

# 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

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Recherches préliminaires</b>	<b>4</b>
2.1	Une scène en 3D avec Blender . . . . .	4
2.1.1	Le maillage . . . . .	4
2.1.2	Les différents formats de fichier . . . . .	5
2.2	Récupérer notre scène en 3D . . . . .	6
2.2.1	OpenSceneGraph . . . . .	6
2.2.2	OpenGL avec nos propres structures de données . . . . .	6
<b>3</b>	<b>Implémentation finale</b>	<b>7</b>
3.1	Principe général . . . . .	7
3.2	Un parser pour récupérer notre scène . . . . .	7
3.3	Nos structures de données . . . . .	7
3.3.1	La scène . . . . .	8
3.3.2	Le graphe généré . . . . .	8
3.4	Travaux effectués sur le graphe . . . . .	8
3.4.1	Création du graphe des way-points . . . . .	8
3.4.2	Simplification du graphe des way-points : merging . . . . .	8
<b>4</b>	<b>Bibliographie</b>	<b>10</b>
<b>5</b>	<b>Conclusion</b>	<b>11</b>

# 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 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

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.

#### 2.1.1 Le maillage

Pour simuler le travail sur un terrain assimilable à un environnement 3D de jeux vidéo, il nous fallait un graphe avec beaucoup de points. L'objectif était de passer

INSERER IMAGES CARRE SIMPLE PUIS MAILLAGE

Créer une scène simple sous Blender a été plutôt facile. Pourtant nous avons eu plus de mal à obtenir un maillage. Nous avons alors exploré plusieurs possibilités.

**Une map Half-Life** Au départ, nous avions dans l'idée de partir d'une map Half-Life. Dans un premier temps nous générions la map avec un générateur de terrain : Gensurf. Ensuite nous l'importions et l'éditions avec l'éditeur de map officiel WorldCraft. Nous arrivions donc à générer une scène rapidement et plus facilement qu'avec Blender. Malheureusement l'éditeur de map WorldCraft ne permet pas d'exporter directement en .obj , après quelques recherches, nous avons contourné le problème en utilisant Object Viewer qui permet de transformer un .map en .obj.

Au final, nous n'avons pas retenue cette solution car le .obj obtenu n'était pas aussi "propre" que celui obtenu par Blender. De plus la scène avait une épaisseur, ce qui ne nous plaisait pas.

**La solution avec Blender** 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 décidé 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

Une fois la scène créée sous Blender le logiciel nous proposait différents format d'exportation. Il y en a un grand nombre et nous en avons étudié principalement deux.

**COLLADA** Collaborative Design Activity (abrégié 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.

**WaveFront** OBJ est un format de fichier contenant la description d'une géométrie 3D. Il a été défini par la société Wavefront Technologies. Ce format de fichier est ouvert et a été adopté par d'autres logiciels 3D (tels que Maya, 3D Studio Max, Lightwave et bien sur Blender) pour des traitements d'import / export de données.

Dans un fichier Wavefront (extension .obj), les formes sont stockées les unes après les autres. Les différentes entités sont écrites par bloc. Donc, pour chaque forme les informations qui la concerne sont stockées ligne par ligne. Nous savons à quoi chaque information correspond grâce à la lettre clef qui débute la ligne ( par exemple : "v" pour un vertex ). Les seuls informations qui nous intéressent sont les vertex et les faces. Une ligne pour un vertex fournit la position de ce vertex (x, y, z) écrit comme suit "v x y z" par exemple "v 2.0 2.0 3.0" pour un vertex de coordonnées (2.0, 2.0, 3.0) Une ligne pour une face fournit le numéro des vertex qui la composent écrit comme suit "f num<sub>1</sub> ... num<sub>n</sub>" par exemple "f 4 6 7 8 11 13" pour une face constituée des vetices numéros 4, 6, 7, 8 et 11. Le numéro d'un vertex n'est pas explicitement donné mais il se déduit par son ordre d'apparition dans le fichier Wavefront. C'est un identifiant unique.

Voici un exemple de code dans un fichier .obj.

Ici il s'agit d'un simple carré. Une forme commence par la liste des points qui la constitue puis la liste de ses faces.

```
v 0 0 0 //premier point de la nouvelle forme
```

```

v 0 0 1
v 1 0 1
v 1 0 0
f 1 2 3 4 //première face constituée des points 1 2 3 4
v ... //un v après un f on est donc sur une nouvelle forme

```

Nous avons donc choisi ce format de fichier pour stocker notre scène 3D car il nous a été très facile de développer le parser adéquat. Les informations fournies sont justes celles dont nous avons besoin. On peut noter toutefois qu'il est possible d'exporter également des données sur les textures et les normales mais nous n'utilisons pas ces fonctionnalités.

## 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 formats de fichier de sortie. Nous devions faire en sorte que les informations soient récupérables et exploitables 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 avons 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

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 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 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'isoler 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 :



---

**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:     enregistrer "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  $x_1 \dots x_n$ " comme nouvelle face dans le vecteur de faces de la
       forme courante
7:     end if
8:   end if
9: end for
```

---

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 :

6

### 3.2.1 La scène

Comment est enregistrée la scène

### 3.2.2 Le graphe généré

Comment est enregistré le graphe

## 3.3 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 appellerons ici *listeDeFormes*. Nous avons présenté précédemment la structure d'un fichier au format WaveFront. Ici nous présenterons le fonctionnement de notre parser.

## 3.4 Travaux effectués sur le graphe

Notre scène est donc récupérée dans notre programme. Nous avons de plus généré le graphe du sol. Il nous faut maintenant y appliquer nos traitements.

### 3.4.1 Création du graphe des way-points

Dans un premier temps, à partir de tous les points du graphe, nous allons chercher à supprimer tous les point sur lesquels un agent ne doit pas pouvoir se déplacer.

---

**Algorithm 2** Parcours du graphe complet : parcourir(sommet)

---

Algorithme lancé à partir d'un sommet non isolé

```
1: if sommet courant non marqué then  
2:   marquer le sommet courant  
3:   for chaque voisin v do  
4:     parcourir(v)  
5:   end for  
6: end if
```

---

---

**Algorithm 3** Merging du graphe : merging(sommet)

---

Algorithme lancé à partir d'un sommet non isolé

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

---

Parcours

Heuristique

### 3.4.2 Simplification du graphe des way-points : merging

# Chapitre 4

## Bibliographie

OpenSceneGraph

OpenGL

Le langage C/C++

Blender

WaveFront (.obj)

SDL

Ogre

Map to obj

Bounding Box

# Chapitre 5

## Conclusion

Bilan technique

Bilan personnel