v(t) + \Delta v)dt$$

On introduit l'Equation 21 dans l'Equation 20, et on obtient :

$$(I - \frac{dt^2}{m} H)\Delta v = (F(t) + dtHv(t))\frac{dt}{m}$$

C'est un système linéaire qu'il faut résoudre pour Δv. Deux possibilités s'offrent à nous, l'une ne résout pas directement le système et l'autre exploite les propriétés de ce système pour optimiser sa résolution.

9.2. Résolution indirecte

La méthode présentée ici a été proposée par Mathieu DESBRUN, Mark MEYER et Alan H. BARR. Elle consiste à séparer la partie linéaire de la partie non linéaire dans l'expression de la force des ressorts.

$$F_{(i,j)} = -k_{ij}(x_i - x_j) + k_{ij} l_{ij}^0 \frac{(x_i - x_j)}{\|x_i - x_j\|}$$

La matrice H pour la partie linéaire est facile à calculer et ne change pas au cours de la simulation :

$$\begin{cases} H_{ij}^{linear} = k_{ij} & i \neq j \\ H_{ii}^{linear} = -\sum_{i \neq j} k_{ij} \end{cases}$$

On pose :

$$W = (I - \frac{dt^2}{m} H)^{-1}$$

La partie non linéaire représente un changement de direction de la force (même intensité) et pourra par conséquent être corrigé ultérieurement. Nous ne détaillerons pas tous les calculs. L'algorithme final peut s'écrire en pseudo code :

- Calculer W.
- Calculer F.
- Filtrer F par W : $F_{filtrée} = FW$
- Apporter le terme correctif pour préserver le moment angulaire. Le moment linéaire est préservé intrinsèquement :

$$\mathbf{d}T = \sum_{i=1}^{n} (X_G - x_i) \times F_i^{filtrée}$$

où n est le nombre de nœuds et X_G est le centre de gravité du vêtement.

$$R_i^{correction} = (X_G - x_i) \times \frac{\mathbf{d}T dt^2}{m}$$

Pour des vêtements de petite taille cette méthode est très intéressante. Elle permet en effet de régler des coefficients de raideur élevés sans introduire d'instabilité. En revanche lorsque le nombre de nœuds devient important, l'inversion de la première matrice est très coûteuse en temps de calcul et la méthode devient inutilisable. Nous n'avons pas trouvé de moyen pour réduire le coût de cette première inversion.

9.3. Résolution directe

On peut remarquer que la matrice H comporte très peu de valeurs non nulles. Cette propriété provient des forces qui existent entre les nœuds. En général, un nœud est relié à très peu d'autres nœuds. Par conséquent, sur une même ligne seules les colonnes qui témoignent de l'existence d'une force entre le nœud i et le nœud j ont une valeur non nulle. Cette constatation nous amène à utiliser un algorithme de gradient conjugué pour résoudre ce système. En effet, le gradient conjugué est très efficace lorsque la matrice est creuse[6].

[6] Matrice creuse : La matrice comporte très peu de valeurs non nulles.

Nous avons donc non seulement implémenté les matrices creuses mais aussi l'algorithme du gradient conjugué. L'implémentation des matrices creuses était nécessaire car on ne pouvait pas stocker directement des matrices de taille N*N avec N pouvant atteindre même pour un vêtement de petite taille plusieurs milliers de lignes et de colonnes. Le détail de l'implémentation ne sera pas expliquer ici. Pour des informations supplémentaires, il est préférable de se référer au code source.

10. Gestion des collisions

Nous allons présenter au cours de ce chapitre la manière dont les collisions sont gérées dans notre simulation. Le processus de gestion des collisions se divise en deux parties : la détection des collisions qui consiste à déterminer si le vêtement entre en collision avec un objet de la scène ; la réponse aux collisions qui précisent la façon dont le vêtement va réagir lors d'une collision. Enfin nous présenterons la perspective d'une nouvelle méthode qui utilise directement les capacités des nouvelles cartes graphiques.

10.1. Détection des collisions

Nous verrons dans un premier temps la détection des collisions avec trois types de primitives : la sphère, le plan et le cylindre. Puis nous nous intéresserons à la détection de collision avec des objets quelconques.

Dans les deux cas, nous ne tiendrons compte que des tests mettant en jeu chaque nœud pris indépendamment l'un de l'autre. Nous ne nous occuperons pas des collisions éventuelles entre une fibre et un objet. Ce choix a été fait d'une part pour ne pas alourdir les calculs ; d'autre part il suffit d'augmenter le nombre de nœuds pour diminuer ces éventuelles collisions (Figure 17).

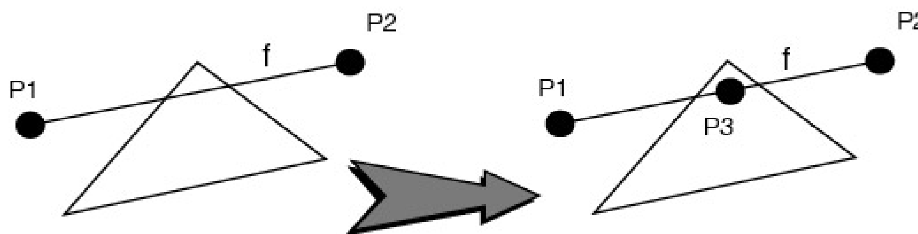


Figure 17 Collision segment triangle. Dans le premier cas, aucune collision n'est détectée. Dans le second cas le rajout du nœud P3 corrige l'erreur.

10.1.1. Collision avec une sphère

Le cas de la sphère est le plus simple. En effet, pour savoir si un nœud du vêtement entre en collision avec une sphère, il suffit de calculer la distance qui sépare le nœud du centre de la sphère et de la comparer au rayon de la sphère (Figure 18) :

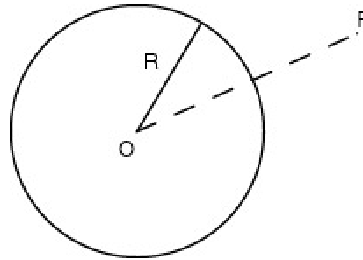


Figure 18 Collision avec une sphère

Si et seulement si $\|P-O\|<R$ alors le nœud P est en collision avec la sphère.

10.1.2. Collision avec un plan

La détection d'une collision avec un plan ne s'avère pas plus compliquée (Figure 19).

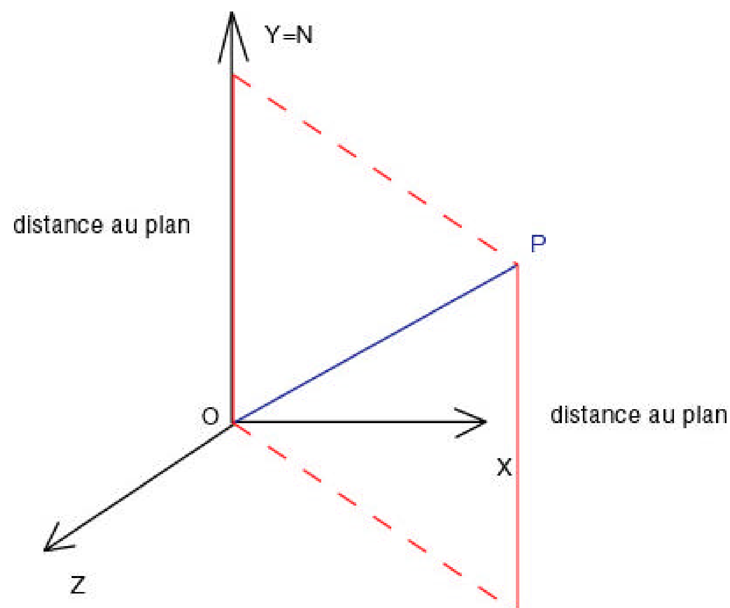


Figure 19 Collision avec un plan

Soit un plan de normale extérieure unitaire N, alors si $(P-O)\bullet N <= 0$, il y a collision. En effet le produit scalaire avec la normale nous donne la distance algébrique au plan. Si celle-ci est négative alors il y a collision.

10.1.3. Collision avec un cylindre

La détection d'une collision avec un cylindre est celle qui parmi les trois demande le plus de calcul. Le cylindre que nous utiliserons ici est complété par deux hémisphères à ses extrémités. Cet ajout permet d'éviter les problèmes de non continuité au bord du cylindre.

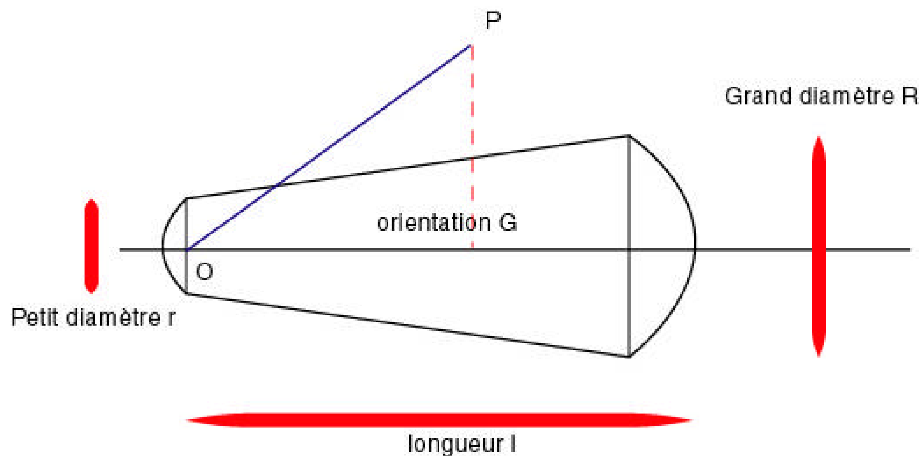


Figure 20 Collision avec un cylindre

Nous devons calculer la projection du vecteur (P-O) sur la droite d'orientation G du cylindre ainsi que la distance de P à cette même droite. Il y a collision si les deux tests suivants sont satisfaits (On suppose ici que $r = R$):

- On pose $D1 = (P-O)\bullet G$. On doit avoir $-r < D1 < (l+R)$.
- On pose $D2 = \left\|(P-O) - D1 * G\right\|$. On doit avoir $0 < D2 < r$

L'utilisation de primitives semble être à première vue assez limitatif. Mais d'une part, elles permettent de construire par exemple des squelettes pour les personnages assez facilement. D'autre part le coût de calcul est très faible et ne nuit pas aux performances de la simulation.

Toutes fois, pour ne pas limiter notre simulation nous avons implémenter une gestion des collisions pour des objets de forme quelconque définis par leur faces.

10.1.4. Collision avec des objets de forme quelconque

Comme précisé plus haut, la gestion des collisions est la partie qui consomme le plus de temps processeur. Si dans le cas des primitives, le coût reste raisonnable, il devient très vite prohibitif dans le cas d'objets définis par plusieurs milliers de faces. Si on suppose N le nombre de nœuds, F le nombre de faces de l'objet, la complexité est alors en N*F. Pour espérer atteindre des temps de calcul acceptables, il a fallu mettre en place des stratégies qui permettent de réduire la complexité. Nous avons donc implémenté un algorithme d'*octree* :

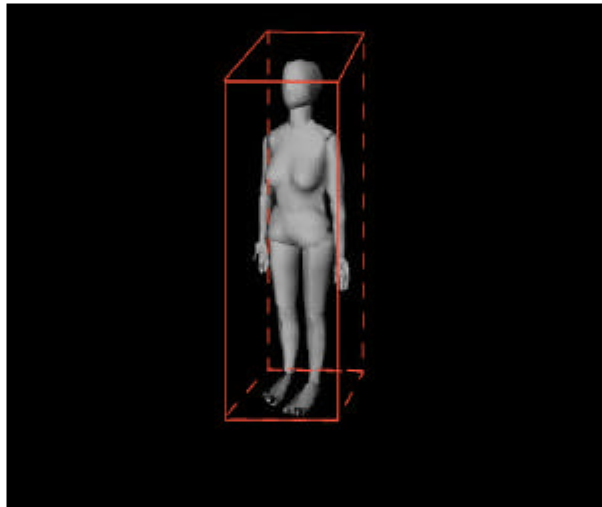


Figure 21 Boîte englobante de l'objet

1. Initialisation de la boîte englobante
2. On divise la boîte englobante en huit boîtes plus petite de tailles identiques.
3. On divise à nouveau chaque boîte en huit.
4. On répète l'étape 3 jusqu'à ce que l'on atteigne la profondeur désirée.

Lorsqu'on atteint la profondeur désirée on arête l'algorithme. On obtient donc un arbre dont chaque nœud possède huit fils (Figure 22).

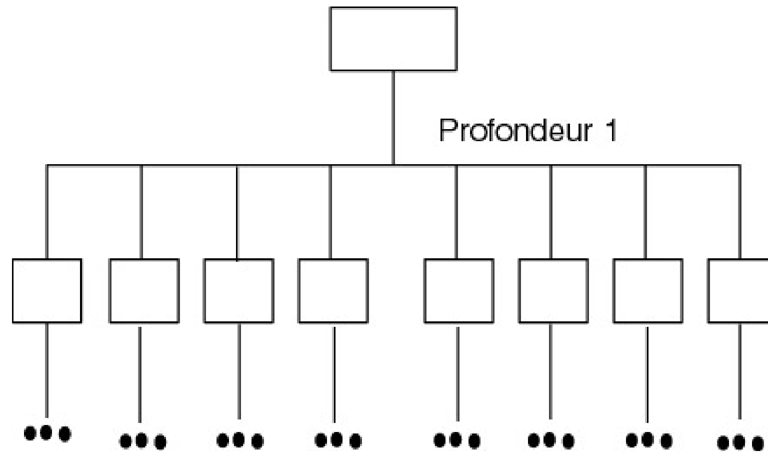


Figure 22 L'*octree* une fois l'algorithme terminé

On associe alors à chaque feuille de l'arbre la liste des faces contenues dans la boîte englobante associée à la feuille.

Nous avons choisi de considérer qu'une face appartient à une feuille si sa boîte englobante intersecte la boîte englobante de la feuille. Ce choix nous permet d'être sûr que toutes les faces soient associées au moins une fois à une feuille de l'arbre. Les redondances que cette technique implique ne sont pas réellement un problème.

Maintenant que l'arbre est construit, nous allons expliquer son utilisation. Pour chaque nœud du vêtement, au lieu de tester s'il y a une collision avec une des faces de l'objet, on commence par tester s'il y a une collision avec la première boîte englobante de l'arbre puis on descend dans la hiérarchie jusqu'à éventuellement arriver sur une feuille (Figure 23). Dans ce cas, on teste uniquement les faces associées précédemment à la feuille lors de la construction de l'arbre.

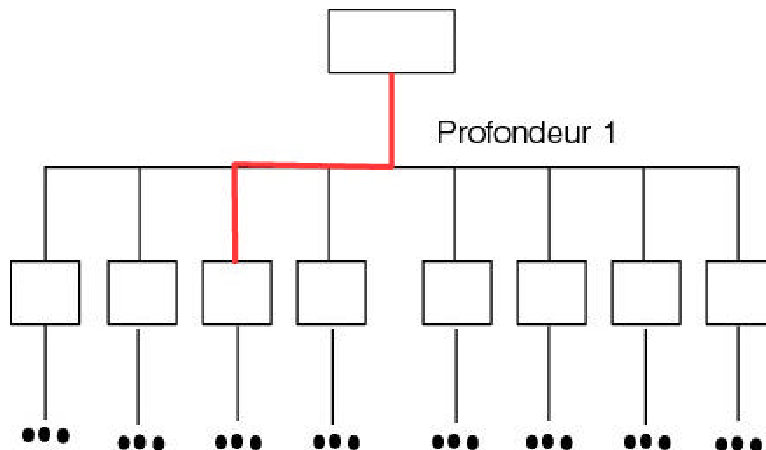


Figure 23 Le chemin rouge dans l'arbre indique les tests de collision successifs. A la fin peu de faces restent à tester.

L'optimisation obtenue par l'utilisation de l'*octree* s'avère essentielle. En effet, sans celle-ci il est impossible d'obtenir une simulation temps réel.

Cependant, malgré cette optimisation, deux problèmes se posent. Les collisions sont testées uniquement entre les nœuds du vêtement et les faces ; les temps de calculs deviennent critiques pour des objets possédant plusieurs dizaines de millier de faces.

C'est pourquoi nous avons essayé de trouver d'autres techniques plus efficaces. Jusqu'ici, nous n'avons pas parlé de la carte graphique utilisée. En effet, tous les calculs de la simulation sont effectués par le processeur principal et non par celui de la carte graphique. Notre idée a donc été de chercher une solution qui utiliserait les ressources de la carte graphique et particulièrement ses capacités en 3D.

10.1.5. Détection avancée des collisions

Nous tenons à préciser que cette technique est à l'état d'étude et qu'elle n'a pas fait l'objet d'une implémentation. L'idée principale est d'utiliser le ''z-buffer'' de la carte graphique pour dans un premier temps décider des objets en collision puis éventuellement calculer la réponse.

Il nous faut tout d'abord calculer deux images de profondeur (''z-buffer'', Figure 24). La position de la caméra pour les deux images est donnée par la Figure 25. On obtient pour ainsi dire deux cartes qui vont nous permettre par la suite de savoir si un nœud du vêtement est en collision avec la sphère.

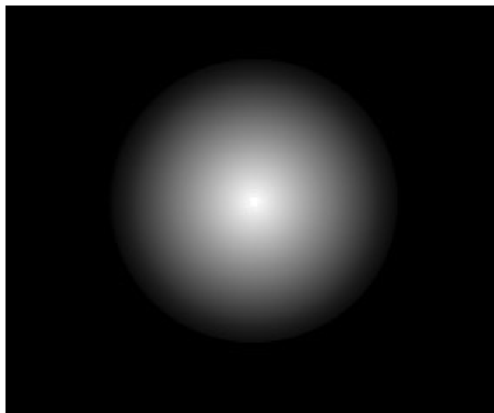


Figure 24 Le ''z-buffer'' de la sphère

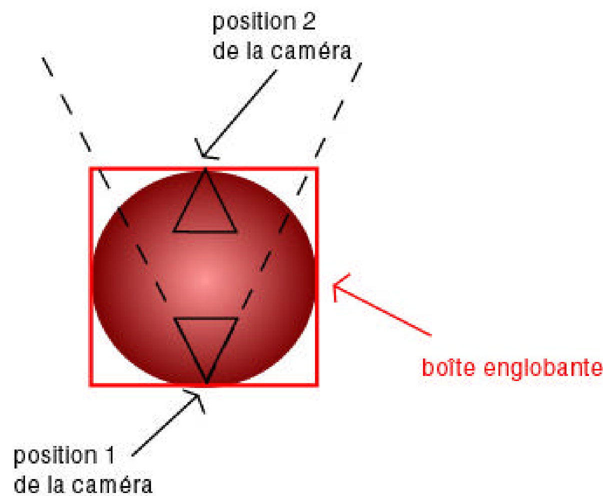


Figure 25 Position des caméras en vue de dessus

Les étapes pour détecter la collision d'un nœud sont les suivantes :

1. Déterminer quelle carte utilisée : Si le nœud se trouve dans la moitié supérieure (en vue de dessus) de la boîte englobante de l'objet alors on utilise l'image de la caméra 2(Figure 25) sinon on utilise l'autre.
2. Faire un rendu du vêtement et comparer la profondeur de chaque nœud avec la carte choisie précédemment.
3. Si la profondeur du nœud est inférieure à la profondeur correspondante sur la carte choisie alors il y a collision.

Cette technique est très efficace. En effet, elle ne dépend pas du nombre de faces de l'objet et les tests effectués sont beaucoup plus précis que dans les techniques précédentes. Les profondeurs sont interpolées directement par le hardware de la carte. Cependant, elle nécessite le stockage de deux cartes (images) calculées au tout début de la simulation et impose quelques contraintes sur les objets de collision. L'utilisation de deux cartes seulement contraint l'objet à être plus ou moins symétrique. C'est une contrainte forte. Mais dans le cas d'un corps humain elle est presque respectée. Par conséquent la technique reste utilisable.

On peut aussi concevoir d'utiliser le hardware de la carte pour obtenir une carte des normales de l'objet. En effet, il suffit pour cela au moment du rendu de remplacer la

couleur d'un vertex[7] par sa normale. On verra dans la partie suivante que la réponse à une collision se calcule en partie grâce à la normale.

Cette technique semble prometteuse mais elle doit encore faire l'objet d'études. L'idéal serait de pouvoir déterminer non pas pour chaque nœud du vêtement mais pour chaque point du vêtement s'il y a collision. On éviterait les erreurs de collision expliquées plus haut (Figure 17). Une autre perspective serait d'utiliser cette technique non pas dans un but d'optimisation des temps de calculs mais plutôt d'affichage toujours pour palier le problème précisé en Figure 17.

10.2. Réponse aux collisions

La réponse aux collisions n'a pas été la partie la plus délicate de la simulation et notre premier choix s'est révélé très bien adapté. Voici comment calculer la réponse à une collision :

1. On recale la particule au niveau de la surface (Figure 26). Pour cela on change la position du nœud par conséquent on doit recalculer sa vitesse. Il suffit d'inverser la relation d'Euler établie précédemment.

$$\vec{x}(t+dt) = \vec{x}(t) + dt * \vec{v}(t) \Rightarrow \vec{v}(t) = \frac{1}{dt}(\vec{x}(t+dt) - \vec{x}(t))$$

Equation 22

2. On modifie la vitesse du nœud en réponse à la collision :

$$\vec{v}'(t) = C * \vec{v}_t(t) + N * \vec{v}_n(t)$$

Equation 23

$\vec{v}'(t)$ représente la nouvelle vitesse, C et N sont des coefficients respectivement de frottement et de « rebond », $\vec{v}_t$ la vitesse tangentielle et $\vec{v}_n$ la vitesse normale. En général, on choisit $N = 0$ car le vêtement ne rebondit pas sur l'objet de collision.

[7] Vertex : nom informatique pour désigner un point 3D en synthèse d'image dans l'espace objet.

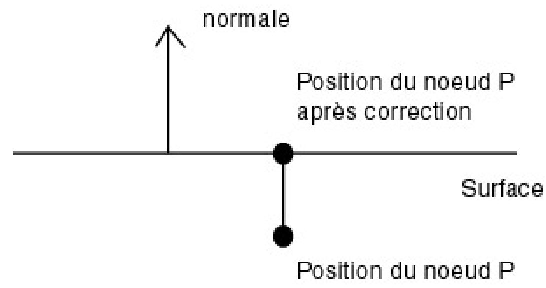


Figure 26 Réponse à une collision

Notons que la réponse aux collisions ne s'effectue pas par l'ajout de forces. Cette alternative aurait entraîné des instabilités dans le processus d'intégration. De plus les résultats obtenus avec la méthode choisie sont très satisfaisants.

(la partie sur le rendu est exclusivement écrite en français à cause du grand nombre de termes techniques toujours anglais utilisés, c'est pourquoi cette partie sera plus intelligible et mieux rédigée entièrement en anglais plutôt qu'avec un mauvais 'franglais')

(the rendering part is entirely written in English because of the big amount of technical English words, so the document is much better understandable in full English rather than in a bad mixed French/English)

11. The Rendering Engine

In this topic we will present the behaviour of the rendering engine we use for the rendering of all objects. The use of a specialized rendering engine became obvious, when we had to deal with issues due to the object graphic representation. We will introduce how the engine provides a robust, customizable solution for most rendering issues and how it is used for smooth objects.

11.1. Common Rendering Issues

11.1.1. Object Coherency

The first evident problem when programming is the separation between the pure object behaviour (in our case, the smooth object mdeling) and its graphic representation. The easiest way when dealing with objects that can be drawn on the screen is to let them handle the way they are rendered. Nonetheless, this pattern is not appropriated for at least two reasons:

- the physically-based modeling must be independant of the graphic representation
- a given object has a limited way to be drawn

The problem is so not only a design issue but also a robust architecture mistake that would severely slow the development process and the software adaptability as well.

11.1.2. Graphic Effects and Genericity

An interesting feature of a good graphic engine is to be able to render a given object in several, different ways. Should we do this, we would have to

rewrite the rendering procedure with parameters depending of the graphic effect we want.

The problem is, we usually do only minor changes, and the basic display structure of the object - that is to say, the tesselation, consisting in triangles most of the time - remains the same. We will see how a generic solution can be adopted for changing the rendering mode while the topology of the object that keeps unchanged.

11.1.3. Customization of Object Rendering

As an object can be graphically representated, every effect could be applied, but generally, the rendering is globally from one object to another, according to its nature. It usually consists in color and texture changes, but it is rather difficult to alter the object rendering without rewriting all the rendering procedure.

This problem requires a way to customize object rendering with no limitations, allowing performing graphic effects that would be much more difficult otherwise.

11.1.4. Multiplatform Robustness

Another common issue for graphics developers is the adapting to the wide range of graphic adapters available today (*n*Vidia is currently one of the most famous constructors of video card, but not the only one). A state-of-the-art graphic application *must:*

3. take advantage of the latest graphic device features (OpenGL extensions...)
4. adapt to older video cards (at least up to 3 years old) for portability

Those two guidelines encounter each other: recent graphic innovations needed for keeping being at the cut-edge technology are often not supported by most video cards, forcing the application to handle every case.

11.1.5. API use and Specific Optimizations

Among the graphic librairies that are usually used today, **Direct3D** and **OpenGL** represent more than 90% the APIs for real-time 3D software, the last 10% are shared by proprietary and software rendering engines. We use OpenGL for our project because of its simplicity and platform independance.

What must be kept in mind is that each API has its own advantages and weaknesses. Not only the rendering implementation of any 3D application should be low-level 3D API independant (being able to switch easily between from OpenGL to Direct3D for instance) but also the implementation should be able to enhance global performances, masking the driver implementation weaknesses. These two tasks require a thorough knowledge of the API, and solid software architecture for the hardware independancy (and external use as well, for developers that do not know the rendering part in detail).

11.2. Global Rendering Engine Architecture

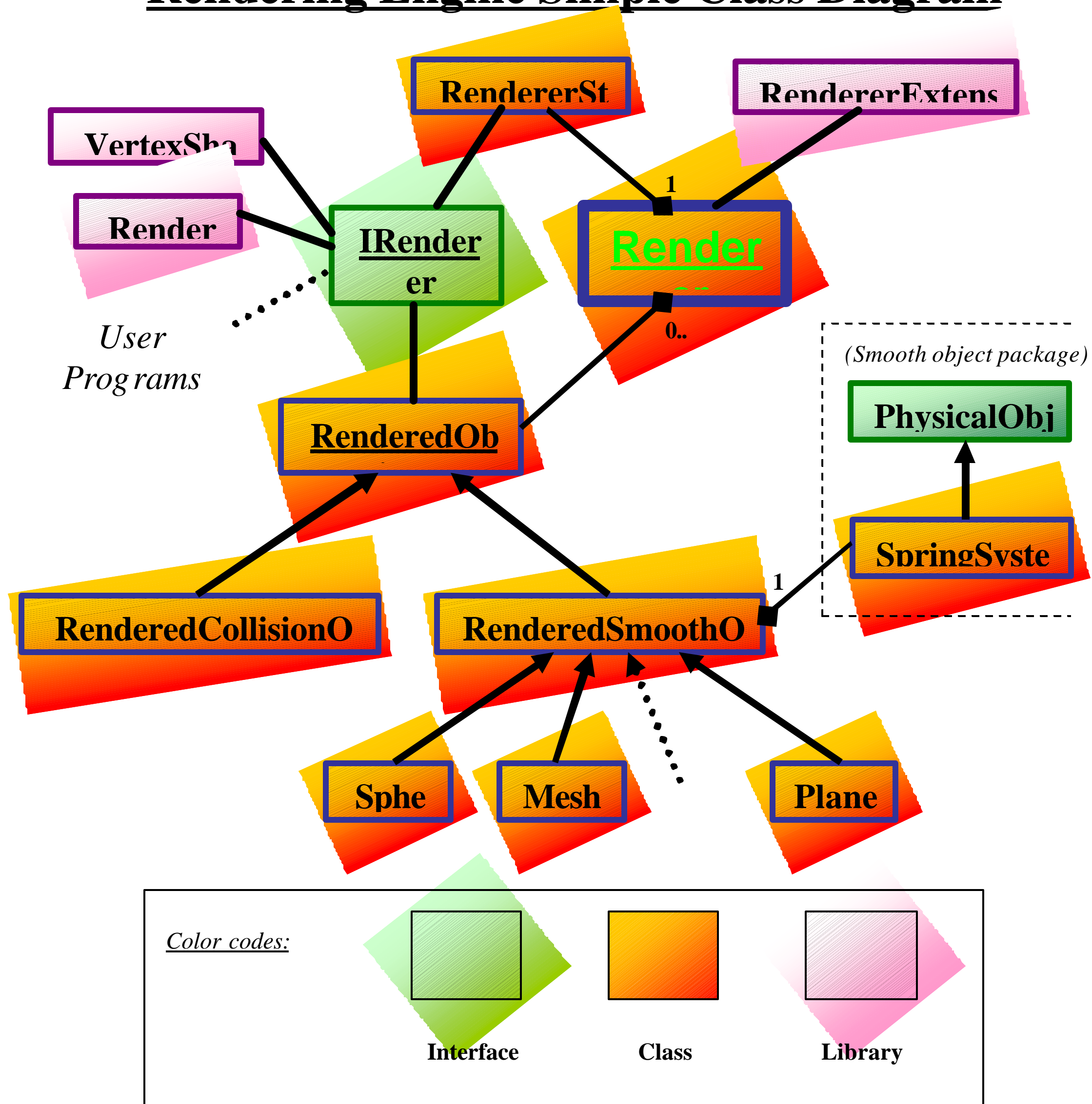
11.2.1. Graphic Layer Abstraction

The engine architecture respects the remark of object coherency, and is designed for useability and independancy as a stand-alone rendering engine. The main thing to bear in mind is how clear is the separation between the physically based modeling part and the graphic rendering part, which consists in the rendering package that constitutes our rendering engine.

In other words, the graphic representation of all objets is totally abstracted from their actual modeling. The advantage is that both parts do not blend with each other, therefore there can be much more than one graphical rendering type for a given object, independantly of it, and this is really cleaner architecturing-wise.

Every object has though its own corresponding graphic implementation in the rendering engine, in respect to a *strategy-like* design pattern.

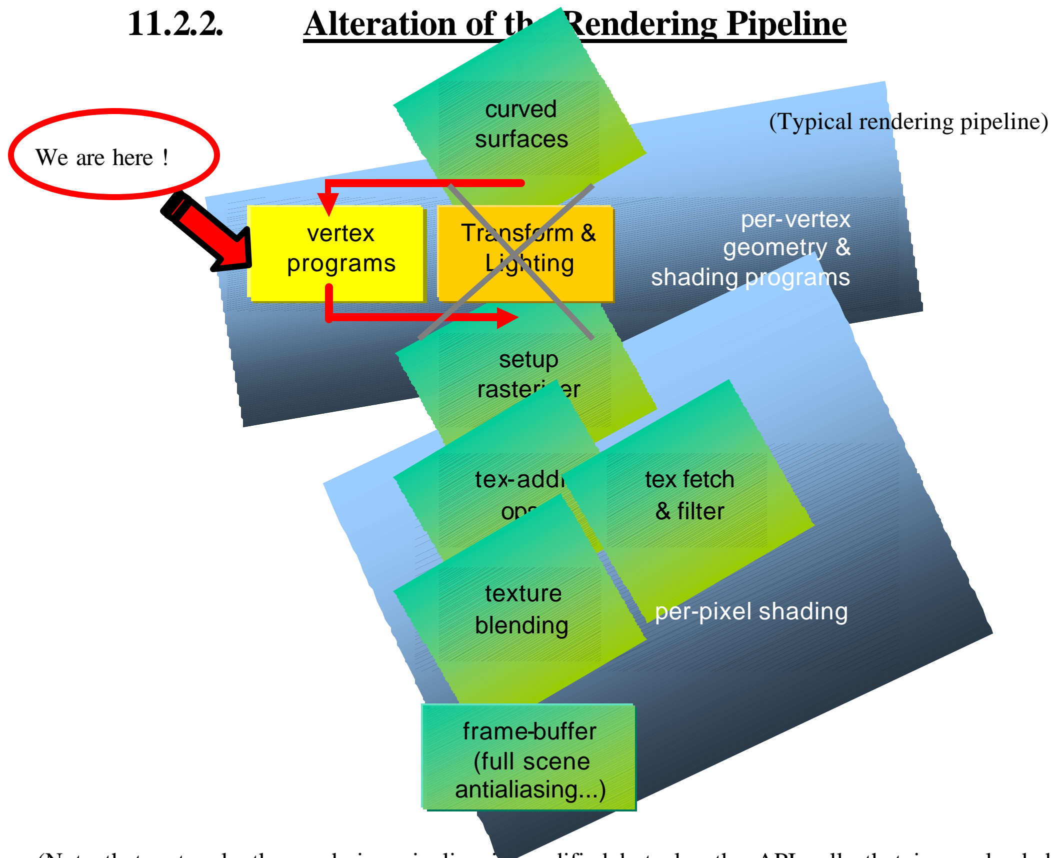
The figure below is a (simplified) class diagram of the rendering engine architecture that is used for the project:



Note:

The **IRenderer** interface offers a convenient interface on the **Renderer** possible actions, and it overloads common low-level graphic API calls (in our case, *OpenGL)*. The reasons for that are explained further, but among them, one is to attempt to optimize the OpenGL driver performances. See 1.4 for a detailed overview of the IRenderer (Rendering Interface).

11.2.2. Alteration of the Rendering Pipeline



(Note that not only the rendering pipeline is modified but also the API calls that is overloaded by the IRenderer interface, see 1.4 for more details)

The standard Transform & Lighting processing, consisting in 3D geometry transformations and lighting-based calculations, are short-circuited by user-defined vertex programs that define the detailed vertex shading behaviour. Such a program is called a **Vertex-Shader**, and all our rendering engine is based on the use of these.

For example, the simplest way for this would be to perform world space transformations for just rendering vertices at their right position.

Shaders that are more complex could add texture coordinates depending on factors such as the light and eye positions. Applications of this huge number of rendering possibilities are complex rendering effects like anisotropic lighting or single-pass refraction and reflection (more details in 3.2).

The very important point is the possibility to create new rendering effects without having to alter the structure rendering base code. That is, once the object

tessellation is set up, it is not to be changed anymore. This is a valuable advantage, mainly because:

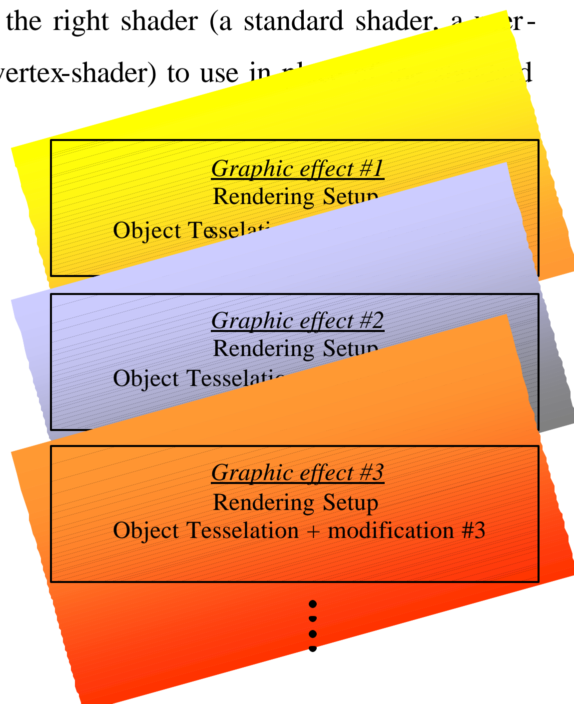
- it greatly improves the code clarity and maintenance
- it allows to write new, original rendering effects very easily and fastly

11.3. Overview of the Vertex Processing Unit

11.3.1. Per-Vertex Renderering Mechanism

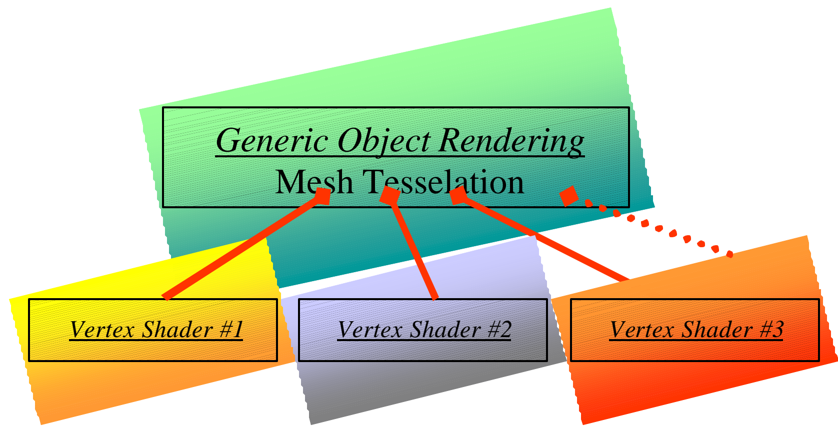
The vertex-shader mechanism does not really allows original effects that could not be realized without them (although not sure) but severely simplify the writing of them. It performs **per-vertex** operations, that is to say, it performs rendering on a per-vertex basis, and these vertices actually define the mesh of the rendered object. One of the most interesting consequences of this system is the dramatically easiness for using user-defined shaders.

Object rendering code usually reuses the base drawing code for the object, altered by the effect-specific features. It implies base code modification, and that is not convenient for third part developing and the availability of a SDK for adding its own effects. With our vertex-shader engine mechanism, the user just needs to write its own rendering effect (the user-defined vertex-shader) and assign it to the desired object, which is actually performed by the rendering engine itself. When the object is going to be rendered, it reminds of the right shader (a standard shader, a user-defined shader or whatsoever chosen vertex-shader) to use in pl[...]d usual transformation and lighting.



Several rendering effects, designed in a common way without vertex-shaders. Note the base object topology that is duplicated in each effect, with the effect-specific modifications.

A greater variety of graphic effects, in an much easier way with vertex-shaders. Note the toplogy of the object that is kept through all rendering effects, how conveniently vertex-shaders are plugged onto the object ro render.



An immediate issue of the consequences of the vertex-shader mechanism is that it is likely to degrade rendering performances: these ‘small programs’ are used in the core of the rendering and are looped a big number of times. It results in very bad performances either if the vertex-shader engine does not process vertex-shader efficiently or if the shader itself is not fast enough, i.e. too complex. The second case is attributed to the developer responsability more than the engine itself, and a too complicated shader should be broken apart with a static part that is run only once, or/and use approximations for aimed effects.

Nonetheless, main optimizations especially rely on the vertex-shader rendering engine itself, rendering performances indeed depending largely in the vertex-shader processing that are in the core of the rendering architecture. That is why strong efforts have been made in this direction, avoiding for instance numerous switch/case statements and multiple, redondant information, and reducing object memory size as much as possible.

11.3.2. Rendering Customization Possibilities

In this part, we will try to show an overall glance at the new possibilities offered by the use of vertex-shaders in the rendering implementation. It is not exhaustive (and could not anyway) and will keep rather general, without detailing why such effect works fine and why another is a good approximation for what we want.

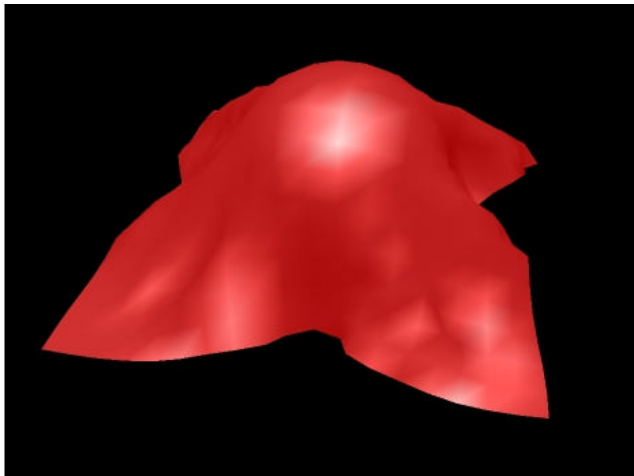
ALL MENTIONED RENDERING, IF NOTHING ELSE MENTIONED, HAVE BEEN IMPLEMENTED BY OUR VERTEX-SHADER RENDERING ENGINE AND WORKS IN REAL-TIME FOR CLOTHS OR ANY 3D OBJECT.

1) Standard lighting calculation

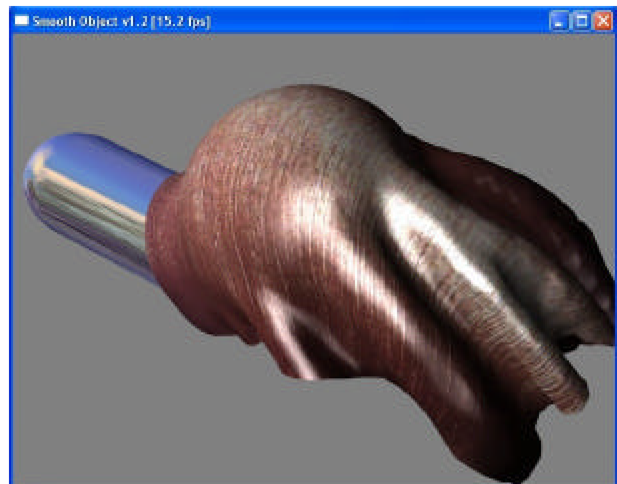
This shader just apply the same lighting as it is commonly in *OpenGL* : (specular + ambient + diffuse) * texture , but because of the low tesselation of simulated cloth for fast computations the lighting often appear segmented and not smooth. These artefacts are particularly visible in an animation.

2) Phong model-based lighting

While the real approach of the Phong lighting model is better than the Gouraud shading, it would require a full-hardware pixel-shader mechanism to be truly implemented. Although it can be rather well approximated by sphere mapping since this computes a reflection vector and the texture can be assimilated to a lookup table with the appropriate intensities, and coordinates are interpolated at pixels. The result is far better visually, and remains still fast, video cards performing sphere mapping in hardware now.



1) Standard Gouraud shading, note the tessellation artefacts



2) Textured Phong shading, lighting is quite smoother

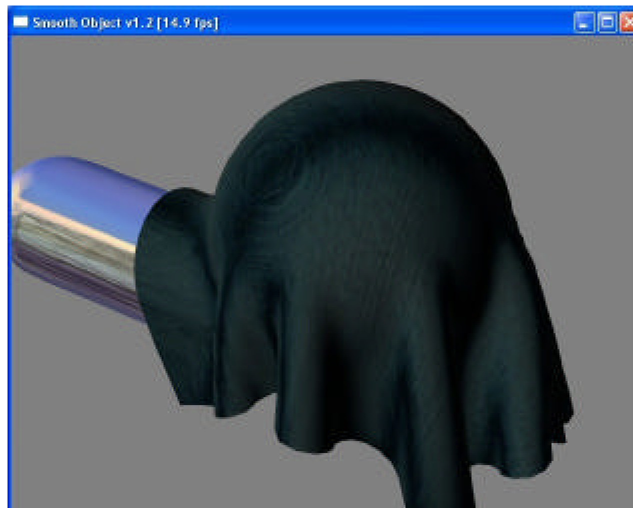
3) Velvet rendering (Minnaert lighting)

This effect is quite easier to implement with vertex-shader. This lighting model is a form of BRDF (Bi-directional Reflectance Distribution Function) which models subtle darkening effects, such as those in velvet. Briefly, it consists in computing texture coordinates depending on the eye vector, the light vector and the vertex normal (can be applied at pixels if

pixel-shaders are supported by the hardware). Dot products between these provide texture coordinates to index into a texture map containing the Minneart lighting values. This effect is particularly useful for cloth rendering, such as satin-like rendering (anisotropic lightin g) or any other cloth-like material effect. The Minnaert shader is defined by:

$$S = F(\vec{N}.\vec{L}, \vec{N}.\vec{E}) * (\vec{N}.\vec{L}) \times \text{material_color}$$

Where F(x, y) is custom (generated) texture,
S is the final vertex shade
N is the vertex normal
L is the light vector
E is the eye vector

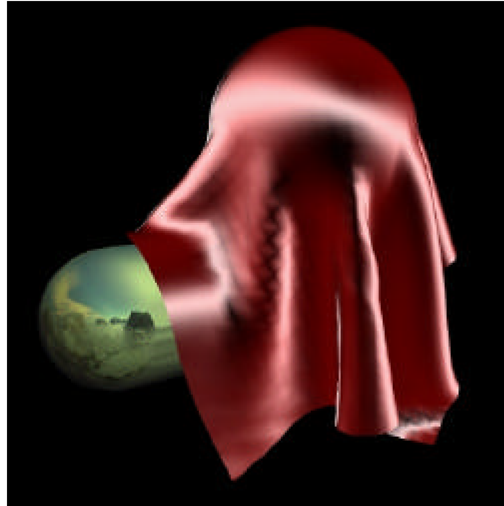


3) Textured Minnaert lighting

4) Anisotropic lighting (non-uniform surfaces like satin...)

This more complex lighting model simulates non-uniform surfaces for reflecting the light. It is generally due to an embossing, repeated pattern on the surface on a vinyl disc for instance. It is also the most suited model for hair and satin-like material, like the balls in the Christmas tree. The rendering is very interesting for clothes. It uses a vertex shader that looks like Minnaert's, but suppose we have additional information about the material orientation. For a Christmas tree ball for example, the satin texture is in fact composed of plenty of tiny weaves. This repeating pattern is responsible for the anisotropic lighting, and the light is reflected depending

on this pattern orientation, which depends on the object nature. We introduce surface tangent normals, which are concentric circles in the case of the Xmas Tree ball.



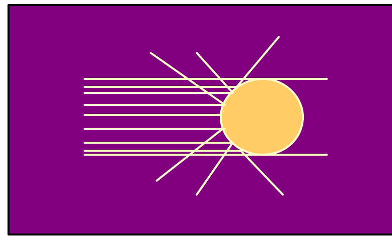
4) Anisotropic lighting model, note the non-uniform lighting pattern on the top of the cloth surrounding the sphere, like on a vynil disc

5) Reflection

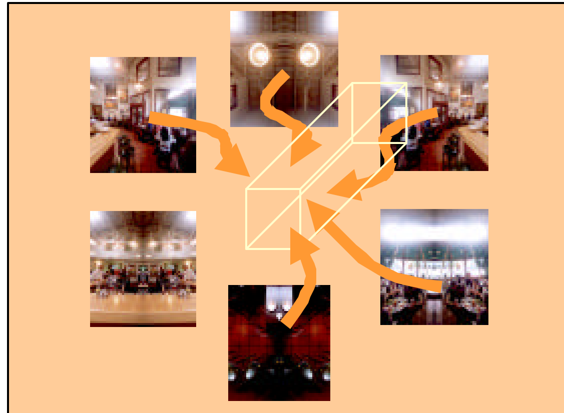
There are actually several methods to implement a reflection or refraction effect, with more or less accuracy. Two different vertex-shaders use at first hand a reflective texture generation, slightly different from the sphere mapping, while at the other hand the reflection is performed by cube mapping, modelling the reflection of a full surrounding environment.

The cube environment mapping use a 6-texture set for describing the environment cube, which can fully represent a surrounding environment. The advantage of the cube mapping over the standard sphere-projected coordinates (that use only one texture) is the very noticeable quality gain when applying the texture on the object (thanks to 6 textures instead of 1) but the sphere mapping also creates artefacts due to the lack of information 'behind' the scene that is currently pre-projected on the texture. In fact, one texture is not enough for fully describe the environment, and that is why this technique is going to be more and more common in 3D graphics today.

Here is how the sphere mapping considerates the reflection vector: the environment is captured in every direction (as seen by infinite viewer) onto the sphere that is assimilated as a single point.

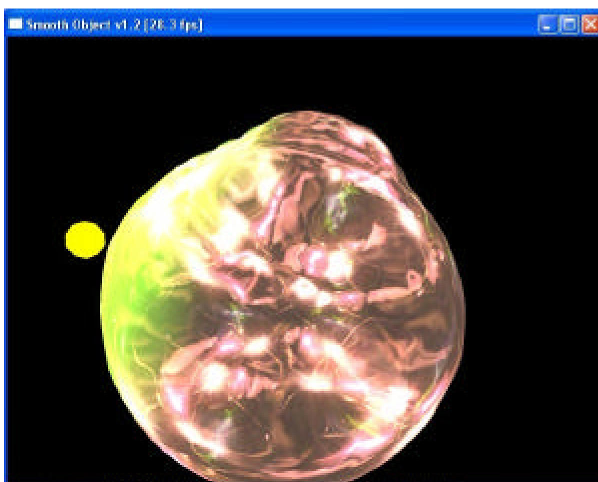


The entire environment is reflected as seen by infinite viewer (courtesy of nVidia)

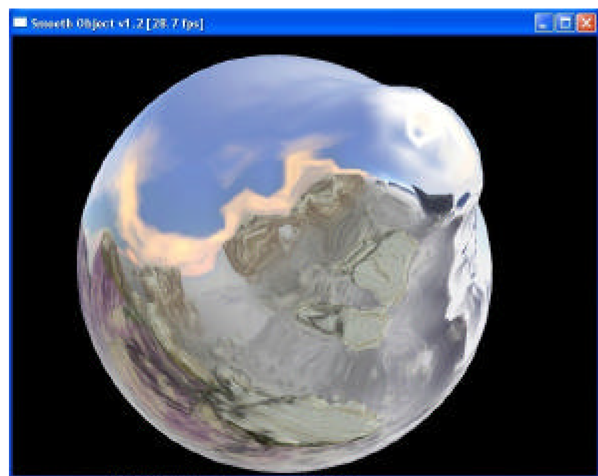


Cube map texture that represent a full-3D environment

The two following pictures use respectively a sphere-like mapping (in fact a slightly modified projected texture coordinate generation) and a reflective cube mapping. The first is rather used for giving a mirror effect (metals, glasses...) while the second truly reflects its environment: in our case we have used a landscape texture set for the 6 cube map textures.



5-1) Reflective transparent sphere, lit on left by the light

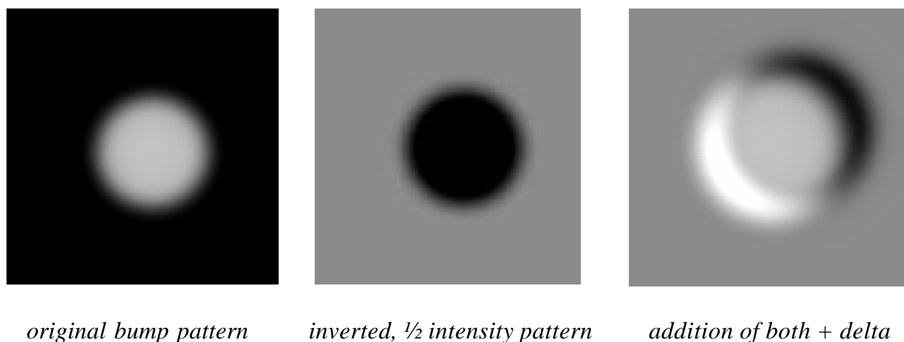


5-2) Cube mapping, reflects a 3D cubic environment

6) Bump mapping

Emboss bump mapping makes the surface of a 3D model look curved, embossed with a given bumping pattern that is defined by a texture that can either represent a height field or the normal (dot-product) perturbation of the surface texture (dot 3 bump mapping). The vertex-shader that we used for achieving this use two versions blended together of the bump texture, modulated with the object texture using multi-texturing capabilities. It is therefore a 2-pass bump mapping, but a 1-pass bump mapping could be realized with hardware-specific extensions such as *nVidia*'s register combiners.

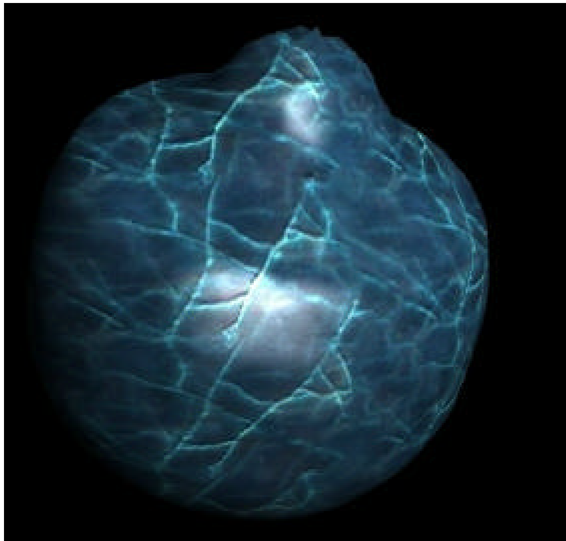
For illustrating a simple bump mapping technique, let us considerate the following bump pattern that we have thereafter inverted and set to half-brightness. The third texture represents the light incidence on the bump pattern, which is calculated by adding the two previous textures together, slightly translated from each other regarding to the light position for simulating the light influence on the curved surface. The first two textures are computed once at the beginning, and the final one is rebuilt on each frame. This operation is not expensive when using texture blending, which can be achieved in one single pass as mentioned above.



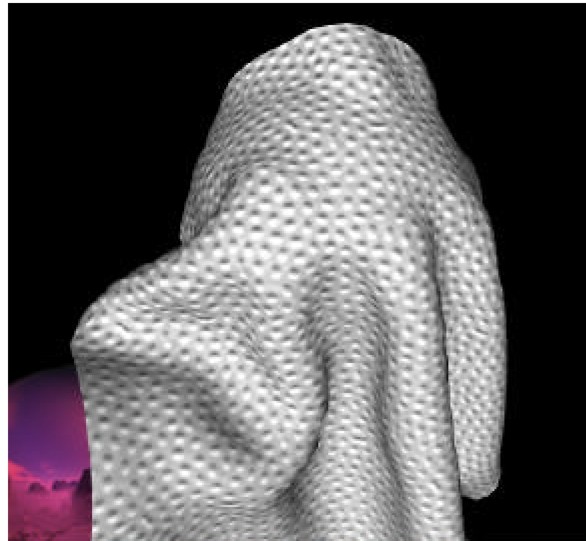
original bump pattern *inverted, ½ intensity pattern* *addition of both + delta*

Note the addition of the mid-grey, giving a bright light intensity, while the darkening opposite side of the bump pattern gives it shadow effect. In this example, the light would typically come from the bottom-left corner.

Here are two bump-mapping examples performed by our rendering engine:



6-1) Bump nerves, noticeable embossed surface feeling



6-2) Bump carves, note the incidence of the light making them particularly noticeable (voluntary magnified for better visibility)

Bump mapping is very useful for simulating rough material, which can be suited for cloth texturing.

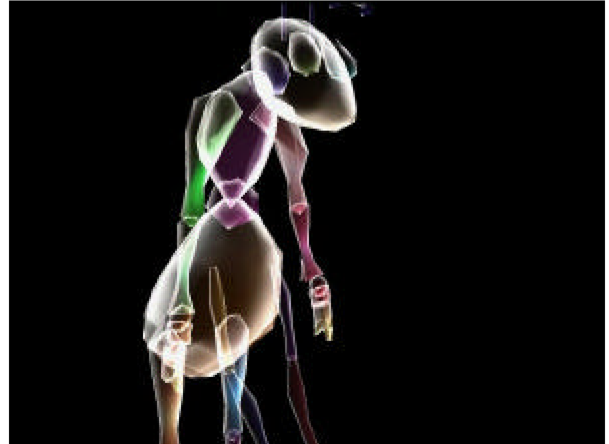
7) Toon shading

Among the latest fashion 3D effects is the cartoon-like rendering, called toon shading. One more time there is still several ways to do this: use a cube map texture for the silhouette, computing manually the light intensity with a threshold table, or using a sphere map that simulates the threshold table. The dark silhouette is created by modifying the model geometry along the surface orientation. The technique we used requires a 2-pass rendering and may be adapted for non-realistic rendering scenes.

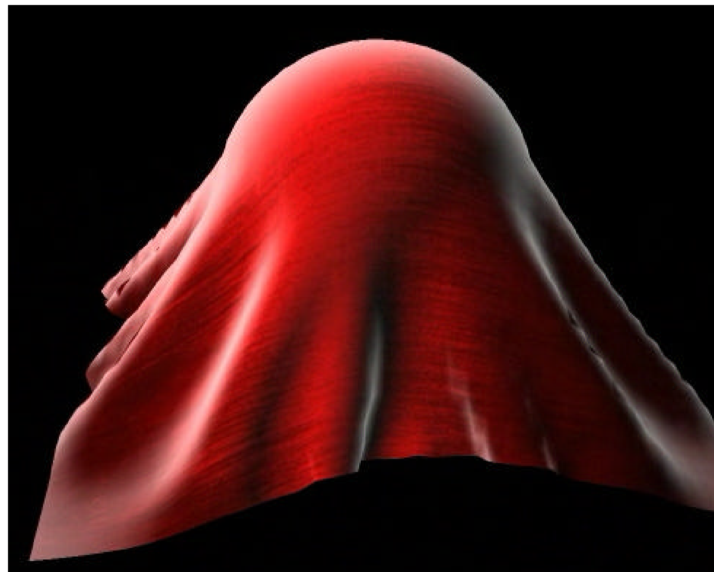
8) Silhouette rendering and glass effect

The silhouette or membrane rendering enhances edges in different ways, and it has been adapted for suit cloth rendering by reflecting more light on edges like some specific cloth textures (theatre curtain sometimes look like this). Another interesting application is to use transparency on the model for achieving what we called a glass effect, because the final rendering look like a transparent, glass-like 3D model.

7) Toon shading, note the dark, pencil-like silhouette



8-2) Transparency applied to the silhouette rendering creates a nice glass-like rendering effect



8-1) Silhouette lighting, particularly suited for certain clothes

9) Geometrical transformations

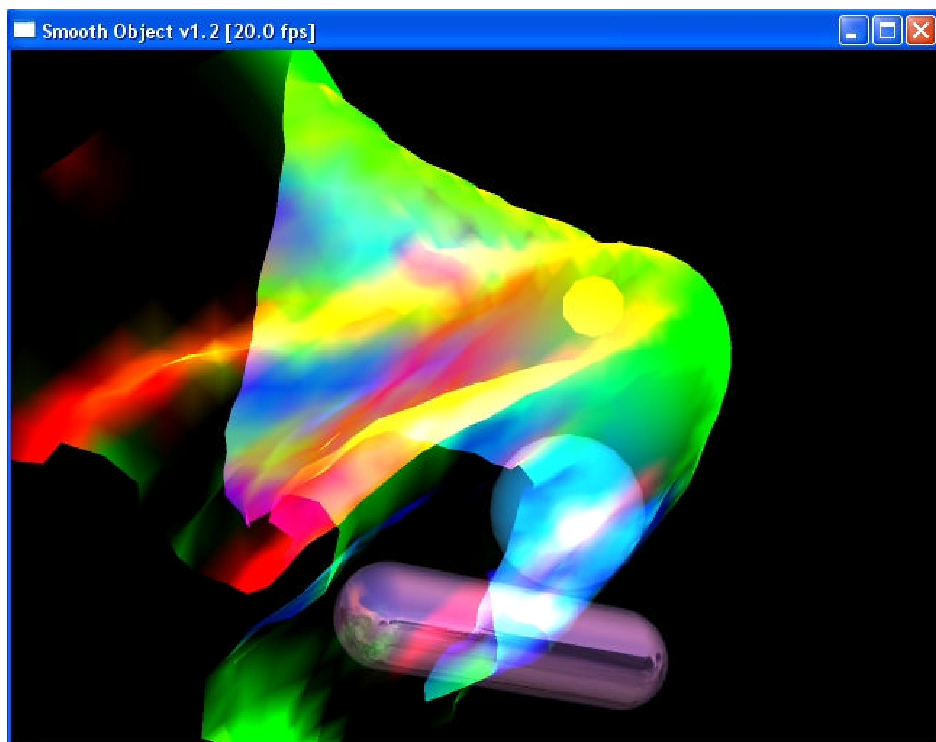
Plenty of different transformations may be applied by vertex shaders, from single rotations or projections to more complex effects like fish-eye lens deformation or mesh deformation / morphing. Although few have been implemented, this vertex-shader type has no direct application to cloth rendering, but pixel-shaders could be used for improve collision rendering for example.

10) Miscellaneous

There is no really limitations for the variety of effects that is possible, but here is a sample list of what can be achieved with vertex-shaders :

- multiple light sources (much more than 8 and CPU free)
- Volumetric fog, layered fog
- Blinn bump reflection and variations (refraction, Fresnel effect…)
- Depth of field effect (camera focus)
- Procedural shapes: Perlin noise, fractals…
- Fur effects, grass effects, particle systems…
- …and much more!

To conclude, we will show a user-defined vertex-shader : the effect could be written in less than 1 minute thanks to the great easiness of rendering effect assignment through vertex-shaders. It uses no lighting, but computes a colour for each vertex depending on its normal. The geometry has been altered to perform a complex translation and rotation in the 3D space in real-time, regardless the actual positions the vertices should have.

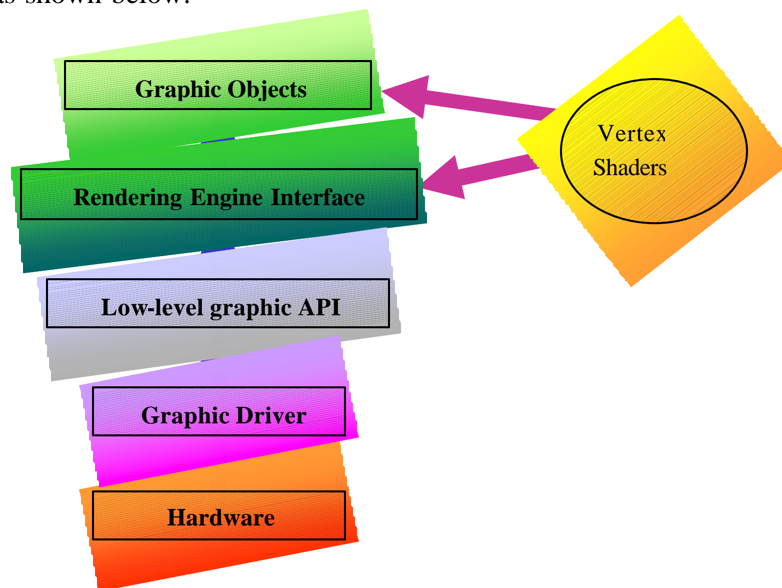


A dynamically linked user-defined vertex-shader applied on a cloth

11.4. The Rendering Interface

11.4.1. A High-Level Abstraction Layer

The interface IRenderer overloads the lower-level graphic API (*OpenGL*). The first reason was to give convenient access to the Renderer main functionalities for rendered objects. The second replace standard OpenGL calls and try to clean them for optimizations. That is, most *OpenGL* state changes are relatively expensive, causing performance losses. That is why this interface is lying just between the application layer and the *OpenGL* layer as shown below:



For achieving this, the Renderer states are managed in order to be as unchanged as possible, because redundant state changes are often expensive (even though drivers are more and more optimized regarding that issue) and, worse, sometimes useless. Performance benefits are immediate, especially when combined with the **asynchronous rendering** mode, which is described below in the section 4.2.

Besides, it would be interesting to tell about '**application robustness**': in fact, nicest effects use specific *ARB* (= standardized by the *OpenGL* ***A****rchitectural* ***R****eview* ***B****oard*) or vendor-specific extensions, but the

problem is that different cards do not always support these extensions, and the application may not work or just crash. The Renderer manages all the needed extensions for special rendering effects, and try to use those available i.e. supported by the graphic device. If not, then the Renderer may try to emulate the desired effect (with lesser performances, though) or just not perform the effect. The aim is to be able to make the application work, despite of the effects that could not be supported by a given hardware configuration.

Especially in the 3D graphic field, which evolves incredibly fast, this compatibility issue is very common: in games for example, there are several patches and updates following the release date of the software. This task is handled by the Renderer and alows a solid, robust engine. Hardware-specific issues do not concern the engine user any more, which is very convenient when developing.

11.4.2. Optimizing the low-level API

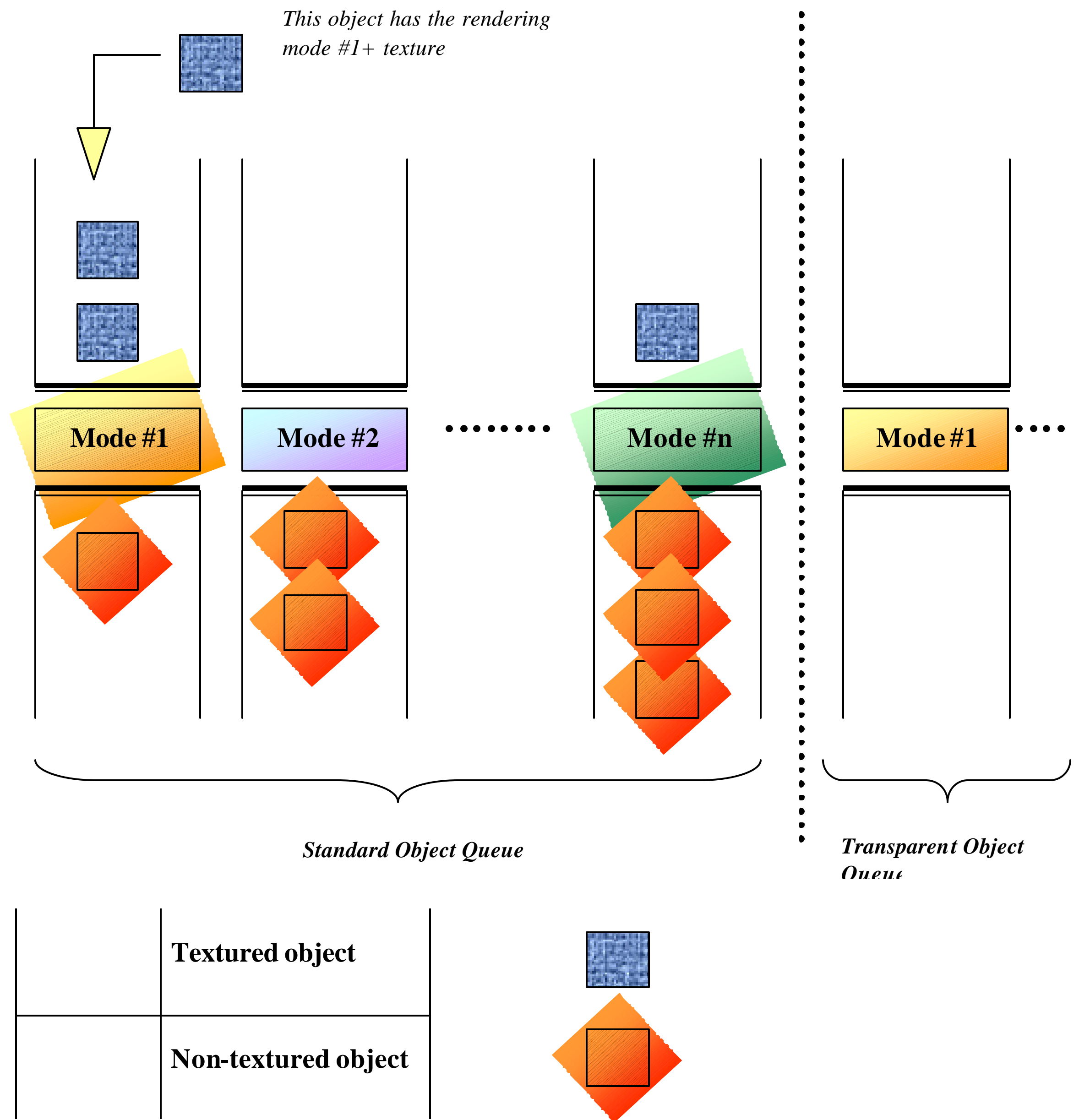
We are now going into the core of the Renderer mechanism, directly responsible for the rendering performances. That is why this is crucial to optimize this part as much as possible. We have just seen the benefits of the use of the Renderer interface that interferes with the low-level API standard calls. Furthermore, since the Renderer provides a set of common special rendering effects, it is able to group rendered objects for the same rendering mode, thus decreasing even more state changes and effect setup computations. This implies that the Renderer must know all objects that have to be rendered, for sorting them and fasten the buffer flushing i.e. the global rendering of all objects.

The **object sorting** is performed at several levels:

1) first, textured objects are pushed on the end of the queue while non-textured ones are inserted on front, decreasing the number of texture bindings
2) second, each object is sorted along its current rendering mode, joining other objects sharing the same mode

3) special rendering states, such as transparent or bumped states are treated in special, independent queues. This stands not for optimizing them but rather for not interfering more 'conventional' modes, that is, in which these states are disabled

Buffer management and object sorting :



The schema above shows the object sorting by rendering mode mechanism, where n rendering modes are defined for each queue. The first queue here stands

for regular rendering settings, while the second queue on the right handles transparent objects. A similar queue could be used for bumped objects or special effects that share a common feature, and for which it is worth to join them together performance-wise.

This asynchronous rendering mechanism, which allows a valuable performance gain, especially for complex scenes with many objets to render, requires this pre-processing consisting in a basic sorting scheme. The latter is done very fast, and so does not bring additional charge on the global rendering pipeline.

11.5. Smooth Object Rendering

11.5.1. Rendering Engine Benefits

Including the previously described performance enhancements brought by our rendering engine, it has also been very benefic for the specific cloth implementation. By handling all the tough graphic stuff, other team members could concentrate their efforts on the numerical simulation. The commodity of the engine API also helped a lot in the development cycle, because its integration comes in a very easy way. Moreover, special engine features allow very nice effects, especially cloth textures as detailed in the next section.

11.5.2. Rendered Smooth Object Examples

Among the huge variety of possible effects with vertex-shaders, not every ones are always adapted to realistic cloth rendering. Nevertheless, it can be quite interesting to test effects that would not exist in practice, for amazing effects and for sketching what could be fashionable garments in the future, maybe with textiles that will only be created in several years.

More precisely, some lighting techniques revealed to be very well suited to clothes, especially satin or velvet effects (refer to the vertex-shader capabilities section 3.2). This leads us to a very important fact to keep in mind: the realism benefits due to the final visual rendering quality are as important as the physical simulation. The physically-based modeling is one part of the problem, the visual simulation is another one as well, and both require a lot of work and both again contribute to a good cloth modeling.

11.6. Discussion about the Rendering Engine

11.6.1. Feature Summary

The rendering engine features finally solve the common rendering issues introduced in at the beginning:

- it increases global rendering performances, directly inferring on the framerate so much important in real-time 3D simulation,
- programming wise the engine architecture is independent and very robust, providing a convenient use and an easy per-user customization,
- the compatibility issues so frequent in the video-game industry are handled by the engine, and the consequences are the global robustness of the application and the optimal use of the hardware capabilities as well
- the vertex-shader opportunities allow astounding effects, increasing even more the physically-based modelling of clothes

11.6.2. Potential Improvements

The buffered rendering proved to be rather efficient, but it could be improved by using a more sophisticated prediction scheme, analyzing what objects are rendered each frame and trying to optimize the scene rendering more and more. Such a scheme though would require a constant scene analysis,

potentially CPU-expensive, and the final benefits would rather result in a performance loss. Nonetheless, hardware implementations are better suited, and this is the case for example for fast Z-occlusion culling, and vertex buffer cache in latest advanced 3D graphic devices.

A big visual quality improvement could be achieved by extending the vertex-shaders to the use of pixel-shaders. That means that we could realize per-pixel rendering, and the realistic lighting models introduced before would be more than ever! The problem is the very high complexity of implementation that requires a hardware solution. That is the case in recent video cards such as *nVidia*'s *GeForce3* and *GeForce4*, which allows incredible per-pixel rendering effects (see conclusion).

11.7. Brief Conclusion on the Rendering Engine

This rendering engine required a big amount of work, but we are very proud of the result, since it can be reused in other applications keeping its intrinsic benefits. The vertex-shader mechanism came from its own, and it has been refined all along the development process. It is interesting to remark that recent video cards (since *nVidia*'s *GeForce3* families, only about 6 month old) are hardware vertex-shader and pixel-shader capable: the mechanism we dealt with is hardware computed, thus very efficient, and even more generic since vertex-shaders are implemented in a specific assembly language, extremely fast to execute. This technology, very convenient for developers, sketches what will be future video cards generation where there will be almost no limitations for possible 3D rendering effects.

CONCLUSION

En conclusion, nous avons atteint les objectifs que nous nous étions fixés. Nous pouvons simuler en temps réel toutes sortes de vêtements. Nous pouvons agir à la fois sur la structure des vêtements ainsi que sur leur texture. De plus, la simulation permet d'utiliser n'importe quel objet en tant qu'objets souples. Nous pouvons par exemple simuler les cheveux, l'herbe…

Ce travail en plus d'être passionnant nous a permis de mettre en pratique de nombreuses connaissances et de les approfondir. Notre travail en groupe a été enrichissant notamment lors de la recherche de nouvelles idées ou de la résolution des problèmes.

Ce projet constituera une bonne carte de visite dans notre carrière professionnelle dans l'image.

REFERENCES

Large Steps in Cloth Simulation
David Baraff Andrew Witkin
Robotics Institute
Carnegie Mellon University

Physically Based Modeling : Principles and Practice
Differential Equation Basics
Andrew Witkin and David Baraff
Robotics Institute
Carnegie Mellon University

Physically Based Modeling : Principles and Practice
Implicit Methods for Differential Equations
David Baraff
Robotics Institute
Carnegie Mellon University

Physically Based Modeling : Principles and Practice
Particle System Dynamics
Andrew Witkin
Robotics Institute
Carnegie Mellon University

Physically Based Modeling : Principles and Practice
Constrained Dynamics
Andrew Witkin
Robotics Institute
Carnegie Mellon University

Vignettes
Faster Cloth Dynamics
David Baraff and Andrew Witkin
School of Computer Science
Carnegie Mellon University

Implementing Fast Cloth Simulation with Collision Response
Pascal VOLINO, Nadia MAGNENAT THALMANN
MIRALab, C.U.I, University of Geneva – CH-1211, Switzerland

Dynamic Cloth Simulation
Using a Novel Fast Collision Detection
Gu Erdan Xu Duanqing Wang Jingbin Chen Chun
Computer Science and Engineering Department, ZheJiang Univ.
State Key Laboratory of CAD&CG,
Hangzhou, P.R China, 310027

Effet de Drappé / Cloth Modeling
Projet de fin d'études : Rapport final
Gérald FAUVELLE, Jérôme LAPORTE
Ecole Supérieure en Sciences Informatiques
Université de Nice – Sophia Antipolis

Versatile and Efficient Techniques
for Simulating Cloth and Other Deformable Objects
Pascal VOLINO (*), Martin COURCHESNE (**)(*),
and Nadia Magnenat THALMANN(*)(**)
(*) MIRAlab, C.U.I. University of Geneva
(**) H.E.C University of Montreal

Representation of Woven Fabrics
Donald H House
Visualization Laboratory
Texas A&M University
David E. Breen
Computer Graphics Laboratory
California Institute of Technology

Deformation Constrains in a Mass-Spring Model
to Describe Rigid Cloth Behavior
Xavier Provot
Institut National de Recherche en Informatique et Automatique (INRIA)
France

Cloth and Clothing in Computer Graphics
Donald H. House
Visualization Laboratory
Texas A&M University
Richard W. DeVaul
Media Laboratory
Massachusetts Institute of Technology

[Published in Eurographics'94 proceedings]
Efficient self-collision detection
on smoothly discretized surface animations
using geometrical shape regularity
Pascal VOLINO and NADIA Magnenat THALMANN
MIRALab, C.U.I University of Geneva
CH-1211, Switzerland

Rappels mathématiques
Transformations géométriques 2D et 3D
Michel Buffa
UNSA, UFR Sciences, 1996

Interactive Cloth Simulation
Matthias Wloka
NVIDIA Corporation, 2001

Higher Order Surfaces in OpenGL with NV_evaluators
Sébastien Dominé
NVIDIA Corporation, Game Developers Conference 2001

OpenGL Performance
John Spitzer
NVIDIA Corporation, Game Developers Conference 2001

Per-Pixel Lighting
D. Sim Dietrich Jr.
NVIDIA Corporation, 2002

Computations for Hardware Lighting and Shading
Mark Kilgard
GDC 2000 Programming Session, March 2000

Phong shading
Michael I. Gold
NVIDIA Corporation, 2000, based on 1997 SIGGRAPH course **Programming with OpenGL: Advanced Techniques** by *David Blythe* & *Tom McReynolds*

More Advanced Hardware Rendering Techniques
Mark Kilgard
NVIDIA Corporation, 2001

Real-time Environment Reflections with OpenGL
Mark Kilgard
NVIDIA Corporation, 2000

Hardware Accelerated Anisotropic Lighting
Mark Kilgard
NVIDIA Corporation, 2000

OpenGL Vertex Programming on Future-Generation GPUs
Chris Wynn
NVIDIA Corporation, 2001

NVIDIA OpenGL Extension Specifications
Mark J. Kilgard
NVIDIA Corporation, February 2002

Order-Independent Transparency
Cass Everitt
NVIDIA Corporation, 2001

OpenGL Performance FAQ for NVIDIA GPUs v2.1
John Spitzer
NVIDIA Corporation, 2001

Automatic Mipmap Generation
Chris Wynn
NVIDIA Corporation, 2001

Implementing Bump-Mapping using Register Combiners
Chris Wynn
NVIDIA Corporation, 2001

Using GL_NV_vertex_array_range and GL_NV_fence on GeForce Products and Beyond
John Spitzer and Cass Everitt
NVIDIA Corporation, 2000

Anisotropic Reflections in OpenGL
Wolfgang Heidrich
Friedrich Alexander University of Erlangen-Nuremberg
Technical Faculty, Computer Science Department
Chair for Computer Science 9 (Computer Graphics Group)

Real-Time BRDF-based Lighting using Cube-Maps
Chris Wynn
NVIDIA Corporation, 2002

• Cours en ligne :

http://www.essi.fr/~buffa/cours/synthese_image/index.html
Supports de Cours OpenGL : Sélection/Picking

http://www.gametutorials.com/Tutorials/OpenGL/OpenGL_Pg4.htm
Moteur d'importation de modèles 3D.

http://www.nivia.com/
OpenGL SDK et toutes les démos avec leurs codes source disp onibles