Prototype

SOMMAIRE

Introduction

Objet

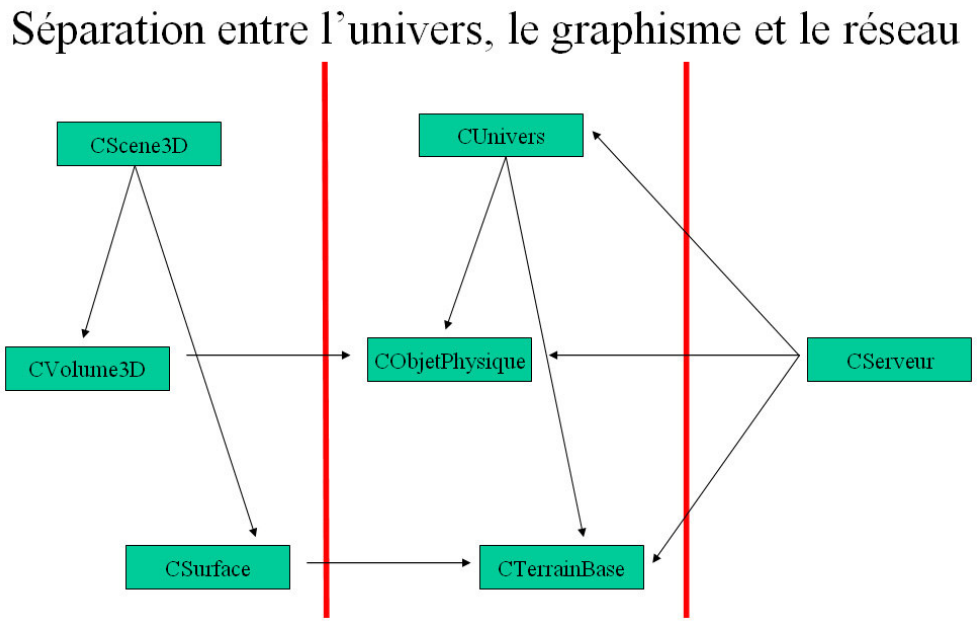
Le but de ce projet est de réaliser un RTS (Real Time Strategy), un jeu de stratégie temps réel. Pour cela nous allons réaliser un mini moteur de jeu pourvu de diverses fonctionnalités. Ce moteur de jeu se nomme G3C.

La plus simple définition de G3C consiste en l'énonciation des termes qui composent son nom : il s'agit d'un noyau générique de jeu GNU. GNU, parce qu'il est lui-même sous licence GPL et que tout jeu basé sur son utilisation devra posséder une licence compatible. Générique, parce qu'il fournit des fonctionnalités de très haut niveau pouvant servir d'épine dorsale à n'importe quel type de jeu se déroulant dans un univers 3D.

L'ensemble des fonctionnalités qui propose G3C se résume en deux modules : un gestionnaire de mémoire et un simulateur physique. La principale caractéristique du gestionnaire de mémoire est qu'il permet aux objets d'être partagés sur plusieurs machines. De façon plus concrète, il met en place une architecture réseau permettant d'avoir plusieurs univers en parallèle sur des machines distantes. Le simulateur physique, lui, permet de créer et de faire évoluer des objets dans un univers 3D, en respectant un certain nombre de règles. On compte parmi ces règles un détecteur de collisions, mais il peut en exister beaucoup d'autres. Ce sera à la charge du programmeur final de choisir lesquelles utiliser et lesquelles ignorer, voir même lesquelles ajouter. En sus de ces deux grosses parties, un certain nombre d'utilitaires sont fournis, parmi eux une intégration avec OpenGL et des outils d'interface avec GTK 2.

Description globale du système

Voici, dans les très grandes lignes, la structure générale du projet :



Description technique

Cette partie décrit tous les aspects techniques essentiels abordés lors de ce projet.

Structure des processus concurrent

Le programme peut se subdiviser en trois sous-programmes qui tournent en parallèle : l'univers, l'interface graphique et le réseau.

L'univers est en fait composé d'une boucle infinie qui appelle à intervalles réguliers des fonctions d'activation pour les règles et les objets. La granule de temps de cette fonction est assurée par une fonction usleep. On réalise ainsi une attente passive.

L'interface graphique est directement gérée dans le programme principal. C'est la boucle d'évènements GTK.

Le réseau concerne uniquement les réceptions d'informations (on écrit directement quand on en a besoin), ou les connexions. Dans le cas d'un serveur, il y a plusieurs threads au niveau réseau, un par client, plus un qui écoute les demandes de connexions.

Ces trois threads accèdent de façon asynchrone à l'architecture de mémorisation. Il est cependant impossible de réaliser une modification dans l'un pendant qu'une lecture est

opérée sur l'autre. On met donc ces opérations en exclusion mutuelle, via un simple sémaphore.

Proposition sur les règles

On introduit dans cette version la notion de règles physiques, modélisées par une classe du même nom. Une règle est capable de faire quatre choses : initialiser les variables d'un objet physique, en contrôler les modifications (et éventuellement les refuser), effectuer des opérations après modification d'une variable et réaliser une suite d'instructions à chaque cycle pour chaque objet.

Bien que pour l'instant, on n'utilise cette structure que pour la gestion des déplacements, on peut imaginer de gérer les dégâts de la même façon, ou une règle sur la pesanteur ou la durée de vie des objets...

Dans le cas qui nous préoccupe, la gestion des déplacements fournit principalement deux services : dans la fonction run (appelée à chaque cycle), on déplace les objets d'un pas. Dans la fonction before, lorsqu'elle est appelée pour contrôler un changement de position, on lance le gestionnaire de détection de collision et, si collision il y a, on interdit le déplacement.

Le problème de chargement de classes

On souhaite une modularité très forte, et une extensivité la plus souple possible. Entre autres, on veut pouvoir charger et sauver des objets, et transmettre un ordre de création d'objet à travers le réseau. Pour transmettre une telle création, il faut que le programme de réception sache quelle classe créer, cf sur quelle classe appeler l'opérateur new. Le réseau doit donc transmettre une information de type.

Lorsque l'on connaît l'ensemble des types susceptibles d'être transmis, on peut les énumérer et leur associer un identificateur. En réception, on choisit le constructeur en fonction de cet id. Dans notre cas, la fonction de chargement est au dessus des classes à charger. En d'autres termes, si les dites classes connaissent cette fonction, la fonction ne sait pas, à priori, quel est l'ensemble des classes qu'elle recevra. De plus, le nombre de ces objets est variable, n'importe quel programmeur pouvant en ajouter à tout moment. On a donc deux problèmes : donner un id unique et automatique à chaque type, et transmettre son constructeur à un objet de plus haut niveau.

Pour le premier problème, il existe en C++ une fonction typeid (classe) qui renvoie un objet type_info, possédant une primitive name. Cette primitive renvoie une chaîne de caractères qui correspond au type de l'objet utilisé dans les résolutions de liaisons dynamiques. Cette chaîne est unique. Le défaut de cette chaîne est qu'elle n'est pas normalisée, elle change d'une implémentation à l'autre. Cependant, les noms étant relativement proches d'une version à l'autre, un algorithme trivial de recherche de ressemblance suffit à faire le lien.

Pour la transmission du constructeur on va, à chaque déclaration de classe, transmettre un couple (nom => constructeur) à une table globale. Pour pouvoir être effectuée automatiquement (et avant la première instruction du main), la seule solution est l'appel d'une fonction à l'initialisation d'une donnée globale.

Le problème, c'est que si la fonction d'initialisation du tableau global est appelée après celle de certains couples (nom => constructeur), ces derniers seront perdus. Il faut donc garantir que, lors de l'initialisation des variables globales, le tableau soit traité avant le reste. Actuellement, il semble que mettre le fichier objet à la fin de la liste des fichiers liés, pour gcc, suffit à obtenir le résultat escompté. A voir.

Pour simplifier, on fournit deux macros, RTTI_DEC et RTTI_IMP (classe), à placer respectivement dans la déclaration d'une classe et dans le cxx. Ces deux macros déclarent et définissent automatiquement toutes les opérations nécessaires à l'insertion dans le tableau global.

Structure réseau

Il est important de différencier deux aspects qui concernent le travail du réseau. D'un côté, on a le protocole réseau à proprement parlé. On y décide entre autres du type d'architecture, ici client/serveur. On y décrit éventuellement les contrôles réalisés. De l'autre côté, on a une structure de mémorisation qui doit être assez souple pour permettre de réaliser les opérations demandées par la structure de l'univers. Bien qu'il y ai d'évidents liens entre ces deux notions, on essaiera de les distinguer le plus possible.

Principe de mémorisation des objets

La structure de mémorisation que nous devons implémenter dans cette version soit supporter une architecture type client / serveur. On considérera que le serveur effectue les calculs et que le client possède une image du résultat de ces calculs. Pour simplifier, l'image du serveur et l'image du client seront quasiment identiques. On dira qu'elles sont synchrones.

On a créé, pour cela, un outil de synchronisation générique : le segment. Chaque objet, de l'univers est un segment à par entière qui s'occupe de façon autonome de sa synchronisation avec ses images. Les clients et le serveurs ont donc les mêmes segments, mais ils sont passif chez les premiers et actifs chez les seconds.

Suivant leur rôle, ces segments ont des modes de synchronisation différents. Le plus classique est Actif2Passif (lire actif to passif). Il concerne les segments qui calculés au niveau du serveur, puis mis à jour dans le client. Le mode Passif2Actif concerne les segments qui envoient des messages du client au serveur. Lorsqu'un client veut faire bouger une unité par exemple, il ne peut pas effectuer de déplacement directement. Il doit demander au server, serveur qui effectuera l'action demandée. Le mode Passif2Passif concerne un certain nombre d'informations à broadcaster, un éventuel chat par exemple. Enfin, le mode Non concerne les segments à usage interne, ceux qui ne peuvent être connus que de leur créateur.

Les opérations de synchronisation que l'on peut opérer sur un segment sont : Initialisation, Création, Retrait et Mise à Jour. Retrait est utilisé au moment de la destruction d'un segment, Mise à Jour au moment de sa modification. La différence entre Initialisation et Création est plus subtile. Dans les deux cas, il s'agit pour le segment synchronisé de créer d'autres segments. Initialisation n'est appelée qu'à la première synchronisation, et évite certains cas où les segments existent déjà. Création est appelée tout au long de la vie du segment.

Chaque objet qui souhaite rentrer dans cette structure rentrer dans cette structure de synchronisation doit être dérivé d'un objet CobjetSegment qui implémente tous ces mécanismes. Il informe de son souhait par un appel à SynchronizeData, qui réalisera entre autres un appel à la fonction virtuelle WriteData, écrite dans le segment en question. Cette fonction WriteData est appelée à chaque fois que des données sont synchronisées, c'est à dire également lors des initialisations et des créations.

La requête du retrait d'un segment et son retrait effectif est différé. De la même manière, sa création et son intégration dans l'univers risquent de l'être également et, bien que ce ne soit pas encore le cas dans la version courante, il faut considérer qu'entre une demande de synchronisation et une synchronisation effective, il peut se passer plusieurs autres évènements. Ceci est lié au fait que les segments ont beaucoup d'autres opérations annexes à effectuer (comme attendre des messages venant du réseau) qui peuvent décaler le moment de leur synchronisation.

Les segments doivent être adressés. Lors d'une mise à jour par exemple, on y accède à l'aide d'une adresse du type 0.1.2.5. Ces numéros représentent chacun un aiguillage à un niveau de l'arborescence. Ils sont attribués dans l'ordre de création. Lorsque l'on détruit un objet, on crée un trou, lequel trou est comblé en priorité lors de la création suivante. Clients et serveurs doivent impérativement être en accord sur leurs adresses. Or, les clients sont autorisés à créer des objets. Ils ne peuvent cependant pas les adresser eux-mêmes, car rien ne garantit qu'un autre client ne souhaite pas aussi en créer un au même endroit. C'est donc au segment actif que l'on demandera une adresse. L'objet ne sera intégré à la structure qu'après réponse du segment actif. Cette interrogation diffère donc le moment où l'on peut utiliser le segment du moment de sa création.

Le système des segments fournit également un support pour les évènements. On peut écouter un évènement en créant une classe écouteur (handler) ou en lever une à tout moment avec un appel à RaiseEvent. Les évènements sont cependant transmis plus tard, lors d'un appel à la fonction DispatchEvents. C'est également à ce moment que l'on détruit effectivement les objets pour qu'à la transmission des évènements de destruction l'objet existe encore. D'où l'aspect différé d'une destruction réelle.

Détections des collisions

Le problème général des collisions consiste à simplifier les tests entre les objets. Sans travail, ou plus exactement dans un espace très réduit, il faut tester tout le monde avec tous le mode, complexité n!. Les algorithmes qui visent à simplifier ce modèle partent tous de la même constatation : il est inutile de réaliser un test entre deux objets que l'on sait trop loin.

L'algorithme des pavés de dégrossissement (c'est nous qui l'appelons ainsi) part du principe que les objets sont en nombre limités (admettons une centaine). Nous allons délimiter notre objet par un cube dons les arrêtes sont colinéaires aux axes X, Y ou Z. On a donc six valeurs, les bornes inférieures en X, Y et Z. On maintient ensuite trois listes d'objets, en X, en Y et en Z. Chaque liste possède les deux bornes (min et max) de chaque objet sur l'axe. On peut donc déceler séparément les collisions sur chaque axe, lorsqu'il y a collision sur les trois, c'est que deux objets sont réellement l'un dans l'autre.

Lorsque l'on déplace un objet, on se contente de décaler ses coordonnées sur les axes et on détecte les collisions au fur et à mesure. Etant donné que les déplacements se font sur d'assez petits pas, il y a très peu de tests à faire.

L'interface

L'étude des besoins

Notre programme d'interface doit se retrouver en première ligne face au joueur. De ce fait, nous avons eu besoins d'un "chef de projet logiciel" qui intègre les besoins de l'utilisateur (le client) et doit les répartir dans son équipe de projet (les autres classes, fonctions …).

Les problèmes

Dès le début, nous avons vu arriver des problèmes futurs :

1) N'importe quelle interface (voir plus tard) doit pouvoir être consultable indifféremment du choix du joueur.

2) Pas toutes les classes ne seront directement impliquées dans l'interface.

3) Le réseau : il faut notifier les changements aux autres joueurs.

4) Mettre des limites aux possibilités d'actions du joueur (jeu par équipe).

5) Rendre indépendant les actions avec l'interface utilisateur.

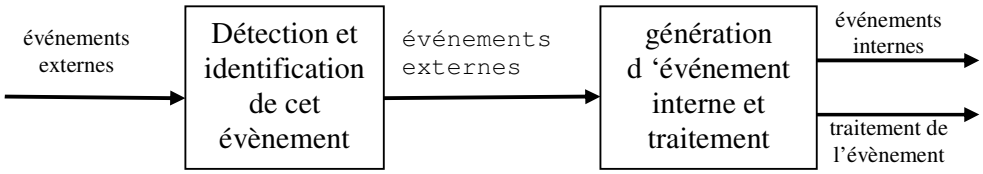
Les solutions

- (1) Créer une classe qui sera le parent de toutes les autres interfaces et faire transmettre un pointeur vers ce parent.

- (2) et (3) Créer des évènements internes que seules les classes concernées récupèreront.

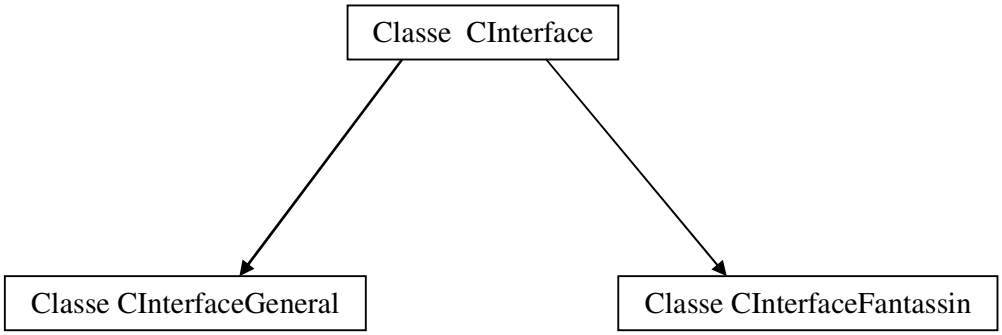
- (4) Mettre en place un système de propriétés qui englobera tous les bons usages de l'utilisateur. Ce système sera externe à l'interface mais sera utilisé par ce dernier.

- (5) Dissocier l'interface en deux sous blocs : utilisateur–interface et interface–univers du jeu.



Les solutions techniques

Pour traiter les évènements externes, nous avons réalisé une classe 'CInterface' qui sera une classe virtuelle pure et père de toutes les autres interfaces souhaitées. Voici notre architecture de classe :



Les évènements que l'on est susceptible de récupérer sont les évènements *clavier*, les évènements *souris* (lorsque l'on clique sur le souris) et les évènements *sourismove* qui sont appelés lors du changement de position de la souris. Ces évènements devront impérativement être implémentés et pour en être sûr, nous les avons mis en virtuelles pures.

De plus, au sein même de la classe CInterface, nous avons implémenté une classe "CObjetPhysiqueHandler" (événement interne lors d'une construction d'un *ObjetPhysique*). Les deux héritiers de cette classe se ressemblent mais sont un peu différentes, i.e que l' "Interface général" dispatche les ordres aux classes concernées et il s'arrête là. La classe "Interface fantassin" doit en plus lever des évènements internes afin de pouvoir gérer le rendu graphique. Voici comment fonctionnent ces deux interfaces :

CInterfaceGeneral

Problème :

Avoir une correspondance entre les objets physiques (là où on les traite) et le graphisme (là où peut agir le joueur).

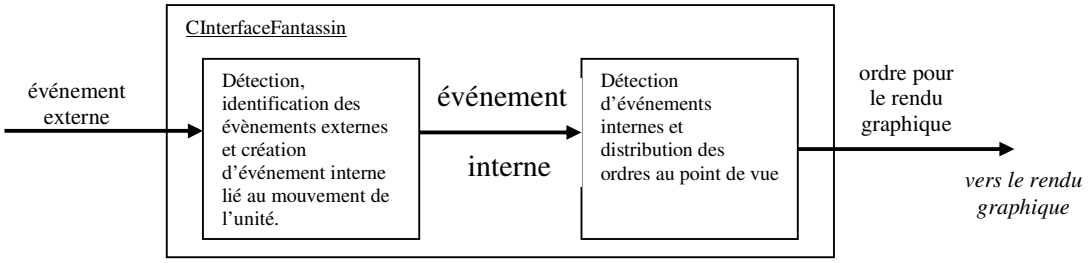
C'est le sens graphisme-physique qui nous pose des problèmes. Pour les événements "clavier", le lien est quasiment fait car on peut les traiter sur un plan physique. Ce sont les évènements "souris" qui sont le dilemme. En effet, les objets bougent et le point de vue bouge aussi donc il est impossible de dire « je clique ici donc c'est cet objet qui est sélectionné » comme on peut le faire dans des graphismes statiques. Nous avons donc du écrire une fonction qui permet cela.

Principe de la fonction :

Nous avons une scène graphique qui se réaffiche souvent. Lors d'un clique souris, on envoie les coordonnées à cette fonction qui met un signal sur ce point. Lors du réaffichage, si ce point est recouvert par un objet physique, la fonction connaîtra alors l'objet sélectionné. C'est grâce à cette fonction que nous pouvons passer de la partie graphique du programme à la partie physique.

C'est sur ce principe que nous avons implémenté la fonction *SelectionerUnite* qui fait partie de la classe CScene3D. Cette fonction prend en paramètre les coordonnées du point cliqué, si il y a quelque chose il renvoie un *CVolume3D* (voir plus tard), sinon il renvoie NULL.

CInterfaceFantassin



Etude des besoins :

Pour ce mode, nous voulions que l'utilisateur donne des ordres à l'unité (ici un fantassin) et que ce soit l'unité qui donne des ordres au point de vue. Nous voulions aussi que les tirs puissent s'effectuer dans toute la fenêtre d'affichage sans interférer le point de vue.

Problème :

(1) Comment faire car les fonctions permettant de réaliser les déplacements voulus ne doivent pas contenir de lien avec le point de vue (compatibilité avec l'interface générale oblige) ?

(2) Comment réussir à changer la direction du tir sans nous rendre incompatible avec les autres modes d'interface ?

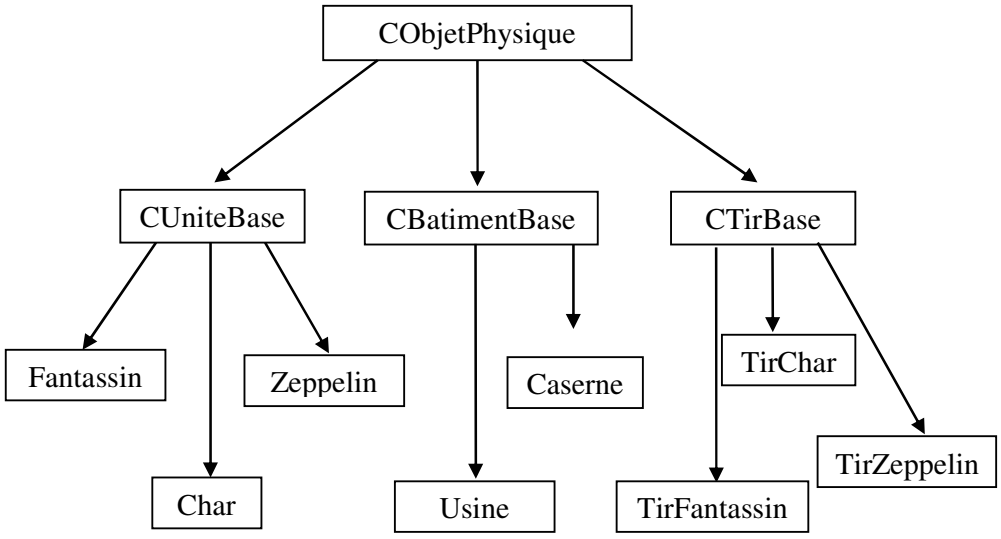
Solutions aux problèmes :

(1) Nous avons posé un Handler (signal d'alerte interne) qui réagit aux déplacements et aux changements de direction de l'unité. Lorsque le Handler est activé (le programme a détecté un changement), il appelle une fonction qui change le point de vue selon la position et la direction du fantassin. Ce Handler n'est écouté que par l'interface fantassin.

(2) La direction du tir est initialisée à la direction de l'unité. Nous lui additionnons un vecteur de décalage qui a été initialisé à nul. Cette méthode satisfait donc l'interface générale mais pas celle de l'interface fantassin. Lorsque le joueur est en mode « 1er personne », quand il clique pour tirer, après un algorithme, modifie le vecteur de décalage qui sera additionné à la direction. Ce principe aura pour effet de tirer là où l'on désire sans interférer avec les autres modes.

Les Unités

Pour créer une atmosphère de wargame comme nous le souhaitions, nous avons implémenté 3 types d'unités : les zeppelins, les fantassins et les chars. De plus, chacun d'entre eux peut tirer. Il existe aussi des usines pour créer des chars et des casernes qui peuvent créer des fantassins et des zeppelins. Ce qui fait pas moins de 8 objets à classer de manière intelligente. Voici notre choix d'architecture :



Etude des besoins

Toutes les unités doivent être indépendantes les unes des autres. Leur partie graphique doivent aussi indépendante de la partie conceptuelle afin de facilité les futurs changements et la transportabilité entre les langages de programmation. Nous voulions également garder une certaine indépendance entre les tirs et les unités.

Problème

Nous ne pouvons pas attendre qu'une unité ait fini son action pour passer la main à une autre unité. Nous ne pouvons pas non plus créer autant de thread qu'il

existe d'unité car trop d'inconvénient et à tous moments, les unités peuvent agir sur les autres. Alors comment faire ?

Comment rendre indépendant la partie logique et la partie graphique ?

Comment déplace t'on les objets ?

Comment rendre l'indépendance des tirs à moindre coût ?

De quoi est constituée une unité ?

Solution

Pour chaque unité, nous implémentons une fonction qui sera appelée à chaque cycle. Ce principe permet de garder toutes les unités dans un même processus et de permettre à tout le monde de réaliser leur action.

Nous avons appelé cette fonction : *Activer*

En ce qui concerne l'indépendance des parties, nous savons que le réaffichage s'effectue souvent. Donc la partie graphique consulte la partie logique pour dessiner. De ce fait, le code pour le graphisme est complètement indépendant du code logique.

Pour pouvoir réaliser un déplacement « naturel », nous avons dissocié le mouvement des unités en deux fonctions :

1/ « avancer » : qui permet à un objet physique de pouvoir avancer.

2/ « tourner » : qui permet à un objet physique de pouvoir tourner.

De plus, nous y avons instauré un paramètre que nous considérons comme un pourcentage de l'action par rapport au total. De ce fait, nous pouvons influer sur ces deux actions primordiales (vitesse de rotation par rapport à la vitesse d'avancement) et réaliser ainsi l'effet escompté.

Le plus souvent, les tirs sont identiques dans le fait qu'ils ont tous une position et une direction initiale. Après cette initialisation, l'unité n'a plus de possibilité d'action sur le tir. Nous avons donc voulu différencier la création, qui a besoins des informations de l'unité d'où il est créé, et sa vie, où il possède ses propres propriétés. Pour ce faire, nous avons implémenté un ensemble de classe (*CGPropertyGun et CPropertyGun*) qui réalise la construction du tir voulu et une classe pour chaque type d'arme qui gère la vie du tir. De cette manière, nous obtenons un bon niveau de généricité puisque nous pouvons affecter les armes aux unités que nous voulons en touchant très peu le code.

Si un nouveau programmeur veut introduire une nouvelle unité, il faudra qu'il renseigne :

- Le type d'arme qu'il veut lui affecter (nouvelle ou déjà créée)
- Sa position initiale

- Sa direction initiale
- La puissance initiale (toujours mettre à 0)
- La puissance maximale (vitesse de rotation ou de progression, en général on l'initialise à 1)
- Les points de vie de son unité
- Son type d'unité (terrestre ou aerien)

Ces paramètres sont indispensables au bon déroulement du jeu. Néanmoins, si le programmeur désire y ajouter une autre variable, il pourra la créer grâce à une fonction à partir de la classe de l'unité et sera vu par tous les objets physiques. En effet, nous avons implémenté une fonction qui nous permet de réaliser cette action : voir et utiliser des paramètres qui ne sont déclarés que dans les sous classes.

Cette fonction s'appelle GetVariable. Cette fonction est générique car on peut avoir besoins de déclarer un *double* ou bien un *vecteur* ou tout autre type.

Exemple d'utilisation

Dans la classe CInfantary

```
CInfantary : :CInfantary (…)
{
   …
   …
   GetVariable <double> (« PtDeVie ») = 100 ;
   …
   …
}
```

Ici *GetVariable* crée une variable PtDeVie (point de vie) de type *double* et l'initialise à 100.

Maintenant, dans le jeu, mon fantassin a reçu un obus. La collision a été détectée entre deux objets physiques.

Dans la classe CTirBase : :CevtCollisionHandler

```
…
{
   …
   CObjetPhysique * objet ;
   CTirBase * Tir ;
   …
   …
   objet->GetVariable <double> (« PtDeVie ») =
      objet->GetVariable <double> (« PtDeVie ») – Tir->GetVariable
<double> (« Damage ») ;
```

}

Comme le montre cet exemple, objet est un objet physique et pourtant on peut accéder à la variable PtDeVie qui a été créée à partir de la classe CInfantary qui est un descendant des objets physiques.

Les Outils mathématiques

Nous créons un jeu en 3D, cela implique que nous devons fournir des outils mathématiques facilitant les opérations sur un point ou un vecteur (les deux ont trois coordonnées dans l'espace donc on les a traités de la même manière). De plus, comme tous bons jeux vidéo qui se respectent, nous voulions que la vue soit dynamique.

Ces besoins nous ont permis de dégager les fonctions dont nous aurons besoins dans l'avenir :

- Surdéfinir les opérateurs
- Pouvoir le normaliser à une longueur donnée

Problèmes rencontrés

Comment rendre la vue dynamique de telle manière que lorsque l'on appuie sur la touche pour aller à droite ou en haut, on aille vraiment dans la direction voulu et non comme OpenGl le voudrait.

Comment calculer un repère relatif avec un vecteur *visee* qui correspondra au vecteur y du repère absolue ?

OpenGl ne détecte pas les « loopings » (garde le sol en bas) mais lorsque l'on appuie sur la touche 'haut', on va en bas.

Solutions

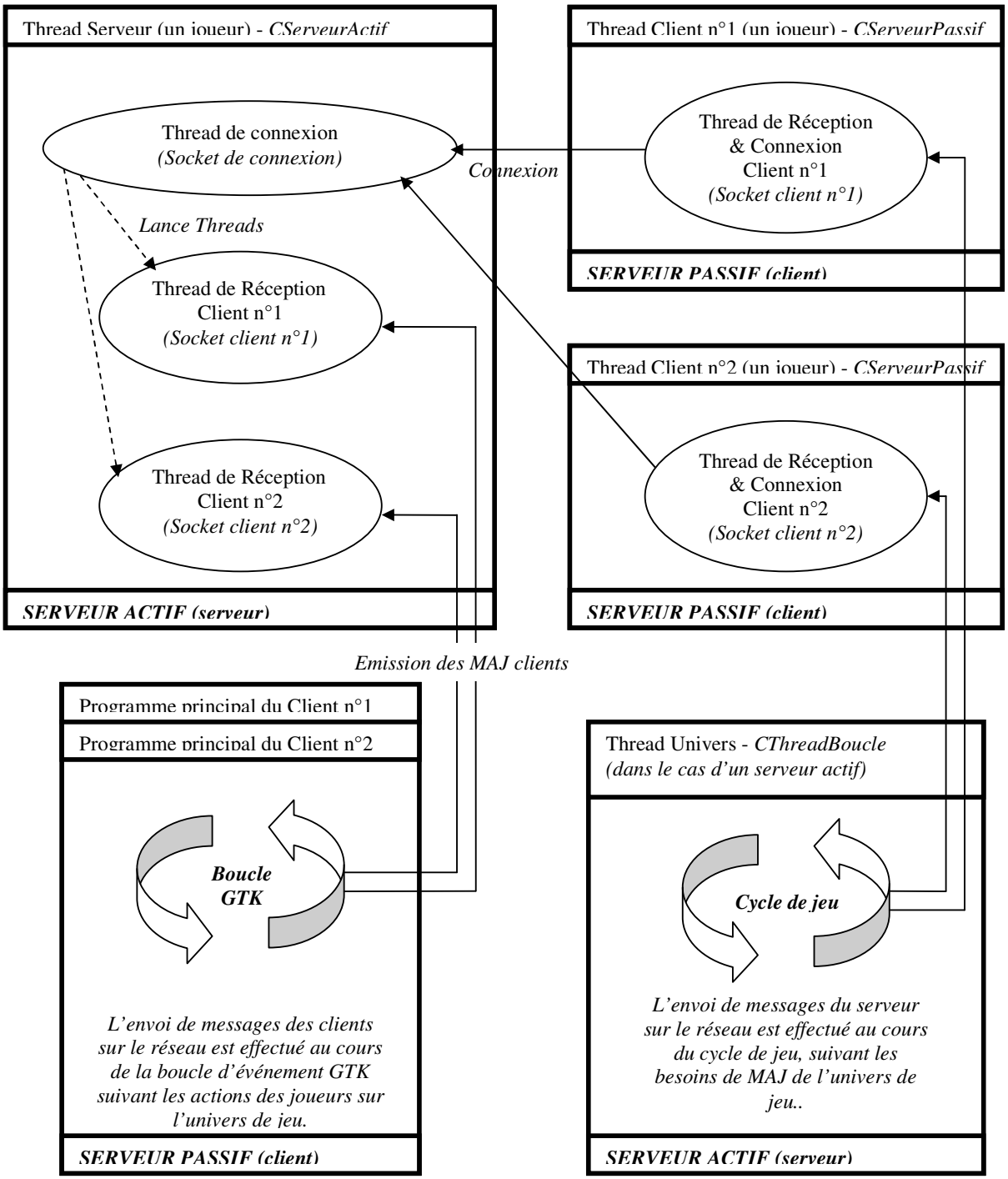
OpenGl ne nous permet que de nous diriger par rapport à un repère fixe (dans notre cas). Il a donc fallu très vite trouvé le moyen, pour le point de vue et pour les calculs par rapport au point de vue, déterminer un vecteur orthonormal au vecteur de visée (direction du regard) et parallèle au plan XY. Pour cela, nous avons implémenté une fonction qui permet de calculer l'angle entre deux vecteurs dans le plan XY. Avec les théorème de trigonométrie, nous avons pu calculer les coordonnées d'un vecteur *droit* (vecteur orthogonal dont la direction est à droite du vecteur appelant). Nous réalisons donc nos déplacements (droite et gauche) selon ce vecteur.

Pour avoir un repère relatif, il a fallu calculer en plus un vecteur *haut*. C'est une fonction qui réalise ce calcul grâce à la trigonométrie dans l'espace (intersection de deux plans : plan orthogonal au vecteur *visee*, plan défini par les vecteurs *visee* et l'axe z).

On n'a du mettre un coefficient qui s'inverse lorsque l'on passe de l'autre côté de l'axe z. De plus, nous avons du implémenter une tolérance pour faciliter les mouvements du joueur dans l'espace.

Architecture réseau

L'architecture réseau développée est indépendante du jeu. Elle fait partie du moteur de jeu G^{3}C.



Remarques sur le serveur :

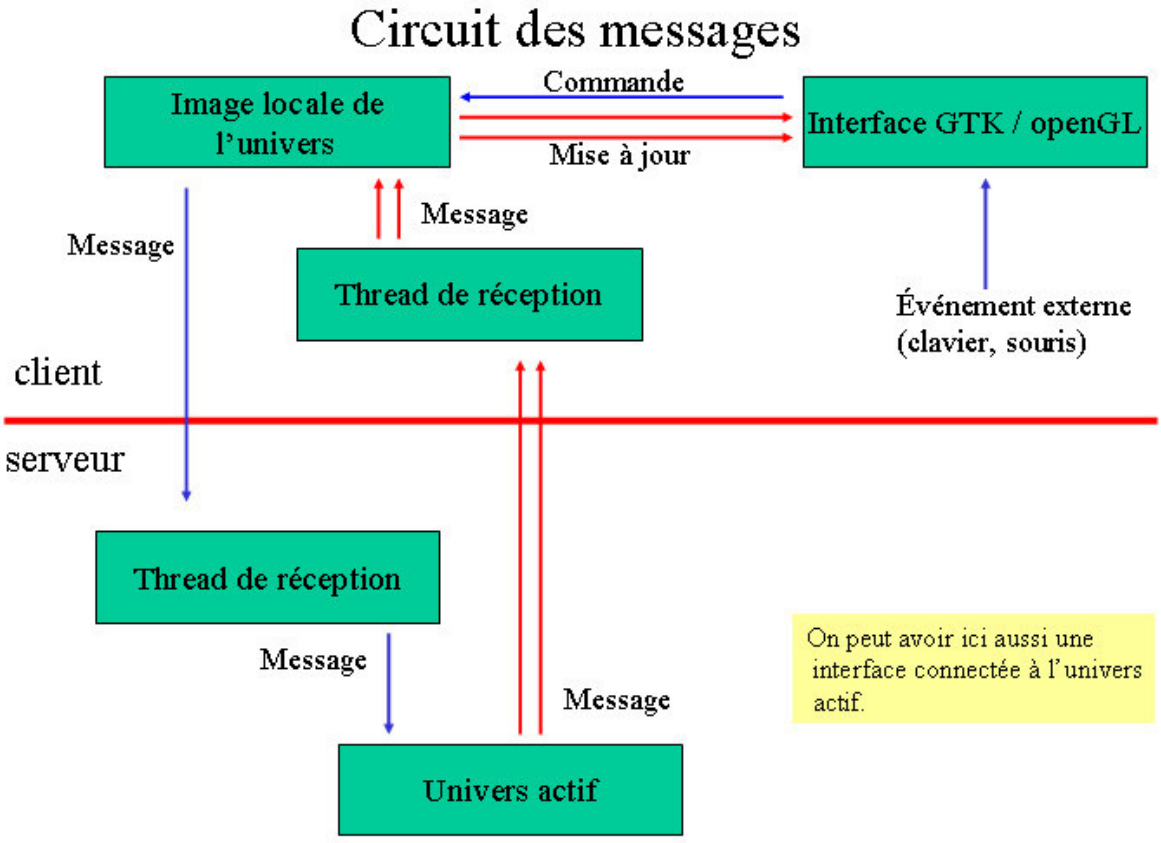
- Le Thread de connexion est lancé au démarrage du serveur.
- Chaque nouveau client qui se connecte au serveur démarre un thread de réception client pour la communication des mises à jour de l'univers de jeu.

Remarques sur les clients :

- Chaque client démarre un thread de réception & connexion qui permet de se connecter au serveur de jeu, puis de recevoir les informations de mise à jour de l'univers de jeu émises par le serveur.

Remarques générales :

Ce diagramme de fonctionnement décrit l'architecture réseau adoptée pour le projet, mais il ne décrit pas l'aspect « synchronisation des informations de l'univers de jeu » envoyées par le réseau.



Gestion de l'affichage en OpenGL dans GTK

L'affichage du jeu est développé dans un module distinct qui initialise le contexte OpenGL dans le thread principal du jeu. Ce module permet d'intégrer un composant GTK (*GtkGLExt - http://gtkglext.sourceforge.net/*) permettant d'afficher une fenêtre OpenGL dans une IHM GTK 2.

Pour des raisons de compatibilités avec la librairie GTK2 disponible sur le serveur galejad, tous les appels de fonctions GTK2 respecte une syntaxe type « C ». En effet la version de GTK2 disponible ne fournissait pas les Wrappers C++ disponible. Pour cela il aurait fallu installer plusieurs librairies afin de faire tourner la librairie « gtkmm » permettant d'utiliser les Wrappers C++ GTK2.

Editeur de propriétés des unités

Notre wargame comprend différentes unités de combat (Tank, Zeppelin et un Fantassin). Chaque unité dispose de règles spécifiques et de propriétés graphiques liées à OpenGL. L'éditeur de propriétés que nous avons développé nous permet de régler facilement ces règles et propriétés graphiques à associer à chaque unité du jeu.

Cet éditeur génère ainsi un fichier pour chaque unité du jeu contenant toutes les caractéristiques de celle-ci. Ce fichier est lu au démarrage du jeu et permet de charger dynamiquement les propriétés de chaque unité.

Conclusion

Ce projet est appelé à être continué, plus exactement à être repris dans un langage qui correspond mieux à notre vision de la programmation objet : Ada 95. Il faut donc considérer la version courante comme un prototype, qui nous as permis de mieux identifier les points clefs de la conception. A suivre maintenant, avec une plus grosse équipe, sur www.sourceforge.net/projects/g3c

Une description exhaustive de l'architecture des classes est disponible sur http://www.esil.univ-mrs.fr/~qochem/g3c/index.html

Christope PAPANDREOU, Marc VENTRE et Quentin OCHEM le 15/01/2004