TER 2009 Groupe J : Mouvement 3D dans les Sims

KOVALEVA Svetlana KOUIDER Sarra LY Paul RYSER Guillaume

11 mai 2009

Ministère de l'Education Nationale

Université Montpellier II

Résumé

De nos jours, le jeu vidéo s'est rapidement hissé à un rang d'activité majeure. L'économie créée autour de cette pratique a eu des conséquences notables sur notre société.

Un des problèmes les plus récurrents depuis l'expansion de cette industrie, comme dans notre cas *les Sims*, est de déplacer un ou plusieurs personnages d'un point à un autre d'une scène 3D sans entrer en collision avec les objets ou les autres acteurs de la scène.

Le rapport suivant traîte donc du déplacement de personnages dans des univers virtuels en 3 dimensions.

Table des matières

Table des figures

Introduction

Des statistiques faites récemment montrent la montée flagrante des ventes de jeux vidéos cette dernière décennie, comme le montre le graphique 1.

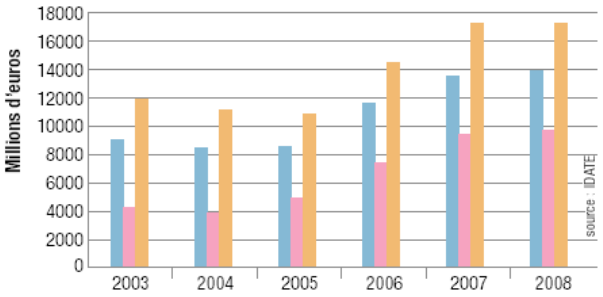

FIG. 1 – Marché mondial du jeu vidéo : Europe en bleu, Japon en rose et USA en orange

Le jeu vidéo le plus vendu au monde reste encore la série des Sims[1], un jeu de simulation de vie, développé par *Electronic Arts* existant aussi bien sur ordinateurs que consoles de jeux.

Rappel du sujet

Notre sujet consiste à proposer une solutions aux problèmes que posent les mouvements de personnages dans des scènes 3D, incluant donc implicitement différents points :

- la recherche d'un chemin du point de départ au point d'arrivée,
- la coordination individuelle qui permet de déplacer les différentes parties du corps en harmonie,
- la coordination collective qui intervient dès lors que l'on cherche à déplacer un groupe,
- la détection de collisions qui est fondamentale dans un univers solide.

Cette étude implique plusieurs domaines comme les mathématiques, la géométrie, la programmation 3D et l'algorithmique, en particulier le *Pathfinding* .

Le but de notre TER est donc de construire une maquette permettant de charger des scènes différentes, et de placer un personnage animé dans la scène choisie afin de le déplacer automatiquement vers un point cible.

Présentation de l'application

L'application consiste à faire évoluer un personnage animé dans un environnement 3D contenant des obstacles, le personnage ayant pour but de les éviter.

Pour se faire, l'utilisateur a le choix entre plusieurs mondes virtuels en 3D. Il sélectionne l'emplacement de départ du personnage ainsi qu'un point d'arrivée, et le personnage se déplace.

Les interactions prévues sont :

- Le choix du personnage
- Le choix de la carte
- Le placement initial du personnage
- La sélection d'un point d'arrivé pour le personnage
- Le choix entre une caméra libre et une caméra de poursuite
- La possibilité d'automatiser les placements de personnages

On notera que la caméra libre s'utilise comme certains jeux de shooter, c'est à dire que la direction est choisie par la souris tandis que le zoom avant-arrière et la translation gauche-droite sont définies par le clavier (touches T,F,G et H). La roulette de la souris définit la hauteur de la caméra.

La figure 2 montre le rendu visuel de l'application.

[1] Tous supports, extensions et versions considérés

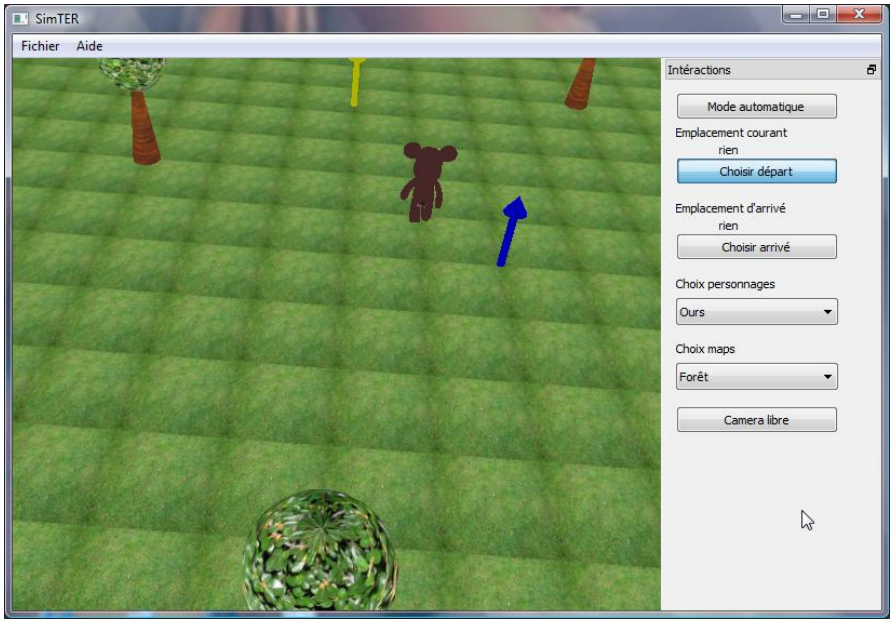

FIG. 2 – Screenshoot de l'application produite

1 Organisation de l'équipe

1.1 Remaniement du diagramme de Gantt

Dans le cahier des charges rendu, le diagramme de Gantt décomposait le travail sous forme de binôme. Après avoir trouvé quelles tâches pouvait convenir au mieux à chacun en fonction de ses acquis et de ses capacités, nous avons pris la liberté de changer totalement notre approche en éclatant le projet en 4 tâches distinctes, qui sont :

- Le moteur de déplacement ;
- L'IHM du projet ;
- L'implémentation des cartes ;
- L'interfaçage entre OPENGL et 3D STUDIO MAX .

Après avoir répartit les tâches entre les membres du groupe, nous avons accordé l'avancement de chacun et l'avancement général du projet autour de réunions et de mailing intensifs.

Toutes les décisions importantes, que ce soit pour choix des structures, du format d'export, ...ont été le fruit de concertations entre tous, ce qui a permis d'assurer une certaine cohésion dans le groupe.

1.2 Répartition des tâches

Les quatre tâches ont été réparties de façon à ce que tous touchent un peu à toutes les approches du projet (algorithmiques et graphiques) : chacun a une tâche dédiée, mais corrige, commente (en bien ou en mal) ou débuggue au moins une autre tâche, ce qui fait que nous obtenons la répartition consignée dans le tableau 1.2. Chacun des chargés de tâche a rédigé sa partie du rapport.

Tâche	Chargé officiel	Correcteurs
Moteur de déplacement	S.Kouider	G.Ryser, P.Ly
IHM	P.Ly	G.Ryser, S.Kovaleva
cartes	G.Ryser	P.Ly, S.Kouider
chargeur / personnage	S.Kovaleva	S.Kouider, P.Ly

Fig. 3 – Répartition des tâches dans le groupe

2 Outils de développement

2.1 Choix des langages

Notre choix en terme de langage de programmation s'est tout naturellement orienté vers le C++, car c'est le langage le plus utilisé quand il s'agit d'utiliser OpenGl .

C++ est un langage de programmation orienté objet, permettant de plus la programmation procédurale ou encore la programmation générique.

Il est actuellement le 3eme langage le plus utilisé au monde, le bloc C/C++ étant classé 1er.

2.2 Outils et librairies

2.2.1 OpenGl

OpenGl (Open Graphics Library) est une API multi-plateforme pour la conception d'applications générant des images 2D et 3D.

OpenGl étant multi-platforme et plus facile d'accès par rapport à DirectX et comme la programmation 3D est une nouvelle expérience pour la plupart d'entre nous, le choix s'est donc naturellement porté sur OpenGL.

2.2.2 La librairie *QGLViewer*

Dans le but d'intégrer un interface homme-machine au TER, nous avons opté pour Qt du fait que c'est la bibliothèque graphique offrant des composants d'interface graphique que nous connaissions le mieux. D'autant plus que Qt comporte des éléments spécialement dédiés à OpenGl grâce à la librairie *QGLViewer*.

2.2.3 Le format d'exportation .md2

Dans le cadre du projet, nous avons adopté le format .md2. Ce format est très intéressant pour une première approche de l'utilisation des animations 3D avec OpenGl . En effet, il permet d'avoir une vue d'ensemble des composants de ce genre de fichiers d'exportation.

Il donne aussi la possibilité de gérer une animation par frames au sein d'une application OpenGl .

2.3 IDE

Les environnements de développement intégré (IDE) utilisés lors de la programmation de notre application sont Dev-C++ et QtCreator.

Dev-C++ est l'un des IDE gratuit pour Windows les plus utilisés. Il possède un débogueur permettant de surveiller l'état des variables pendant l'exécution du programme, et de stopper l'exécution en cours à l'aide de "token".

QtCreator est, quant à lui, un IDE permettant de développer en C++ avec Qt. QtCreator permet (de la même manière que Eclipse) de créer des projets, les éditer, les débugger, de consulter la doc,...

2.4 Logiciel de modélisation

Après hésitation entre plusieurs logiciel de modélisation, notre choix s'est finalement porté sur 3D Studio Max de Autodesk au détriment de *Maya* ou *Blender*.

Puisque aucun membre du groupe n'avait jusqu'alors l'habitude de travailler avec ce genre de logiciel,3D Studio Max nous semblait le juste milieu pour débuter car il répondait tout à fait à nos attentes :

- solution puissante de modélisation, d'animation et de rendu ;
- version d'évaluation gratuite ;
- large documentation sur le net et en bibliothèque.

3D Studio Max est conçu sur une architecture modulaire et supporte des plugins (extensions). Il est principalement utilisé dans l'industrie du jeu vidéo mais également dans plusieurs autres domaines, notamment les films tels que X-Men II, Bulletproof Monk, Kaena la prophétie ...

3 Moteur de déplacement

Il faut savoir qu'en intelligence artificielle on distingue deux grandes classes d'algorithmes de recherche : les algorithmes non-informés (ou aveugles), réalisant une recherche exhaustive et les algorithmes informés, qui utilisent des sources d'information supplémentaires en donnant ainsi de meilleures performances.

En général, on notera l'utilisation de la seconde catégorie pour résoudre le problème de *Pathfinding* vu la complexité de l'espace d'états du problème.

Il existe de nombreux algorithmes de calcul de plus courts chemins. Cette partie du rapport sera dédiée à l'explication de deux algorithmes que nous avons implémentés dans le cadre de ce projet : A* (très utilisé dans les jeux vidéos aux vues de ses performances) et SMA* .

3.1 A*

3.1.1 Principe de l'algorithme

L'idée générale de l'algorithme Astar est très simple : à chaque itération, on tente de se rapprocher de la destination. On va donc privilégier les possibilités les plus proches de la destination, en mettant de côté toutes les autres.

Toutes ces possibilités ne sont pas supprimées, elles sont mises dans une liste de possibilités, explorée si jamais la solution proposée actuellement s'avère mauvaise. En effet, on ne peut pas savoir à l'avance si un chemin va aboutir ou sera le plus court. Il suffit que ce chemin mène à une impasse pour que la solution devienne inexploitable.

L'algorithme va donc d'abord se diriger vers les chemins les plus directs, et si ces chemins ne conviennent pas, il examinera les solutions mises de côté. Ce backtracking nous garantit que l'algorithme retournera, s'il en existe une, une meilleure solution. Cette solution est garantie comme étant la plus courte.

Nous avons choisi de représenter A* comme une classe C++ à part entière, utilisant une classe Graphe, mais n'étant pas une simple fonction de la dite classe.

Pour que A* fonctionne normalement il a besoin de connaître un point de départ, un point d'arrivée, ainsi que le graphe de recherche. L'appel se fera alors comme suit :

Astar a (<Point de départ>, <point d'arrivée>, <graphe>)

A* utilise également deux listes, une liste dite "ouverte" *openList* qui contient les noeuds non visités, et une liste dite "fermée" *closedList* qui contient les noeuds déjà visités.

L'algorithme se présente comme suit :

```
/* Methode principale de l'algo de Astar */
void Astar::algo()
{
        /* on initialise le point de depart */
        Noeud*courant=&m_depart;

        /*on l'ajoute dans la liste ouverte*/
        m_ouverte.push_front(courant);

        /*traitement sur les noeuds voisins */
        Ajout_voisins(courant);


        /* On supprime le noeud de la liste ouverte
        et on l'ajoute dans la liste fermee pour dire
        qu'il a ete deja traite */
        m_ouverte.remove(courant);
        m_ferme.push_front(courant);


        /* tant que la destination n'a pas ete atteinte
        et qu'il reste des noeuds a explorer, faire*/
```

```
        while((!Nidentiques(*courant,m_arrivee)&&(!m_ouverte.empty()))) {
                courant= meilleur_noeud();//recherche du meilleur noeud

                Ajout_voisins(courant); /*ajout des noeuds adjacents*/

                /*suppression du noeud courant la liste ouverte*/
                m_ouverte.remove(courant);
                /*ajout du noeud courant dans la liste fermee  */
                m_ferme.push_front(courant);
        }
        if(!m_ouverte.empty()){
                retrouver_chemin();
        }/*chemin trouve*/

        else{
                cout<<"pas de solution"<<endl;
        } /* pas de solution */
}
```

Lors de la présentation de l'algorithme dans sa globalité deux méthodes importantes n'ont pas été expliquées : Ajout_voisins() et meilleur_noeud().

La première est une simple procédure dont les étapes se résument comme suit :

- On regarde tous ses noeuds voisins
- si un noeud voisin est un obstacle, on l'oublie
- si un noeud voisin est déjà dans la liste fermée, on l'oublie
- si un noeud voisin est dans la liste ouverte et que le noeud dans la liste ouverte est de moins bonne qualité, on met à jour la liste tout en mettant à jour le parent du noeud
- sinon, on ajoute le noeud voisin dans la liste ouverte avec comme parent le noeud courant

La deuxième fonction quant à elle, retourne le meilleur noeud de la liste ouverte avec la meilleure heuristique.

A* combine à la fois les avantages des algorithmes de recherche en coût uniforme et de recherche gourmande (Greedy Search), en utilisant une fonction heuristique h : $f(n) = g(n) + h(n)$

La fonction f estime le coût total du chemin entre l'état initial et l'état solution passant par le noeud courant n. $g(n)$ est la distance réelle entre l'état initial jusqu'au noeud n et $h(n)$ est une estimation de la distance séparant le noeud n de la solution.

Le choix sur la méthode de calcul de g et h est déterminant pour le bon déroulement de l'algorithme. Pour notre application nous avons choisi la distance euclidienne élevée au carré, ce qui évite l'utilisation de la fonction "racine carré", relativement lourde pour le processeur et nous permet de manipuler des entiers et non des nombres flottants. Cette distance est l'une des distances les plus connus et les plus utilisée quand il s'agit d'un terrain plat (ce qui est le cas de notre map).

La distance que nous utilisons est donc donnée par la formule suivante :

$$\sum_{i=1}^{n} (x_i - y_i)^2 \tag{1}$$

Dans le cas d'un terrain contenant des denivelés importants, il nous faudrait utiliser la distance topologique pour la fonction $g(n)$, et la distance euclidienne dans R^3 pour la fonction h.

3.1.2 Analyse

Complet : pour les espaces d'états avec un coût du chemin optimal qui relie le noeud n au noeud solution s avec $f(s) \leq f(but)$

Complexité : exponentielle en temps et en espace.

Optimalité : l'algorithme A* est optimal, il étend toujours les noeuds dans l'ordre croissant des valeurs heuristiques, l'optimalité de A* est garantie si on utilise une heuristique admissible.

3.2 SMA* (Simplified Memory-Bounded A*)

3.2.1 Principe de l'algorithme

SMA* est l'une des variantes de A* qui vise à améliorer les performances de ce dernier. Le fonctionnement est totalement similaire à A* à la différence qu'il effectue la gestion de sa propre mémoire disponible en utilisant une liste ouverte d'une taille fixe, ainsi il élimine les noeuds ayant les valeurs de $f(n)$ plus élevées quand la file ouverte ordonnée est pleine lors de l'ajout.

```
/* le noeud n'est pas present dans la liste ouverte, on l'ajoute */
if (m_ouverte.size()< taille_memoire){
        //la file n'est pas encore pleine on ajoute le noeud directement
        m_ouverte.push_front(new Noeud(i,j,temp.getZ(),N->getX(),N->getY(),
        temp.getH(), temp.getG(), temp.getF()));
}

else{
        //la liste ouverte est pleine
        eliminer_mauvais_noeud(); //supprimer le mauvais noeud
        m_ouverte.push_front(new Noeud(i,j,temp.getZ(),N->getX(),N->getY(),
        temp.getH(), temp.getG(), temp.getF())); //l'ajout du nouveau  noeud
}
```

3.2.2 Analyse

Complet : si l'espace mémoire disponible est suffisant pour contenir le chemin état initial – état solution.

Optimalité : SMA* retourne toujours la meilleure solution pouvant être obtenue avec l'espace mémoire alloué. Le gain est énorme au niveau de l'espace mémoire utilisé, donc la taille du graphe peut être plus importante.

4 Infographie

4.1 Le personnage

4.1.1 Modélisation

Pour modéliser notre personnage en 3D , nous avons utilisé le logiciel 3D Studio Max . Ce logiciel est principalement utilisé dans l'industrie du jeux vidéo et de scènes fixes en 3D. Dans le cadre du projet, ce logiciel nous permet de modéliser notre personnage et ses animations.

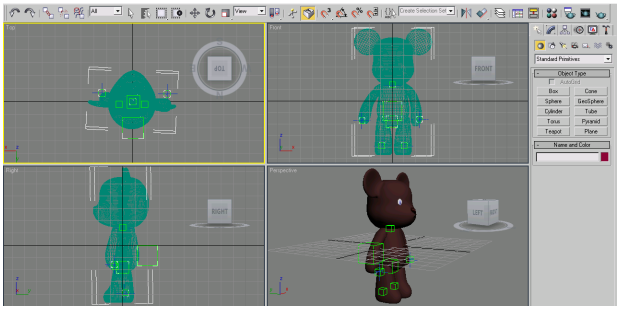

Fig. 4 – Screenshoot du logiciel 3D Studio Max

Dans un logiciel tel que 3D Studio Max la modélisation d'un objet 3D se fait par mesh[2], comme le montre la figure 5.

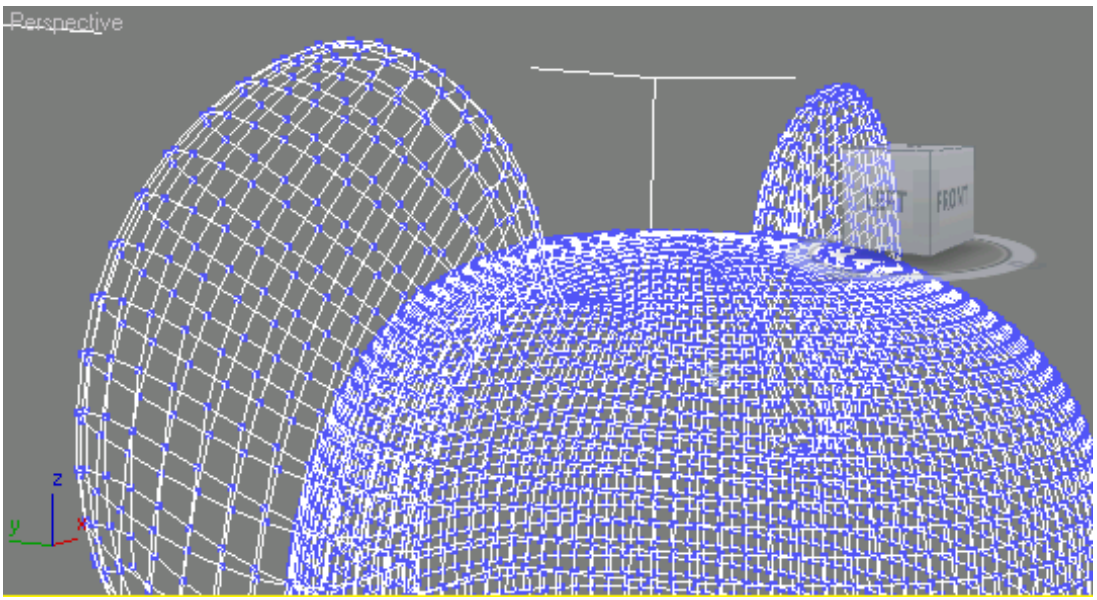

Fig. 5 – Exemple de maillage produit par 3D Studio Max

De cette façon il est possible de modéliser tout sorte d'objets. Plus le maillage utilisé est fin, plus l'objet est détaillé et réaliste.

[2]Mesh, ou maillage : Discrétisation d'un milieu continu. Dans notre cas, 3D Studio Max décompose le personnage en un nombre fini de carrés

4.1.2 Animation

Une fois l'objet modélisé, 3D Studio Max nous permet de l'animer. La technique d'animation utilisée est l'animation par bones[3]. Il s'agit de créer la squelette de l'objet et d'associer ensuite à chacun des bones une partie précise de l'objet. C'est la façon dont l'association entre le squelette et l'objet est faite qui va permettre un mouvement plus ou moins souple.

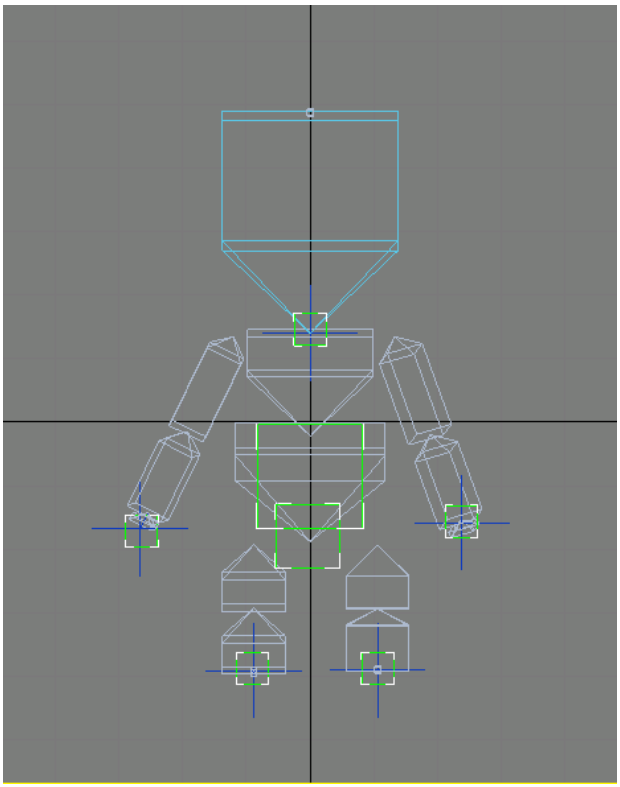

Fig. 6 – Squelette du personnage utilisé lors de l'animation

Pour permettre à notre personnage de marcher[4], il a fallu associer les bones aux contraintes IK Solvers, représentés sur la figure 6 par les croix bleu. Ces contraintes permettent de définir la dépendance des bones dans leurs mouvements[5].

Pour animer un personnage il faut définir ses mouvements par frames. Chaque frame détermine une position de tous les vertices[6] de l'objet à un moment donnée.

Dans cette étape il s'agit de choisir les frames avec un intervalle représentant au mieux le mouvement désiré. Ensuite il faut enregistrer pour chacun d'eux la position du personnage, telle qu'elle devrait être dans l'animation à l'instant donné. 3D Studio Max se charge de remplir les frames intermédiaires.

[3] Bones ou squelette : trame utilisée pour l'animation. Le regroupement de tous les bones forme le squelette
[4] La marche prend en compte divers éléments : plier les jambes, bouger les mains, modifier la posture, ...
[5] Exemple de dépendance : Lorsque le personnage bouge sa main, les bones du bras vont proportionnellement se plier au niveau du coude
[6] Vertices : sommets composant le maillage du personnage

4.1.3 Exportation

3D Studio Max pocède plusieurs formats d'exportation, cependant certains d'entre eux ne prennent pas en charge l'animation tandis que d'autres sont compliqués dans leur utilisation ultérieure (importation dans notre application).

Dans le cadre du projet, nous avons adopté le format .md2[7]. Ce format est très intéressant pour une première approche de l'utilisation des animations 3D avec OpenGl . En effet, il permet d'avoir une vue d'ensemble des composants de ce genre de fichiers d'exportation. Il donne aussi la possibilité de gérer une animation par frames au sein d'une application OpenGl .

Les modèles .md2 sont composés :

- de données géométriques ;
- d'animations par frame ;
- de commandes OpenGl .

Les données géométriques sont les triangles et les sommets du maillage du modèle, ainsi que les coordonnées de texture associées.

Un frame de l'animation correspond à une séquence de l'animation, chacun d'eux contenant les données géométriques du modèle à l'instant donné. Le fichier .md2 est donc composé d'une multitude de frames qui, affichés successivement et rapidement, donnent un effet d'animation.

4.2 Les cartes

4.2.1 Première carte, premier graphe

La première carte, dont nous avons gardé la base pour la carte "forêt" permettait de créer aléatoirement, en fonction d'un graphe donné, une carte 3D avec 5% de chances d'avoir un arbre à chaque noeud. Cette carte avait pour but de nous familiariser avec OpenGl , et a contribué à la création du graphe. En effet, il y avait plusieurs possibilités pour forme du graphe de recherche, chacun avec des avantages et des inconvénients.

Par exemple, un système de "WayPoints" n'était pas assez généraliste, car les chemins étant prédéterminés, le croisement éventuel de deux personnages n'était pas envisageable.

Notre choix s'est porté sur un système de grille. Le graphe est représenté sous forme de matrice, chaque élément de la dite matrice correspondant à un noeud. Si un obstacle se trouve sur ce noeud, on le marque, et ainsi, le personnage peut se déplacer sur tous les neouds non marqués. Le problème de croisement de deux personnages était alors en partie résolu, car si on marque le noeud sur lequel est un personnage, il devient impossible d'en avoir deux au même endroit, et il suffit alors de contourner ce nouvel obstacle.

4.2.2 Le moteur graphique

L'implémentation des cartes telle qu'elle a été faite a pour but principal de faciliter la création et la compréhension de celles-ci.

Les cartes sont consignées dans des **Headers C++**. A l'interieur, le créateur de la carte doit compléter deux fonctions :

- draw[nom de la map]Floor() : Permet d'initialiser le sol de la carte ;
- draw[nom de la map]Obstacle() : Permet d'initialiser les obstacles de la carte, et de mettre à jour le graphe en indiquant les lieux inaccessibles au personnage.

Ces deux dernières fonctions permettront la modélisation des cartes et par conséquent appeleront les différentes méthodes d'OpenGl liées au dessin.(voir B.1)

Elles seront alors appelées dans un autre *header* qui regroupe toutes les cartes : *ChooseMap.h*. (voir B.2)

Les fonctions de *ChooseMap.h* (*drawFloor* et *drawObstacle*) ne sont que des analyses de cas convoquant les fonctions de dessin vues plus haut, qui seront appelées par les méthodes *draw()* et *drawWithNames()* de la classe *MyOpenGL*, l'entier passé en argument correspondant à l'indice de la carte courante.

Ainsi, pour ajouter une nouvelle carte, il suffit de créer un *header* avec les fonctions appropriées et insérer de nouveaux cas.

[7] Le format .md2 a été créé pour le jeu vidéo Quake II

Le personnage est ajouté sur la carte construite à la position courante indiquée par l'algorithme de *Pathfinding* [8] via les méthodes de la classe *Entity* définies plus loin, qui gèrent intégralement le dessin du personnage et ses animations.

L'éclairage est géré par la méthode *lumieres()* qui le paramètre à l'aide des divers méthodes OpenGL conçu pour : *glLightfv, glLightModelfv, glLightModeli.* (voir B.3).

5 Chargeur d'objets 3D

Notre chargeur permet de produire un objet OpenGl en utilisant les données fournies par le fichier .md2.

Comme on l'a vu dans la partie précédente, un fichier .md2 est composé de plusieurs types de données. Ces données sont encapsulées par notre chargeur à la fin de lecture du fichier.

Le code de la méthode *loadModel* (voir B.4) de la classe *Chargeur* présente la fonction principale de ce chargeur : après son exécution toutes les données nécessaires à la représentation du modèle sur l'écran sont enregistrées. Leur utilisation se fait au niveau de la fonction *RenderFrame* (voir B.5).

Cette fonction dessine le modèle d'une frame[9], donnée en paramètres. Elle récupère les coordonnées des vertices pour cette frame, et grâce à la boucle ci-dessous, permet de construire les triangles OpenGl (GL_TRIANGLES) composant notre modèle.

Pour permettre que l'objet s'anime, il faut, à partir d'un dessin de frame, générer une animation, ie. une séquence consécutive de frames affichées à un intervalle précis.

Pour ce faire, on utilise une autre classe, qui d'une part permet de charger les informations relatives au modèle une fois, puis d'afficher plusieurs entités de l'objet sur l'écran, et d'autre part de générer la séquence des frames composants l'animation.

Il s'agit ici de la classe *Entity*. Cette classe contient la fonction *Animate* (voir B.6).

Elle prend en paramètres l'identifiant du premier frame et du dernier de l'animation qu'on voudrait afficher.

Fpercent permet de donner l'interpolation linéaire[10] à utiliser dans l'animation (en pourcentage).

Pour charger notre objet 3D ainsi importé, il suffit de déclarer les deux instances *Chargeur* et *Entity* que l'on place ensuite à l'extérieur de la boucle d'affichage, le fichier .md2 n'étant lu qu'une seule fois, la fonction *LoadTexture*, qui permet de charger les images utilisées pour la texture du modèle. Il suffit ensuite d'instantier une entité du modèle chargé.

Pour finalement afficher notre personnage , il suffit d'appeler les méthodes *animate* et *drawEntity* à l'endroit voulu (dans notre cas dans la méthode *draw* de la classe *MyOpenGL*).

[8] A savoir : A* ou SMA*

[9] Frame ou cadre : Une image d'une animation à un temps donné. Une animation est une séquence de plusieurs frames défilant rapidement.

[10] L'interpolation linéaire dans une animation consiste à afficher des frames supplémentaires calculés à partir de deux frames données. Ce rajout de frames intermédiaires permet un lissage de l'animation.

6 Interface Homme-Machine

L'interface homme machine (IHM) est une interface permettant à une personne non programmeur (ou utilisateur) de manipuler simplement l'application sans pour autant connaître le code derrière.

Nous avons choisi pour implémenter cette IHM la bibliothèque Qt. Qt est une bibliothèque logicielle orientée objet et développée en C++ par la société Qt Software. Elle offre des composants d'interface graphique (widgets), d'accès aux données, de connexions réseaux, de gestion des fils d'exécution, d'analyse XML, ...Ce qui nous intéresse plus particulièrement ici, ce sont les composants d'interface graphique dont une conçue spécialement pour les applications OpenGl : la QGLViewer

6.1 QGLViewer : classe héritée de QGLWidget

La classe QGLWidget est une widget conçu pour le rendu graphique OpenGl . Elle permet une utilisation simplifiée d'OpenGl . Cette classe hérite de QGLWidget est par conséquent apporte encore plus de fonctionnalité pour l'utilisation de la librairie graphique (caméra, animation, gestion de la souris, ...). Pour de plus amples renseignements sur cette classe, ou parcourir les nombreux exemples d'utilisation, il est conseillé de visiter le site [Sita]. Nous utilisons ainsi cette classe pour implémenter l'IHM de notre projet, ce qui nous donne le rendu donné par la figure 7.

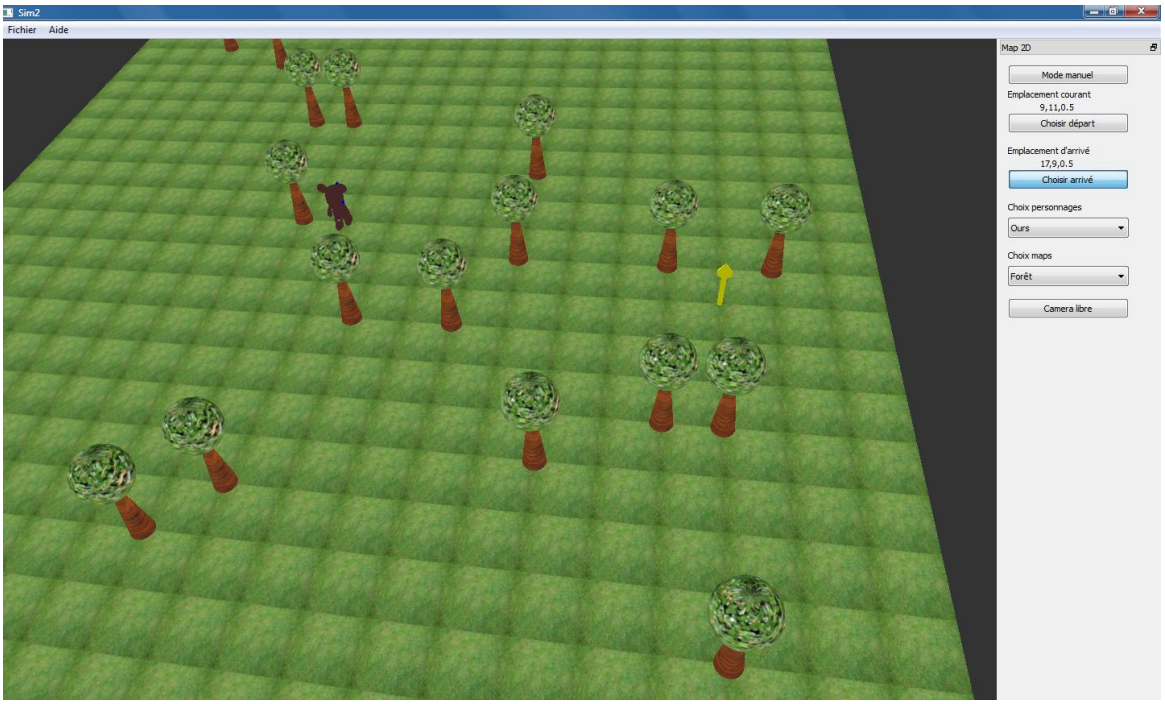

FIG. 7 – IHM de l'application

Nous avons la QGLViewer au centre et autour les différentes interfaces permettant d'intéragir avec l'environnement.

6.2 Les fonctionnalités de l'IHM

L'interface créée permet d'interagir de différente manière sur l'environnement 3D ainsi que sur les personnages. Les différentes options sont montrées par la figure 8.

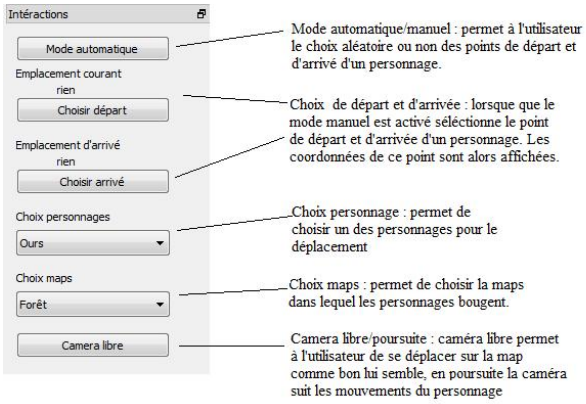

FIG. 8 – Fonctionnalité pour l'utilisateur

6.3 Implémentation de la classe QGLViewer

La classe QGLViewer ayant de nombreuses fonctionnalités déjà disponibles, les méthodes utilisées sont des redéfinitions des méthodes suivantes :

- animate ;
- draw ;
- drawWithNames ;
- postSelection ;
- init.

init()

Initialise tout ce qui doit l'être : les différents attributs, le paramétrage d'OPENGL , le chargement des textures, des personnages, ...

draw() et drawWithNames()

La méthode *draw()* dessine tout les éléments de la map ainsi que les personnages. La méthode *drawWithNames()* permet de donner un nom aux éléments présents sur la map. Cela permet par conséquent une différenciation plus facile des endroits où un personnage peut ou pas se déplacer.

postSelection()

Cette méthode permet de traiter la sélection sur la map (le point est donné en paramètre) :

- si oui ou non l'endroit choisi par l'utilisateur est bon
- si le point sélectionné est le point de départ ou d'arrivée.

animate()

Méthode permettant l'animation des personnages. Elle se relance selon une certaine périodicité. C'est ici que les nouvelles coordonnées des personnages sont mise à jour.

6.4 Implémentation de la fenêtre principale

Les composant d'interface graphique sont des encapsulations de widgets. Il n'est donc pas étonnant que la fenêtre principale soit une widget. Elle est construite de manière basique : une barre de menu, une dockwidget[11] composée d'une widget pour l'intéraction vu plus haut et de la fameuse QGLViewer.

Chaque bouton et combobox de la widget d'intéraction est relié à la QGLViewer par des "slots" et "signals"[12] gérant la gestion d'évènements sous Qt.

[11] Dockwidget : widget décomposée en plusieurs partie

[12] Ex : QObject : :connect(chooseStart, SIGNAL(clicked()), this, SLOT(selectStart()));

7 Les tests

Plusieurs séries de tests ont été réalisées sur l'algorithmique de notre projet (sur les algorithmes A* et SMA*). Nous avons volontairement évité de faire des tests sur la partie graphique, qui n'auraient eu que peu de sens, car trop dépendant du *hardware*[13]. Le code utilisé est donné par la figure 7. Les tests consistent donc à faire parcourir une carte donnée à notre personnage en diagonale et à mesurer le temps que l'algorithme met à trouver une solution.

```
double temps;
clock_t start;
start = clock();
SMAstar a(*(g.graphe[1][1]), *(g.graphe[99][99]), g);
list <Parent> l = a.getCheminFinal();
temps = (double)(clock()-start)/(double)CLOCKS_PER_SEC;
cout<<"temps execution : "<<temps<<endl;
a.~SMAstar();
```

FIG. 9 – Code inséré pour les tests. Ici : Algo SMA* / taille 100*100

7.1 Première phase

Dans la première série de tests 7.1, la variable choisie est la taille du graphe, et on compare les performances obtenues pour A* , puis pour SMA* avec la taille de la mémoire fixée à 20. On remarque très bien que SMA* est bien plus efficace que A* en terme de temps pour une carte de grande taille, mais que la différence est minime pour un petit graphe.

Taille du graphe	Temps d'execution pour A* (s)	Temps d'execution pour SMA* (s)
20*20	0.004	0.004
50*50	0.022	0.010
100*100	0.083	0.029
200*200	0.296	0.091
400*400	1.172	0.392

FIG. 10 – Temps d'exécution des algorithmes en fonction de la taille du graphe

Aux vues de ces résultats, nous avons décidé d'incorporer l'algorithme SMA* à l'application, car nos cartes sont relativement de grande taille (basé sur un graphe de taille de l'ordre de 100*100). Le choix n'est pas donné à l'utilisateur, mais les codes sources sont fournis, il est donc possible pour un utilisateur qualifié, s'il le souhaite, de modifier tous les appels à SMA* en appels à A* , les fonctions étant identiques, seule la classe appelée diffère.

7.2 Seconde phase

Les seconds tests ont eu pour but de qualibrer la mémoire allouée à SMA* .

Nous avons donc bloqué la taille du graphe à 100*100 (taille d'une carte standard) et fait varier la mémoire allouée. La figure 7.2 consigne les résultats obtenus. On y observe que plus la mémoire allouée est importante, plus le temps processeur est grand, et à partir d'une mémoire de 500 noeuds, SMA* et A* sont équivalents en terme de temps. Ceci pourrait s'expliquer par le fait que A* garde surement en mémoire, pour un graphe de taille 100*100,

[13] Particulièrement la carte graphique

moins de 500 noeuds. On a estimé que l'algorithme SMA* , sur un graphe de taille 100*100 se relance en moyenne 2 fois pour une mémoire de 20 noeuds, ce qui reste acceptable.

Mémoire allouée (noeuds)	Temps d'execution pour SMA* (s)
20	0.029
50	0.037
100	0.048
200	0.079
500 et plus	0.086

FIG. 11 – Temps d'exécution de SMA* en fonction de la mémoire allouée, graphe de taille 100*100

Aux vues des résultats, il paraît évident que plus la mémoire est petite, meilleur est le rendu en terme de temps processeur. Cependant, prendre une mémoire trop petite ferait perdre le bénéfice gagné par cet algorithme, du fait qu'il devrait se relancer plus souvent, ce qui multiplierait le temps donné par le tableau 7.2 par autant de foisq ue SMA* se relancerait.

Notons par ailleurs que plus la taille de la carte est grande, plus la mémoire allouée devra être importante, pour éviter que SMA* se relance trop souvent et fasse perdre plus de temps qu'il n'en fait gagner par rapport à A* .

Conclusion

La programmation 3D est une expérience nouvelle dans le cadre de nos études, de plus l'univers 3D étant très intéressant, le travail sur le projet s'est donc avéré aussi éducatif que ludique tant dans l'utilisation du logiciel 3D Studio Max , que le maniement d'OpenGl ou bien dans l'intégration des algorithmes de *Pathfinding* dans nos cartes.

Il est clair que ce TER était pour nous une sorte d'initiation dans l'univers 3D et dans le monde du jeu vidéo car il nous a permis de découvrir la complexité du développement d'une telle application, ainsi que la diversité des compétences requises (intelligence artificielle, infographie, moteur graphique).

Nous pouvons ajouter le fait que l'élaboration du TER nous a apporté un plus dans la coordination et la conduite d'une tâche en équipe.

Par conséquent, le TER "Mouvement collectif de personnages 3D dans les Sims" fût une expérience enrichissante sous tous les aspects, expérience dont nous sommes fier d'avoir menée à bien, même si un léger pincement de coeur persiste du fait qu'avec un peu plus de temps nous aurions pu faire plus, comme :

- l'ajout d'autres personnages ;
- la création d'autres maps plus complexes ;
- l'étude du mouvement collectif et des algorithmes de collision.

Annexes

A La longue et fastidieuse compilation sous Qt...

La librairie QT ainsi que celle de la QGLViewer ont énormément de fonctionnalités et ont un rendu final très satisfaisant. Néanmoins, la compilation d'un projet QT est un véritable parcours du combattant comme vont le démontrer les explications sur les différentes étapes de la compilation d'un projet.

A.1 Etape 1 : téléchargement et installation de QT

Vous pouvez télécharger Qt via le lien [Sitb]. Néanmoins, la version 2009.02 semble poser des problèmes (la commande make, essentielle pour la compilation, ne semble pas être reconnue par Windows. Soit le nom de la commande a changer, soit la version n'est pas stable). C'est pour cette raison que nous fourniront une version antérieur à celle-ci. Il n'y a plus qu'a lancer l'exécutable téléchargé et suivre les indications de l'installation en incluant bien MinGW et les source de l'ancienne version de qt.

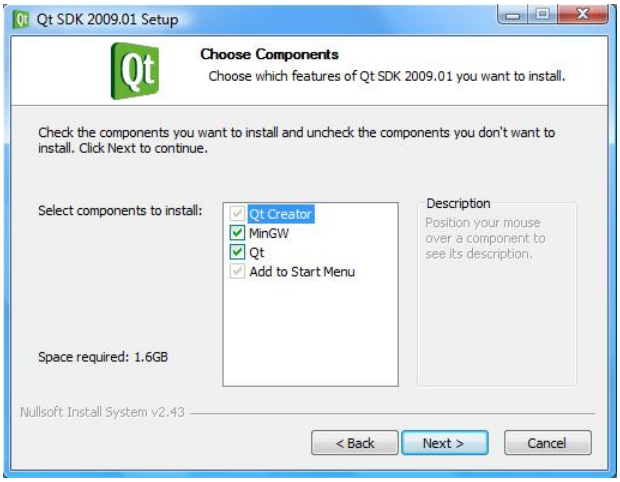

FIG. 12 – Installer Qt

A.2 Etape 2 : téléchargement et installation de la librairie qglviewer

Vous pouvez télécharger cette librairie sur le site [Tel]. Une fois l'archive décompressée, ouvrir le terminal de qt[14].

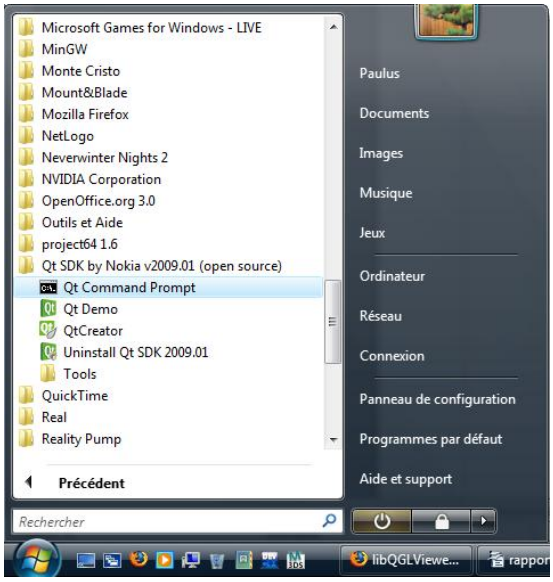

FIG. 13 – Invite de Commande Qt dans Windows

Se rendre à l'aide du terminal dans le dossier QGLViewer de l'archive décompressée précédemment. Taper la commande qmake puis make. Un fichier QGLViewer2.dll a du se créer, il faut le copier dans le dossier System32 de Windows. Déplacer ensuite le dossier libQGLViewer-2.3.1 dans le dossier qt (à l'endroit où se trouve les dossier bin, include etc.).

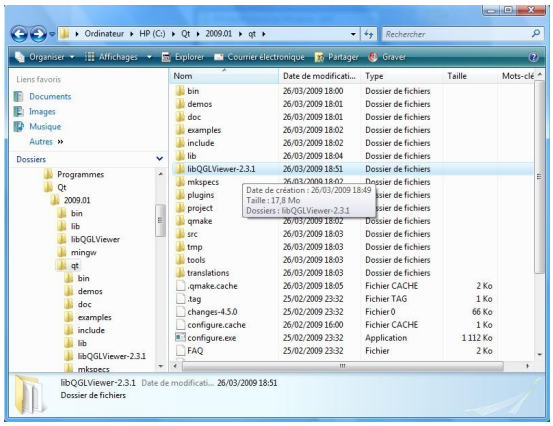

FIG. 14 – Retrouver QGLViewer dans Windows

L'explication de l'installation de cette librairie est aussi expliqué sur le site de cette dernière [Ins].

[14] *Démarrer/Tout les programmes/Qt SDK by Nokia v2009.01 (open source)* pour l'installation par défaut

A.3 Etape 3 : compilation du projet

Déplacer le dossier du projet vers le dossier de qt (figure 14). La création d'un nouveau dossier pour y déposer tout les projets qt est une solution (par exemple, le dossier project dans la figure 14). Se rendre dans le dossier du projet à l'aide du terminal de qt puis lancer la commande *qmake -project*. Un fichier .pro a été créer dans la dossier. L'ouvrir à l'aide d'un éditeur de texte (le bloc-note suffit, mais Dev-C++ permet une meilleure lecture) puis effectuer les modifications suivantes

`Remplacer la ligne`
`INCLUDEPATH += .`
`par`
`INCLUDEPATH *= . C:/Qt/2009.01/qt/libQGLViewer-2.3.1`
`puis ajouter en dessous les lignes`
`LIBS *= -L[lien de l'INCLUDEPATH]/QGLViewer -lQGLViewer2`
`QT *= opengl xml`

Bien sûr le chemin de la librairie est a changer selon l'endroit où l'installation de qt a été faite.

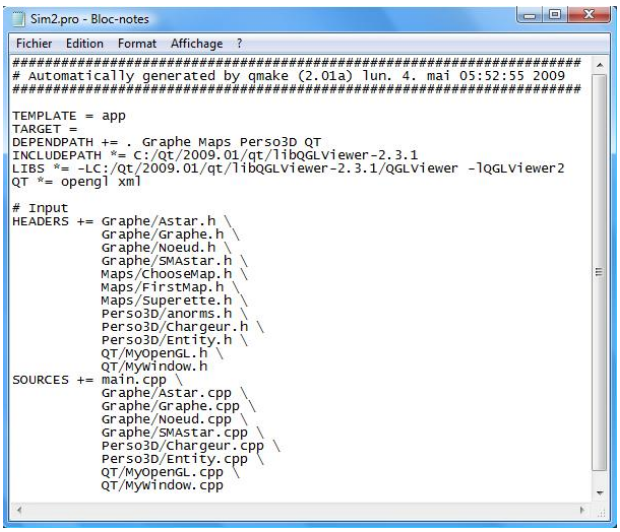

FIG. 15 – Edition du fichier .pro

Cela fait, retourner sur le terminal de qt et taper la commande qmake puis make. Se lance alors une longue phase de compilation. Une fois fini, un dossier release s'est créé. Toujours dans la console Qt, taper *cd release* puis

`./nomProjet`

pour exécuter le projet.

L'application devrait alors se lancer à la grande joie de l'utilisateur.

B Codes sources

B.1 Exemple de Header codant une carte (FirstMap.h)

```
//inclue la lib QGLViewer (par consequent la widget Qt ainsi que tout OpenGL)
#include <QGLViewer/qglviewer.h>
#include "../Graphe/Graphe.h" //la classe Graphe
Graphe g1; //on fait correspondre un graphe a la carte

//on specifie les textures utilisees (seront chargees dans la classe MyOpenGL)
GLuint solFirst;
GLuint tronc;
GLuint feuilles;

//fonction qui dessine un obstacle (ici : un arbre)
static void DessinerArbre(){
        /* cette fonction dessine un arbre et replace le repere a son etat initial */
        // tronc
        glPushMatrix();
        glPushName(2);
        glBindTexture(GL_TEXTURE_2D, tronc);
        GLUquadric* base;
        base = gluNewQuadric();
        gluQuadricTexture(base,GL_TRUE);
        gluCylinder(base,0.2,0.02,1.5,20,1);
        glPopName();
        glPopMatrix();

        // feuillage
        glPushMatrix();
        glPushName(2);
        glTranslated(0,0,1.5);
        glBindTexture(GL_TEXTURE_2D, feuilles);
        GLUquadric* arb;
        arb = gluNewQuadric();
        gluQuadricTexture(arb,GL_TRUE);
        gluSphere(arb,0.45,20,20);
        glPopName();
        glPopMatrix();
}

//fonction qui dessine le sol
static void drawFirstMapFloor(){
        for(int i=0;i<20;i++){
                for(int j=0;j<20;j++){
                        glPushMatrix();
                        glPushName(1);
                        glTranslatef(i, j, 0);
                        glColor3f(255, 255, 255);
                        glBindTexture(GL_TEXTURE_2D, solFirst);
                        glBegin(GL_QUADS);
                                glTexCoord2d (0, 0);     glVertex3d(0,0,0);
                                glTexCoord2d (1, 0);     glVertex3d(1,0,0);
```

```
                                glTexCoord2d (1, 1);     glVertex3d(1,1,0);
                                glTexCoord2d (0, 1);     glVertex3d(0,1,0);
                        glEnd();
                        glPopName();
                        glPopMatrix();
                }
        }
}

//fonction qui dessine les obstacles
static void drawFisrtMapObstacle(){
        for(int i=0;i<=20;i++){
                for(int j=0;j<=20;j++){
                        if(g1.graphe[i][j]->getBool()){
                                // Il y a un arbre ici
                                glTranslated(g1.graphe[i][j]->getX(),g1.graphe[i][j]
                                        ->getY(),0);
                                DessinerArbre();
                                glTranslated(-g1.graphe[i][j]->getX(),-g1.graphe[i][j]
                                        ->getY(),0);
                                glBindTexture(GL_TEXTURE_2D, solFirst);
                        }
                }
        }
}
```

B.2 Code de la bibliothèque de cartes : ChooseMap.h

```
#include <QGLViewer/qglviewer.h>
//on inclue toutes les carte creees
#include "Superette.h"
#include "FirstMap.h"

//analyse de cas selon un entier qui correspond a une des cartes
static void drawFloor(int x){
        if (x == 0)
                drawSuperetteFloor();
        if (x == 1)
                drawFirstMapFloor();
}

static void drawObstacle(int x){
        if (x == 0)
                drawSuperetteObstacle();
        if (x == 1)
                drawFisrtMapObstacle();
}
```

B.3 Fonction codant l'éclairage

```
MyOpenGL::lumieres( ){
        glEnabl e (GL_LIGHTING) ;
        glLightfv(GL_LIGHT0,GL_AMBIENT, ambient ) ;
        glLightfv(GL_LIGHT0,GL_DIFFUSE, diffuse);
        glLightfv(GL_LIGHT0,GL_POSITION, position);
        glLightfv(GL_LIGHT0,GL_SPECULAR, specular;
        glLightModelfv(GL_LIGHT_MODEL_AMBIENT, model_ambient);
        glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, 0 );
        glEnable (GL_LIGHT0);
}
```

B.4 Chargeur : chargement d'un fichier .MD2

```
bool Chargeur::LoadModel(string szFilename){
        fstream file;// fichier
        file.open( szFilename.c_str(), ios::in | ios::binary );

        if( file.fail() ){
                cout<<"a ce niveau la "<<szFilename<<endl;
                return false;
        }
        // lecture du header
        file.read( (char *)&m_kHeader, sizeof( md2_header_t ) );

        // verification de l'authenticite du modele
        if( (m_kHeader.version != MD2_VERSION) || m_kHeader.ident != MD2_IDENT )
                return false;

        // allocation de memoire pour les donnees du modele
        m_pSkins = new md2_skin_t[ m_kHeader.num_skins ];
        m_pTexCoords = new md2_texCoord_t[ m_kHeader.num_st ];
        m_pTriangles = new md2_triangle_t[ m_kHeader.num_tris ];
        m_pFrames = new md2_frame_t[ m_kHeader.num_frames ];
        m_pGLcmds = new int[ m_kHeader.num_glcmds ];

        // lecture des noms de skins
        file.seekg( m_kHeader.offset_skins, ios::beg );
        file.read( (char *)m_pSkins, sizeof( char ) * 68 * m_kHeader.num_skins );

        // lecture des coordonnees de texture
        file.seekg( m_kHeader.offset_st, ios::beg );
        file.read( (char *)m_pTexCoords, sizeof( md2_texCoord_t ) * m_kHeader.num_st );

        // lecture des triangles
        file.seekg( m_kHeader.offset_tris, ios::beg );
        file.read( (char *)m_pTriangles, sizeof( md2_triangle_t ) * m_kHeader.num_tris );

        // lecture des frames
        file.seekg( m_kHeader.offset_frames, std::ios::beg );

        for( int i = 0; i < m_kHeader.num_frames; i++ ){
                // allocation de memoire pour les vertices
                m_pFrames[i].verts = new md2_vertex_t[ m_kHeader.num_vertices ];

                // lecture des donnees de la frame
                file.read( (char *)&m_pFrames[i].scale, sizeof( vec3_t ) );
                file.read( (char *)&m_pFrames[i].translate, sizeof( vec3_t ) );
                file.read( (char *)&m_pFrames[i].name, sizeof( char ) * 16 );
                file.read( (char *)m_pFrames[i].verts, sizeof( md2_vertex_t )
                        * m_kHeader.num_vertices );
        }
        // lecture des commandes OpenGL
        file.seekg( m_kHeader.offset_glcmds, std::ios::beg );
        file.read( (char *)m_pGLcmds, sizeof( int ) * m_kHeader.num_glcmds );
```

```
        // fermeture du fichier
        file.close();

        return true;
}
```

B.5 Chargeur : transformation .MD2 vers OpenGL

```
void Chargeur::RenderFrame( int iFrame ){
        // calcul de l'index maximum d'une frame du modele
        int iMaxFrame = m_kHeader.num_frames - 1;

        // verification de la validite de iFrame
        if( (iFrame < 0) || (iFrame > iMaxFrame) )
                return;

        // activation de la texture du modele
        glBindTexture( GL_TEXTURE_2D, m_uiTexID );

        // dessin du modele
        glBegin( GL_TRIANGLES );
        // dessine chaque triangle
                for( int i = 0; i < m_kHeader.num_tris; i++ ){
                // designe chaque vertex du triangle
                        for( int k = 0; k < 3; k++ ){
                                md2_frame_t *pFrame = &m_pFrames[ iFrame ];
                                md2_vertex_t *pVert = &pFrame->verts[
                                        m_pTriangles[i].vertex[k] ];
                                // [coordonnees de texture]
                                GLfloat s = (GLfloat)m_pTexCoords[ m_pTriangles[i].st[k]
                                        ].s / m_kHeader.skinwidth;
                                GLfloat t = (GLfloat)m_pTexCoords[ m_pTriangles[i].st[k]
                                        ].t / m_kHeader.skinheight;
                                // application des coordonnees de texture
                                glTexCoord2f( s, t );

                                // [normale]
                                glNormal3fv( m_kAnorms[ pVert->normalIndex ] );

                                // [vertex]
                                vec3_t v;

                        // calcul de la position absolue du vertex et redimensionnement
                                v[0] = (pFrame->scale[0] * pVert->v[0] + pFrame->
                                        translate[0]) * m_fScale;
                                v[1] = (pFrame->scale[1] * pVert->v[1] + pFrame->
                                        translate[1]) * m_fScale;
                                v[2] = (pFrame->scale[2] * pVert->v[2] + pFrame->
                                        translate[2]) * m_fScale;
                                glVertex3fv( v );
                        }
                }
        glEnd();
}
```

B.6 Animation du personnage

```
void Entity::Animate( int iStartFrame, int iEndFrame, float fPercent ){
        // m_iCurrFrame doit etre compris entre iStartFrame et iEndFrame
        if( m_iCurrFrame < iStartFrame )
                m_iCurrFrame = iStartFrame;

        if( m_iCurrFrame > iEndFrame )
                m_iCurrFrame = iStartFrame;

        m_fPercent = fPercent;

        // animation : calcul des frames courante et suivante
        if( m_fInterp >= 1.0 ){
                m_fInterp = 0.0f;
                m_iCurrFrame++;

                if( m_iCurrFrame >= iEndFrame )
                        m_iCurrFrame = iStartFrame;

                m_iNextFrame = m_iCurrFrame + 1;

                if( m_iNextFrame >= iEndFrame )
                        m_iNextFrame = iStartFrame;
        }
}
```

Références

[Alg] Algorithmes de recherche dans les systèmes à agents. http ://turing.cs.pub.ro/auf2/html/chapters/chapter3/sommaire.html.

[Ins] Installation QGLViewer. http ://www.libqglviewer.com/installWindows.html.

[Mas99] Mason Woo, Jackie Neider, Tom Davis Dave Shreiner. *OpenGL 2.0 Guide Officiel*. CampusPress, 1999.

[Mic08] Michèle Bousquet. *L'art du bluff avec 3ds Max*. Pearson Education France, 2008.

[Rec] Recherche de chemin par l'algorithme A*. http ://khayyam.developpez.com/articles/algo/astar/.

[Rus95] Russell Norving. *Artificial Intelligence A Modern Approach*. Pearson Education, Inc, 1995.

[Sch09] Schwab B. *AI Game Engine Programming*. COURSE Technology, CENGAGE Learning, 2009.

[Sita] Site officiel de QGLViewer. http ://www.libqglviewer.com/.

[Sitb] Site officiel de QT. http ://www.qtsoftware.com/downloads.

[Tel] Telechargement QGLViewer. http ://www.libqglviewer.com/src/libQGLViewer-2.3.1.zip.

[Tut] Tutorial OpenGL. http ://www.siteduzero.com/tutoriel-3-5014-creez-des-programmes-en-3d-avec-opengl.html.