

Navigation dans un environnement 3D

Marc BEYSECKER, Tom GIMENEZ, Valentin HIRSON, Léo RIZZON

Professeur encadrant : Mr Frédéric Koriche

Université Montpellier II

Master 1 Informatique

Table des matières

1	Introduction	3
2	Recherches préliminaires	4
2.1	Une scène en 3D avec Blender	4
2.1.1	Les différents formats de fichier	4
2.2	Récupérer notre scène en 3D	4
2.2.1	OpenSceneGraph	4
2.2.2	OpenGL avec nos propres structures de données	4
3	Implémentation finale	6
3.1	Un parser pour récupérer notre scène	6
3.2	Nos structures de données	6
3.2.1	La scène	6
3.2.2	Le graphe généré	6
3.3	Travaux effectués sur le graphe	6
3.3.1	Création du graphe des way-points	6
3.3.2	Simplification du graphe obtenu : merging	7
4	Bibliographie	8
5	Conclusion	9

Remerciements

Ma mère, ta mère. Merci mon cul bonsoir messieurs dames.

Chapitre 1

Introduction

Contexte

Dans les jeux vidéos, les personnages non joués par des humains doivent pouvoir se déplacer de manière autonome et cohérente. Un environnement 3D est constitué d'un graphe énorme avec des milliers de sommets. Même en ne prenant que le sol, le graphe est encore très gros et, surtout, n'est pas seulement constitué des points naviguables. C'est à dire que le personnage ne doit pas pouvoir se déplacer dessus.

Comme nous l'avons étudié, la recherche de chemins dans un graphe est un problème classique mais qui peut s'avérer très lourd sur de gros graphes. En prenant en compte que les jeux mettent en scène un grand nombre d'agents il s'agit de minimiser les temps de calculs.

Il s'agit donc d'une part de ne sélectionner que les points naviguables et de simplifier le graphe obtenu pour limiter les espaces de calculs. De ces opérations naît le graphe de way-points. Il s'agit donc du graphe sur lequel vont se déplacer les agents.

Dans la plupart des jeux actuels les environnements sont créés "à la main" par les créateurs du jeu et les game designers placent eux même les points du graphe des way-points. Cela représente un gros travail et c'est même impossible dans le cas d'environnements aléatoirement générés.

Dans le cadre de notre unité d'enseignement intitulée Algorithmes de l'Intelligence Artificielle, nous avons réalisé un projet qui consiste en l'implémentation d'un algorithme de génération automatique du graphe des way-points.

Objectifs

La base de notre travail est une scène 3D créée à l'aide d'un logiciel d'éditions d'objets 3D, par exemple Blender. Il s'agit alors de charger cette scène pour y appliquer nos traitements. Tout d'abord, il faut générer le graphe à partir de tous les points composant les différentes formes pour représenter les arêtes. Ensuite il faut épurer ce graphe pour ne garder que les sommets et les arêtes "emruntables". Enfin on va chercher à appliquer un algorithme de simplification pour ne garder que les points réellement utiles. C'est une procédure appelée merging.

Le résultat serait donc un graphe des way-points, simplifié, automatiquement généré.

Chapitre 2

Recherches préliminaires

2.1 Une scène en 3D avec Blender

2.1.1 Les différents formats de fichier

COLLADA Collaborative Design Activity (abrégé en COLLADA, signifiant activité de conception collaborative) a pour but d'établir un format de fichier d'échange pour les applications 3D interactives.

COLLADA définit un standard de schéma XML ouvert pour échanger les acquisitions numériques entre différents types d'applications logicielles graphiques qui pourraient autrement conserver leur acquisition dans des formats incompatibles. Les documents COLLADA, qui décrivent des acquisitions numériques, sont des fichiers XML, habituellement identifiés par leur extension .dae («digital asset exchange», traduit par «échange numérique d'acquisition»). C'est un format supportant la vaste majorité des fonctionnalités modernes requises par les développeurs de jeux vidéo.

Il fallait donc faire un choix, COLLADA étant plus conséquent que .obj, et le sujet de notre TER n'étant pas porté sur le format de notre scène, nous avons donc choisi de faire simple et de choisir .obj. De plus le format XML nous obligeait à utiliser des bibliothèques dédiées, complexes et non standardisées, qui nous aurait pris beaucoup plus de temps pour écrire le parser.

.obj Notre choix A quoi ça ressemble, pourquoi on a choisit ça

2.2 Récupérer notre scène en 3D

Une fois notre scène créée sous Blender s'est posé le problème de l'exploiter. Nous avons alors cherché du côté des bibliothèques existantes, en C/C++ surtout. En effet, Blender nous proposait divers format de fichier de sortie. Nous devons faire en sorte que les informations soient récupérables et exploitable pour nos algorithmes. L'idée générale est d'utiliser un parser qui va lire les données du fichier pour les transformer en données compréhensibles par notre programme.

Voici les solutions que nous avons envisagées.

2.2.1 OpenSceneGraph

2.2.2 OpenGL avec nos propres structures de données

Une fois l'expérience OpenSceneGraph terminée et mise de côté, nous avons dû de nouveau chercher une solution. Après de longues recherches, nous avons trouvé quelques possibilités comme OGRE (Object-Oriented Graphics Rendering Engine), nous avons finalement décidé de créer nous même ce dont nous avions besoin. C'est la solution finale.

L'utilisation d'OpenGL, conseillée par notre professeur encadrant, était un choix établi. Libre et accessible, OpenGL est de plus au programme de l'année prochaine : il est toujours intéressant de prendre un peu d'avance. OpenGL nous sert à afficher notre scène 3D.

Coder en C++, pour la performance du programme, pour l'aspect objet intéressant dans notre contexte, pour l'utilisation d'OpenGL, était presque évident.

Finalement nous avons donc implémenté notre propre parser pour remplir nos propres structures de données. En fait, on a recodé une partie des fonctionnalités des bibliothèques que nous avons étudiées pour n'en garder que ce dont nous avons besoin et pour avoir des structures que nous maîtrisons pleinement. Au final, cela nous a fait gagné du temps.

Chapitre 3

Implémentation finale

3.1 Un parser pour récupérer notre scène

3.2 Nos structures de données

- Forme :

Une forme est une figure géométrique qui contient une liste de vertex et une liste de faces. À partir d'une forme, nous pouvons générer un graphe qui lui correspond.

- Vertex :

Un vertex est un point de notre scène. Il possède 3 coordonnées x, y et z. Lorsque qu'un graphe est généré, tous les vertex d'une forme voient leur liste de voisins mise à jour. Grâce à cette liste de voisins (ou d'adjacences), les parcours dans le graphe sont possibles (cette liste représente les arêtes du graphe). Nous avons choisi les listes de voisins car chaque sommet possède au plus 4 voisins si le maillage est fait de carrés ou 3 voisins s'il est fait de triangles. Ce qui permet un coût de stockage très faible et un temps de calcul bien maîtrisé. Prenons l'exemple de la suppression d'un voisin : Il suffit de l'isolé pour qu'il ne soit plus parcouru ; soit le supprimer de la liste de voisins de tous ses voisins. Au pire des cas : 4 x 4 tests.

- Faces :

Les faces possèdent une liste contenant les numéros des vertex qui les composent. C'est grâce aux faces que nous pouvons déduire les voisins des sommets du graphe. Elles servent aussi à effectuer un affichage cohérent des formes.

3.2.1 La scène

3.2.2 Le graphe généré

3.3 Travaux effectués sur le graphe

3.3.1 Création du graphe des way-points

Parcours

Heuristique

3.3.2 Simplification du graphe obtenu : merging

Chapitre 4

Bibliographie

Chapitre 5

Conclusion

Bilan technique

Bilan personnel