REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mohamed Khider – BISKRA
Faculté des Sciences Exactes, des Sciences de la Nature et de la Vie
Département d'informatique

N° d'ordre : IVA08/M2/2021

Mémoire

Présenté pour obtenir le diplôme de master académique en

Informatique

Parcours : **Image et Vie Artificielle (IVA)**

Planification de chemin en temps réel basé sur RRT*

Par :

ZENDAGUI RAFIK MOUNTASSER

Soutenu le 27/06/2022 devant le jury composé de :

Babahenini Med Chaouki	Prof	Président
BOUCETTA Mebarek	MAA	Rapporteur
Benameur Sabrina	MCB	Examinatrice

Année universitaire 2021-2022

Remerciement

Avant tout, nous remercions ALLAH, le Tout-Puissant qui m'a donné l'envie et la force pour mener a terminé ce travail.

Je tiens à exprimer mon vif remerciement a ***Mr Boucette Mebark*** *mon encadreur pour son aide, ses conseils et ses encouragements et sa patience.*

Mes remerciements vont également aux membres du jury d'avoir accepté d'évaluer mon travail.

Dédicace

Mes chers parents mon père Mourad et ma mère Que Dieu ait pitié d'elle Pour leur patience ; leur amour ; soutien et leurs encouragements.

Mes frères : Amine, Achraf , Karim , Islem

Mes amies de mon parcours : L Taha , T Abdelaziz, S Mohammed , Moncef , DBadrdine , Med trad, L Haroun.

Table de matière

Table de figure

Résumé

Il est nécessaire pour un robot mobile d'être capable de planifier efficacement un chemin de sa position de départ, ou actuelle, à un objectif souhaité. C'est une tâche triviale lorsque l'environnement est statique. Cependant, l'environnement opérationnel du robot est rarement statique, et il comporte souvent de nombreux obstacles mobiles. Le robot peut rencontrer un, ou plusieurs, de ces obstacles inconnus et imprévisibles. Obstacles mobiles inconnus et imprévisibles. Le robot devra décider comment procéder lorsque l'un de ces obstacles obstrue sa trajectoire. Une méthode de planification dynamique utilisant RRT* est présentée. Le robot va modifiera son plan actuel lorsqu'un obstacle inconnu, se déplaçant au hasard, obstruera le chemin. Inconnu et aléatoire. Divers résultats expérimentaux montrent l'efficacité de la méthode proposée.

Abstract

It is necessary for a mobile robot to be able to efficiently plan a path from its starting, or current, location to a desired goal location. This is a trivial task when the environment is static. However, the operational environment of the robot is rarely static, and it often has many moving obstacles. The robot may encounter one, or many, of these unknown and unpredictable moving obstacles. The robot will need to decide how to proceed when one of these obstacles is obstructing its path. A method of dynamic planning using RRT* is presented. The robot will modify its current plan when an unknown random moving obstacle obstructs the path. Various experimental results show the effectiveness of the proposed method.

التلخيص

من الضروري أن يكون الروبوت المتحرك قادرًا على التخطيط الفعال للمسار من موقع البداية أو الوضع الحالي إلى الهدف المنشود. هذه مهمة تافهة عندما تكون البيئة ثابتة. ومع ذلك ، نادرًا ما تكون بيئة تشغيل الروبوت ثابتة ، وغالبًا ما يكون بها العديد من العوائق المتحركة. قد يواجه الروبوت واحدًا أو أكثر من هذه العوائق غير المعروفة وغير المتوقعة. عوائق متحركة غير معروفة وغير متوقعة. سيتعين على الروبوت أن يقرر كيفية المضي قدمًا عندما يعيق أحد هذه العوائق مساره. تم تقديم طريقة تخطيط ديناميكي باستخدام RRT *. سيغير الروبوت خطته الحالية عندما يعيق عائق غير معروف يتحرك بشكل عشوائي المسار. غير معروف وعشوائي يعيق المسار. النتائج التجريبية المختلفة تظهر فعالية الطريقة المقترحة.

Introduction général

Introduction général

Les mondes virtuels sont aujourd'hui utilisés dans de nombreux domaines applicatifs. Ces mondes virtuels sont généralement peuplés à l'aide d'agents autonomes qui rendent ces environnements plus vivants. Dans le cadre de cette thèse nous nous sommes intéressés à la navigation de tels agents autonomes au sein de ces environnements virtuels.

La planification de chemin est une tâche cruciale pour ces agents puisqu'elle va influencer directement leur autonomie de mouvement et leur capacité à agir au sein de ces mondes. Nous nous sommes focalisés au cours de nos travaux sur la planification de chemin pour des agents peuplant des environnements dynamiques dont la configuration évolue au cours de la simulation de manière non connue a priori.

Afin de répondre aux contraintes temporelles imposées par de nombreuses applications interactives, telles que les jeux vidéo, nous nous sommes également donné l'objectif de proposer une solution de planification assez performante pour être compatible avec de telles applications. Ce mémoire de thèse présente donc une des nouvelle méthode de planification de chemin au sein d'environnements dynamiques non connus a priori et compatible avec des applications interactives.

Chapitre 1 :

Représentations topologiques de l'environnement et les algorithmes de recherche de chemin.

1 INTRODUCTION

Un certain nombre de méthodes de représentation des environnements pour la planification de chemin ont été proposées .Une partie d'entre elles S'intéressent uniquement à la géométrie de l'environnement, alors que d'autres se basent sur des environnements informés pour obtenir des comportements de navigation plus crédibles.

1.1 Représentation topologiques de l'environnement

1.1.1 Représentations approximatives de l'environnement

Les représentations approximatives de l'environnement sont les plus utilisées en animation comportementale, du fait de leur facilité de mise en œuvre. Ces représentations vont décrire l'espace libre – de navigation – avec des formes géométriques simples telles que des carrés ou des segments. Deux modèles entrent dans cette catégorie : les modèles à base de grilles, et les cartes de cheminement.

1.1.1.1 Modèles à base de grilles

Le premier modèle de représentation approximative utilise des grilles régulières, formées de cellules carrées en deux dimensions (Figure 1.1(a)) et cubiques en trois dimensions. L'environnement est donc pavé par ces cellules, qui peuvent avoir trois états : libre, partiellement obstruée, et obstacle.

La précision de la représentation obtenue dépend directement de la taille des cellules utilisées: plus les cellules sont grandes, moins la représentation est précise. Bien entendu, l'augmentation de la précision induit une augmentation proportionnelle de l'occupation mémoire. L'occupation mémoire de cette méthode constitue ainsi son premier point faible, se répercutant directement sur la complexité de recherche de chemin à l'intérieur de l'environnement. **[1]**

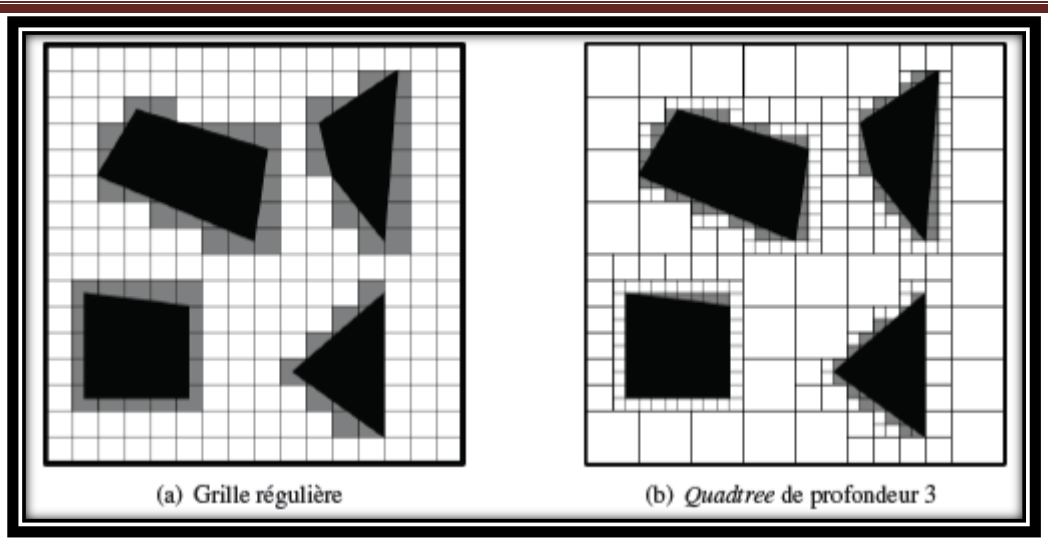


Figure 1.1 – Exemple de grilles régulières.

Pour réduire ce problème, une évolution de ce modèle est apparue sous la forme des grilles hiérarchiques. Cette méthode décrit l'espace navigable par une succession de grilles de plus en plus précises, organisées sous forme d'arbre. L'algorithme de construction va commencer par paver l'environnement avec les cases les moins précises, puis découper récursivement les cases partiellement obstruées, en quatre parties pour la 2D (formation d'un quadtree – Figure 1.1(b)), ou en huit pour la 3D (formation d'un octree). L'algorithme arrête les subdivisions dès qu'une précision fixée est atteinte, ou si la case produite n'est plus partiellement obstruée. Cette méthode est donc d'autant plus avantageuse que l'environnement est peu dense en obstacles.

1.1.1.2 Cartes de cheminement

Les cartes de cheminement discrétisent l'espace navigable sous la forme d'un réseau de chemins. Ce réseau est obtenu en reliant des points clefs répartis à l'intérieur de l'environnement (Figure 1.2). Plusieurs méthodes existent pour créer des cartes de cheminement, différentes dans la manière de créer et de relier ces points clefs.

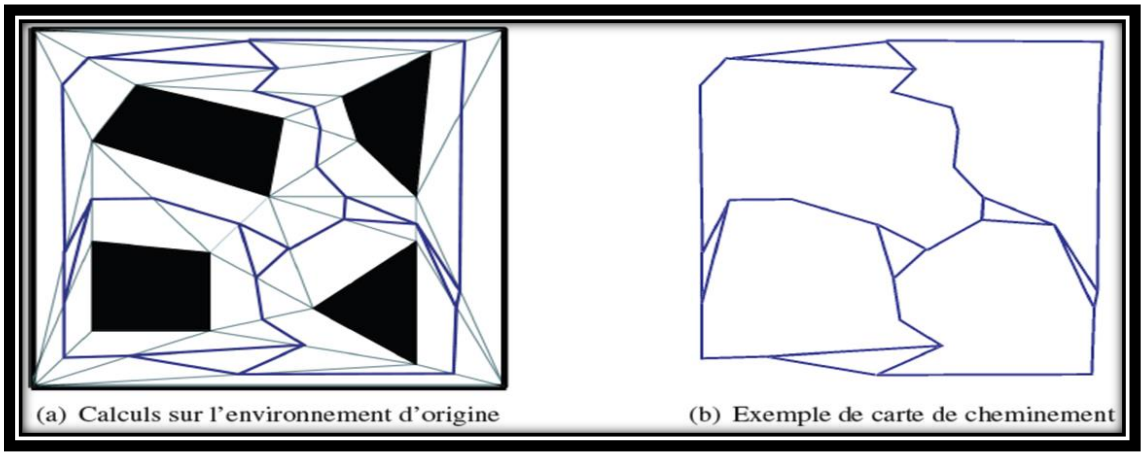


Figure1.2– Exemple de carte de cheminement. A gauche, triangulation de Delaunay (gris) de l'environnement d'origine, et carte de cheminement déduite (bleu). A droite, un exemple de carte de cheminement obtenue.

- **Graphe de visibilité**

Cette méthode utilise les sommets des polygones représentant les obstacles comme points clefs. Les points clefs sont ensuite reliés deux à deux s'ils sont mutuellement visibles, c'est à dire si l'on peut tracer une ligne droite passant par les deux points sans rencontrer d'obstacle. Les graphes ainsi créés minimisent les distances parcourues, assurant d'obtenir des chemins de longueur minimale. La taille du graphe généré est ici dépendante du nombre de points mutuellement visibles, et est donc d'autant plus importante que l'environnement est ouvert avec des obstacles ponctuels et disparates.

- **Diagramme de Voronoï généralisé**

Cette méthode est basée sur une notion d'équidistance aux obstacles de l'environnement. Pour se faire, un ensemble de sites sont évalués au sein de l'environnement, correspondant aux obstacles, dont les intersections vont former les points clefs. Les chemins ainsi générés maximisent la distance aux obstacles. Ce calcul s'avère complexe, mais son approximation peut être obtenue rapidement par des méthodes utilisant les cartes graphiques, où le calcul est obtenu directement en effectuant le rendu des sites. Une autre méthode utilise la triangulation de Delaunay pour obtenir les points clefs du diagramme de Voronoï généralisé, en se servant du centre des triangles calculés. La carte de cheminement ainsi produite peut être considérée comme un condensé des informations de la

triangulation, ne contenant plus la définition géométrique des obstacles de l'environnement. **[1]**

- **Cartes de cheminement probabilistes**

Avec cette méthode, la définition géométrique de l'environnement n'est pas utilisée comme support direct de la construction. En effet, les points clefs sont ici obtenus par une distribution aléatoire au sein de l'environnement navigable. Deux points sont ensuite reliés s'il existe un chemin libre de collision entre eux. Ce test est effectué en se basant sur la géométrie de l'environnement, mais aussi sur la taille de l'entité qui se déplace. Cette méthode est plus largement utilisée en planification de mouvement qu'en navigation, afin de générer des déplacements assez complexes (sauter, se baisser, marcher sur des piliers).

Pour conclure, les cartes de cheminement présentent le net avantage de fournir une description très condensée de l'environnement, ne nécessitant une prise de décision qu'au niveau des points clefs. Néanmoins, leur définition seule n'est pas suffisante pour gérer la complexité de la locomotion humaine. Par exemple, leur exploitation devient très difficile lorsqu'il s'agit de gérer le croisement des entités le long d'un même chemin. Certaines méthodes proposent de régler ce problème en subdivisant chaque chemin dans sa largeur, les gérant ainsi comme un ensemble de rails parallèles entre lesquels vont pouvoir transiter les entités en mouvement. Mais, même si une telle méthode produit des animations visuellement plausibles, son fonctionnement est trop éloigné du comportement humain pour être exploité dans des simulations réalistes. Les représentations sous forme de cartes de cheminement semblent donc bien adaptées à un processus de planification de chemin général, ne tenant pas compte des autres entités, mais beaucoup moins à un processus de navigation, gérant lui les mouvements locaux.

1.1.1.3 Le modèle de champs de potentiels

Le modèle à base de champs de potentiels consiste en une définition de l'environnement permettant directement de résoudre les déplacements de personnes. Les champs de potentiels caractérisent les obstacles de l'environnement par des forces de répulsion, et les buts par des forces d'attraction. Un gradient de forces, ou champ de potentiel, est alors déduit en chaque point de l'environnement comme étant une somme pondérée, le plus souvent par la distance,

des potentiels de répulsion et du potentiel lié au but (Figure 1.3). La navigation d'une entité est alors simulée par une descente de gradient depuis sa position dans l'environnement. **[1]**

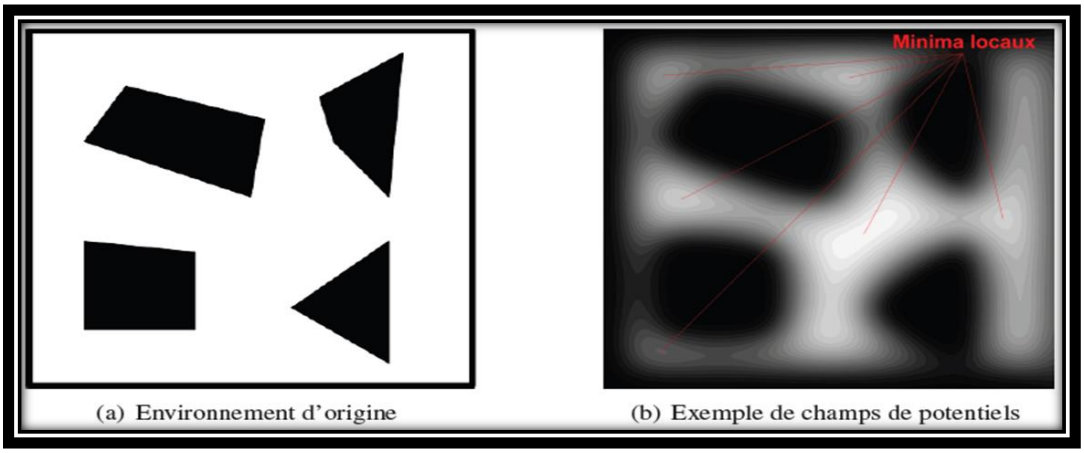


Figure 1.3 – Carte de champs de potentiels : en noir les obstacles (répulsion), en blanc les zones de navigation (attraction). Le gradient de gris exprime la valeur de potentiel dans l'environnement. Cet exemple contient 6 minima locaux : zones isolées à potentiel d'attraction maximal.

Les méthodes basées sur les champs de potentiels s'avèrent simples et efficaces, mais posent le problème des minima locaux : zones de l'environnement où un potentiel minimal isolé apparaît (Figure 1.3 (b)). Ainsi, la méthode de navigation associée va pousser l'entité à se déplacer vers le minimum local le plus proche, qui ne représente pas forcément son but. Pour pallier ce type de problème, des méthodes à base de marche aléatoire sont utilisées. Par exemple, dans RPP (Random Path Planner). Lorsqu'un minimum local est atteint, un ensemble de configurations aléatoires sont tirées puis testées avec une phase de sortie du minimum et une phase de convergence vers le prochain minimum. Les informations sont alors stockées dans un graphe dont les nœuds sont les minima locaux et les arcs traduisent des chemins entre deux minima. D'autres méthodes existent à base de tirage aléatoire de direction à suivre. Plutôt que de configuration, pour échapper au minimum local. **[1]**

Ces méthodes ne sont cependant pas très adaptées à l'animation comportementale. Les comportements induits par les tirages aléatoires, s'ils sont acceptables pour des robots, ne le sont pas pour des humanoïdes car ils sont en dehors de la logique de navigation humaine qui comporte une part importante de planification

1.1.2 Représentations exactes de l'environnement

Les représentations exactes de l'environnement vont chercher à organiser les données spatiales tout en conservant intégralement les informations qu'elles contiennent à l'origine. La méthode utilisée consiste généralement à découper l'espace navigable en cellules convexes de différentes formes. Il existe plusieurs modèles pour effectuer cette subdivision, mais nous nous attacherons ici à la plus utilisée qu'est la triangulation de Delaunay, ainsi qu'à ses évolutions.

1.1.2.1 Triangulation de Delaunay

La triangulation de Delaunay crée un ensemble de triangles en réunissant des points fournis en entrée. L'algorithme d'unification respecte pour contrainte que le cercle circonscrit à un triangle ne contienne aucun autre point que les trois sommets qui le composent. Une conséquence de cette propriété est que l'angle minimum d'un triangle produit est maximisé.

La propriété la plus intéressante de cette triangulation est que chaque point y est relié à son plus proche voisin par l'arête d'un triangle. Ainsi, cette triangulation peut être utilisée pour représenter l'espace navigable, les points d'entrée étant issus des obstacles. Une autre utilisation possible de cette triangulation est cette fois dynamique lors de la simulation, pour calculer un graphe de voisinage entre les entités mobiles, qui serviront cette fois-ci de point d'entrée.

1.1.2.2 Triangulation de Delaunay contrainte

La triangulation de Delaunay contrainte permet de modérer les contraintes de la version standard afin de conserver certaines arêtes de la définition graphique d'origine. Pour se faire, la contrainte du cercle circonscrit à un triangle est modifiée, pour spécifier que tout point inclus dans ce cercle ne peut être relié à tous les points du triangle sans intersecté un segment contraint.

Cette triangulation étend la propriété du plus proche voisin en permettant d'y associer une notion de visibilité. Si l'on considère que les arêtes contraintes coupent la visibilité d'un point à l'autre, la propriété de plus proche voisin devient : chaque point est relié à son plus proche voisin visible.

La triangulation de Delaunay contrainte est aussi utilisée pour obtenir une subdivision spatiale en triangles, en introduisant les obstacles à la navigation sous la forme d'autant de segments contraints (Figure 1.4). Il a été prouvé que le nombre de triangles produits lors d'une telle subdivision est linéaire en fonction du nombre de points, indiquant que la discrétisation est donc directement proportionnelle à la complexité géométrique de l'environnement.

Cette propriété présente un avantage certain comparativement aux méthodes approximatives, où la discrétisation est fonction de la précision désirée de la représentation.

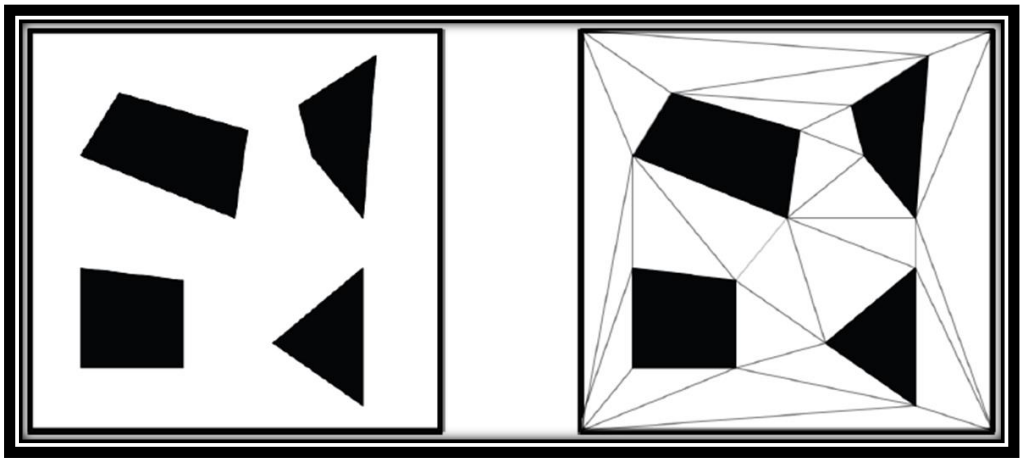


Figure 1.4– Exemple de triangulation de Delaunay contrainte.

1.1.2.3 Triangulation de Delaunay filtrée

La triangulation de Delaunay filtrée est une extension de la version contrainte. Deux types de filtrages sont proposés, l'un ayant pour effet d'augmenter le nombre de triangles produits afin d'affiner la représentation de l'environnement, l'autre de le diminuer afin de tenir compte de la visibilité pour l'élaboration d'un graphe de voisinage. Premièrement, concernant son application à la subdivision spatiale, la triangulation.

Dans F. Lamarche et S. Donikian proposent ce type de décomposition, afin d'optimiser l'organisation de l'espace. Le premier point est de ramener un environnement 3D en un environnement 2D par projection sur un plan 2D. Grâce à une triangulation de Delaunay, ils subdivisent l'espace une première fois en cellules triangles.

Une caractéristique importante pour la navigation d'humanoïdes est la présence de goulots d'étranglement dans l'environnement ainsi que leurs positions. Pour connaître ces emplacements, il faut calculer la distance minimale entre les coins et les murs, ils vont ainsi

avoir une carte 2D de l'environnement avec une subdivision en cellules efficace comme il est montré sur la figure 1.5.

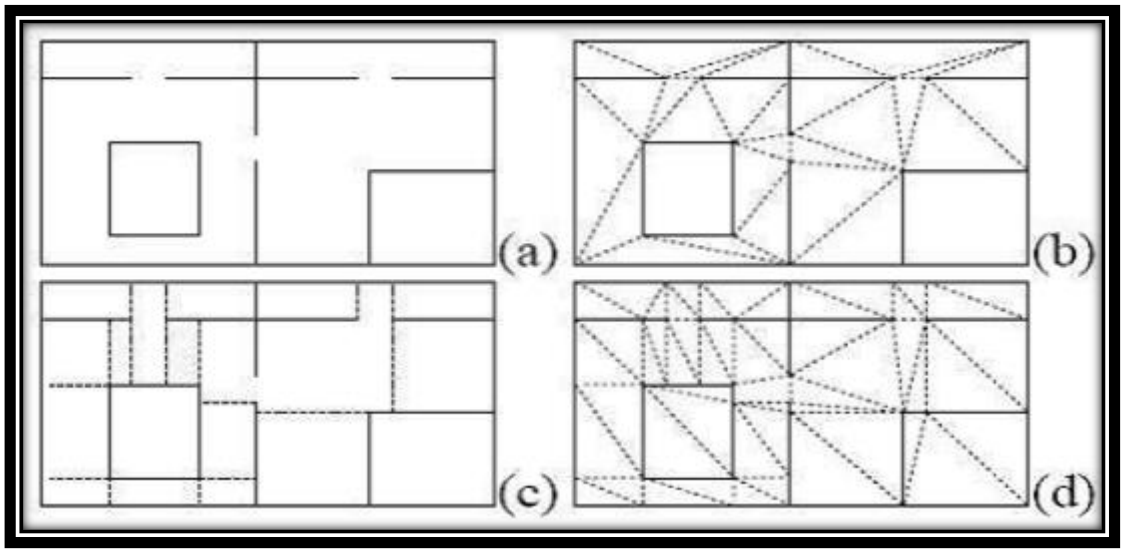


Figure 1.5 – Construction de l'environnement : (a) Carte 2D. (b) Triangulation de Delaunay. (c) Distances minimaux coins murs. (d) Triangulation et distances minimales coins murs.

1.2 Qu'est-ce que la planification de chemin et planification de trajectoire?

La planification de chemin et la planification de trajectoire sont des questions cruciales dans le domaine de la robotique et, plus généralement, dans le domaine de l'automatisation. En effet, la tendance pour les robots et les machines automatiques est de fonctionner à une vitesse de plus en plus élevée, afin d'obtenir des temps de production plus courts. La vitesse de fonctionnement élevée peut nuire à la précision et à la répétabilité du mouvement du robot, car des performances extrêmes sont exigées des actionneurs et du système de commande. Par conséquent, il convient d'accorder une attention particulière à la génération d'une trajectoire qui puisse être exécutée à grande vitesse, mais qui soit en même temps inoffensive pour le robot, en évitant les accélérations excessives des actionneurs et les vibrations de la structure mécanique. Une telle trajectoire est définie comme lisse. Pour ces raisons, les algorithmes de planification de chemin et de trajectoire prennent une importance croissante en robotique. Les algorithmes de planification de trajectoire génèrent une trajectoire géométrique, d'un point initial à un point final, en passant par des points de passage prédéfinis, soit dans l'espace articulaire, soit dans l'espace opérationnel du robot,

tandis que les algorithmes de planification de trajectoire prennent une trajectoire géométrique donnée et la dotent d'informations temporelles. Les algorithmes de planification de trajectoire sont essentiels en robotique, car la définition des temps de passage aux points de passage influence non seulement les propriétés cinématiques du mouvement, mais aussi les propriétés dynamiques. En effet, les forces d'inertie (et les couples) auxquelles le robot est soumis dépendent des accélérations le long de la trajectoire, tandis que les vibrations de sa structure mécanique sont essentiellement déterminées par les valeurs de la saccade (c'est-à-dire la dérivée de l'accélération). Les algorithmes de planification de chemin sont généralement divisés en fonction des méthodes utilisées pour générer le chemin géométrique, à savoir

- les techniques de feuille de route
- les algorithmes de décomposition cellulaire
- les méthodes de potentiel artificiel.

Les algorithmes de planification de trajectoire sont généralement nommés en fonction de la fonction à optimiser, à savoir

- Temps minimum
- Énergie minimale
- Secousse minimale.

1.3 Les algorithmes de parcours de chemin

1.3.1 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.

1.3.2 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.

1.3.3 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).

1.3.4 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.

1.3.5 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.

1.3.6 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.

1.3.7 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. **[7]**

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. **[4]**

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'heuristique est surtout la caractéristique première de l'algorithme A*. S'il n'y a aucune heuristique alors 1 '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 1 ' 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. **[4]**

1.3.8 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. **[3]**

1.3.9 Alternative-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* **[6]** 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).

1.3.10 Hierarchical Pathfinding A* (HP A*)

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. **[**

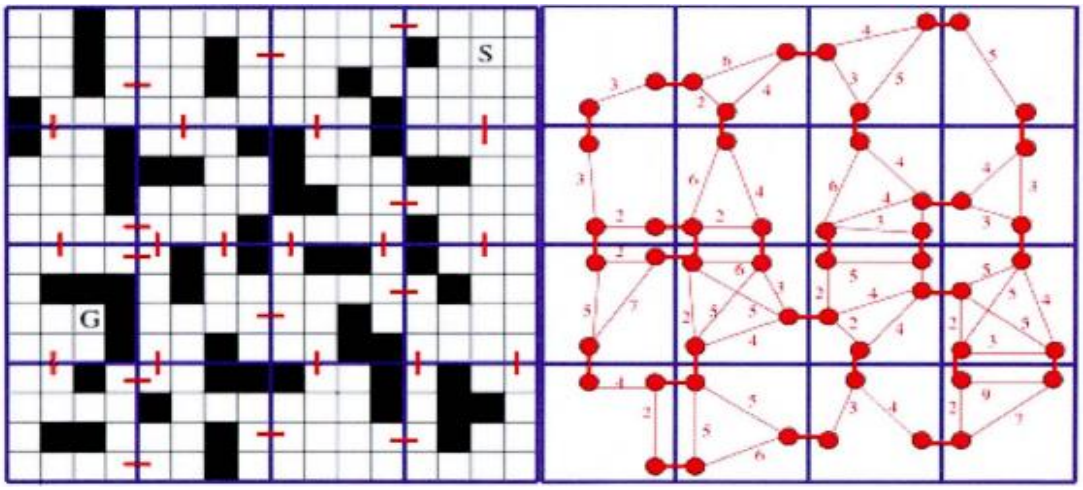


Figure 1.6 HPA* (Botea, Müller et Schaeffer, 2004)

1.3.11 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***[6]**, 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.

L'intérêt croissant pour la résolution de problèmes de haute dimension, tels que la planification de mouvements pour les humanoïdes ou les bras à plusieurs degrés de liberté, a donné naissance à une classe d'outils de résolution de problèmes.

Les planificateurs basés sur l'échantillonnage. La version la plus populaire des planificateurs basés sur l'échantillonnage est basée sur des arbres aléatoires à exploration rapide (RRT) **[15].** Ils recherchent le chemin vers le but en sondant l'espace de configuration (peut également être fait dans l'espace de travail) et en développant de manière incrémentielle un arbre sans collision à partir d'une configuration initiale. Parce que l'ensemble de la recherche se fait "dans l'obscurité", le planificateur tente d'atteindre des parties inexplorées de l'espace de recherche.de l'espace de recherche, ce qui permet d'obtenir une couverture uniforme de l'espace libre. L'efficacité de ces planificateurs provient de la couverture incomplète de l'espace libre et du fait que la recherche se termine lorsque la configuration initiale est terminée. L'espace libre et de la fin de la recherche lorsque le but est atteint pour la première fois. La solution trouvée est faisable mais en aucun cas optimale. Les planificateurs basés sur le RRT peuvent ne pas produire solutions optimales même en explorant tout l'espace de recherche. De plus, ils ont par nature des difficultés avec les espaces restreints. Néanmoins, les algorithmes RRT ont été démontrés pour résoudre des problèmes très complexes.

1.4 Arbres aléatoires d'exploration

1.4.1 Principe des RRT

Dans sa forme originale, la méthode *RRT* est un arbre $G=(V, E)$ avec V l'ensemble des éléments de l'espace de recherche[1] de l'arbre et E l'ensemble des arêtes de l'arbre. À partir d'une configuration initiale q_{star}, l'objectif est d'énoncer une suite de commandes, permettant au mobile M, d'explorer l'espace des configurations C. Pour résoudre ce problème, la méthode *RRT* recherche une solution par construction d'un arbre dont le nœud-racine est la configuration q_{init}. Les nœuds de l'arbre sont des configurations admissibles de M. Les arcs de l'arbre sont les commandes à appliquer pour passer d'une configuration à une autre. La méthode *RRT* est une recherche incrémentale aléatoire des commandes permettant une exploration uniforme de l'espace. Elle répète successivement une boucle composée de trois

phases : génération d'une configuration q_{rand}, sélection d'une configuration q_{prox}, génération d'une nouvelle configuration q_{new}.

Lors d'une phase de génération d'une configuration q_{rand}, une fonction aléatoire sélectionne un élément de l'espace des configurations. La phase suivante sélectionne l'élément de G, le plus proche nœud de q_{rand} ; cet élément est appelé q_{prox}. ; Cette sélection est fondée sur la définition d'une distance métrique ρ. La phase de génération d'une nouvelle configuration q_{new} est l'application d'une commande visant à rapprocher q_{prox} de q_{rand}. La nouvelle configuration q_{new} est créée par intégration des contraintes du mobile M pendant un intervalle de temps fixé à partir de la configuration q_{prox}.

1.4.2 Construction du graphe de la méthode RRT

Initialement, la méthode *RRT* considère le problème de planification de mouvement pour un mobile quelconque. L'espace est de dimension arbitraire et dénué d'obstacle. L'algorithme 1 présente la construction élémentaire du graphe de la méthode *RRT*.
Les trois étapes de construction sont ici conduites dans les fonctions 1 ,2 et
3 (ALG. 1). Conf Aleatoire assure une répartition uniforme des tirages aléatoires dans C et garantit ainsi une exploration uniforme de C. **[9]**
confLa Plus Proche sélectionne la configuration de G la plus proche de q_{rand}. Cette relation de proximité est définie par la distance métrique ρ (ALG. 2).
Nouvelle Conf définit une nouvelle configuration q_{new} à partir de q_{prox} en direction de q_{rand}. Le mobile étant ici considéré comme holonome, il est toujours possible d'appliquer une commande visant à déplacer q_{prox} en direction de q_{rand}. L'amplitude de ce déplacement est définie par la valeur Δt. La fonction ajouterConf ajoute q_{new} dans la liste des sommets de G et la fonction ajouterArc ajoute un arc reliant q_{prox} et q_{new}.

consRrt(q_{nit},k,delt,C)
q_{lnt}(q_{init}, G) ; ,*i*
pour i «- 1 à k
qrand <- confAleatoire(c) ;
q_{prox} <- confLaPlusProche(q_{rand}, G) ; ,2
q_{new} <-nouvelleConf($q_{proX;}q_{rand/}$At) ; ,3
ajouterConf($q_{neW/}$G) ;
ajouterArc(q_{prox},q_{new},G);
retourner Q ;

ALG. 1 : Construction élémentaire du graphe de la méthode*RR.T.*

CONFLAPKISPROCHE(QRAND , G)
d←+ 00 ;
pour chaque q g G
si p(q, qrand) < d
qpr QX q '
d <- p(q, q-and) '
| retourner q_{prox} ;

ALG. 2: Recherche du plus proche voisin

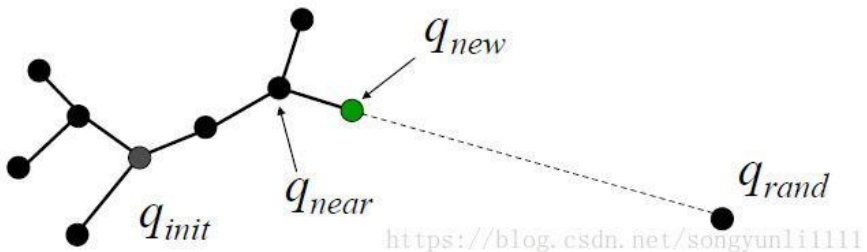


Figure 1.7 Construction élémentaire du graphe de la méthode RRT

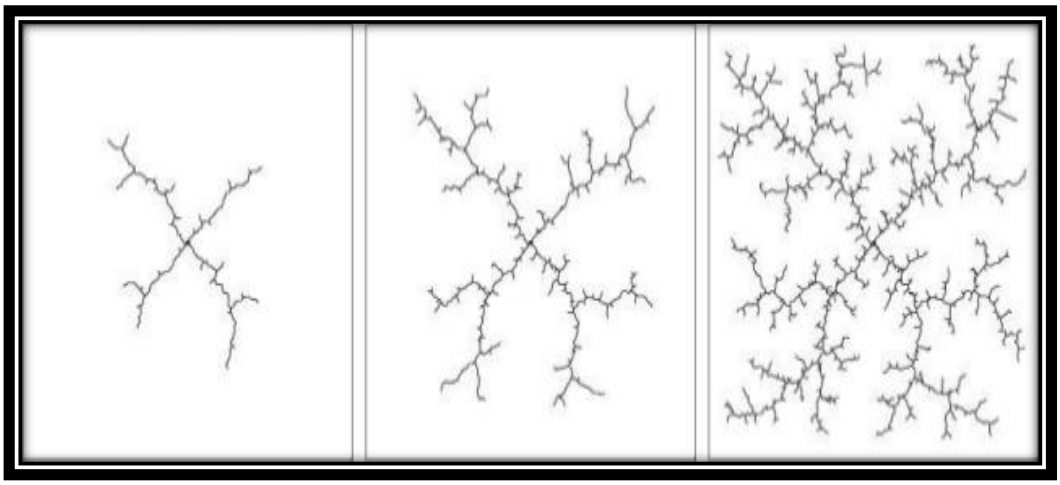


Figure 1.8 Déroulement de RRT selon nombre d'itération

1.4.3 évolution des RRT

1.4.3.1 RRT simple

- RRT est un algorithme de planification probabiliste « single-query »
- L'algorithme ne généra pas une roadmap qui représente d'une façon exhaustive la connectivité de l'espace libre dans tout l'environnement. On explore seulement une portion de l'espace libre qui est pertinente à la solution du problème (il en résulte une réduction importante du temps de calcul)
- La méthode RRT utilise une structure des données qui s'appelle Rapidly-exploring Random Tree (RRT)
- L'extension incrémentale de l'RRT (on appelle cet arbre « T ») est basée sur une simple procédure stochastique répétée à chaque itération

1.4.3.1.1 Algorithme

• On génère un échantillon aléatoire de l'espace des configurations en utilisant une pdf uniforme (comme pour la méthode Probabilistic RoadMaps (PRM) Algorithme qrand

• La configuration en T la plus proche à est déterminée, et une nouvelle configuration candidate est produite sur le segment joignant à, à une distance préfixée de qnear qrand qnew qnear qrand qnear

• On vérifie que soit soit le segment de à n'engendre pas de collisions. Si tel est le cas, T est élargi en incluant et le segment le joignant à (n'est pas rajouté à T, donc il n'est pas nécessaire de vérifier s'il engendre des collisions: sa seule fonction est d'indiquer la direction d'extension de l'arbre T)

```
Algorithm 1: Original RRT
 1  T.add(qstart )
 2  while iteration < K do
 3    qrand = random configuration
 4    qnear = nearest neighbor in      tree T to qrand
 5    qnew = extend qnear toward      qrand
 6    if Çnew can connect to qneaT      then
 7         T.addVertex(gnew);
 8         T.addEdge(qnear.qnew);
 9    end
10       if Q(qnew qgoal) < distanceToGoalTh then
11       | break;
12       end
13  end
```

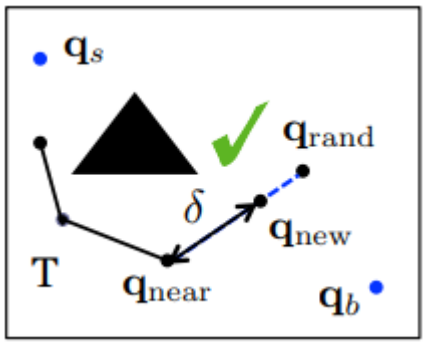


Figure 1.8 Construction élémentaire du graphe de la méthode RRT lors d'un obstacle

1.4.3.2 RRT*(RRT star) :

RRT* est une version optimisée de RRT. Lorsque le nombre de nœuds approche l'infini, l'algorithme RRT* fournira le chemin le plus court possible vers le but. Bien qu'irréalisable d'un point de vue réaliste, cette affirmation suggère que l'algorithme s'efforce d'élaborer le chemin le plus court. Le principe de base de RRT* est le même que celui de RRT, mais deux ajouts clés à l'algorithme donnent des résultats sensiblement différents.

Tout d'abord, RRT* enregistre la distance parcourue par chaque sommet par rapport à son sommet parent. C'est ce qu'on appelle cost()le sommet. Une fois le nœud le plus proche trouvé dans le graphe, un voisinage de sommets dans un rayon fixe à partir du nouveau nœud est examiné. Si un nœud avec un nœud moins cher cost()que le nœud proximal est trouvé, le nœud le moins cher remplace le nœud proximal. L'effet de cette caractéristique peut être vu avec l'ajout de brindilles en forme d'éventail dans la structure arborescente. La structure cubique de RRT est éliminée. **[10]**

La deuxième différence ajoutée par RRT* est la reconnexion de l'arbre. Une fois qu'un sommet a été connecté au voisin le moins cher, les voisins sont à nouveau examinés. Les voisins sont vérifiés si le fait d'être recâblé au sommet nouvellement ajouté réduira leur coût. Si le coût diminue effectivement, le voisin est recâblé au sommet nouvellement ajouté. Cette fonctionnalité rend le chemin plus fluide. Différent

Algorithm : RRT*($X_{start} \in X_{free}, X_{goal} \in X$)

1 $V \leftarrow \{x_{start}\}; E \leftarrow 0; T = (V, E);$

2 **for** $i = 1 \ldots q$ **do**

3 x_{rand} **Sample,**

4 $v_{nearest}$ **Nearest** (V, x_{rand}),

5 x_{new} **Steer** $(x_{nearest}, x_{rand})$,

6 **If CollisionFree** $(x_{nearest}, x_{new})$ **then**

7 $V \leftarrow \{x_{new}\}$;

8 v_{near} **Near** (V, x_{new}, r_{rewire});

9 $v_{min} \leftarrow v_{nearest}$;

10 **forall** $v_{near} \in V_{near}$ **do**

11 $c_{new} \leftarrow gT(v_{near}) + c(V_{near} - X_{new})$,

12 **if** $c_{new} < gT(v_{min}) + c(v_{min}, x_{new})$ **then**

13 **if CollisionFree** (v_{near}, x_{new}) **then**

$v_{mîn} \leftarrow x_{near}$;

14

15 $E \leftarrow \{(v_{min} x_{new})\}$;

16 **forall** $v_{near} \in x_{near}$ **do**

17 $C_{near} \leftarrow gT(x_{new}) + c(x_{new}, V_{near})$;

18 **if** $C_{near} < gT(v_{near})$ **then**

19 **if CollisionFree** (x_{new}, v_{near}) **then**

20 $V_{parent} \leftarrow$ **Parent** (v_{near});

21 $E \leftarrow \{(V_{parent}, V_{near})\}$;

22 $E \{(x_{new}) V_{near})\}$;

23 ***Return N;***

1.4.3.3 RRT* smart :

RRT*-Smart est une extension de RRT* avec la convergence plus rapide en comparaison de ses prédécesseurs. RRT*-Smart au hasard fouille l'espace public comme RRT* fait. De même le premier sentier est trouvé comme le RRT* essaierait de trouver un sentier par l'échantillonnage au hasard dans l'espace de configuration. Dès que ce premier sentier est trouvé, il l'optimise alors en raccordant les noeuds directement visibles. Ces productions de sentier optimisées influant sur les points pour l'échantillonnage intelligent. À ces points influants, l'échantillonnage survient à intervalles réguliers, qui sont gouvernés par b constant qui dépend à son tour du rapport biaisng expliqué dans la Section 6. Ce processus est continué, comme les progrès d'algorithme et le sentier est optimisé constamment. Chaque fois que, un sentier plus court est trouvé, les changements influants vers le nouveau sentier.

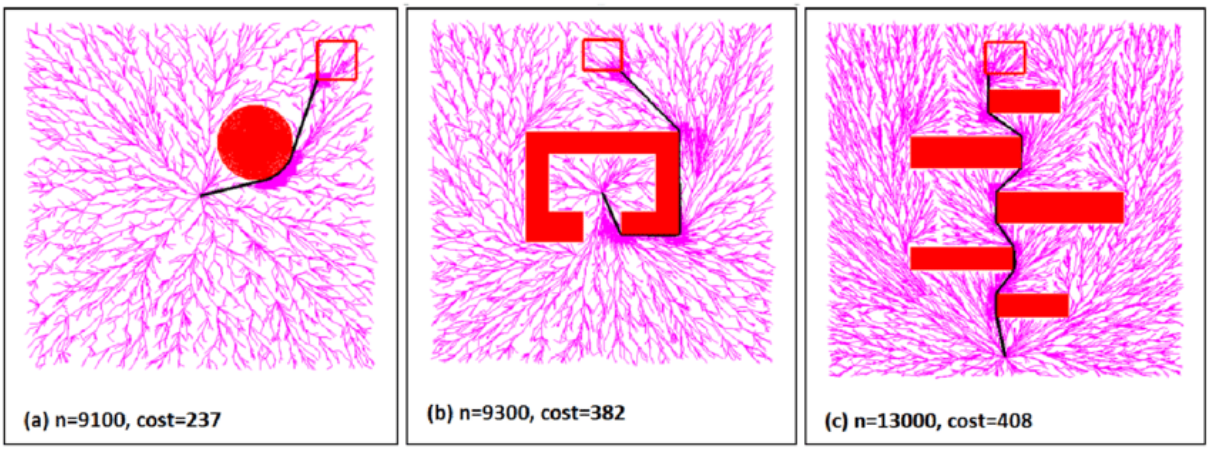


Figure 1.10: RRT-smart dans des environnements de défirent obstacles

1.4.3.4 Informed RRT*

Informed RRT* est une modification simple à RRT* qui démontre une amélioration claire. Dans la simulation, il joue aussi bien qu'existant des algorithmes de RRT* sur les configurations simples et démontre des améliorations d'ordre de grandeur comme les configurations deviennent plus difficiles. À la suite de sa recherche concentrée, l'algorithme à moins de dépendance à la dimension et au domaine du problème de planification aussi bien que la capacité de trouver mieux topologiquement des sentiers distincts plus bientôt.

C'est aussi capable de trouver des solutions dans les tolérances plus serrées de l'optimum que RRT* avec le compte équivalent et faute des obstacles peut trouver la solution optimale de dans le zéro de machine dans le temps fini .Il pourrait aussi être utilisé dans la combinaison avec d'autres algorithmes, comme path_smoothing, davantage réduire l'espace de recherche.

1.4.3.5 Spline-RRT*

spline-RRT ∗ l'algorithme de planification de chemin a été proposé pour fixer UAVs qui peut trouver des chemines dans les environnements de planification hautement contraints. L'algorithme proposé peut améliorer l'aspect lisse du chemin produit, mais ne fait rien pour accélérer le taux de convergence.

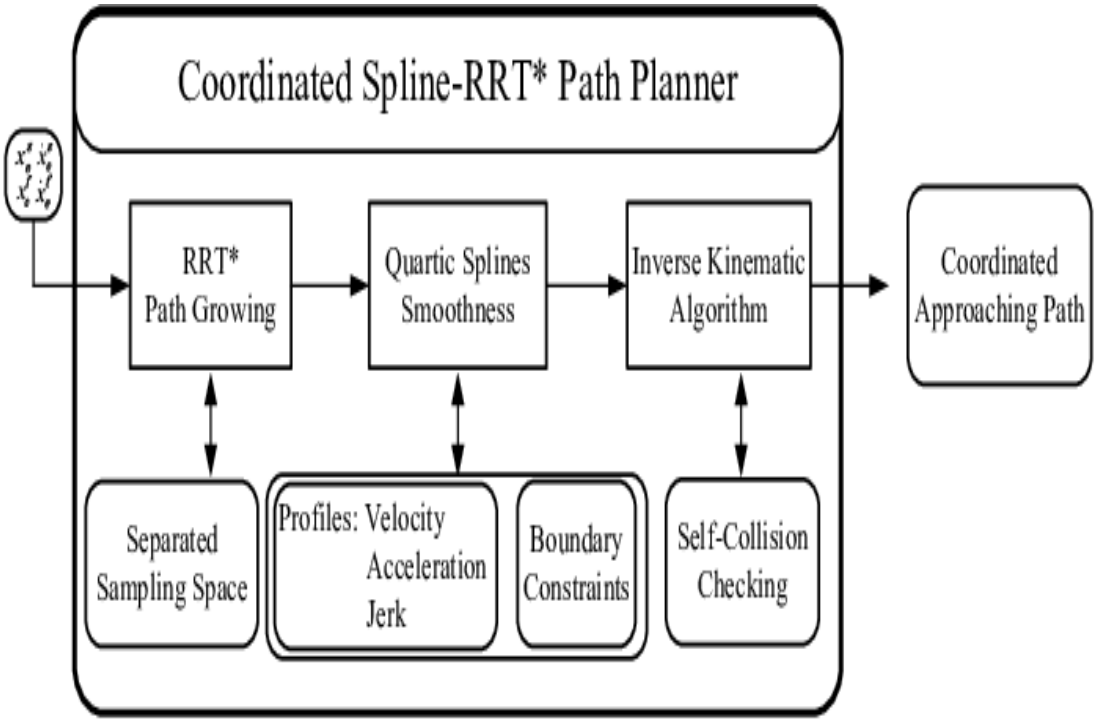


Figure 1.11 : Aperçu du planificateur coordonné spline-RRT*.

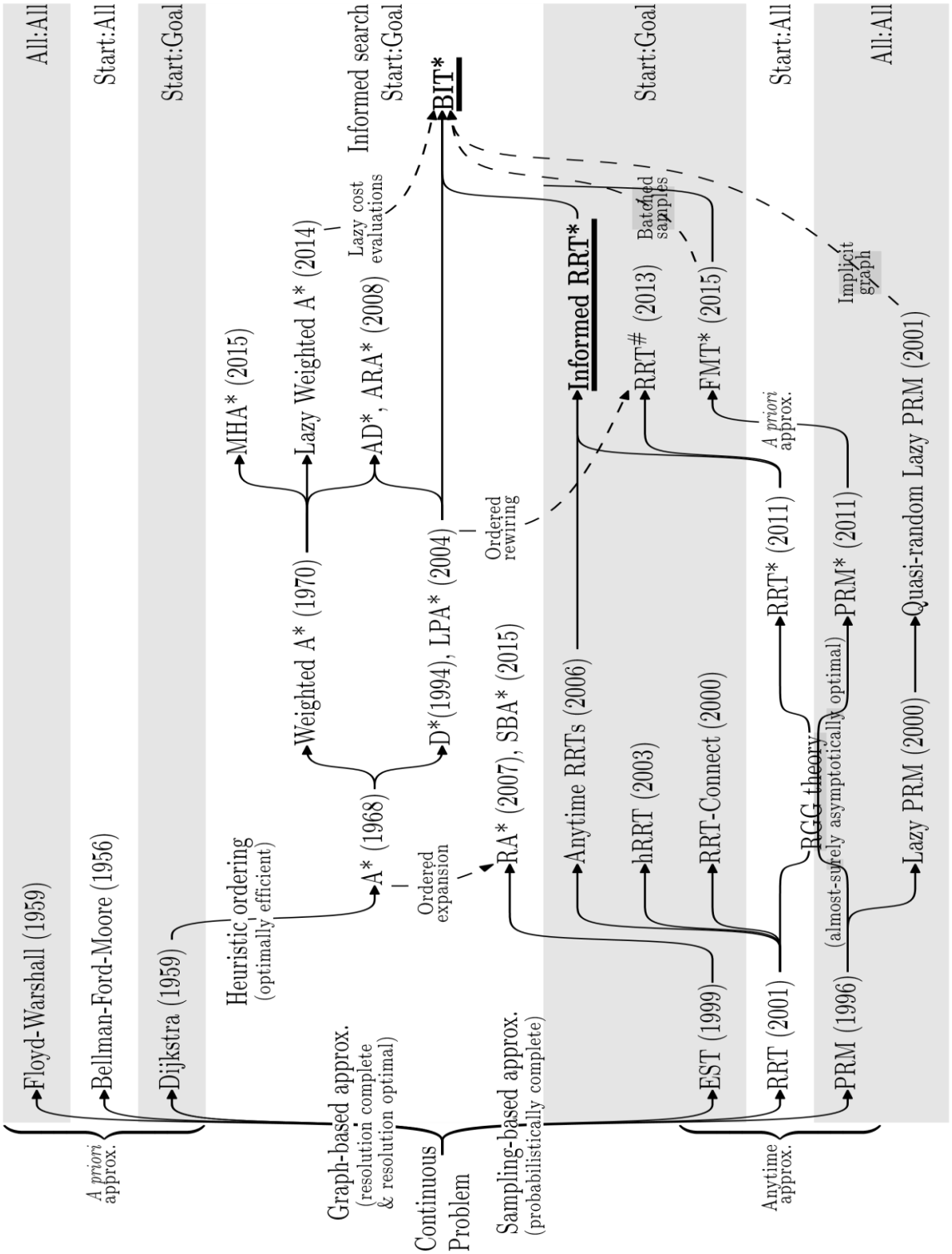


Figure 1.12: historique simplifier des algorithmes de planification

1.5 Tableau de comparaison entre les algorithmes de parcours de chemin

Algorithm	Complexité	Structure de donné	Heuristique
Djikstra	N^2	Liste	**Pas d'Heuristique**
A*	Nlog(n)	File	Manhattan/ euclidienne
B*	**Nlog(n)**	File	Manhattan/ euclidienne
Algorithme funnel	**Nlog(n)**	File	**Pas d'Heuristique**
Best-firs-search	**Nlog(n)**	**Arbre/file**	**Heuristique**
RRT	**Nlog(n)**	**Arbre**	**Pas d'Heuristique**
RRT*	**Nlog(n)**	**Arbre**	**Pas d'Heuristique**

1.6 Conclusion

Des méthodes de planification de chemin se basant uniquement sur la géométrie de l'environnement existent, mais le manque d'information sémantique nuit à la crédibilité de ces chemins. L'utilisation d'environnements informés a été proposée dans le but d'augmenter le réalisme du comportement par la prise en compte de la sémantique associée aux zones de navigation. Cependant l'information manuelle de ces environnements peut s'avérer longue et fastidieuse. Notre travail consiste donc à proposer une solution permettant l'extraction automatique de zones de navigation sur la base d'une géométrie 3D, ainsi que de l'étiquetage de ces zones par de l'information pertinente sans intervention manuelle nécessaire.

Chapitre 02 :

Planification de chemin

Dans un environnement

2.1 Introduction à la planification de chemin

La problématique de la planification de chemin est définie ainsi : étant donné un robot possédant un nombre variable de degrés de liberté, et une description de l'environnement où ce robot est immergé, comment trouver un chemin libre de toutes collisions entre deux configurations du robot au sein de l'environnement. De nombreux exemples illustrant cette problématique peuvent être cités. Le rôle de la planification va être par exemple de déterminer un chemin entre deux positions dans un environnement. Il peut s'agir ainsi d'un agent virtuel cherchant son chemin dans un environnement tel qu'un Personnage Non Joueur (PNJ) dans un jeu vidéo. On peut également s'intéresser au déplacement d'un bras mécanique industriel possédant de nombreux degrés de libertés et étant placé dans un environnement très contraint. La nature du robot (ou de l'agent) considéré ainsi que les propriétés de l'environnement dans lequel il va évoluer vont définir différents types de problèmes. Vu la grande variété de problèmes abordés, la planification de chemin a fait l'objet de nombreuses recherches durant les dernières décennies, de nombreux résultats sont discutés dans les états de l'art proposés par Latombe **[11]** et LaValle **[14].**

2.2 Planification de chemin en environnements statiques

Les environnements statiques se caractérisent par leur structure qui n'est pas amenée à évoluer au cours du temps. Cette propriété implique donc qu'il n'y a pas de nouvelles accessibilités ou obstructions créées pendant le déroulement de la simulation. Puisque la configuration de ces environnements ne change pas, il est donc possible d'effectuer des prés calculs afin de proposer des structures de données performantes lors de requêtes de planification de chemin ultérieures. Les méthodes proposées pour caractériser les environnements navigables vont se diviser principalement en deux familles. D'un côté, les méthodes de décomposition en cellules vont représenter l'espace navigable à l'aide de zones interconnectées de formes prédéfinies permettant d'appréhender les limites de C-Free et de C-Obstacles. D'un autre côté, les méthodes utilisant les cartes de cheminement vont représenter un sous-ensemble de C-Free par un ensemble de chemins standardisés permettant la navigation de l'agent.

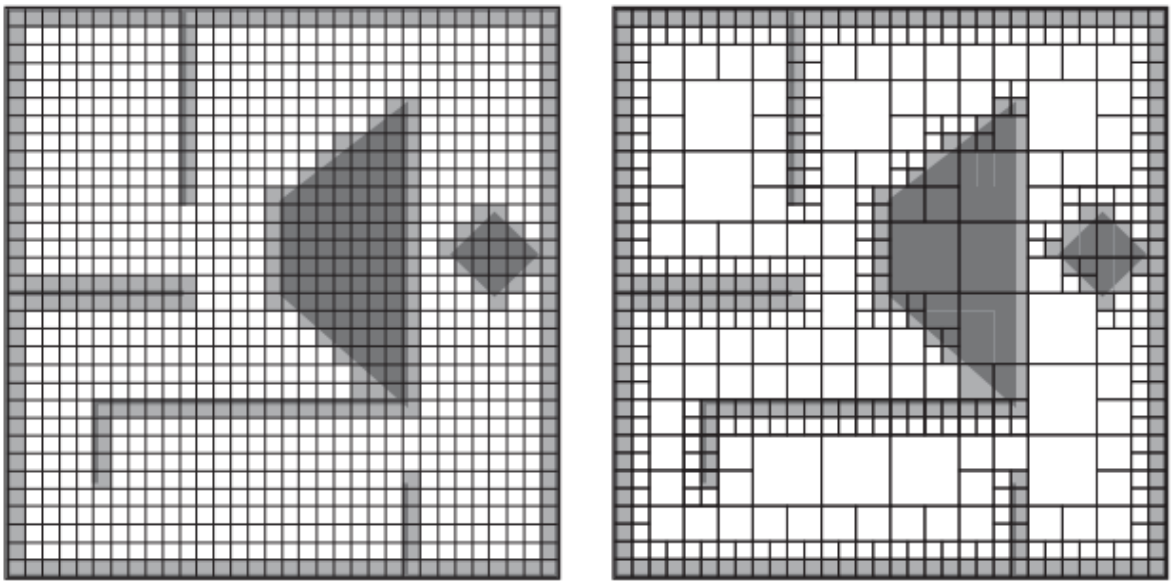
2.2.1 Décomposition en cellules

La décomposition en cellules propose de diviser l'espace des configurations en cellules de formes prédéfinies. Celles-ci sont ensuite caractérisées selon leur appartenance à C-Free ou C-Obstacle afin de construire une structure de données plus adaptée à la planification de chemin. Les méthodes de décomposition en cellules se répartissent en deux familles : les décompositions

approchées et les décompositions exactes. Les décompositions approchées construisent une représentation approchée de l'environnement en ne cherchant pas à caractériser précisément C-Free mais en en définissant un sous-ensemble dont la représentation est plus simple. Les décompositions exactes sont des structures généralement plus coûteuses à calculer mais qui permettent une description plus fine de l'environnement en décrivant exactement C-Free et, par conséquent, C-Obstacle.

2.2.1.1 Décomposition approchée en cellules

Les méthodes de décompositions approchées proposent l'utilisation d'une forme prédéfinie de cellules afin de caractériser l'environnement .Ces cellules représentent des sous-ensembles de l'espace des configurations et sont interconnectées afin de capturer la connectivité de l'environnement. Les cellules obstruées par une partie de C-Obstacles sont ensuite annotées et interdites à la navigation lors de la planification de chemin. Lors de l'utilisation de décompositions approchées, le rôle de la planification va être d'identifier une séquence de cellules interconnectées et appartenant à C-Free permettant de relier la position courante de l'agent à son but. Ces décompositions en cellules sont généralement effectuées à l'aide de grilles régulières 2D composées de cellules carrées. Les obstacles de l'environnement sont ensuite projetés dans cette grille permettant de marquer les cellules obstruées (voir Fig. 2.1a). La représentation de l'environnement sous la forme d'une grille régulière 2D permet de projeter les obstacles de l'environnement dans une bitmap 2D et de pouvoir ainsi profiter de la carte graphique afin d'accélérer la planification de chemin lors des requêtes. Au cours de cette planification, chaque cellule dans C-Free permet l'accès à ses 8 cellules avoisinantes si celles-ci appartiennent également à C-Free (4 cellules directement voisines, 4 cellules dans les diagonales). Cette exploration de la grille régulière peut également être améliorée en utilisant des heuristiques afin d'empêcher les changements de direction trop brusques de l'agent ce qui permet également de réduire l'espace de recherche. Afin de gérer des environnements plus complexes, certaines techniques proposent l'utilisation de plusieurs grilles régulières dans un même environnement afin de pouvoir représenter simultanément plusieurs capacités de déplacement. Ceci permet également de supprimer la contrainte 2D amenée par l'utilisation d'une unique grille régulière et de gérer en conséquence des environnements 3D avec différents étages .En se basant sur les caractéristiques de l'agent, cette approche permet, en plus, de localiser dans la grille régulière les accès existants entre des zones déconnectées de l'environnement, gérant ainsi la planification de chemin au sein d'environnements non connexes.**[11, 12, 13].**



(a) Grille régulière

(b) Grille régulière couplée à un quad-tree

Figure 2.1 – 2.1a : Grille régulière composée de cellules carrées.2.1b : Grille régulière couplée à un quad-tree afin de diminuer le nombre d'éléments pour représenter l'environnement. Les cellules grisées correspondent aux cellules annotées comme obstruées, les cellules blanches comme les cellules libres.

Les solutions proposées pour la modélisation de C-Space sous forme de grilles régulières requièrent de trouver un compromis entre la précision de la représentation utilisée, ou en d'autres termes la résolution de la grille, le coût de mémoire et le temps de planification. En effet, en utilisant une résolution plus fine permettant de décrire l'environnement précisément, l'information stockée en mémoire sera plus importante et le nombre d'éléments à considérer durant la planification sera également plus grand, abaissant dès lors les performances. D'un autre côté, si la résolution de la représentation n'est pas assez précise, il est possible de passer à côté de solutions existantes. D'autre part, les chemins générés ne seront pas de bonne qualité et la représentation de l'environnement sera assez éloignée de sa configuration réelle, générant ainsi des chemins peu réalistes. Plusieurs solutions alternatives ont été proposées afin de construire une décomposition approchée permettant de mieux gérer l'occupation mémoire. En se basant sur la structure des grilles régulières, il est, par exemple, proposé de définir une représentation hiérarchique permettant de représenter l'environnement avec plusieurs niveaux

de détails. L'utilisation d'une structure hiérarchique permet de réduire le nombre de cellules dans la grille en conservant de l'information détaillée là où il y en a besoin. Cette structuration hiérarchique est faite à l'aide d'un quadtree (ou d'un autre type de 2n -tree) structurant les données des cellules. Ce quadtree est une structure de donnée de type arbre dont les nœuds peuvent être soit des nœuds terminaux, soit divisés en 4 nœuds fils. L'espace des configurations est représenté dans un premier temps avec une résolution assez basse : chaque cellule est représentée par un nœud du quadtree. Les cellules peuvent se trouver dans l'un des trois cas suivant : (1) la cellule est entièrement incluse dans C-Free, (2) la cellule est entière incluse dans C-Obstacle,

(3) la cellule est à la frontière de C-Free et C-Obstacle et contient un sous ensemble de chaque. Dans les cas (1) et (2), la cellule contient toute l'information requise et n'a pas besoin d'être plus détaillée au niveau de la représentation. Dans le cas (3), il faut raffiner la représentation afin de caractériser plus précisément C-Free et C-Obstacle. Dans ce cas de figure, la cellule est divisée en 4 cellules filles de résolution équivalente et la même classification est faite entre les nouvelles cellules créées (voir Fig. 2.1b). L'utilisation d'un quadtree permet ainsi de réduire considérablement le nombre de cellules de la grille, et donc l'utilisation mémoire. Il est ainsi plus efficace de représenter de larges environnements ouverts en focalisant le raffinement sur les frontières entre C-Free et C-Obstacle

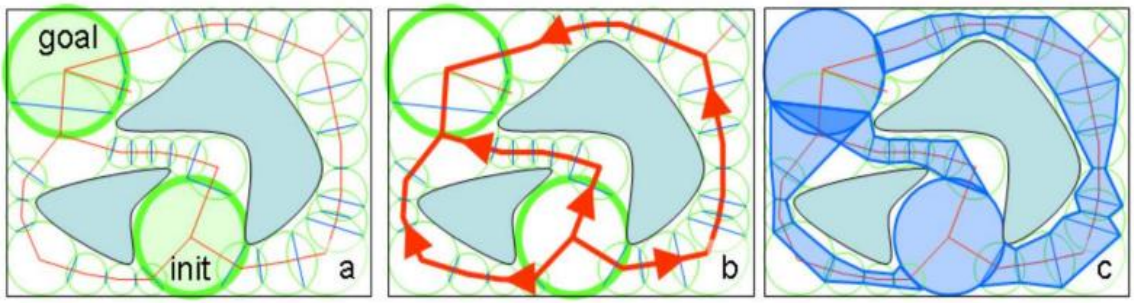


Figure 2.2 – Pettré [16] propose une décomposition de C-Free à l'aide de cylindre (a). Cette décomposition permet l'identification des principaux chemins de navigation (b) et également la création de couloirs de navigation permettant de générer des trajectoires plus variées pour gérer la navigation d'une foule d'agent.

Diverses approches n'utilisant pas de structures à base de grilles régulières ont également été proposées. Il a ainsi été, par exemple, proposé de représenter l'environnement navigable à l'aide d'une décomposition en cylindres de navigation **[16].** L'utilisation de ces cylindres lui

permet de créer une structure comprenant un nombre assez restreint d'éléments et permettant la navigation de plusieurs milliers d'agents en temps réel. Les axes médians de C-Free, c'est à dire les axes les plus éloignés des obstacles et présentant donc la plus grande sécurité au niveau de la navigation, sont tout d'abord identifiés. Des cylindres, centrés sur ces axes et dont le rayon correspond à la distance de l'obstacle le plus proche, sont définis le long de ces axes. Ces cylindres représentent donc des sous-espaces de C-Free où la navigation est possible. En connectant ensuite les cylindres qui s'intersectent, des couloirs de navigation où les entités peuvent naviguer librement sont créés (voir Fig.2.2). En identifiant les axes médians et l'éloignement maximum aux obstacles, cette méthode construit une représentation relativement proche d'un diagramme de Voronoï généralisé.

2.2.1.2 Décomposition exactes en cellules

À l'inverse des décompositions approchées, les décompositions exactes permettent de capturer entièrement la représentation de C-Free. Parmi les techniques utilisées on retrouve, par exemple, des décompositions en cellules verticales **[14]** ou des triangulations de Delaunay. Ces méthodes cherchent à caractériser précisément C-Free en utilisant les bordures des obstacles comme des contraintes formant les données d'entrées. Les méthodes basées sur les triangulations de Delaunay contraintes utilisent ainsi les frontières des obstacles de l'environnement afin de définir la triangulation **[17].** C-Free est donc représenté par un ensemble de triangles dont les arêtes libres représentent des connexions entre zones navigables et les arêtes contraintes des obstacles. La planification de chemin se ramène alors à identifier une séquence de triangles reliés par des bordures franchissables. En analysant de manière plus poussée la triangulation obtenue, certaines méthodes proposent d'extraire des informations supplémentaires sur l'environnement. L'analyse d'une triangulation de Delaunay contrainte peut par exemple permettre d'extraire les goulets d'étranglement présents dans un environnement **[18].** L'extraction de ces goulets a pour objectif de détecter précisément les endroits les plus contraints de l'environnement permettant d'utiliser cette information lors de la navigation de l'agent. D'un autre côté, un raffinement de la triangulation de Delaunay contrainte peut être utilisé afin de pouvoir planifier les chemins les plus courts pour des agents de tailles différentes en identifiant également les passages contraints de l'environnement ainsi que des couloirs de navigation **[15].**

Bien que la majorité des représentations exactes permettent une représentation d'un environnement 2D, certaines approches se sont intéressées à gérer des environnements 3D à l'aide de décompositions exactes **[19].** Une approche utilisant une subdivision exacte en 3D

incluant les contraintes de sol et de plafond permet par exemple d'ajouter des informations de hauteur à l'analyse de l'environnement. Cette subdivision, dite prismatique, permet ensuite de générer des cartes 2D interconnectées et d'augmenter automatiquement les cartes 2D d'information tridimensionnelle permettant entre autre de différencier les obstacles franchissables, tels que les marches, des obstacles infranchissables, tels que les murs (voir Fig.2.3).

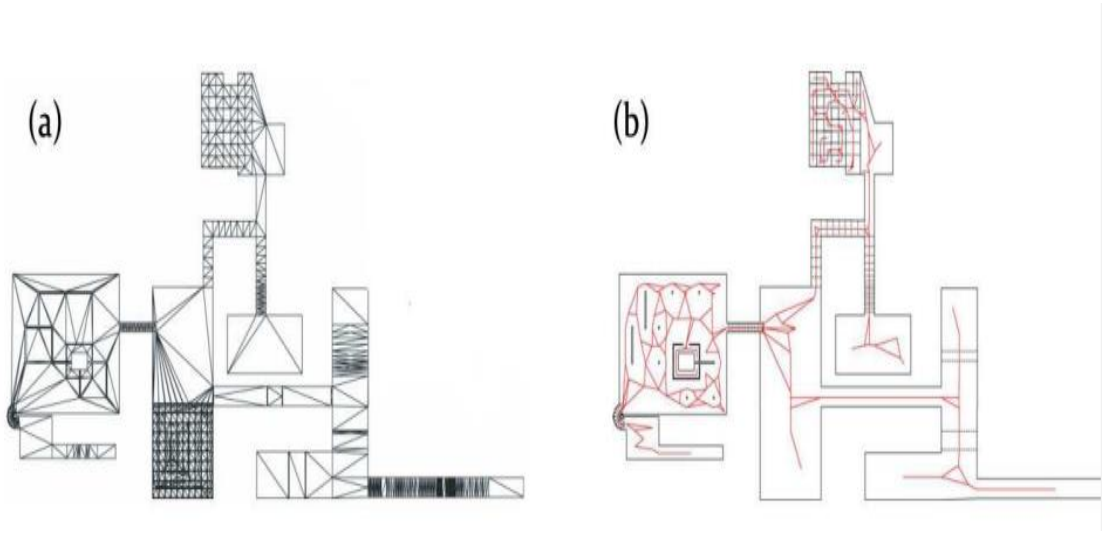


Figure 1.3 – Méthode de décomposition exacte proposée par Lamarche [19]. (a) : Représentation de l'environnement à l'aide de cellules 2.5D prenant en compte les contraintes de hauteur. (b) : Carte topologique de l'environnement prenant en compte les capacités de déplacement de l'agent. On peut notamment l'observer dans le carré supérieur où les déplacements de l'agent entre les différents carrés sont limités par les hauteurs de marche entre ces carrés.

Les méthodes de décompositions en cellules, exactes ou approchées, s'adressent principalement à des problèmes de planification de chemin de faibles dimensions. En effet, lorsque la dimension des espaces de recherche augmente, il est plus difficile de caractériser et de représenter les frontières entre C-Free et C-Obstacle. Dans le cas des méthodes exactes, les représentions de l'environnement deviennent plus difficiles et surtout beaucoup plus coûteuses en mémoire au-delà d'un espace de recherche en 2 dimensions. Pour les représentations approchées, elles deviennent difficilement utilisables au-delà de 4-5 dimensions. Les cartes de cheminement, que nous allons maintenant aborder, ont donc été conçues pour la planification de chemin dans des espaces de plus grandes dimensions

2.2.2 Cartes de cheminement probabilistes

Les cartes de cheminement (ou roadmap) permettent de représenter un sous-ensemble de C-Free en déterminant des configurations libres de l'environnement et en reliant ces configurations par des arcs de transition étant également contenus dans C-Free. En procédant ainsi les cartes de cheminement permettent une exploration beaucoup plus rapide de l'espace des configurations. Les différentes méthodes utilisant des cartes de cheminement vont donc s'intéresser principalement à déterminer des configurations libres de C-Free et à les connecter. Cela est fait en essayant de proposer la meilleure couverture possible sur l'ensemble de l'environnement afin de le représenter au mieux. Les techniques utilisant les cartes de cheminement se décomposent en deux familles : les méthodes à requêtes multiples (multiple-query) et les méthodes à requêtes simples (simple-query). Dans le premier cas, une exploration de l'ensemble de l'environnement est effectuée a priori et ré-utilisée ensuite à de multiples reprises pour la planification de plusieurs chemins. Dans le second cas, l'environnement est exploré seulement partiellement au moment de la demande de planification, chaque nouvelle requête entraînant une nouvelle exploration partielle de C-Free.

2.2.2.1 Méthodes à requêtes multiples

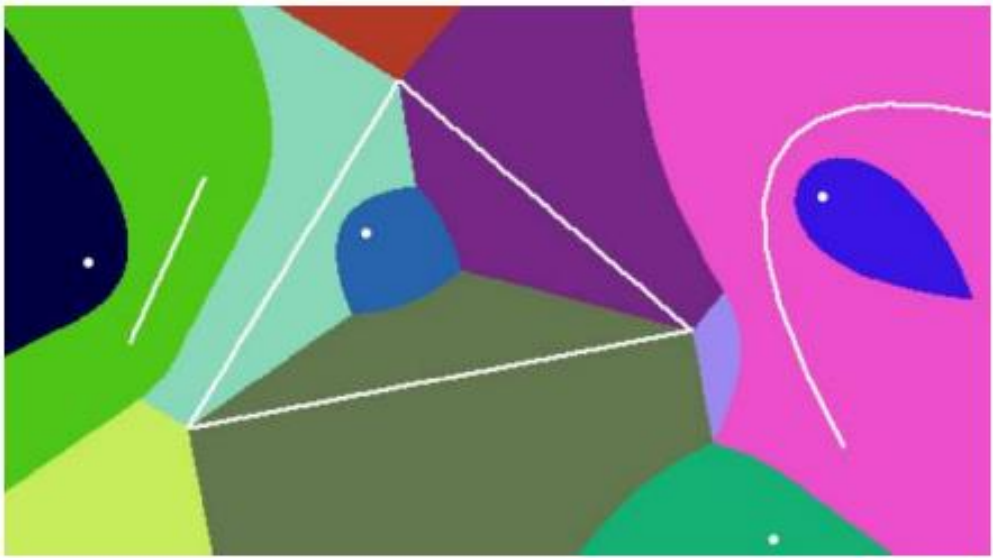
Les méthodes à requêtes multiples sont généralement composées de deux phases. Dans un premier temps, une phase de construction permet d'explorer l'espace des configurations et de capturer la topologie de l'environnement dans la carte de cheminement. Dans un second temps, la phase de requête permet d'explorer la carte de cheminement afin de calculer un chemin pour l'agent. La carte de cheminement représentant l'environnement est donc calculée au cours d'une phase de précalcul et utilisée au cours de la simulation pour répondre aux différentes tâches de planification de l'agent. Lors de la planification, la position courante de l'agent ainsi que sa destination sont rattachées aux cartes de cheminement. La construction des cartes de cheminement est en général basée soit sur des diagrammes de Voronoï généralisés soit sur des méthodes probabilistes. Il existe toutefois d'autres méthodes permettant de représenter l'environnement à l'aide de carte de cheminements, en les couplant par exemple avec des quad-tree **[20].**

Les diagrammes de Voronoï généralisés permettent de calculer les axes d'éloignement maximum des obstacles. En renseignant les frontières de C-Obstacle, le calcul d'un diagramme de Voronoï généralisé va en effet déterminer les ensembles de points les plus éloignés des obstacles en identifiant les bordures des cellules de Voronoï. L'utilisation de ces points pour la

création d'une carte de cheminement permet donc de générer des chemins présentant une bonne marge de sécurité vis-à-vis des obstacles et de limiter les chemins rasant les bordures de C-Free. Afin d'accélérer le calcul et le rendu de telles cartes de cheminement, certaines méthodes **[24]** utilisent des algorithmes de rendus issus de l'informatique graphique afin de représenter le diagramme de Voronoï obtenu dans un Z-buffer (voir Fig. 2.4a) grâce à une parallélisassions des calculs sur la carte graphique. La méthode des "cartes de couloirs", ou Corridor Maps, utilise aussi les diagrammes de Voronoï dans la construction de leur carte de cheminement **[25]** (voir Fig. 2.4b). En effet, les nœuds de la carte de cheminement sont des configurations qui sont éloignées au maximum des obstacles entourants. Ces configurations sont de plus annotées de leur distance d'éloignement aux obstacles ce qui permet ainsi de leur attribuer un rayon de sécurité permettant de définir un cercle autour de cette configuration. L'ensemble des configurations et des chemins qui sont contenus au sein de ce cercle sont donc par définition libres de toute collision avec l'environnement. En joignant l'ensemble des configurations de la carte de cheminement et en faisant l'union de leurs cercles de sécurité respectifs, on obtient donc des couloirs de navigation au sein desquels l'agent va pouvoir naviguer librement. Les chemins générés entre les configurations de la carte sont ensuite lissés lors de la navigation afin d'obtenir des trajectoires plus naturelles mais restant toujours à l'intérieur des couloirs de navigation (voir Fig.2.4c).

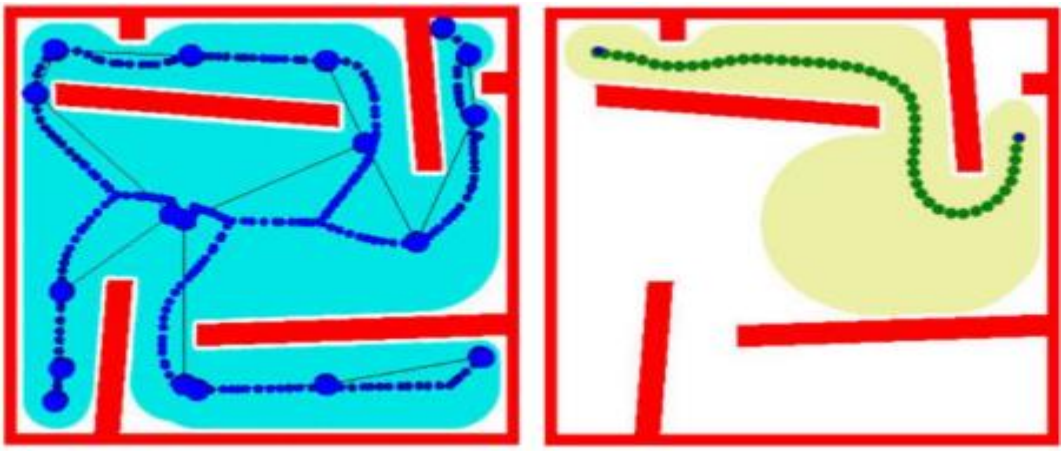
Parmi les méthodes probabilistes à requêtes multiples, l'une des plus utilisée est sans doute celle des cartes de cheminements probabilistes, ou Probabilistic RoadMaps (PRM). Les PRMs ont été introduites simultanément par Kavraki **[22]** et Overmars **[21]** avant de faire l'objet d'une étude commune de leur part **[23].** Pendant la phase de construction de la carte de cheminement, l'exploration de l'espace des configurations est faite de manière aléatoire en tirant des configurations au hasard. Cet échantillonnage tend à suivre généralement une distribution uniforme des échantillons mais de nombreuses méthodes proposent l'utilisation d'autres lois de répartitions en fonction de l'environnement étudié. Un algorithme de détection de collision est utilisé pendant l'échantillonnage afin de déterminer si les configurations sont valides ou si elles sont en collision avec C-Obstacle. Une fois l'échantillonnage terminé, un planificateur local est utilisé afin de connecter les configurations valides à leurs voisins. Si ceux-ci répondent à des critères de proximité tels que les k plus proches voisins, ou que les configurations sont à l'intérieur d'une sphère de proximité, alors ils sont connectés. Si le chemin qui permet de les relier reste dans C-Free alors il est conservé dans la carte de cheminement (voir Fig.2.5a). Une fois la carte de cheminement créée, la planification de chemin se fait simplement par l'insertion

des configurations de départ et d'arrivée dans la carte de cheminement et par l'utilisation d'un algorithme tel qu'un Dijkstra ou un A* afin de trouver le chemin.



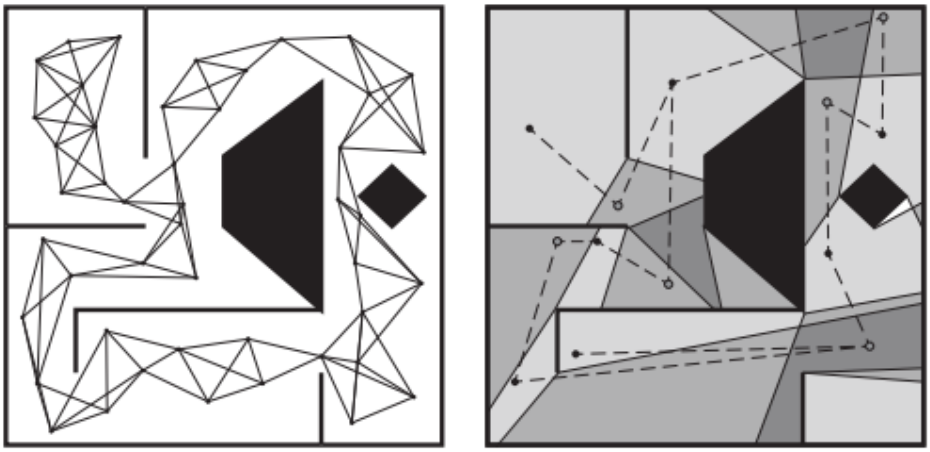
(

(a) Représentation utilisant des cellules de Voronoï



(b)Corridor Maps (c) Planification utilisant les Corridor Map

Figure 2.4 – 2.4a : En utilisant les bordures des obstacles (en blanc) comme contraintes dans l'environnement, HoffIII propose une représentation de l'environnement dans des bitmaps en utilisant des cellules de Voronoï. 2.4b et2.4c : Geraerts utilise les points les plus éloignés des obstacles afin de construire une carte de cheminement (bleu foncé) et associe également à la carte des couloirs de sécurité (bleu clair). Un chemin (vert) est ensuite identifié puis lissé tout en restant éloigné des obstacles.



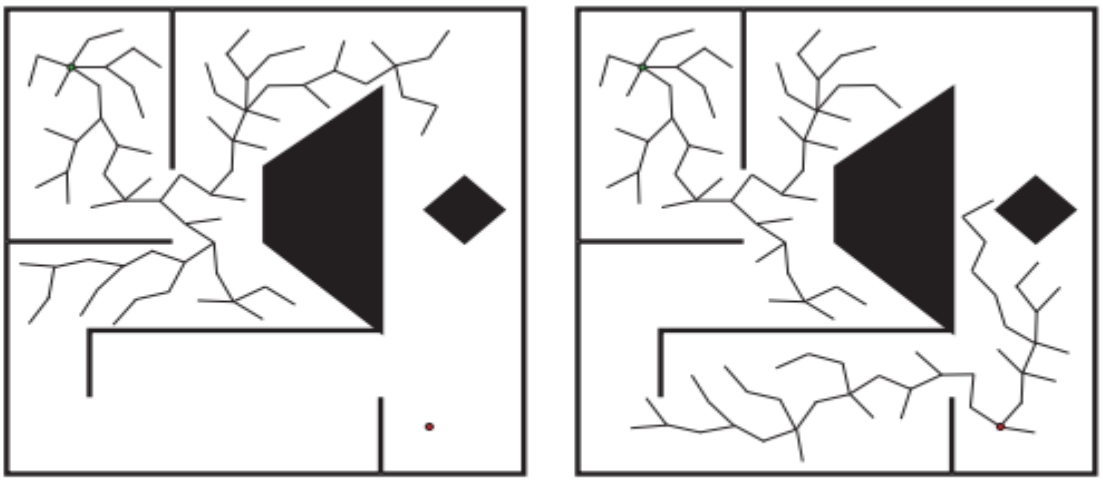
(a) Probabilistic RoadMap (PRM) (b) Visibility PRM

Figure 2.5 – 2.5a : Le premier environnement montre une carte de cheminement probabiliste où chaque nœud est connecté à ses k plus proches voisins (k = 4). 2.5b : Visibility PRM. Les nœuds noirs sont des gardiens et les cercles gris des connecteurs. Les zones gris clair sont observées par un gardien, les gris moyen par deux gardiens et les plus sombres par 3 gardiens. La zone blanche n'est observée par aucun gardien mais la couverture de l'environnement est considérée comme suffisante

2.2.3 Méthodes à requêtes simples

Contrairement aux méthodes à requêtes multiples qui cherchent à construire une représentation de l'environnement dans son ensemble, les méthodes à requêtes simples proposent d'explorer et de construire la représentation d'un sous-ensemble de l'environnement lorsqu'une requête de planification est formulée. Ces méthodes favorisent donc la création d'une structure par requête plutôt que la construction d'une structure globale plus lourde (notamment en mémoire) mais utilisée pour l'ensemble des requêtes de planification. L'une des structures les plus utilisée s'appelle les Rapidly-exploring Random Trees (RRT) dont le but est d'explorer l'espace de configuration à l'aide d'un arbre de recherche **[15].** La racine de l'arbre d'exploration est fixée au niveau de la configuration courante de l'entité et des échantillons sont ensuite tirés aléatoirement dans l'environnement, entrainant la croissance de l'arbre d'exploration (voir Fig. 2.6a). En limitant l'extension maximale de l'arbre à chaque pas d'exploration, la croissance de l'arbre est biaisée et tend à explorer en priorité les zones de l'environnement qui n'ont pas été explorées jusqu'alors, ce qui permet de générer rapidement un arbre possédant une bonne couverture de l'espace de recherche. L'exploration de l'environnement s'arrête lorsque la destination de l'agent est atteinte par l'arbre. Contrairement aux PRMs, cet arbre de par sa construction permet de conserver une structure d'exploration qui est toujours connexe et toujours reliée à la configuration courante de l'entité. Afin d'améliorer les performances des RRTs et de connecter plus rapidement la configuration de départ à la configuration cible de l'agent lors de la planification de

chemin, une amélioration de l'algorithme de base propose l'utilisation simultanée de deux arbres de recherche pour explorer l'environnement dans une méthode appelée RRT-connect **[28].** La racine du premier arbre est fixée comme précédemment à la configuration courante de l'entité, tandis que la racine du second est rattachée à la configuration cible. Les deux arbres sont étendus l'un après l'autre et, après chaque phase d'extension, l'algorithme essaie de voir s'il est possible de trouver une connexion directe des deux arbres afin d'accélérer la recherche (voir Fig.2.6b). Bien que les méthodes basées sur les RRT produisent des résultats rapides, la solution générée n'est pas optimale. Afin de palier à cela, certaines méthodes proposent en se basant sur les RRTs de converger vers des solutions optimales **[29]** ou encore d'utiliser un paramètre faisant un compromis lors de la planification entre continuer l'exploration de l'environnement ou raffiner la représentation existante en ajoutant de nouveaux arcs de meilleure qualité à la représentation courante **[30].** L'utilisation de ces arbres de recherche permet tout d'abord de trouver des chemins dans des environnements très contraints sans pré-calculs, ni paramètres à ajuster par l'utilisateur. De plus ces algorithmes permettent l'exploration, et donc la planification de chemin, dans des environnements de très grandes dimensions et sont donc très adaptés à des problèmes de planification avec de nombreux degrés de liberté.



(a) Rapidly-exploring Random Trees (RRT)

(b) RRT Connect

Figure 2.6 :2.6a : Exploration de l'environnement à l'aide d'un Rapidly-exploring Random Tree. La position courante de l'agent est le point en haut à gauche de l'environnement et la cible à atteindre, le point en bas à droite. 2.6b : Utilisation d'un RRT-Connect. Un arbre de recherche est attaché à la position courante de l'agent tandis qu'un second est rattaché à la cible à atteindre. Les deux arbres de recherches explorent l'environnement de manière alternative.

Les solutions abordées jusqu'ici ont été proposées pour des environnements statiques. Puisque l'environnement n'est pas modifié lors de la simulation de nombreux pré-calculs

peuvent être effectués. Cela simplifie a posteriori la tâche de planification. Toutefois, les environnements où sont amenés à naviguer des robots ou des agents virtuels, sont rarement entièrement statiques. Il convient donc maintenant de s'intéresser, et de s'interroger, sur la manière de prendre en compte des modifications pouvant arriver au cours de la simulation. En considérant avoir des objets qui peuvent bouger, ou être déplacés dans l'environnement, comment est-il possible de planifier un chemin ? Comment prendre en compte ces changements dans l'espace des configurations afin de mettre à jour les structures de données ? Et, enfin, comment est-il possible de remettre à jour un chemin planifié si une obstruction qui n'était pas présente précédemment est subitement détectée ? Les méthodes qui ont été introduites dans cette section ne présentent pas de solutions permettant de gérer ces différents aspects. Toutefois, les majeures parties des environnements considérés vont être composés d'une large composante statique. En partant de cette observation il va donc être possible de s'appuyer sur des méthodes de planification prévues pour des environnements statiques puis de les adapter afin de pouvoir gérer des environnements présentant des aspects dynamiques. Nous nous intéresserons dans la section suivante à présenter les algorithmes proposés pour gérer la planification de chemin au sein de tels environnements.

2.3 Planification de chemin en environnements dynamiques

Afin de pouvoir gérer une plus grande variété de problèmes, certaines méthodes se sont adressées à la représentation d'environnement pouvant être peuplés par des éléments dynamiques. Ces éléments dynamiques vont augmenter la complexité du problème de planification de chemin en lui ajoutant un aspect temporel qui n'était pas présent précédemment. En effet, en considérant l'environnement à des temps différents il est possible que celui-ci ait évolué du fait de l'ajout, la suppression ou le déplacement de certains de ses éléments. L'espace des configurations qui lui est associé doit donc être mis à jour en conséquence afin de maintenir une représentation de l'environnement permettant la planification de chemin. Alors que les pré-calculs étaient possibles pour appréhender la totalité des environnements statiques vus précédemment, ils sont plus limités dans le cas présent puisque la configuration de l'environnement va évoluer au cours de la simulation. Toutefois, les environnements représentés sont généralement composés d'une grande partie statique dans laquelle sont insérés des éléments dynamiques. Différentes approches ont donc été proposées pour permettre la planification de chemins au sein de tels environnements. Certaines proposent de calculer des chemins dans une représentation de l'environnement et de replanifier ensuite

ces chemins lorsqu'ils sont invalidés par la présence d'objets dynamiques. D'autres approches proposent de planifier une trajectoire qui intègre directement au cours de la planification l'aspect temporel de l'environnement. Au cours de cette thèse, une trajectoire sert à désigner un chemin dans l'espace spatio-temporel. Une trajectoire, contrairement à un chemin, va donc chercher à anticiper les déplacements des objets dynamiques dans le domaine spatio-temporel pour proposer une solution de planification prenant l'aspect temporel en considération. Nous verrons ainsi dans un premier temps, les méthodes proposant de mettre à jour la représentation de l'environnement seulement au moment de la planification. Un chemin est ensuite planifié dans cette structure mise à jour et une nouvelle planification permet ensuite la mise à jour de ce chemin si une collision avec un élément de l'environnement est détectée lors de la navigation de l'agent. Dans un second temps, d'autres approches proposent de maintenir la représentation de l'espace des configurations à jour lorsque les changements sont détectés permettant ainsi d'avoir directement la structure adéquate lorsqu'une requête de planification de chemin est effectuée. Enfin, une troisième famille de méthodes propose de planifier directement dans l'espace spatio-temporel. Alors que les méthodes précédentes se focalisent sur le calcul d'un chemin dans l'environnement à un temps donné, celles-ci cherchent à planifier directement une trajectoire dans le domaine espace-temps en anticipant du mieux possible les déplacements des éléments dynamiques.

2.3.1 Adaptation ponctuelle de la représentation de l'environnement pour planification de chemin

Les Rapidly-exploring Random Trees (RRTs), initialement utilisés dans le cadre des environnements statiques, présentent l'avantage de ne pas nécessiter de pré-calculs de l'environnement. Cette propriété les rend très attrayants pour les environnements présentant des aspects dynamiques dont les pré-calculs sont compliqués par une topologie constamment changeante et généralement non connue à l'avance. De nombreuses extensions des RRTs ont ainsi été proposées afin de gérer la navigation d'agents dans des environnements comportant des éléments dynamiques.

Les RRTs permettent d'explorer une partie de l'environnement lorsque des requêtes de planification de chemin sont formulées. Au sein d'environnements dynamiques, l'agent va être amené à devoir replanifier son chemin lorsque des obstacles sont détectés le long du chemin originellement défini. Une solution simple est d'utiliser un RRT afin de planifier de nouveau le déplacement de l'agent. Toutefois, dans ce cas, l'information apprise lors de l'exploration de

l'environnement par le RRT ayant permis de calculer le premier chemin n'est pas conservée et donc pas réutilisée lors de la replanification.

Ainsi, lorsqu'un chemin calculé par un RRT est altéré par la présence d'un obstacle, plusieurs méthodes partent de l'hypothèse qu'un chemin alternatif peut être retrouvé dans un voisinage proche du chemin original. En partant de cette observation, ces méthodes proposent donc de réutiliser le RRT calculé précédemment afin d'orienter la nouvelle planification de chemin en biaisant le nouvel échantillonnage des configurations pour la construction du nouvel arbre de recherche. L'utilisation des informations contenues dans l'arbre de recherche va ainsi permettre d'optimiser la recherche en utilisant le RRT précédent comment un heuristique lors de la recherche. Parmi ces méthodes on peut notamment distinguer les Exécution Entente RRT (ERRT) **[32]** ainsi que les Dynamics RRT (DRRT) **[31].** Lors de la génération d'un trajet valide, les ERRT proposent de sauvegarder les configurations appartenant au chemin. Lorsqu'une mise à jour de la trajectoire est nécessaire, les points appartenant au chemin original ou le but visé par l'agent ont alors une certaine probabilité d'être utilisés comme échantillons lors de la construction du nouveau RRT (voir Fig. 2.7e). En biaisant ainsi l'échantillonnage des configurations, les ERRT permettent d'obtenir rapidement une nouvelle solution restant proche de la planification précédente. Les DRRT se basent sur le principe de la continuité spatiale et temporelle et proposent de vérifier la validité des branches présentes dans le RRT précédent (voir Fig. 2.7c). Les branches valides et connectées à la racine sont donc conservées. Cette méthode, bien qu'ayant un léger surcoût à l'initialisation du nouveau RRT, permet également de trouver une solution généralement assez proche de la solution d'origine. De plus, en plaçant la racine des DRRT au niveau du but et non de la position courante de l'agent, cette méthode permet également d'éviter de recalculer entièrement l'arbre de recherche lorsque l'agent s'est déplacé. Enfin, les Multipartite-RRTs utilisent un principe similaire afin de guider la nouvelle planification. Dans un premier temps, le chemin de l'agent est planifié à l'aide de RRTs. Lorsque des replanifications sont nécessaires, l'échantillonnage des nouvelles configurations est alors biaisé en utilisant les nœuds des sous-arbres encore valides dans le RRT précédent comme des candidats à l'extension du nouvel arbre d'exploration [33] (voir Fig. 2.7d).

Réutiliser les RRTs calculés précédemment lorsqu'il est nécessaire de replanifier un chemin permet d'accélérer les nouvelles planifications de l'agent lorsqu'un obstacle est détecté dans le chemin. Afin d'étendre cette idée, certaines méthodes proposent de conserver au sein d'une structure l'ensemble des RRTs déjà calculés pour pouvoir s'en servir ultérieurement lorsque de nouvelles requêtes de planification sont formulées. Alors que ces RRTs étaient

précédemment conservés afin de biaiser la nouvelle planification, ces méthodes proposent de les stocker dans une structure et de construire ainsi une "forêt" de RRT permettant de créer, au fil des planifications successives, une représentation de plus en plus complète de l'espace des configurations. Lors des planifications de chemin, une solution est donc d'abord recherchée dans la forêt à disposition et si elle n'est pas trouvée un nouvel RRT est en général calculé et ajouté à la forêt afin de compléter la connaissance de l'environnement. Puisque nous nous plaçons dans le contexte

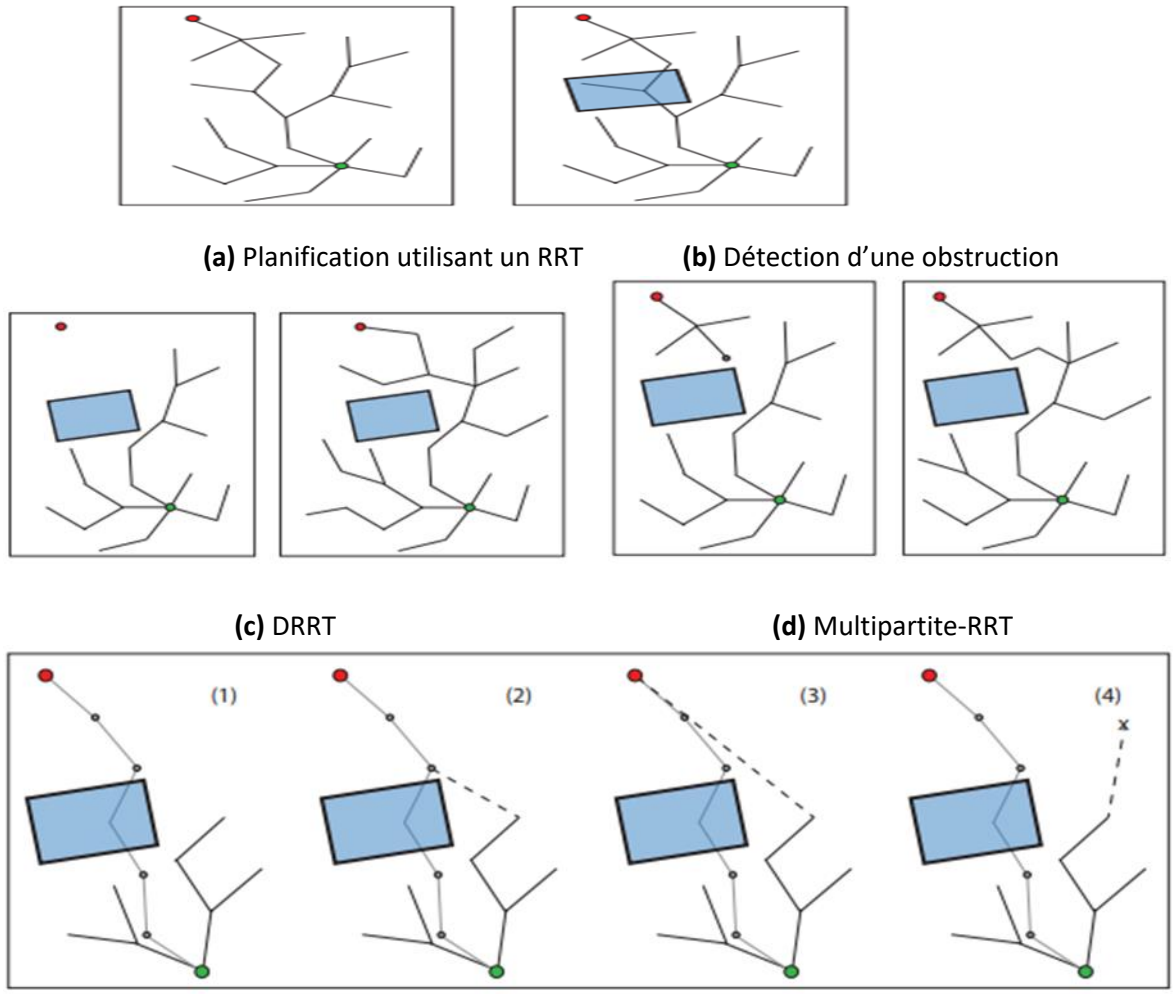


Figure 2.7 (e) ERRT : Le nouvel échantillonnage a une probabilité de tirer soit une configuration de L'ancienne planification (2), soit la cible de la planification (3), soit une nouvelle configuration(4).

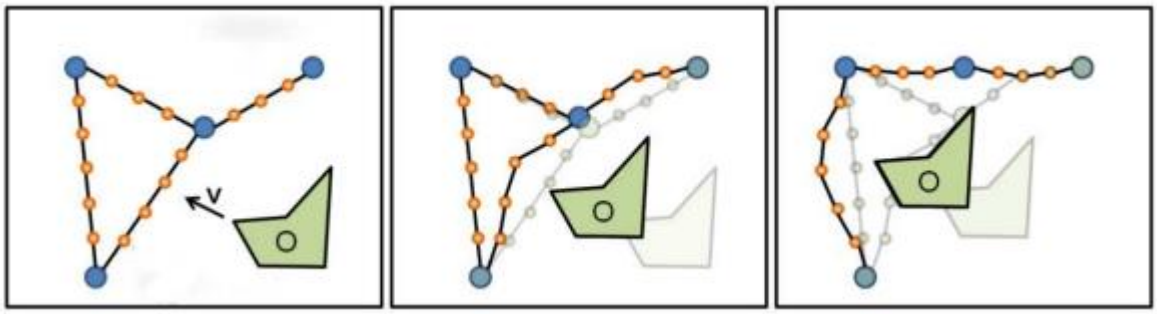
D'environnements dynamiques où les éléments peuvent être ajoutés, déplacés ou enlevés, la configuration de l'environnement va être amenée à évoluer. L'utilisation de RRTs permet donc d'explorer rapidement les parties de l'espace des configurations qui ont été libérées mais également de supprimer de la "forêt" les arbres qui ont été invalidés par l'ajout d'un obstacle, ce qui permet de prendre en compte toutes les modifications observées. Les Reconfigurable

Random Forest (RRF) permettent ainsi de conserver les arbres calculés lors des premières planifications et de simplifier de temps en temps la structure, en supprimant les nœuds redondants pour ne garder qu'une représentation compacte de l'environnement [34]. Lors de chaque planification de chemin, une solution est d'abord recherchée dans la forêt de RRT existante et, si elle n'est pas trouvée, de nouveaux arbres sont alors calculés et ajoutés à la représentation. Le concept introduit par les RRFs est ensuite étendu dans l'utilisation des Lazy Reconfigurable Forest (LRFs) **[35].** La forêt est alors construite en explorant l'environnement à l'aide de plusieurs RRTs simultanément et en tentant de les inter-connecter. L'agent de son côté navigue dans l'environnement en adoptant trois comportements selon sa situation courante. Si une solution pour atteindre le but est trouvée dans les LRFs, alors l'agent suit en priorité ce chemin. Si ce chemin n'existe pas ou qu'il est rendu invalide par la présence d'obstacles dynamiques, l'agent avance sur un chemin qui le rapproche du but, même s'il ne l'atteint pas. Pendant ce temps, les RRTs continuent d'explorer l'environnement jusqu'à pouvoir fournir une nouvelle solution à la planification. Enfin, s'il n'existe pas de chemin atteignant ou se rapprochant du but de la planification, l'entité peut également simplement attendre. Cette boucle perception/action permet notamment à l'agent de commencer à se déplacer localement vers son but avant même qu'une solution globale ne soit proposée.

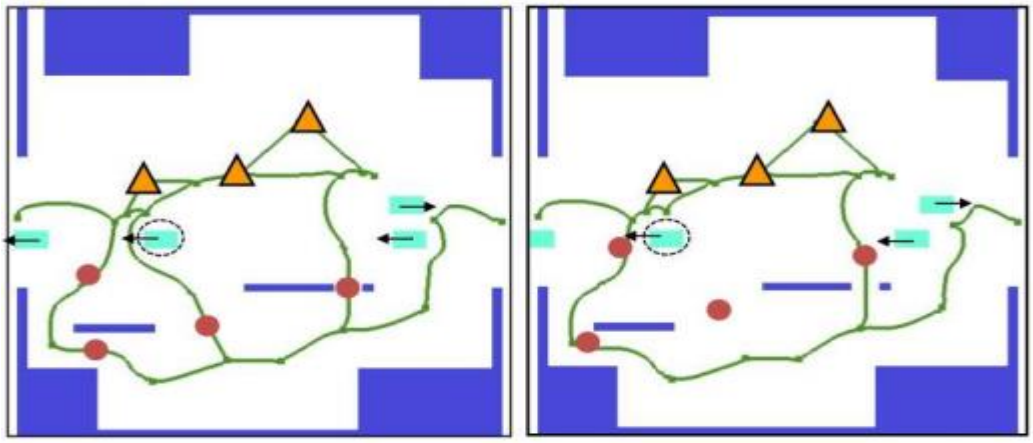
2.3.2 Adaptation dynamique de la représentation de l'environnement pour la planification de chemine

Les techniques présentées dans la partie précédente utilisent une représentation statique de l'environnement et cherchent ensuite à y planifier un chemin. Les obstacles dynamiques sont évités dans un second temps en planifiant, de nouveau, un chemin lorsqu'une obstruction est détectée. D'autres méthodes, s'intéressent à tenir compte de la dynamique de l'environnement directement au niveau de la représentation de l'environnement. Les cartes de cheminement générées par ces méthodes sont donc dynamiques et sont modifiées directement au fil de la simulation par le déplacement des éléments de l'environnement. Grâce aux mises à jour permanentes, la représentation de l'environnement reste ainsi correcte au cours de temps et les chemins planifiés sont également adaptés aux contraintes créées par la présence des obstacles dynamiques. En effet, la modification de la carte de cheminement va permettre d'adapter directement le chemin assigné à un agent en le déformant de la même manière. Il sera toutefois nécessaire de replanifier parfois un nouveau chemin lorsque des liens de la carte de cheminement seront, par exemple, rompus à cause de la proximité dangereuse d'un obstacle.

La dynamique des obstacles entraine des changements constants dans la configuration de l'environnement. Afin de gérer ces changements, certaines méthodes proposent de mettre à jour, au cours, de la simulation les cartes de cheminement associées à l'environnement. Celles-ci sont généralement qualifiées de cartes de cheminement "déformables" car elles réagissent aux déplacements des obstacles de l'environnement, repoussant les nœuds de ces cartes de cheminement et déformant les arcs existants. Ces cartes de cheminement sont généralement construites de manière à compter un nombre de nœuds restreint, ce qui permet une mise à jour plus rapide de la structure. D'autre part, ces représentations sont généralement associées à des systèmes de particules. L'avantage d'un tel système est que chaque élément de l'environnement, ainsi que chaque nœud de la carte, peut générer des forces d'attraction, ou de répulsion, sur la carte de cheminement. Les forces d'attraction permettent en général de représenter les forces internes entre les éléments de la carte de cheminement, ce qui permet de conserver la connectivité entre les nœuds. Les forces de répulsion, à l'opposé, permettent de représenter les forces externes exercées par les objets dynamiques, ou les autres agents, traduisant ainsi la volonté de trouver un chemin loin de potentiels obstacles. De plus, les liens de la carte peuvent également être rompus lorsque les forces externes exercées sont trop fortes, ce qui peut créer des déconnexions. Les Reactive Deforming Roadmaps (RDR) sont, par exemple, composées de nœuds, appelés "balises", et de liens qui sont tous deux gérés par un système de particule (voir Fig.2.8b). Cette carte est déformée en fonction des forces internes et externes qu'elle subit. Les liens sont ensuite cassés lorsque des conditions maximales d'élasticité sont atteintes. De nouveaux liens ou nœuds sont échantillonnés au besoin pendant la simulation, afin de garder une bonne représentation de l'environnement. L'utilisation de ce système permet de gérer la planification de chemin au sein d'un environnement avec des obstacles dynamiques, sans qu'aucune connaissance a priori sur les mouvements des obstacles ne soit nécessaire



(a) Reactive Deforming Roadmaps (RDR)



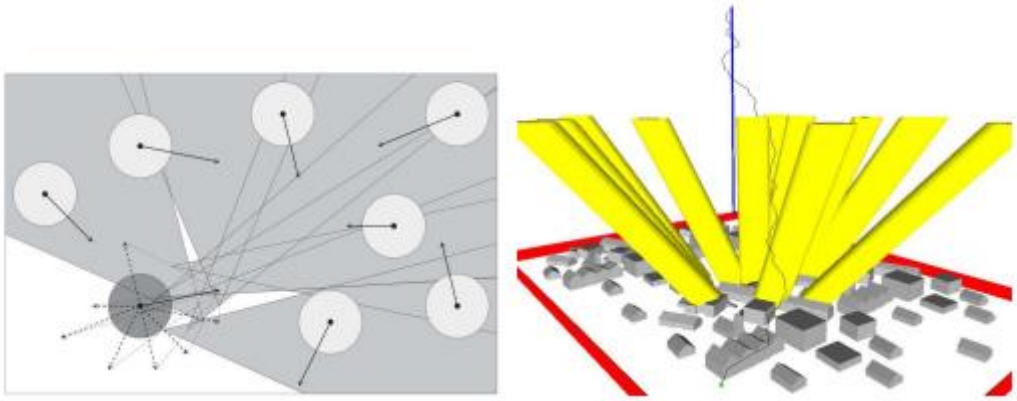
(**b)** Adaptative Elastic ROadmaps (AERO

Figure 2.8 – Les Reactive Deforming Roadmaps 2.8a ainsi que les Adaptative Elastic ROadmaps 2.8b sont des cartes de cheminement dont la structure est mise à jour au file de la simulation par le déplacement des obstacles

Une seconde alternative pour gérer les obstacles dynamiques d'un environnement est de créer une carte de cheminement basée sur des diagrammes de Voronoï généralisés. Celle-ci est ensuite remise à jour lorsque des obstacles sont détectés. Les diagrammes de Voronoï généralisés permettent de déterminer les points les plus éloignés des obstacles. Une carte de cheminement mise à jour de manière dynamique en utilisant ces diagrammes permet donc de déterminer directement des chemins présentant une marge de sécurité maximale vis-à-vis des obstacles de l'environnement. Plusieurs techniques telles que les Adaptative Elastic ROadmaps (AERO) **[37]** ou encore les graphes de navigation multi-agents (MaNG) **[36]** se basent sur ces techniques pour gérer la navigation de foules d'agents au sein d'environnements dynamiques. AERO propose d'utiliser ainsi des couloirs de navigation associés aux cartes de cheminement élastiques, afin de relâcher les contraintes sur la navigation des agents, et leur permettre

d'exploiter au maximum l'espace libre à leur disposition dans l'environnement (voir Fig.2.8a). MaNG utilise des rendus sur cartes graphiques afin de calculer les diagrammes de Voronoï du premier et du second ordre et d'utiliser l'union de ces deux diagrammes pour la navigation. Cette méthode utilise également des forces d'attraction répulsion entre les agents durant la navigation afin de créer des comportements de groupe lors des déplacements de ces derniers.

2.1.1. Planification de trajectoire



(a) Vélocité d'Obstacle

(b) Extrapolation des obstacles dans Les domaines spatio-temporels

Figure 2.9 – 2.9a : La représentation des obstacles à l'aide de Vélocité Obstacles permet d'identifier les différentes vitesses et directions de l'agent permettant de naviguer sans collisions dans l'environnement. 2.9b : Les déplacements des obstacles peuvent être extrapolés dans le domaine spatio-temporel (jaune) ce qui permet de planifier une trajectoire valide (noire) afin de les éviter.

Les techniques présentées jusqu'alors se focalisent sur le calcul de chemin dans l'environnement et sur des méthodes de replanification rapide lorsqu'un obstacle est détecté, ou sur une mise à jour au fil de la simulation de la carte de cheminement. Ces approches, toutefois, subissent la dynamique de l'environnement et s'y adaptent pour proposer des solutions de navigation. Certaines méthodes proposent également de gérer la dynamique de l'environnement en planifiant dans le domaine spatio-temporel. Augmenter la représentation géométrique de l'environnement par la dimension temporelle permet notamment d'anticiper le déplacement des éléments de l'environnement pour pouvoir en tenir compte lors de la planification. En considérant une représentation spatio-temporelle, la planification permet de calculer une trajectoire pour l'agent puisque l'information apportée par la dimension temporelle sera également incluse dans la planification. De plus cette planification permet de mieux

prendre en compte l'aspect dynamique de l'environnement, dès la recherche de chemin, et de réduire ainsi les besoins de replanification a posteriori durant la navigation.

Certaines techniques se basent ainsi sur un arbre de recherche, afin d'explorer l'environnement. Chaque nœud de l'arbre correspond à la représentation, à intervalles réguliers, de configurations valides de l'agent dans l'environnement. L'identification d'une séquence de configuration, lors de la planification, permet donc de déterminer la trajectoire que l'agent doit emprunter. L'utilisation de cet arbre de recherche permet également d'identifier, à court terme, les manœuvres à effectuer afin d'éviter les obstacles dynamiques présents. Ces techniques sont notamment basées sur l'utilisation des vélocité Obstacles **[39]** ou de leur extension, les Vélocité Réciproque Obstacles **[40].** L'agent et les obstacles sont alors représentés dans l'espace de configuration par des cercles possédant un centre, un encombrement et une vitesse de déplacement. En regardant les vitesses relatives entre l'agent et les obstacles dynamiques, des cônes de vitesse sont extraits. Ces cônes de vitesse permettent de déterminer s'il y aura une collision entre l'agent et l'obstacle en vérifiant si l'extrémité du vecteur de déplacement de l'agent arrive dans le cône associé à l'obstacle. Ces cônes sont appelés les Vélocité Obstacles (voir Fig. 2.9a). En utilisant des heuristiques de recherche dans l'arbre, la trajectoire calculée satisfait une liste d'objectifs prioritaires tels qu'atteindre le but, maximiser la vitesse de déplacement ou conserver la structure de la trajectoire initiale. De plus, la vitesse et la direction de déplacement de l'agent sont également adaptées afin de rester en dehors de tous les Vélocité Obstacles et donc d'éviter les collisions avec les obstacles environnant. Dans le cadre des Réciproqua Vélocité Obstacles, les agents font l'hypothèse supplémentaire que les autres agents naviguent également de façon réactive dans l'environnement, ce qui permet de réduire les oscillations et de créer des trajectoires plus lisses. **[38]**

Une alternative existe au calcul de configurations finies de l'agent dans son environnement par la construction d'un arbre de recherche. Cette solution consiste à utiliser des cartes de cheminement et à anticiper les déplacements des obstacles durant la planification. Certaines solutions, utilisant cette approche, permettent notamment de simplifier le problème, en considérant que les déplacements des obstacles sont entièrement connus a priori. Certaines techniques utilisent une planification à deux étages **[38].** Cela permet, dans un premier temps, d'identifier des sous-buts locaux appartenant à la trajectoire globale puis, ensuite, de calculer les mouvements de l'agent permettant de naviguer entre ces sous-buts en s'adaptant aux contraintes dynamiques de l'environnement. La planification locale est ensuite effectuée également à l'aide d'un arbre de recherche. Toutefois, cette solution a ses limites. Si les

changements dans l'environnement local arrivent de manière imprévue une fois que cette planification locale a eu lieu l'agent ne peut pas adapter sa trajectoire à ces changements inopinés.

Afin de gérer la planification de trajectoire pour un agent au sein d'environnements dynamiques où les mouvements des obstacles peuvent être définis, il est possible d'utiliser des intervalles de sécurité afin de déterminer des configurations où l'agent peut attendre durant sa navigation **[41].** Une autre alternative consiste à représenter les obstacles de l'environnement sous la forme de volumes spatio-temporels La représentation de ces obstacles va être définie en fonction des connaissances disponibles sur leurs déplacements. Ainsi, un obstacle, dont la trajectoire est entièrement connue a priori, sera représenté par une extrusion de sa géométrie le long de sa trajectoire au cours du temps. À l'inverse, un obstacle dont la trajectoire est inconnue aura un volume spatio-temporel qui sera borné par son déplacement maximum dans le temps. La méthode proposée utilise une PRM, afin de capturer la topologie de la composante statique de l'environnement. La planification est effectuée dans cette PRM en utilisant les volumes spatio-temporels afin de calculer une trajectoire évitant les obstacles dynamiques (voir Fig. 2.9b). Lors de la navigation, l'information concernant les trajectoires des obstacles dynamiques se raffine et la trajectoire de l'agent est donc mise à jour en utilisant un algorithme de type D* **[42].**

2.4 Conclusion

De nombreuses solutions ont été proposées pour la planification de chemin au sein d'environnements virtuels. Dans Ce chapitre nous avons démontré la navigation au sein d'environnements statiques et à l'environnement dynamique .La problématique de la planification de chemin s'est étendue à des environnements peuplés d'obstacles dynamiques. Au sein de tels environnements, l'agent doit être capable de détecter les obstacles qui se sont déplacés, et d'adapter son chemin. Cette approche rend le problème de navigation plus complexe, mais également plus intéressant, en élargissant les possibilités d'applications. Certaines techniques se sont basées sur une représentation de l'environnement statique couplée à des méthodes de replanification. Cela permet de réparer les chemins des agents invalidés lors de déplacements d'obstacles.

Chapitre03 :

Conception de système

3 Introduction

Après l'étude détaillée des représentations topologiques de l'environnement et la planification de chemin et ces algorithmes dans les sections précédentes, dans ce chapitre en va mettre la conception globale et la conception détailler de notre model (system) afin faire une solution concernant notre problème de la planification de chemin en temps réel basé sur RRT*(star) en évitent les obstacles et les collisions avec des agents dynamiques.

3.1 Objectif

L'objectif principal de notre travail est l'implémentation d'un algorithme de planification de chemin en temps réel dans un environnement dynamique tel qu'un jeu vidéo. Nous utilisons une approche d'échantillonnage en temps réel basée sur l'algorithme RRT (Rapidly Exploring Random Tree) qui a connu un grand succès en robotique.

Plus précisément, notre algorithme est basé sur l'algorithme RRT*. Nous apportons notre contribution en introduisant une stratégie de reconnexion de l'arbre qui permet à la racine de l'arbre de se déplacer avec l'agent sans rejeter les chemins précédemment échantillonnés.

Notre méthode n'a pas besoin d'attendre que l'arbre soit entièrement construit, puisque l'extension de l'arbre et les actions sont intercalées.

Donc, il s'agit de la variante temps réel de RRT*.

3.2 Conception de notre système

3.2.1 Conception général

Dans cette partie, on décrit l'architecture globale de notre système.

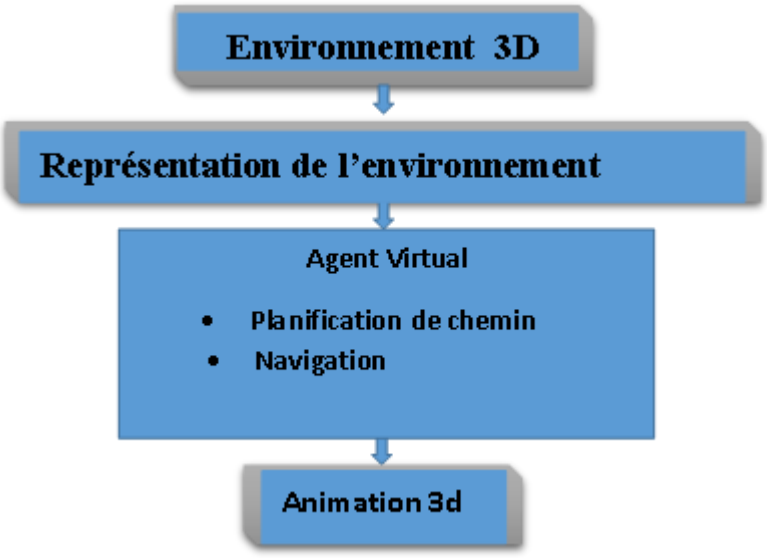


Figure 3.1 : schéma générale de l'application

Donc, on a l'environnement 3D comme entrée et Chemin optimal sans collision avec des obstacles dynamique pour arriver au but mise manuellement ou automatique comme sortie

3.2.2 Conception détaillée

Dans cette partie, on décrit l'architecture détaillée de chaque processus de notre système.

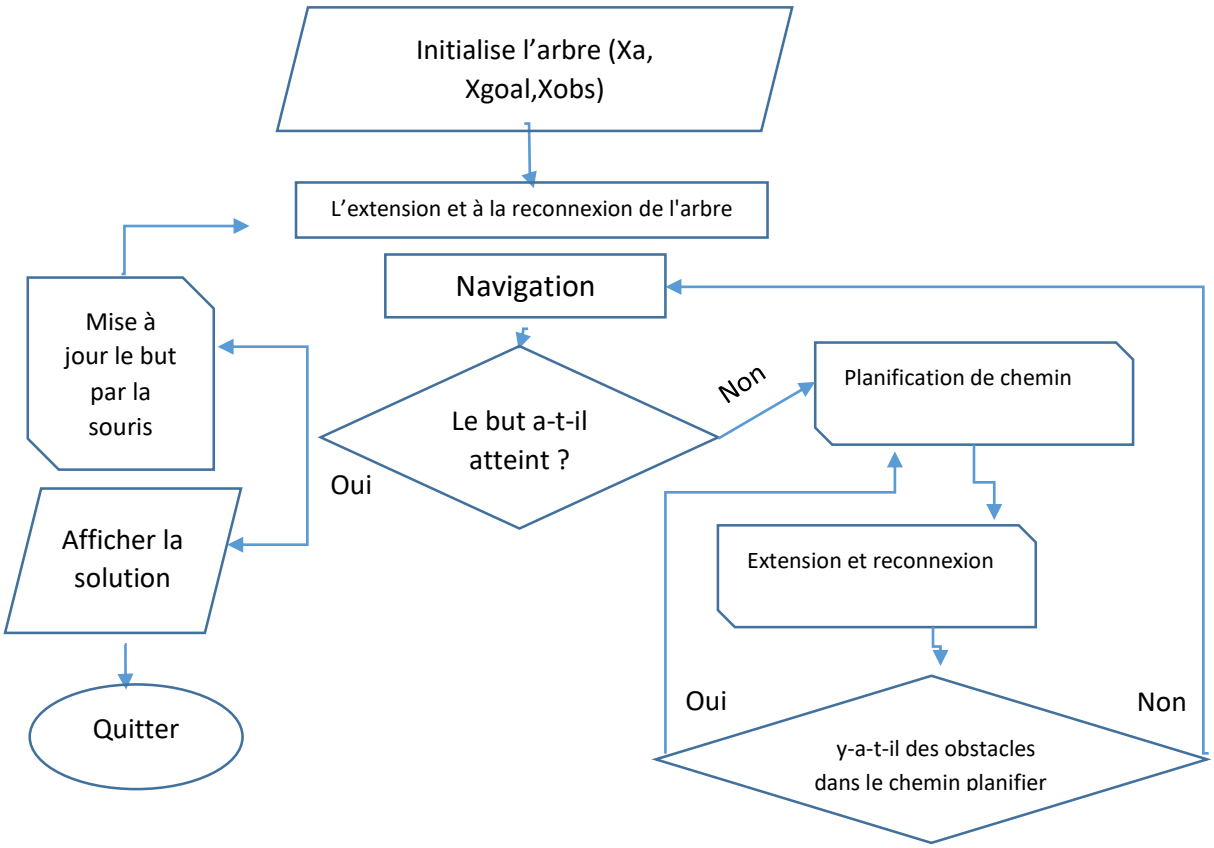


Figure 3.2 : organigramme de l'application

Notre méthode, qui est présentée dans l'algorithme 1, mêle la planification du chemin à l'extension et à la reconnexion de l'arbre. Nous initialisons l'arbre avec Xa comme racine (ligne 2). À chaque itération, nous étendons et reconnectons l'arbre pendant un temps limité défini par l'utilisateur (lignes 5-6). Ensuite, nous planifions un chemin à partir de la racine actuelle de l'arbre pour un nombre limitée définie par l'utilisateur d'étapes supplémentaires (k à la ligne 7). Le chemin planifié est un ensemble de nœuds partant de la racine de l'arbre, (x0 ; x1 ; …; xk). A chaque itération, nous déplaçons l'agent pour un temps limité afin de le garder près de la racine de l'arbre, x0 (ligne 10). Lorsque la planification du chemin est terminée et que l'agent est à la racine de l'arbre, nous changeons la racine de l'arbre pour le nœud immédiat suivant x0

dans le chemin planifié, x1 (lignes 8-9). Ainsi, nous permettons à l'agent de se déplacer sur le chemin planifié sur l'arbre vers le but.

Algorithm 1 RT-RRT*: Real-Time Path Planning
1: Input: Xa, Xobs. Xgoal
2: Initialize T with xa, Qr, Qs
3: loop
4: Update Xgoal, xa, Xfree and Xobs
5: while time is left for Expansion and Rewiring do
6: Expand and Rewire T using Algorithm 2
7: Plan (x0, x1... xk) to the goal using Algorithm 5
8: if xa is close to x0 then
9: x0← xi
10: Move the agent toward x0 for a limited time
11: end loop

3.2.2.1 Extension et reconnexion de l'arbre

L'algorithme2 présente l'extension et la reconnexion de l'arbre. Des nœuds échantillonnés, xrand à la ligne 2, sont ajoutés à l'arbre jusqu'à ce qu'il couvre complètement l'environnement (ligne 7). Le nœud échantillonné xrand est toujours utilisé pour reconnecter des parties aléatoires de l'arbre, soit autour de lui-même, soit autour de son nœud le plus proche, xclosest (lignes 8, 10, 11). Ceci est nécessaire en raison des changements de la racine de l'arbre et des obstacles dynamiques. La ligne 6 indique la condition à remplir pour cela. kmax désigne le nombre maximal de voisins autour d'un nœud tel que xrand et Qs est utilisé pour la distance euclidienne maximale autorisée entre les nœuds de l'arbre.Ces deux valeurs contrôlent ensemble la densité de l'arbre. Comme avec RRT*, Xnear à la ligne 5 est l'ensemble des noeuds voisins de xrand.La ligne 4 vérifie si le chemin entre Xrand et Xclosest est sans collision ou non. Les lignes 11 et 12 exécutent les deux différentes méthodes de reconnexion dans un grand arbre dont la racine change.

Extension de l'arbre : En conservant *T* entre les itérations, l'arbre devient trop grand (mais avec un nombre limité de nœuds) pour être entièrement utilisé dans la planification de chemin en temps réel. Au lieu de cela, nous utilisons un sous-ensemble de nœuds, désigné par XSI. Pour des raisons de simplicité et d'économie de mémoire, nous utilisons une indexation spatiale basée sur la grille (Fig 2, à gauche), mais on pourrait utiliser l'indexation spatiale KD-Tree pour gagner encore en rapidité. XSI est utilisé pour trouve Xclosest et Xnear de xrand dans l'algorithme 2.

Blocage des nœuds par les obstacles dynamiques : Lorsqu'un obstacle dynamique bloque des nœuds à l'intérieur de lui-même, nous fixons leurs valeurs de coût d'accès c, à l'infini. Par

conséquent, tous les nœuds des branches qui découlent des nœuds dans la région de l'obstacle obtiennent une valeur c, infinie. Ainsi, lorsque les algorithmes de reconnexion connectent progressivement et continuellement les nœuds avec un c, élevé à leurs voisins pour réduire la reconnexion créera un autre chemin pour les nœuds avec un c, infini.

Algorithm 2 Tree Expansion-and-Rewiring

```
1:  Input: T, Qr, Qs, kmax, ra
2:  Sample Xrand using (1)
3:  Xclosest = arg minx=Xsi dist(x, Xrand)
4:  if line(Xclosest; xrand) C Xfree then
5:      Xnear ∈FindNodesNear(xrand, Xsi)
6:    if |Xnear| < kmax or |Xclosest -Xrand| > rs then
7:        AddNodeToTree(T,xrand, xclosest, xnear)
8:        Push xrand to the first of Qr
9:    else
10:         Push xclosest to the first of Qr
11:     RewireRandomNode(Qr, T)
12: RewireFromRoot(Qs, T)
```

3.2.2.1.1 Ajout d'un nœud à l'arbre

Lorsque nous voulons ajouter Xnew à l'arbre, de la même manière qu'avec RRT*, l'Algorithme 3 trouves le parent avec la valeur minimale du coût à atteindre (ci) à l'intérieur de Xnear (lignes 2-6). Il faut noter que le calcul de ci est lié à la longueur du chemin de x0 à xi. Ainsi, cost(xi) doit calculer ci chaque fois que cj d'un nœud intermédiaire du chemin vers xi est modifié. Par conséquent, lorsque cost(xi) est appelé, il recalculera ci de xi jusqu'à x0 si un nouveau nœud a été ajouté au chemin vers xi (comme RRT*) ou si des changements dans x0 ou des obstacles dynamiques à l'intérieur de *r*a se sont produits (contrairement à RRT*).

Notez que lorsque cost(xi) est appelé, si l'un des ancêtres de xi est bloqué par un obstacle dynamique, c'est-à-dire ci = 1, nous bloquons également ce nœud. L'introduction de cost(xi) est utilisée dans les algorithmes 3-6.

Algorithm 3 Add Node To Tree

1: **Input**; Xnew ; xclosest; Xnear
2: xmin = xclosest, cmin = COSt (xclosest) + dist(xclosest; xnew)
3: **for** Xnear ∈ Xnear do
4: cnew = cost(xnear) + dist(xnear; xnew)
5: **if** Cnew < Cmin and line(Xnear,Xnew)=Xfree **then**
6: | cmin = cnew, xmin = xnear
7: VT ←VT U (xnew) ET←ET (Xmin,Xnew)

Algorithm 4 Rewire From the Tree Root

1: **Input:** $\mathcal{Q}_s, \mathcal{T}$
2: **if** $\mathcal{Q}_s$ is empty **then**
3: | Push $\mathbf{x}_0$ to $\mathcal{Q}_s$
4: **repeat**
5: | $\mathbf{x}_s$=PopFirst($\mathcal{Q}_s$), $\mathcal{X}_{near}$=FindNodesNear($\mathbf{x}_s, \mathcal{X}_{SI}$)
6: | **for** $\mathbf{x}_{near} \in \mathcal{X}_{near}$ **do**
7: | | c_{old}=cost($\mathbf{x}_{near}$), c_{new}=cost($\mathbf{x}_s$)+dist($\mathbf{x}_s, \mathbf{x}_{near}$)
8: | | **if** $c_{new} < c_{old}$ and line($\mathbf{x}_s, \mathbf{x}_{near}$) $\in \mathcal{X}_{free}$ **then**
9: | | | $E_{\mathcal{T}} \leftarrow (E_{\mathcal{T}} \setminus \{\text{Parent}(\mathbf{x}_{near}), \mathbf{x}_{near}\}) \cup \{\mathbf{x}_s, \mathbf{x}_{near}\}$
10: | | **if** $\mathbf{x}_{near}$ is not pushed to $\mathcal{Q}_s$ after restarting $\mathcal{Q}_s$ **then**
11: | | | Push $\mathbf{x}_{near}$ to the end of $\mathcal{Q}_s$
12: **until** Time is up or $\mathcal{Q}_s$ is empty.

3.2.2.1.2 La reconnexion de l'arbre

La reconnexion est effectuée lorsqu'un nœud (xi) obtient une valeur de coût pour atteindre *C*i inférieure en passant par un autre nœud au lieu de son parent. Ainsi, dans notre algorithme, la reconnexion doit être effectué autour du nouveau nœud (xnew) qui est ajouté (comme RRT*) ainsi qu'autour des nœuds déjà ajoutés en raison de la modification de la racine de l'arbre (x0) et Obstacles dynamiques. Lorsque x0 ou tout obstacle dynamique à l'intérieur de *r*o se change, nous devons reconnecter une grande partie de l'arbre, ce qui est fait par les moyens Reconnexion d'une partie aléatoire de l'arbre (Algorithme 2) Reconnexion en commençant par la racine de l'arbre (Algorithme 4). Les lignes 3 à 7 et les lignes 5 à 9 des algorithmes 4 et 5 reconnectent les nœuds voisins (Xnear) de xr et xs, respectivement. Ainsi, la reconnexion est effectuée dans xnear si en changeant son parent actuel en Xj, ci pour Xnear réduit où Xj est Xr et Xs dans les algorithmes 4 et 5, respectivement. La différence entre ces algorithmes est le point central de la reconnexion. L'algorithme 4 reconnecte une partie aléatoire de l'arbre en commençant par les

nœuds autour de xrand ou xclosest qui sont ajoutés à Qr dans l'algorithme 2. Ensuite, si la reconnexion concerne un xnear, l'algorithme 4 ajoute xnear à Qr puisque les nœuds autour de xnear sont susceptibles d'être reconnecté (ligne 8).

Cependant, l'algorithme 5 se concentre sur la reconnexion de l'arbre autour de x0 et donc autour de l'agent.

Il commence donc la reconnexion à partir de x0 (ligne 3) et pousse tous ses nœuds voisins vers Qs. Ensuite, il continue à pousser des nœuds plus éloignés de x0 en retirant des nœuds (xs) de Qs (ligne 5) et en poussant xnear autour de xs (ligne 11) lorsque la condition de la ligne 10 est remplie. L'utilisation de Qs nous permet de reconnecter les nœuds voisins avec les mêmes valeurs ci de x0. Nous nous référons aux nœuds avec les mêmes cis comme le cercle de reconnexion de x0. La reconnexion se poursuit à chaque itération jusqu'à ce que la condition des lignes 9 et 12 des algorithmes 4 et 5 soit remplie, respectivement. Si le temps est écoulé et qu'il y a encore des noeuds dans Qr ou Qs qui doivent être reconnecté, nous pouvons continuer la reconnexion dans les itérations suivantes.

3.2.2.2 Planification de chemin

La ligne 7 de l'Algorithme 1 utilise l'Algorithme 6 pour planifier un chemin à k étapes à partir de x0. L'algorithme 6 planifie le chemin de deux manières : 1) lorsque l'arbre atteint xgoal (lignes 2-4), ce qui se produit lorsque nous développons l'arbre à l'aide de l'algorithme 3 ; 2) lorsque l'arbre n'a pas atteint xgoal (lignes 6-10).Dans le premier cas, le chemin depuis xgoal jusqu'à x0 est dans l'arbre et nous n'avons donc besoin que de mettre à jour le chemin (ligne 3).

Dans la deuxième méthode, nous planifions un chemin pour nous rapprocher le plus possible de xgoal dans l'arbre. Nous utilisons fonction de coût fi = ci + hi pour se rapprocher de xgoal et pour prendre un chemin court sur l'arbre. Un chemin court sur l'arbre. Ainsi, l'utilisation de fi peut nous piéger dans des minima locaux.

Pour éviter le piège dans les minima locaux et permettre à la planification de d'autres branches, nous planifions un chemin à k étapes en utilisant fi à chaque itération (lignes 6,7) et nous bloquons les nœuds déjà vus (ligne 10) en fixant leur nombre à l'infini. Lorsque la planification du chemin atteint un nœud qui ne peut pas aller plus loin (ligne 8), nous retournons le chemin planifié et bloquons ce nœud (lignes 9 et 10). Ensuite, nous mettons à jour le meilleur chemin déjà trouvé si le chemin planifié nous conduit à un emplacement plus proche de xgoal (ligne 11). Cependant, l'agent suit le meilleur chemin s'il mène à un lieu plus proche que le lieu actuel de l'agent (ligne 12).

Notez que H (xi) renvoie l'infini si xi est bloqué et que xgoal n'a pas été modifié. C'est-à-dire que xi est déjà visité pour le xgoal actuel. Sinon, si xgoal a été modifié ou si xi n'a pas été visité pour le xgoal actuel, H (xi) renvoie l'infini.

Chaque nœud a une chance d'être visité à nouveau, c'est-à-dire que tous ses ancêtres sont débloqués, lorsqu'un autre nœud débloqué est ajouté comme son ancêtre. Débloqué est ajouté comme enfant par la reconnexion ou l'ajout d'un nœud. Il est à noter que lorsque l'arbre a atteint xgoal et que le chemin est bloqué par un obstacle, nous suivons le chemin jusqu'à l'obstacle jusqu'à ce que la reconnexion trouve un autre chemin ou que le chemin soit dégagé.

Algorithm 5 Plan a Path for k Steps

Input: T, Xgoal
if Tree has reached xgoal then
  Update path from Xgoal to xo if the path is rewired
  (X0…Xk) ←(X0…:. Xgoal)
else
  for Xi ∈ (x1, X2,..., Xk) **do**
    Xi=child of xi-1 with minimum fc=cost(xc)+H(xc)
    if Xi is leaf or its children are blocked then
      (X0,..., Xk) ←(xo ,..., Xi)
      Block Xi and **Break**;
  Update best path with (x0,..., xk) if necessary
  (xo,..., Xk) ← choose to stay in xo or follow best path
return (xo, ..., Xk)

3.3 Conclusion

Dans ce chapitre, nous avons donné une spécification sur la conception de notre système. Phase représente l'une des phases les plus importantes du processus de développement de notre système

Elle décrit la conception de notre système d'un point de vue général et détaillé pour comprendre et réussir la phase de programmation.

Dans le prochain chapitre, nous illustrerons la réalisation de notre système en représentant certaines interfaces et certains résultats obtenus, avec les structures qui sont choisies pour implémenter ce système.

Chapitre04 :

Implémentation et résultat

4 Introduction

Après l'établissement de la conception de notre système d'un point de vue général et détaillé qui est la base de notre système en général. Donc dans ce chapitre nous Nous montrerons la représentation et les résultats obtenus par la simulation de notre système

4.1 Configuration matérielle

- Pc
- Microprocesseur : Intel(R) Core(TM) i5-4440 CPU @ 3.20GHz
- Fréquence du processeur : 3.20GHz.
- RAM : 8 GB.
- Carte graphique gt 660 ti boost twin frozer 2
- Système d'exploitation : Windows 11, 64 bits

4.2 Environnements et outils

4.2.1 L'environnement de développement

4.2.1.1 Visual studio 2017

Visual Studio est un ensemble complet d'outils de développement permettant de générer des applications web ASP.NET, des services web XML, des applications bureautiques et des applications mobiles. Visual Basic, Visual C++, Visual C# utilisent tous le même environnement de développement intégré (IDE), qui leur permet de partager des outils et facilite la création de solutions faisant appel à plusieurs langages. Par ailleurs, ces langages permettent de mieux tirer parti des fonctionnalités du framework .NET, qui fournit un accès à des technologies clés simplifiant le développement d'applications web ASP et de services web XML grâce à Visual Web Developer.

Figure 4.1 : Visual studio logo

4.2.1.2 Unity

Unity est un moteur de jeu multiplateforme (smartphone, ordinateur, consoles de jeux vidéo et Web) développé par Unity Technologies. Il est l'un des plus répandus dans l'industrie du jeu vidéo, aussi bien pour les grands studios que pour les indépendants du fait de sa rapidité aux prototypages et qu'il permet de sortir les jeux sur tous les supports.

Figure 4.2 : logo unity

4.2.2 Langage de développement

4.2.2.1 C# (Langage de programmation)

Le langage de programmation C# a été conçu par Anders Hejlsberg de Microsoft en 2000 et a ensuite été approuvé comme norme internationale par l'Ecma (ECMA-334) en 2002 et l'ISO/IEC (ISO/IEC 23270) en 2003. Microsoft a introduit C# en même temps que .NET Framework et Visual Studio, qui étaient tous deux à code source fermé. À l'époque, Microsoft n'avait pas de produits à code source ouvert. Quatre ans plus tard, en 2004, un projet gratuit et open-source appelé Mono a vu le jour, fournissant un compilateur et un environnement d'exécution multiplateforme pour le langage de programmation C#. Dix ans plus tard, Microsoft a lancé Visual Studio Code (éditeur de code), Roslyn (compilateur) et la plate-forme unifiée .NET (cadre logiciel), qui prennent tous en charge C# et sont gratuits, open-source et multiplateformes. Mono a également rejoint Microsoft mais n'a pas été fusionné avec .NET.

En 2021, la version la plus récente du langage est C# 10.0, qui a été publiée en 2021 dans .NET 6.0.

Figure 4.3 : logo C#

4.3 Implémentation

4.3.1 Création de l'environnement

Nous avons créé la scène sur unity 3d qui est un plan rectangulaire entouré par des cubes montrent les limites de la scène avec des obstacles dynamique (sphères) et statique (cubes) et l'attacher avec Visual studio IDE.

- **Division de la scène**

  En a diviser la scène a des régions de taille donnée. Pour minimiser le temps de calcule et faciliter l'exploration de la scène alors qu'en aura 9 régions.

```
public regionManager(Vector2 p1, Vector2 p2, float size)
       {

           float cur_x = p1[0];
           float cur_y = p1[1];
           Regions = new ArrayList();
           while (cur_y < p2[1])
           {
               Regions.Add(new ArrayList());
               while (cur_x < p2[0])
               {
             ArrayList row_regions_i = (ArrayList)Regions[Regions.Count - 1];
                    (row_regions_i).Add(new regionInfo(new Vector2(cur_x,
cur_y), size, p2, new Vector2(Regions.Count - 1, row_regions_i.Count)));
                    cur_x += size;
               }
               cur_x = p1[0];
               cur_y += size;
           }
       }
```

- **Créer le nœud de la scène**

Un SceneNode est un objet invisible auquel on va pouvoir attacher un nombre indéfini d'entités On devra passer par un nœud déjà existant, ce qui va nous permettre d'avoir des

relations d'héritage entre nos noeuds. Et on va attacher les entités à nos noeuds, ceux-ci peuvent avoir un noeud parent, lui-même enfant d'un autre noeud, et ainsi de suite jusqu'au noeud appelé « racine » de la scène.

```
//ajouter un noeud a la région la plus proche
           regionInfo closest_region = getRegionFromIndex(min_index);
           closest_region.nodes_inside_region.Add(nNode);
```

Elle retourne l'index minimal de la région et la considère comme la plus proch et ajout un nœud a la région.

- **Création des obstacles**

Nous avons placé des objets dans notre environnement des cubes comme des obstacles statiques et des sphères comme des obstacles dynamique dispatcher dans la scène et l'agent Virtual essaye de les éviter en fonction de l'algorithme utiliser (RRT*).
Après avoir placé les obstacles dans la scène on doit affecter leur collisionneur (collider) et la vitesse (vilocity) pour les obstacles dynamique

```
 targetBody = (GameObject.Find("Target") as GameObject);
        targetBody1 = (GameObject.Find("Target1") as GameObject);
        targetBody2 = (GameObject.Find("Target2") as GameObject);
        targetBody3 = (GameObject.Find("Target3") as GameObject);
        (GameObject.Find("Target") as
GameObject).GetComponent<Collider>().enabled = true;
        (GameObject.Find("Target1") as
GameObject).GetComponent<Collider>().enabled = true;
        (GameObject.Find("Target2") as
GameObject).GetComponent<Collider>().enabled = true;
        (GameObject.Find("Target3") as
GameObject).GetComponent<Collider>().enabled = true;

     targetBody.GetComponent<Rigidbody>().velocity = new Vector3(-1, 0, -1);
     targetBody1.GetComponent<Rigidbody>().velocity = new Vector3(1, 0, 1);
     targetBody2.GetComponent<Rigidbody>().velocity = new Vector3(1, 0, -1);
     targetBody3.GetComponent<Rigidbody>().velocity = new Vector3(-1, 0, 0);
```

- **Camera**

La camera est élément qui définit la position de notre point de vue (observateur) dans la scène dans laquelle direction on regarde.
La fonction suivante permet de crée une caméra qui suit notre agent appeler sphère

```
// Update est appelée une fois par image(frame)
    void Update () {
        transform.LookAt(GameObject.Find("Sphere").transform.position);
    }
```

Cette fonction est appelée une fois par image (frame)

- **Résultat**
  Voici quelque résultat obtenu et vue général sur la scène

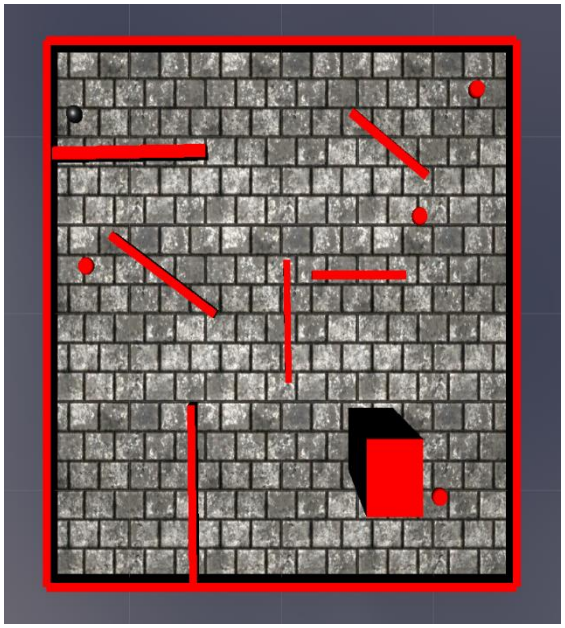


Figure 4.4: vue vertical sur la scène

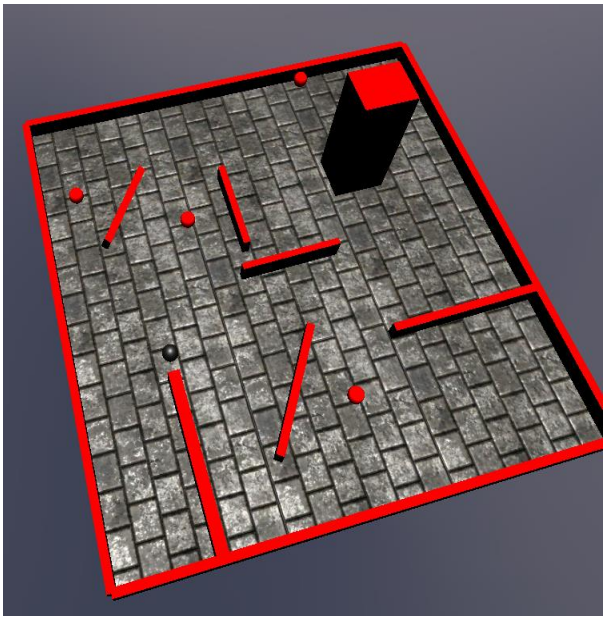


Figure 4.5 : vue perspective sur la scène

4.4 Représentation de l'espace de navigation :

L'espace de navigation et le chemine planifier est représenter par la class RRT

```
public class RRT
{


    ////////////////////// definition des variables ////////////////////////
    //Simulation
    public int mSimulation_numberOfNodes = 2000;//pour que l'agent se déplace
    public ArrayList cost_of_found_paths;//pour sauvgarder le coût
    public int numberOfChangedInPath = 0;//le nomber des changment
    public float cost_of_taken_path = 0;

    //les variable pour l'arbre
    public ArrayList TreeNodes;/
    int current_root = 0;
    public Vector2 goal_point;// 2D: x,z;
    private float rateTowardGoal = 0.1f;
    int numberOfRRTSamples = 500;
```

```
        //planification en temps real
        ArrayList best_path_soFar;
```

4.5 Des methods utiliser:

4.5.1 Ajouter un noeud:

```
1  public int add_node(Vector3 pos, Vector2 dir, float radius, int
   indexOfFather)
2          {
3              mNodeRRT new_node = new mNodeRRT(pos, dir);
4              new_node.indexOfFather = indexOfFather;
5              new_node.node_index_inTree = TreeNodes.Count;
6              //region
7              mRegionManager.addNodeToRegion(new_node);
8
9              if (indexOfFather >= 0)
10             {
11                 mNodeRRT mParent = (mNodeRRT)TreeNodes[indexOfFather];
12                 restart_parent_node_values(mParent.node_index_inTree,
   goal_pos_3D);
13                 new_node.cost_to_reach =
   new_node.getCostToReach(TreeNodes, numberOfUpdatingRoot +
   numberOfUpdatingObstacles + numberOfUpdatingStr);/
14             }
15
16             new_node.node_value = dis_toGoal;
17             TreeNodes.Add(new_node);
```

Si l'index de père est supérieur à 0 en initialise le cout à atteindre le nœud (cost_to_reach) et ajouter à l'arbre par l'instruction treeNodes.add(new.node).

4.5.2 Changer la position but

En change la position but par deux méthodes

- **Méthode automatique**
  Elle génère une position but automatiquement après la génération d'un nombre précis des nœuds qui est 2000 dans notre cas.

```
public void change_goal_point(Vector2 new_goal)
       {
           if ((goal_point - new_goal).magnitude > cRadius/2)//accuracy
           {
               numberOfUpdatingGoals++;
               flag_find_path = false;
```

```
                goal_index_inTree = -1;

                set_goal_point(new_goal[0], new_goal[1]);
                cur_cost_path = float.MaxValue;
                best_cost_path = float.MaxValue;
                cost_path_inside_notReachArea = 0;
                flag_calculate_path_cost = false;

                //For simulation
                cost_of_found_paths = new ArrayList();
                numberOfChangedInPath = 0;//during the movement the path is
going to change a couple of times, this will tell how many times it changed
                iteration_needed_for_firstPath = 0;//iteration needed for
returning the first path
                cost_of_taken_path = 0;//iteration needed for returning the
first path
            }
            return;
        }
```

- **méthode manuel**
  En change la position but (goal position) manuellement par souris (mouse)

```
void UpdateTargetWithMouse()
    {
        if (Input.GetMouseButtonDown(0))
        {
            RaycastHit hit;
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

            if (Physics.Raycast(ray, out hit))
            {
                Transform objectHit = hit.transform;
                if (objectHit.name == "Plane")
                {
                    Vector2 goal_point = mRRT.mPoint3Dto2D(hit.point);
                    xd = goal_point[0];
                    yd = goal_point[1];
                }

            }
        }
    }
```

4.5.3 Détection de collision

Les obstacles dynamique ou statique ils ont un collisionneur (colider), pour les sphères connus par leurs rayons (radius) et pour les box connus par ces axes (X, Y)

donc chaque nœud tombe à l'intérieur, l'index de père va être supprimé seulement par la reconnexion donc le nœud reste mais pas connecter, cette méthode est la détection par (raycast)

```
public bool collisionDetection(Vector3 p1, Vector3 p2)//p1 and p2 represent le
centre de la mass de l'agent
        {
            RaycastHit mhitinfo;
            bool isHit = Physics.Raycast(p1, (p2 - p1) / (p2 - p1).magnitude,
out mhitinfo, (p2 - p1).magnitude);
            Vector3 XPlus = new Vector3(0.5f, 0f, 0f);
            Vector3 ZPlus = new Vector3(0f, 0f, 0.5f);
            float dis_val = (p2 - p1).magnitude + 0.00001f;
            Vector3 dir_p1top2 = (p2 - p1) / dis_val;
            if (!isHit)
            {
                isHit = Physics.Raycast(p1 + dir_p1top2 * cRadius, dir_p1top2,
out mhitinfo, dis_val);
            }
            if (!isHit)
            {
                isHit = Physics.Raycast(p1 + XPlus, dir_p1top2, out mhitinfo,
dis_val);
            }
            if (!isHit)
            {
                isHit = Physics.Raycast(p1 - XPlus, dir_p1top2, out mhitinfo,
dis_val);
            }
            if (!isHit)
            {
                isHit = Physics.Raycast(p1 + ZPlus, dir_p1top2, out mhitinfo,
dis_val);
            }
            if (!isHit)
            {
                isHit = Physics.Raycast(p1 - ZPlus, dir_p1top2, out mhitinfo,
dis_val);
            }

            return isHit;
        }
```

RaycastHit, dans Unity, est un objet de données structuré qui est renvoyé lorsqu'un rayon frappe un objet pendant un raycast

4.5.4 Dessiner l'arbre :

Pour dessiner l'arbre faut savoir le nombre des nœuds dans l'environnement(TreeNods.count) et choisir seulement les nœuds accessibles (reachable_node) et les en dessiner par la fonction Drawline.

```
public void drawTree()
        {
            for (int i = 0; i < TreeNodes.Count; i++)
            {
                mNodeRRT mN = (mNodeRRT)TreeNodes[i];

if (mN.reachable_node)
                {
                    for (int j = 0; j < mN.child_nodes_index.Count; j++)
                    {
                        mNodeRRT mN_child =
(mNodeRRT)TreeNodes[(int)mN.child_nodes_index[j]];
                        if (mN_child.reachable_node && FlagUseColor)
                            Debug.DrawLine(mN.objectPosition - (new
Vector3(0f, 0.3f, 0f)), mN_child.objectPosition - (new Vector3(0f, 0.3f, 0f)),
Color.white);
                    }
                }
            }
            return;
```

4.6 Résultats

Dans cette section, nous présentons les résultats obtenus pour le modèle de planification en temps réel en utilisant RRT*.

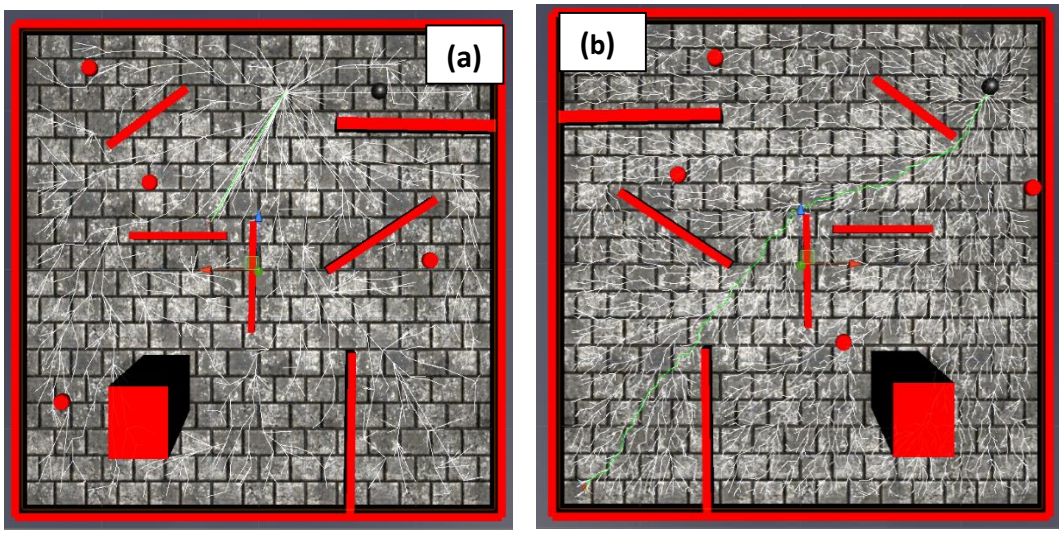


Figure 4.6 : l'exploitation de la scène par l'arbre en fonction de nombre d'itération (a)500 itérations (b) 2000 itérations

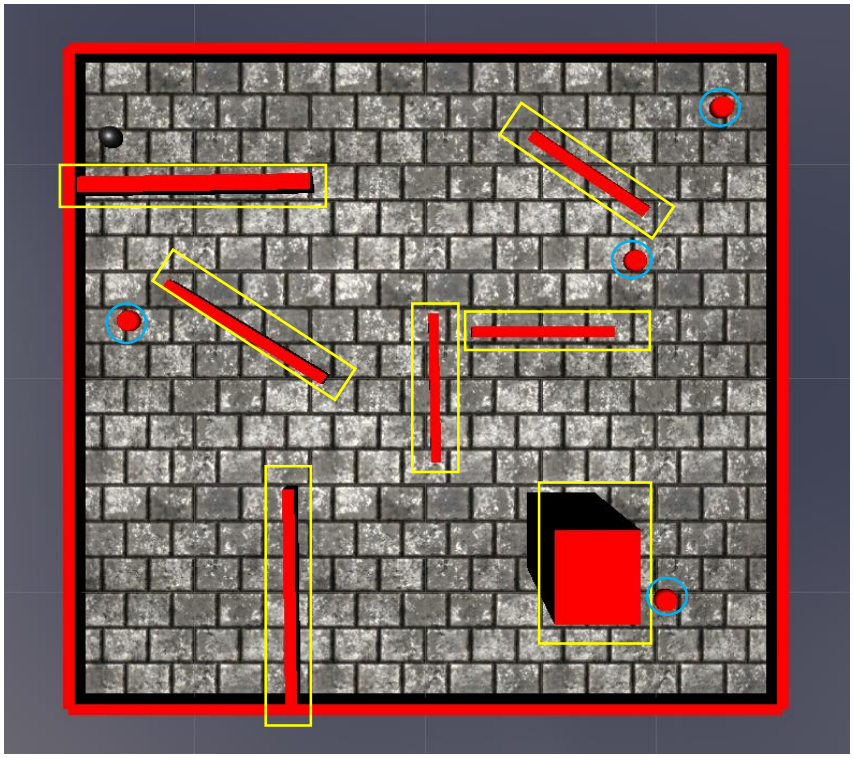


Figure 4.7 : les obstacles (les obstacles dynamique entouré par un cercle bleu et statique encadré en jaune)

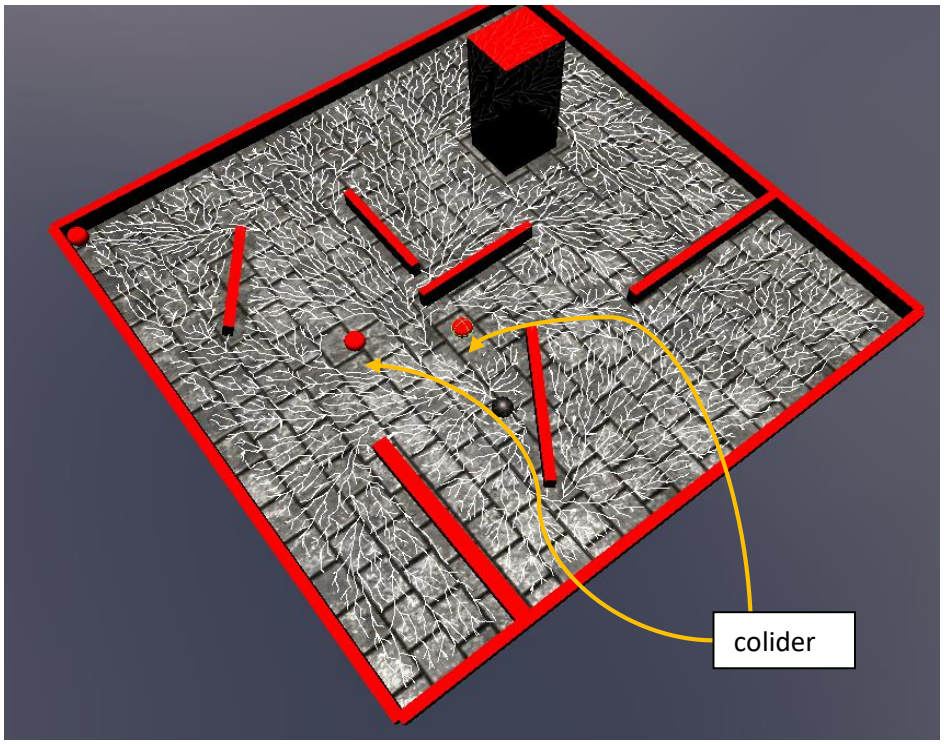


Figure 4.8 : image montre les collisionneurs et les zones inaccessible

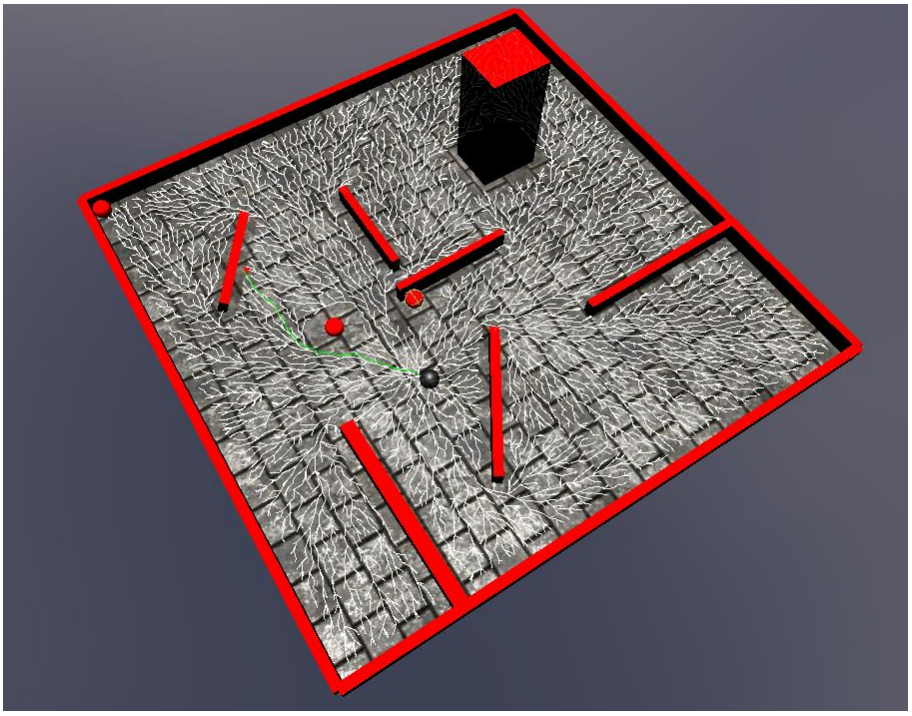


Figure 4.9 : image montre l'évitement de collision avec un obstacle dynamique

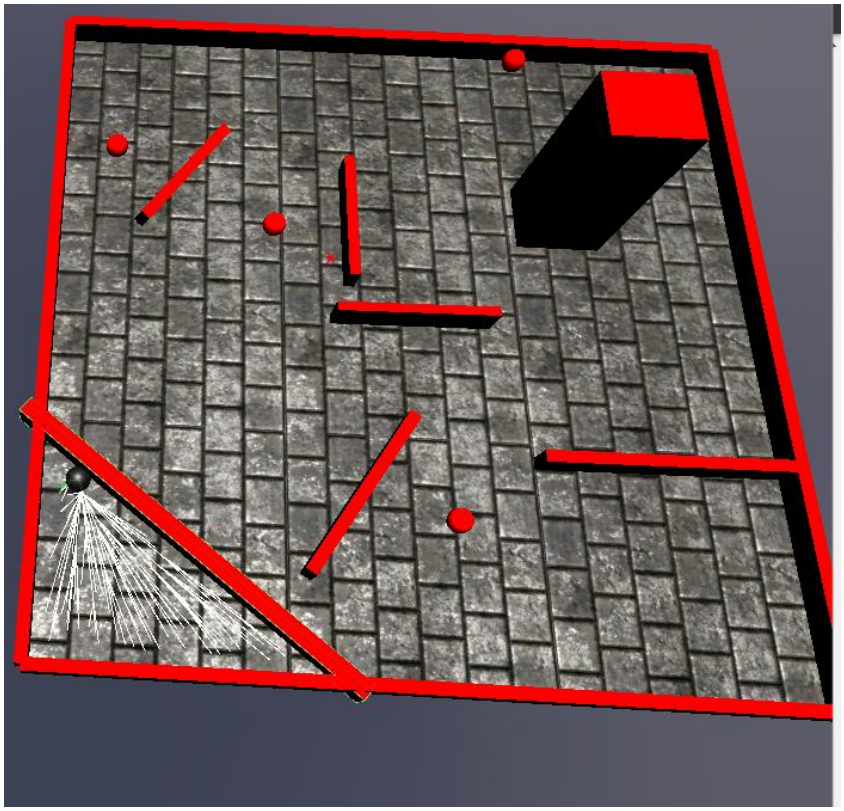


Figure 4.9 : si on bloque l'agent par un obstacle

Dans les images des résultats suivant en vont démontrer la notion de reconnexion, alors on change la position des obstacles dans la scène et relancer la simulation donc les endroits inaccessibles va être accessible par la méthode de reconnexion

Figure 4.11 : la notion de reconnexion dans la simulation

4.7 Conclusion

Dans ce chapitre, nous avons décrit l'implémentation détaillée de notre système de planification de chemin en temps réel basé sur l'algorithme RRT* avec les outille utiliser pour terminer ce travaille et en a montrer à la fin le résultat obtenu.

Conclusion générale

Conclusion générale

Dans cette mémoire, nous avons présenté une version en temps réel de RRT* La capacité en temps réel a été obtenue en entrelaçant la planification du chemin avec l'expansion de l'arbre et la reconnexion .De plus, nous déplaçons la racine de l'arbre avec l'agent pour conserver l'arbre au lieu de le reconstruire à chaque fois. L'arbre au lieu de le reconstruire à chaque itération. L'arbre continue de croître jusqu'à ce qu'il couvre l'environnement. Nous avons également introduit deux modes de reconnexion pour avoir des chemins plus courts dans le grand arbre avec un nombre limité de nœuds. IL s'agit de:

- la reconnexion à partir de la racine,
- la reconnexion de parties aléatoires de l'arbre.

La première méthode crée un cercle croissant centré sur l'agent. Dans ce processus, chaque nœud à l'intérieur du cercle est reconnecté, et ce cercle reconnecte le plus souvent les nœuds autour de l'agent et donc l'arbre. Reconnexion des nœuds autour de l'agent et donc de la racine de l'arbre. Le second est réalisé en utilisant à la fois l'échantillonnage focalisé et uniforme, mais en patches au lieu d'un seul nœud. Planification de chemin en temps réel. Nos simulations montrent que la combinaison de la conservation de l'arbre avec les deux méthodes de reconnexion du grand arbre dans RT-RRT* permet à l'algorithme de trouver des chemins plus courts avec moins de itérations vers un ou plusieurs points d'arrivée. Il convient de noter que, Pour des raisons de simplicité et de mémoire, nous avons utilisé une grille pour accélérer notre recherche des nœuds voisins. Accélérerait encore plus la recherche et permettrait d'utiliser D'utiliser des budgets d'échantillonnage plus importants. Le RT-RRT* a ses limites, par exemple il nécessite une grande capacité de mémoire car l'arbre entier est stocké à tout moment. L'une des principales limites de notre algorithme est qu'il ne fonctionne que dans un environnement délimité. L'échantillonnage ciblé à l'intérieur d'un ellipsoïde fonctionne quelque peu dans un environnement non limité, mais la reconnexion souffre si les distances sont grandes. Distances sont grandes. Il reste donc à relever les défis des environnements non limités et à grande distance restent à relever. L'algorithme permet à l'agent de suivre le chemin planifié en douceur et de planifier des mouvements autour des obstacles, il ne nous fournit malheureusement pas de modèle des obstacles.

Références bibliographiques

Références

[1] S. Thrun et A. Bücken. Integrating grid-based and topological maps for mobile robot navigation. Dans Proc. of the AAAI Thirteenth National Conference on Artificial Intelligence, pages 944–951. AAAI Press / MIT Press, 1996.

[2] M. David Solomon, «Development of a Real-Time Hierarchical 3D Path Planning Algorithm for Unmanned Aerial Vehicles,» University of Maryland, College Park, 2016.

[3] H. Samet , «AN OVERVIEW OF QUADTREES, OCTREES, AND RELATED HIERARCHICAL DATA STRUCTURES,» University of Maryland, College Park, 1988.

[4] K. Daniel, A. Nash et S. Koenig, «Theta*: Any-Angle Path Planning on Grids, » University of Southern California, Los Angeles, 2010.

[5] O. Arikan, S. Chenney, et D. A. Forsyth. Efficient multi-agent path planning. Dans Computer Animation and Simulation ’01, pages 151–162. Springer-Verlag, 2001.

[6] H. Choset, A. Mills-Tettey, K. Tantisevi et V. Lee-Shue Jr. Prasad Narendra Atkar, «Robotic Motion Planning: A* and D* Search,» Carnegie Mellon University, Pittsburgh, 2005.

[7] B. Logan et N. Alechina. A* with bounded costs. Dans Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), pages 444– 449, 1998

[8] M. H. Overmars. Recent developments in motion planning. Dans International Conference on Computational Science (3), pages 3–13, 2002.

[9] J.D. BOISSONNAT, O. DEVILLERS, R. SCHOTT, M. TEILLAUD et M. YVINEC : Applications of random sampling to on-line algorithms in computationnal geometry. In Discrete Comput. Geom., 1992.

[10] B. TOVAR, S.M. LAVALLE et R. MURRIETA: Optimal navigation and object finding without geometric maps or localization. In Int. Conf. on Robotics and Automation (ICRA’03), 2003.

[11] J. C. Latombe. Robot Motion Planning. Boston: Kluwer Academic Publishers, Boston, 1991

[12] J.J. Kuffner. Efficient optimal search of euclidean-cost grids and lattices. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 4466–4471. Citeseer, 2004.

[13] Z. Shiller, K. Yamane, and Y. Nakamura. Planning motion patterns of human figures using a multi-layered grid and the dynamics filter. Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation, pages 1–8, 2001

[14] Steven M. LaValle. Planning Algorithms. Complexity, 2006.

[15] S.M. LaValle. Rapidly-exploring random trees: A new tool for path planning. (98-11), 1998.

[16] J. Pettre, J.P. Laumond, and D. Thalmann. A navigation graph for real-time crowd anim-Action on multilayered and uneven terrain. In First International Workshop on Crowd Simulation pages 81–90. Citeseer, 2005.

[17]M. Kallmann, H. Bieri, and D. Thalmann. Fully dynamic constrained DelaunayTriangulations. Geometric Modelling for Scientific Visualization, 3, 2003.

[18] Fabrice Lamarche and Stephane Donikian. Crowd of Virtual Humans: a New Approach for Real-Time Navigation in Complex and Structured Environments. Computer Graphics Forum, 23(3):509–518, September 2004.

[19] F. Lamarche. TopoPlan: a topological path planner for real-time human navigation underfloor and ceiling constraints. Computer Graphics Forum, 28(2):649–658, April 2009.

[20] K. Wong and C. Loscos. Hierarchical path planning for virtual crowds. Motion in Games, pages 43–50, 2008

[21] M.H. Overmars and P. Svestka. A probabilistic learning approach to motion planning. UU-CS, (1994-03), 1994.

[22] L. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration for fast-path planning. Proceedings of the 1994 IEEE International Conference on Robotics and Automation, pages 2138–2145, 1994.

[23] LE Kavraki, P Svestka, JC Latombe, and M OverMars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Transactions on Robotics and Automation, 1996

[24] Kenneth E. Hoff, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99, pages 277–286, 1999.

[25] R. Geraerts and M.H. Overmars. The corridor map method: Real-time high-quality path planning. In Robotics and Automation, 2007 IEEE International Conference, pages 1023–1028. IEEE, 2007

[26] Brian Salomon, Maxim Garber, Ming C. Lin, and Dinesh Manocha. Interactive navigation in complex environments using path planning. Proceedings of the 2003 symposium on Interactive 3D graphics - SI3D '03, page 41, 2003.

[27] T. Simeon, J.P. Laumond, and C. Nissoux. Visibility-based probabilistic roadmaps for motion planning. Advanced Robotics, 14(6) :477–493, 2000.

[28] J.J. Kuffner and S.M. LaValle. RRT-connect: An efficient approach to single-query path planning. Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065), (Icra) :995–1001, 2000.

[29] S. Karaman, M.R. Walter, A. Perez, E. Frazzoli, and S. Teller. Anytime motion planning using the rrt*. In Robotics and Automation (ICRA), 2011 IEEE International Conference, pages 1478–1483. IEEE, 2011.

[30] R. Alterovitz, S. Patil, and A. Derbakova. Rapidly-exploring roadmaps: Weighing exploration vs. refinement in optimal motion planning. In Robotics and Automation (ICRA), 2011 IEEE International Conference, pages 3706–3712. IEEE, 2011

[31] D. Ferguson, N. Kalra, and A. Stentz. Replanning with rrts. In Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference, pages 1243–1248. IEEE, 2006.

[32] J. Bruce and M. Veloso. Real-time randomized path planning for robot navigation. In Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference, volume 3, pages 2383–2388. Ieee, 2002.

[33] Matt Zucker, James Kuffner, and Michael Branicky. Multipartite RRTs for Rapid Replanning in Dynamic Environments. Proceedings 2007 IEEE International Conference on Robotics and Automation, pages 1603–1609, April 2007

[34] Tsai-yen Li. An incremental learning approach to motion planning with roadmap management. Proceedings 2002 IEEE International Conference on Robotics and Automation, pages 3411–3416, 2002.

[35] Russell Gayle, Kristopher R. Klingler, and Patrick G. Xavier. Lazy Reconfiguration Forest (LRF) - An Approach for Motion Planning with Multiple Tasks in Dynamic Environments. Proceedings 2007 IEEE International Conference on Robotics and Automation, (April):1316–1323, April 2007.

[36] Avneesh Sud, Erik Andersen, Sean Curtis, Ming C Lin, and Dinesh Manocha. Real-time path planning in dynamic virtual environments using multiagent navigation graphs. IEEE transactions on visualization and computer graphics, 14(3):526–38, 2008.

[37] Avneesh Sud, Russell Gayle, Erik Andersen, Stephen Guy, Ming Lin, and Dinesh Manocha. Real-time navigation of independent agents using adaptive roadmaps. Proceedings of the 2007 ACM symposium on Virtual reality software and technology - VRST ’07, 1(212):99, 2007

[38] M. Lau and J.J. Kuffner. Precomputed search trees: planning for interactive goal-driven animation. In Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation, pages 299–308. Eurographics Association, 2006

[39] P. Fiorini and Z. Shiller. Motion Planning in Dynamic Environments Using Velocity Obstacles. The International Journal of Robotics Research, 17(7):760–772, July 1998.

[40] Jur van den Berg and Dinesh Manocha. Reciprocal Velocity Obstacles for real-time multi-agent navigation. 2008 IEEE International Conference on Robotics and Automation, pages 1928–1935, May 2008.

[41] M. Phillips and M. Likhachev. Sipp: Safe interval path planning for dynamic environments. In Robotics and Automation (ICRA), 2011 IEEE International Conference, pages 5628–5635. IEEE, 2011.

[42] J. van den Berg, D. Ferguson, and J. Kuffner. Anytime path planning and replanning in dynamic environments. Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006. (May) :2366– 2371, 2006.