UNIVERSITÉ DU QUÉBEC À MONTRÉAL

CONCEPTION ET IMPLÉMENTATION D'UN ALGORITHME DE PLANIFICATION DE CHEMIN DANS UN JEU VIDÉO COMPORTANT UN ENVIRONNEMENT TRIANGULARISÉ

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MARC SHAKOUR

SEPTEMBRE 2012

REMERCIEMENTS

Je tiens à remercier Monsieur Roger NKAMBOU, professeur à l'UQAM (Université du Québec à Montréal), ainsi que Monsieur Jörg KIENZLE, professeur à l'université McGill, pour leur co-supervision respective lors de ma maitrise en informatique au Québec, Canada.

Ils ont su tous les deux m'accorder suffisamment de leur temps afin de répondre à mes très nombreuses questions. Ils m'ont également guidé et donné suffisamment d'idées afin de mener à bien ce mémoire. Je tiens à les remercier aussi pour leur infinie patience que j'ai mise souvent à l'épreuve.

Enfin je tiens également à remercier toute l'équipe d'étudiants ayant travaillé sur le projet Mammoth qui m'a très aimablement accueilli au sein de leur laboratoire à McGill et tout particulièrement Alexandre DENAULT, Christopher DRAGERT, Teodora DAN, Etienne PEROT ainsi que Joachim DESPLAND pour leur dévouement et leur aide précieuse sur le projet.

Ce mémoire aura été une longue et fastidieuse tâche. En effet, d'abord mes connaissances concernant la planification de chemin étaient proches de zéro lorsque j'ai abordé ce sujet et ensuite travailler dans un projet où des dizaines de personnes ont déjà travaillé n'est pas facile, tant chacun à sa manière de coder.

J'ai travaillé durant ma maitrise en informatique entre l'UQAM et l'université McGill donc, situées heureusement à proximité l'une de l'autre en plein centre-ville de Montréal.

TABLE DES MATIÈRES

LISTE DES FIGURES

Figure Page

LISTE DES TABLEAUX

Tableau Page

LISTE DES ALGORITHMES

Algorithme Page

GLOSSAIRE

Apex. Dans ce contexte, sommet correspondant à l'intersection entre le chemin et le funnel.

Canal. Série de triangles sélectionnée dans l'environnement triangularisé afin de créer un chemin allant du point de départ au point d'arrivée. Cette série de triangles forment un polygone composé de portails. Terme anglais : channel

Configuration. Position, état et direction du personnage se déplaçant.

Deque. Double-ended queue, file ou pile à partir de laquelle on peut accéder aux éléments soit par le premier d'entre eux soit par le dernier. En langage Java, cela peut être transcrit par une LinkedList ou une ArrayDeque.

Distance euclidienne. est la distance à vol d'oiseau qui existe entre 2 points dans un plan.

Distance de Manhattan. est la distance qu'il faut pour parcourir un plan d'un point à un autre lorsque le personnage peut se déplacer seulement dans 4 directions.

Environnement ouvert. Environnement comportant de nombreuses zones sans obstacles.

Environnement triangularisé. Environnement possédant une triangulation à un niveau abstrait

FPS. Acronyme pour First Person Shooter. Jeu de tir à la première personne.

Funnel. Traduit littéralement en français par entonnoir ou encore cheminée, l'algorithme est appelé de cette manière à cause de la forme que fait la recherche avec les 2 segments. En effet, l'algorithme recherche dans la série de triangles à l'aide de 2 segments dont la proximité s'agrandit au fur et à mesure. Ils sont représentés par les couleurs orange et bleu sur la Figure 1.14.

MMOG. Acronyme de « Massively Multiplayer Online Game », traduit par jeu massivement multi-joueur jouable en ligne en langue française.

Mesh de navigation. Terme anglais correspondant aux zones ou un personnage peut se déplacer sur une carte de jeu. C'est un type de structure abstraite comme les quadrillages ou la triangulation.

Nœud. Étape, case par laquelle le personnage va passer lors de la planification de chemin. Avec A* et chacune des variantes, chaque nœud a un coût. Un nœud peut être un carré, un triangle, un hexagone… Dans un graphe, un nœud correspond à un sommet.

NPC. Non Playable Character. Personnage non jouable par le joueur.

Package. Terme anglais pour désigner « paquetage ». Regroupement de plusieurs fichiers de code. Nous préférons utiliser le terme anglais à cause de sa proéminence dans le domaine de l'informatique

Pathfinding . Terme anglais correspondant à l'expression française « planification de chemin » ou « planificateur de chemin ».

Planificateur de chemin. Programme permettant de calculer un chemin entre le départ et la destination.

Portail. Côté intérieur commun à deux triangles dans le canal.

Sérialisation. Processus informatique permettant de convertir un certain nombre de données en un fichier binaire.

Structure abstraite. Structure invisible au joueur.

STR. Stratégie temps réel. Acronyme français équivalent au terme anglais RTS (Real Time Strategy).

Tile. Terme anglais traduit par « tuile » ou « carreaux ». Dans Mammoth il désigne les cases abstraites sur lesquelles le joueur peut se déplacer. Les Tiles peuvent avoir n'importe quelle forme : rectangulaires, carrées, triangulaires ou hexagonales.

Tileset. Ensemble de tiles.

Triangularisation. Processus informatique permettant de calculer la triangulation d'une carte de jeu.

Triangulation. Position des triangles sur une carte de jeu. Structure abstraite.

Triangulation Delaunay. Type de triangulation inventé par Boris Delaunay en 1934 où chaque point d'une triangulation dans un plan ne peut se trouver à l'intérieur du cercle circonscrit d'un triangle.

Triangulation contrainte. Triangulation prenant en compte les obstacles de la carte de jeu. La triangularisation place alors les triangles en fonction des contraintes.

Vertex. Terme anglais traduit littéralement par « sommet », un vertex correspond plutôt à une intersection de deux lignes.

Waypoints. Traduit par « points de navigation ». Points sur le chemin d'un personnage qui lui permettent de changer de direction.

RÉSUMÉ

La planification de chemin est un domaine de l'intelligence artificielle permettant à un personnage, un objet, une unité de se déplacer automatiquement, dans un environnement, en évitant les obstacles et sans intervention humaine. Ce déplacement s'effectue entre une configuration de départ et une configuration d'arrivée. Lors de ce déplacement, à aucun moment le personnage ne doit se retrouver dans une configuration invalide.

Dans le contexte des jeux vidéo commerciaux actuels, cette technique est malheureusement encore peu utilisée correctement. De plus, peu de publications concernent la planification de chemin lorsqu'effectuée dans un environnement triangularisé. Lorsqu'utilisé dans un jeu vidéo, il est nécessaire que le calcul du chemin soit effectué très rapidement afin de fournir un temps de réaction presque immédiat au joueur. La solution calculée n'est alors pas forcément optimale mais propose un bon compromis si l'on souhaite mettre l'accent sur la vitesse de réponse.

Nous expliquons d'abord le principe de la planification de chemin en détail et ensuite certaines des techniques les plus connues tels que Dijkstra, A*, Breadth First Search, Depth First Search, etc. Nous détaillons les avantages et inconvénients des environnements triangularisés par rapport aux autres techniques de maillage.

Nous exposons aussi dans ce mémoire le fruit de nos recherches et expérimentations sur l'environnement de recherche Mammoth en comparant l'évolution de la planification de chemin dans ce contexte précis. Nous utilisons également les techniques existantes pour parcourir les cartes du jeu et discutons des problèmes rencontrés et des évolutions possibles.

De nos jours, lorsque la planification de chemin est effectuée dans un environnement triangularisé, on parle souvent de A* pour la sélection des triangles et ensuite de parcours par les milieux des côtés des triangles ou bien par leur centre. Or, cela est loin d'être optimal et bien que rapide, propose un déplacement nettement plus long dans ce cas. Pour répondre à cette problématique, nous utilisons en particulier trois algorithmes spécifiques : Triangulation A* et Triangulation Reduction A* afin de sélectionner les triangles par lesquels le personnage va passer et l'algorithme funnel modifié pour le déplacement à l'intérieur de ces triangles.

Pour conclure, nous comparons ces différentes techniques de planification de chemin dans ce contexte en comparant la vitesse d'exécution, le nombre de nœuds explorés et la distance parcourue notamment afin d'en déduire la technique la plus appropriée pour Mammoth.

Mots-clés : Planification de chemin, pathfinding, triangulation, algorithmes, A*, funnel, TA*, TRA*

INTRODUCTION

Ce mémoire a été écrit dans le cadre de ma maitrise en informatique à l'UQAM (Université du Québec à Montréal).

Il a été réalisé pour un projet de l'université McGill à Montréal : Mammoth.

Mammoth est un jeu vidéo servant d'environnement de recherche à ses étudiants. Il a été créé en 2005 par l'étudiant Alexandre DENAULT. Il s'agit d'un MMOG (Massively Multiplayer Online Game) c'est-à-dire un jeu jouable en ligne massivement multi-joueurs. Le jeu est programmé dans le langage Java et peut ainsi fonctionner sur les ordinateurs PC et Mac et donc sur tous les principaux systèmes d'exploitation (Windows, Mac OS X, Linux). Comme le projet existe depuis 2005, de nombreuses versions ont vu le jour grâce aux 30 étudiants ayant travaillé dessus alternativement depuis des années. C'est pourquoi les captures d'écran du jeu peuvent sembler très différentes entre elles, car elles n'ont pas toutes été effectuées avec la même version du jeu.

Plusieurs domaines de recherche sont possibles avec Mammoth :

- Intelligence artificielle : comportements et planification de chemin;
- Réseau : distribution des ressources, code réseau et distribution Serveur/Client;
- Gestion de la 3D : graphismes, transparence et caméra;
- bases de données, tolérances aux pannes, etc.

Chaque étudiant travaille sur un de ces domaines en particulier. Le jeu est divisé entre de nombreux modules, packages permettant ainsi aux étudiants de ne pas déranger le travail des autres. On utilise Subversion pour centraliser le travail de chacun. Subversion est un outil de collaboration informatique de centraliser les fichiers d'un projet commun sur un serveur et de garder toutes les versions de ce projet en mémoire. Cela permet à plusieurs personnes de travailler en même temps sur le projet tout en ayant toujours les dernières versions des fichiers.

Dans Mammoth (Figure 0.1), on incarne un avatar virtuel symbolisé par un personnage attribué par défaut. Les actions possibles par le joueur sont de se déplacer sur

la carte de jeu, récupérer des objets et les poser à terre, déplacer la caméra, et communiquer avec les autres joueurs.

Figure 0.1 Capture d'écran de Mammoth en version 3D

Des tests ont été effectués sur ce jeu et ont permis à environ 250-300 joueurs virtuels de se connecter simultanément sur la même carte.

Mammoth ne comporte pas d'objectif pour les joueurs. Certains mini-jeux ont été implémentés à un moment donné et proposaient alors des objectifs. Ce jeu vidéo se déroule dans un environnement en 3 dimensions. Certains objets et personnages restent malgré tout en 2 dimensions car les plus anciennes versions de Mammoth se déroulaient dans un environnement en 2 dimensions.

Le but de ce mémoire est d'améliorer les techniques de planification de chemin afin de proposer des techniques adaptées aux jeux vidéo en général, et à Mammoth en particulier.

Notre problématique est donc la suivante : « Comment, dans le contexte du jeu Mammoth possédant un environnement triangularisé, peut-on réaliser une planification de chemin efficace et adaptée aux besoins d'un jeu vidéo ? »

Pour répondre à cela nous expliquons d'abord le principe de la planification de chemin, puis nous exposons les techniques existantes. Ensuite nous voyons l'amélioration effectuée dans Mammoth à ce sujet. Nous terminons sur une explication des techniques les plus appropriées dans ce contexte.

CHAPITRE I

ÉTAT DE L'ART DE LA PLANIFICATION DE CHEMIN

1.1 Qu'est-ce que la planification de chemin ?

La planification de chemin est un problème classé dans le domaine de l'intelligence artificielle en informatique. Des nombreux ouvrages ont été écrit à ce sujet comme (Russell et Norvig, 2010), (LaValle, 2006) et (Boulic, 2010). Elle permet à un personnage, objet ou une unité de se déplacer dans un environnement en partant d'un configuration A (départ) et en se dirigeant vers un configuration B (destination) tout en évitant les obstacles présents dans cet environnement. L'objet doit rester dans une configuration valide durant toute la durée du trajet. L'environnement peut se retrouver modifié lors du déplacement de l'objet dépendamment du contexte.

La planification de chemin est un problème utilisé principalement dans la programmation d'assistants de navigation GPS (Global Positioning System), de jeux vidéo ou encore dans la robotique.

Le déplacement se fait donc automatiquement dans l'environnement et l'ordinateur possède les informations suivantes : configuration de départ, configuration d'arrivée et configuration des obstacles dans les cas les plus simples.

Le but de ce processus est de calculer un des chemins les plus adaptés au personnage entre le départ et l'arrivée (Figure 1.1).

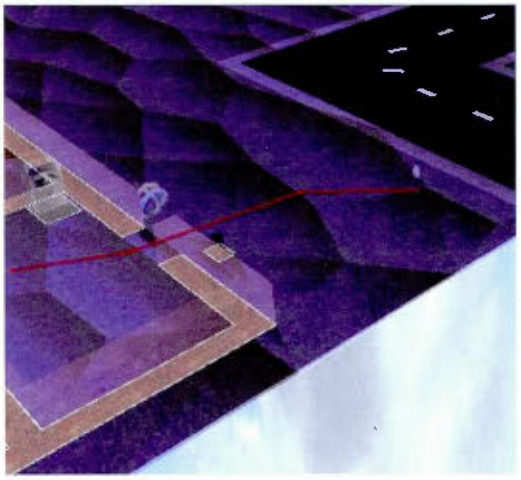

Figure 1.1 En rouge le chemin emprunté par le personnage

Le processus de la planification de chemin s'effectue en trois étapes :

- Discrétisation de l'espace continue en un espace discret
- Recherche de chemin dans l'espace discret trouvé
- Lissage d'un chemin obtenu.

Nous expliquons les types d'environnement que l'on peut rencontrer dans un jeu vidéo notamment.

1.2 Types d'environnement

Lorsqu'il s'agit de calculer la planification de chemin, les jeux vidéo utilisent des structures abstraites et donc invisibles pour le joueur afin de représenter les obstacles dans les environnements de jeu. Même si les environnements peuvent avoir l'air détaillés quand on joue à un jeu, en réalité seulement des structures simplifiées de ceux-ci sont prises en compte pour la planification de chemin. Certaines zones ou certains détails sont donc oubliés ou seulement présents pour le décor. Ainsi un personnage non-jouable peut seulement se déplacer dans ce type de structure. Voici quelques-uns des types de structures abstraites les plus répandus permettant de réaliser la planification de chemin.

1.2.1 Quadrillage

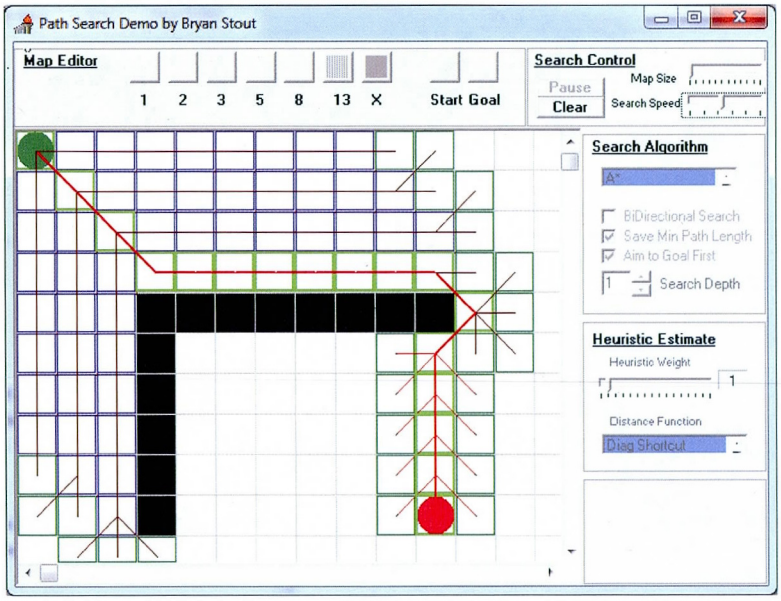

Figure 1.2 Quadrillage dans le logiciel Path Search Demo de Bryan Stout

Le quadrillage est l'un des types de structures abstraites les plus communs dans la planification de chemin. Le personnage peut se déplacer de case en case, horizontalement et verticalement. Dans les jeux vidéo utilisant un quadrillage, les déplacements en diagonales sont autorisés également. Dans la plupart des cas, toutes les cases ont la même taille et la même forme. Cela permet d'effectuer les calculs plus facilement et la représentation est plus simple. Plus le nombre de cases est grand, plus les performances demandées sont élevées. La Figure 1.2 montre un exemple de quadrillage. La planification de chemin s'effectue à partir du cercle en haut à gauche du quadrillage et se termine dans le second cercle.

1.2.2 Triangulation

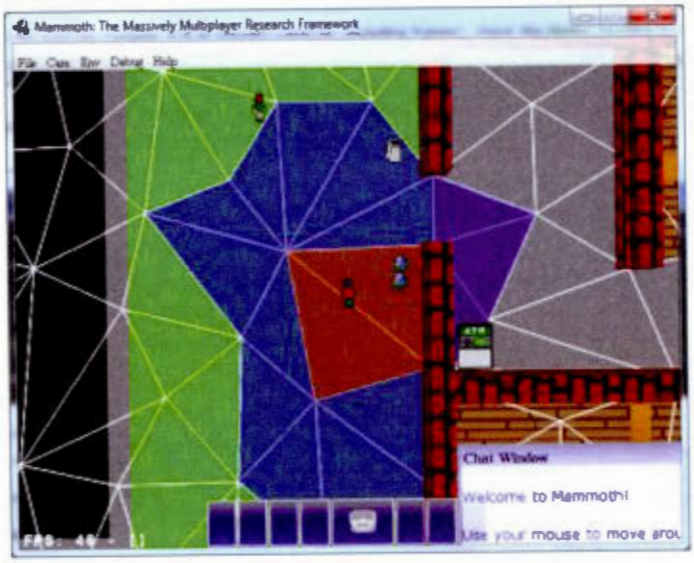

Figure 1.3 Exemple de triangulation dans Mammoth

Le personnage se déplace ici de triangle en triangle en ayant le choix de pouvoir se déplacer dans tous les triangles adjacents à celui correspondant à sa position actuelle. Dans un jeu vidéo, par exemple, on utilise les environnements triangularisés (Figure 1.3) pour mieux déterminer les limites des obstacles non rectangulaires. Là où il était évident, dans un environnement quadrillé, de passer d'une case à une autre en passant simplement par le centre, cela devient une tâche complexe lorsqu'il s'agit de triangles. En effet les triangles ici n'ont pas tous la même forme et la même taille, ne sont pas générés de façon identique.

Si on souhaite le faire dans un environnement quadrillé, cela nécessite également un nombre plus important de cases pour être précis et il faut délimiter les formes des obstacles correctement surtout lorsqu'ils ne sont pas rectangulaires. Avec un nombre si important de carrés, les performances de l'application se voient ainsi réduites et la mémoire saturée.

De plus, les objets dans un jeu ont une position sur la carte. Déterminer l'emplacement d'un objet cible sur une carte est une tâche aisée. Cependant, déterminer quels objets sont proches de ce même objet cible est une tâche bien plus complexe. Si la

carte est composé d'un seul espace continu, alors nous devons comparer chaque objet sur la carte à notre objet cible pour déterminer s'il est proche. Cela peut prendre beaucoup de ressources très rapidement si la carte contient un grand nombre d'objets. En divisant une carte en plusieurs petites zones, nous pouvons réduire le nombre de vérifications nécessaires pour déterminer le voisinage de l'objet cible.

Lorsque l'on utilise des zones sous la formes de cases (carrés, triangles...), nous devons simplement vérifier la case de l'objet cible et les cases voisines. Cela permet un très grand gain en performance pour de nombreux composants du jeu qui nécessitent de connaitre l'emplacement des objets, tels que la détection des collisions ou encore le comportement des NPCs.

Un dernier avantage à signaler pour la triangularisation est le fait que les triangles possèdent chacun 3 triangles voisins (ou moins si les triangles sont sur les bords d'une carte) ce qui fait moins d'informations à traiter comparées à un carré qui a toujours 4 carrés voisins.

Cette séparation est aussi très utile pour les algorithmes de planification de chemin. Le temps nécessaire pour trouver un chemin jusqu'à un emplacement spécifique peut être grandement réduit si le chemin est trouvé à un niveau d'abstraction d'abord. La planification de chemin peut ensuite être faite localement pour chaque case.

1.2.3 Waypoint graph

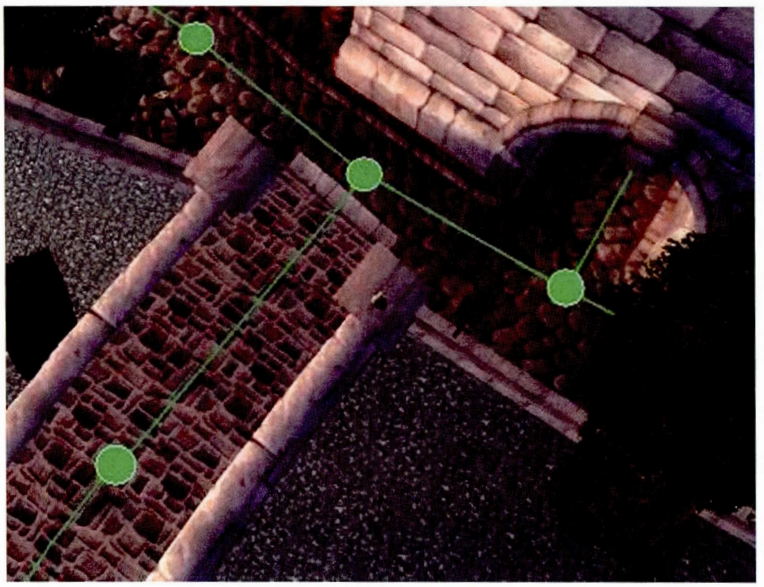

Figure 1.4 Exemple de waypoint dans World of Warcraft (Tozour, 2008)

Dans ce type de structure (Figure 1.4), tous les personnages passent par le même chemin prédéfini. C'est comme si les personnages suivaient un rail. C'est alors bien plus facile de prévoir les comportements et les collisions mais dans les environnements ouverts cela n'a pas beaucoup d'intérêt. En effet les personnages suivent un chemin et ne tiennent pas compte du reste des zones où ils peuvent circuler. C'est une technique utile pour les environnements exigus et complexes à calculer en temps normal par l'algorithme choisi. On définit alors un chemin fixe qui simplifie les trajectoires des personnages dans le tunnel ou la portion de terrain complexe.

Si un obstacle se trouve sur le chemin, le personnage se trouve bloqué car ce type de structure n'indique pas comment faire autrement. Nous pouvons bien entendu relancer une planification de chemin pour contourner l'obstacle mais cela prend alors plus de temps qu'une seule recherche.

1.2.4 Mesh de navigation

Figure 1.5 Mesh de navigation dans World of Warcraft (Tozour, 2008)

Un mesh de navigation est un autre type de structure abstraite au même titre que les quadrillages ou la triangulation. Elle permet aux personnages de se déplacer librement entre les différentes zones. Chaque zone peut être traversée en ligne droite par un personnage. Les zones ne sont donc pas des structures complexes et sont le plus souvent de formes rectangulaires ou triangulaires. Les environnements ouverts sont mieux couverts et sont donc séparés en zones.

C'est une structure utilisée dans les jeux vidéo afin de définir les endroits qui peuvent être parcourus par les NPCs dans les jeux. Ainsi les NPCs sauront quel chemin prendre dans une carte, quelles actions sont possibles lorsqu'ils voient un objet en hauteur, une fenêtre par laquelle passer ou une échelle.

La société Valve utilise des meshes de navigation notamment dans leurs jeux Counter Strike : Source et Left 4 Dead. (Nec, 2009). Les meshes de navigation sont plus adaptés aux structures en 3D sur plusieurs niveaux. Ils contiennent les informations nécessaires pour l'intelligence artificielle des NPCs tel que les endroits où ceux-ci peuvent grimper, se cacher, récupérer un objet....

À noter que certains auteurs parlent de meshes de navigation pour les quadrillages ou les triangulations également, il s'agit plus généralement du maillage permettant d'indiquer aux personnages les endroits où ils peuvent circuler.

Nous avons vu les types de structures abstraites les plus communes. Cependant, des contraintes liées à la planification de chemin peuvent se retrouver dans plusieurs de ces structures. Nous exposons alors les contraintes.

1.3 Contraintes de la planification de chemin

Au fil de nos recherches, nous avons pu observer que la planification de chemin est un domaine très vaste. Ainsi, nous expliquons certaines des contraintes existantes.

1.3.1 Taille de l'objet en déplacement

Lorsqu'un personnage se déplace dans une carte dans un jeu vidéo, il a toujours une taille minimum, un rayon minimum. Dans le cas contraire, il correspondrait à une forme sans masque de collision. Il pourrait donc traverser les murs puisque n'entrant pas en collision avec le reste des éléments. Ce dernier cas n'aurait alors que peu d'intérêt. Dans un environnement virtuel, un personnage a donc toujours un rayon de collision et a généralement une taille fixe généralement. Le personnage peut passer sur certaines routes, choisir certains chemins alors que d'autres se révéleront trop étroits. L'algorithme de planification de chemin peut malgré cela prendre quand même considérer les chemins trop étroits et petits et les inclure dans sa recherche. Il faut alors veiller à ce qu'il prenne en compte le rayon de collision pour chaque case qu'il envisage de parcourir.

Même si le personnage trouve un chemin approprié suffisamment large, il faut veiller à ce qu'il contourne les obstacles correctement. En effet, son rayon de collision peut bloquer le personnage dans un obstacle lorsqu'il essaye de tourner. De plus, si le personnage peut contourner d'une façon correcte et proche du naturel en faisant une courbe suffisamment large pour son rayon lorsqu'il tourne, cela est mieux pour la cohérence visuelle.

1.3.2 Vitesse

La vitesse du personnage peut influer sur plusieurs facteurs. Tout d'abord dépendamment de sa vitesse et de la vitesse des autres éléments de l'environnement, il

peut entrer en collision avec ceux-ci ou les éviter. Ensuite si sa vitesse lui permet de passer sur certains terrains (voir Nature du chemin) plus rapidement que sur d'autres, cela doit également être pris en compte. On peut également penser à des chemins différents en fonction de la vitesse du personnage comme dans l'exemple du GPS, où les autoroutes et les zones piétonnes peuvent être prises seulement par les véhicules motorisées ou les piétons respectivement.

1.3.3 Autres objets en déplacement

Naturellement les autres personnages, véhicules, éléments en déplacement ou en stationnement doivent être pris en compte dans la planification de chemin. Dans le cas, où d'autres éléments se déplacent, leurs chemins et leurs rayons de collision doivent être calculés. Si les chemins ne sont pas fixes et, par exemple, contrôlé par un joueur, il faut calculer la vitesse des autres éléments et leurs positions constamment pour éviter au personnage central de se retrouver bloqué.

1.3.4 Forme des autres objets

Dans le contexte des jeux vidéo, les jeux actuels contiennent de nombreux bâtiments, éléments de décor qui ne sont ni symétriques ni de forme rectangulaire. On a ici affaire à des courbes, lignes, triangles, rectangles. Il convient alors de trouver un moyen de prendre en compte ces formes dans la gestion des collisions. Lorsque l'environnement est quadrillé et qu'il comprend de nombreux détails, le nombre de cases doit être très important. Pour un environnement triangularisé, le nombre de triangles est moins important pour représenter le même nombre de détails au niveau des collisions. Les triangles n'ont cependant pas la même taille, forme et orientation.

Et voici ce qui se passe lorsque l'on essaye de le représenter de manière abstraite dans une grille (à gauche) et par une triangulation « contrainte » (à droite).

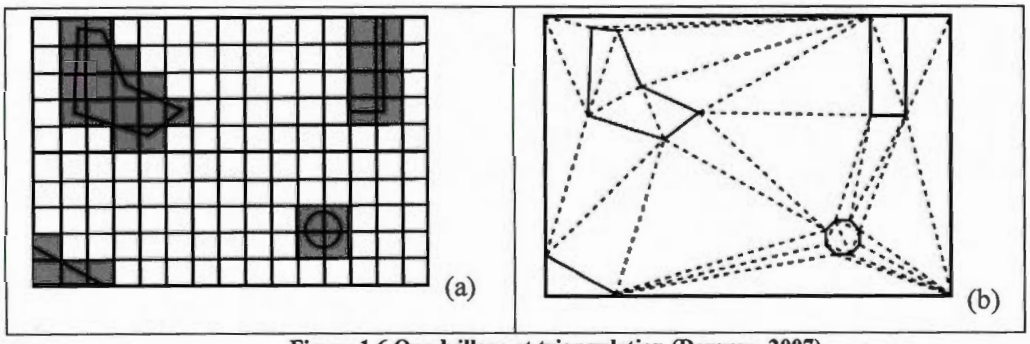

Figure 1.6 Quadrillage et triangulation (Demyen, 2007)

Sur la Figure 1.6 (a), les cases grisées représentent les masques de collision. Sur la Figure 1.6 (b), les obstacles sont bien déterminés par les triangles et donc les formes des obstacles constituent les zones où le personnage ne pourra se déplacer. Le cercle est alors composé par une approximation de plusieurs triangles.

Un autre point est le fait que, lorsque les obstacles sont trop petits dans un environnement virtuel, il convient de ne pas les prendre en compte dans les collisions. Le personnage peut alors passer à travers. Si on souhaite les prendre en compte, il faut alors avoir un niveau de détail accru pour les collisions et donc de nombreux carrés ou triangles supplémentaires dépendamment du type de structure abstraite.

1.3.5 Validité de la destination

Il se peut que le point de destination indiqué au personnage soit incorrect. Pour que le point soit incorrect, cela peut être soit un élément du décor, soit un emplacement hors de la carte de jeu ou soit un emplacement qui n'a aucun chemin valide. Le personnage peut alors réagir de plusieurs façons :

- ne pas bouger de son emplacement;
- se diriger à l'endroit le plus proche du point incorrect;
- tourner aux alentours du point incorrect jusqu'à ce qu'un chemin menant au point d'arrivée se libère et rende ce point valide.

L'idéal est que l'algorithme détecte directement que le point n'est pas accessible. Suivant les jeux, le personnage réagit le plus souvent de la deuxième façon ou dans une moindre mesure de la première façon.

1.3.6 Nature du chemin

Suivant les caractéristiques du personnage, sa vitesse peut être ralentie ou accélérée en fonction du terrain sur lequel il passe. Admettons qu'un personnage soit humain et qu'il passe sur des cases représentant du sable, il est donc ralenti. Dans la planification de chemin, cela pose un problème car un chemin peut paraitre plus court que les autres mais peut se révéler plus long à parcourir car contenant un terrain où le déplacement est ralenti. Il convient alors d'attribuer une pondération à chaque case ralentissant ou accélérant afin de pouvoir comparer les chemins entre eux. Les cases ralentissant le personnage comptent alors comme s'il s'agissait de plus de cases qu'il n'y en a en réalité.

Ainsi on retrouve cette pondération au niveau des cases traversées et de l'heuristique.

1.3.7 Position du personnage

La planification de chemin s'effectue principalement dans un environnement en 3 dimensions, cependant les déplacements ont lieu sur 2 dimensions seulement dans le contexte des GPS ou des jeux vidéo en général. Dans le cadre des jeux vidéo, il est difficile pour le joueur d'indiquer une position en 3 dimensions à partir d'un curseur et de la représentation du monde sur 2 axes correspondant à un écran. De plus, percevoir les chemins et les déplacements automatiques en 3 dimensions ainsi que l'action est une tâche difficile. Il existe cependant quelques exemples faisant exception. Ils ne sont cependant pas réputés pour leur prise en charge de la planification de chemin. La prise en compte de la 3ème dimension peut se limiter alors à quelques niveaux prédéfinis de hauteur par exemple (Pinter, 2001).

1.3.8 Impasses

Lorsque le point de destination ne se situe pas dans une des impasses de la carte de jeu, celles-ci deviennent une contrainte à prendre en compte lors de la planification de chemin. En effet, l'heuristique peut indiquer au planificateur que le point de destination est très proche alors qu'il se trouve de l'autre côté du mur de l'impasse. Il convient alors d'utiliser une technique pour éviter que la recherche ne calcule les cases, les triangles dans les impasses.

L'une d'entre elles est le fait de donner de l'importance à chacun des nœuds, des cases composant l'environnement. Ainsi les nœuds les moins connectés aux autres nœuds seront moins prioritaires dans la planification de chemin.

1.3.9 Couloirs

Lorsqu'un couloir est trouvé dans l'environnement, il convient alors de le calculer une fois et de le mettre en mémoire afin de simplifier toutes les planifications de chemin suivantes le concernant. Cependant dans le cas d'une triangulation, le chemin que le personnage emprunte dans un couloir peut être différent en fonction de son point de départ et point d'arrivée.

1.3.10 Tolérance aux passages étroits

Lorsque l'on programme un algorithme de planification de chemin, il faut inclure une donnée supplémentaire : une tolérance aux passages étroits. C'est-à-dire que plus cette variable est élevée, moins le personnage est enclin à passer dans des passages inférieurs à un certain rayon.

Au contraire, lorsque la variable est basse, le personnage prendra en compte les chemins étroits pour passer. Si cette solution parait la plus adaptée, elle est en réalité plus lente car elle doit chercher un nombre de chemins plus important.

Il faut donc trouver une variable de tolérance équilibrée entre le minimum et le maximum afin de combiner performances et efficacité de l'algorithme.

1.3.11 Heuristique

L'heuristique est une estimation de la distance entre deux nœuds. Elle se choisit en fonction de l'environnement et du nombre de directions autorisé par celui-ci et l'unité qui se déplace.

Un mauvais choix de l'heuristique peut biaiser la recherche et déséquilibrer le coût du déplacement vers l'arrivée par rapport au coût du déplacement déjà effectué depuis le point de départ. Il faut éviter principalement que l'heuristique ne surestime le coût pour qu'elle soit admissible.

1.3.12 Rayon de déplacement et direction d'une unité

Lorsqu'un personnage se déplace dans un environnement, il peut se diriger dans toutes les directions, à n'importe quel moment.

Lorsqu'une unité comme un véhicule se déplace, nous ne pouvons pas lui faire exécuter des rotations à 180° soudainement ou faire des tournants à 90° si nous souhaitons rendre le jeu crédible et réaliste. Ainsi la planification de chemin est très différente et nécessite des algorithmes appropriés à cette situation (Pinter, 2001).

Il faut ainsi prendre en compte le rayon de rotation du véhicule ou de l'unité.

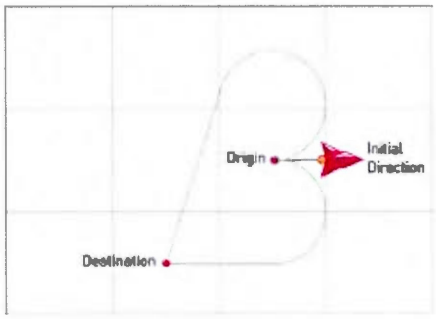

Figure 1.7 Rayon de rotation (Pinter, 2001)

La direction de l'unité est aussi importante. Le déplacement est calculé en fonction de celle-ci. Par exemple, toujours pour le cas d'un véhicule, celui-ci ne peut pas soudainement démarrer dans la direction inverse de celle où il pointe. De la même manière les tournants s'effectuent à partir de la direction du véhicule comme montré à la Figure 1.7.

Après avoir vu quels pouvaient être le type de problèmes rencontrés lors de la planification de chemin, nous allons voir les différents algorithmes permettant de discrétiser un espace continu en un espace discret.

1.4 Techniques de discrétisation de l'espace continu en un espace discret

Dans cette section nous verrons les algorithmes les plus connus en commençant par les algorithmes les plus basiques pour finir sur A* et certaines de ses variantes. Tous les

algorithmes suivants peuvent être adaptés, modifiés pour d'autres types de structures comme des triangles, des hexagones également. Ce sont des algorithmes abstraits.

1.4.1 Dijkstra

L'algorithme de Dijkstra (Dijkstra, 1959) parcourt toutes les cases possibles en tenant compte de leurs coûts respectifs. Lorsque tous les chemins sont visités, l'algorithme compare les coûts. Il trouvera ainsi tous les chemins minimums possibles ou le seul chemin minimum. C'est l'un des algorithmes les plus connus dû à son efficacité pour trouver les chemins les plus courts. Il a été conçu par Edsger Dijkstra en 1959.

1.4.2 Breadth-First Search (BFS)

Il s'agit d'un algorithme de parcours en largeur. Il parcourt tous les chemins d'une seule case à chaque itération. Il s'arrête lorsqu'il a trouvé un chemin entre le départ et l'arrivée.

1.4.3 Depth-First Search (DFS)

Cet algorithme est traduit par « parcours en profondeur » en français. Il va au bout de chaque chemin. Il les parcourt un à un au fur et à mesure. Ce n'est pas un algorithme très efficace lorsqu'il y a beaucoup de chemins et que ceux-ci sont longs. Il s'arrête lorsqu'il a trouvé un chemin (Cormen, 2001).

1.4.4 Iterative-Deepening Depth-First Search (IDDFS)

Il s'agit également d'un parcours en profondeur à la différence près que le nombre de cases explorées dans chaque chemin va augmenter à chaque itération (Demyen, 2007). Il va explorer le chemin 1 jusqu'à une profondeur seuil, puis le chemin 2 jusqu'à la même profondeur et ainsi de suite pour tous les chemins. À la prochaine itération, la profondeur seuil augmente et l'algorithme parcourt chaque chemin de la même manière. Il s'agit d'un mélange entre parcours en largeur et parcours en profondeur. L'algorithme s'arrête lorsqu'il a trouvé un chemin.

1.4.5 Best-First-Search

Best First Search (Dechter et Pearl, 1985) est un type d'algorithme de planification de chemin, qui calcule d'abord le chemin le plus prometteur avant tous les autres. Pour

estimer quel est le chemin le plus intéressant, ce type d'algorithme utilise une heuristique, c'est-à-dire une estimation de la distance jusqu'à l'arrivée. A* et B* sont donc des algorithmes de type Best First Search.

1.4.6 B * (B Star)

B Star est un algorithme de type Best First Search (Berliner, 1979). Il utilise un arbre pour répertorier tous les chemins possibles. Ensuite il va se concentrer sur la branche avec le coût le moins important. Si celle-ci ne parvient pas à l'arrivée, il prendra une autre branche avec un coût peu important.

1.4.7 A * (A Star)

A-Star Search (A*) (Hart, Nilsson et Raphael, 1968) est un des algorithmes classiques les plus robustes car il trouve toujours un chemin s'il en existe un. Il allie également un bon ratio performance/qualité du chemin trouvé comparativement aux autres algorithmes, ce qui est nécessaire dans un jeu vidéo. Il faut que le temps de réaction du programme par rapport à l'ordre du joueur, soit proche de l'instantané. Contrairement à la majorité des algorithmes, il utilise une heuristique de la distance entre la position du personnage et le but, c'est une des caractéristiques qui permet de le différencier. Son seul désavantage notable est le fait qu'il ne trouve pas toujours le meilleur chemin si l'heuristique n'est pas admissible alors que l'algorithme de Dijkstra le trouvera toujours. Pour obtenir une heuristique admissible, il faut que l'heuristique ne surestime dans aucun cas la distance jusqu'à la destination.

De nombreuses variantes ont vu le jour depuis sa création en 1968 et sont expliquées plus tard dans ce mémoire.

A* utilise deux listes de d'états : une liste « ouverte » et une « fermée ». La liste ouverte contient toutes les cases où le personnage peut éventuellement se déplacer, alors que la liste fermée contient les cases déjà parcourues.

L'heuristique évalue la distance jusqu'à l'arrivée à chaque case qu'elle traverse. Généralement notée h, elle peut diminuer à chaque pas si elle s'approche du point de destination par rapport au précédent pas. Une autre variable est g, mesurant le chemin parcouru. Ainsi plus le nombre de déplacements est élevé, plus la variable est importante.

L'algorithme additionne les variables g et h pour donner la variable f. Toutes les variables f de chacun des chemins pris par l'algorithme sont ensuite comparées afin de trouver le chemin le plus court. L'algorithme s'arrête dès qu'il a trouvé le point d'arrivée et retourne le chemin parcouru pour s'y rendre.

L'Algorithme 1.1 présente le pseudo-code :

```
Fonction A*(départ, destination)
Initialise Liste ouverte et Liste fermée à liste vide
Ajoute départ à Liste ouverte
Coût g(départ) = 0
Coût f = Coût g(départ) + Coût h (départ, destination)

Tant que la liste ouverte n'est pas vide
    NoeudActuel = Nœud de la liste ouverte avec Coût f minimum
    Si NoeudActuel = destination
        Retourne chemin

    Supprime NoeudActuel de la liste ouverte
    Ajoute NoeudActuel à la liste fermée
    Pour chaque Voisin dans noeudsvoisins(NoeudActuel)
        Si le Voisin est dans la liste fermée
            Continue
        Coût g temporaire = coût g (NoeudActuel) + distance(NoeudActuel,
      Voisin)
        Si Voisin n'est pas dans la liste ouverte ou si le coût g temporaire < coût
        g (voisin)
            Si voisin n'est pas dans la liste ouverte
                Ajoute voisin à la liste ouverte
            Coût g(voisin) = Coût g temporaire
            Coût f(voisin) = Coût g (voisin) + + Coût h (voisin, destination)
retourne "impossible de trouver le chemin"
```

Algorithme 1.1 Pseudo Code de A Star (A*)

L'heuristique est surtout la caractéristique première de l'algorithme A*. S'il n'y a aucune heuristique alors l'algorithme A* est équivalent à un algorithme Dijkstra. Dans le cas contraire où l'heuristique est trop importante par rapport au coût de départ (ou historique), alors A* dépend majoritairement voire complètement de l'estimation et se transforme en Best-First-Search car se basant totalement sur l'heuristique et ne prenant plus en compte le coût G du chemin. Il faut donc trouver un équilibre entre le coût g et le coût h pour que l'algorithme se révèle efficace.

Comme A* est un algorithme datant de 1968, de nombreuses variantes de celui-ci ont vu le jour. Dans la suite du mémoire, nous expliquons le principe de certaines de ces variantes.

1.4.8 D * (D Star)

D* correspond à Dynamic A* (Stentz, 1995). Il s'inspire donc de A*. Il est dynamique car il prend en compte le fait que les chemins puissent changer de coût pendant que l'algorithme s'exécute. Quelques variantes existent possédant des améliorations dépendamment du contexte d'utilisation.

1.4.9 Iterative-Deepening A-Star Search (IDA*)

Cet algorithme parcourt chacun des chemins trouvés par A* et au lieu de stopper le parcours des chemins avec une profondeur seuil, il est stoppé lorsque que le coût du chemin calculé par A* dépasse un seuil précisé. Ainsi les chemins avec un coût trop important sont écartés. Si les chemins sont tous écartés, alors le coût du seuil est naturellement augmenté (Demyen, 2007).

1.4.10 Hierarchical Pathfinding A* (HPA*)

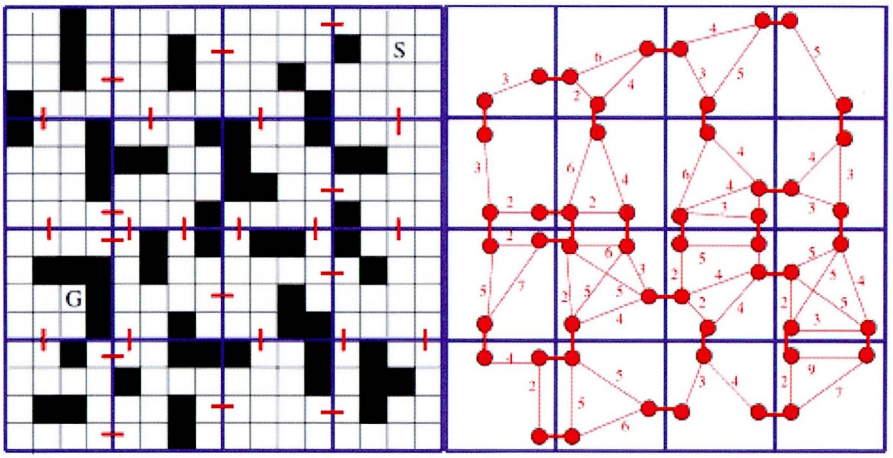

Figure 1.8 Hierarchical Path Finding (Botea, Müller et Schaeffer, 2004)

Cet algorithme (Botea, Müller et Schaeffer, 2004) divise une carte de jeu en plusieurs sections de façon à simplifier les déplacements entre les grandes zones. Les déplacements à l'intérieur des zones sont calculés normalement, alors que lorsqu'ils seront effectués entre plusieurs zones, les personnages se déplaceront sur des chemins prédéfinis et déjà calculés. La Figure 1.8 montre la division de la carte en de nombreux points. Les chemins entre ces points sont donc précalculés.

1.4.11 Generalized Hierarchical Pathfinding A* (gHPA*)

Cette variante de HPA* (Koch, 2011) permet de mieux prendre en compte les graphes orientés lorsque ceux-ci n'ont pas des arêtes avec un poids négatif par rapport au HPA* standard. En fait cette variante agrège les arêtes des graphes et rend ceux-ci plus abstraits. Ainsi les arêtes seront regroupées en fonction de leur distance euclidienne. Plus les arêtes sont proches, plus elles auront de chances de faire partie du même groupe.

1.4.12 Learning Real Time A* (LRTA*)

Cette variante d'A* (Bulitko et Lee, 2006) utilise une heuristique qui s'adapte en temps réel. C'est-à-dire que l'heuristique est révisée plusieurs fois au cours du parcours du chemin. Ce type de recherche permet d'être efficace lorsque les positions d'objets avec collision se déplacent également, ou alors que le terrain change.

Malheureusement le temps d'apprentissage d'une heuristique approximative est assez long pour ce type d'algorithme et ralentit donc la recherche d'après.

1.4.13 Partial Refinement A* (PRA*)

Cet algorithme (Sturtevant et Buro, 2005) produit plusieurs niveaux d'abstraction entre les cases d'un quadrillage afin de trouver un chemin entre le départ et la destination plus rapidement. De cette manière, l'algorithme permet de voir directement quelles cases sont connectées.

1.4.14 Probabilistic Road Map (PRM)

PRM (Kavraki *et al.*, 1996) est un algorithme de planification de chemin surtout utilisé dans la robotique. PRM, aussi discuté dans (Belghith *et al.*, 2006), utilise A* comme base. Le robot a une configuration de départ et une d'arrivée à atteindre en évitant évidemment les collisions. L'algorithme se base sur les espaces libres de l'environnement. Il teste chaque échantillon aux alentours du robot pour voir si c'est un chemin possible. Ensuite un algorithme de recherche de graphe est utilisé sur les résultats pour trouver un chemin.

Après avoir vu les techniques utilisables en environnement quadrillé mais aussi dans les environnements possédant des cases ayant la même taille, nous présentons les techniques spécifiques aux triangulations.

1.5 Techniques de discrétisation de l'espace continu en un espace discret dans une triangulation

1.5.1 Triangulation A* (TA*)

L'algorithme « Triangulation A* » permet d'abord de repérer par quelle série de triangles le personnage, l'unité ou l'objet doit passer dans la triangulation. Cette série de triangles contient donc l'un des plus courts chemins pour aller d'un point A à un point B. Pour déterminer cette série de triangles, l'algorithme se base sur l'algorithme classique A* et utilise donc une liste ouverte, une liste fermée et surtout une heuristique pour estimer les coûts de déplacements jusqu'à la destination.

L'algorithme considère naturellement chaque triangle comme un noeud. Pour estimer l'heuristique dans un environnement triangularisé, la distance euclidienne est utilisée entre le côté (connectant le nœud parent et le nœud enfant) et le point de destination. Le coût *g*, lui aussi, est mesuré entre le côté et le point de départ. Le côté du triangle est choisi en fonction de celui par lequel le personnage est entré.

1.5.2 Triangulation Reduction A* (TRA*)

Afin d'éviter d'effectuer des calculs supplémentaires à chaque planification de chemin, l'algorithme Triangulation Reduction A* (Demyen et Buro, 2006), (Demyen, 2007), (Erdtman et Fylling, 2008) produit une triangulation avec de nombreuses informations supplémentaires pour chacun des triangles. Les informations comme la taille maximum d'un masque de collision acceptée par un triangle, les triangles adjacents, la distance entre les triangles adjacents ou la zone d'appartenance du triangle sont autant de détails qui peuvent être inclus dans chacun des triangles. TRA* produit également un graphe à un niveau abstrait permettant d'observer quels sont les triangles interconnectés. Ainsi des structures d'arbres apparaissent tels que des boucles, des arbres avec ou sans racine, et autres points de décision.

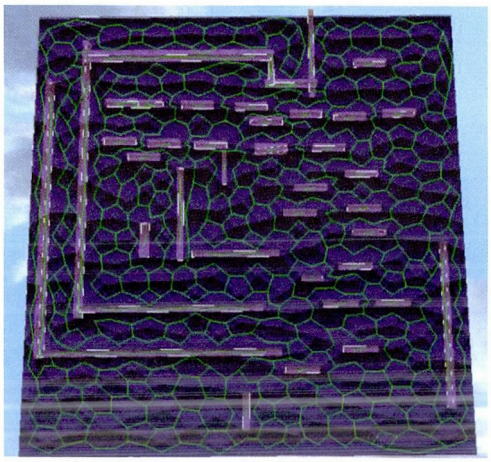

Figure 1.9 Graphe de TRA*

Sur la Figure 1.9, le graphe d'abstraction est représenté de couleur verte et montre la connexion entre chacun des triangles. Les triangles sont connectés et nommés adjacents lorsqu'il n'y a pas de côté bloquant ou d'espace vide entre eux.

Chacun des triangles du graphe a un niveau allant de 0 à 3 correspondant au nombre de triangles adjacents à celui-ci :

- Lorsqu'un triangle possède 3 triangles adjacents il est considéré comme un nœud décisionnel dans le graphe. Il a donc une importance particulière car le personnage peut prendre 3 chemins différents à partir de celui-ci.
- Lorsqu'un triangle possède 2 triangles adjacents, il s'agit d'un triangle qui sert juste de passage entre ses triangles adjacents. Il peut se trouver notamment dans un couloir.
- Un triangle possédant un seul triangle adjacent est considéré comme la fin d'un chemin, une impasse, et ne devrait pas être pris en compte lors de la planification de chemin sauf s'il s'agit du triangle de destination ou de départ.

Cette information est importante pour TRA* car certains cas spéciaux sont directement configurés dans l'algorithme. Par exemple, si le triangle de départ et de destination sont chacun de niveau 2 et sont situées dans une même boucle et on peut alors joindre le départ et l'arrivée des 2 côtés de la boucle. Ainsi les 2 distances sont calculées et comparées dans ce cas et la distance minimum est bien évidemment choisie.

Pour les cas spéciaux, des algorithmes de parcours de graphe sont utilisés. En dehors des cas spéciaux, il s'agit alors d'un algorithme TA* classique qui est utilisé.

Un algorithme comme TRA* permet de simplifier de nombreux calculs et est ainsi plus rapide que TA* lorsqu'il effectue la recherche dans des environnements exigus. Cependant dans des environnements ouverts, il ne semble pas proposer beaucoup plus de rapidité. De plus, l'inconvénient majeur est le fait qu'il prenne plus d'espace en mémoire à cause des informations supplémentaires et n'est donc pas adapté aux systèmes possédant peu de mémoire vive (Erdtman et Fylling, 2008). Sa taille en mémoire augmente de façon linéaire en fonction du nombre de triangles présents sur la carte.

Après avoir vu deux des algorithmes permettant de sélectionner les triangles dans une triangulation, nous voyons les algorithmes concernant le déplacement dans les triangles.

1.6 Techniques de recherche de chemin dans un espace discret triangularisé

1.6.1 Parcours par milieu des côtés intérieurs

Une des techniques consiste à parcourir la série de triangles (déterminée auparavant) par le milieu des côtés intérieurs. Malheureusement cette technique ordonne au personnage de marcher en zig-zag lorsque les triangles sont en ligne droite. De plus, la distance parcourue est supérieure lorsque le personnage prend un tournant, car au lieu de prendre la courbe la plus courte, le personnage prendra donc le milieu de chaque triangle.

Cette technique a beau être peu appropriée, elle est malgré tout utilisée dans certains jeux vidéo récents comme Left 4 Dead (Booth, 2009) ou d'autres situations (Ufkes, 2010). Sur la Figure 1.10 on peut observer la différence entre le chemin naturel et celui passant par les milieux des triangles

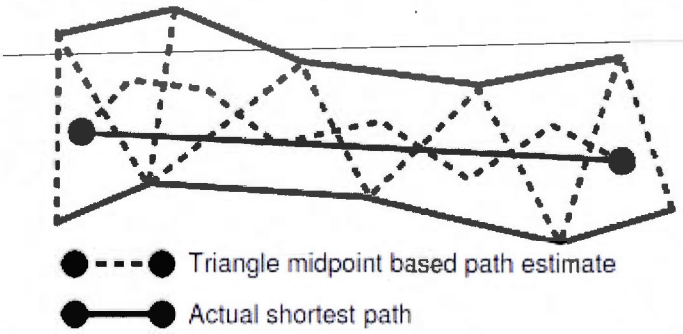

Figure 1.10 Centres des triangles.(Demyen, 2007)

Comme on peut le voir sur la Figure 1.11 donc, la distance parcourue est bien supérieure (chemin en rouge) au chemin le plus court:

Figure 1.11 Parcours des milieux des côtés intérieurs.

Cette technique est surtout utilisée à des fins de simplicité mais n'est pas très réaliste.

1.6.2 Parcours par centre des triangles

De la même manière que pour le parcours par milieu des côtés intérieurs, on peut simplifier le parcours en passant par le centre des triangles. Malheureusement les inconvénients sont les mêmes que pour la technique pré-citée. La Figure 1.12 montre cela en prenant un parcours plus long que nécessaire.

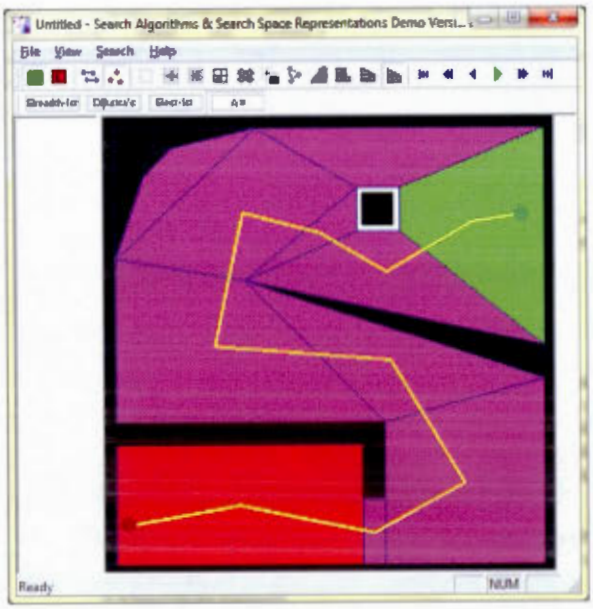

Figure 1.12 Pathfinding Demo par (Tozour, 2008)

1.6.3 Algorithme funnel

L'algorithme funnel (Lee et Preparata, 1984) est un algorithme de planification de chemin prévu pour les environnements triangularisés seulement. Il intervient après un algorithme ayant sélectionné les triangles comme nœuds auparavant comme Triangulation A*, par exemple. L'algorithme utilise alors les côtés intérieurs des triangles (dans la série de triangles déterminée par l'algorithme pré-cité) pour se repérer. C'est pour cela qu'il est nécessaire d'utiliser un algorithme comme celui présenté, permettant de prendre le minimum de trajet dans chacun des triangles traversés.

L'algorithme se déroule de la manière suivante : le principe est d'essayer de faire traverser le personnage en ligne droite dans le plus de triangles possibles dans la série de triangles. Toutes les directions dépendent de l'emplacement du point d'arrivée, le chemin peut changer complètement pour une même série de triangles dépendamment de la destination. Le chemin s'oriente donc constamment vers la destination.

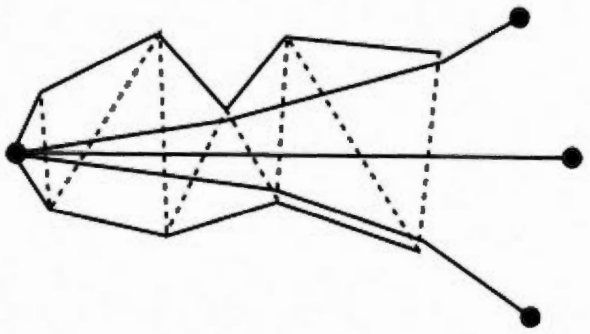

Figure 1.13 Trajectoires dépendant de la destination (Demyen, 2007)

Lorsque l'algorithme funnel ne peut plus se déplacer de manière droite dans les triangles, il va alors utiliser les tournants du polygone pour tourner dans la direction de l'arrivée, puis recommencer à aller droit puis contourner le prochain tournant, etc. jusqu'à ce que la destination soit atteinte. Pour arriver à cela, l'Algorithme 1.2 algorithme opère en suivant ces étapes (illustré par la Figure 1.14):

Chaque fois que l'on traite un nouveau portail:
- On vérifie que les points à gauche et à droite sont à l'intérieur de l'entonnoir

(funnel en anglais), s'ils le sont, on rétrécit simplement l'entonnoir. (A-D)

- Si le nouveau point de fin à gauche est en dehors de l'entonnoir, alors on ne met pas à jour les points de l'entonnoir (F).
- Si le nouveau point à gauche (en bleu) se superpose à celui du droit de l'entonnoir (en orange) (E), alors on ajoute la nouvelle ligne comme un des segments du chemin final et on place le nouveau sommet de l'entonnoir à la fin de ce segment. On redémarre l'algorithme à partir de ce point. (G)

Source : (Mononen, 2010)

Algorithme 1.2 Algorithme funnel simple

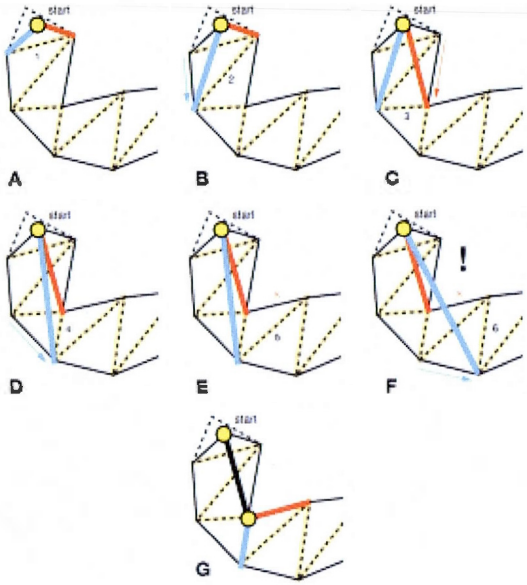

Figure 1.14 Le fonctionnement de l'algorithme funnel (Mononen, 2010)

On met à jour le funnel en ajoutant peu à peu les points à chaque côté de la deque. Ce sont les côtés intérieurs qui sont pris en compte pour la boucle. Le nombre de côtés intérieurs est égal à $(n-1)$, le nombre de triangles. Pour deux côtés intérieurs qui se suivent il y a toujours un côté qu'ils ont en commun (Figure 1.14).

Pour que les segments soient mis à jour à gauche et à droite, on utilise une astuce de programmation visible ici :

```
vl' = LeftEndpoint(ci)
        vr' = RightEndpoint(ci)
  //si le nouveau point gauche est égal au précédent
If vl' = vl then
        vr = vr'
        Add(f, vr, Right, p)
Else
        vl = vl'
        Add(f, vl, Left, p)
End if
```

Algorithme 1.3 Alternance gauche/droite pour l'algorithme funnel

A l'initialisation, les variables « *vl* » et « *vr* » (pour *vertex left* et *vertex right*) sont initialisées comme le point de départ. Ensuite « *vl'* » et « *vr'* » prennent les points du premier côté intérieur (interior edge). La structure conditionnelle permet ensuite de comparer s'il s'agit du premier point gauche du côté est égal au point de départ. Ce n'est généralement pas le cas, donc l'algorithme va passer dans le « *else* ». Dans le « *else* », « *vl* » devient le point gauche du premier côté intérieur et c'est cela qui va permettre de passer dans la partie if de cette structure conditionnelle à la prochaine itération. De la même manière « *vr = vr'* » va permettre d'aller dans le « else » à la prochaine itération etc. À noter que « *vl'* » et « *vr'* » changent à chaque itération pour devenir les nouveaux points des côtés intérieurs.

Une autre version de l'algorithme funnel a été proposée par Mikko Mononen sur son site (Mononen, 2010) en mars 2010. Elle ne prend pas en compte le rayon de l'objet non plus. Sa particularité réside dans le fait qu'au lieu d'utiliser des files et des listes, l'algorithme tourne en boucle et redémarre la boucle au début lorsqu'un nouveau tournant est ajouté. L'algorithme en question est disponible en langage C++ à l'adresse indiquée dans la référence.

L'algorithme funnel est donc adapté au parcours des séries de triangles, cependant il comporte un inconvénient majeur qui l'empêche d'être utilisé en pratique : il ne prend pas en compte le rayon de collision du personnage se déplaçant. Nous voyons donc maintenant l'algorithme funnel modifié résolvant ce problème.

1.6.4 Algorithme funnel modifié

L'algorithme funnel modifié prend en compte le fait que les personnages, véhicules et autres unités qui se déplacent ont un masque de collision avec une taille supérieure à 0. Même dans le cas où le personnage a un rayon égal ou proche de 0, il convient de lui faire traverser le monde de manière réaliste et humaine. En effet, lorsque l'on circule dans la vie, on évite notamment de s'égratigner les épaules sur les murs ou autres obstacles que nous contournons.

Dans un jeu et plus particulièrement dans le cas de cet algorithme, nous essayons de reproduire le même comportement. Ci-dessous l'objet est en gris. C'est celui qui va se déplacer dans le monde. Sur la Figure 1.15, on voit tous les obstacles que l'objet doit éviter lors de son déplacement.

On peut également ajouter un rayon (dilatation) à tous les obstacles afin de permettre à l'objet de les contourner correctement. Cette deuxième technique est nommée la somme de Minkowski. Cette distance est nécessaire pour un réalisme accru.

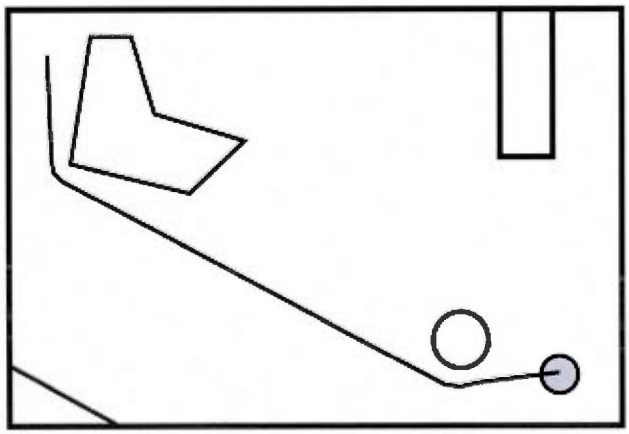

Figure 1.15 Chemin que l'objet avec un rayon peut prendre (Demyen, 2007)

Le chemin est alors calculé en fonction des obstacles. L'objet peut donc se déplacer sur le chemin calculé en évitant de manière fluide les obstacles. L'algorithme utilise le même principe que l'algorithme classique pour le reste des opérations. Le pseudo-code de l'algorithme modifié pour prendre en compte les rayons des objets est disponible dans la référence (Demyen, 2007).

1.6.5 Complexité de l'algorithme funnel

Le problème pour trouver le chemin le plus court en distance euclidienne tout en évitant les obstacles est un problème connu qui a donné lieu à de nombreux algorithmes (Lee et Preparata, 1984). L'algorithme funnel est l'un d'entre eux. A part l'algorithme expliqué dans ce rapport, l'un des algorithmes connus est un algorithme de Dijkstra. Celui-ci s'exécute en temps :

$$O\left((n+m)\log n\right)$$

n étant l'ensemble de sommets et m le nombre d'arêtes dans le graphe où l'on cherche le chemin (Lee et Preparata, 1984). L'auteur de l'article (Lee et Preparata, 1984) décrit les manières pour trouver la distance euclidienne dans plusieurs cas. Lorsqu'il s'agit de traverser un polygone simple, il faut d'abord le trianguliariser. Cette triangularisation est faite en temps $O\ (n\ log\ n)$. Le parcours de ce polygone est ensuite exécuté en temps $O(n)$. Les deux tâches sont exécutées en temps $O(n\ log\ n)$ lorsqu'on combine les temps, et lorsqu'on précalcule celles-ci. La complexité se réduit à $O(n)$ pour trouver le chemin entre 2 points dans un polygone avec n sommets. Les vertices sont en fait les emplacements dans le polygone ou plusieurs segments se rejoignent. Pour trouver le chemin le plus court dans un environnement avec des obstacles représentés par des lignes parallèles, l'auteur décrit que cela peut être trouvé en temps $O(n\ log\ n)$ et espace $O(n)$ (Lee et Preparata, 1984). Il s'agit donc de deux types de recherche de plus court chemin supplémentaires, l'une dans un polygone, l'autre dans un environnement avec obstacles représentés par ligne parallèle.

L'algorithme funnel peut être exécuté en temps linéaire dépendant du nombre de triangles présents dans la série, c'est-à-dire que sa complexité correspond à (n) , n étant donc le nombre de triangles (Demyen, 2007).

On remarque qu'il y a une seule boucle lancée dans la fonction « add » dans cet algorithme. Cette boucle dépend de la condition « if else » mais pour chaque point que l'on ajoute, on passe une seule fois dans cette boucle. La fonction « add » est appelée plusieurs fois par contre dans la fonction « funnel ».

```
Funnel(Canal c, Rayon r, Point s, Point g) : Path
1.  p.Clear()
2.  if NumEdges(c) < 1 then
        i.   p.Add(s);
        ii.  p.Add(g);
        iii. return p
3.  end if

4.  f = FunnelDeque(s, r)
5.  vl = LeftEndpoint(c0)
6.  Add(f, vl, Left, p)
7.  vr = RightEndpoint(c0)
8.  Add(f, vr, Right, p)

9.  for i = 1 to NumEdges(c) do
        i.   vl' = LeftEndpoint(ci)
        ii.  vr' = RightEndpoint(ci)
        iii. if vl' = vl then
        iv.  vr = vr' ;
        v.   Add(f, vr, Right, p);
        vi.  else
        vii. vl = vl';
        viii. Add(f, vl, Left, p);
10. end if
11. end for
12. Add(f, g, Point, p)
13. return p
```

Algorithme 1.4 Algorithme funnel

Dans la boucle « for »(ligne 9) de la fonction « funnel », la fonction « add » est appelé à chaque itération. Cette boucle « for » (ligne 9) a alors une complexité $O(n)$ alors que la boucle dans « add » (ligne 6,8, 9.v, 9.viii) a une complexité constante $O(1)$. La fonction « add » (ligne 6,8, 9.v, 9.viii) parcourt toujours les côtés des triangles qu'elle traite mais ce nombre ne change pas. Un triangle a toujours le même nombre de côtés et le même nombre de voisins. L'algorithme a une complexité de $O(n)$ dépendant du nombre de côtés présents dans la série de triangles. S'il y a 200 triangles à traiter alors la boucle « for » mettra deux fois plus de temps à s'exécuter qu'avec 100 triangles. Les ajouts de l'algorithme funnel modifié ne changent pas la complexité de celui-ci.

La complexité de l'algorithme de Mikko Mononen (Mononen, 2010) ne change pas. Elle dépend toujours du nombre de triangles dans la série.

Nous avons ainsi vu les techniques de planification de chemin principales dans une grille et dans une triangulation.

CHAPITRE II

CONTEXTE ET MÉTHODOLOGIE

2.1 Contexte

Nous présentons ce qui existe dans Mammoth avant le travail de recherche concernant la planification de chemin.

2.1.1 Mammoth en général

Mammoth étant un MMOG avec de très nombreux joueurs, il faut que les personnages puissent se déplacer dans un environnement ouvert pour avoir suffisamment d'espace pour se déplacer. Plusieurs versions de Mammoth ont existées en 2D d'abord puis en 3D. Cependant le déplacement s'effectue toujours sur un plan 2D.

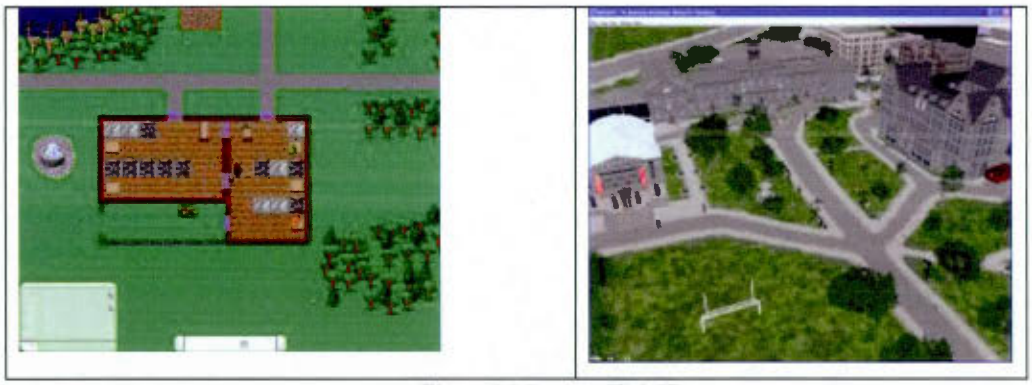

Figure 2.1 Passage 2D à 3D

Même après le passage à la 3ème dimension, le moteur graphique et les graphismes ont changé. Le jeu utilise désormais la version 3 du JMonkey Engine au lieu de la 2.

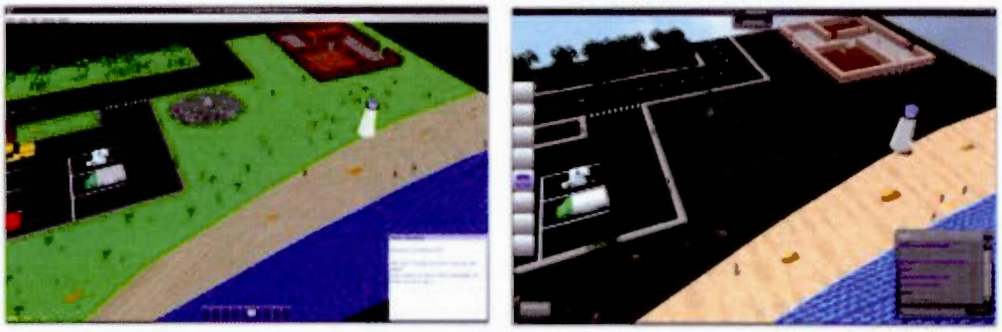

Figure 2.2 Passage JMonkey Engine 2 à 3

Mammoth possède un éditeur de cartes de jeux. Très utile, il permet de créer une carte, de la modifier, de la sauvegarder afin de l'utiliser dans le jeu actuel. C'est aussi à partir de l'interface de l'éditeur que l'on sélectionne le type de triangulation que l'on souhaite, les paramètres correspondants ainsi que les méthodes que l'on souhaite lancer.

2.1.2 Contrôle

Dans Mammoth, le personnage peut être dirigé de plusieurs façons. Il peut tout d'abord être orienté à l'aide des touches multi-directionnelles du clavier vers le haut, le bas, la gauche ou la droite. Le mouvement est alors relatif à la caméra (originellement le mouvement était absolu et basé en fonction de la carte du jeu mais il a été modifié car le contrôle relatif est bien plus naturel pour les joueurs). Le personnage peut ensuite être dirigé grâce à la souris : un clic gauche sur la carte permet de le déplacer en ligne droite vers l'endroit pointé alors qu'un clic droit utilise l'algorithme de planification de chemin choisi dans les propriétés du jeu pour se rendre à ce même point. Il va de soi que lorsque le personnage se déplace en ligne droite et qu'il rencontre un obstacle, il s'arrête contre l'obstacle.

Le contrôle au clavier est également désormais disponible à des fins de tests. En effet, lorsque des éléments bloquent le personnage dans le décor, il est plus facile de le sortir de cette situation avec les contrôles au clavier, car ils sont plus précis que les contrôles de souris.

2.1.3 Planification de chemin A*

L'algorithme A*, au lieu d'utiliser un quadrillage, utilise des positions dans la carte de jeu qu'il transforme en nœuds. Les nœuds sont alors considérés comme des cases et possèdent alors chacun un coût f égal à g et h.

Pour éviter que l'algorithme ne prenne trop de temps à chercher un chemin, une limite de temps est fixée à une seconde. Si l'algorithme ne trouve pas suffisamment rapidement, alors la recherche s'annule et le personnage reste sur place. Cela permet de ne pas monopoliser les ressources de l'ordinateur mais le joueur doit alors choisir une destination plus proche pour que le personnage se déplace.

Pour A*, la fenêtre de débogage affiche les cases explorées. Les cases parcourables sont affichées en bleu et les obstacles en blanc.

2.1.4 Balayage de la zone (A*)

Puisque l'algorithme A* implémenté dans Mammoth ne se base pas sur des cases ou des triangles, il doit trouver des nœuds symbolisés par des positions clés. Ces positions sont déterminées par le niveau de granularité choisi. Cependant pour déterminer si le personnage peut se rendre d'un point A à un point B, il ne suffit pas seulement de tester si le point B est un élément bloquant, il faut également tester si tous les points entre les deux sont bloquants. En effet, si le niveau de granularité est élevé, cela signifie qu'une grande distance se trouve entre les deux points et qu'un obstacle peut se trouver entre ceux-ci. De plus, cela permet d'éviter les zig-zags dans les couloirs en diagonales.

Sur la Figure 2.3, on peut observer que l'algorithme ne peut avoir connaissance de l'obstacle entre les deux points de navigation sans utiliser la méthode de balayage entre eux.

Figure 2.3 Exemple de balayage de la zone 1(Despland, 2011)

De la même manière, le balayage de la zone permet un chemin en ligne droite dans le cas de la figure ci-dessus.

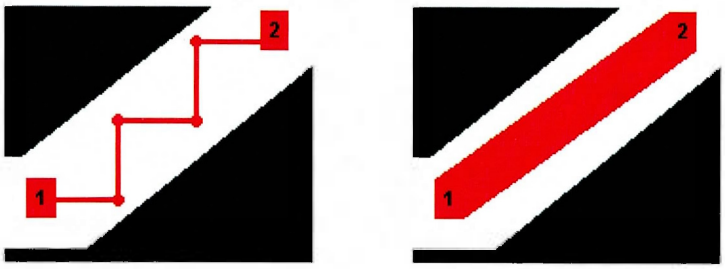

Figure 2.4 Exemple de balayage de la zone 2 (Despland, 2011)

C'est pourquoi il est préférable d'effectuer un balayage de la zone de déplacement entre les 2 points en tenant toujours compte du rayon, du masque de collision du personnage.

2.1.5 Planification de chemin TA*

Mammoth possède un environnement triangularisé à un niveau abstrait.

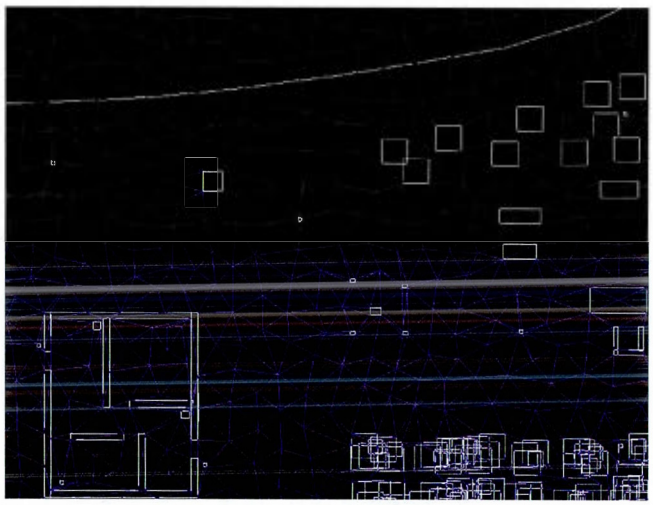

Figure 2.5 Triangulation d'une carte du jeu

Il existe deux types d'algorithmes implémentés afin de trianguláriser des cartes de jeu : Chew (Chew, 2007) et Quake (Shewchuk, 2005).

Ces algorithmes se basent sur la triangulation Delaunay. Ils prennent en compte également les obstacles, il s'agit alors d'une triangulation « Delaunay contrainte ». Chew et Quake vont produire des résultats différents sur une même carte. Sur la Figure 2.5, il s'agit d'une fenêtre de débogage permettant de voir la triangulation produite pour la carte. Dans ce cas, la triangulation Quake a été appliquée. Les triangles par lesquels le personnage peut passer sont de couleur bleue.

Il existe quelques méthodes implémentées dans Mammoth permettant de produire une triangulation plus efficace :

- Combiner les objets se superposant : afin de former un seul obstacle continu et d'éviter d'avoir des lignes bloquantes pour rien car se situant à l'intérieur de la superposition des obstacles.
- Convertir les murs en des lignes.
- Combiner les points à proximité : pour former un seul obstacle.
- Supprimer les petits polygones : ils n'auront ainsi pas de masque de collision mais ils ne nécessiteront pas d'augmenter le nombre de triangles pour les inclure dans le détail des obstacles.
- Convertir les petits polygones en lignes.

Il s'agit d'optimisations qui permettent d'obtenir une triangulation plus « propre », c'est-à-dire avec un nombre de triangles constant sur toute la carte, et avec des zones pouvant être parcourues facilement.

Pour la triangulation, Mammoth possède des fenêtres de débogage apparaissant lorsque l'utilisateur a recours à la planification de chemin.

Le débogage comporte deux fenêtres : une avec tous les triangles de la carte en bleu et les obstacles en blanc, l'autre avec le canal (ou la série) de triangles choisi et le parcours du personnage à l'intérieur de celui-ci. La première fenêtre affiche le parcours du personnage sur la carte en illuminant les triangles en jaune et en représentant les triangles d'arrivées et les triangles de départ en rouge et en vert respectivement.

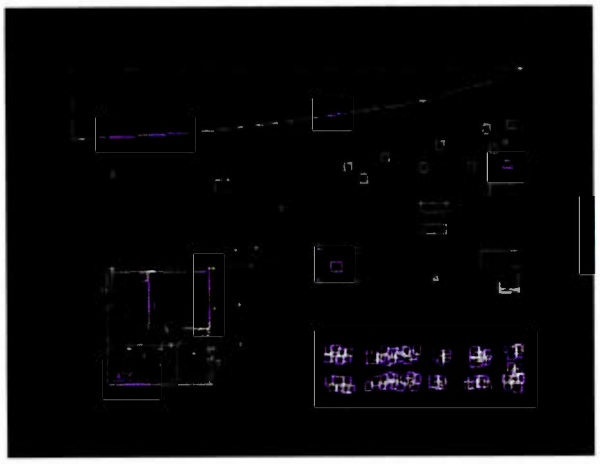

Figure 2.6 Fenêtre de débogage de la carte

La seconde fenêtre contient le détail du chemin calculé par l'ordinateur en montrant le canal en rouge, les côtés intérieurs en jaune et le chemin en vert.

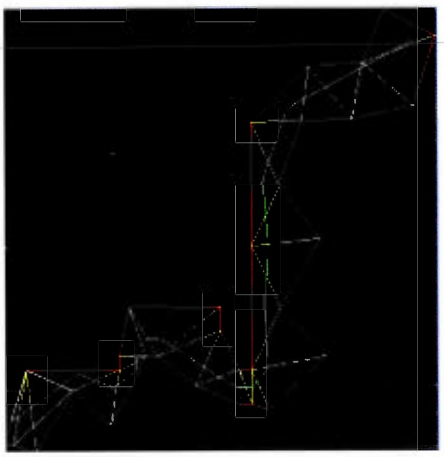

Figure 2.7 Fenêtre de débogage du canal

La planification de chemin est effectuée en deux étapes : d'abord les triangles reliant le départ et l'arrivée sont sélectionnés, ensuite le chemin est calculé dans la série de triangles.

La fenêtre, affichant la carte de jeu ainsi que tous les triangles, affichait auparavant tous les triangles sans distinction.

2.1.6 Clic du joueur transformé en triangle de destination

Dans Mammoth, afin de déterminer le chemin dans un environnement triangularisé, il faut transformer l'information du clic du joueur en point d'arrivée pour le personnage. Pour cela, lorsque le joueur clique à un endroit dans la fenêtre du jeu, à partir de son clic est projeté un rayon sur l'environnement de jeu. Ce rayon traduit le clic en coordonnées sur la carte. Ces coordonnées sont ensuite transformées afin de donner le triangle correspondant dans la triangularisation. Ainsi on obtient le triangle de destination. Les coordonnées restent en mémoire afin de déterminer la destination précise à l'intérieur du triangle d'arrivée.

Dans un environnement triangularisé, ce n'est pas toujours un clic qui détermine la position. A partir du moment où l'on possède les coordonnées de la destination, on peut déterminer dans quel triangle elles sont placées.

2.2 Méthodologie

Dans cette section, nous voyons les outils et techniques créés afin d'améliorer le confort d'utilisation de la planification de chemin pour les développeurs.

2.2.1 Validité de la destination

Les deux planifications de chemin se comportaient de cette manière : l'utilisateur cliquait à un endroit sur la carte, le programme initialisait les variables nécessaires à la planification de chemin, vérifiait ensuite si l'endroit où avait cliqué l'utilisateur était valide et lançait la planification de chemin. Il a alors été décidé qu'il est alors plus approprié de vérifier si la destination est valide juste après le clic de l'utilisateur afin d'éviter des calculs inutiles supplémentaires.

2.2.2 Outils pour la planification de chemin

Une fenêtre spécifique d'options pour la planification de chemin apparaît en appuyant sur la touche P (pour ***PathFinding***). Cette fenêtre (visible sur la Figure 2.8) permet de déboguer plus facilement le planificateur car elle permet de changer la position du personnage à des coordonnées spécifiées. Elle permet également de lancer la planification de chemin vers une destination spécifique, là encore en entrant les coordonnées. Ainsi en reproduisant les mêmes conditions à plusieurs reprises il est plus

facile de repérer une erreur qu'en essayant d'en faire autant en visant avec la souris. De plus les deux actions précisées ci-dessus s'effectuent à volonté en cliquant sur les boutons correspondants dans la fenêtre.

Enfin, la fenêtre permet de consulter la longueur du chemin emprunté (en cumulant les distances entre chaque point de navigation du chemin), la durée du calcul du chemin, les coordonnées actuelles du personnage ainsi que le statut de la planification de chemin.

Tous ces outils permettent d'obtenir des informations supplémentaires et de mieux comprendre le fonctionnement de la planification de chemin sélectionnée.

Figure 2.8 Fenêtre pour la planification de chemin

Une liste déroulante permettant de changer d'algorithme cherchant le chemin dans le canal, à la volée a été incluse de même. Ainsi les expérimentations entre les trois algorithmes traversant le canal sont bien plus faciles à exécuter.

2.2.3 Mesure du temps d'exécution du calcul de chemin

La mesure de la vitesse de calcul s'effectue en comparant le temps écoulé dans le jeu avant et après le calcul du chemin. À tout moment il est possible de prendre le temps écoulé grâce à une variable système. Nous soustrayons donc le temps écoulé avant le

calcul à celui après le calcul de temps. La différence donne le temps d'exécution du chemin.

Il est nécessaire pour ce genre de calcul d'éviter de le baser sur le cycle d'horloge du jeu, ou encore sur la mesure du temps dans le jeu lui-même si celui-ci en possède une. En effet, le jeu tournant sur un ordinateur, les vitesses et les configurations de multiples ordinateurs sont souvent tellement différentes que le jeu ne tournerait pas de la même manière dépendamment de la machine. On obtiendrait alors des résultats bien éloignés.

Mesurer le temps permet de comparer les algorithmes sont les plus efficaces et d'apprécier les optimisations.

2.2.4 Mesure de la distance parcourue

Pour mesurer la distance parcourue, on récupère le chemin calculé par l'algorithme et ensuite on calcule la somme de la distance de chacun des points. La somme donne naturellement la distance parcourue. Cela permet de comparer avec le chemin le plus court que ce soit avec Djisktra ou avec l'algorithme A*.

2.2.5 Planification de chemin A*

En ce qui concerne A* dans Mammoth, la fenêtre de débogage affiche désormais des informations comme les points de navigation empruntés ainsi que le point de départ et le point d'arrivée. Les points de navigation sont en jaunes. La case de départ en vert, et l'arrivée en rouge.

2.2.6 Fenêtres de débogage pour la triangulation

Les fenêtres de débogage effectuaient auparavant des boucles infinies afin de redessiner chacune des nouvelles informations au fur et à mesure. Les fenêtres réagissent désormais au clic de la souris seulement permettant un gain de performance non négligeable lors du chargement initial.

Avec un rafraichissement constant des fenêtres de débogage, le jeu nécessitait entre deux minutes et dix minutes pour se lancer suivant le nombre de triangles. Autant dire que pour déboguer le programme cela était très difficile, les multiples exécutions prenant énormément de temps. Dorénavant, le jeu se lance en moins de 30 secondes sur les ordinateurs du laboratoire de McGill et l'ordinateur de test.

Lorsqu'une carte de jeu a été créée avec les anciens outils de Mammoth, elle s'affiche inversée horizontalement dans la fenêtre de débogage. Nous avons conçu une méthode activable au besoin pour pallier ce défaut. Cette méthode inverse l'abscisse des points qui doivent être dessinés. Ainsi la carte apparait correctement dans le débogage.

2.2.7 Affichage de la triangulation en surimpression dans le jeu

L'étudiant à McGill, Etienne PEROT lors de la refonte du moteur graphique du jeu, a ajouté une couche d'affichage dans la fenêtre du jeu. Cet affichage montre la triangulation de la carte en surimpression et en transparence. On peut ainsi distinguer quels sont les triangles utilisés et comment par chacun des éléments de la carte. La couche affiche également les côtés bloquants des triangles, pour les obstacles donc et les chemins trouvés par le planificateur de chemin lorsque le joueur clique sur une destination.

Nous avons adapté cette couche à nos besoins en réduisant les lignes délimitant les obstacles notamment et en changeant la couleur afin de rendre l'affichage similaire à celles de la fenêtre de débogage. Nous avons aussi ajouté une fonction qui permet de calculer automatiquement le chemin lorsque l'on passe le curseur sur la carte. Cela nous permet de repérer directement les cas où la planification de chemin ne fonctionne pas.

Les trois sections suivantes concernent d'autres ajouts effectués à cette couche d'affichage supplémentaire.

2.2.8 Graphe abstrait pour Triangulation Reduction A*

Afin de visualiser directement quels sont les triangles connectés entre eux, nous avons implémenté la visualisation du graphe abstrait (Figure 2.9) dans Mammoth. Celui-ci se situe au même niveau que la couche d'affichage des triangles. Cette visualisation peut être activée ou désactivée selon les besoins dans le menu « Tile Window » qui permet de gérer l'apparition des triangles également.

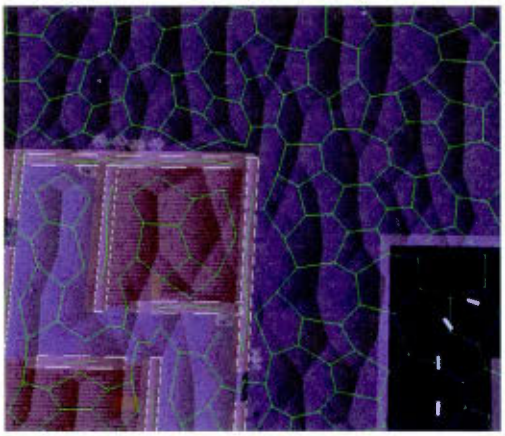

Figure 2.9 Graphe abstrait dans Mammoth

Dans le cas de l'algorithme TRA*, cela permet de visualiser directement l'importance des triangles. On voit directement les impasses, corridors et autres carrefours pour les personnages. Lorsque des bugs se produisent dans le voisinage des triangles dans certaines cartes de jeux, cet ajout permet également de les déceler plus rapidement. De la même manière que Douglas Demyen dans sa thèse (Demyen, 2007) qui donnait des formes différentes à ces triangles en fonction de leur niveau, nous avons donné des couleurs différentes aux centres des triangles dépendamment du niveau du nœud. Ainsi dans la version actuelle de Mammoth, les triangles de niveau 0 (îles) sont rouges, 1 (impasses) sont noirs, 2 (corridors) sont jaunes, 3 (nœuds décisionnels) sont verts comme montré sur la Figure 2.10.

Figure 2.10 Différents niveaux des triangles

Nous avons aussi permis de colorer les triangles en fonction de leur niveau, ce qui permet de voir directement lors d'un zoom plus distant, où se trouve les corridors, boucles et autres impasses. Sur la Figure 2.11, les triangles rouges sont de niveau 3, jaunes niveau 2, blanc niveau 1 et vert niveau 0. La coloration a changé entre les deux visualisations afin que chaque type de triangles soit repéré facilement dans chacune des situations.

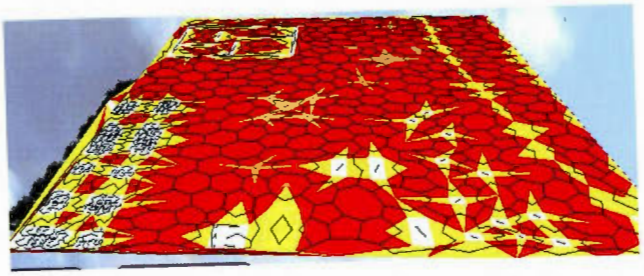

Figure 2.11 Différents niveaux des triangles avec colorisation

2.2.9 Affichage des triangles voisins

Comme le graphe abstrait affiche les triangles adjacents à chaque triangle, nous avons également ajouté la possibilité de voir les triangles calculés comme voisins dans la triangulation. Les triangles voisins sont les triangles partageant un sommet avec le triangle d'origine.

Ces triangles peuvent être utiles pour les fonctions réseaux de Mammoth et bien sur la planification de chemin.

Figure 2.12 Affichage triangles voisins

2.2.10 Affichage des groupes de triangles

Le précédent développeur de la planification de chemin sur Mammoth avait laissé un algorithme permettant de regrouper les triangles en plusieurs groupes. Ensuite l'algorithme TA* était lancé sur ces groupes.

Afin de distinguer quel triangle appartenait à quel groupe, nous les avons colorés lors de l'affichage des triangles. Cette coloration est désactivable. Cependant elle pose un problème de Z-Buffer. En effet, le programme ne sait pas quoi afficher à l'endroit des triangles car recevant des informations contradictoires (Figure 2.13).

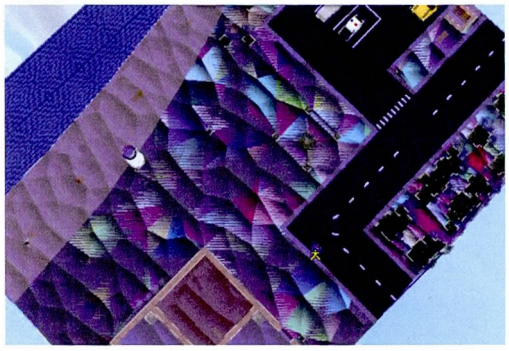

Figure 2.13 Affichage des groupes de triangles

Pour résoudre ce problème, il faut ajuster la hauteur d'affichage des triangles en fonction de la hauteur de la caméra.

Nous avons alors préféré relier les triangles se trouvant dans un même groupe en traçant des lignes entre le centre de chacun des triangles. De la même manière que pour les précédents affichages précités. Cela consomme ainsi moins de ressources et il n'y a pas besoin de modifier la caméra pour cela. Cependant l'affichage n'est pas très clair de cette façon. Après avoir corrigé le bogue de l'affichage des couleurs, nous obtenons ainsi les deux affichages suivants (Figure 2.14), disponibles au choix :

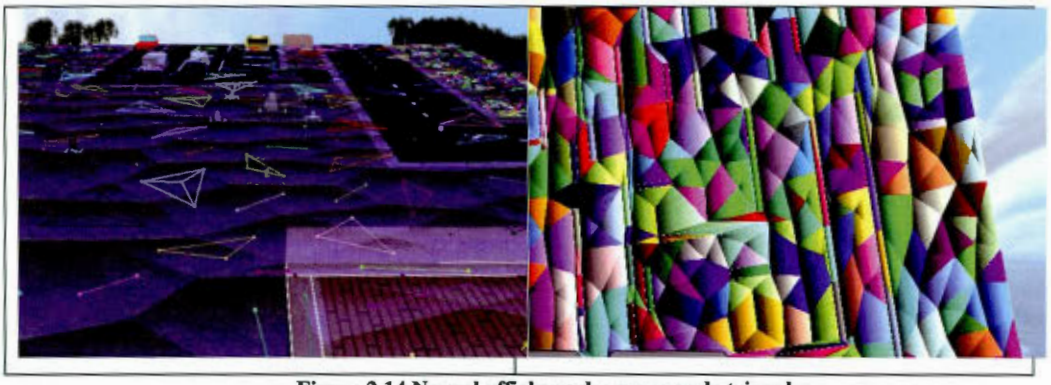

Figure 2.14 Nouvel affichage des groupes de triangles

Si l'affichage des groupes créés par l'ancien algorithme d'abstraction n'a pas d'incidence sur notre travail, il permet de réaliser directement de quelle manière il fonctionne en affichant les triangles appartenant au même groupe d'une même couleur. Ainsi la prochaine personne nécessitant d'utiliser cet algorithme pourra utiliser cet outil.

2.2.11 Paramétrage

Mammoth possède un fichier de configuration dans lequel on peut placer les paramètres afin de lancer le jeu de la manière que l'on souhaite. Le nombre de paramètres pour la planification de chemin a augmenté depuis notre travail sur le projet. Nous avons rendu paramétrables de nombreuses données qui étaient codées en dur auparavant.

Ainsi on peut activer les informations supplémentaires sur la planification de chemin dans la console et à l'écran, augmenter ou diminuer la granularité de A* ou le rayon de collision pour TA*. De la même manière on peut désormais activer ou désactiver l'ancien processus d'abstraction, la fenêtre de débogage pour l'algorithme funnel, celle pour TA* et enfin les informations liés à TA* dans la console. Tout ceci était lié dans un seul paramètre auparavant.

CHAPITRE III

IMPLÉMENTATION

3.1 Problématiques fondamentales

Dans cette partie, nous expliquons les problématiques rencontrées sur la planification de chemin lors de nos recherches.

3.1.1 Temps réel

Dans le contexte d'un monde dynamique, il convient de vérifier plusieurs fois le chemin sélectionné par le planificateur. En effet, si un chemin est sélectionné et que l'environnement est modifié, ou qu'un élément change de position sur le chemin, le personnage peut se retrouver bloqué. De plus si un élément avec un masque de collision croise le chemin calculé du personnage, cela bloque également le personnage.

Afin de résoudre cela, un calcul en temps réel du chemin peut être nécessaire. Le chemin est alors modifié au fur et à mesure du parcours et de l'avancement jusqu'à la destination. Cela demande bien sûr plus de temps de calcul et en général les objets dynamiques avec masque de collision sont peu nombreux dans les jeux vidéo pour éviter les calculs superflus.

Comme de nombreux MMOG, Mammoth n'a pas forcément de nombreux éléments dynamiques afin de minimiser les ressources. Ainsi presque toutes les contraintes rencontrées lors de la planification de chemin sont des contraintes pour un monde statique.

3.1.2 L'algorithme TA* n'est pas adapté aux environnements ouverts

Dans certains cas Triangle A* prend un chemin en fonction de l'heuristique et sélectionne tous les triangles adjacents les plus près de la destination. Cependant il peut

apparaitre qu'un chemin en ligne droite aurait été bien plus court que le chemin calculé. De la même manière même lorsqu'il s'agit de se diriger en ligne droite, il est difficile pour l'algorithme de trouver tous les triangles adjacents qui forment une ligne droite. Ainsi on pourra voir des zig-zags ou des courbes dans certains cas.

Une solution peut être la vérification pour chaque point de navigation s'il peut être atteint via une trajectoire en ligne droite. Si c'est le cas, la ligne droite est effectuée et la trajectoire continue d'être suivie. Si ce n'est pas le cas, le chemin est suivi normalement. Il est alors recommandé de lancer le test de ligne droite en partant du dernier point de navigation trouvé jusqu'aux premiers.

3.1.3 Réaction du jeu pendant le calcul du chemin

Pendant le calcul de la planification de chemin, il parait une bonne idée de faire diriger le personnage dans la direction de la destination pour que le joueur obtienne une réaction rapide de la part du programme suite à son clic. Cependant plusieurs problèmes se posent :

Pendant le calcul, le personnage va changer de point de départ. Ainsi le chemin ne sera plus exact puisque calculé pour des points différents.

- Faut-il alors que le personnage revienne sur ses pas pour reprendre le chemin ?
- Faut-il à nouveau calculer un chemin ?
  Lors du nouveau calcul, le personnage va de nouveau s'orienter dans une direction.
- Faut-il que le personnage se téléporte au point de départ initial ?

Une solution peut être de commencer à orienter le personnage dans la direction de la destination et lorsque le calcul du chemin complet est effectué, le personnage alors utilise à nouveau la planification de chemin afin de se rendre au premier point de navigation et de continuer le chemin.

Une seconde solution peut être d'utiliser, dans les cas où un algorithme de type hiérarchique est implémenté, les chemins principaux déjà calculés. Ainsi le personnage s'oriente et commence à suivre le chemin, et pendant ce temps-là, le reste du chemin est calculé en arrière-plan.

3.1.4 Threads

Lorsque l'on programme un algorithme de planification de chemin dans le contexte d'un jeu vidéo, il est recommandé d'utiliser un thread séparé pour cette fonction afin de ne pas stopper les autres fonctions actives étant en train d'être calculées également.

Plus la planification de chemin est éloignée et compliquée, et plus elle peut prendre des ressources. Dans certains cas spéciaux, elle peut prendre beaucoup de temps pour être calculée, stoppant donc le fonctionnement du jeu pendant le calcul.

Ainsi, en utilisant un thread, seul un personnage restera sur place le temps du calcul du chemin au lieu de ralentir le reste. Cependant, le calcul prend en général quelques millisecondes et reste invisible pour le joueur.

3.1.5 Solutions pour la validité de la destination

Lorsque le joueur clique sur un obstacle au lieu d'un endroit où le personnage peut circuler, Mammoth détecte que la destination entre en collision avec le personnage.

Le jeu essaye alors d'ajuster la destination afin d'approcher le personnage de celle-ci. Il existe alors deux solutions pour corriger la validité de de la destination dans Mammoth

- Faire varier aléatoirement le point où le joueur a cliqué aux alentours
- Changer le point de destination en le faisant reculer en ordonnées et en abscisses d'un pas. Un pas correspond à une valeur arbitraire par défaut multiplié par une variable de granularité qui indique la précision.

La deuxième solution peut sembler plus appropriée, car évitant le caractère aléatoire de la position de destination. Cependant après vérification, il s'avère que cette solution produisait des coordonnées pour la destination dans une seule direction. Dans le cas où un obstacle se trouve à l'extrémité d'une carte, la deuxième solution trouve de nouvelles coordonnées dans une seule direction et donc sort de la carte. Il s'agit en fait d'une mauvaise adaptation de la première solution. Nous avons alors corrigé les deux solutions et les avons fusionnées en une seule appropriée au contexte.

3.1.6 Techniques d'adoucissement de chemin

Lorsqu'un chemin a été calculé, il existe de nombreuses techniques pour le rendre soit plus naturel, soit plus proche du chemin optimal. En effet le personnage dans Mammoth représente un humain, nous souhaitons donc éviter des changements de direction trop brusques plus proches d'un robot que d'un humain.

Le nombre de zigzag intervenant lorsque le personnage contourne un mur ou s'oriente dans un couloir peut ainsi être réduit en ajoutant un coût supplémentaire à chaque mouvement non-linéaire comme une rotation.

3.1.7 Limites des types de données réel

Les langages de programmation perdent de la précision lorsqu'ils effectuent des opérations sur des variables de type « double ». Ainsi un « double » copié ne sera pas strictement le même entre les deux variables. Dans Mammoth, on compare les coordonnées des points dans plusieurs méthodes à l'aide des « doubles » et d'un seuil de tolérance, aussi appelé un epsilon. Ainsi les coordonnées des points similaires sont considérées comme égales lors des tests dans les structures conditionnelles.

Cependant, il y a de trop nombreuses méthodes dans Mammoth et un seuil de tolérance qui favorise le bon fonctionnement d'une méthode, nuit à une autre. Ainsi lorsque les coordonnées de points sont considérés comme égales dans certains cas, dans d'autres cas cela pose problème car certains points sont trop proches l'un de l'autre et sont assimilés comme un. Par exemple, la fonction, détectant si les triangles sont voisins compare si les points du côté en commun sont identiques, nécessite un epsilon de 0,001. Au contraire, la fonction, détectant si un triangle est suffisamment grand pour que le personnage puisse y passer, nécessite un epsilon de 0,0001.

Si on diminue le seuil de tolérance on obtient alors le cas inverse. La solution choisie a été de créer une méthode de comparaison des points prenant en paramètre un seuil de tolérance que l'on adapte en fonction de la situation et de la méthode.

3.1.8 Choix de l'heuristique

Il existe différentes manières de calculer cette distance lorsque les déplacements d'effectuent dans un plan 2D, voici les plus courantes :

- 4 directions : distance de Manhattan

$$h(n) = D * (|start.x - goal.x| + |start.y - goal.y|)$$

- 8 directions : distance diagonale (aussi appelée la distance de Chebyshev)

$$h(n) = D * max(|start.x - goal.x|, |start.y - goal.y|)$$

- Environnement ouvert : distance euclidienne

$$h(n) = D * \sqrt{(start.x - goal.x)^2 + (start.y - goal.y)^2}$$

Source : (Patel, 2011)

D est le coût minimum pour le déplacement d'une case à une case adjacente.

La distance euclidienne correspond à la distance à vol d'oiseau entre un point et un autre dans un environnement.

La distance de Manhattan correspond à la distance entre un point et un autre lorsque les cases possèdent quatre voisins.

Ainsi si les déplacements dans toutes les directions sont possibles, la distance euclidienne sera plus appropriée. En revanche si les déplacements sont seulement permis dans 4 directions, il semble plus approprié alors de calculer l'heuristique grâce à la distance de Manhattan.

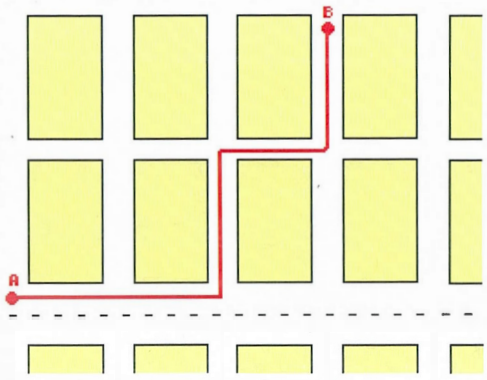

Figure 3.1 Distance de Manhattan (Despland, 2011)

Sur la Figure 3.1, nous pouvons ainsi observer que la longueur du chemin entre le point A et le point B est mesuré par la distance de Manhattan. La ligne droite entre les

deux points correspondant à la distance euclidienne ne convient pas ici car les déplacements s'effectuent seulement dans 4 directions (nord, sud, est, ouest).

Pour notre implémentation, nous utilisons la distance euclidienne puisque Mammoth est composé d'environnements ouverts.

3.2 Solutions conceptuelles

Dans cette partie, nous expliquons les algorithmes, techniques et solutions imaginées et implémentées dans Mammoth.

3.2.1 Étapes pour une planification de chemin dans une triangulation

Suite à nos recherches et nos travaux sur Mammoth, nous avons déterminé que les étapes suivantes étaient nécessaires pour une planification de chemin dans le contexte du jeu et d'une triangulation :

1. Récupère les coordonnées de la position du personnage et de la position cliquée par le joueur (destination)
2. Vérifie si la destination est valide
3. Transforme les coordonnées des deux positions en triangles
4. Triangulation A* : Trouve les triangles par lesquels le personnage va passer entre le triangle de départ et d'arrivée (triangle de départ, triangle de destination)
5. Algorithme funnel modifié : Trouver le chemin à l'intérieur des nœuds sélectionnés (point de départ, point de destination, canal)

Algorithme 3.1 Pseudo-code de la planification de chemin dans des triangles

Nous expliquons en détail dans la section 3.2.8 Package TAStar certains de ces algorithmes.

3.2.2 Refactorisation de la classe Tile

Mammoth possède de nombreux modules et la plupart sont interdépendants. La classe « Tile » représentant un triangle dans le jeu héritait auparavant des attributs de la classe Polygon qui héritait elle-même de StaticObject. Nous souhaitions sérialiser l'ensemble des triangles d'une carte du jeu, cependant cet héritage complexe posait problème dans le processus. En effet, les informations héritées étaient trop nombreuses et trop interconnectées. Dans un souci d'optimisation la vitesse de chargement du jeu, nous

avons rendu la classe « Tile » indépendante du reste afin de pouvoir créer, charger et afficher des triangles plus rapidement.

Pour lire les cartes du jeu, Mammoth a un module lui permettant de lire et d'écrire dans les fichiers XML. Lorsqu'une carte a été triangularisée, l'information des triangles est enregistrée également dans le même fichier que l'information du reste de la carte.

Malheureusement au lancement d'une carte, l'information des triangles prend énormément de temps (plusieurs dizaines de secondes) à être lue dans le fichier XML et à être retranscrite sur la carte. Ainsi, après avoir refactorisé la classe « Tile » afin de supprimer les informations inutiles dues à son héritage, nous avons placé les informations des triangles (position, tailles, voisinage) dans un fichier sérialisable.

Dorénavant, le jeu se lance quelques secondes soit plus rapidement qu'auparavant. La classe « Tile » possède désormais plus de méthodes, car certaines dépendaient de son héritage. Une grande différence vient aussi du fait que les points d'un triangle ont désormais leurs coordonnées en distance absolue plutôt que relative. Ce choix se justifie par une certaine logique dans le placement des triangles sur une carte. En effet les triangles ne dépendent ainsi plus les uns des autres. Le centroïde est aussi indépendant par souci de simplicité. Il a donc fallu adapter moult méthodes pour changer cela.

La classe « Tile » a aussi été adaptée pour être utilisée par d'autres types de quadrillages tels que des carrés ou des hexagones. Ces attributs sont ainsi devenus génériques et cette classe peut être utilisée comme classe mère.

Comme « Tile » héritait auparavant notamment de la classe « Polygon », nous avons dû refactoriser une centaine de classes qui impliquaient les deux classes en question. Grâce à cela de nombreuses lignes de codes ont heureusement perdu en verbosité car étant composées de trop nombreuses conversions en objet « Polygon » ou « Tile ».

Pour finir sur ce point, il a fallu rendre la classe « TileSet » et chacun de ces éléments, sérialisable.

3.2.3 Calcul des triangles voisins

La triangulation est calculée lorsqu'une carte de jeu est créée dans l'éditeur de carte de Mammoth. Cette triangulation propose plusieurs caractéristiques que nous avons modifiées au cours de nos recherches. Nous expliquons ainsi ces caractéristiques et les choix qui ont influencé les changements.

Le calcul des triangles voisins est une particularité de Mammoth. Nous nécessitons ce calcul car il permet de simplifier la tâche aux personnes travaillant sur la partie réseau du jeu et permet de s'affranchir de certains autres calculs dans TRA* notamment.

Le calcul des triangles voisins remplissait les listes de chaque triangle de nombreux triangles superflus. En effet de nombreuses boucles comparant 2 triangles, ajoutaient un triangle voisin lorsque les conditions étaient remplies, aux 2 triangles en question. Malheureusement ces 2 boucles agissaient sur la même liste et ajoutaient donc 2 fois chacun des triangles voisins.

Dans les cas où les cartes étaient boguées cela produisait des connexions visibles sur le graphe présent à la Figure 3.2.

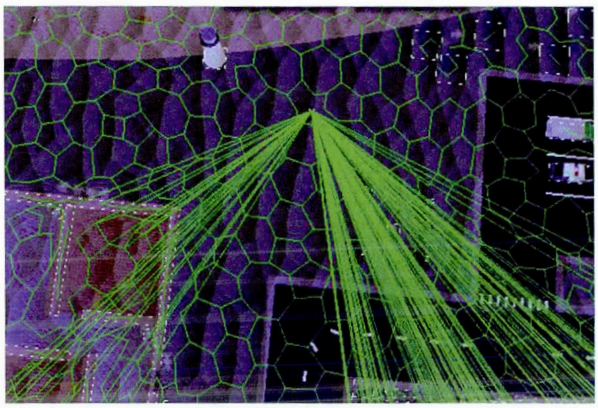

Figure 3.2 Bogue de connexions entre triangles

Dans cette image, chaque ligne verte indique quel triangle est connecté avec quel autre triangle. On peut observer ici que les triangles à l'intérieur de l'obstacle au centre de l'image avaient des connexions partout sur la carte alors qu'ils auraient dû être connectés juste entre eux deux.

Pour détecter si un triangle est voisin, Mammoth compare les côtés des triangles et détecte s'ils sont égaux De cette manière les références des triangles adjacents sont enregistrées dans un triangle.

Nous avons imaginé que pour détecter les triangles voisins non adjacents on pouvait procéder de la même manière en vérifiant si les sommets du triangle que l'on teste sont égaux à ceux d'autres triangles. Cependant, cela produit vraiment de trop nombreux triangles voisins car lorsque l'on compare si des points sont égaux, beaucoup trop d'entre eux sont détectés comme similaires à cause du seuil de tolérance du moteur physique de Mammoth. Ainsi quand les points sont situés à proximité ils sont tous considérés comme appartenant aux mêmes triangles et faussent ainsi la structure des triangles voisins. Faire varier le seuil de tolérance ne résout pas le problème dans ce cas car cela va biaiser la comparaison des points pour d'autres méthodes. Nous avons alors préféré laisser l'ancienne méthode se basant sur les côtés.

Le calcul de la triangulation enregistrait également un nombre trop important de côtés bloquants à cause de l'enregistrement dans les 2 boucles donc nous parlons ci-dessus. Nous avons également corrigé ceci en supprimant un des enregistrements.

3.2.4 Suppression des triangles dans les obstacles

Cette méthode intervient après le calcul de la triangulation et des triangles voisins. Dans une triangulation, des auteurs comme (Demyen et Buro, 2006) préfèrent supprimer les triangles à l'intérieur des obstacles. Sachant que les obstacles et leurs intérieurs ne sont pas accessibles par le personnage, il semble évident que les supprimer ne peut être que bénéfique, ne serait-ce qu'en termes de gain de performances et de mémoire.

Cependant d'un point de vue implémentation et jeu vidéo, cela peut poser certains problèmes. Par exemple, lorsque le joueur clique sur un obstacle, le jeu transforme habituellement les coordonnées de la fenêtre de jeu en coordonnées dans le monde et ensuite en triangle. Sans triangles, cela effectue naturellement une erreur.

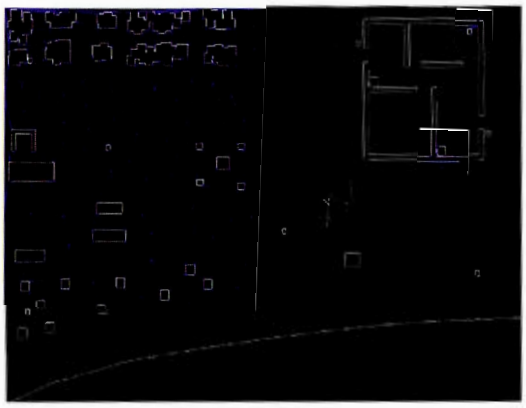

Figure 3.3 Carte sans triangles dans les obstacles

La Figure 3.3 montre le nombre de triangles divisé par 2 dans cette carte lorsque la méthode est activée. La fonction agit de cette manière ; elle cherche un point à l'extérieur d'un obstacle, transforme le point en triangle et utilise tous les triangles adjacents pour se répandre sur la carte de jeu. Ainsi les triangles non adjacents et possédant donc des côtés bloquants ne sont pas inclus.

Auparavant, la fonction se servait des triangles voisins également, mais nous avons jugé cela superflu et avons commenté cette caractéristique.

3.2.5 Suppression des triangles hors de la carte de jeu

Cette méthode intervient après le calcul de la triangulation et des triangles voisins. Il parait évident que les triangles en dehors de la carte de jeu sont inutiles car le personnage ne peut sortir de la carte. Il n'existe rien hors de la carte. Cependant, les algorithmes de triangulation produisent parfois plus de triangles que prévu, cette méthode existe pour corriger ce fait. Auparavant cette méthode traitait seulement un seul triangle. Nous avons adapté cela à tous les triangles hors de la carte. Ils sont ainsi supprimés correctement.

Dans certains cas, lorsque les triangles sont trop grands, ils sont à la fois situés sur la carte et en dehors de celle-ci. Cela pose problème car les zones sans triangles ne sont alors plus accessibles. Nous corrigeons cela en lançant et adaptant les méthodes seulement lorsque nécessaire c'est-à-dire lorsque les tests de nos méthodes sont terminés.

3.2.6 Package Pathfinding

Le jeu Mammoth possède 3 packages dédiés à la planification de chemin. Si ceux-ci n'ont pas changé fondamentalement entre nos recherches, ils ont été adaptés afin d'être plus vite compréhensible par les futurs étudiants développeurs. Ainsi la documentation a été accrue et les noms des classes sont devenus normalisés. En dehors des packages cités ci-dessus, il existe quelques packages dédiés à la gestion de la triangulation. Ils ont été adaptés pour devenir plus génériques. Nous parlons de ce point en section 3.2.2.

Le package « Pathfinding » contient notamment le « Pathfinding Manager » qui permet d'utiliser l'algorithme choisi dans les propriétés de Mammoth. C'est en effet ce fichier qui met en place les attributs et contraintes nécessaires pour chaque algorithme. C'est aussi dans ce package que les threads de la planification de chemin sont initialisés.

Nous avons également centralisé les fonctions permettant de mesurer les métriques (temps pour calculer, longueur du chemin) dans ce package. Comme indiqué auparavant, nous préférons vérifier la validité de la destination avant de calculer le chemin, c'est pourquoi cette fonction se retrouve également au même endroit.

3.2.7 Package AStar

Le package contient les fichiers nécessaires pour faire fonctionner A*, ainsi que deux heuristiques différentes : euclidienne et la « 8 directions ».

Le package contient également les fichiers pour la « mémorisation des raccourcis » (Shortcut Memory, (Despland, 2011)). Cette fonction opère de cette manière : à chaque fois que la planification de chemin complète un "sous-chemin direct" d'un nœud à un autre (après avoir trouvé un chemin valide, quand l'algorithme teste si certaines nœuds peuvent être sautés sans collisions pour aller en diagonales), si le sous-chemin est assez utile (basé sur la taille et la fréquence d'utilisation), il est rajouté à un ensemble de raccourcis.

Si pendant le calcul du chemin, le nœud actuel est à proximité de l'une ou l'autre extrémité d'un raccourci, en plus de regarder le coût des 4 ou 8 directions habituelles (haut, bas, gauche, droite avec ou sans les diagonales), l'algorithme évalue le coût de prendre le raccourci et choisit de le prendre si ça le rapproche assez de son but.

3.2.8 Package TAStar

Le package possédait une classe Funnel pour l'algorithme funnel simple. Cependant celle-ci n'était pas implémentée et n'était pas très utilisable dans le jeu car ne prenant pas en compte le rayon du personnage.

Nous avons implémenté une fonction funnel bien plus rapide et plus claire dans son code : celle de (Mononen, 2010) au lieu de la fonction classique de (Demyen et Buro, 2006). Elle existe seulement à des fins de tests pour voir si la fonction funnel modifiée est proche du chemin le plus court possible dans les triangles sélectionnés.

De la même manière et malgré son inefficacité, nous avons implémenté le parcours par milieux des côtés intérieurs à des fins de test et comparaison.

Nous décrivons le fonctionnement des algorithmes par défaut à l'aide des pseudo-codes suivants.

L'algorithme Triangulation A* est présenté à l'Algorithme 3.2:

Triangulation A* : Trouve les triangles par lesquels le personnage va passer entre le triangle de départ et d'arrivée (triangle de départ, triangle de destination)

```
Initialise chaque triangle comme un nœud.
Initialise le nœud de départ. G = 0, H = distance euclidienne jusqu'à
destination et le nœud n'a donc pas de nœud parent.
Initialise la liste ouverte en ajoutant le nœud de départ.
Initialise la liste fermée à la liste vide.
Boucle infinie
    Si la liste ouverte est vide
        Fin de l'algorithme TA*
    Initialise le nœud parent actuel (NPA) avec le nœud ayant le coût
    F minimum de la liste ouverte
    Ajoute le NPA à la liste fermée
    Si le triangle du NPA est égal au triangle de destination
        Nous avons atteint le but de l'étape 4, fin de l'algorithme
        TA*.
    Pour chaque triangle enfant (TE) parmi les triangles adjacents
    du NPA
        Si la liste fermée contient le TE
            Continue à la prochaine itération
        Assigne au côté d'entrée le côté commun entre le TE et le
        triangle du NPA
        Si la taille du TE n'est pas assez grande pour le rayon du
        masque de collision du personnage
            Continue à la prochaine itération
        Estime et crée le nœud enfant à partir du TE et des coûts G
        et H.
        Ajoute le TE à la liste ouverte
```

Algorithme 3.2 Triangulation A*

Chaque nœud au départ a un coût G, H et F, un triangle correspondant, un côté indiquant par où est entré le personnage et un triangle parent. Le coût G est la distance depuis le départ. Le coût H est la distance euclidienne du côté du triangle actuel jusqu'à la destination. Le coût F est la somme des deux.

La liste ouverte comporte les nœuds à parcourir, la liste fermée comporte les nœuds parcourus.

Lorsqu'on estime et crée le nœud enfant à partir du TE et des coûts G et H, les coûts se basent sur le côté d'entrée.

La fonction funnel modifié correspond après modifications à l'Algorithme 3.3.

Algorithme funnel modifié : Trouver le chemin à l'intérieur des nœuds sélectionnés (point de départ, point de destination, canal)

Récupère dans un « canal » la liste des triangles sélectionnés par TA*.
Si on a un seul triangle
    Ajoute le point de départ et le point d'arrivée au chemin.
    Fin de l'algorithme funnel modifié
Initialise le chemin qui contient seulement le point de départ.
Pour chaque triangle connecté avec un autre dans le canal
    Récupère les côtés communs à deux triangles.
Initialise les 2 premiers points VL et VR avec les 2 points du premier côté de la liste des côtés intérieurs.
Initialise le funnel, le sommet, le point à gauche, et à droite au point de départ.
Pour chaque côté intérieur CI dans la liste des côtés intérieurs
    Récupère les points du CI courant VL2 et VR2
    Si VL2 est égal à VL
        VR = VR2
        Ajoute le point VR au funnel en tant que point ajouté sur le côté gauche du funnel (voir Algorithme 3.4)
    Sinon
        VL = VL2
        Ajoute le point VL au funnel en tant que point ajouté sur le côté droit du funnel (voir Algorithme 3.4)
Ajoute le dernier point de destination au funnel
Récupère le chemin à partir du funnel
Fin de l'algorithme funnel modifié

Algorithme 3.3 Pseudo-code du Funnel Modifié

Le funnel est une structure de données représentant la recherche des points significatifs dans le canal. Ce funnel contient le sommet de celui-ci, un point à gauche et un point à droite du canal, le chemin, le rayon du masque de collision ainsi que l'orientation du sommet (gauche ou droite du funnel). Chaque point est considéré comme un nœud. Dans chaque nœud, il existe une référence au nœud de droite et au nœud de gauche.

Au moment où on récupère les côtés communs à deux triangles, ceux-ci sont placés dans une liste ordonnée suivant l'ordre des triangles (du triangle de départ au triangle de destination).

Lorsque l'on ajoute un point au funnel, l'algorithme permettant cela correspond à l'Algorithme 3.4.

Fonction ajouter un point au funnel (point, orientation, funnel)

Récupère le point comme point testé (PT)
Récupère l'orientation du point. (OP)

Si OP est égal RightTangent, on ajoute le PT sur le côté gauche ci-dessous
Boucle jusqu'à ce que le PT soit ajouté au funnel
Si le sommet est le seul point dans le funnel
Ajoute simplement le PT au côté correspondant
S'il n'y a pas de points sur le côté gauche du funnel
Point 1 = Point à gauche du funnel
Point 2 = Point à droite du point 1 dans le funnel
Dans le cas contraire
Point 1 = Point à droite du point 2 testé dans le funnel
Point 2 = Point à gauche du funnel
Calcule l'angle 1 entre Point 1 et Point 2
Calcule la distance 1 entre Point 1 et Point 2
Calcule la distance 2 entre Point 1 et PT
Calcule l'angle 2 entre le point à gauche du funnel et le PT
Si l'angle 1 est antihoraire à l'angle 2
Ajoute PT à la fin du funnel
Fin de de la fonction
S'il est horaire
Si le point gauche du funnel est égal au sommet
Si la distance 2 est inférieure à distance 1
Ajoute sommet au chemin
Ajoute PT au chemin

L'élément à droite de PT devient
l'élément à droite du sommet
L'élément à gauche de l'élément à droite
du sommet devient PT
Sommet = PT
Ajoute PT à la gauche du funnel
Fin de de la fonction
Sinon,
Ajoute sommet au chemin
Ajoute l'élément à droite sommet au chemin
Supprime l'élément à droite du sommet
Supprime le point le plus à gauche du funnel

Si OP est égal LeftTangent, on ajoute le PT sur le côté droit ci-dessous
(Les opérations sont identiques pour le côté gauche, il faut juste inverser les directions, gauche devient droite, et horaire devient antihoraire et vice versa)

Si le type du point est égal à POINT
Alors ajoute le point directement au chemin sans modification
Fin de l'algorithme

Algorithme 3.4 Pseudo-code de la fonction "ajouter" du funnel modifié

Cet algorithme a été simplifié lors de l'implémentation à des fins de compréhension future et de clarté. La verbosité et les répétitions sont désormais moins importantes dans le code.

Nous avons corrigé des bogues et des omissions afin de le rendre fonctionnel.

Ainsi au lieu des structures conditionnelles « Si le point est ajouté à droite » / « Sinon »/ »Si le point est égal à POINT » utilisées par le précédent utilisateur dans l'algorithme ci-dessus, nous avons mis « Si le point est ajouté à droite » / «Sinon Si le point est ajouté à gauche » / »Sinon Si le point est égal à POINT ». Auparavant dans le cas où un point était ajouté en tant que POINT, il passait dans la condition « Sinon » et dans la condition vérifiant « POINT » ajoutant alors plus de points dans le chemin qu'il ne fallait et provoquant souvent un zigzag dans le parcours.

Un autre bogue consistait en le fait que les segments créant le funnel traversait soudainement le canal au début de l'algorithme. On pouvait observer que les segments prenaient dans un ordre étrange chacun des points des côtés intérieurs. Après constatation, nous avons pu observer que les points du premier côté intérieur n'étaient pas ajoutés correctement comme l'était le reste des côtés. Les ajouter de la même manière a résolu le problème.

La rotation des segments dans cet algorithme était également configurée à horaire quand on ajoutait un point sur le côté gauche et sur le côté droit du funnel. En fait, il s'agit d'une rotation horaire pour le côté gauche et d'une rotation anti-horaire pour le côté droit.

Pour finir, le point de destination est particulier car on ne connait pas son orientation par rapport au reste des points. À la fin du traitement des côtés intérieurs, le point de destination peut se retrouver à l'intérieur funnel où bien à gauche ou à droite de celui-ci. Dans la Figure 3.4, on peut observer que le triangle de destination (le triangle tout en bas de l'image) est plus grand que le reste et le point de destination se trouve donc à l'extrémité inférieure du chemin (ligne verte).

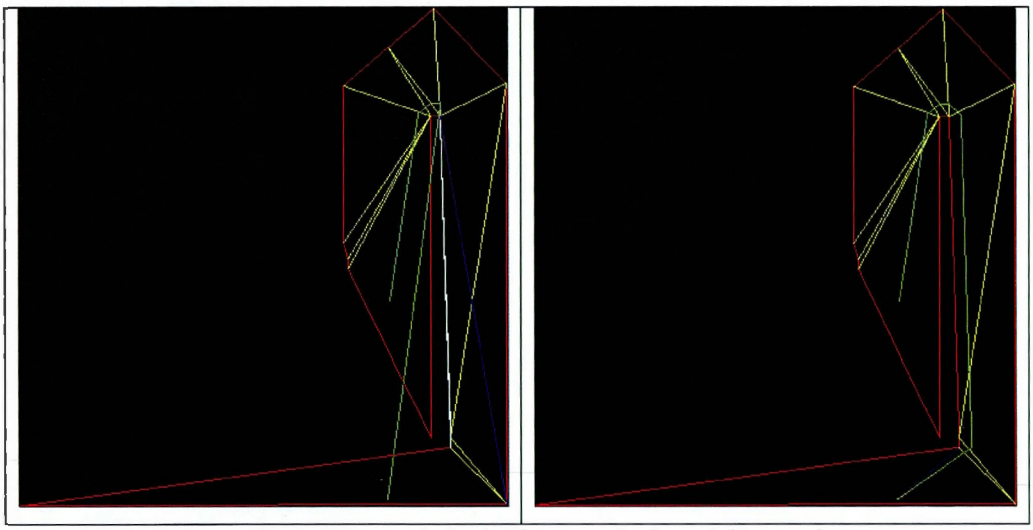

Figure 3.4 Point de destination à gauche des précédents points

Auparavant, l'algorithme ajoutait simplement le dernier point à la fin du chemin et produisait une ligne droite quoiqu'il se passe à la fin. Il traversait du coup le canal directement dans le cas où le point de destination n'était pas juste en dessous du funnel au lieu d'ajouter des points intermédiaires.

Pour corriger cela, nous vérifions de quel côté du segment formé par le sommet et le point gauche du funnel, le point de destination se trouve. S'il est à gauche on l'ajoute comme si c'était un point à gauche. Évidemment dans le cas contraire, on teste avec le sommet et le point droit du funnel.

Dans le cas, où le point de destination se trouve juste en dessous du funnel, on l'ajoute normalement comme auparavant. Cette amélioration permet de corriger le problème montré par les images ci-dessus.

3.2.9 Simplification de la planification de chemin dans un cas très simple

Suite à nos recherches sur TRA*, nous avons pensé à implémenter une petite vérification au début de TA*. En effet, lorsque le triangle de destination appartient aux triangles voisins du triangle départ, le chemin est directement effectué en ligne droite entre les deux. Comme les triangles sont voisins, il n'y a aucun obstacle entre eux et le déplacement peut s'effectuer sans encombre. Ainsi pour les déplacements proches du

point de départ on évite de multiples boucles et structures conditionnelles et on gagne environ 30 ms en temps de réaction.

3.2.10 Vérification de la taille des triangles pour le passage du personnage

Lorsque l'algorithme TA* choisit des triangles à traverser, il doit naturellement prendre en compte le masque de collision du personnage. Il existait dans Mammoth des fonctions permettant cela mais non implémentés dans le code. C'est-à-dire que l'algorithme choisit les triangles directement sans vérifier si le personnage peut les traverser. Dans le cas de grands triangles, cela ne pose pas de problème, mais lorsque le chemin indiqué est inférieur au rayon de collision, le personnage essaye de le traverser et même de le contourner correctement mais finit par entrer dans les murs adjacents car sortants du canal.

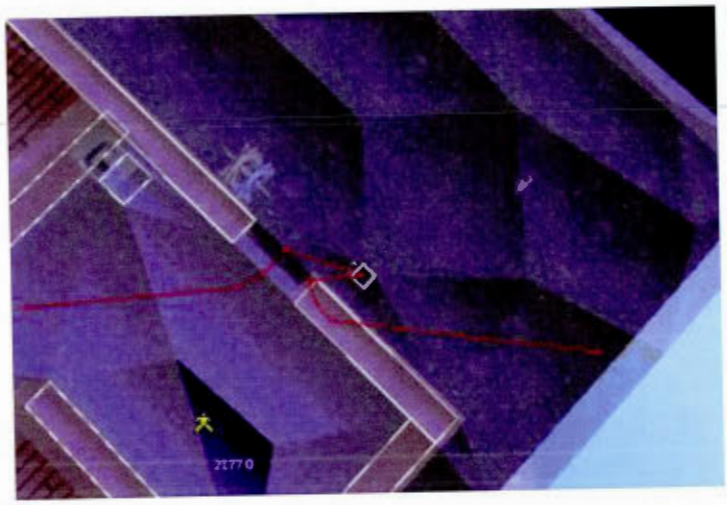

Figure 3.5 Non prise en compte du masque de collision

La Figure 3.5 montre le défaut de la planification de chemin implémenté auparavant : un mauvais choix des triangles pour le canal à cause de non prise en compte du masque de collision.

Lors de la recherche des enfants de la case parente testée, il faut veiller à ce que le masque de collision puisse entrer par un côté du triangle et sortir par un autre. Comme le

masque de collision est un cercle, il doit avoir suffisamment d'espace dans le triangle pour ne pas en sortir ni se retrouver bloqué par le troisième côté du triangle.

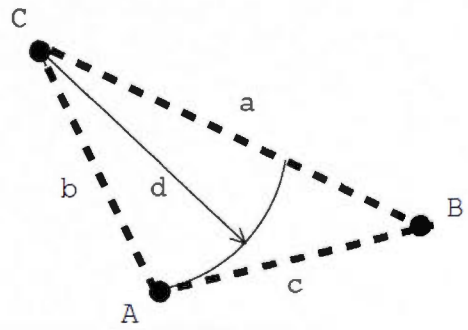

Figure 3.6 Triangle avec rayon maximum (Demyen, 2007)

Sur la Figure 3.6, on considère entrer par le côté « a » et sortir par le côté « b ». On peut observer que le rayon maximum du masque de collision ne doit pas dépasser l'arc de cercle « d ». En effet si le côté « c » est bloquant, le déplacement du personnage s'arrête dans ce côté en cas de dépassement du masque de collision. Si « c » n'est pas bloquant, et que le masque de collision dépasse l'arc de cercle « d », il convient de vérifier avec le triangle adjacent au côté « c » si cela pose problème. De la même manière on observe si le dépassement peut être passé dans les côtés du triangle adjacent. Dépendamment de la taille des triangles, cela peut continuer sur de nombreux triangles adjacents.

Dans le cas où le triangle enfant testé du triangle parent ne convient pas, on prend alors un autre triangle enfant même si le coût est supérieur. Si les trois triangles enfants ne conviennent pas, alors on change de parent, jusqu'à trouver un chemin convenable pour le masque de collision.

La Figure 3.7 montre l'apparence en rouge du chemin lorsque les triangles inférieurs au masque de collision ne sont pas sélectionnés. Ainsi le personnage n'essaye plus de passer entre la poubelle et le mur.

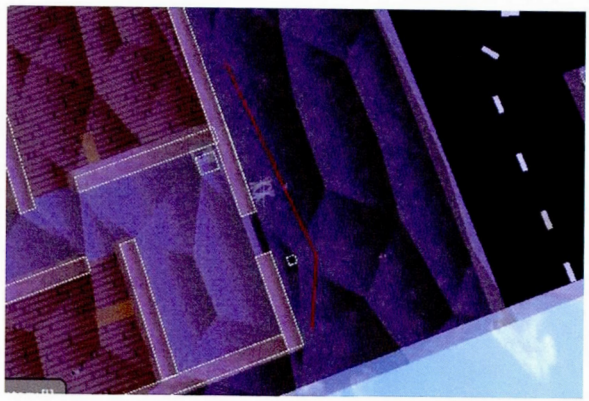

Figure 3.7 Chemin prenant en compte le rayon

3.2.11 Suppression des points supplémentaires d'un chemin

Cette méthode intervient après le calcul du chemin final. Suite à nos tests, nous remarquons que dans certains cas, le chemin, calculé par les algorithmes expliqués précédemment, comporte des points en trop.

Par exemple dans le cas ci-dessous :

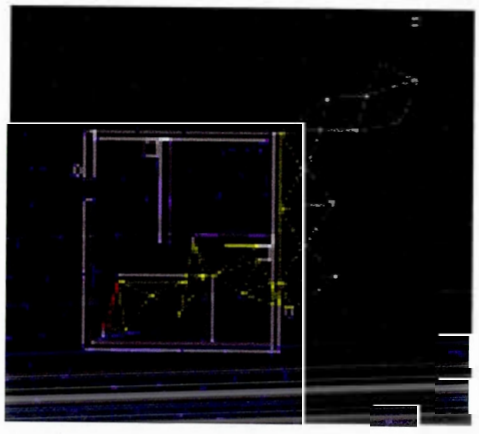

Figure 3.8 Cas particulier du TA*

On peut observer qu'entre le point de départ (triangle vert) et l'entrée du bâtiment, le personnage peut se déplacer en ligne droite directement. Il suit pourtant la série de triangles trouvé par TA*. TA* trouve un tel chemin car il se base sur l'heuristique de la destination, c'est pour cela qu'il s'oriente directement vers le mur du bâtiment. Avec

tellement de triangles, nous ne pouvons pas être certains qu'un chemin en ligne droite aurait pu être trouvé. C'est pour ces raisons que nous imaginons une méthode supprimant les points inutiles. Pour cela la méthode teste les premiers points avec ceux de la fin du chemin et vérifie s'il y a des collisions entre eux. Si c'est le cas, l'algorithme avance et passe aux points intermédiaires. S'il n'y a pas de collision, les points intermédiaires peuvent être supprimés et le personnage peut se déplacer en ligne droite directement entre les points testés.

3.2.12 Détection de collisions

Un système de collision dans un jeu est un système qui permet de rendre bloquant les obstacles aux personnages. Sans celui-ci, les personnages traverseraient librement bâtiments, murs, arbres, étendues d'eau sans aucune distinction. Pour rendre un jeu avec déplacement en 2 ou 3 dimensions intéressant et imposer quelques contraintes dans le principe du jeu, il est donc nécessaire d'avoir un système équivalent. Le plus souvent, le système de collision est inclus dans le moteur physique du jeu.

Il existe deux systèmes de collision dans Mammoth :

- Un pour les carrés

Détecte si la position donnée est bloquante en comparant le masque de collision maximal de l'objet à la position donnée avec la liste des obstacles du monde.

- Un pour les triangles

Détecte si la position donnée est bloquante en comparant la position donnée avec les côtés bloquants du triangle dans lequel la position se trouve.

Auparavant le système de collision pour les triangles opérait de cette manière :

1) Récupère les objets situés dans le triangle, et vérifie si la position donnée entrecroise les objets du triangle.

2) Récupère les objets des triangles voisins de la position donnée, et vérifie si la position donnée entrecroise les objets dans ces triangles.

Nous avons modifié les 2 méthodes ci-dessus car nous pensons qu'elles ne sont pas adaptées à un environnement triangularisé car les triangles sont déjà contraints par les

obstacles lors de la triangulation. Nous préférons utiliser l'information de cette contrainte dans le système de collision.

En général, quel que soit le type de structure abstraite, il y a deux manières de tester les collisions : soit on teste si le côté de la case (triangle, carré) est bloquant, soit on teste la projection du masque de collision de personnage sur le masque de collision de l'obstacle. La deuxième solution est cependant bien plus coûteuse en termes de temps de calcul et de code.

C'est pour cette raison que nous avons inclus la variable « blocking » (bloquant en anglais) dans la classe « Edge ». La classe « Edge » correspond à un côté d'un « Tile ». Ainsi lorsqu'un « tileset » est calculé, les informations concernant la collision provoquée par un « edge » sont enregistrées dans le fichier de la carte du jeu. On peut ainsi directement savoir si le côté est bloquant ou non à l'aide de cette variable au lieu de faire des tests complexes comme auparavant. Même en terme de verbosité cette solution est plus pratique puisqu'on effectue une simple structure conditionnelle pour tester « Edge.isblocking() ».

Lorsque la fonction qui supprime les triangles dans les obstacles est activée, le nouveau système de collision implémenté ne fonctionne pas. Pour les cartes dont la triangulation a été calculée avec cette fonction activée, nous renvoyons une collision lorsque le personnage n'est pas sur un triangle.

3.2.13 Triangulation Reduction A*

Triangulation Reduction A* (TRA*) est donc une variante de TA* mais contenant (Erdtman et Fylling, 2008) plus d'informations pour chaque triangle. Il s'agit d'un algorithme relativement récent et possédant peu de références notamment au niveau des codes sources et autres pseudo-codes. Une des difficultés a donc été d'implémenter cet algorithme en rassemblant les quelques sources mentionnés dans l'état de l'art.

Cas spéciaux

L'une des particularités de TRA* est ces cas spéciaux qui permettent de donner un résultat très rapidement lorsque le triangle contenant le départ et le triangle contenant l'arrivée obéissent à des conditions spécifiques (Algorithme 3.5).

```
Récupère les triangles StartTriangle et EndTriangle dans lesquels sont situés le point de
départ et le point de destination
Si StartTriangle = EndTriangle
        Ajoute EndTriangle au canal
        Retourne canal et fin de l'algorithme TRA*
Si le type de StartTriangle ou de l'EndTriangle est égal à 0
        Pathfinding impossible
Si le StartTriangle et l'EndTriangle ne sont pas dans la même zone
        Pathfinding impossible
Sinon si StartTriangle et EndTriangle sont membres du même arbre de type 1
        Effectue un Parcours de l'arbre entre StartTriangle et EndTriangle
Sinon si StartTriangle ou EndTriangle est la racine de l'arbre contenant l'autre triangle.
        Effectue un chemin entre à l'intérieur de l'arbre jusqu'à sa racine.
Sinon si StartTriangle et EndTriangle sont sur la même boucle de type 2.
        La boucle est traversée dans les 2 sens. Le chemin le plus court est choisi.
Sinon si StartTriangle et EndTriangle sont dans le même couloir.
        Un chemin est trouvé en marchant dans le couloir et un autre est trouvé en
passant par le graphique de type 3 connecté au couloir. Le chemin le plus court est
choisi.
Sinon
        Utilise les points de décision, les extrémités des corridors ou les points de
        décision des racines des arbres pour se déplacer entre le point de départ et de
        destination.
```

Algorithme 3.5 Cas spéciaux TRA*

Informations supplémentaires

Chaque triangle possède donc des informations supplémentaires par rapport aux triangles utilisés par TA*. Ces informations sont calculées lors de la triangulation.

- Niveau : déjà expliqué dans l'état de l'art (1.5.2 Triangulation Reduction A*), c'est l'information principale qu'il faut vérifier avant le reste. Détermine ce que va donner un triangle dans un graphe.
- Composant connecté : correspond à un graphe inter-connecté, à un sous-graphe. Il s'agit d'une zone où tous les triangles sont connectés. Il peut y avoir plusieurs zones dans une carte. Par exemple, un graphe peut être à l'intérieur d'un bâtiment fermé et un autre peut exister pour l'environnement extérieur. Ainsi on détermine pour chaque triangle, à quelle « zone » il appartient grâce à un nombre.
- Structures adjacentes : Il s'agit des triangles voisins déjà inclus auparavant dans la structure de chaque triangle. Ils sont maintenant utilisés pour cet algorithme de planification de chemin. En plus de cela, il est indiqué pour certains niveaux de triangles, les racines des arbres, les points de décisions et les fins des couloirs.

- Points d'étranglement : correspond au diamètre du plus grand objet qui peut entrer dans un triangle. C'est une information non utilisée pour les triangles de type 0, pour les triangles de type 1 correspondant à des arbres sans racine et les boucles de type 2.
- Largeur des triangles : Pour chaque triangle, il faut garder la largeur minimum par laquelle le masque de collision du personnage peut passer. Ainsi on peut utiliser une structure conditionnelle afin de vérifier directement si tel ou tel triangle est approprié pour le canal du chemin.
- Distance minimum parcourue : Il s'agit de la distance entre chacun des triangles voisins et le triangle racine dans un arbre avec racine. Pour les autres triangles, il s'agit de leur distance jusqu'aux points de décision.

Grâce à ces spécificités, TRA* permet d'éviter de gaspiller des ressources lorsqu'il détecte directement qu'un chemin est impossible au lieu de le calculer comme les autres algorithmes. Lorsque le chemin est possible et que les conditions satisfont un cas spécial, l'algorithme sait alors le traverser plus rapidement.

Pour résumer cette partie, nous avons rendu facile à explorer et modifier les modules de la planification de chemin dans Mammoth. De nombreux bogues ont été résolus et il est désormais possible de paramétrer et de changer d'algorithme de planification de chemin très facilement dans Mammoth. Les algorithmes présents ont été améliorés et TRA* a été implémenté. Les méthodes traitant les triangles ont également été mises à jour pour correspondre aux autres méthodes actuelles de Mammoth.

Nous avons ainsi vu les fonctions et méthodes implémentées afin de rendre la planification de chemin possible dans Mammoth. Nous expliquons maintenant les expériences effectuées afin de déterminer quelles sont les méthodes les plus efficaces.

CHAPITRE IV

EXPÉRIMENTATION, CONCLUSIONS, LIMITES ET PERSPECTIVES

Dans ce chapitre, nous exposons le fruit des changements effectués et les résultats produits à des fins de tests. Nous souhaitons ainsi déterminer quel algorithme est le plus efficace dans quel cas.

4.1 Configuration

Nous avons effectué les tests sur l'ordinateur suivant:

PC de référence: processeur Intel Core i7 2617M CPU (4 cœurs) @ 1,50 Ghz, 8 Go de RAM, Java Runtime Environment 1.6, Windows 7

4.2 Justification des métriques utilisées

Dans cette partie, nous justifions le choix et l'utilisation des métriques. L'intérêt principal de ces mesures réside dans la possibilité de comparer les différentes techniques de planification de chemin. Voici donc les métriques utilisées :

- Le **nombre total de triangles** permet de savoir le nombre de triangles sur la carte. Ainsi on peut observer si les performances sont accrues avec un grand ou un petit nombre de triangles.
- La **distance euclidienne** permet de savoir la distance réelle entre le point de départ et la destination.

- La **longueur du chemin** permet de comparer la différence entre le chemin emprunté en tenant compte des obstacles et la distance réelle.
- Le **temps pour calculer le chemin** permet de vérifier que le temps de calcul ne nuit pas trop aux performances et permet de comparer le résultat de l'algorithme en question aux autres.
- Le **nombre de nœuds utilisés** permet de comparer les différents chemins choisis par l'algorithme lorsque la destination est sensiblement la même. Il s'agit du nombre de triangles empruntés par le personnage.
- Le **nombre de points** dans le chemin est le résultat de l'algorithme traversant les triangles.

Pour chacune de ses métriques nous effectuons ensuite la moyenne.

4.3 Tâches exécutées

Nous testons dans différentes situations la planification de chemin dans Mammoth afin de déterminer quel algorithme est le plus approprié dans chacune des situations. Les tests sont effectués sur le client simple de Mammoth. Les autres personnages et leur intelligence artificielle sont désactivés ou non présents.

Pour cela, nous avons conçu un module de tests spécifique. Il se lance à l'aide du bouton « Launch Tests » dans la fenêtre « PathFinding » dans Mammoth.

Lorsque les tests sont lancés, des points de départs et des points de destination sont sélectionnés aléatoirement dans les limites de la carte de jeu. Ensuite la planification de chemin est effectuée entre les points sélectionnés. À la fin de la recherche, on récupère les métriques et résultats de celle-ci.

On effectue **les tests 10 000 fois** et on enregistre les résultats dans un fichier CSV afin de pouvoir les consulter et analyser à tout moment. Les distances sont exprimées dans l'unité de Mammoth. Comme les déplacements s'effectuent sur un plan, les distances se calculent en fonction des coordonnées du plan.

Comme l'algorithme funnel simple ne permet pas au personnage de se rendre à la destination aisément, nous avons seulement effectué des tests sur les méthodes funnel modifiées et celles traversant les triangles par le milieu des côtés.

Les tests de TRA* sont effectués avec l'algorithme funnel modifié.

Les résultats dépendent naturellement de nombreux facteurs :

- l'efficacité des algorithmes;
- le nombre de triangles sur la carte;
- du fait que les triangles soient inclus ou non dans les obstacles.

Nos tests ont été effectués sur un ordinateur assez puissant pour Mammoth. Nous avons lancé des méthodes n'entrant pas en conflit. Le nombre de triangles est celui par défaut avec les triangles inclus dans les obstacles.

La **granularité de A*** est la valeur de la tolérance accordée à l'algorithme A* pour passer dans les passages étroits. Nous avons utilisé la valeur par défaut configuré par les précédents développeurs de l'algorithme c'est-à-dire 0.5.

Le **rayon du masque de collision** correspond au rayon du cercle dans lequel le personnage va rentrer en collision avec d'autres éléments. Nous avons utilisé la valeur par défaut configuré par les précédents développeurs de l'algorithme c'est-à-dire 0.3.

TA* + modified correspond à l'utilisation de Triangulation A* conjointement avec l'algorithme du funnel modifié.

TA* + midpoints correspond à l'utilisation de Triangulation A* conjointement avec l'algorithme de parcours par le milieu des côtés intérieurs.

Temps écoulé total correspond au temps nécessaire total pour effectuer les 10 000 tests.

4.4 Résultats

4.4.1 Test 1 : Town 20-2-Triangles

La carte « Town20-2 » (Figure 4.1) correspond à un environnement ouvert avec de nombreux arbres d'un côté et un bâtiment ouvert de l'autre. Il s'agit de la carte par défaut de Mammoth. La triangulation de cette carte comporte 2031 triangles.

Le Tableau 4.1 présente les valeurs correspondantes à la moyenne pondérée par le nombre de tests:

	Temps de calcul (ms)	Longueur	Distance euclidienne	Nombre nœuds	Nombre points	Temps écoulé total (s)
A*	38	11.62	11.59	/	2.07	506
TA*/modified	26	15.78	11.61	19.8025	10.54	272
TA*/midpoints	28	14.37	11.54	19.2508	19.53	286
TRA*	30	12.81	11.66	19.7734	10.48	310

Tableau 4.1 Tests sur la carte Town20-2

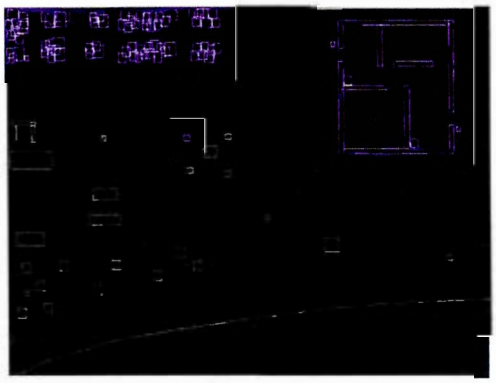

Figure 4.1 Carte « Town20-2.xml »

4.4.2 Test 2 : Corridor2

La carte « Corridor2 » (Figure 4.2) correspond à une carte de test avec de nombreux obstacles placés aléatoirement ainsi que deux corridors et une impasse. La triangulation de cette carte comporte 358 triangles.

Le Tableau 4.2 présente les valeurs correspondantes à la moyenne pondérée par le nombre de tests:

	Temps de calcul (ms)	Longueur	Distance euclidienne	Nombre nœuds	Nombre points	Temps écoulé total (s)
A*	201	11.19	10.47	/	3.89	2058
TA*/modified	1.2	13.78	10.44	19.31	9.93	16
TA*/midpoints	1	21.44	10.34	19.21	19.35	13
TRA*	1.3	19.33	10.55	24.40	12.13	16

Tableau 4.2 Tests sur la carte Corridor2

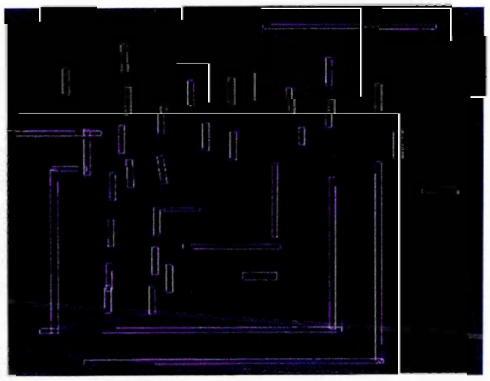

Figure 4.2 Carte « Corridor2.xml »

4.4.3 Test 3 : TwoSquares

La carte « TwoSquares » (Figure 4.3) correspond à un environnement ouvert avec 2 obstacles principaux au centre. La triangulation de cette carte comporte 405 triangles.

Le Tableau 4.3 présente les valeurs correspondantes à la moyenne pondérée par le nombre de tests:

	Temps de calcul (ms)	Longueur	Distance euclidienne	Nombre nœuds	Nombre points	Temps écoulé total (s)
A*	106	9.01	8.95	/	2.43	1073
TA*/modified	2	8.48	8.12	14.96	8.84	27
TA*/midpoints	3	10.89	8.07	14.73	15.12	45
TRA*	3	9.12	8.15	15.88	9.60	37

Tableau 4.3 Tests sur la carte TwoSquares

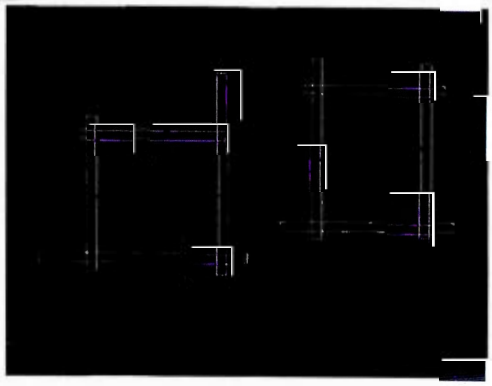

Figure 4.3 Carte « TwoSquares.xml »

4.4.4 Test 4: Corridor-Ring-with-Deadend

La carte « Corridor-Ring-with-Deadend » (Figure 4.4) correspond à un environnement minuscule avec 2 obstacles. L'intérêt est dans le fait que les triangles sont très grands et correspondent aux cas spéciaux de TRA*. L'intérêt est de démontrer que TRA* fonctionne plus rapidement dans les cas les plus limités puisque correspondant au cas spéciaux. La triangulation de cette carte comporte 16 triangles.

Le Tableau 4.4 présente les valeurs correspondantes à la moyenne pondérée par le nombre de tests:

	Temps de calcul (ms)	Longueur	Distance euclidienne	Nombre nœuds	Nombre points	Temps écoulé total (s)
A*	3.1	2.70	2.70	/	2.00	36
TA*/modified	0.0872	2.80	2.69	2.64	2.21	3
TA*/midpoints	0.0804	3.48	2.69	2.65	3.60	4
TRA*	0.0841	3.11	2.67	3.86	2.41	4

Tableau 4.4 Tests sur la carte Corridor-Ring-with-Deadend

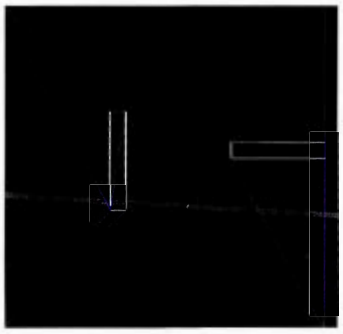

Figure 4.4 Carte « Corridor-Ring-with-Deadend.xml »

4.5 Analyse des résultats

Nous expliquons les résultats pour chacun des algorithmes.

4.5.1 A*

Suite à nos expérimentations, nous pouvons observer que l'algorithme A* original est le plus lent dans Mammoth. C'est en effet dû à son implémentation. Comme précisé auparavant, A* (dans Mammoth seulement) génère une grille de position à chaque fois qu'il doit trouver un chemin. Ainsi il doit tout calculer à chaque planification de chemin, prenant un temps plus grand que les autres algorithmes pour exécuter cette tâche. De plus l'algorithme a été implémenté en 2007 et Mammoth a subi de très nombreuses modifications en de si nombreuses années. L'algorithme A* de Mammoth n'a pas été adapté depuis et parvient désormais rarement à trouver le chemin même lors d'une utilisation traditionnelle. L'algorithme met trop de temps se déplacer sur la carte et le réglage de la granularité lui empêche d'accéder à certaines zones. La longueur des chemins et la distance euclidienne ont des valeurs si proches car dans la plupart des cas, A* fonctionne seulement avec des chemins contenant 2 points : Départ et arrivée. Ainsi de nombreuses valeurs sont communes entre la longueur du chemin calculé et la distance euclidienne, rendant la moyenne similaire. Le reste des cas, A* original met trop de temps à le calculer ou bien se retrouve coincé dans un obstacle et est alors redémarré avec de nouveaux points plus simples.

4.5.2 TA* + modified

Cet algorithme nous sert d'algorithme de référence pour les autres. C'est celui qui fonctionne de la même manière dans tous les cas et qui propose un bon compromis entre vitesse de réaction et qualité du chemin trouvé.

4.5.3 TA* + midpoints

Comme prévu, pour TA* utilisant le milieu des côtés intérieurs, on observe une augmentation de la longueur par rapport aux autres techniques. Cette technique de parcours est donc loin d'être optimale mais est l'une des plus simples à implémenter.

4.5.4 TRA*

Les résultats produits par TRA* ne semblent pas différents de ceux de TA*. Comme dans un environnement ouvert, la plupart des triangles sont des triangles de niveau 3 ou des triangles de niveau 2 connectant des triangles de niveau 3, TRA* n'utilise pas les cas spéciaux. Ces cas spéciaux lorsque utilisés permettent de calculer plus rapidement les chemins afin de donner une réponse plus rapide. Dans une carte normale de Mammoth, les triangles sont de niveau 3 en grande majorité. Lors de nos tests dans une carte plus petite, Corridor Ring with DeadEnd, la différence de rapidité n'est pas flagrante entre TRA* et TA* malgré le fait que dans ce cas-là, seulement des cas spéciaux se présentent.

4.5.5 Synthèse

Grâce aux résultats obtenus, nous pouvons ainsi conclure sur les techniques les plus appropriées au contexte de l'environnement de recherche Mammoth. Pour des environnements ouverts, TA* ainsi que l'algorithme funnel modifié sont les plus adaptés à Mammoth. TRA* ne semble pas adapté à un jeu vidéo. Il fonctionne correctement dans les petits environnements d'après (Demyen, 2007) mais les jeux vidéo en comprennent peu car cela manque d'intérêt pour un joueur. Lorsque des tests sont effectués sur une Nintendo DS, une console portable avec des capacités réduites et donc des environnements réduits, avec TRA*, l'algorithme est alors trop volumineux en mémoire à cause des nombreuses informations contenues pour chaque triangle et se révèle inefficace selon (Erdtman et Fylling, 2008).

4.6 Limites

4.6.1 Taille des triangles

Au niveau de la taille des triangles dans une triangulation, nos quelques tests à ce sujet nous permettent d'émettre les hypothèses suivantes.

Lorsque la triangulation est composée de petits triangles, le nombre de triangles pouvant être éliminés, car étant trop petit pour le masque de collision du personnage, est plus important. Cependant les chemins trouvés sont un peu plus détaillés.

En revanche, pour une triangulation composée de grands triangles, des triangles très minces sont placés à côté des obstacles. Lorsque la vérification de la taille des triangles est faite, ses triangles sont occultés car très minces rendant difficiles le passage et l'accès par ceux-ci. Les chemins sont plus composés de lignes droites dans ce contexte.

4.7 Perspectives

Dans cette section, nous expliquons les évolutions possibles des algorithmes dans Mammoth.

4.7.1 Différents types d'environnements

Au niveau des types d'environnements, une grille de carrés, rectangles ou encore, d'hexagones pourraient être implémentées par la suite. Cela permettrait de nouvelles comparaisons. Un générateur de navigation mesh pourrait être incorporé également. Il existe un système de points de navigation dans Mammoth mais un tel système est limité pour les déplacements comme indiqué dans l'état de l'art.

Dans l'éditeur de carte et dans le code de Mammoth, certaines structures et classes sont déjà prêtes pour accueillir de nouveaux environnements.

4.7.2 Nouveaux algorithmes

Des algorithmes de déplacement dans des hexagones pourraient être testés. Également les déplacements dans des groupes de nœuds pourraient être envisagés. Certains des algorithmes expliqués dans l'état de l'art peuvent être essayés alors.

4.7.3 Nouvelles conditions

Si des événements, changeant les conditions de déplacement, se produisaient dans Mammoth, il faudrait alors un algorithme s'adaptant en temps réel aux nouvelles conditions. Cela nécessiterait plus de ressources sur chacun des ordinateurs.

Des tests peuvent être effectués au niveau des déplacements de groupe. Les algorithmes changent alors. Un passage étroit où un seul personnage peut passer, pose problème pour un groupe car chacun des membres doit alors attendre son tour pour passer. Il est alors préférable dans certaines situations d'emprunter un chemin plus long si celui-ci est plus large.

CONCLUSION

L'objectif de notre recherche était de rendre fiable la planification de chemin dans Mammoth. Travailler dans le contexte d'un jeu vidéo, implique un temps de réaction rapide de la part de l'algorithme. Les solutions ne doivent pas forcément être optimales mais proposer un résultat convenable par rapport à l'ordre du joueur.

Pour réaliser ce mémoire, nous avons d'abord rassemblé le plus de publications concernant le sujet. La planification de chemin étant un domaine relativement récent, il existe peu de livres consacrés à ce sujet. De plus, les techniques créées par l'industrie vidéo ludique sont souvent conservées dans les entreprises car il s'agit d'un milieu très concurrentiel. Heureusement, des thèses, articles et autres papiers sont disponibles à ce sujet en ligne. La deuxième étape consiste à « inventorier » l'existant, cela a donné naissance à l'état de l'art présent.

Nous avons ensuite présenté l'existant dans Mammoth. Les questions qui se sont posées ont ensuite été décrites. Les chapitres de l'implémentation et des tests indiquent les travaux effectués concrètement dans l'environnement de recherche. Enfin, nous avons indiqué quelles sont les évolutions possibles pour la planification de chemin dans ce jeu.

Pour conclure, la planification de chemin TA* fonctionne désormais correctement dans le projet Mammoth dans un environnement triangularisé. Nos recherches dans ce domaine nous ont permis de découvrir et d'implémenter des techniques adaptées à ce jeu.

La planification de chemin dans Mammoth est devenue standard et centralisé. De nombreux outils ont été créés afin de déboguer plus facilement. Il est désormais possible de paramétrer les différents algorithmes disponibles à l'aide du fichier de propriétés propre à Mammoth.

Le domaine de la planification de chemin est très vaste. Il reste de nombreuses perspectives d'évolution comme citées auparavant notamment concernant les techniques de planification de chemin dans les autres types de structure abstraite.

BIBLIOGRAPHIE

Belghith, K., F. Kabanza, L. Hartman et R. Nkambou. 2006. «Anytime dynamic path-planning with flexible probabilistic roadmaps». In *ICRA 2006. Proceedings 2006 IEEE International Conference on Robots and Automation*, p. 2372-2377: IEEE.

Berliner, H. 1979. «The B* tree search algorithm: A best-first proof procedure* 1». *Artificial Intelligence*, vol. 12, no 1, p. 23-40.

Booth, Michael. 2009. «The AI Systems of Left 4 Dead». In *Artificial Intelligence and Interactive Digital Entertainment Conference* (Stanford). Stanford. En ligne. <http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf>.

Botea, A., M. Müller et J. Schaeffer. 2004. «Near optimal hierarchical path-finding». *Journal of game development*, vol. 1, no 1, p. 7–28.

Boulic, R. 2010. *Motion in Games: Third International Conference, MIG 2010*, Proceedings. Utrecht, the Netherlands: Springer-Verlag New York Inc.

Bulitko, Vadim, et Greg Lee. 2006. «Learning in real-time search: a unifying framework». *J. Artif. Int. Res.*, vol. 25, no 1, p. 119-157.

Chew, Paul. 2007. «Voronoi Diagram / Delaunay Triangulation ». En ligne. <http://www.cs.cornell.edu/home/chew/Delaunay.html>. Consulté le 19 mai 2011.

Cormen, T.H. 2001. *Introduction to algorithms*. Cambridge, Mass.: The MIT press, 1312 p.

Dechter, Rina, et Judea Pearl. 1985. «Generalized best-first search strategies and the optimality of A*». *Journal of the ACM*, vol. 32, no 3, p. 505-536.

Demyen, D., et M. Buro. 2006. «Efficient triangulation-based pathfinding». In *Proceedings of the 21st national conference on Artificial intelligence*, p. 942-947 AAAI Press.

Demyen, Douglas Jon. 2007. «Efficient Triangulation-Based Pathfinding». Boston, Thesis University of Alberta. En ligne. <http://skatgame.net/mburo/ps/thesis_demyen_2006.pdf>.

Despland, Joachim. 2011. «Joachim Despland website». En ligne. <http://www.joachimdespland.com/mammoth.html>. Consulté le 30 janvier 2012.

Dijkstra, E. W. 1959. «A note on two problems in connexion with graphs». *Numerische Mathematik*, vol. 1, no 1, p. 269-271.

Erdtman, S., et J. Fylling. 2008. «Pathfinding with Hard Constraints: Mobile Systems and Real Time Strategy Games Combined». Blekinge Institute of Technology.

Hart, P. E., N. J. Nilsson et B. Raphael. 1968. «A Formal Basis for the Heuristic Determination of Minimum Cost Paths». *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no 2, p. 100-107.

Kavraki, L.E., P. Svestka, J.C. Latombe et M.H. Overmars. 1996. «Probabilistic roadmaps for path planning in high-dimensional configuration spaces». *Robotics and Automation, IEEE Transactions on*, vol. 12, no 4, p. 566-580.

Koch, U. 2011. «Applying graph partitioning to hierarchical pathfinding in computer games». Institut fur Mathematik und Informatik, Universitat Leipzig.

LaValle, S.M. 2006. *Planning algorithms*: Cambridge University Press 842 p.

Lee, D.T., et F.P. Preparata. 1984. «Euclidean shortest paths in the presence of rectilinear barriers». *Networks*, vol. 14, no 3, p. 393-410.

Mononen, Mikko (2010). Simple Stupid Funnel Algorithm. Digesting Duck, Blog about AI and prototyping En ligne. <http://digestingduck.blogspot.com/2010/03/simple-stupid-funnel-algorithm.html>. Consulté le 14 mars 2011.

Nec. 2009. «Zero Game - Trouver son chemin (Path finding)». En ligne. <http://nec3.free.fr/Joomla/index.php?option=com_content&view=article&id=3:trouver-son-chemin-path-finding&catid=3:intelligence-artificielle&Itemid=1>. Consulté le 23 août 2011.

Patel, Amit. 2011. «Amit's A* Pages». En ligne. <http://theory.stanford.edu/~amitp/GameProgramming>. Consulté le 31 janvier 2012.

Pinter, Marco. 2001. «Toward More Realistic Pathfinding». Gamasutra. En ligne. <http://www.gamasutra.com/view/feature/3096/toward_more_realistic_pathfinding.php>. Consulté le 11 mai 2011.

Russell, S.J., et P. Norvig. 2010. *Artificial intelligence: a modern approach*: Prentice hall, 1152 p.

Shewchuk, Jonathan Richard. 2005. «Triangle A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator. ». En ligne. <http://www.cs.cmu.edu/~quake/triangle.html>. Consulté le 19 mai 2011.

Stentz, A. 1995. «The focussed D* algorithm for real-time replanning». In *Proceedings of the 14th international joint conference on Artificial intelligence*, p. 1652-1659: Citeseer.

Sturtevant, N., et M. Buro. 2005. «Partial pathfinding using map abstraction and refinement». In *Proceedings of the 20th national conference on Artificial Intelligence*, p. 1392-1397: AAAI Press.

Tozour, Paul (2008). Fixing Pathfinding Once and For All. Game/AI. 2011 En ligne. <http://www.ai-blog.net/archives/000152.html>. Consulté le 23 mai 2011.

Ufkes, Alex. 2010. «World Representation and Path Planning Using the A* Algorithm ». En ligne. <http://www.computerrobotvision.org/2010/slam_camp/ufkes_PathfindingTutorial.pdf>. Consulté le 10 septembre 2011.