Vincent Lachenal et Raphaël Ducom

Rapport de Projet Professionnalisant en Équipe

Outils d'Édition d'Animation d'Humanoïde de Synthèse

Encadrant : Alexis Nédélec et Thomas Jourdan

Cerv : Centre Européen de Réalité Virtuelle

Enib : École Nationale d'Ingénieurs de Brest

Résumé

Ce rapport présente les travaux effectués au cours du Projet Professionnalisant en Équipe de 4^{eme} année de École National d'Ingénieurs de Brest intitulé :

«Outils d'Édition d'Animation d'Humanoïde de Synthèse»

Ce Projet Professionnalisant en Équipe a été réalisé au *CERV*, avec pour objectif de fournir à la librairie *hLib*, développée par Thomas Jourdan, des outils d'édition d'humanoïdes de synthèse.

Ces outils auront pour objectif de faciliter l'édition d'animations de systèmes polyarticulés, en permettant de spécifier des contraintes sur leur liberté mouvements, ou d'en éditer des ensembles d'animations qui leurs sont associés.

hLib est une librairie complémentaire d'*ARéVi*, développée par Fabrice Harrouet, qui ajoute au moteur de réalité virtuelle d'*ARéVi* une gestion de chaînes polyarticulées tel qu'un squelette d'humanoïde (ou n'importe quel squelette de robot, d'animal), et permet une intéraction en cinématique directe ou cinématique inverse sur ces chaînes, de façon à manipuler de manière directe l'orientation d'une articulation, ou de manière indirecte, la position de l'extremité d'une chaine (par ex. la main d'un humanoïde).

Les deux logiciels développés au cours de ce PPE utilisent les fonctionnalités offertes par ces deux bibliothèques, et fournissent à l'utilisateur une interaction directe sur des modèles d'humanoïdes, facilitée par l'utillisation d'une IHM simple et intuitive. Ces deux logiciels sont l'*Éditeur de Contraintes*, permetant de définir très précisément des contraintes de libertés de mouvement sur chaque articulation d'un humanoïde, et l'*Éditeur de Motion Blending*, permetant la fusion controlée de différentes animations d'humanoïdes afin d'obtenir de nouvelles animations «mixées» à partir d'animations existantes, et d'appliquer le résultat à un squelette humanoïde.

Table des matières

Table des figures

Chapitre 1

La Réalité Virtuelle et le CERV

1.1 La Réalité Virtuelle

Les systèmes qu'on cherche à modéliser sont de plus en plus complexes. Cette complexité provient essentiellement de la diversité des composants, de la diversité des structures et de la diversité des interactions mises en jeu. Un système est alors a priori un milieu ouvert (apparition/disparition dynamique de composants), hétérogène (morphologies et comportements variés) et formé d'entités composites, mobiles et distribuées dans l'espace, en nombre variable dans le temps.

Ces composants, parmi lesquels l'homme avec son libre arbitre joue souvent un rôle déterminant, peuvent être structurés en différents niveaux connus initialement ou émergeant au cours de leur évolution du fait des multiples interactions entre ces composants.

Les interactions elles-mêmes peuvent être de natures différentes et opérer à différentes échelles spatiales et temporelles. Mais il n'existe pas aujourd'hui de théorie capable de formaliser cette complexité.

La réalité virtuelle fournit aujourd'hui un cadre conceptuel, méthodologique et expérimental bien adapté pour imaginer, modéliser et exprimenter cette complexité.

La simulation en réalité virtuelle autorise une véritable interaction avec le système modélisé. L'utilisateur spectateur-acteur-créateur peut ainsi se focaliser sur l'observation d'un type de comportement particulier, observer l'activité d'un sous-systéme ou bien l'activité globale du système. A tout moment, l'utilisateur peut interrompre le phénomène et faire un point précis sur les corps en présence et sur les interactions en cours ; puis il peut relancer la simulation là où il l'avait arrêté. A tout moment, il peut interagir avec le système (modification/ajout/suppression d'élèments afin de tester un comportement particulier) et observer les conséquences sur le fonctionnement du système.

Dépassant la simple observation, l'utilisateur peut tester la réactivité et l'adaptabilité du modèle en fonctionnement c'est l'expérimentation in virtuo.

Une expérimentation in virtuo est ainsi une expérimentation conduite dans un univers virtuel de modèles numériques en interaction et auquel l'homme participe. La réalité virtuelle implique pleinement l'utilisateur dans la simulation.

L'expérimentation in virtuo implique ainsi un vécu que ne suggère pas la simple analyse de résultats numériques. Entre les preuves formelles a priori et les validations a posteriori, il y a aujourd'hui la place pour une réalité virtuelle vécue par l'utilisateur qui peut ainsi franchir le cap des idées reçues pour accéder à celui des idées vécues.

1.2 Le CERV : Centre Européen de Réalité Virtuelle

Le Centre Européen de Réalité Virtuelle (*CERV*) est un centre de recherche publique de l'Ecole Nationale d'Ingénieurs de Brest. Le CERV a été créé à l'initiative du Laboratoire d'Ingénierie Informatique (LI2) de l'ENIB ; il est devenu opérationnel en juin 2004 et occupe $2000m^2$ sur le site du Technopôle Brest-Iroise. Il s'inscrit à Brest dans un dispositif portant sur la thématique des Sciences et Technologies de l'Information et de la Communication (STIC).

Le CERV constitue à Brest un pôle d'excellence en réalité virtuelle à vocation européenne. Il participe à la production de connaissances, à leur diffusion, à leur valorisation, et intègre en un même lieu

- un centre de recherche inter-établissements pour fédérer les travaux de différentes équipes de recherche en privilégiant la vocation pluridisciplinaire de la réalité virtuelle,
- un centre de valorisation pour favoriser les relations recherche-entreprise autour de projets innovants utilisant la réalité virtuelle,
- un centre de formation à la réalité virtuelle,
- un centre de découverte de la réalité virtuelle pour sensibiliser les scolaires et le grand public aux sciences et technologies du futur.

Fig. 1.1 – Le Centre Européen de Réalité Virtuelle

Chapitre 2

Les Outils

2.1 ARéVi : Atelier de Réalité Virtuelle

Contexte scientifique : Le contexte dans lequel s'inscrit ce projet est l'utilisation conjointe de la réalité virtuelle (RV) et des systèmes multi-agents (SMA[1]) : d'un côté, les systèmes multi-agents permettent d'enrichir les mondes virtuels en leur ajoutant des entités autonomes dotées d'un comportement propre ; de l'autre, il est possible d'interagir avec ces systèmes multi-agents au moyen d'interfaces mettant en œuvre les techniques liées à la réalité virtuelle.

Thématiques : Le projet *ARéVi* se démarque sur deux points : le premier concerne la nature de l'approche qui repose sur l'utilisation conjointe de la RV et des SMA ainsi que sur l'emploi de langages pour décrire les comportements des agents et pour échanger des connaissances entre entités (agents, avatars et utilisateurs). Le second est lié à la méthodologie retenue : rechercher très rapidement, à travers les activités des autres thèmes du laboratoire, à évaluer les méthodes et outils ainsi qu'à les faire évoluer en fonction des résultats d'expérimentations faites sur des applications "dimensionnantes". Les thèmes de recherches de ce projet reposent spécifiquement sur le langage de prototypage, la modélisation comportementale d'agents, la communication entre entités, la perception des environnements virtuels et les plateformes d'environnements virtuels distribuées.

- **Langage de prototypage :** tendre à faire évoluer le langage dynamique de prototypage vers une approche multiparadigmes intégrant de manière homogène des aspects déclaratifs et impératifs. Le but est de pouvoir décrire des entités autonomes capables de raisonner sur les traitements quelles savent faire pour que les agents bénéficient ainsi d'une véritable autonomie de décision.
- **Modélisation comportementale :** s'atacher à étudier et modéliser plus spécifiquement les comportements humains sous l'angle de la psychologie expérimentale

[1]Un système multi-agents (SMA) est un système (ou ensemble) de programmes autonômes, doués des capacités de perception, de décision, et d'action dans leurs environnements d'execution. Ce cycle Perception/Décision/Action permet de modéliser des comportements physiques, mécaniques, voir psychologiques, propres à une entité individuelle, et de les simuler.

et neurophysiologie. Les travaux dans ce domaine sont basés sur l'étude de cartes cognitives floues, d'affordances et d'anticipation pour spécifier les comportements perceptifs et obtenir des acteurs virtuels aux décisions crédibles.
- **Communication entre entités :** cet aspect de communication concerne l'échange de connaissances entre agents sous forme d'actes de langage. Le but est de permettre d'une part, de faire communiquer agents, avatars et utilisateurs à l'aide d'un même langage de communication et d'autre part, de faire évoluer le module décisionnel des agents de l'environnement en fonction des nouvelles connaissances acquises.
- **Perception des environnements virtuels :** il s'agit ici de développer des mécanismes permettant à des utilisateurs humains ou à des acteurs virtuels autonomes de percevoir leur environnement (sous forme picturale ou symbolique) afin de prendre des décisions conformes à leurs objectifs. Les contraintes liées à cette étude concernent à la fois l'efficacité des calculs et le réalisme du rendu. Les aspects temporels et distribués de la création des mondes virtuels et de leur restitution seront pris en compte.
- **Plate-forme de réalité virtuelle :** dans le cadre des travaux de recherches, il est nécessaire d'intégrer une plateforme d'édition et d'exécution d'applications dans un environnement de réalité virtuelle distribuée. Dans ce contexte, le développement des outils permettra de visualiser, interagir et communiquer avec les entités pour tester rapidement la validité des modèles de comportement dont seront dotés les entitès de l'environnement virtuel.

Applications : Les domaines d'applications de nos travaux de recherches sont ceux des outils d'aide à la décision comme, par exemple, la prise de décision lors d'une attaque sur un réseau informatique, application que nous développons actuellement au sein de ce projet.

La bio-informatique constitue aussi un domaine d'application, des expérimentations in virtuo en immulogie et en hématologie ont déjà étés mises en œuvre avec nos outils. Les résultats de ces premières expérimentations montrent d'une part l'adéquation de la simulation individu centrée pour cette problématique ainsi que l'intérêt de l'approche réalité virtuelle pour bien appréhender les résultats obtenus et dialoguer avec la simulation.

Les environnements virtuels de formation (VET) constituent aussi un champ d'applications privilégié pour exploiter nos travaux et nos outils de prototypage interactif, collaboratif et coopératif. Un projet de réalisation d'un voilier virtuel pour l'entraînement sportif, intégrant des entités dotées de comportements spécifiques à cet environnement, est actuellement en cours. Cette réalisation est basée sur nos travaux dans le cadre de la modélisation comportementale.

Collaborations - Complémentarité : La complémentarité de ce projet avec les autres travaux du laboratoire du Centre Européen de Réalité Virtuelle se situe au niveau de la demande et de l'exploitation de nos méthodes et outils comme, par exemple, dans le cadre du projet SPI pour le développement d'applications de simulations participatives et immersives. La collaboration avec les différents projets du laboratoire est d'autant plus renforcée qu'elle nous permet d'obtenir un retour d'expérimentation rapide sur nos travaux. C'est en ce sens que nos outils jouent un rôle fédérateur au sein du LI.

2.2 hLib : Bilbliothèque de gestion d'humanoïdes de sythèse.

Contexte scientifique : *hLib* est une librairie reposant sur ARéVi permettant de gérer les mécanismes d'animation et de rendu spécifiques d'humanoïdes de synthèse.

Pour ce faire, hLib se propose, à terme, de développer une API (interface de programmation) afin de permettre de manière simple et rapide l'intégration d'humanoïdes dans un univers 3D. L'API devra rester simple et proposer à l'utilisateur final les fonctionnalités suivantes :

- création et importation d'humanoïdes,
- animation d'humanoïdes,
- planification et fusion de mouvements,
- animation finale.

De plus, hLib ne se limite pas aux seuls modèles humains. En effet, cette librairie permet la manipulation de chaînes polyarticulées quelconques, sur chacune desquelles est attachée une représentation graphique. Ce qui permet donc aussi bien de décrire un humain qu'un robot, un animal ...

Thématiques :

- **Création et importation de squelettes :** *hLib* permet à son utilisateur de créer ses propres squelettes, mais aussi d'importer des modèles développés sous d'autres plateformes de développement. À ce jour, *hLib* peut importer des formats *Biovision Motion Capture*, *Milkshape3D* et *Half-Life*, format de fichiers encapsulant un squelette et une animation qui lui est associé. On peut ainsi récupérer non seulement le squelette mais aussi les animations contenu dans ces fichiers.
- **Animation par cinématique directe et inverse :** *hLib* permet aussi d'animer les humanoïdes. Pour cela, elle met à notre disposition deux méthodes : la cinématique directe et la cinématique inverse. La cinématique directe est la spécification des positions successives pour chaque articulation d'un l'humanoïde, de façon à spécifier directement la posture d'une chaine articulée, alors que la cinématique inverse a pour vocation d'appliquer à un point donné d'une chaine, une position spécifique, en mettant à jour la partie amont de cette chaine de façon cohérente. (ex : placer la main sur le tiroir va modifier la position du coude, de l'épaule, etc...).
- **Fusion de mouvements :** *hLib* permet le motion blending de plusieurs animation. C'est à dire de faire une moyène de mouvements de ces deux animations. Par exemple : une animation d'un humanoïde qui court que l'on fusionne avec une animation d'un humanoïde qui marche donnera une animation d'un humanoïde qui marche vite ou qui court lentement selon les poids qu'on attribue aux animations que l'on fusionne.
- **Correction de mouvements en fonction du squelette :** hLib se donne pour objectif d'adapter une animation d'un squelette à un autre squelette qui possède

une structure différente.

Applications :

Les champs d'application de l'animation d'humanoïde de synthèse sont vastes. Cette librairie peut être utilisée dans n'importe quel projet nécéssitant l'exploitation d'humanoïdes de synthèse (simulations, jeux vidéos, ...), ou encore n'importe quel système polyarticulé.

Collaborations - Complémentarité :

Actuellement *hLib* est principalement destinée à des projets du CERV ayant besoin d'integrer des modèles d'humanoïdes au seins de simulations : **GRACE**, **HandiSpace** et **Handiposte**.

- **GRACE** est un projet de simulation de comportements humains. Pour une meilleure crédiblité de la représentation visuelle des comportements, le projet hLib est d'une grande importance .
- **HandiSpace** est, quant à lui, un projet visant à adapter un espace de vie à des personnes handicapées. Grâce à léditeur de contrainte, *hLib* facilite la spécification et la simulation d'handicaps, dans l'objectif de garantir l'accessibilité de cet espace.
- **Handiposte** Constitue un projet d'aménagement des postes de travail pour personnes handicapées en vue d'une réinsertion en milieu professionnel. *hLib* permet comme pour **HandiSpace** des simulations d'accessibilité réalistes.

2.3 IHM : Interface Homme Machine

2.3.1 GTK+ .

La bibliothèque GTK+ permet de créer, et manipuler des interfaces graphiques (GUI : *Graphic User Interface*), et offre un large choix d'outils permettant d'implémenter très facilement des élements standards d'une IHM, (boutons, menus déroulants, zone de texte ...).

A l'origine, GTK+ a été développé pour donner des bases solides au logiciel de traitement d'images GIMP[2]. Aujourd'hui, le domaine d'application de cette bibliothèque ne se limite pas à GIMP : elle est largement utilisée dans de nombreux projets, et est devenue une GUI standart sous linux. Le développement phare est l'environnement de bureau GNOME[3].

L'utilisation de GTK+ pour la création de GUI est très intéressante :

- GTK+ est sous licence GNU[4] LGPL[5]. Cela fait de GTK+ une bibliothèque accéssible gratuitement, permettant ainsi de l'utiliser voire de la modifier sans aucune contrainte financière. Si vous souhaitez en savoir plus, le plus simple est de visiter le site du projet GNU.
- GTK+ est multi-plates-formes : Linux et Bsd, Windows, BeOs. Ce qui permet de dévelloper des logiciels facilements portables sur une autre architecture.
- GTK+ fournit un ensemble complet de Widget permettant de s'adapter à de très nombreux projets spécifiques, et de réaliser et d'implémenter des interfaces abouties rapidement.

L'utilisation de cette bibliothèque nous a permis de créer et d'implémenter beaucoup des composants de nos éditeurs : fenêtres, boutons, menus déroulants, treeviews (arborescence d'élements), boutons ou barre de status, scrollbar ...

[2]GIMP : (GNU Image Manipulation Program) un éditeur d'image des plus reconnu sous linux.

[3]GNOME (GNU Network Object Model Environment) : un gestionnaire de fenêtres très utilisé. **http ://www.gnome.org**

[4]GNU est un acronyme récursif pour «GNU's Not Unix».(littéralement, GNU N'est pas UNIX). Cependant, ceci doit être compris comme une blague. **http ://www.gnu.org**

[5]*Lesser General Public License*, license GNU libre de droits

2.3.2 Gnome Canvas

Gnome Canvas est un Widget composant de GDK, qui fournit un moteur de rendu graphique structuré, et offre une interface relativement riche de formes de base, de grandes performances de rendu, et une API haut niveau héritant de GTK. Le tout entièrement orienté objet. Il offre le choix entre deux types de rendu, l'un basé sur Xlib pour un affichage rapide, et l'autre basé sur Libart, et permet de gérer l'anti-crenélage et la transparence.

Cette librairie permet d'utiliser une surface de dessin compatible GTK, de façon à créer et manipuler dynamiquement des objects graphiques, et d'utiliser les signaux GTK sur cette surface pour prendre en charge les évenements clavier - souris, ou encore d'implémenter des mécanismes de Drag'N Drop.

Fig. 2.1 – Gnome Canvas est largement utilisé dans de nombreuses applications gnome.

Cette librairie nous permet de représenter, composer, et utiliser un arbre d'animation dans *l'Éditeur de Motion Blending*, symbolisation cruciale pour la représentation par l'utilisateur des «blendings» successifs qu'il peut mettre en œuvre.

2.3.3 Glade

Glade est un éditeur d'interface graphiques gratuit pour GTK+ et GNOME, sous license GNU GPL.

Les interfaces utilisateurs créées avec Glade sont sauvegardées en XML, et, en utilisant la librairie libglade, elles peuvent être chargées par les applications, dynamiquement si besoin est. (Glade peut aussi générer un code en C, bien que ce ne soit pas recommandé pour la plupart des applications.)

En utilisant libglade, les fichiers Glade XML peuvent être utilisés avec de nombreux langages de programmation dont le C, C++, Java, Perl, Python, C#, Pike, Ruby, Haskell, Objective Caml and Scheme. De plus, ajouter la prise en charge d'autres langages est facile.

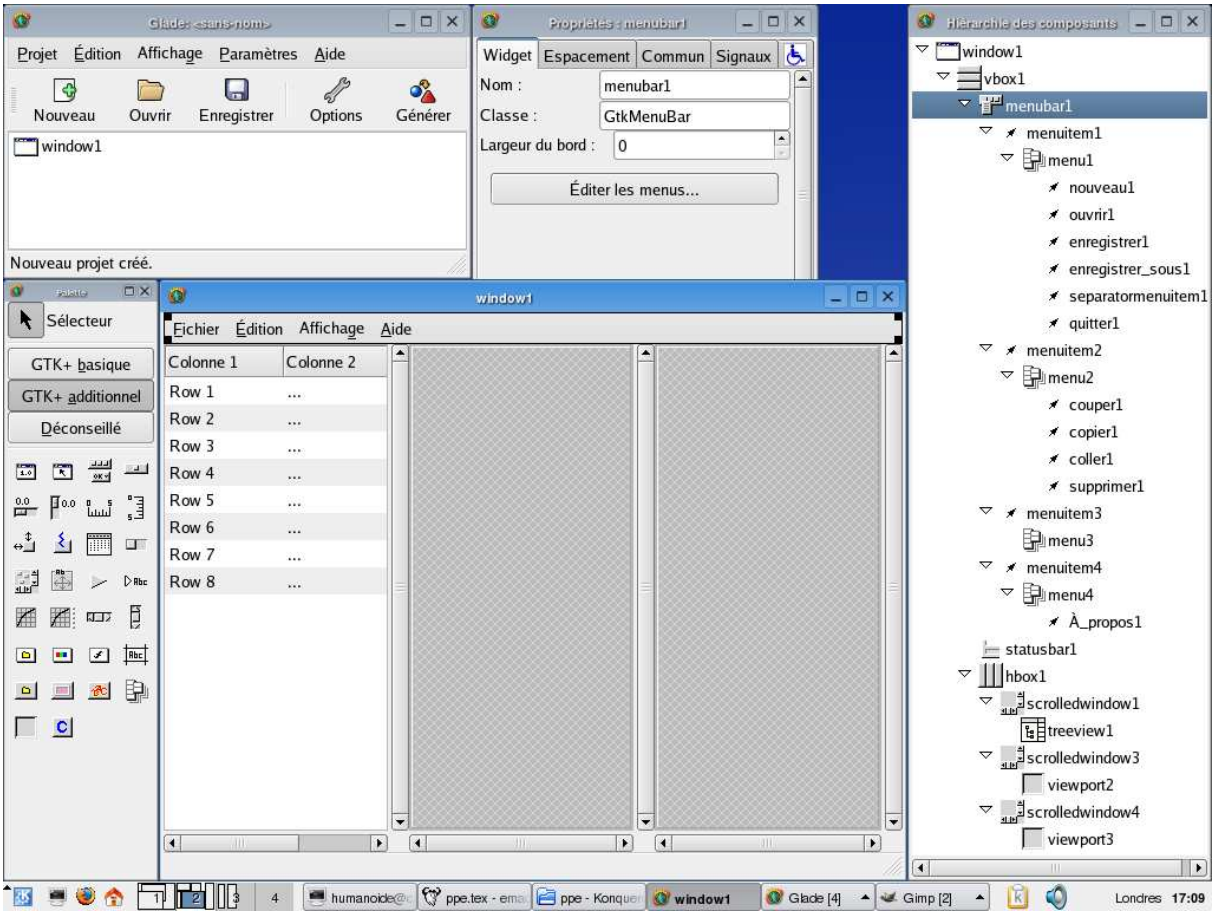


FIG. 2.2 – Exemple de prise en main de Glade

Chapitre 3

Le Projet Professionnalisant en Équipe

Outils d'Édition d'Animation d'Humanoïde de Synthèse

Ce projet a pour objectif de créer un ensemble de logiciels permettant à un utilisateur d'interagir sur tous les paramètres haut-niveaux de l'animation d'un humanoïde, de façon simple et rapide. L'utilisateur devra pouvoir effectuer des réglages fins afin d'obtenir un ensemble cohérent et réaliste lors de l'animation.

Deux domaines distincts sont visés :
- L'animation dynamique d'humanoïdes, en spécifiant des contraintes sur les mouvements possibles de celui-ci.
- L'animation «prédéfinie», en enrichissant un ensemble d'animation existantes, via la création de nouvelles animations «mixées» de celles existantes, et via la mise en cohérence de cet ensemble.

Ce projet repose en grande partie sur les travaux de Thomas Jourdan. *hLib*, la bibliothèque utilisée pour gérer l'animation d'humanoïdes, repose elle-même sur la librairie de réalité virtuelle ARéVi. Ces outils permettront à l'utilisateur final de décrire tous les mouvements et enchaînements de mouvements possible pour son humanoïde, ceci afin d'atteindre le niveau le plus proche possible de la réalité.

Le projet se divise en trois lots :

- **Éditeur de contraintes :** Spécifie les libertés de mouvement de chaque articulation d'un système polyarticulé, de façon à simuler dynamiquement ce système contraint.
- **Éditeur de Motion Blending :** Permet la création de nouvelles animations en faisant une «moyenne pondérée» d'animations existantes.
- **Éditeur de Motion Graph :** Définit la cohérence d'un ensemble d'animations, en assurant les transitions d'une animation à l'autre au travers d'un graphe de ces animations.

3.1 Prise en main ARéVi / hLib :

Afin de prendre en main les librairies ARéVi et hLib, nous avons mis en place une petite application regroupant le plus de fonctionnalités de ces deux librairies. En ce sens, nous avons décidé de faire un échiquier sur lequel des agents modelisés par des pions, jouent à un jeu un peu particulier : chaque agent (décrit par un cycle Observation / Décision / Action) a pour objectif de choisir un adversaire dans son environnement, et de prendre de la vitesse de façon a le pousser en dehors de l'échiquier, selon la règle des combats sumos. Ces pions on étés modélisés avec des cônes dans un premier temps, puis par des humanoïdes animés par la suite.

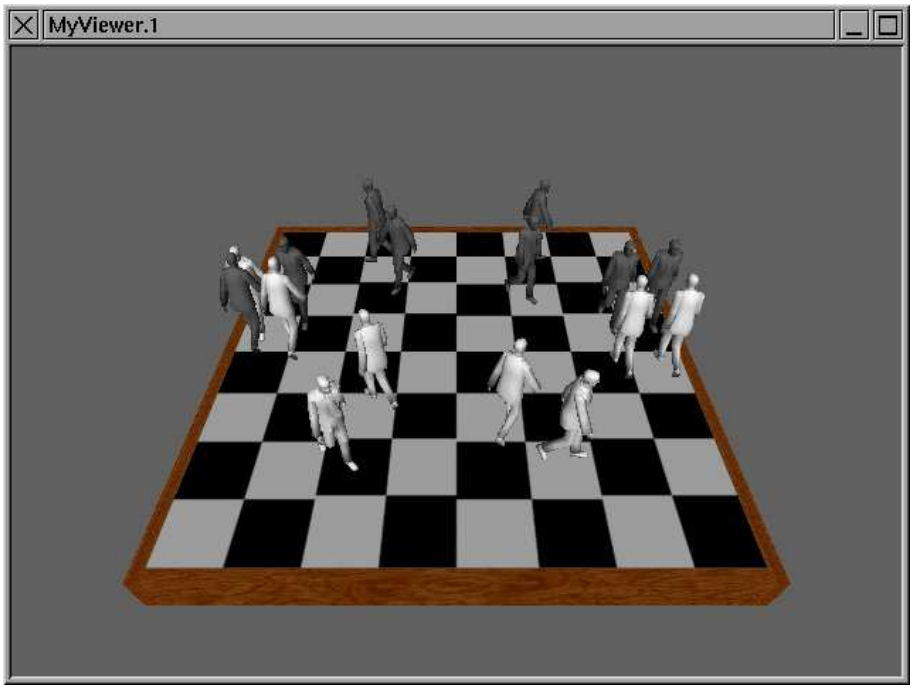


Fig. 3.1 – Exemple de prise en main d'ARéVi et de hLib

La réalisation de cet échiquier nous a permis de bien intégrer les notions de système multi-agents, de gestion de collisions, de gestion d'activités et d'animations que proposent ARéVi et hLib. Cette étape de familiarisation avec les API d'ARéVi et d'hLib nous à été très profitable pour la suite du projet, car cela nous a permis d'acquerir les bases indispensable pour travailler sur l'Éditeur de contraintes déja en partie implémenté.

Vous pouvez retrouver ce programme de test sur le CD dans :
/src/testChessSumo/ChessSumo.tgz

3.2 Éditeur de contraintes :

3.2.1 Description du projet

L'Éditeur de Contraintes est un logiciel qui a pour but de faciliter l'édition des caractéristiques physiques d'un humanoïde de synthèse, en spécifiant les limites de rotation de chaque articulation de son squelette. A terme, celà nous permet non seulement de décrire un modèle réaliste d'humanoïde, mais aussi de spécifier un ou des handicap sur ce modèle.

Un humanoïde de synthèse est modélisé par un squelette décrit sous la forme d'une chaîne polyarticulée. Chaque articulation est modélisé par un *Joint* et ces *Joint* sont reliés entre eux par des «fils de fer». Le but de l'Éditeur de Contraintes est de pourvoir définir des contraintes sur chacune des articulations afin d'obtenir des mouvements et un comportement le plus proche possible de la réalité.

Afin de tester la validité des ses travaux, l'utilisateur a la possibilité d'interagir avec son humanoïde de synthèse via une interaction en cinématique inverse.

La cinématique inverse repose sur la mise en mouvement d'une chaîne polyarticulée : en ne spécifiant qu'un point à atteindre par une des parties de cette chaine, l'algorithme de cinématique inverse redéfinit les positions et orientations de l'ensemble des articulations en amont de la chaîne, de façon à ce que le point soit atteint. Au lieu de décrire précisément et pour chaque articulation la chaîne de l'humanoïde sa position, la cinématique inverse propose de «tirer» sur l'objet vers un but pour calculer le meilleur mouvement possible afin d'atteindre ce but.

Pour mettre en pratique la cinématique inverse, l'humanoïde de synthèse posséde des *ancres* sur lesquelles l'utilisateur pourra «tirer» pour tester les contraintes qu'il aura définies. Si le modèle d'humanoïde de synthèse ne posséde pas d'ancres, l'utilisateur aura la possibilité d'en ajouter lui-même sur les articulations de son choix.

Afin de faciliter son utilisation, l'Éditeur de Contraintes est doté d'une *IHM* (Interface Homme-Machine) réalisée en GTK+ à l'aide de Glade. Ceci permet d'offrir à l'utilisateur une interface standardisée, dans laquelle il sera intuitif d'interagir.

L'Éditeur de Contraintes est le seul outil de notre Projet Professionnalisant en Équipe qui avait déjà été partiellement implémenté. Il proposait déjà des fonctionnalités d'importation et de sauvegarde d'humanoïdes de synthèse à partir d'autre plateforme de développement, d'édition de contraintes sur un et deux degrés de liberté ainsi que de la gestion de la cinématique inverse via les ancres du squelette.

Notre travail a donc consisté à améliorer cet éditeur afin qu'il puisse proposer les fonctionnalités suivantes :

- **Gestion des contraintes liaisons sphériques à trois degrés de liberté :** Comme pour les contraintes à deux degrés de libertée, l'utilisateur à besoin d'une représentation claire d'une contrainte sphérique afin de pouvoir la manipuler et l'éditer intuitivement.
- **Implementation d'une fonction Défaire/Refaire :** Cette fonction devenue standard dans tout éditeur, permet à l'utilisateur de facilement controler son utilisation de l'environnement d'édition. Dans notre cas, cela facilite grandement la souplesse d'utilisation de l'éditeur.
- **Interaction avec l'environnement virtuel** : Permettre à l'utilisateur une interaction intuitive avec son environnement virtuel, et une manipulation aisée de l'humanoïde. Ceci grâce à un affichage plan/perspective de cet environnement, et à une «refonte» complète de la gestion de la souris dans cet environnement.

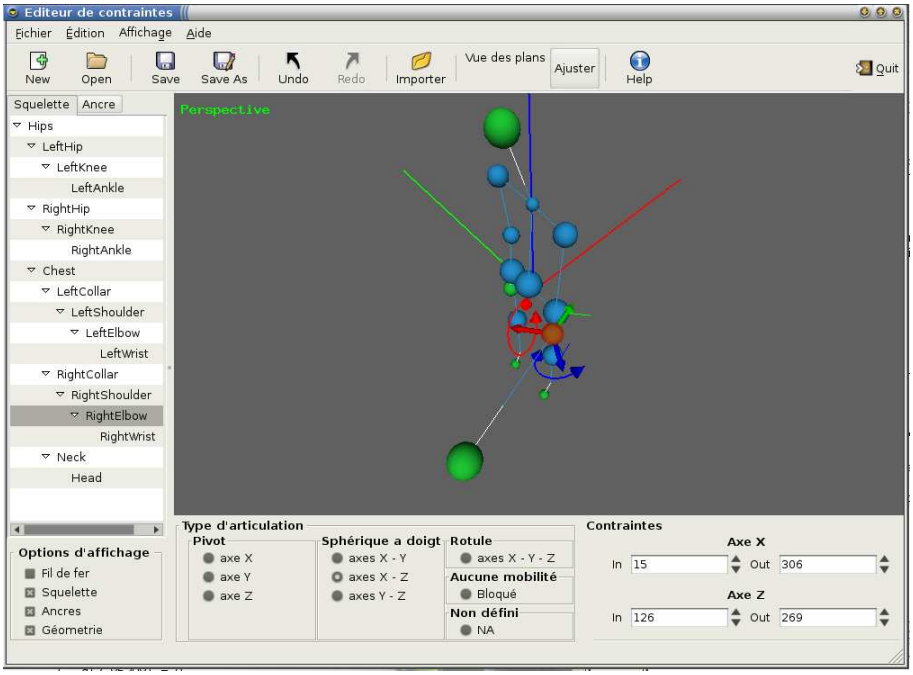


Fig. 3.2 – Utilisation de l'Éditeur de Contraintes

3.2.2 Mise en Œuvre :

Amélioration de l'IHM

L'éditeur permet en premier lieu à l'utilisateur d'ouvrir un projet qu'il a déjà commencé ou d'importer un nouveau modèle sur lequel il souhaite travailler.

Une fois le modèle chargé, l'éditeur crée une arborescence des articulations dans l'onglet «Squelette» ainsi qu'une liste d'ancres, si le modèle en propose, dans l'onglet «Ancre». De plus, le modèle est chargé dans la fenêtre de visualtion ARéVi de l'éditeur.

L'utilisateur peut dès lors commencer son travail sur le modèle. En sélectionnant une articulation, il peut en définir les contraintes à l'aide du panneau de configuration des contraintes. Pour tester son résultat, il lui suffit de manipuler les ancres du squelette pour visualiser si la contrainte ainsi ajustée correspond à son attente.

Si le modèle ne possède pas d'ancres ou si l'utilisateur souhaite en ajouter, il peut créer à partir de l'onglet «Ancre» de même il peut en supprimer s'il le souhaite.

Enfin, l'utilisateur pourra auvegarder son travail en un «Serialized humanoid» (extension : hlh) qui peut être repris dans n'importe quelle application basée sur hLib.

La première partie de notre travail a consisté à améliorer l'IHM déjà implémentée en ajoutant une barre de menu, une barre de status, à compléter la barre d'outils, et celle d'édition de contraintes

Nous nous sommes rendu alors compte qu'il manquait un certain nombre de fonctionnalités qu'on retrouve généralement dans les logiciels classiques. Nous avons donc décidé d'implémenter une fonction de Défaire/Refaire afin que l'utilisateur final puisse revenir sur ses pas en cas d'erreurs ou si le résultat obtenu ne le satisfaisait pas.

Fonction : Défaire/Refaire

À chaque fois, que l'utilisateur ajoute ou supprime une ancre, le squelette entier ainsi que ses ancres est enregistré pour pouvoir être restauré. Il en va de même pour la manipulation des contraintes mais dans un souci de gain de place en mémoire, le squelette et les ancres ne sont sauvegardés qu'une fois qu'une autre articulation est sélectionnée. Il nous a donc fallu mettre en place un algorithme de test pour savoir si le squelette devait être ou non sauvegardé.

Fig. 3.3 – Undo / Redo

La fonction Défaire/Refaire a été réalisée en gérant une pile de ArRef<MemoryBlock>. Chaque ArRef<MemoryBlock> contient les informations relatives à l'état précédent du squelette.

On y stocke :

- le squelette (ArRef<Skeleton>)
- les joints ainsi que leurs contraintes (respectivement ArRef<Joint> et ArRef<JointConstraint>)
- les ancres (ArRef<Anchor>)

Fonction : Vues Séparées

Nous avons aussi implémenté une fonction permettant à l'utilisateur final de visualiser l'humanoïde de synthèse soit en une seule vue en perspective, soit en vues séparées selon les plans xy, zy, zx et la perspective. Cette fonctionnalité permet aussi à l'utilisateur d'avoir une plus grande maîtrise sur ses interactions de manipulation dans l'espace de l'humanoïde de synthèse. Si l'utilisateur interragit avec le squelette en manipulant ses ancres dans une vue de côté, le mouvement résultant à cette manipulation s'effectuera uniquement dans le plan associé.

Fonction : Cadrage des Vues

L'implémentation de cette nouvelle fonctionnalité a soulevé un autre problème : l'utilisateur devait régler le centrage de la caméra pour chaque vue. Avant l'ajout de cette fonctionnalité, ce problème n'en était pas vraiment un, puisque l'utilisateur ne disposait que d'une vue. Nous avons donc réalisé une fonction de positionnement et de centrage automatique de la caméra selon la taille de l'humanoïde.

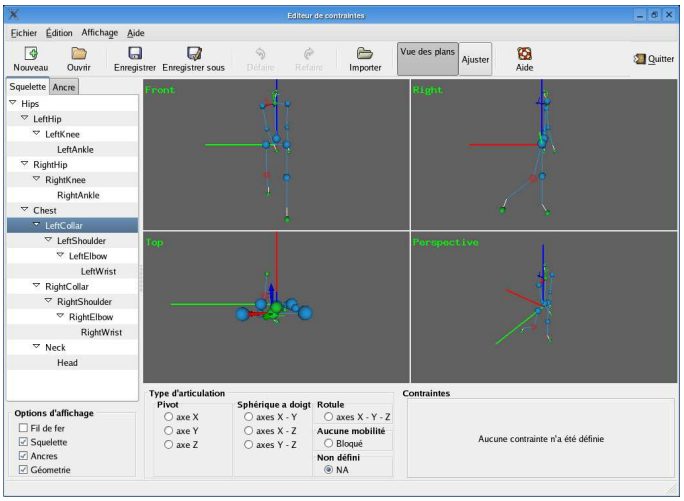


Fig. 3.4 – Éditeur de contraintes : vues séparées

Refonte de l'Interacteur

L'interacteur est la fonction qui permet à l'opérateur d'interagir sur les objets d'un environnement virtuel, ici un humanoïde, et la caméra. L'interacteur d'origine ne convenait pas car il s'agissait d'une classe dérivée de *SimpleInteractor*, fournie par ARéVi, qui n'était pas adaptée à la manipulation des humanoïdes de synthèse. Nous avons donc conçu un nouvel interacteur afin de fournir à l'utilisateur une manipulation aisée et intuitive de l'humanoïde de synthèse.

Ainsi, l'interacteur propose désormais les fonctionnalités suivantes :

- **bouton gauche :** Si l'objet qu'on actionne est une ancre, l'utilisateur peut manipuler l'humanoïde avec un traitement de cinématique inverse en gardant le bouton de la souris enfoncé.
  Si l'objet actionné est une articulation, l'utilisateur sélectionne alors cette articulation et peut l'éditer . La vue est alors centrée sur l'articulation.
  S'il n'y a pas d'objet, en gardant le boutton de souris enfoncé, l'utilisateur effectue un déplacement libre de la caméra dans l'environnement virtuel, en déplaçant la souris.
- **bouton du milieu :** L'utilisateur déplace la caméra en translation x,y dans le plan de vue perpendiculaire à l'axe de la caméra, en gardant le bouton de souris enfoncé.
- **molette :** L'utilisateur peut zoomer ou dézoomer en actionnant la molette de la souris.
- **bouton droit :** L'utilisateur, en cliquant sur une articulation, centre la vue sur celle-ci et la sélectionne et, en gardant le boutton de souris enfoncé, effetcue une rotation autour du modèle.

Si l'utilisateur ne dispose pas d'une souris à molette, voir d'aucune souris du tout, il sera grandement ralentit dans l'utilisation du logiciel, mais pouras néamoins utiliser les flèches de son clavier pour se dèplacer dans l'environnement ARéVi.

Lors du changement de version de la librairie hLib (qui n'est pour l'instant qu'une version alpha), nous avons perdu la fonctionnalité de cinématique inverse. Dès que l'utilisateur définit une contrainte sur une articulation, celle-ci se bloque. On ne peut donc pas, pour l'instant visualiser le résultat obtenu sur les mouvements d'un humanoïde en définisant des contraintes sur les articulations. Il faudra attendre une version plus aboutie de la librairie hLib révisant son algorithme de cinématique inverse pour pouvoir utiliser correctement LÉditeur de Contraintes. Cependant, toute l'architecture de la librairie ayant été mise en place, il suffira d'installer la nouvelle librairie pour pouvoir avoir de nouveau accès à cette fonctionnalité au sein de l'*Éditeur de Contraintes*.

Contrainte liaison sphérique à trois degrés de liberté

La librairie initialement fournie ne proposait pas la gestion des contraintes sphériques (à trois degré de libertés). Quand la nouvelle version de la librairie *hLib* a été disponible, notre travail a consisté à intégrer cette nouvelle fonctionnalité de la librairie dans l'éditeur.

De plus, les prototypes des fonctions ayant changé pour toutes les contraintes, nous avons du adapter notre code à la nouvelle version de la hLib.

Les contraintes à trois degrés de libertés servent pour toutes articulations dont les mobilités ne peuvent se définir à partir de seulement deux axes. En effet, les articulations comme l'épaule ou le poignet ne peuvent être définies par des contraintes à un ou deux degrés de libertés mais ont cependant des contraintes. C'est pour cette raison que hLib propose désormais un système afin de modéliser ces contraintes.

La définition d'une contrainte sphérique se fait à partir de d'une *portion de surface sphérique*, que l'on délimite à l'aide de plusieurs points reliés par des segments. Ainsi, on peut facilement restreindre la marge de mouvement d'une articulation à une marge complexe, chose que l'on ne peut pas faire même avec trois axes de contrainte. Les libertés de mouvement sont définies en déplaçant les extremités de cette portion de surface sphérique. La portion de surface une fois définie représente l'espace dans lequel la partie avale de l'articulation peut se dplacer.

L'implémentation de cette nouvelle fonctionnalité s'est fait par l'intermédiaire des classes *C3DOF*, *C3DOFSh* et *KeyOrientation*. *C3DOF* (Constraint 3 Degree Of Freedom) contient toutes les informations fournies par le *JointConstraint3DOF* (la contrainte à trois degrés de liberté du joint) et est représenté à l'aide des classes *C3DOFSh* et *KeyOrientation*.

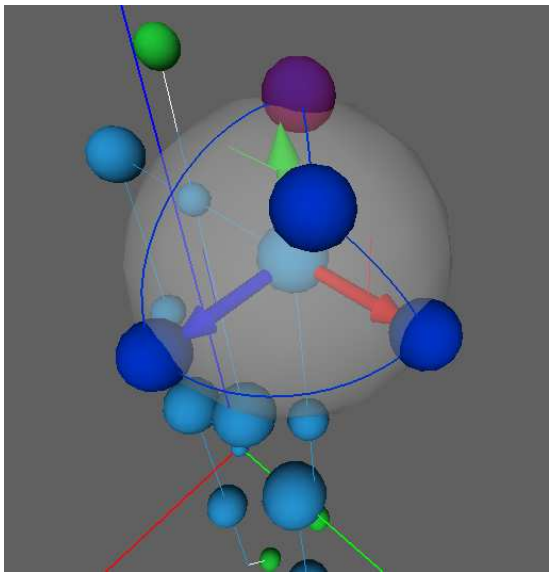


Fig. 3.5 – Edition d'une contrainte 3 degrés de libertée

3.2.3 Aspects techniques

Fonctionnement de l'interacteur du l'Éditeur de Contraintes

Un interacteur est un ensemble de fonctions appelées lors d'un évènement clavier/souris. Il s'agit d'interpréter ces actions vis-à-vis de l'environnement dans lequel ces actions ont lieu. Par exemple, faire le lien entre les coordonnés (x,y) d'un click sur la fenêtre AReVi, et l'action que ce click va produire dans l'environnement virtuel (qui lui est en 3D).

Pour ce faire, la quasi-totalité des environnements graphiques (3D ou non) sont basés sur l'utilisation des signaux (ou évènements) produits par les périphériques d'entrée.

Chaque fois qu'une touche est pressée, relâchée, que la souris est en mouvement, ou qu'il y a un click, un event (évenement) est activé. Cet évènement est connecté à une fonction via l'utilisation des CALLBACKS, une macro ARéVi, de sorte que lorsque l'évenement est lancé, la fonction correspondante soit appelée.

A chaque évenement souris, un «lancé de rayon» est effectué. L'origine de ce rayon définit par le point de vue (la caméra), et sa direction passe par la position du curseur. A la fin du lancé de rayon, nous recupérons le premier élément que ce rayon a intersecté, et suivant la classe de cet élement, une action est effectuée, selon que l'évènement soit un mouvement seul de souris, un click, ou un clik + mouvement.

Par exemple, lors d'un mouvemet seul au dessus de la fenêtre AReVi, nous modifions le curseur suivant l'objet trouvé pour informer l'utilisateur de la possibilitée d'effectuer une action a cet endroit. Si l'utilisateur clique à cet endroit et reste appuyé, lors du mouvement suivant, l'élement visé va voir sa position modifiée dans l'environnement AReVi, de l'origine du click, jusq'à la nouvelle position de la souris interprété en une position 3D dans l'espace d'AReVi.

L'interacteur est au final une des interface principales entre l'utilisateur et son environnement 3D. Il est primordial de fournir à ce niveau un ensemble intuitif et efficace d'action, sans quoi l'application dans sa globalité perd en facilité d'utilisation.

Sérialisation : Sauvegarde et Undo/Redo

La sauvegarde de l'humanoïde se fait par la méthode de sérialisation de la classe *Body*.

Le principe de la sérialisation est de sauvegarder les attributs d'une instance d'une classe vers un flux de donnée.

La méthode de sérialisation de la classe *Body* appelle récursivement la méthode de sérialisation de la classe *Joint* pour chaque articulation qui va elle-même appeler la méthode de sérialisation de la contrainte associée.

Les informations ainsi sauvegardées sont donc :
- l'humanoïde,
- la position de sa chaîne d'articulations,
- les contraintes de chaque articulation
- les valeurs de chaque contrainte.

La fonctionnalité Undo/Redo de l'Éditeur de Contraintes passe par la classe UndoRedo. Cette classe posséde deux piles de sauvegarde des paramêtres :

```
StlDeque<ArRef<MemoryBlock> > _undoStack;
StlDeque<ArRef<MemoryBlock> > _redoStack;
```

Les ArRef<MemoryBlock> sont obtenus lors de la sérialisation de l'humanoïde. La classe *Body* de hLib posséde ses propres méthodes de sérialisation.

Exemple de sérialisation pour la classe UndoRedo :

```
//allocation d'un espace mémoire
  ArRef<MemoryBlock> memoryBlock = new_MemoryBlock();

//calcul des dépendances résultantes pour la sérialisation de l'humanoïde
  SerializationDependencies dep;
  body->extractDependencies(dep, true);
  dep.addReference( body );

//lien entre l'espace mémoire et les dépendances via un flux de sortie
  ArRef<MemoryOStream> output = new_MemoryOStream(memoryBlock, 0);
  dep.writeToStream(output);

//sérialisation des références des objets contenus dans les dépendances
  const StlVector<ArRef<ArObject> >& refsOut = dep.getReferences();
  for(size_t i = 0; i < refsOut.size(); i++) {
    refsOut[i]->serialize(dep, output);
  }
  _undoStack.push_back( memoryBlock );
```

Pour la récupération des données, on utilise les méthodes de désérialisation de la classe *Body*.

La récupération des ancres se fait par le biais d'un test de classe sur chaque objet désérialisé. En effet, le squelette ne fait pas la différence entre une articulation et une ancre.

Exemple de désérialisation pour la classe UndoRedo :

```
//récupération des dépendances d'un MemoryBlock via un flux d'entrèe
    SerializationDependencies dep;

    ArRef<MemoryIStream> input = new_MemoryIStream(_undoStack.back(), 0);
    dep.readFromStream(input);

    anchorCtrl->clearAnchors();

//réccccupération des références des objets contenus dans le MemoryBlock
    const StlVector<ArRef<ArObject> >& refsIn = dep.getReferences();
```

```
//désérialisation et récupération des ancres
    StlVector<ArRef<Anchor> > anchors;
    for(size_t i = 0; i < refsIn.size(); i++) {
      refsIn[i]->unserialize(dep, input);
      if(refsIn[i]->getClass()->isA(Anchor::CLASS())) {
         anchors.push_back(ar_down_cast<Anchor>(refsIn[i]));
      }
    }

    bodyCtrl->setBody( ar_down_cast<Body>(refsIn.back()) );
```

3.2.4 Contraintes d'utilisation

Cette partie présente la façon dont il faut utiliser l'Éditeur de Contraintes. Elle représente aussi la façon dont nous avons conçu le logiciel.

Importation des modèles d'humanoïdes

L'utilisateur a le choix soit d'importer un modèle dévelloppé sous une autre plateforme avec la fonction *Importer* de l'Éditeur de Contraintes, soit ouvrir un modèle du format *hLib* via sa fonction *Ouvrir*.

Dès lors, le modèle est chargé et apparaît dans la fenêtre de visualition ARéVi. L'arbre des articulations et la liste des ancres éventuelles sont elles aussi mise à jour.

L'utilisateur peut alors commencer l'édition des contraintes des articulation de l'humanoïde.

Édition des contraintes d'articulation

Cette fonctionnalité passe tout d'abord par la sélection de l'articulation sur laquelle l'utilisateur souhaite définir les contraintes.

Pour ce faire, l'utilisateur dispose de deux méthodes :

- L'utilisateur peut sélectionner l'articulation de son choix dans l'arbre des articulations. Cette solution est surtout valable pour les squelettes disposant d'un nombre limité d'articulations ayant une nomination claire et sans ambiguité.
- L'utilisateur dispose aussi de la fonction de sélection de l'interacteur : il lui suffit de sélectionner l'articulation de son choix directement dans la fenêtre de visualition ARéVi.

Une fois l'articulation sélectionnée, l'utilisateur a accès au panneau de définition des contraintes. L'éditeur de contraintes offre la possibilté à son utilisateur de définir les contraintes selon les trois axes de l'espace virtuel, ses trois plans, de définir une contraintes sphérique (définie par une surface) ou encore bloquer l'articulation.

Fonctionnalité Défaire/Refaire

Toutes les informations concernant l'huanoïde sont sauvegardé à chaque ajout ou suppression d'une ancre.

Pour la sauvegarde des informations après la définition d'une contrainte sur une articulation, l'utilisateur doit préablement sélectionner une autre articulation afin de valider la contrainte définie.

La restauration des informations se fait via les fonctions *Défaire* et *Refaire* de l'Éditeur de Contraintes.

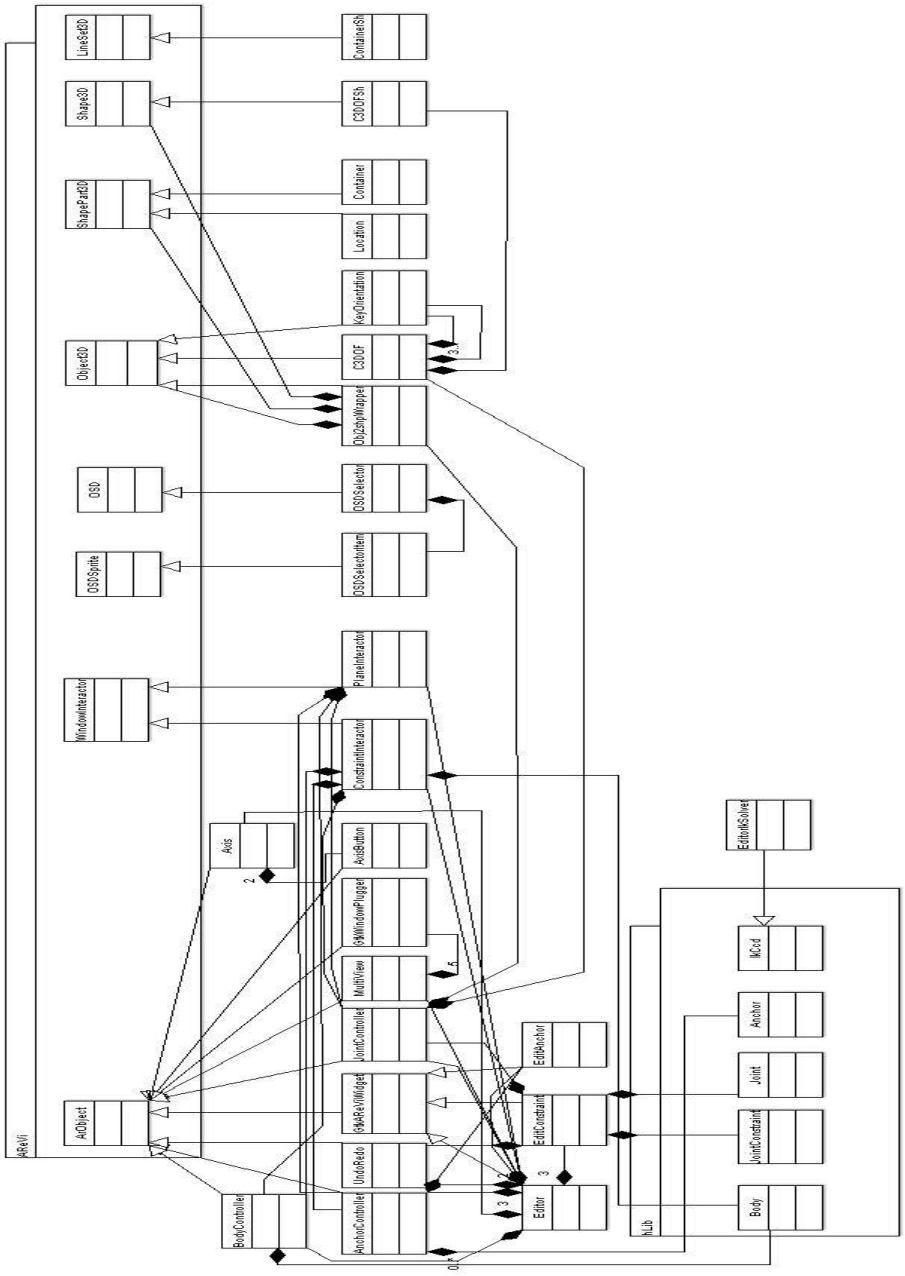
Sauvegarde

L'Éditeur de Contraintes ne peut sauvegarder le travail de son utilisateur que sous le format *Serialzed Humanoid*. En effet, il s'agit du seul format pris en compte par *hLib* disposant de la définition des contraintes sur ses articulations. L'utilisateur passe par les fonctions *Enregistrer* ou *Enregistrer sous* de l'éditeur.

De plus, ce format dispose aussi de la gestion des ancres pour la manipulation du l'humanoïde en cinématique inverse. Et seul le format *Biovision Motion Capture* possède lui aussi cette fonctionnalité.

Le format de hlib offre donc énormément davantage. Cependant, il ne peut être réutilisé que dans des programmes ARéVi.

3.2.5 Diagramme UML



3.2.6 Remarques de développement

Facilités et utilisation d'ARéVi :

L'Éditeur de Contraintes et l'Éditeur de Motion Blending sont essentiellement basés sur l'utilisation d'ARéVi. Cette biblihiothèque propose de nombreuses facilités :

- Un référencement via les templates ArRef< T > ou ArPtr< T > de chaque instance, qui permet en interne l'utilisation d'un Garbage Collector, et qui reste compatible avec les standards C++ (on peut récupérer le pointeur associé à une référence ARéVi). Le Garbage Collector[1] permet de s'affranchir de la suppression des objets autant au niveau des destructeurs qu'au sein des méthodes : son rôle est d'effacer automatiquement de la mémoire les objets instanciés, dès lors qu'ils ne sont plus référencés nulle part. Il suffit alors pour supprimer explicitement un objet de le déréférencer.
- L'utilisation d'une architecture de callback's propre à ARéVi, qui permet de définir entièrement nos propres signaux et handlers associés, de façon à pouvoir utiliser des appels inter-classes facilement.
- La possibilité de faire de l'introspection de classe, c'est à dire d'accéder dynamiquement aux élements de définition de celle-ci, permet d'outrepasser un grand nombre de structures de contrôle, en permettant de tester le type d'un objet et ce dynamiquement.
- Une gestion des activités simplifiée par l'utilisation de la classe *Activity*, qui permet de définir dans le temps l'éxecution de méthodes. Cette fonctionalité est une sorte de simplification des threads, permettant de spécifier et coordonner simplement le déroulement d'un programme. De cette façon, il est relativement simple définir un système multi-agents.

Fabrice Harrouet fournit avec sa librairie un guide d'utilisation sommaire mais essentiel, qui permet de prendre en main facilement les points-clefs de son utilisation.

Globalement, nous avons tenté d'utiliser systématiquement ces particularités d'ARéVi, ce qui s'est fait sans problème particuliers compte tenu de la stabilité de cette librairie. Ces quelques fonctionnalités empruntées à JAVA permettent de simplifier drastiquement l'implémentation du code, et nous ont fait gagner un temps précieux.

Intéraction ARéVi :

Lors du développement du *Constraint Editor*, il nous a parut nécéssaire de reprendre complètement la classe *Constraint Interactor* qui héritait de *Simple Interactor*[2] en surdéfinissant les «handlers» de celle-ci, de façon à ajouter la prise en charge de la cinématique inverse. Or *Simple Interactor* est une classe de tests, qui a une structure très «rafistolée», et qui ne propose que des déplacements de base dans l'environnement virtuel

[1] Appelé Ramasse Miettes : Concept emprunté à JAVA

[2] *Simple Interactor* est une classe d'ARéVi fournissant un intéracteur «de base» pouvant être réutilisé dans n'importe quelle application ARéVi

De façon à pouvoir fournir nos propres méthodes d'intéraction, et ce de la manière la plus claire possible, nous avons complètement repris *Constraint Interactor*, en réutilisant ce qui nous intéressait dans *Simple Interactor*, et en héritant de sa classe de base : *WindowInteractor*.

Dans cet objectif, nous avons implémenté notre utilisation spécifique de la souris (que l'on retrouve en partie dans l'Éditeur de Motion Blending), intégré la gestion de la cinématique inverse, et celle du déplacement des sphères des contraintes 3DOF (3 degrés de libertée). Au final, *Constraint Interactor* est une classe plus légère et plus claire que d'origine.

Cependant, au moment où ces lignes sont écrites, l'intégration de la gestion des contraintes 3DOF peut être mieux implémentée, car l'intéraction «décroche» fréquemment au moment où l'on déplace les sphères de définition du 3DOF.

Les Callback ARéVi

L'Éditeur de Contraintes utilise massivement les Callback pour son bon fonctionnement. Les Callback permettent de déclencher des appels de fonction d'une classe à l'autre sans qu'il y est de relation particulière entre ces classes. Cela permet globalement d'alléger le code et l'instanciation à outrance des classes les unes dans les autres, ce de façon à faire ces appels de fonction.

L'utilisation des Callback allége donc les classes en supprimant un certain nombre d'attributs que nous aurions dû instancier. L'allégement des attributs s'est répercuté sur l'allégement des algorithmes.

3.3 Éditeur de Motion Blending

3.3.1 Description du projet :

Dans le cadre de l'édition d'animations des humanoïdes de synthèse, il est pratique de pouvoir générer de nouveaux mouvements à partir de mouvements déjà définis. C'est dans cette optique que nous avons réalisé l'éditeur de Motion Blending.

Le principe du Motion Blending est de fusionner deux animations entre elles. C'est-à-dire qu'à partir de deux mouvements, hLib est capable de calculer leur moyenne afin d'obtenir un mouvement mixé des deux mouvements précédents tout en conservant le maximum de réalisme dans la cohérence de ce mouvement. L'utilisateur peut affecter des poids aux mouvements qu'il souhaite fusionner afin d'otenir une moyenne pondérée des mouvements ce qui permet une totale maîtrise du résultat.

Par exemple, faire le «blend» d'un humanoïde qui marche, et d'un autre qui lève le bras, donnera un humanoïde qui marche en levant le bras.

Cet éditeur a pour objectif de permettre le «blend» multiple d'animations, c'est-à-dire de pouvoir effectuer ce traitement sur un nombre infini d'animations, et ceci en pouvant bien entendu réutiliser les «blends» d'animations pour fusionner avec d'autres animations ou d'autres «blends».

Pour ce faire nous avons mis en place une surface graphique (gnome canvas) sur laquelle l'utilisateur dépose ses animations, et les lient de façon graphique, ce qui permet une vision claire, précise, et très rapide pour l'utilisateur de ses modifications.

Le Cahier des Charges du Motion Blending Editor comprend donc les fonctionnalités suivantes :

- **Importation d'animations :** l'utilisateur pourra importer ses propres animations qu'il aura défini pour un autre humanoïde sous d'autres plateformes de développement. (formats multiples)
- **Assemblage «graphique» :** construction, manipulation et modification du graphe des «blends» via une représentation graphique des animations, et des liens entre celles-ci.
- **IHM intuitive :** l'utilisation du drag'n drop pour placer les animations sur la surface graphique et d'une barre de coefficient simple pour modifier la pondération des nœuds.
- **Visualisation temps réel :** en donnant à l'utilisateur un rendu temps réel, une indication de sa position dans le temps de l'animation et la possibilité de modifier celle-ci.
- **Sauvegarde / exportation :** l'utilisateur pourra sauvegarder le graphe d'animation complet afin de le réutiliser dans le futur. Il pourra aussi sauvegarder l'animation résultante à son graphe.

3.3.2 Mise en Œuvre

Conception de l'IHM

Comme pour l'Éditeur de Contraintes, l'IHM a été réalisée en GTK+, à l'aide du logiciel Glade. Cette IHM se compose :

- d'un arbre dans lequel on visualise les animations que l'utilisateur a chargé (*gtk treeview*).
- d'une fenêtre de visualisation de l'animation sélectionnée (fenêtre AReVi).
- de boutons classiques de lecture multimédia (lecture, arrêt, pause, début).
- d'une barre de défilement de temps sur laquelle l'utilisateur peut agir.
- d'une barre de contrôle de pondération des animations d'une animation fusionnée
- de boutons et racourcis claviers pour l'édition du graphe d'animations
- d'une zone graphique pour l'affichage et l'édition du graphe d'animations (gnome canvas).

La réalisation de l'IHM est réellement facilitée par l'utilisation de Glade, qui permet en quelque clicks de mettre en place tous les principaux widgets [3] GTK que nous allons utiliser. Cette approche graphique dispense d'effectuer de nombreux tests sur la mise en place de la fenêtre principale, de ses proportions, et de sa cohérence structurelle.

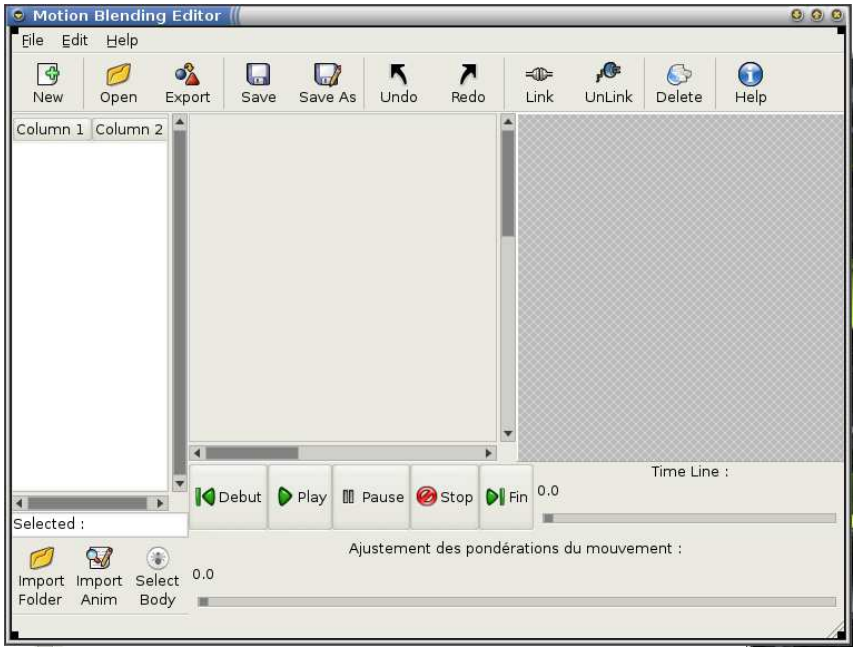


FIG. 3.6 – IHM du Motion Blending Editor.

[3] les objets graphiques de Gtk+ sont appelés des widgets (Window Gadget). Un widget est une structure définissant les propriétés d'un objet associé à une large panoplie de fonctions permettant de manipuler ces objets.

Le Canvas Controller

De façon à avoir une grande rapidité et simplicité d'utilisation de l'Éditeur de Motion Blending, avec lequel nous devons manipuler un nombre important d'animations, il est apparu naturel de se tourner vers une représentation symbolique de ces animations. Dans cette logique, l'éditeur utilise une zone graphique sur laquelle chaque animation est représentée par un carré portant le nom du fichier de l'animation.

En poursuivant cette logique, le lien de parenté entre deux animations et le «blend» qui leur est associé est représenté par un lien (un trait), dont l'existence assume la dépendance du «blend» à ses parents.

Cette symbolisation presque materielle nous permet implicitement d'utiliser les réflexes courants de manipulation des objets, comme par exemple la sélection de cet objet avec le curseur dans le but de le déplacer, ou encore de le supprimer. L'approche visuelle permet via les reflexes d'interprétation communs, une grande intuitivité dans l'utilisation finale de l'éditeur.

D'autre part, de façon à pouvoir simplement selectionner plusieurs objets à la fois sur le canvas, dans le but de les lier, délier ou encore de les supprimer, l'utilisateur utilise les touches *Ctrl* ou *Maj*, qui sont les modificateurs les plus couramment utilisés dans les environements graphiques pour gérer la sélection multiple. De la même façon, utiliser la touche *Suppr* supprimera les objets sélectionnés.

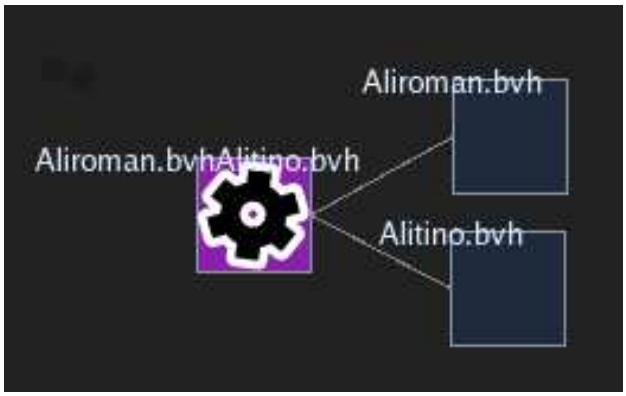


Fig. 3.7 – Symbolisation objet des blends et animations dans le canvas.

Concepts de blending

La fonction principale de l'Éditeur de Motion Blending reste d'effectuer des blendings ou «mélanges» entre différentes animations. Le blending est une technique qui a de nombreuses implémentations différentes, certaines effectuant de simples moyennes entre les positions successives des points-clefs d'un humanoïde à chaque keyframe[4], d'autres plus complexes tiennent compte de la cohérence du résultat, en se basant sur la localité d'un mouvement sur une partie d'un humanoïde.

L'implémentation du blending dans la *hLib* pondère dynamiquement (en interne) chaque articulation de la structure du squelette, de façon à ce que lors du blending, une partie du squelette qui reste statique ne soit pas pris en compte dans le calcul de «la moyenne» des animations. Par exemple,«le blend» d'une animation d'un homme qui lève son bras en restant statique, et d'une animation d'un homme qui marche, donneras un homme qui marche en levant son bras, et non pas l'animation d'un homme qui marche lentement et a petits pas, en levant le bras. Ce qui est beaucoup plus réaliste.

L'Éditeur de Motion Blending exploite la possibilité, fournie par la *hLib*, d'associer à chaque animation une pondération globale, permettant de contrôler le résultat du blend. Il s'agit ici de pouvoir donner la priorité à l'une ou l'autre des animations. Cette pondération se répercute sur l'amplitude donnée à chaque mouvement dans le résultat final. De cette façon, en réutilisant sur plusieurs niveaux cette technique, on dispose au final d'une liberté d'édition totale, et d'une infinité de résultats possibles.

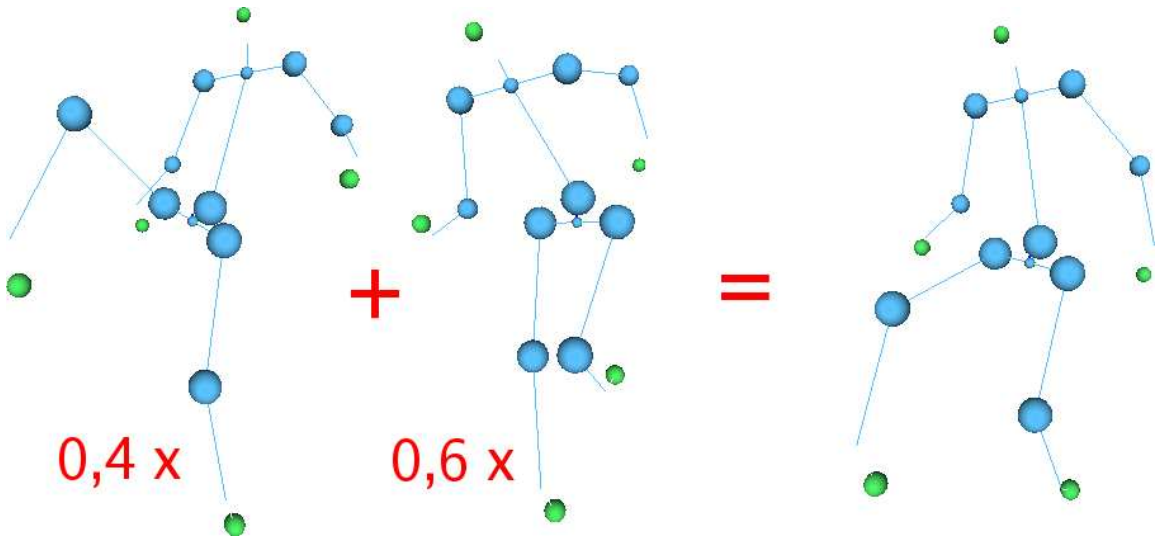


Fig. 3.8 – Blend pondéré de deux animations.

[4]Une keyframe est une image clef. C'est un «instantané» des positions de l'objet en mouvement à un moment donné.

Le résultat final que l'on visionne dans la fenêtre *ARéVi* est un résultat dynamique du «mélange» des deux animations, c'est-à-dire que ce que l'on voit n'est pas une animation en tant que tel, mais un calcul dynamique effectué au moment de l'affichage, du mélange des deux animations.

Si le simple blend de deux animations est relativement simple, le blend d'animations sur plusieurs niveaux est moins évident : il s'agit de réutiliser un blend en tant qu'animation pour créer un nouveau blend.

Dans ce cas, il faut calculer l'animation résultante du premier blend, sous la forme d'une animation, de façon à pouvoir la réutiliser par la suite.

Drag'n Drop

L'interface graphique de notre application se divise globalement en deux partie :
- Sur la gauche on retrouve l'ensemble des outils permettant d'accéder aux ressources (aux animations) dont l'utilisateur part pour créer de nouvelles animations. Par exemple, le *TreeView* qui liste les animations courament chargées.
- Sur la droite l'ensemble des outils permettant l'édition conrète du «blend» des animations. C'est à dire le *gnome canvas*, sur lequel l'utilisateur manipule ses animations.

La façon la plus simple pour l'utilisateur d'intégrer à la volée les animations de son choix à l'environement d'édition est de lui permettre les faire glisser d'un coté à l'autre avec son curseur.

Le *Drag'N Drop* (ou «Glisser - Déposer») est une fonctionnalité fournie par GTK, standardisée, permettant ce «glisser-déposer» entre différents widgets d'une même application, ou entre apllications distinctes, et ce, en associant à chaque type d'objet à manipuler, un format et des méthodes qui lui sont spécifiques. De plus, chaque widget, en tant qu'émetteur ou récepteur, peut implémenter ses propres méthodes pour traiter l'objet en quesion.

L'Éditeur Motion Blending est conçu pour permettre le *Drag'N Drop* entre le *TreeView* regroupant les fichiers d'animation, et le Gnome Canvas (la zone d'édition).

Côté *TreeView*, les méthodes implémentées permettent de fournir à la destination des informations sur le nom de fichier sélectionné ainsi que l'animation qui lui est associée, et côté *Gnome Canvas*, ces méthodes permettent l'appel de la création de l'objet graphique qui lui sera associé, avec pour paramêtres les informations récupérées côté *TreeView*.

Au final, pour l'utilisateur, l'utilisation du Drag'N Drop est de l'ordre du réflexe, et se fait dans une totale transparence. Tous ces «raccourcis» utilisés (interaction dans la fenêtre ARéVi, interactions avec les objets du canvas ...) affranchissent de l'utilisation fastidieuse de boutons sur l'IHM, et réduisent considérablement le temps d'édition.

3.3.3 Aspects techniques

Réutilisation du code existant

L'*Éditeur de Motion Blending* est très similaire à l'*Éditeur de Contraintes* : par exemple, on y gère une liste de fichiers d'animation, une fenêtre AReVi, et le tout repose sur GTK. Le choix de réutiliser la structure de base de l'Éditeur de Contraintes est alors évident. A ceci près que seul le squelette du précédent code reste, de nombreuses nouvelles fonctionnalités y ont été ajoutées, et beaucoup d'autres enlevées.

Globalement, réutiliser le précédent code a deux avantages. Cela a permis de gagner un temps non négligeable, pour rapidement mettre en place les fonctionnalités de l'*Éditeur de Motion Blending*. Et d'un autre côté, cela assure une certaine homogénité du code entre les deux applications.

Structure de CanvasController :

De façon à fournir à notre code une interface de gestion des objets graphiques simple et efficace, nous avons développé les classes *CanvasController*, *Item* et *Link*, de façon à avoir un contrôle total sur la gestion des objets graphiques du canvas.

Ces classes se situent à deux niveaux : *CanvasControler* récupère l'instance du *Gnome Canvas* via *Glade*, et s'occupe de la gestions des objets graphiques (*Item* et *Link*) sur le canvas, de leur instanciation, de leur déplacement... Cette classe fournit ensuite au widget gnome canvas la possibilité d'être une cible de Drag'N Drop, permettant ainsi à l'utilisateur de déposer facilement ses animations. Enfin, elle fournit les méthodes nécessaires à la gestion des objets graphiques sur le canvas, à la gestion des animations ou «blends» associés à chaque objet graphique, et à la gestion des opérations sur ces objets (Link / Unlink / Création / Suppression / Déplacement ...).

Dans cet objectif, les classes *CanvasController* et *Item* implémentent chacune un interacteur. Le *handler*[5] de *CanvasController* permet la gestion de la séléction des *Item*'s, à travers les évènèment «appuit d'une touche» ou «appuit d'un bouton de souris». Le *handler* de *Item* quant à lui s'occupe du déplacement des objets sur le canvas, via les évènement «déplacement et bouton souris».

Les classes *Item* et *Link* fournissent, quant à elles, la gestion des objets graphiques associés à chaque animation et à leurs liens, mais aussi des réferences sur ces animations en elles même, ainsi que différents status de chaque objet (par exemple si cet objet est une simple animation ou un blend).

[5] Un *handler* est une méthode connectée à un signal, invoquée ici pour interpréter les évènèment clavier - souris

La particularité de ces trois classes *CanvasController*, *Item*, et *Link*, est d'un côté d'hériter d'ArObject (C++), de façon à utiliser les macros ArRef< > [6], le garbage collector, les callback's, etc..., et d'un autre côté de travailler sur des instances d'objets gnome canvas, (en C), et ceci de façon transparente : chaque ensemble d'objet (par ex : une animation = 4 lignes + texte + boite) est instancié au sein d'une instance de Item ou Link.

Le CanvasController instancie les Item et Link sous la forme d'une STL MultiMap[7], de façon à pouvoir avoir plusieurs instances d'une même animation sur le canvas.

La suppression de ces instances se fait naturellement avec le Garbage Collector d'ARéVi : il suffit de supprimer toutes les références à une instance pour s'assurer de sa destruction. Les destructeurs sont néanmoins implémentés de façon à supprimer les objets non référencés à l'aide d'ARéVi, comme les objets Gnome Canvas.

Motion Blending :

Un fichier d'animation contient en général la description d'un squelette et les déplacements successifs de chaque articulation de ce squelette. Afin de pouvoir facilement accéder à ces éléments, la *hLib* fournit la classe *LoaderData* qui permet le chargement d'un fichier compatible hLib (format pour lequel il existe un plugin). Une fois instancié, l'accès au squelette ou à l'animation se fait simplement en accédant aux attributs *body* (ArRef<Body>) et *animation* (ArRef<AnimationCycle>) de cette instance.

Du point de vue programmation, la *hLib* propose une API relativement simple pour permettre d'effectuer ces «blend's», à travers la classe *AnimationBlendCycle*[8] qui fournit les methodes nécessaires pour spécifier les animations à «mixer», ainsi que leurs poids respectifs.

De façon à pouvoir réutiliser un blend d'animation comme animation source pour un nouveau blend, la classe *AnimationBlendCycle* fournit une methode *computeAnimation(int nbKeyframes)* qui retourne l'animation résultante aux poids affecté dans le blend. Cette methode prend en entrée le nombre d'images clefs que l'on souhaite avoir dans l'*AnimationCycle* résultante.

Seulement, la méthode *getNbKeyframe*( ) de *AnimationBlendCycle* n'étant pas pour l'instant implémentée (retourne 0 dans tous les cas), nous avons choisis empiriquement de récupérer la durée des animations parentes (qui sont des AnimationCycle), et de la multiplier par 0,04 (1/25 pour 25 image/s), de façon à obtenir le nombre d'image

[6] ArRef<objet> est une macro qui fournit une référence sur objet, et utilise un ramasse miettes (Garbage Collector) pour automatiquement détruire ces objets dès qu'ils ne sont plus référencés.

[7] Conteneur de la librairie STL,d'objets de type pair¡Clef, Donnée¿. Permet d'avoir de multiples occurences d'une même clef.

[8] *AnimationBlendCycle* hérite de *AnimationCycle*, la classe dans laquelle on stocke un animation simple

clef souhaité. Seulement, cette méthode connait ses limites dès que l'on a deux durées d'animation différentes pour chaque animation.

Dans ce cas, l'animation résultante est saccadée. A l'avenir, il suffira d'utiliser les méthodes de la classe *AnimationBlendCycle* pour remédier à ce problème, lorsqu'elles seront implémentèes.

L'intégration du blending au sein du MBE se fait à deux niveaux : au sein de la classe *CanvasController* pour la création du blend (méthode makeBlend()) , et au sein de la classe *Editor*, dans la méthode liée au callback de sélection *_onSetanimCB*() , qui rafraichit au moment de la sélection d'une animation les animations sources (au cas où l'utilisateur ait modifié la podération d'un blend parent à celui séléctionné).

Drag'n Drop :

GTK fournit une interface complète pour utiliser la métaphore du *Drag'n Drop* (DnD) à travers plusieurs protocoles de l'environnement X Window(Xdnd et Motif). De plus, le DnD utilise les signaux de la classe GtkWidget, de façon à ce que l'on puisse implémenter au sein des classes dérivant de GtkWidget nos propres méthodes de support DnD.

L'implémentation du *Drag'N Drop* passe par la définition d'un widget source, d'un widget destination, par la spécification des «types» admissibles par ces widgets (texte, fichier, URL...), et par l'implémentation des methodes nécessaires à la prise en charge en emission et réception des objets passés à travers la procédure de Drag'N Drop.

Le déroulement d'un cycle typique de DnD s'effectue en plusieurs étapes qui correspondent à l'emission successive des signaux suivants :

1. **src : «drag_begin»** au moment au l'objet commence à glisser hors de la source.
2. **src : «drag_data_request»** transfert des informations sur l'objet.
3. **dst : «drag_motion»** à chaque déplacement du curseur sur un widget destination.
4. **dst : «drag_drop»** lorsque la «dépose» a lieu.
5. **dst : «drop_data_recieved»** récupération des données concernant l'objet déposé.
6. **dst : «drag_data_delete»** si ce DnD était un simple déplacement, il faut supprimer la source.
7. **src : «drag_end»** finalise la procédre.

A chacun de ces évènements on connecte nos propres methodes (appellés «handler») à l'aide de *g_signal_connect( )*. Ainsi, chaque émission de signal provoque l'appel de la methode correspondante.

Dans le cas du Motion Blending Editor, la source correspond au *gtk_treeview* (qui liste les fichiers chargés), et la destination au *Gnome Canvas*. Lors du DnD, nous passons

comme information le nom de fichier et l'animation associée, de façon à pouvoir donner une référence sur ces paramètres au sein de chaque instance d'*Item*. Afin de pouvoir représenter un objet graphique sous le curseur dès que celui-ci est au dessus du Canvas, nous avons dû transgresser les conventions du protocole de DnD, en passant dès le signal «drag_begin» les informations nécessaires à l'instantiation de l'*Item* via une méthode du canvas controller.

Sauvegarde / Serialisation

La sauvegarde de l'humanoïde se fait par la méthode de sérialisation de la classe *AnimationCycle* et la classe *AnimationBlendCycle*.

Le principe de la sérialisation est de sauvegarder les attributs d'une instance d'une classe vers un flux de donnée.

3.3.4 Contraintes d'utilisation

Cette partie présente la façon dont il faut utiliser l'Éditeur de Motion Blending. Elle représente aussi la façon dont nous avons conçu le logiciel.

Importation de modèles et d'animations

L'utilisation peut à tout moment importer des animations et des modèles d'humanoïdes en utilisant la fonction *Importer animation* de l'Éditeur de Motion Blending.

L'animation apparaît alors dans la liste d'animations. Si l'animation est associée à un modèle , on peut le visualiser à l'aide de la la fonction *Select Body* de l'éditeur quand on sélectionne l'animation.

Incompatiblités possibles et résultat abhérants

Il est a souligné que l'utlisateur ne peut pas fusionner n'importe qu'elle animation entre elles.

En effet, l'application des animations sur les modèles d'humanoïdes passe par le calcul de la position de chaque articulation du modèle. Un modèle ne pourra donc pas se voir appliqué une animation d'un modèle disposant d'un nombre d'articulations différent.

Si le nombre d'articulations du modèle est supérieur à celui du modèle de l'animation, l'application de cette animation générera un résultat complétement erroné.

Si, au contraire, le nombre d'articulations du modèle est inférieur à celui du modèle de l'animation, l'application de cette animation sera tout simplement impossible à ce modèle.

Enfin, la fusion d'animations de deux mouvements complétement différents donnera souvent un résultat abhérant : en effet, la fusion d'une animation d'un humanoïde qui court et celle d'un humanoïde qui nage donnera un résultat ... mais certainement pas un résultat exploitable pour une animation réaliste d'un humanoïde.

L'utilisateur doit connaître précisément les animations qu'il manipule.

Graphe d'animations

Pour ajouter une animation au graphe d'animations, l'utilisateur doit faire glisser une animation de la liste d'animation vers le *Canvas*.

La sélection de deux animations sur le canvas se fait en gardant la touche *Ctrl* enfoncée pendant la sélection des animations avec la souris. Quand deux animations sont ainsi sélectionnée, l'utilisateur peut actionner le bouton *Link* afin de lier les deux animations et de créer ainsi une fusion d'animation qui apparaîtra comme un nouvel objet du canvas.

En sélectionnant les objets du canvas (qui sont tous des animations), l'utilisateur peut visualiser l'animation sur le modèle qu'il a choisi et peut, si l'animation est une fusion d'animations, ajuster au mieux son animation.

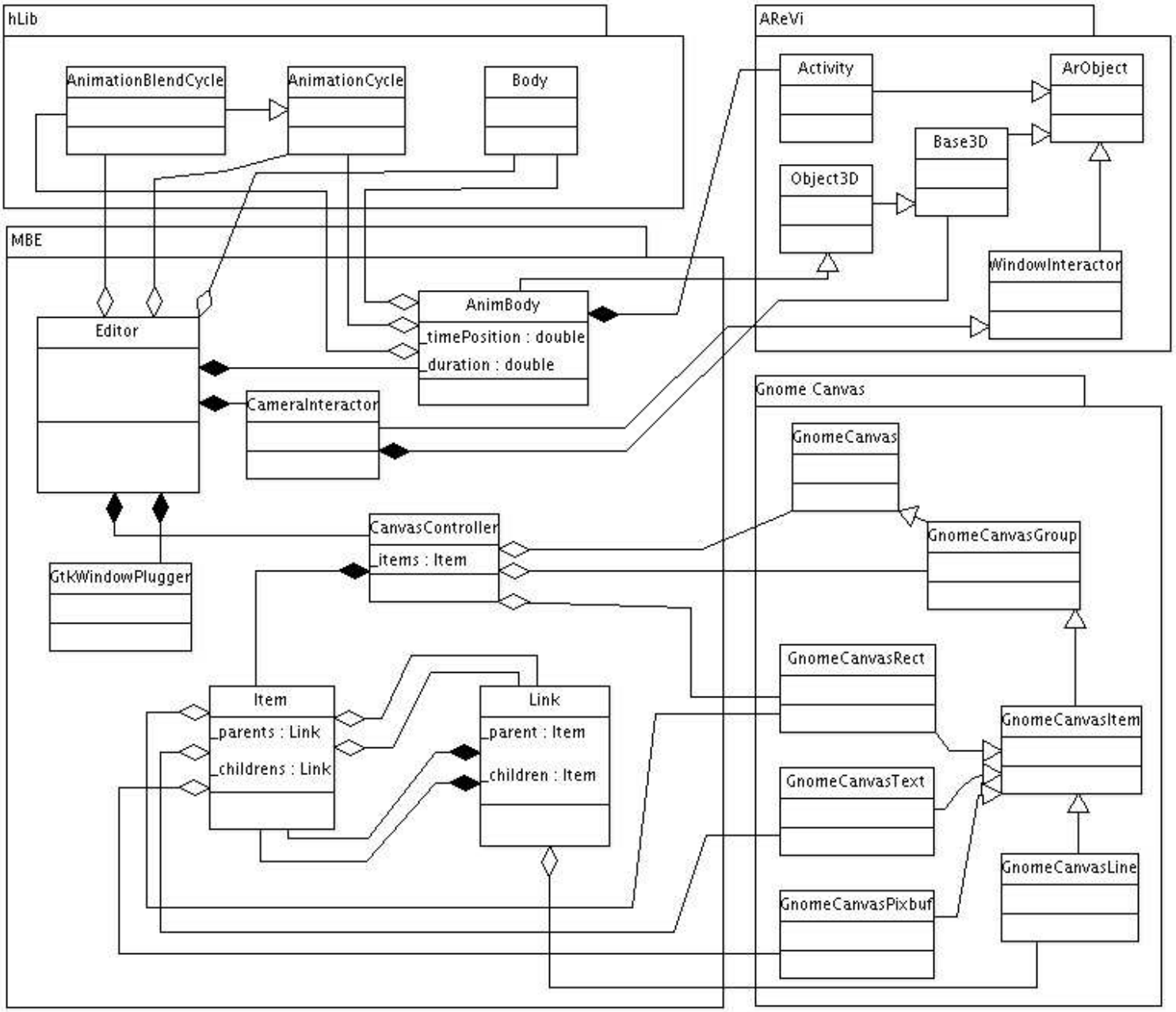
L'utilisateur a la possiblité de modifier à volonté le graphe d'animations. Il peut supprimer des éléments du graphe via la fonction *Supprimer* après avoir préablement sélectionner l'animation.

L'utilisateur doit aussi faire attention aux «blends» d'animations basés sur d'autre(s) blend(s) d'animations. En effet, s'il obtient un résultat qui lui convient pour son dernier «blend» et qu'il modifie les «blends» précédents, son blend final en sera affecté. C'est la raison pour laquelle nous conseillons à l'utilisateur d'utiliser la fonction *Unlink* de l'éditeur afin de «détacher» son résultat du graphe.

Sauvegarde

Une fois son travail achevé, l'utilisateur pourra sauvegarder le graphe d'animations complet pour pouvoir le reprendre dans le futur mais aussi exporter l'animation résultante du graphe pour l'appliquer à d'autres applications.

3.3.5 Diagramme UML



3.3.6 Remarques de développement

L'intégration de *Gnome Canvas* à notre projet n'a pas été très évidente, cette librairie étant très superficielement documentée. Dans un premier temps, l'utilisation le la librairie d'abstraction *GnomeCanvasMM* fournissant une API C++ à *GnomeCanvas* originellement en C, à permis de commencer rapidement l'implémentation de l'ensemble des classes. Cepandant, dans un souci d'homogénéité et de compatibilité avec GTK (qui repésente une part non négligeable du code) , nous nous sommes tournés «après coup» vers *Gnome Canvas* (en C).

L'utilisation de *Gnome Canvas*, qui «dépend» au sens objet du terme de GDK [9] a impliqué l'intégration de l'initialisation de *libGnome* de façon à pouvoir utiliser ce widget.

Nous avons rencontré un problème de capture du focus par notre widget *gnome canvas*, qui empéchait d'utiliser des évènements autres que «event», ce qui nous a bloqué dans la mise en place du Drag'N Drop.

Nous avons contourné ce problème en appelant systèmatiquement la méthode *gtk_widget_grab_focus( )* à chaque appel de l'interacteur du *CanvasController*, ce qui n'est pas viable à long terme. Cela empêche la capture du focus par les autres widget et surcharge le poids de cette fonction appelée extrêmemment fréquement.

A terme, il serait judicieux de ne proposer plus qu'un seul et unique interacteur, de façon à limiter la redondance de code et à rendre plus cohérent l'ensemble des classes *CanvasController* - *Item* - *Link*. Fusionner ces deux interacteurs permettrais aussi de pouvoir implémenter un «carré de séléction» que l'on retrouve dans l'utilisation des bureaux Windows et Linux. Nous n'avons pas pu implémenter cette fonctionnalité au sein de *CanvasController* à cause de ce double handler pour le signal «event», car il est apparement impossible de bloquer ou déconnecter le handler de *Item* lors de l'appel de celui de *CanvasController*, ce qui au final ferait tracer un «carré de séléction» lorsqu'on déplace un objet.

[9] GDK est une couche d'abstraction qui permet a GTK+ d'utiliser de nombreux systèmes de fenètres, et de fournir de nombreuses facilités d'affichage à GTK.

Chapitre 4

Epilogue

4.1 Bilan du projet

Ce Projet Professionnalisant en Équipe prend place à travers des domaines de l'informatique qui nous tiennent particulièrement à cœur, et nous a séduit par sa nature concrète, complexe et pluridisciplinaire. Il requiert à priori de bonnes notions d'informatique, d'IHM, et de RV, que nous ne possédions qu'à travers notre culture générale.

Nous avons entrepris ce projet sans l'ensemble des connaissances préalables qu'il nécessitait, ce qui nous a poussé à questionner et explorer ce que nous ne maîtrisions pas. C'est à travers cette démarche de recherche, d'expérimentation et d'auto-formation que nous avons eu le plaisir de découvrir, et de mettre en pratique ces différents domaines. Cette apprentissage par la pratique nous sera certainement très profitable non seulement pour les cours à venir, mais aussi dans nos futurs projets.

Concevoir ces éditeurs nous a permis d'envisager le développement dans un objectif aboutit. Nous devions au final proposer des applications fonctionnelles, relativement stables, et dans l'anticipation de leur utilisation finale par un utilisateur normal. Le fait d'avoir put mener ce projet jusqu'à son terme est une réelle satisfaction.

Ce PPE a été une experience et une occasion unique de développement, de part son cadre, entouré des enseignants chercheurs du CERV qui nous ont aidé lorsque nous avions besoin d'informations, de part son sujet, qui étais extrèmement motivant à nos yeux, mais aussi de part sa nature conrète, presque palpable, lié à la manipulation des environnements virtuels.

4.2 Perspectives

L'ensemble des Outils d'Édition d'Animation d'Humanoïde de Synthèse permet déjà une grande liberté d'édition à l'utilisateur final. Cependant, il manque encore à ce projet une certaine maturité sur le long terme, au sens ou nous n'avons pas eu l'occasion de les faire tester par des utilisateurs pour en avoir leur impression, ni l'occasion de tester la stabilité de nos éditeurs sur le long terme.

L'Éditeur de Motion Blending quant à lui sera un atout majeur dans la future utilisation de l'Éditeur de Motion Graph. Nous espérons sincèrement que ce projet sera poursuivis, car l'ensemble des trois éditeurs offrira au final une solution complète et simple d'outils d'édition d'animations, qui donnera une grande liberté de composition aux personnes travaillant sur des environnements virtuels, de façon à leur permettre d'intégrer le plus rapidement possible à leur projets des animations très variées.

En ce sens, ces trois éditeurs peuvent devenir à terme des outils fréquemment utilisés pour de nombreux développeurs.