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

Remerciements

Ma mère, ta mère. Merci mon cul bonsoir m'ssieurs 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 nait 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 Le maillage complet

Blender est un logiciel d'édition d'objets 3D libre sous licence GPL. Bien que nous ayons eu la possibilité d'utiliser des logiciels payant très évolués, nous avons fait le choix, approuvé par notre professeur encadrant, d'utiliser Blender. Blender est multi-platerformes et c'est cet aspect qui nous a décidé car nous travaillons tous sous Linux. Bien que gratuit, il offre largement toutes les fonctionnalités dont avions besoin. Pour fabriquer notre sol nous avons longtemps cherché un modèle pré-conçu mais cela s'avéra infructueux. Nous avons alors découvert qu'il était possible de diviser une forme de base. Nous avons décider de prendre un simple carré, de l'agrandir à la taille souhaitée et de le diviser en autant de petites cases que nécessaires. Cela nous a donc donné notre maillage sur lequel poser nos formes. Nous avons posé des pavés sur notre sol car se sont les formes que prennent les bounding box. Une bounding box est la forme simplifiée que prend un modèle 3D ; cela permet de simplifier les calculs d'intersection entre la forme et son environnement.

2.1.2 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 librairies

dédiées, complexes et non standardisées, qui nous aurait pris beaucoup plus de temps pour écrire le parser.

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

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 librairies existantes, en C/C++ surtout. En effet, Blender nous proposait divers format de fichier de sortie. Nous devions 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 librairies que nous avions étudiées pour n'en garder que ce dont nous avons besoin et pour avoir des structures que nous maitrisons pleinement. Au final, cela nous a fait gagné du temps.

Chapitre 3

Implémentation finale

Nous présenterons ici les éléments que nous avons effectivement implémentés.

3.1 Principe général

Voici le principe général de notre programme. Ci-après les différentes étapes successives en considérant que l'on part d'une scène 3D en format WaveFront.

- Parser le fichier .obj de la scène 3D
- Isoler le sol des autres formes
- Générer le graphe correspondant au sol
- Détecter les sommets invalides et les retirer du graphe
- Simplifier le graphe (merging)

3.2 Un parser pour récupérer notre scène

Nous récupérons donc un fichier au format WaveFront. Il s'agit de remplir nos structures de données, un vecteur de forme que nous appelerons ici *listeDeFormes*. Nous avons présenté précédemment la structure d'un fichier au format WaveFront. Ici nous présenterons le fonctionnement de notre parser.

3.3 Nos structures de données

Nous présenterons ici nos structures de données, utilisées dans notre projet. Comme dit précédemment, nous avons exploité le potentiel objet du C++.

- 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

Algorithm 1 Parser de fichier .obj

Require: fichier .obj

1: **for** chaque forme (c'est à dire une apparition de "v" après une série de "f") **do**
2: **if** la ligne commence par un "v" **then**
3: enregister "v x y z" comme nouveau vertex de coordonnées (x,y,z) dans le vecteur de vertex de la forme courante
4: **else**
5: **if** la ligne commence par un "f" **then**
6: enregistrer "f x_1 ... x_n" comme nouvelle face dans le vecteur de faces de la forme courante
7: **end if**
8: **end if**
9: **end for**

Algorithm 2 Parcours du graphe complet

1: //Algorithme lancé à partir d'un sommet non isolé
2: **if** sommet courant non marqué **then**
3: marquer le sommet courant
4: **for** chaque voisin v **do**
5: parcourir(v)
6: **end for**
7: **end if**

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 maitrisé. 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.
- BoundingBox :

3.3.1 La scène

3.3.2 Le graphe géneré

3.4 Travaux effectués sur le graphe

3.4.1 Création du graphe des way-points

Parcours

Heuristique

Algorithm 3 Merging

```
1: //Algorithme lancé à partir d'un sommet non isolé
2: if sommet courant non marqué then
3:    dupliquer la liste des voisins
4:    if sommet courant a 4 voisins then
5:       if sommets voisins du sommet courant ont tous au moins 3 voisins then
6:          isolation du sommet courant
7:       end if
8:    end if
9:    for chaque voisin v sauvegardé do
10:      merging(v)
11:   end for
12: end if
```

Merging (simplification du graphe)

3.4.2 Simplification du graphe obtenu : merging

Chapitre 4

Bibliographie

Chapitre 5

Conclusion

Bilan technique

Bilan personnel