EXHAUSTIVE EXPLORATION STRATEGIES FOR NPCS IN GAME MAPS

by

Muntasir Chowdhury

School of Computer Science
McGill University, Montréal

December, 2015

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Abstract

Much work has been done on automated terrain mapping in the field of robotics. And while there has been interest in Artificial Intelligence (AI) navigation in video games, there currently exists no formalized approach by which a non-player character (NPC) can automatically explore and map out an entire game level.

In this dissertation, we present a method that generates an exploratory path in the game map, using which an NPC will be able to uncover all parts of the map. Our problem statement resembles that of the Watchman Route Problem, studied theoretically in computational geometry. We model a game map as a 2-dimensional polygon with holes, then using the Art Gallery Theorem compute a set of camera points that collectively guarantee full coverage of the map, and connect these cameras using motion planning graphs called roadmaps. From there we use visibility polygons, polygon merging and shortest-path calculations, to develop five different strategies for tours that visit the cameras.

We identify factors that influence the performance of these strategies, and experimentally analyse their influence using metrics we have developed. Additionally, we compare the performances of the strategies themselves, as well as test on different types of roadmaps. Our experiments are carried out on large and complex maps from commercial games. We have devised a simple procedure for the collection and conversion of such maps to an easily readable format.

The strategies proposed can be used by both hostile and non-hostile NPCs in different spatial search scenarios. For example, to locate a player, or to help a player uncover a map in *fog-of-war* settings. We hope our efforts will encourage further work on this relatively nascent topic. The development of more complex metrics, and more human-like strategies, could help quantify the "explorability" of a map, and aid in level design decisions.

Résumé

Beaucoup de travail a été fait sur la cartographie de terrain automatisée dans le domaine de la robotique. Bien qu'il y ait eu un intérêt pour la navigation de jeux vidéo à l'aide de l'Intelligence Artificielle (AI), il n'existe actuellement aucune approche formalisée par laquelle un personnage non joueur (PNJ) pourrait automatiquement explorer et cartographier tout un niveau de jeu.

Dans cette thèse, nous présentons une méthode qui génère un chemin d'exploration dans la carte de jeu, dans laquelle un PNJ sera en mesure de découvrir toutes les parties de la carte. Notre problématique ressemble à celle du problème Route Watchman, étudiée théoriquement en géométrie algorithmique. Nous modélisons un plan de jeu comme un polygone à 2 dimensions avec des trous. En utilisant le théorème de la Galerie d'Art, nous calculons ensuite un ensemble de points de la caméra qui garantissent collectivement une couverture complète de la carte et de relier ces appareils à l'aide de graphiques de planification de mouvement appelés feuilles de route. De là, nous utilisons des polygones de visibilité, des polygones de fusion et des calculs du plus court chemin pour développer cinq stratégies d'exploration différentes. Ces explorations servent à visiter les caméras.

Nous identifions les facteurs qui influencent la performance de ces stratégies et analysons expérimentalement ces influences en utilisant des mesures que nous avons développées. De plus, nous comparons les performances des stratégies elles-mêmes tout en faisant des essais sur différents types de feuilles de route. Nos expériences sont réalisées sur des cartes de jeux commerciaux vastes et complexes. Nous avons mis au point une procédure simple pour la collecte et la conversion de ces cartes dans un format lisible.

Les stratégies proposées peuvent être utilisées par des PNJ aussi bien hostiles que non hostiles dans des scénarios de recherches spatiales variés. Par exemple, elles peuvent être

appliquées pour situer un joueur ou pour l'aider à découvrir une carte dans une situation de brouillard de la guerre. Nous espérons que nos efforts encourageront la poursuite de travaux sur ce sujet relativement nouveau. Le développement de mesures plus complexes et de stratégies plus humaines pourrait aider à quantifier les conditions d'exploration d'une carte et faciliter les décisions de conception de niveau.

Acknowledgements

I would like to thank my supervisor, Prof. Clark Verbrugge, for the fruitful discussions, valuable feedback, and kind encouragement he provided during the course of this thesis. I am grateful to my parents for their constant and unfaltering support. A thank you, also, to my friends and family in Montreal and elsewhere, for keeping my spirits up during struggles, big and small.

Contents

List of Figures

List of Algorithms

Chapter 1
Introduction

Exploration behaviour in video games is a subject that has not been studied in depth, yet, it is one of the fundamental elements that motivate people to play games [4]. In games today, a significant amount of resources are spent in constructing rich and complex worlds, largely for the purpose of enhancing the player's sense of discovery. Players are made to feel that they have vast possibilities for exploration, and also that their choice to engage in such activities will result in an experience that is interesting i.e. not repetitive or trivially randomized. The increasing demand for this type of game design emphasizes the importance of exploration in gaming.

In games of certain genres (e.g. roguelike, turn-based) non-player characters (NPCs) are required to aid human players in exploration. The automated exploration algorithms used by these NPCs are not explicitly clear, since, to the best of our knowledge, there is no published works describing them. However, by observing the gameplay and examining the code in games like *Civilization 5* [11] and *Dungeon Crawl Stone Soup* [14], it has been deduced that a trivial greedy approach is utilized. The results of these are often not considered satisfactory by players [10]. Regardless, these games demonstrate an existing scenario where good exploration techniques for NPCs would be needed.

One way of meeting this need would be to develop an algorithm that allows an NPC to automatically explore a map, such that it eventually uncovers or "sees" all parts of that map. To be effective, this approach must be methodical, and guaranteed to carry out an exhaustive exploration. It must also be reasonably efficient, but not necessarily optimal, as

that is not how players expect exploration to work. The approach should, of course, avoid gross redundancies and appear somewhat intelligent.

Given that it has overlaps with objectives in pathfinding (e.g. searching, moving through space), automated exhaustive search of an area should be a potentially useful tool for many scenarios. Such a method could be used to allow companion (non-hostile) NPCs to aid a player in exploring a map or in searching for items. Enemy NPCs can be instructed to search for a human player in an area as part of an attack strategy. In stealth games this can be used as an alternative or parallel approach for creating guard paths [43]. For example, some guards may cover specific areas with more aggressive techniques, while other guards cover the entire map inspecting all parts of it. Lastly, in open-world games, players often encounter important free roaming NPCs (e.g. a trade merchant) in a way that is made to feel spontaneous but useful. This design could be enhanced by an automated exploration method that guarantees the NPC will be visible to each part of the map at some point on its journey. This means the probability of encountering a specific NPC anywhere in the map will be better distributed.

In this thesis, we present a methodology for exhaustive exploration. We abstract real game maps as 2-dimensional polygons with holes. Using the Art Gallery Theorem, we compute a set of points called cameras, which collectively see all parts of the map. We build a graph in the form of lines running across the map. This graph or *roadmap* can be used to move between regions inside the polygon. We connect the cameras to the roadmap, and then create a tour on it that visits all the cameras. This tour is the exploration tour that allows the NPC to see the map in its entirety. Since creating this tour is an NP-hard problem resembling the Travelling Salesman Problem, we develop several heuristic based strategies that focus on achieving 100% visual coverage of the map as quickly as possible. For our experiments, we develop metrics and list a number of factors that can affect such strategies. We then experimentally test the influence of these factors, along with a comparative analysis of the strategies proposed. Our results show that a complex strategy, that considers potential increase in coverage and distance at different points of the tour, works better than simpler heuristics.

To strengthen the applicability of our work, we developed and integrated all the necessary tools inside the widely used game development environment known as Unity [12].

The incorporation of our work in an industry relevant tool enables the use of our techniques in real game design.

1.1 Contributions

The specific contributions of this study are listed below.

- A simple but necessary procedure for collecting maps from proprietary games, and converting them to a format suitable for a range of experiments. It was developed in response to the general unavailability of such maps (in forms other than crowd-sourced screenshots), and the difficulty of extracting relevant data from them.
- A procedure that takes a game map as input, and outputs an exploratory path or tour within the map. An NPC that travels along this path is guaranteed to see all parts of the map by the end of its journey.
- 5 different strategies for computing an exploration tour based on varying heuristics. The foundation on which these strategies rely, including the work that guarantees they will achieve full coverage of a game map, can be considered independently. The development of this independent procedure allows further strategies to be created simply by formulating new heuristics.
- A detailed exploration of a variety of factors that may affect such strategies, along with an overall assessment of the strategies themselves.

1.2 Thesis Outline

The topics in this thesis are divided as follows.

- In *Chapter 2* we start with an overview of basic concepts in video games to better contextualize the overall problem. We then move on to a discussion of how maps are modelled and obtained by our system.

- In *Chapter 3* all the topics related to coverage are described. We show how the art gallery theorem can be used compute points, known as cameras, that allow us to guarantee coverage of a map. After that we describe tools for measuring coverage individually and collectively, using visibility polygons and polygon merging algorithms.

- *Chapter 4* explains how we connected the camera points using a type of navigation graph known as roadmaps, and then how to create a tour of the cameras using the roadmap. We relate this to the Travelling Salesman Problem, and show how using Dijkstra's algorithm and our coverage measuring tools, we can create different heuristic tour strategies.

- *Chapter 5* goes into detail about our experiment methodology, shows the different game maps we have used, and describes our test environment. We then present a summary of the results that were obtained along with comments and inferences.

- *Chapter 6* contains a discussion of research related to this study. We touch upon work done on visual coverage and exploration in the fields of computational geometry and robotics, respectively.

- *Chapter 7* summarizes the insights from this thesis, and lists the different ways this work can be extended in the future.

Chapter 2
Context and Maps

In our problem the primary input data being dealt with is a game map. How we model a map, and the way in which we deal with its nuances and complexities affect all the problem solving techniques and tools that will be used later on. This chapter discusses all map related issues in *Section 2.2*. Before that we first touch upon general concepts related to video games in order to better contextualize the problem of exploration. In *Section 2.1* we talk about games in terms of their structure and the experience they provide to the people who play them i.e. gamers or players.

2.1 Contextualizing Exploration

Fig 2.1 shows a map from the role-playing-game (RPG), *Pillars of Eternity*. Here we have an indoor environment consisting of rooms in a building that are separated from each other by walls and doorways. A player's most general objectives in such a game involve moving their character(s) around within this space, while collecting items and interacting with NPCs. NPCs are simply characters in the game that are controlled by the game's artificial intelligence system. The interactions may be friendly or hostile (i.e. involves fighting) depending on the nature of the NPC. If it's the latter a player may sometimes need to evade the NPC depending on the game's objectives.

The player's perspective in the game may either be a first-person view (*Fig 2.2*) or some form of an isometric or overhead view where they can see their character(s) in relation to

Figure 2.1 Screenshot of a game map (`game-maps.com`)

the rest of the game world (*Fig 2.3*). When players move around with the aim of achieving an in-game objective, they often do so in an exploratory manner. By this we mean the player moves inside the map discovering it part-by-part as they try to collect items, fight NPCs, complete challenges, find an exit, etc. Sometimes the player simply does this with the aim of exploring and "seeing" the game world.

The way NPCs move about in the game world is somewhat different. Sometimes they simply stand in one spot waiting for the player to come and interact with them. Other times they go from a point A to a point B in a previously defined or automatically generated route. They may also be walking around in some fixed region controlled by an algorithm that tells them to keep moving while avoiding obstacles. Lastly, they might follow or chase the player as the player moves. When a group of similar NPCs move together from one location to another, game developers often try to make them appear as if they are coordinating their movements.

In this thesis, we attempt to formalize an approach by which an NPC demonstrates exploration-like behaviour in its movements. More specifically, we want the NPC to undertake a tour, i.e. walk a finite distance, during which it uncovers the map incrementally

Figure 2.2 First-person view in Skyrim (mynexusreview.com)

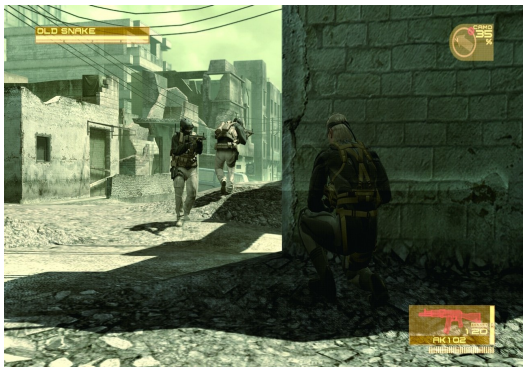

Figure 2.3 Third-person view in Metal Gear (eldojogamer.com)

and by the end of which it has seen all areas of the map. This "uncovering" and "seeing" is referred to as *coverage* or visual coverage in our work. Furthermore, we want to try several different methods for achieving full coverage and compare these methods.

2.2 Maps

If we look at *Fig 2.1* again, we can see that it contains a considerable amount of detail. The amount of graphical detail, such as texture, hue, and lighting of surfaces, might appear overwhelming when thinking about modelling all such features. Fortunately, they can be ignored, since for the purposes of testing exploration behaviour, our essential requirement is the geometric shape of the game world. There is, of course, significant variation and irregularity in the shapes of walls, floors, and objects, all of which affect a character's motion and visibility. For example, in certain places the floor dips or elevates, or there are objects in the room which one can climb on to see over a wall or other obstructive element. Therefore, even though the complexity is greatly reduced it is still of a considerable magnitude, and so deciding how to model a map for our experiment requires some thought.

We have chosen a polygonal model and manually drawn the maps from the images. Using real game maps means that they have to be abstracted to a general model that makes a comparative analysis possible. We will now discuss our choice of using a polygonal model for abstracting maps in *Section 2.2.2*, and the process of collecting and converting

maps from existing games to our model in *Section 2.2.3*. Before that we clarify some notions regarding the terminology we are using, and the game genres whose maps we are considering, in *Section 2.2.1*.

2.2.1 Terminology and Genre

We will be using the terms "map" and "level" interchangeably for video games. In practice they often equate to the same thing. There are cases in which a single level can consist of multiple maps or multiple levels of the game are played on the same map. But for the purposes of this work a map or a level is simply an enclosed geometric space where the game's most commonly used way of dividing the narrative into distinct blocks ("mission", "battle", "quest", "challenge") takes place. Let us further clarify our definition of maps by excluding game genres where there is little or no opportunity for exploration either on the part of the player or the NPCs. *Beat-em-up games*, where two characters fight each other face to face, sports games, and most driving games are good examples of this. The geometric shape of the levels in these genres are fairly simple and more or less fixed. In case of driving games, the player's journey through a race track is more or less bound and controlled.

Maps in genres such as role-playing, strategy, first-person shooter, and third-person shooter games have grown more elaborate in detail over time. There has been a rise in popularity of games with an open-world, such as the Grand Theft Auto series. This has resulted in more game developers trying to create a sense of seamless gameplay for players by creating maps that appear boundless. In reality, these maps have an outer boundary and do not go on infinitely. Despite the large size these games still have specific areas within which a challenge takes place. These areas are usually not much larger than the maps used in close-world scenarios. And, even though the player may go anywhere in the map NPCs are most often bounded to such areas.

The maps used in our experiments are all sourced from commercially available games. This is to help us argue the real-world applicability of the methods we have used. The games chosen are primarily role-playing games (RPG) because the concept of exploring a space is very relevant in this genre, although these methods should apply to maps of other

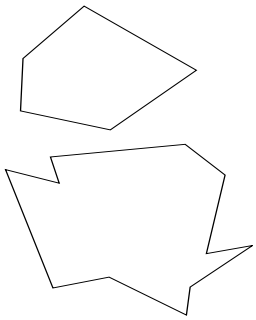

Figure 2.4 Simple polygons

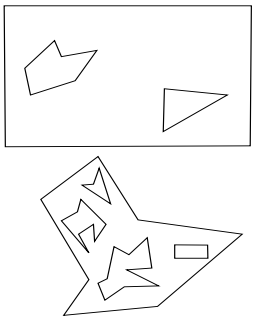

Figure 2.5 Polygons with holes

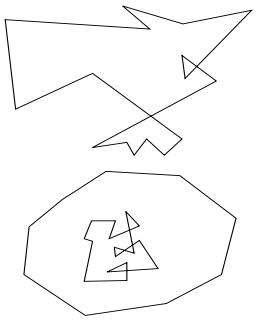

Figure 2.6 Complex polygons

genres as well. RPGs favour a more relaxed pace for their narratives, often allowing the gamer to determine the speed at which to finish the story. They also have many side-quests, which are secondary challenges not requisite to completing the primary narrative arc. A larger variety of NPCs are present in games of this genre, and there is more focus on making the characters fit naturally into the story. Intelligent behaviour from NPCs heightens the feeling of immersion for the gamer, which is a big priority for RPG developers.

2.2.2 Abstraction Model

Game-maps are at the foundation of our experiments in this dissertation. In fact, they are the only external data source we will rely upon. As mentioned before, they also happen to have a considerable amount of complexity. Therefore, choosing an appropriate way of abstracting them is quite important. The model we choose greatly influences the range and difficulty of algorithms and solutions that can be applied to the problem. We will now briefly talk about polygons, then outline our chosen polygonal model, and lastly discuss the subtleties the model sacrifices and accounts for.

Polygons

One common way of representing geometric spaces is a polygon. Polygons can be categorized in many ways. We are interested in three major subgroups: simple, polygons with holes, and complex.

A simple polygon (*Fig 2.4*) is a collection of distinct line segments that are connected

to form a closed path [20]. Every segment is connected on each of its endpoints by one and only one adjacent line segment. Line segments do not intersect each other except at these endpoints and no line segments exist outside the closed loop. If we add a smaller simple polygon inside a simple polygon and neither set of line segments intersect or are self-intersecting, the collective structure is then referred to as a polygon with holes. A polygon with holes is shown in *Fig 2.5*. A polygon where the line segments intersect each other outside the endpoints is known as a complex polygon (*Fig 2.6*).

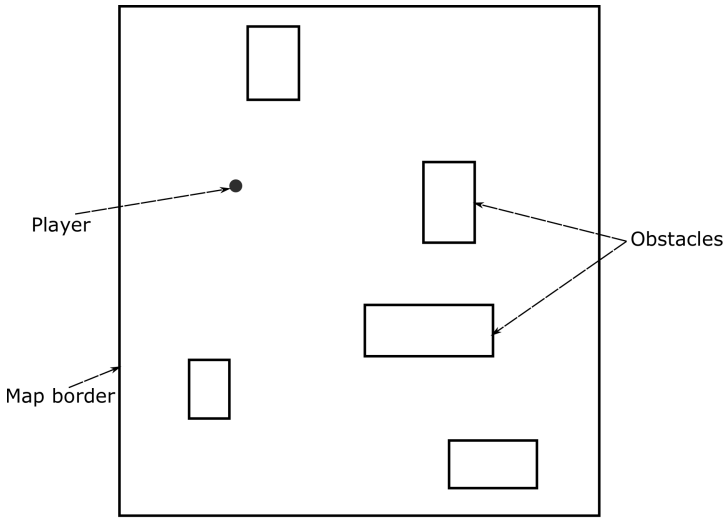

Figure 2.7 Map made of rectangles

Our Model

We have constrained our analysis of three-dimensional game maps to a two-dimensional space. To be more specific, we consider maps in the form of polygonal structures where the NPC is a point inside the polygon. The polygonal structure can be a simple polygon but is most often a polygon with holes. *Fig 2.7* shows a simple example map composed of rectangles. In contrast, *Fig 2.8* illustrates the type of arbitrary shapes and terrains the model is potentially capable of handling.

The NPC can move about in every direction on the x and y axes and is only prevented from doing so by obstacles. The boundaries of the map form the primary polygon, which we will refer to as the *exterior*. Any obstacles inside the map are represented as holes

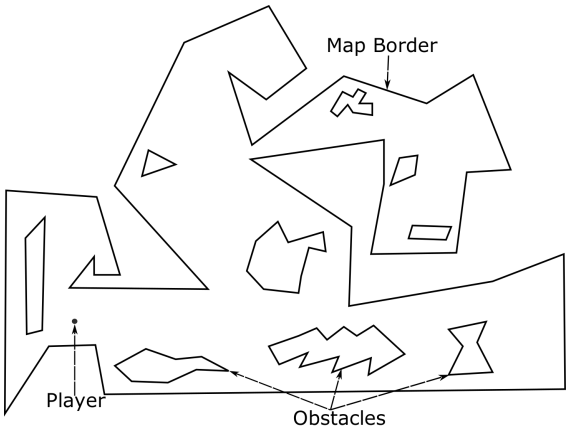

Figure 2.8 Map consisting of arbitrary shapes

(simple polygons) inside the primary polygon. The polygon is representative of the floor or the base upon which players and NPCs travel.

Visibility and Mobility

Modelling a game map as a polygon with holes offers a great deal of flexibility. It accounts for enough subtlety to be used with a vast number of real game maps, while being abstract enough to admit the use of algorithms and representations whose complexity is not far outside the scope of a master's thesis. The nuances of this model can be understood through two factors: visibility and mobility. In this section, we first explain what the terms mean and then illustrate several scenarios where mobility is greater than visibility and vice-versa. After that our focus turns to how these properties are affected by variation in the heights of in-game objects, and the range of movements a player has.

Definitions - Visibility refers to the area of the map visible to the character from their position inside it. Mobility refers to their ability to move around. Let p be the position of the NPC in the map. Let V be the set of all points that are visible from p, and M be the set of all points that the character can reach by walking in a straight line from p. For points inside a polygon with holes, V and M are equivalent i.e. all visible points are reachable and vice-versa.

Discrepancies - However, this is not always true in a real game-world. Sometimes, the sets

V and M have points exclusive to them i.e. all points that can be seen from p cannot be reached directly from it and vice versa. For example, if there is a glass wall NPCs are able to see right through it but cannot move through it. In this case, there is greater visibility than mobility. Another example is a pool of water at the center of a room that the player cannot interact with. On the other hand, there are cases when a player cannot see what is on the other side of a closed door while inside the room, but they can easily open the door and walk out. Teleportation is an effect where this is especially true. Sometimes when players stand on particular objects or walk through certain doorways, they are instantly transported to an entirely different part of the map. In this case, they have greater mobility than visibility. Teleportation is one feature we do not attempt to consider at all.

Object Heights - By converting everything to 2-dimensions our model implicitly assumes that objects inside the 3-dimensional space extend from the floor to the ceiling. This is obviously not true in the game-world, and there are many objects as well as characters of various heights. Height can create the problem of providing partial visibility while cutting out mobility. A crate lying on the floor obstructs a certain part of the floor behind it but not the entire space behind it. Though it is not possible to walk through the crate, the region behind it is partially visible. We deal with this problem by judging the amount of visibility. If the object is a minor obstruction to visibility we do not put it in the map and the NPC has no obstruction to mobility because of it. If it will considerably affect the view of the character we signify it as a hole in the polygon.

Multiple Floors - Of course, the biggest problem here is when the height of the base or the floor on which the NPC stands varies. We maybe able to ignore small drops and elevations on the surface but the same cannot be done when the elevation is significant. Typical examples of this are multi-storied buildings. This is made even harder when the contents of one floor is visible from the other. With some loss of accuracy, it is not too difficult to model two floors that are connected only by an isolated set of stairs and aren't visible to each other. The individual floors can be modelled as polygons with holes and the stair can simply be two edges that connects them. But when there are many points of connection at different points of the floors, a 2-dimensional model no longer suffices. In these scenarios the only solution is to let the NPC explore each floor at a time and treat them as individual

maps.

Varied Movement - Lastly, the ability of players to execute different types of movement such as jumping, crawling and climbing also creates issues. For example, there might be a wall with a gap at the bottom that players can crawl through but which serves as a visual obstruction. The same situation arises when there is a gap between the top of the wall and the ceiling, and characters have the ability to climb over.

The advantage of using a 2D polygonal model is that the algorithms we will employ will be comparatively less complex than a 3D model. They are also computationally faster and easier to debug, which is an important factor due to the inherent numerical stability issues related to computational geometry. Many of the scenarios can be solved by using two different polygons for visibility and mobility, and stacking them. This is not done in our study. Our solution still has great deal of applicability in the context of testing exploration behaviour, without the substantial increase in complexity polygonal overlay would cause.

2.2.3 Collection and Modelling

Obtaining game maps for use in a specific experiment environment is not trivial. As games are proprietary products, most developers and publishers do not make their levels readily available for use outside the game. Even if they were easy to obtain, it would be difficult to extract the relevant spatial information from each developer's specific format and translate it to the generalized model used in the experiment. Our solution is to manually draw these maps. This approach maintains the essential geometric information about the map while greatly simplifying the conversion to a 2-dimensional model. It is relatively easy to implement, and can be applied to wide variety of experiments that require the 2D geometry of a game map.

Initially, we drew our own maps as we built and tested our system before moving on to recreating real game maps for the results phase. *Fig 2.9* outlines how we prepared maps for the experiments. The first step is to obtain a screenshot or some visual presentation of the map. Then we draw the map as a polygon in Inkscape, a graphics editor, which saves the image as an SVG (Scalable Vector Graphics) file. The SVG file is run through a parser we created and transformed into a CSV (Comma Separated Values) file. The information

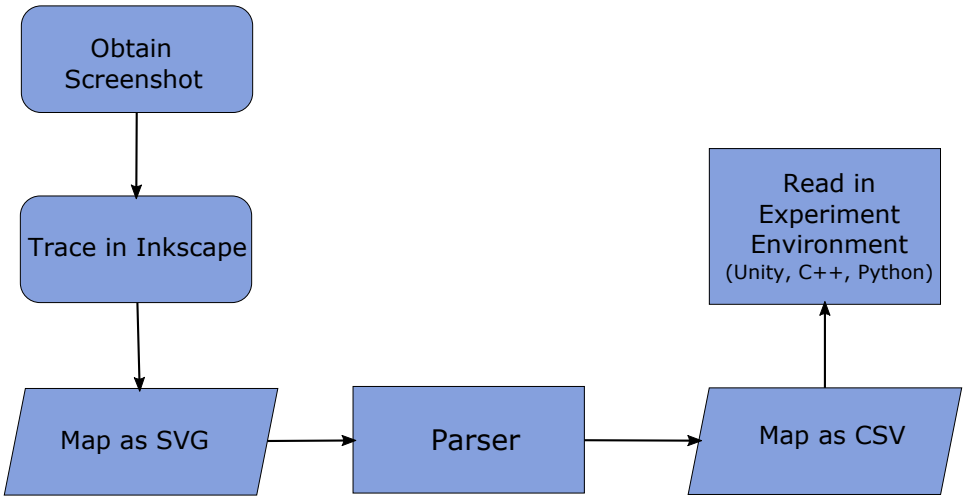

Figure 2.9 Process for map collection and modelling

in this CSV file is then easily read in Unity and other programs we have used. Both SVG and CSV are text formats.

Obtain Screenshot

To reproduce existing video game levels in our chosen model we require at the least a depiction of the level from a bird's-eye or an isometric view. Ideally this would be a blueprint-like schematic where all obstacles (walls, objects, buildings) are represented as lines and polygons. Since these are difficult to find we most often used in-game screenshots and abstracted complex structures to their basic outlines. The screenshots are all from online sources. The site `game-maps.com`, which specializes in walkthroughs for games, was useful in particular. They had a full collection of maps for several recently released RPGs.

Trace in Inkscape

After obtaining the screenshots we drew the 2D representation in Inkscape, an open-source vector graphics editor. The screenshot is first opened in Inkscape and then traced over. For 3D images we decided to always trace along the top of the image, i.e. the tops of walls and other structures. We did this because overhead shots will usually obscure some of the visual details of the floor. On the rare occasion when a 2D blueprint-like schematic was

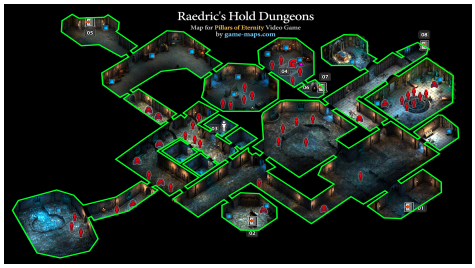

Figure 2.10 Tracing by hand in Inkscape over the map image

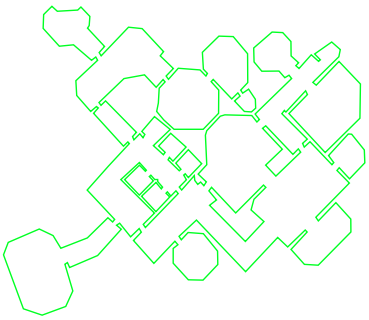

Figure 2.11 The trace

available for the level, this process was easier and more accurate.

Most obstacles and walls were represented as holes (simple polygons themselves) in our model. All walls that are somehow connected to each other are drawn as a single polygon. Similarly, walls that were connected to the map border are drawn as part of it. Geometrically this doesn't change anything. But it helps avoid numerical stability issues that arise from two parallel line segments drawn too close to each other.

Many of the screenshots we worked with provided an isometric view of the map. That is to say the viewer is not perpendicular to the base of the map, and the view is not a genuine overhead shot. Since we do not know the exact angles and transformations the game used to do this, we applied some horizontal transformations in Inkscape until the image looked like a regular 2D view.

Parser

Inkscape saves the drawing as an SVG file, which is an XML (Extensible Markup Language)-like format for vector images. We created a simple parser in Python to extract the relevant spatial information i.e. the coordinates of the lines that make the exterior (map boundary) and all the holes (obstacles) inside it. To help the parser differentiate between the holes and the map boundary we added a "title" to each hole. The list of lines for each polygon is listed in a CSV that can be trivially read by tools in the experiment environment.

Read in Experiment Environment

The presentation of the data in the CSV file is straightforward. Each line contains the description of a line segment. The first column of each row is a label stating whether the line belongs to an obstacle in the map or to the map border. A particular obstacle's description is grouped together. There will first be a row with a single column with the label "OBStart" implying that the description of a new obstacle follows. At the end of the line segment descriptions, there is a label "OBEnd" to signify that a complete polygon can be created with these line segments. The line segment description is in the format: *label*, x_1, y_1, x_2, y_2. Where x_1 and y_1 are the x and y coordinates of the first endpoint of the line segment, and x_2 and y_2 of the second. A sample from one of our map descriptions is shown in *Fig 2.12*.

	Label	x_1	y_1	x_2	y_2
233	M	-309.359	476.7331	-309.555	450.9366
234	M	-309.555	450.9366	-358.162	452.951
235	M	-358.162	452.951	-358.815	425.2626
236	M	-358.815	425.2626	-371.75	426.0438
237	OBStart				
238	OB	32.83549	136.9861	33.03079	149.0954
239	OB	33.03079	149.0954	-23.2192	149.0954
240	OB	-23.2192	149.0954	-23.4145	126.8298
241	OB	-23.4145	126.8298	-33.5707	126.6345
242	OB	-33.5707	126.6345	-33.7661	155.5407
243	OB	-33.7661	155.5407	44.55429	154.7595
244	OB	44.55429	154.7595	44.94489	136.5954
245	OB	44.94489	136.5954	32.83549	136.9861
246	OBEnd				

Figure 2.12 Excerpt from a map description (in CSV) produced by the parser

After knowing the format, it is trivial to write a method in any commonly used programming language to read in the data. We have done so in the various environments (Unity, Python, C++) we have used for our experiments.

Chapter 3
Coverage

Once we have our map in the form of a polygon with holes we can think about coverage in more tangible terms. The NPC's objective is to see the entire map. Geometrically that translates to seeing all the points inside the polygon. For a polygon with holes, this of course excludes the area inside the holes. We have chosen to treat our NPC as a point inside the polygon as this reduces complexity and helps with generality. A different shape (e.g. a line or circle) can always be thought of as a set of points. But the reverse is not true, and therefore our methods here have a better chance of being extended to other models.

As the NPC is a point, we can use the concept of point visibility to determine what it has seen. Point visibility creates a polygon V that contains all the points and only those points that are visible from a point p. V is also called a visibility polygon.

Let P be a polygon with holes. For each point p inside S there is a visibility polygon V. V is contained entirely within P, and represents a portion of P that has been seen. If we can obtain a set of visibility polygons that collectively cover all of the area within P we can solve the problem of seeing the entire map. Since each visibility polygon is created from a particular point in P the problem of ensuring full coverage then becomes one of finding an appropriate set of points for the NPC to visit. Specifically, we have to find a set of points from which all points of the polygon (map) can be collectively seen. This is a discretisation of our original task and is the same as the Art Gallery Problem.

Section 3.1 describes cameras, which are the geometric points from which we consider coverage. We explain how coverage is calculated from these points using visibility poly-

gons in *Section 3.2*. *Section 3.3* focuses on determining if the whole map has been seen. This is done by merging individual visibility polygons to obtain aggregated coverage. Each of these sections are divided into two parts. The first part relates to the theoretical ideas necessary to understand the concept or technique in question, and the second part outlines how those ideas were implemented in our work.

3.1 Cameras

Cameras are strategically chosen points of visual coverage. Individual cameras provide a partial view that is essential to aggregating a full view of the level. Choosing the right set of cameras is therefore important to our exploration algorithm and in this section we will describe how that's done. The Art Gallery Problem was solved using several mathematical ideas, the principle one of which is triangulation. We will first discuss the concepts related to triangulation in *Section 3.1.1*, and then in *Section 3.1.2* discuss how camera positions were computed from the maps we have abstracted. References to the mathematical ideas discussed below can be found in the book "Art Gallery Theorems And Algorithms" by Joseph O'Rourke [31].

3.1.1 Triangulation

A triangulation of a polygon divides the area inside the polygon strictly into a set of triangles (see *Fig 3.1*). The vertices of the triangles are exclusively from the set of points that make up the vertices of the polygon i.e. no new vertices are created. Triangulation is a decomposition mechanism that helps us deal with complex shapes by considering them as smaller building blocks. We will be using triangulation to determine a suitable set of cameras for a game map. In this section we first explain the triangulation of simple polygons, the 3-colourability of such triangulations, the triangulation of polygons with holes, and, finally, how all those concepts relate to finding the minimum number of cameras required to visually cover a polygon with holes.

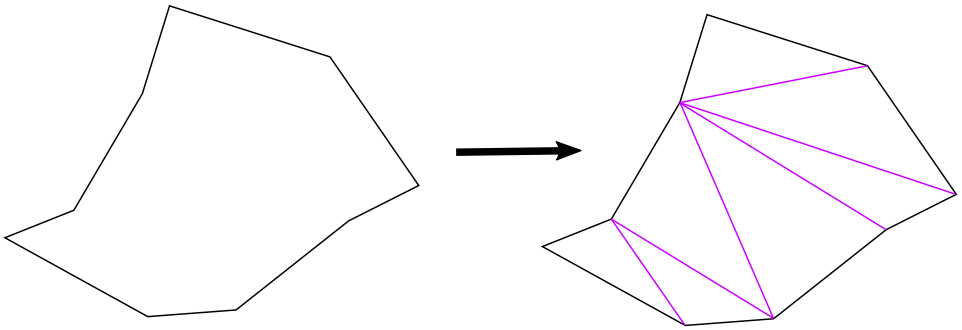

Figure 3.1 A triangulation of a polygon

Simple Polygons

All simple polygons admit a triangulation. This can be proven easily with a process known as "ear clipping." An ear of a polygon is a triangle composed of three consecutive vertices. This triangle contains no other vertex of the polygon on or inside it. Every polygon with $n >= 4$ vertices has at least two non-overlapping ears. For a polygon of n points we can find an ear and then remove it to get a polygon of $n-1$ vertices. Note that in the process we have removed a triangle. We can recursively remove the ears from each new polygon until we are left with a polygon of 3 vertices i.e. a triangle.

A consequence of this proof is that a triangulation of a polygon will always yield $n-2$ triangles. Observe that we have removed $n-3$ triangles from the polygon until we are left with the last remaining triangle. Therefore, $t = (n-3)+1 = n-2$, where t is the number of triangles.

3-Colourable

Triangulations of simple polygons are 3-colourable [17]. 3-colourable implies that if we take a set of three labels, e.g. {1,2,3}, it is always possible to label the nodes of the triangles in such a way such that no two adjacent nodes have the same label. The term "colourable" is used as colours are often used for the labels.

This theorem can also be proven by the "ear clipping" method. Let us imagine the clipping process in reverse, starting from the last remaining ear or triangle. First, we colour the three nodes with three different colours. If we now add the last vertex that was removed

it can only be adjacent to two of the existing nodes, and can be given the colour of the node it is not adjacent to. By induction this process can be continued until all n vertices have been added back and coloured producing a 3-coloured polygon.

Cuts and Polygons with Holes

The triangulation of polygons with holes takes place slightly differently. The principle idea here is convert the polygon to a simple polygon. Once converted the same ideas of triangulation and colouring that work for simple polygons can be applied. The conversion is done by removing holes. The removal process does not change the original shape of the polygon. It simply treats the position of the cut as a boundary edge instead of a free interior region of the polygon. The last image in *Fig 3.2* is modified to highlight that the hole no longer exists. In reality, there will be no new gap created because of the cut. It should also be noted that this converted polygon is only used for the triangulation and colouring phase.

To remove a hole we draw a line from a vertex on the exterior of the polygon (i.e. the border) to a vertex on the hole. This line must not intersect any other vertex or line segments. The line segment is then duplicated allowing the edges of the hole to become a part of the exterior of the triangle. By repeating the process for all holes this essentially puts all the line segments of the polygon on its exterior making it a simple polygon.

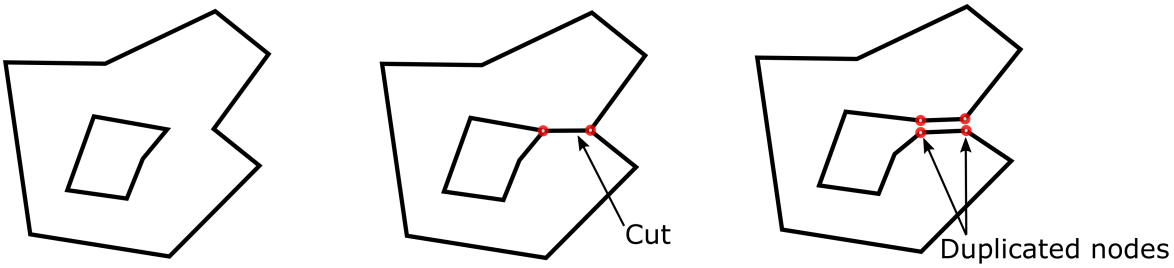

Figure 3.2 Converting a polygon with holes to a simple polygon

This process is also referred to as making a cut. In the event where there is a hole whose vertices are not visible to any of the vertices on the exterior, we combine it with another hole whose vertices are visible. All holes must either be visible to another hole in the polygon and/or to the exterior. We can keep combining holes until they connect to one whose vertices are visible to the exterior. If we consider each of the holes and the exterior

as nodes of a planar graph, and consider the cuts made to be the edges connecting these nodes, the graph produced is always a spanning tree. This property is helpful in practical implementations.

Each time we make a cut we are duplicating vertices on either two holes or a hole and the exterior. A vertex on each end of the line segment is being duplicated. The resulting simple polygon therefore has $n+2h$ vertices, where h is the number of holes, as two vertices are being added for every hole. Following from the proof in the previous section, the triangulation of a polygon with n vertices and h holes will always produce $t = (n+2h)-2$ triangles.

Minimum Number of Cameras

The Art Gallery Problem asks the following question, "What is the minimum number of cameras that are needed to ensure complete visual coverage of a polygon?"

Consider the problem for $n = 3$, a triangle. All points of a convex polygon are visible from any of its vertices. The same is true for a triangle since it is a convex polygon. Recall that a simple polygon will always give a triangulation that is 3-colourable. After such a colouring any set containing all the vertices of a single colour is a valid set of cameras for which the problem can be solved. This follows because every triangle has a vertex of each of the three colors. By choosing the colour that appears the least times we get the minimum number of cameras. Since this colour appears at most $\lfloor n/3 \rfloor$ times, we can say that $\lfloor n/3 \rfloor$ cameras are sufficient and sometimes necessary to solve the art gallery problem for a simple polygon.

This solution can then be extended to a polygon with holes once we have converted it into a simple polygon. In this case $\lfloor (n+2h)/3 \rfloor$ cameras are sufficient and sometimes necessary to guarantee complete coverage for a polygon with n vertices and h holes. A slightly stricter bound of $(n+h)/3$ and more complex proofs can be found in O'Rourke's book [31].

3.1.2 Computing Camera Points

With the theoretical background set we can now explain our implementation of the solution. After reading the map from the CSV file we select our camera points through a three step process. If our map has any holes, it is first converted to a simple polygon by making *cuts*. Below we discuss how to get the right set of cuts by forming a spanning tree. Then, a triangulation is computed, and, lastly, we do a 3-colouring to get the camera points themselves.

Spanning Tree

Cuts can be made between any visible pair of vertices on two holes *or* a hole and an exterior. Therefore, there are many possible combinations of cuts that can remove all holes from a polygon with holes. We must ensure that there are no unnecessary cuts as these may result in the polygon splitting off into smaller parts. All holes must either have a cut connecting them to the polygon border, or to a hole connected to the border by a cut. This condition is necessary to create a single connected border, and, hence, a simple polygon. A corollary of this condition is that for h holes there will be h cuts.

Planar Graph - As mentioned before, we can think of the individual holes and the polygon border to each be nodes of a planar graph. The h cuts can be thought of as edges between these nodes. We then have a connected graph of $n = h + 1$ nodes and $n - 1$ edges, in other words, a tree. Because these nodes can have a wide variety of connections constructed between them, this tree can be considered a subgraph of a much larger graph. Therefore, it is a spanning tree.

Connecting Simple Polygons - We have devised a simple algorithm, reminiscent of Breadth-first Search, to compute this spanning tree. The procedure is described in *Algorithm 1*. The *MakeSpanningTree* function takes the game map in the form of a list of simple polygons, P. P contains each of the obstacles in the map and the map boundary itself. The variable $exteriorindex$ indicates the index of P that contains the exterior. *MakeSpanningTree* first makes cuts between the exterior and all the holes visible from it. Each connected hole is then used to attempt a connection to other unconnected holes. This process is continued

Algorithm 1 Spanning Tree

```
1:  procedure MAKESPANNINGTREE(P, exteriorindex)
2:      for i ← 1 to |P| do
3:          visited[i] ← false                          ▷ All nodes set to unvisited
4:      cuts ← ∅                          ▷ List of cuts i.e. the edges of the spanning tree
5:      Q ← ∅                                                          ▷ Queue of indices
6:      Q.Add(exteriorindex)
7:      visited[exteriorindex] ← true
8:      while Q ≠ ∅ do
9:          u ← Q.Head()
10:         Q.RemoveHead()
11:         for v ← 1 to |P| do
12:             if visited[v] ≠ true and Connectable(P,u,v,cuts) = true then
13:                 visited[v] ← true
14:                 cuts.Add( MakeConnection(P,u,v,cuts) )
15:                 Q.Add(v)
16:     return cuts
```

until all holes are connected.

We treat each of the polygons in P as a node. The array *visited* indicates whether a node has been visited. We take an empty queue Q and add the exterior to it. Next, a *while* loop commences that will only exit after Q is empty. In each iteration of the loop we take the first element, u, in Q and remove it. Then for each polygon v in P that has not been connected we check if a cut can be made between u and v through the function *Connectable*. If *Connectable* returns $true$, we find a possible connection through another function *MakeConnection*. Both functions check if a line can be drawn between polygons u and v given all the lines in the map including the cuts made so far. *MakeConnection* returns such a line, and this is added to *cuts*. We also set v as visited and add it to Q.

Exterior First Approach - Note that the process need not begin with the exterior to produce a connected graph of all polygons in P. However, the costs of the functions *Connectable* and *MakeConnection* is not negligible. Starting with the exterior increases the chance of connecting all or most of the holes in the first iteration of our *while* loop. This is because practically the probability of an obstacle being visible to the map border is better than not. *Fig 3.3* shows a map with the cuts visible in red.

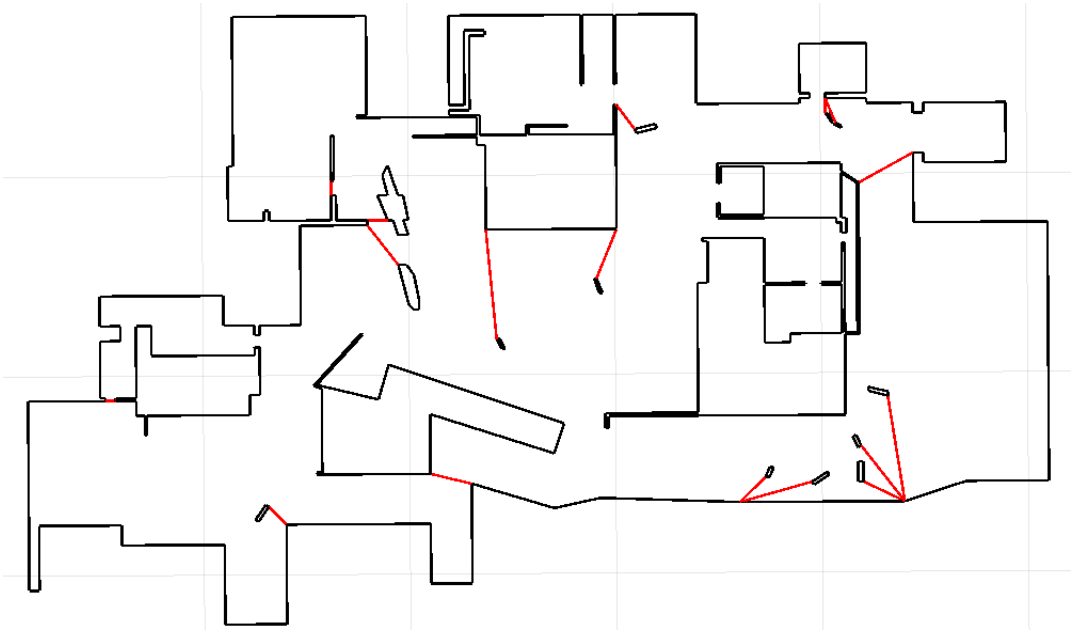

Figure 3.3 Map with cuts (represented by red lines)

Triangulate

Once the cuts have been computed via the spanning tree algorithm, the newly converted simple polygon can be triangulated. A valid triangulation can be obtained by simply drawing non-colliding lines or diagonals from each vertex of the polygon to the rest of the vertices until no such lines can be drawn. The triangulation lines are then used to create individual triangles, and the neighbours of each triangle are also determined.

Validation Functions - "Non-colliding" in the previous paragraph implies that any line drawn must not intersect any existing line except at endpoints, whether this line is an existing diagonal, a cut, an edge of a hole of the polygon, or an edge of the exterior of the polygon. For existing line segments this can be easily validated using line intersection checks. We call *MultiLineIntersection* for this purpose. However, it must also be ensured that diagonals do not fall outside of the map as certain cases are not caught by intersection checks. This is validated by checking that no points of the diagonal fall inside any of the simple polygons representing the holes, and no such points fall outside of the simple polygon representing the exterior. These checks are carried out in the *PolygonCollision*

function.

Triangulation Lines - This main algorithm for triangulation is described in *Algorithm 2*. The *Triangulate* procedure takes four parameters. The simple polygon representing the map boundary (*exterior*), the list of polygons containing the obstacles (*holes*), the list of lines containing cuts (*cuts*), and a list containing all the vertices of the map (V). To clarify, V contains vertices from both the holes and the exterior. We maintain a list of lines called *trilines*, which contains the lines of the triangulation. As the triangulation will inherently contain all edges of the polygon we add these in *trilines* from the onset. And since the *cuts* are also edges of the converted simple polygon, they will be added to *trilines* as well.

Individual Triangles - After all the lines for the triangulation have been obtained, we need to represent them as individual triangles. This representation, which is simply a set of 3 vertices, will help us in computing camera positions in the next section and in creating a "triangulation roadmap" in *Chapter 4*. We use a simple brute force approach as the triangles need to be calculated only once for each map. In the preprocessing portion efficiency can be compromised for simplicity. For each combination of 3 line segments from *trilines*, we try to create a triangle t using the *MakeTriangle* function. If each endpoint of a line segment in the combination can be found in exactly one of the other segments, and no two line segments overlap, then *MakeTriangle* returns a triangle, otherwise, it returns null. This triangle is represented by the *Triangle* data structure shown in *Algorithm 2*. *vertex* stores the coordinates of the 3 vertices, and *colours* stores the corresponding colours. There is also a list, *neighbours*, to store the indices of neighbouring triangles.

Setting Neighbours - Once all distinct triangles from the triangulation have been computed we have to set each triangle's *neighbours* list. A neighbour of a triangle is simply any triangles that shares an edge with it. Since, the triangulation is in respect to our converted simple polygon, a cut cannot be considered a shared edge. Therefore, we send both triangles and the list of cuts as parameters to the function *SharedEdge*, which determines if two triangles are neighbours. Lastly, our procedure returns *triangles*, which contains the triangles themselves and indicates their neighbours.

Polygon Representation - Note that the representation of the original polygon has not been changed after the spanning tree algorithm. We do this, firstly, because *cuts* only affect

Algorithm 2 Triangulation

```
1:  data type Triangle
2:      vertex[3]                          ▷ Array where each element is an (x, y) pair
3:      colours[3]  ▷ Array where each element is an enumerable: white, red, green, blue
4:      neighbours ← ∅                     ▷ A list of indices of other triangles

5:  procedure TRIANGULATE(exterior, holes, cuts, V)
6:      ▷ Get all the line segments of a triangulation
7:      trilines ← ∅
8:      for each edge e ∈ exterior do      ▷ Add the edges of the exterior
9:          trilines.Add(e)
10:     for each h ∈ holes do              ▷ Add the edges of every hole
11:         for each edge e ∈ h do
12:             trilines.Add(e)
13:     for each c ∈ cuts do               ▷ Add the cuts from the spanning tree
14:         trilines.Add(v)
15:     for each v1 ∈ V do                 ▷ Try all possible diagonals
16:         for each v2 ∈ V, v2 ≠ v1 do
17:             line = MakeLine(v1, v2)
18:             ▷ Validate the diagonal
19:             if MultiLineIntersection(line, trilines) = true then
20:                 Do Nothing
21:             else if PolygonCollision(line, exterior, holes) = true then
22:                 Do Nothing
23:             else
24:                 trilines.Add(line)
25:     ▷ Create triangles from line segments
26:     triangles ← ∅
27:     for each a ∈ trilines do
28:         for each b ∈ trilines, b ≠ a do
29:             for each c ∈ trilines, c ≠ b, c ≠ a do
30:                 T ← MakeTriangle(a,b,c)   ▷ On success returns a Triangle type
31:                 if T ≠ null and triangles.contains(T) = false then
32:                     triangles.Add(T)
33:     ▷ Set neighbours for each triangle
34:     for each s ∈ triangles do
35:         for each t ∈ triangles, t ≠ s do
36:             if SharedEdge(s, t, cuts) = true then
37:                 s.neighbours.add(t.index)
```

our triangulation through line collision, disallowing certain diagonals that may have gone through them. Secondly, it is easier to deal with the holes and exterior as separate simple polygons for the *PolygonCollision* function than it is to rearrange our original polygon.

Colouring and Least Cameras

The vertices of the triangles, and hence the vertices of the polygon, can now be coloured. *Algorithm 3* shows our procedures for doing so. We will continue to use the *Triangle* data structure described in *Algorithm 2*. The primary procedure here is *GetCameras*, which first calls a function that recursively colours every triangle's vertices. After that we find the colour that has been used the least number of times, and finally obtain our set of camera points by choosing vertices of this colour.

Colouring - We say two triangles are connected if they are neighbours i.e. they share a common edge that is not a cut. Since the triangles all fill up an enclosed space they form a connected graph. We use this notion in the *ColourNeighbours* function, which recursively colours all the vertices of a triangle's neighbours. The function takes two parameters. *parent* is the index of the triangle that called the function, and *child* is the neighbour of the parent that will be coloured. If this is the first triangle, we randomly assign the three different colours (red, green and blue) to each of the vertices. Otherwise, we find the two common vertices between parent and child, and ensure that the child's vertices have the corresponding colours. The vertex that is not common will get the third colour. Once the child triangle has been coloured we colour every unvisited triangle in its neighbour list by calling the same function with new parameters.

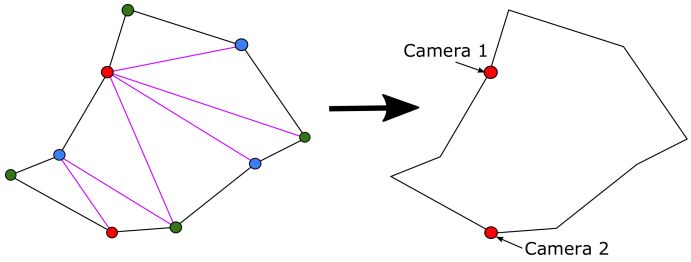

Figure 3.4 After colouring, the minimum occurring colour's points (in this case, red) is used as cameras.

Algorithm 3 Colour Cameras

```
1:  global variables: triangles
2:  procedure ColourNeighbours(parent, child)
3:      if parent = −1 then
4:          triangles[child].vertex[1] = red
5:          triangles[child].vertex[2] = green
6:          triangles[child].vertex[3] = blue
7:      else
8:          for i ← 1 to 3 do
9:              matchfound ← false
10:             for j ← 1 to 3 do
11:                 if triangles[parent].vertex[i] = triangles[child].vertex[j] then
12:                     triangles[child].color[j] ← triangles[parent].color[i]
13:                     matchfound ← true
14:             if matchfound = false then
15:                 thirdcolour ← triangles[parent].color[i]
16:         for i ← 1 to 3 do
17:             if child.colour[i] = white then
18:                 triangles[child].color[i] ← thirdcolour
19:     for i ← 1 to |triangles[child].neighbours| do
20:         if triangles[i].color[1] = white then
21:             ColorNeighbours(child, i)

22: procedure GetCameras
23:     ColourNeighbours(−1,0)          ▷ Colour the first camera and then its neighbours
24:     leastcolour ← FindLeastColour()  ▷ Finds the colour that has been used the least
25:     cameras ← GetVertices(leastcolour)  ▷ Gets all the vertices of a particular colour
```

Least Occurring Colour - Any set consisting of all the vertices of a single colour is a valid set of camera points. Since we want the minimum number of cameras we will pick the smallest such set. *FindLeastColour* determines which colour has been used the least times in the colouring process. Note that this has to be done for the set of unique vertices as the same vertex gets coloured multiple times by different triangles containing it. This is always guaranteed to be the same colour except if the vertex lies on a cut. For these vertices the colour used by triangles on both sides of the cut have to be counted.

Getting the Cameras - We use *GetVertices* to get the set of unique vertices that are all of

the colour chosen by *FindLeastColour*. This set is stored in *cameras*, which is simply all the cameras required for full coverage of the polygon.

3.2 Visibility Polygon

With the camera points in hand we can now device a method by which the NPC can visit cameras incrementally, guaranteeing that it sees the entire map by the end of its tour. But to understand how well any such method would work we will have to determine the exact amount of space the NPC sees from a point on its path. This can be done by calculating a visibility polygon. An example of one is shown in *Fig 3.5*. As mentioned before, a visibility polygon, VP, is usually computed from a point, p, inside a polygon, P. P maybe a simple polygon or a polygon with holes. VP consists of all the points in P that are "visible" to p i.e. all the points from which a straight line can be drawn to p without any obstruction. An intuitive way of looking at it is thinking of the point as a light source with rays of light coming out of it and hitting obstacles. The resulting illuminated area is VP. Since light cannot warp around, VP will not have any holes or self-intersect, and is therefore a simple polygon.

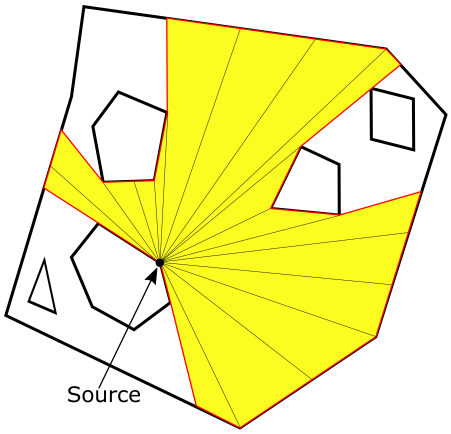

Figure 3.5 The illuminated area, shown in yellow, is visible to the source, and is bounded by its visibility polygon, whose edges are shown in red.

There has been a good deal of research on algorithms for computing visibility polygons. Along with complex approaches, there are some simpler ones that are easy to prototype.

However, creating a robust implementation that will work perfectly on large complicated polygons is not a trivial task. This is reflected in the fact that there is, to our knowledge, only a single existing publicly available implementation that works well in the scenarios we are dealing with. Even the highly popular C++ library, CGAL (Computational Geometry Algorithms Library), is still planning the release of the feature as of this writing. This maybe attributable to lack of formalization regarding the ordering of collinear points for simple approaches, the difficulty of implementing more complex approaches, and, perhaps most importantly, to problems with numerical stability i.e. floating point calculations. The first and last issues will be discussed later in this section.

Another point to note is that the final NPCs exploration path will be in the form of edges, and therefore it would seem to make more sense to consider visibility from an edge instead of a point. We do not use edge visibility algorithms because they are highly complicated and are computationally expensive (see *Section 6.1*). Instead we use point visibility from each endpoint of an edge as an efficient approximation. Later in *Section 5.2.4*, we experimentally test the strength of this approximation. To obtain the visibility polygon we need to compute its vertices as well as the order in which these vertices appear on the polygon's edge. Our algorithm works in two phases. *Section 3.2.1* describes the collection of initial vertices, and explains how line extension is used to gather the rest. *Section 3.2.2* outlines the procedure for putting these vertices in the correct order.

3.2.1 Computing Vertices

It helps to think of the point from which the visibility polygon is being computed as a light source. A light source projects infinitely thin rays of light from all sides. These rays can stop right at the source or travel infinitely depending on the position of obstacles around the source. For sources inside a polygon, however, they will eventually stop at some edge of the polygon, either of the exterior or of a hole. All turns and corners in such a visibility polygon will be due to corners of the map i.e. its vertices, or points on an edge that are collinear to some vertex. Therefore, to find the vertices of a visibility polygon, VP, we need only consider the rays that intersect the vertices of the polygon with holes, P.

The source from which a visibility polygon is computed is usually referred to as a

kernel. We will, henceforth, be using this term. *Algorithm 4* describes how we shoot and extend rays from the kernel to the vertices of P to figure out which points are visible. It is based on the well known angular plane sweep algorithm by Suri et al [36].

Initial Points

We will first create lines from the kernel to every vertex of the polygon, P, and check if this line intersects with any of the polygon's edges. If it does not then this vertex is visible from the kernel, and we will add it to our list of initially visible points. We will also consider collisions that take place at corners i.e. the endpoints of edges. This means that among multiple vertices of P, that fall on the same ray from the kernel, only the one closest to the kernel will be added to the list of initial vertices (*Fig 3.6*). This is done for numerical stability reasons. Each point is stored as a *VisibilityPoints* data type. A *VisibilityPoints* element contains both the coordinates of a point and the angle it makes with the kernel. The *GetPoint* function is used to get an element of this type.

Numerical Stability

Implementations of computational geometry algorithms often run into numerical stability issues. This is especially true when the geometric space is complex and has a wide range of dimensions. In most programs, real numbers are stored as floating point numbers. Floating points are approximations of real numbers, and cannot always store all the digits needed to represent them. Therefore, as we compute and reuse floating points in calculations, we begin to lose information about the actual real number. When we say "numerical stability", we are referring to this loss of information and its potential to affect logical outcomes in a program.

In general, we counter floating point problems by using a machine epsilon value (e.g 10^{-5}), which is a marker for what number of decimal places we will consider in our calculations. We consider up to 5 digits after the decimal point. This number was experimentally derived and works for the scale of our maps. Choosing an epsilon value is tricky as it has a relation with the dimensions of the objects in the geometric space. As a result higher values do not guarantee improvements, and did not do so in our case.

Algorithm 4 Compute Visibility Polygon Points

```
1:  data type VisibilityPoint
2:      point                                              ▷ An (x,y) tuple
3:      float angle
4:  procedure GETVPPOINTS(V, kernel)          ▷ V is a list of the polygon's vertices
5:      VPInitial ← ∅
6:      for each v ∈ V, v ≠ kernel do
7:          line ← makeLine(kernel, v)
8:          if PolygonCollision(line, exterior, holes) = false then
9:              ▷ GetPoint returns a VisibilityPoint type containing the angle
10:             VP.Add(GetPoint(v, kernel))
11:     ▷ Sort by angle and deal with numerical stability issues
12:     VPInitial ← ClosestPointsUniqueAngles(VPInitial)
13:     VPExtended ← ∅
14:     for each vpt ∈ VPInitial do
15:         currentangle ← vpt.angle
16:         a ← kernel
17:         b ← vpt.point
18:         VPExtended.Add(b)
19:         while b ≠ null do
20:             ▷ Extend the ray going from a to b and return the closest intersection point
21:             newpoint ← MorePoints(a, b)
22:             if newpoint ≠ null then
23:                 a ← b
24:                 b ← newpoint
25:                 ▷ VisibilityPoint() simply returns the data type of the same name
26:                 VPExtended.Add(VisibilityPoint(b, currentangle))
27:     return VPExtended
```

For small maps of our own design, the technique of using a machine epsilon value works quite well. But visibility polygons require very acute calculations in the geometric space, and therefore some issues remain, which we deal with in alternative ways. For example, we stated that we ignored points connected by rays that have intersections with corner points i.e. if multiple points are on the same ray, and therefore have the same angle, we chose only the one closest to the kernel. But in reality, even when the line intersection function says there are no corner intersections, the angle calculations show points with the same angular

value. *ClosestPointsUniqueAngles* function checks against this and keeps only the point on a ray closest to the kernel. Here, our approach is to double check the angles, and consider those that have the same angular value (according to a machine epsilon comparison) to be collinear or on the same ray from the kernel.

Extending Rays

The items in *VPInitial* can be thought to represent the rays that have been shot out from the kernel. We will now take each ray, and find the remaining points where it intersects with polygon, P. Note that not all of the intersection points, including the ones we have so far, will necessarily be vertices of our visibility polygon. Rather, these are all points of interest that we will require later to correctly order the vertices. All points of interest are now added to a list called $VPExtended$, whose elements are of type $VisibilityPoint$, which contains both the coordinates and angle of a point. In *Fig 3.6*, the image in the middle shows extension points in red.

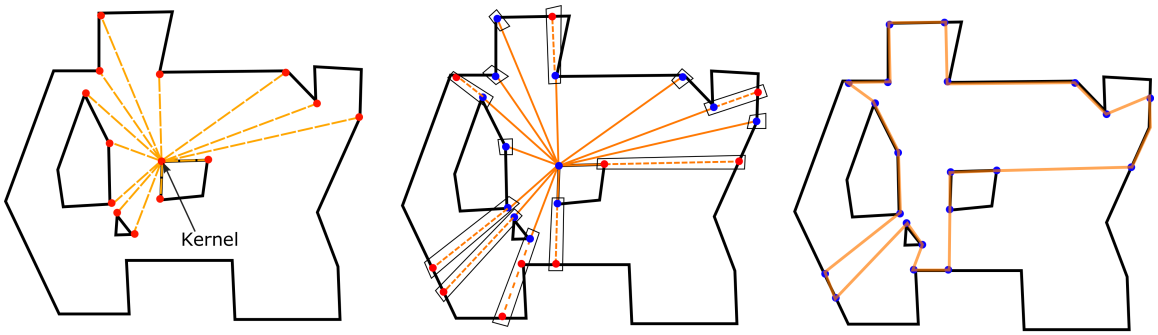

Figure 3.6 From left to right, the three stages of getting initial points, extension points and then ordering to get the visibility polygon.

Collecting Extension Points - For each point in $VPInitial$ we use the *MorePoints* function to get our extension points. *MorePoints* takes two points, a and b, and draws a ray in the direction of b that is long enough to go outside the polygon representing the map. It then finds and returns the intersection point closest to b. We then add this new point to $VPExtended$ along with the angle it forms with the kernel, which is simply the angle for the first vertex on this ray and iteration. At the beginning of each iteration the point from

VPInitial is added as well. Hence, upon termination we will have all the points sorted by angle and distance. Assigning the same angle ensures that no problems occur due to small differences in angular value calculations, and collinear points are correctly identified as such. It is possible to rewrite *MorePoints* to send all the collinear points on the ray in a single iteration. But the function is already quite complex, especially because it has to figure out if the ray does anything invalid (going inside a hole or outside the exterior) before intersecting the next point. Therefore, in the interest of abstraction and ease of implementation, we chose to compute one point at a time.

3.2.2 Ordering

After gathering all the necessary points, we have to order them correctly to get a valid visibility polygon. A valid visibility polygon will cover only the portion of the map visible to the kernel, and will be a simple polygon i.e. not self-intersecting and without any holes. *Algorithm 5* outlines the ordering process we have created.

By Angle

As we iterate through the points to order them, we will consider two groups of points at a time instead of individual points. Any particular group will consist of all the points that lie on the same ray from the kernel (i.e. collinear points). As such points have the same angle, they will appear consecutively and in order of distance in *VPExtended*.

Connecting Groups

At each iteration of the ordering algorithm we take two consecutive groups (rays), the first of which is *listA*, and the second, *listB*. Our aim is to carry out two different types of actions every time. We want to 1) remove any extraneous points, and 2) add the elements of *listA* in the correct order to the visibility polygon. The latter issue will be discussed first. Generally there are two possible scenarios when trying to connect two rays. Either there is a line that can be drawn directly from a point on one ray to the other, or they are joined by the kernel. This happens because all our kernels are polygon vertices, by virtue of being

camera points. In any other case the kernel would not be part of the visibility polygon's vertex set.

Principle of Connectivity

Recall that the points in each group or ray are sorted by their distance from the kernel, the first point being the closest, and the last being the farthest. It is important to have this sense of order as we talk about connectivity. Two consecutive rays A and B may hit many edges or obstacles. We are only concerned with the ones that lie in the space *between* them. Of these occlusions the one closest to the kernel, and hence visible to it, will *always* connect the two rays. That is to say this will be an edge that has an endpoint in A and an endpoint in B. Otherwise, whatever occlusion is present would have a vertex or multiple vertices that appear in between ray A and ray B, implying there are rays between the consecutive rays we have chosen.

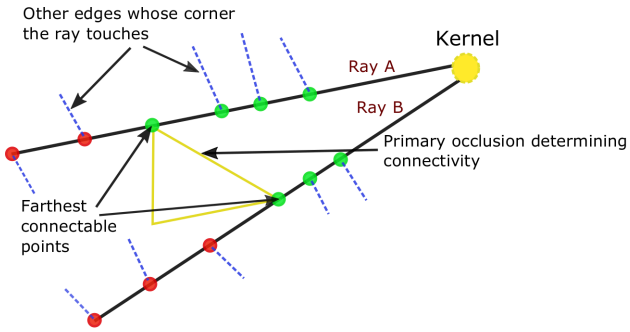

Figure 3.7 Connectivity issues between rays

It follows that points in group A appearing past the first occlusion cannot connect to points in group B appearing before this occlusion, and vice versa. This is illustrated in *Fig 3.7* where the green vertices on one ray cannot make valid connections to the red vertices on the other ray. Now, consider the subset of points in A and subset of points in B that do not have any obstacles between them i.e. all points of both subsets can be connected. It is sufficient to find the farthest points in each of these "connect-able" subsets to determine where the connection between the two rays should be.

An Intuitive Approach

Visibility polygons often have a shape where the angles between consecutive edges alternate between convex and concave, producing a "zig-zag" shape. Therefore, connections between rays tend to alternate between last points (convex angles) and first points (concave angles), as illustrated in *Fig 3.8*. But because of complex obstacle layouts we run into a plethora of corner cases that can initially make a generalized ordering algorithm seem out of reach. It is helpful to visualize this "zig-zag" movement as we build a robust algorithm that tackles all the nuances of ordering points.

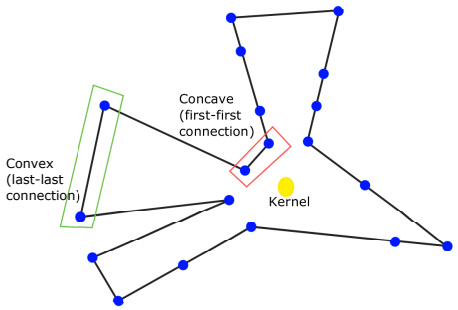

Figure 3.8 The "zig-zag" intuition of connecting rays

Connecting by Kernel

Normally a visibility polygon does not contain the kernel as one of its vertices. The situation arises here because our kernel is always a camera point, and therefore a vertex of the map. There are 3 cases when two rays have to be connected by a kernel (see *Fig 3.9*). The most obvious of these follows from the ideas discussed above. If the first point of *listA* cannot be connected to the first point of *listB* then the rays cannot be directly connected in a valid manner. Even if the rays were connected at some other pair of points it would create a polygon area with a hole inside it, which is not allowed as the visibility polygon is a simple polygon. The second case occurs when a direct line can be drawn between the two first points but this line is occluded from the kernel by an obstacle. Finally, we also use the kernel when the difference in angle between the rays is more than or equal to 180 degrees. As this means they cannot be joined by a straight line or that the straight line joining them would go through the kernel.

Algorithm 5 Ordering Visibility Polygon Points

```
1:  data type VisibilityGroup
2:      pointlist                                   ▷ List of points a ray intersects
3:      angle                                       ▷ Angle of the ray
4:  procedure ORDERVPPOINTS(VPExtended, kernel)
5:      VPGroups ← GroupPoints(VPExtended)          ▷ A list of VisibilityGroup elements
6:      VPGroups.Add(VPGroups[1])                   ▷ Last ray has to be connected to first ray
7:      VPOrdered ← ∅                               ▷ The vertices of the visibility polygon in order
8:      for i ← 1 to |VPGroups − 1| do              ▷ Ordering and adding points on the i-th ray
9:          listA ← VPGroups[i].pointlist
10:         listB ← VPGroups[i + 1].pointlist
11:         if ArePolygonEdges(kernel, listA[1], listB[1]) = true then
12:             ▷ Case 1: Has to be indirectly via kernel
13:             listA.Reverse()
14:             VPOrdered.AddMultiple(listA)
15:             VPOrdered.Add(kernel)
16:         else
17:             ▷ Case 2: Find farthest connectable points
18:             connectorA ← GetConnector(listA, listB, 1)
19:             connectorB ← GetConnector(listB, listA, connectorA)
20:             ▷ Remove extraneous points
21:             if connectorA! = 1 and connectorA! = |listA| then
22:                 listA.RemoveAllAfter(connectorA)
23:             if connectorB! = 1 and connectorB! = |listB| then
24:                 listB.RemoveAllAfter(connectorB)
25:             if connectorA = 1 then
26:                 listA.Reverse()
27:             VPOrdered.AddMultiple(listA)
28:             VPGroups[i + 1].pointlist ← listB   ▷ For use in the next iteration
29:     return VPOrdered
30: procedure GETCONNECTOR(listA, listB, target)
31:     connector ← 1                               ▷ We know the first element can connect to target
32:     for j ← 2 to |listA| do
33:         if connectable(listA[j], listB[target]) then
34:             connector ← j
35:         else
36:             break
37:     return connector
```

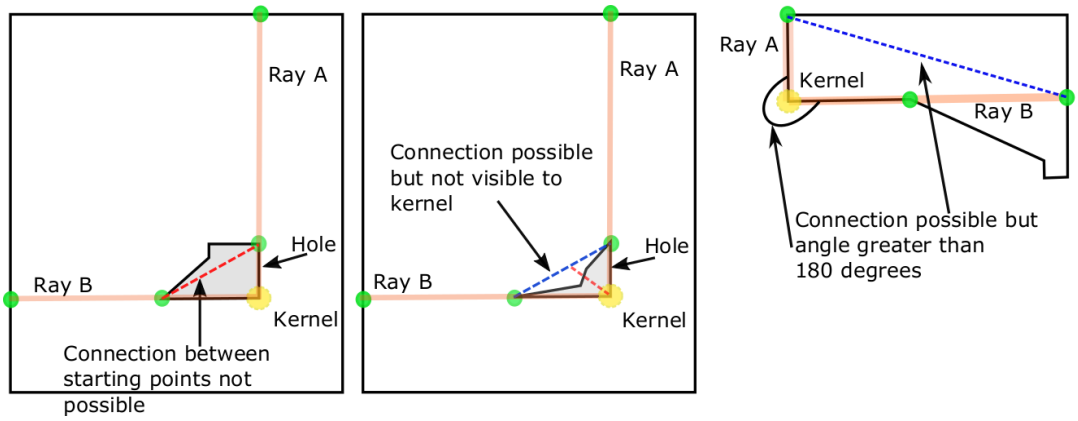

Figure 3.9 Cases where rays have to be connected by the kernel

We observe that all three cases have a common feature. If we create a line between the kernel and the first point of each ray, both lines will be edges of either the exterior or of the holes. *ArePolygonEdges* is responsible for checking this condition. If it returns true, the points of *listA* will be added to the visibility polygon from last to first i.e. in reverse order since they are moving into the kernel. After that the kernel is added, and in the next iteration the points of *listB* will end up being added from first to last, as they are moving outward from the kernel.

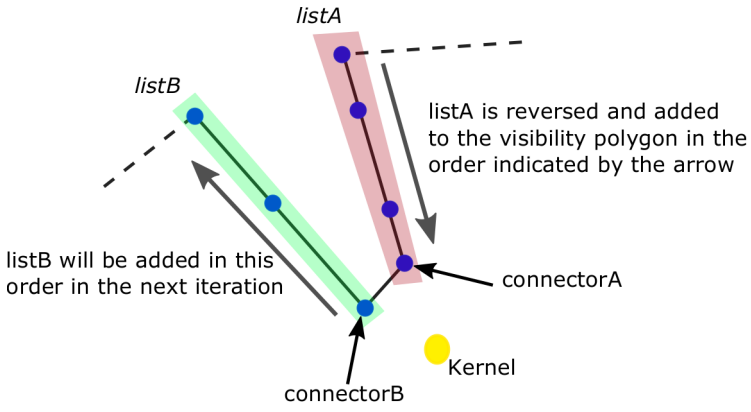

Figure 3.10 Adding *listA* when *connectorA* is the first point

Farthest Connect-able Points

If the rays do not need to be connected by the kernel then according to the principle of connectivity it is possible to connect them with some edge. Many connections maybe possible but from these we must choose the one that is farthest on the ray. We call these points *connectorA* and *connectorB*, for *listA* and *listB*, respectively. *GetConnector* is used to find these points. It incrementally goes through the points in the first list in its parameters, and tries to connect it to a target point, *target*, in the second list. It then returns the last point that was successful in making such a connection. *connectorA* is the farthest point in *listA* that can connect to the first point in *listB*, whereas *connectorB* is the farthest point in *listB* than can connect to *connectorA*. If *connectorA* is the first point in *listA*, then *listA* will have to be added to the polygon in reverse order (*Fig 3.10*), as *connectorA* is the last point before the points of the next ray are added in the next iteration.

Extraneous Points

Portions of some rays are either not visible to both the preceding and succeeding rays. Sometimes they are visible but no valid connections can be made to these subsets of points. *Fig 3.11* shows extraneous points in red boxes. The only way such a subset of points can be added to the polygon is if we take this ray's farthest connect-able point, draw an edge out from it to the last point, then draw an edge back into the farthest connect-able point, and continue connecting the rays the way we have been. This would, however, create overlapping polygon edges and mean that our visibility polygon is complex. Such an output would make merging polygons in the aggregated coverage algorithm, used later, more complex. Additionally, in terms of visual information it represents very little as it is only a single ray of light, and would not even be discernible in the case of a human player. That is why we choose to remove such points and form our visibility polygon without them.

Detecting extraneous points is fairly simple. By the "zig-zag" intuition, rays will always connect either at their first point or their last point. Therefore, if any connector appears somewhere in the middle, this point can be considered the last point and all others after it removed. Note that this process can also be done after we have ordered all the polygon points.

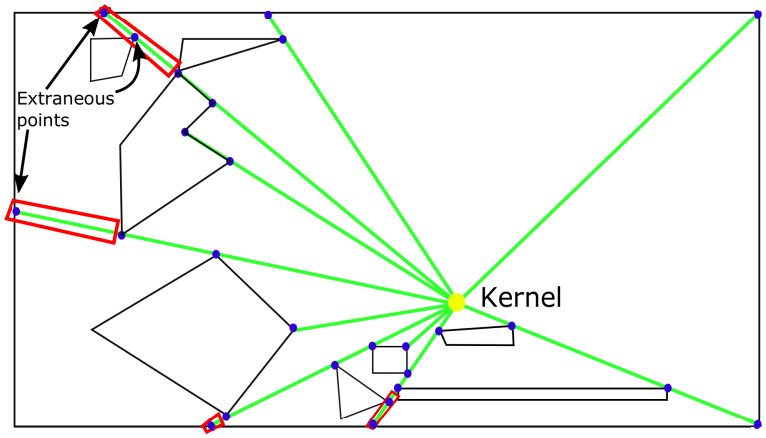

Figure 3.11 The red portions contain extraneous points that have to be ignored

3.3 Merging Polygons

Now that we have the tools to measure the exact amount of coverage from a single point, we want to measure the aggregated coverage from a set of points i.e. the total amount of the map that can be seen from a set of cameras. Since coverage is defined by the visibility polygon, which is a simple polygon, aggregated coverage will be represented by a union of multiple polygons. To obtain the union we merge two polygons at a time. The merger of two simple polygons may create a polygon with holes, and our algorithm accounts for that. The ultimate result will be a polygon that covers all the points, and only the points, that are in the polygons we are merging. *Fig 3.12* shows an example of such a merger.

We merge polygons in two steps. *Section 3.3.1* describes how intersecting edges between the starting polygons are decomposed into smaller edges. *Section 3.3.2* describes the process for filtering out the invalid instances of these smaller edges to obtain the final merged polygon. In this section, the visibility polygons are represented as a set of edges. These edges can be easily produced by connecting the ordered set of vertices obtained in the previous section.

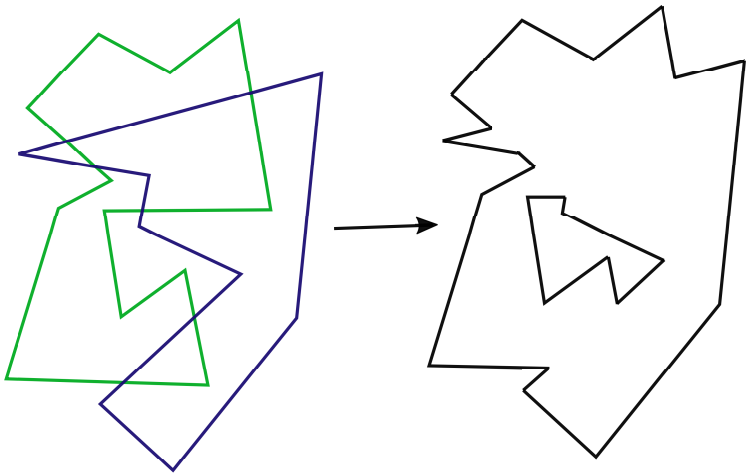

Figure 3.12 Merging two polygons

3.3.1 Decomposing Edges

First note that two polygons can only be merged into a single polygon if they intersect, or one polygon is fully inside the other. If the polygons are disjoint, the union of the geometric space they cover would have to be represented by two separate polygons i.e. the original polygons. We are concerned with the case where intersection takes place. If a polygon is fully inside another, the result is simply the larger one.

If a polygon A intersects a polygon B, and neither is fully inside the other, then at least one edge from A will intersect at least one edge from B. Our aim is to decompose edges that intersect. Every intersection will have one or more intersection points. Edges will be broken down into smaller line segments where the intersection point(s) become endpoints (see *Fig 3.13*). Lines may intersect at a single point (regular intersections) or may overlap (collinear).

Algorithm 6 outlines our decomposition process. *DecomposeEdges* takes the edges of two polygons $P1$ and $P2$, breaks them down, and puts the decomposed edges in another polygon, $P3$. First, all the edges from $P1$ and $P2$ are added to $P3$'s edge list, which is initially empty. There are no duplicates in this list. Then we iterate through the edges in $P3$, and for each consider all the edges that come after it in the sequence. The two line segments in consideration at each stage are represented by LA and LB. $Endpoints$ is a list

Algorithm 6 Merging Polygons

```
1:  procedure DECOMPOSEEDGES(P1, P2)
2:      P3.edges ← P1.edges ∪ P2.edges
3:      for each LA ∈ P3 do
4:          for each LB ∈ P3, LB ≠ LA do
5:              casetype ← LineIntersection(LA, LB)
6:              endpoints ← GetEndPoints(LA) ∪ GetEndPoints(LB)
7:              if casetype = "nointersection" then                ▷ No intersection
8:                  continue
9:              else if casetype = "regular" then      ▷ Non-collinear, no shared endpoint
10:                 interpoint ← GetIntersectionPoint(LA, LB)
11:                 for k ← 1 to |endpoints| do
12:                     P3.edges.Add(MakeLine(endpoints[k], interpoint)
13:             else if casetype = "overlapping" then          ▷ Collinear and overlapping
14:                 endpoints ← SortXY(endpoints)
15:                 for k ← 1 to |endpoints| − 1 do
16:                     P3.edges.Add(MakeLine(endpoints[k], endpoints[k + 1]))
17:             P3 ← P3 − {LA, LB}                             ▷ Remove redundant original edges
18:             break
19:     return P3
20: procedure MERGE(P1, P2)
21:     decomposed ← DecomposeEdges(P1,P2)
22:     filtered ← Filter(decomposed,P1,P2)
23:     return filtered
```

containing the endpoints of the two lines. We use *LineIntersection* to determine whether there is an intersection, and, if so, what type. If there is none, no action is taken. Otherwise, segments are broken down according to the intersection type, and added to the end of $P3$'s edge list. LA and LB are then removed from $P3$ as their composites have just been added, and we break from the current loop as $P3$'s element composition has changed.

Regular Intersections

This case happens when the intersecting line segments are at an angle to each other. Intersections that take place at a shared endpoint are ignored. We first find the intersection point using *GetIntersectionPoint*, and then create lines from this point to each of the points

in *endpoints* (*Fig 3.13*).

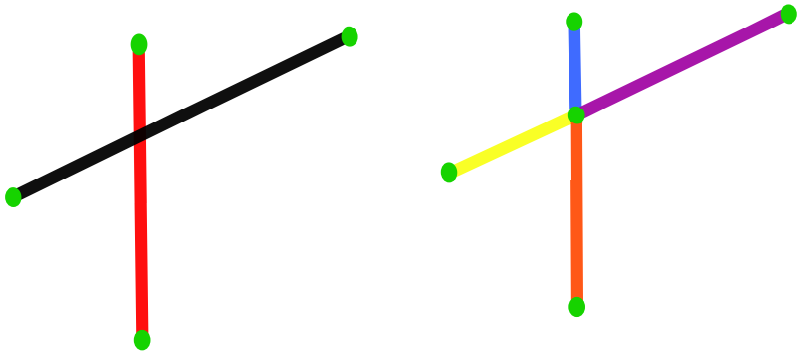

Figure 3.13 Breaking down regular intersecting edges

Collinear and Overlapping

Line segments will sometimes overlap i.e. they will intersect at more than a single point. This only happens when lines are collinear (see *Fig 3.14*). In these cases, the endpoints of the two line segments are all collinear to each other. We want to make smaller line segments between adjacent endpoints. Therefore, all the endpoints are sorted by their x coordinates, and if these are equivalent then their y coordinates. After that, we take each point and create a line segment with the next point in the sorted sequence.

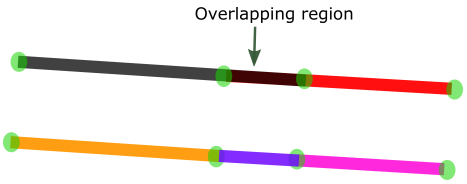

Figure 3.14 Breaking down collinear intersecting edges

3.3.2 Filtering Edges

Some of the decomposed line segments we have computed are not valid edges of the merged polygon because they lie inside it. We can detect and remove them by the following methods.

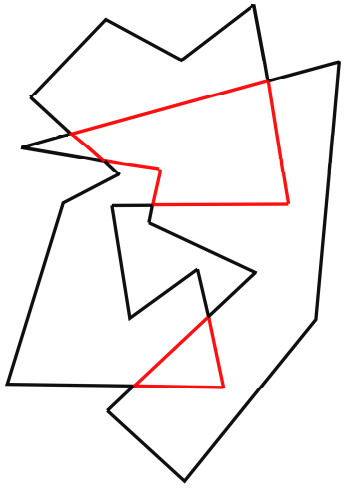

Figure 3.15 Decomposed edges (shown in red) that need to be removed to get the final polygon

Inside Original Polygons

Most of the invalid edges fall inside either polygon $P1$ or polygon $P2$, but not both as such segments would have been broken down in the decomposition stage. Therefore, we only need to check if the midpoint of segment falls inside either $P1$ or $P2$.

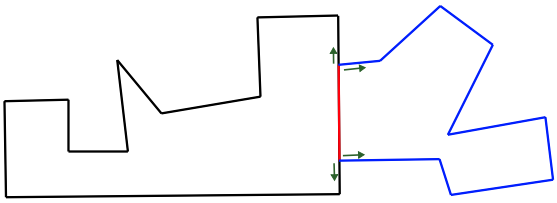

Figure 3.16 Invalid edge (shown in red) on shared boundary that does not fall inside either polygon

On Shared Boundary

In some cases the invalid edge is a line segment that is on the shared boundary of both $P1$ and $P2$, but is otherwise inside the merged polygon. This means that the line is not inside either $P1$ or $P2$ but is inside the merged polygon. Since it is a shared boundary line it will have exactly two other lines (one in $P1$, one in $P2$) going out from each of its endpoints. As a result, the invalid line will be connected to a total of four lines. For valid lines this

number will usually be a total of two, and at most three. Therefore, invalid edges on shared boundaries can be filtered out on this basis (see *Fig 3.16*).

Chapter 4
Roadmaps and Tours

This chapter describes how exactly an NPC should move about in a game map, what is the path it should walk, and what decisions it may need to make at different points of its walk. These questions fall under a class of problems known, in both video games and robotics, as motion planning or pathfinding. We have so far determined a set of camera points. We know that visiting these will guarantee the NPC complete visual coverage of the map. In addition, we know how to measure the exact amount of coverage from these points, both individually and collectively. Our task now is to specify in geometric terms the exact movement by which the NPC will be able to visit all cameras points during the game. To be clear, we are not talking about animation or movement related to specific parts of an NPC's body. In our model, the NPC remains a point and we are concerned with the displacement of this point inside a polygon with holes.

Our solution is two part. We create a planar graph of straight lines (edges), the endpoints of which can be considered the nodes. This planar graph is called a *roadmap*, and it will ultimately contain the computed camera points in its node set. Therefore, travelling between any two camera points now becomes a graph traversal problem. This traversal is called a *tour*. We try to traverse the roadmap in different ways, and each method produces a different type of tour. Each tour uses either distance, coverage, or both as heuristics. This results in the NPC uncovering the map at different rates.

Section 4.1 deals with the creation of roadmaps. Shortest path and triangulation roadmaps are discussed in *Section 4.1.1* and *4.1.2*, respectively. Tours are described in *4.2*. *4.2.1* out-

lines how we use Dijkstra's algorithm to preprocess distance calculations, and *4.2.2* details the different types of tours and the heuristics used in them.

4.1 Roadmaps

Whenever a polygon is concave and/or has holes, the task of going from a point A to a point B becomes non-trivial, since a straight line between the two might be met by many obstacles. At its simplest, the connection between A and B will be a series of connected, non-collinear, straight lines. However, in a continuous geometric space the possible combination of such lines are infinite. A roadmap is a planar graph that give us a defined set of lines to work with, allowing us to easily generate valid paths (i.e. ones avoiding obstacles) in such a space. A roadmap can be thought of as a network of roads and highways that the NPC can hop on to get from one place in the map to another. The "hopping on" happens by connecting both the source, A, and the destination, B, to the roadmap. In that sense there is an initial set of nodes and a final set of nodes.

There are many different types of roadmaps. While they aim to do similar things, a large part of the variation lies in the methods used to compute them, specifically the efficiency and ease of implementation of these methods. There are also differences in the properties of the generated paths. Edges in one roadmap might be short and concentrated, meaning there are lots of turns. While in another, there might be a focus on maintaining the maximum possible distance between edges and obstacles. These have different consequences when considering how players perceive NPCs as they move along roadmaps.

A lot of roadmaps are created using a technique called cell decomposition. It is a way of dividing up a polygonal space into cells, and then using some method to connect these cells with edges. Another class relies on computational algebraic geometry, a theoretically powerful approach that is often inefficient and extremely difficult to implement [26]. The solution we will be primarily using, shortest-path roadmaps (*4.1.1*), does not fall into either category. It aims to create the shortest possible path between points in the space. We will also look at triangulation roadmaps (*4.1.2*), which use the cell decomposition approach. We obtained the ideas and rules for both roadmaps from the book *Planning Algorithms* by Steven LaValle [26].

4.1.1 Shortest-Path Roadmap

Shortest-path roadmaps are created by finding *reflex vertices*, and then connecting these vertices where appropriate. Now, if we want to use the roadmap to find a path from a point A to a point B, we must connect A and B to all possible vertices of the roadmap. After that the roadmap can be used as a graph to find a particular path between the two points. In our case, we will be doing this for all our camera points.

In a shortest-path roadmap the initial nodes are a subset of the polygon's set of vertices. Many of the edges go along the sides of the holes or the exterior i.e. they are edges of the polygons, and some are long straight lines that cut across the space. This is because the aim is to always find the shortest path whether it is in an open space or when navigating around obstacles. A lot of the nodes in the graph that are visible to each other will most likely be connected by a direct edge. This is why shortest-path roadmaps (see *Fig 4.1*) are also known as *reduced visibility graphs*. The roadmap can become quite dense, which is why a magnified portion is also shown in the figure. In the example, cameras have not yet been connected. We will now describe reflex vertices and the rules for connecting them.

Reflex Vertices

Reflex vertices are vertices of the polygon with an interior angle strictly greater than 180 degrees. This applies to vertices on the holes as well. The interior angle is the angle falling on the inside of the polygon. Therefore, it should be noted that if we choose to treat the holes as separate simple polygons, then their reflex vertices will be the ones with an exterior angle greater than 180 degrees.

Connecting Reflex Vertices

We consider every pair of reflex vertices and connect them with an edge if they meet one of two conditions. First, when there are two **consecutive reflex vertices** i.e. they lie on the endpoints of an edge of the polygon. And secondly, when a **bitangent edge** can be formed. A bitangent edge is an edge between two points that are visible to each other. It must be possible to slightly extend the edge in both directions, and this extension should be strictly

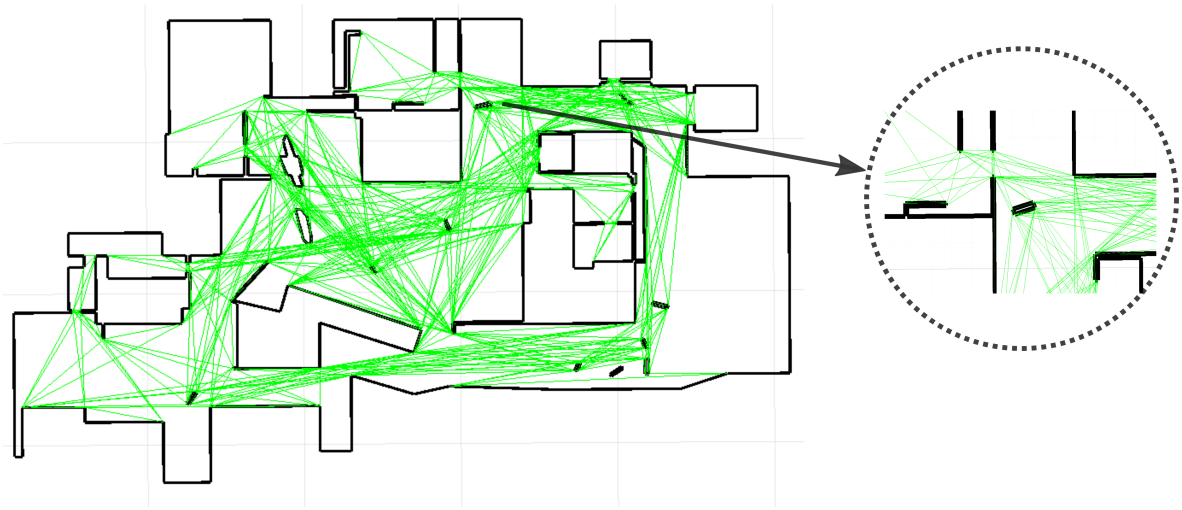

Figure 4.1 Shortest-path roadmap (represented by green lines)

interior to the polygon.

Connecting Cameras

We take each camera and connect it to every visible reflex vertex. If two cameras are visible from each other we connect them as well since that would be the shortest path between them.

4.1.2 Triangulation Roadmap

As indicated by the title, triangulation roadmaps rely on decomposing the map by way of triangulation. Once the polygonal space has been divided into triangles, we will create edges connecting them. This will form a connected graph of edges that is our roadmap. We already have a triangulation from *Section 3.1.2*. We connect triangles by drawing a line from the centroid of a triangle to the midpoint of each of its shared sides. A shared side is a side that the triangle shares with a neighbouring triangle. Once the roadmap has been calculated we connect the cameras in the same manner as before. A triangulation roadmap will inherently produce longer paths and tours compared to a shortest-path roadmap. *Fig 4.2* shows a triangulation roadmap before being connected to cameras.

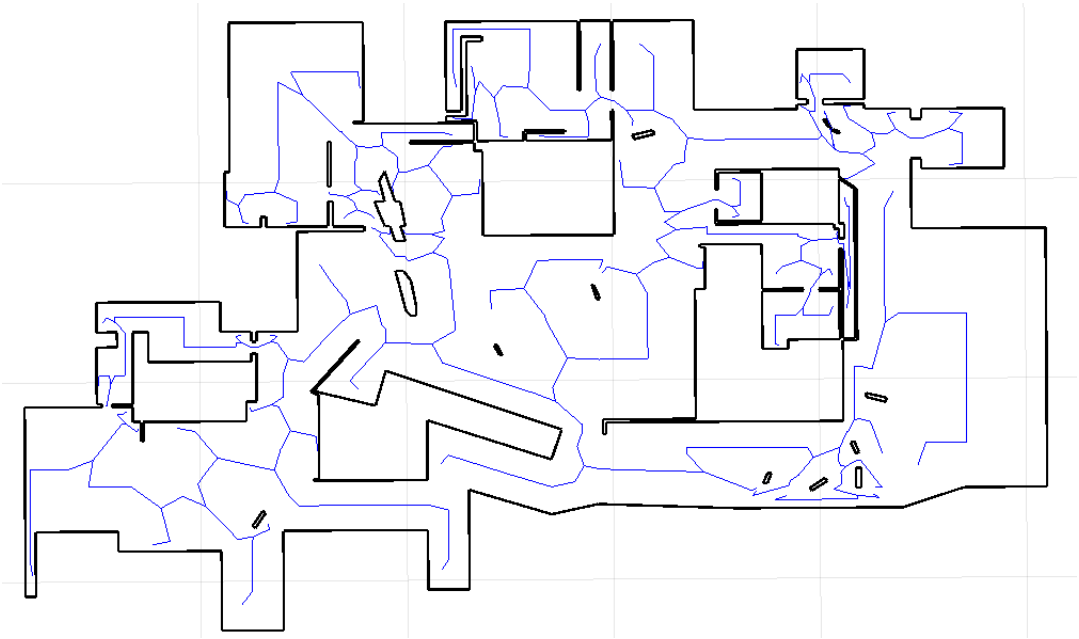

Figure 4.2 Triangulation roadmap (represented by blue lines)

4.2 Tours

Once we have a roadmap, we can use it to compute paths between cameras. And by connecting paths to visit all cameras, we can create the NPC's exploratory tour of the game map. The tour's geometric nature is tied to the particular roadmap being used. For example, if we use a shortest-path roadmap, the tour will be a series of straight lines between reflex nodes and cameras. For a triangulation roadmap it will be smaller straight lines that connect cameras, the centroids of triangles and the midpoints of triangles. It is apparent that once we have chosen our roadmap, this particular geometric element of the tour becomes fixed. But considering the number of nodes and edges in the graph, as well as the number of possible paths between any two nodes, other decisions we make will still significantly impact the characteristics of a tour.

Of these characteristics, the two that have immediate relevance to exploration are a) the total length of a tour, and b) the rate at which the map is seen or uncovered. The decisions that will affect these characteristics are described in *Section 4.2.2*, where we describe how different types of tours can be created. To make any decision regarding

distance and coverage we need to have a method of calculating their values. In the case of coverage, we use the visibility polygon and polygon merger algorithms from *Chapter 3*. For distance, we will use Dijkstra's shortest path algorithm, which is discussed in *Section 4.2.1*.

4.2.1 Dijkstra

A roadmap is a planar graph consisting of nodes and weighted edges. The weight of each edge is simply its length in the Cartesian plane. We are faced with the problem of determining a path between any two nodes in such a graph, and of finding the length of that path. Because of how this information will be used later on, we need to specifically calculate the optimal path or the shortest possible path. Dijkstra's single-source shortest path algorithm is a convenient way of doing that. Given how well known the algorithm is, our discussion of it will be brief.

Dijkstra's algorithm computes the shortest possible path in a graph G, from a source node u to all other nodes. It does so by maintaining a distance value and a parent value for every node. For each node, v, the distance value, $d_{u,v}$, is simply the shortest distance from u to v known at the time. If we consider the current known path from u to v as a sequence of successive nodes $v_1, v_2 ... v_{n-1}, v_n$ where $v_1 = u$ and $v_n = v$, then v's parent value, p_v, is simply the index of the node that appears immediately before v in the sequence i.e. v_{n-1}. The algorithm works by taking a node and updating the distance values of its neighbouring nodes using its own value and that of the edges connecting it to its neighbour. Whenever an update takes place the parent value is also changed. Once the algorithm terminates, it is guaranteed that each node's distance value is the distance of the shortest possible path from u to the node. In addition, the final parent values can be used to recursively find out the exact sequence of nodes in that shortest path.

Note that instead of Dijkstra we could have used the A* algorithm, which is a very popular solution for path-finding problems. But we can afford to use the more computationally expensive Dijkstra approach, especially because we run the algorithm on every node of the roadmap once and save all the results. This preprocessing is useful since we repeatedly use the same path information many times when constructing a tour. We do something similar

with coverage by computing the visibility polygons of all nodes beforehand.

4.2.2 Types of Tours

Now that we can compute shortest paths between cameras, we can combine the paths to form a tour visiting all cameras. So far a tour has been described as a visit to all camera points. This was done for the sake of comprehensibility. The point of visiting all the cameras is to mathematically guarantee that the full map will eventually be seen. In reality, we check the coverage from all traversed roadmap nodes while the tour is being constructed, not after the whole process is complete. As a consequence, it is possible that the map will be fully seen before all cameras have been visited. Therefore, during the analysis of the experiment results, we consider the tour to have ended once the map has been fully covered.

When constructing a tour, our primary motivation (discussed further in *Section 5.1.1*) is to minimize its total length, which can only be done by covering the map as quickly as possible. However, unlike the individual paths computed using Dijkstra, finding a tour that is optimal is quite difficult. The number of possible tours of all cameras is exponential in the number of nodes and edges in the graph. The tour problem itself is in essence a version of the Travelling Salesman Problem (an NP-hard problem), the difference being that here the distance is inherently tied to coverage. Therefore, it is computationally intensive and impractical to find a tour with optimal distance or coverage for an NPC during gameplay. That is why we employ an approach that can be thought of as heuristic or greedy. We will now describe the general framework used for our tours followed by the specifics of each of the five tour types.

Framework

We start the tour from a node in the roadmap. We then choose an unvisited camera to go to, and use the path generated by Dijkstra's algorithm to go there. Once we are at this new camera, we try to find a new unvisited camera in the same manner and continue the process until all cameras have been visited, or earlier if the full map has already been uncovered. The choice we make at each camera point is based on a specific heuristic and represents the "greedy" aspect of our algorithm. Each heuristic was designed with an intuition or

hypothesis in mind. We test out this intuition later in the experimentation phase. Each type of tour we have used is characterized by a different heuristic.

An example of a tour is shown in *Fig 4.3*. The NPC starts from the camera point (marked in green), and follows along the path represented by the blue edges until it reaches the endpoint (marked in red). Sometimes an edge will be traversed twice, on the way to a camera and from it to another camera. This is why, in the diagram, it sometimes looks like the path ends and stops at certain points.

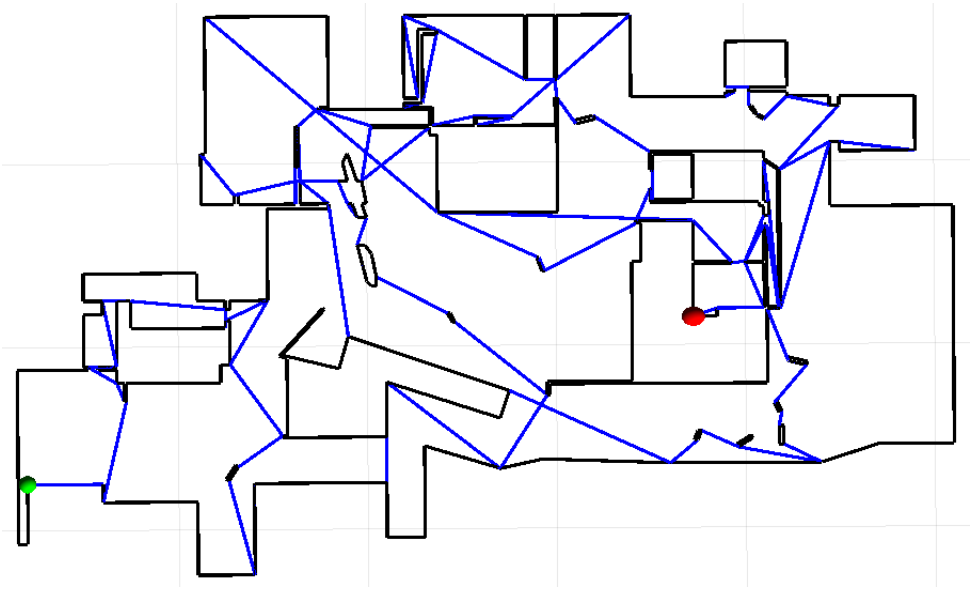

Figure 4.3 An example of a tour

Nearest

In nearest tours, we always try to get to the camera that is nearest to the current point. We do this by simply checking our precomputed Dijkstra distance values for cameras, and choosing the minimum one each time. The aim is to visit all the cameras in one region first so that the NPC does not have to return to it creating a tour whose overall length is short. The potential disadvantage is that in terms of coverage we may only have incremental increase since the cameras are covering the same general region.

Farthest

The farthest tour is the opposite of the nearest tour. Here, we always attempt to reach the camera that is farthest from our current location. Therefore, we always pick the maximum of the Dijkstra values. The idea here is that the farthest camera likely covers a region that shares the least overlap with the current camera, and, therefore, will yield the largest increase in coverage.

Nearest Non-Visible

The nearest non-visible tour works similarly to the nearest tour. But instead of considering all cameras, we consider only the ones that are not directly visible to our current camera. We are essentially trying to do what the farthest tour does i.e. attempting to get a larger increase in coverage by going to a region that has less overlap with the current one. At the same time we are trying to make sure that the increase in tour length is not too great by visiting a camera that is nearby.

Maximum Union

In the maximum union tour we start measuring coverage instead of making guesses. This increases computational cost because even though the individual visibility polygons have been preprocessed, their mergers still need to be calculated on the go as these can occur in many possible combinations. In maximum union, we first consider the area of the map that has been seen up to this point. This is represented by the merger of the visibility polygons from not just the cameras visited so far, but also the non-camera nodes of the roadmap that have been visited. Then, each unvisited camera's visibility polygon is merged with the polygon representing the currently uncovered portions of the map, and the increase in coverage is noted. The camera that provides the maximum increase is chosen.

Maximum Union:Distance

The maximum union:distance tour accounts for both coverage and distance. We now take the increase in coverage that a camera would provide and divide it by the increase in tour

distance that would be incurred when travelling to that particular camera. This result is called the union:distance ratio. Each time, we choose the camera that gives the maximum union:distance ratio. This makes sense since we want to maximize coverage and minimize distance travelled.

Chapter 5
Experiments and Results

We now have the tools to create different types of exploratory tours for an NPC, all of which will guarantee that the NPC ends up seeing the entire game map. We also have methods and metrics to compare these tours. To make such comparisons we ran various experiments on an array of real game maps. This chapter will detail the methodology used for those experiments and the results they produced, and in the process provide some insights into exploration behaviour for NPCs in video games.

Section 5.1 describes the methodology used to carry out the experiments. There we will talk about the features that were focused on during testing, the different game maps on which tests were run, and the test environment used. *Section 5.2* presents the results of the experiments. This is organized according to the features we have tested on.

5.1 Experiment Methodology

The general objective of each experiment is to make observations that will help us better understand issues related to an NPCs exploration of a game map. More specifically, we are trying to test the effectiveness of different exploration tours, by modifying different factors that we hypothesize have influence on said effectiveness. These factors, and what we imply by "effective", are discussed in *Section 5.1.1*. Of course, all our tests must be run on a map. To increase the applicability of the results we have chosen a series of maps from commercially available video games played by millions of gamers. The maps and the

Table 5.1 Features and Metrics

Roadmaps	Tour Types	Starting Positions	Granularity	Target	Metrics
Shortest-path	Nearest	Arbitrary	Regular	Camera	Distance
Triangulation	Farthest	Extreme Position	Double	All	Coverage
	Nearest-non visible	Centre	Triple		
	Max Union				
	Max Union: Distance				

reasons for choosing them are outlined in *5.1.2*. Our programs were written in different languages and platforms, and made use of some third-party libraries. These and other practical issues related to implementing the experiments are described in *5.1.3*.

5.1.1 Features and Metrics

Table 5.1 lists 5 different feature categories that could affect a tour in a map, along with the metrics that we use to indicate the quality of a tour. Below, the latter is described first, followed by a breakdown of each of the categories and what effect we think they may have on the metrics. It should be noted that the categories in the table are independent, and can be set in many possible combinations over many maps. Our challenge will be to isolate the effects of each group.

Metrics

What constitutes an effective tour can be difficult to define. Since the whole purpose of making an NPC display exploration behaviour is to make AI seem more intelligent, it would make sense that an effective tour is one that is humanlike. Again, humanlike is difficult to define, and trying to comprehensively define it would be beyond the scope of this thesis. Instead, we posit that our tours are already humanlike to a degree as they guarantee that an NPC visually discovers the entire game map by walking around (i.e. exploring).

Building on this assumption, a tour that explores efficiently (covers the most ground in the least time and, hence, distance) would appear as effective, or at the very least, provide an

important baseline against which all other tours can be compared. Efficiency is determined by two metrics - distance and coverage. Distance is the total length of the tour and is our primary metric. Distance depends on coverage because the end of the tour occurs when the full map has been seen, regardless of whether all the cameras have been visited. We represent coverage as the percentage of the map area that has been seen. In a qualitative sense, we are also interested in the rate at which coverage increases throughout the tour.

Roadmaps

As mentioned in *Section 4.2*, a roadmap heavily influences a possible tour's geometric nature by defining the set of possible movements from a point A to a point B. A tour on a shortest-path roadmap will always be shorter than one on a triangulation roadmap given all other features are kept the same. This is because of how edges are constructed in each. Shortest-path roadmaps cut across large spaces, and go along the sides of obstacles, if necessary, to reach points in a minimal distance. Edges in a triangulation roadmap create a zig-zag shape, as they move from one triangle to another, increasing distance overhead. However, edges do not graze the sides of obstacles, making any tour constructed on a triangulation roadmap appear arguably more humanlike. A different type of roadmap known as widest-clearance [26] focuses on keeping the most distance between the path and any obstacle edge, making it seem even more humanlike if we put a premium on obstacle avoidance. However, it is not guaranteed that gamers will exhibit any one behaviour (obstacle avoidance or choosing the shortest route) the most. Such issues are difficult to quantify. In our experiments we focus less on whether one roadmap is better than another, and more on determining if the effect of other features carry over between roadmaps.

Tour Types

This is the feature category of most interest. We will be measuring the effectiveness of each tour type described in *Section 4.2.2*, and figuring out how they are affected by other categories.

Starting Positions

Starting position refers to the initial location of the NPC in the map or the first point on the tour. We always start from some node that is on the roadmap, and this node is always a camera. If the node is at one of the corners (i.e. extreme ends/positions) of the map, maybe it will take it more time to go figure out the rest of the space, or the exploration will be more gradual. Perhaps, if it is in the centre of the map, the NPC will have more options of immediate destinations, and therefore will discover the map more quickly. We compare these two types of positions with arbitrarily chosen nodes to check whether starting positions have significant impact on tour length.

Because a game map is rarely a regular geometric shape, points from a central region or extreme region are hard to define and compute. However, in order for the choice to have some uniformity across maps, a defined mathematical method must be used instead of a manual one based on visual judgement. The procedure we have devised to compute what is our definition of cameras at extreme and central positions are described below.

Extreme Camera - We first sort the cameras according to their x coordinates, and after that their y coordinates if the x coordinates match. We then choose our four extreme cameras by looking at the first 5 cameras and the last 5 cameras in the list. The first set has low x values and the second set has high x values. From each of these sets we choose the two cameras with the lowest y value and the highest y value, respectively. These four are our extreme cameras.

Centre Camera - For the centre camera we find the average of the highest and lowest x value, then find the camera that has an x value closest to this average. We make a set consisting of this camera, and the two cameras appearing before and after it in the sorted list. Following this the average of the highest and lowest y value is calculated. We then choose the camera with a y value closest to this average (from the set we created), to get the centre camera.

Target

At each decision point in the tour we try to find a target camera (one that is unvisited) to go to next. However, there are nodes on the roadmap that are not cameras. It is not necessary that the result of the greedy decision we are trying to make (e.g. nearest point, farthest point, point providing most increase in coverage) is a camera point. If we considered all the unvisited nodes of the roadmap it is possible for the result to be a non-camera node. But whether this will have a significant impact on tour length is difficult to predict. Therefore, in our experiments we will try both approaches, one where target nodes are exclusively camera points, and one where all types of nodes are considered.

Granularity

The roadmap is a series of edges. Currently we are only checking visibility at the endpoints of these edges as a way of approximating everything that the NPC sees as it walks between two endpoints. As mentioned, this is because edge visibility algorithms are costly to run and difficult to implement. A possible way to improve this approximation is to check visibility at additional points between the endpoints. We have to choose these points in an uniform manner so that the effect of increased granularity is uniform across the roadmap, and therefore the tour. To do this, we start at either endpoint, and add points that are at fixed intervals from it, until we reach or overshoot the other endpoint. To compute this interval, which we call *unitdistance*, we first compute the average distance of all the edges of the roadmap. Then, the unit distance is simply the average distance divided by a number of our choosing. We increase granularity in degrees. For double granularity, the average distance would be divided by 2. For triple, by 3. The increase in granularity will increase the number of points considered on a roadmap edge. Regular granularity is simply our default mode when only the endpoints of edges are considered. Our intuition is that considering more points between two nodes will increase the new area being covered while travelling between two roadmap nodes.

5.1.2 Maps

Game maps are the primary test subjects on which we can execute our ideas, and ultimately where real world applications of the work in this thesis would be presented. The nuances and issues related to maps were discussed in *Section 2.2*. In *2.2.3* we specifically described how real game maps were obtained and converted to work in the context of our experiments. Below we present 6 such maps and some of the reasons behind their usage.

Map 1 - Pillars of Eternity: Raedric's Hold Dungeons

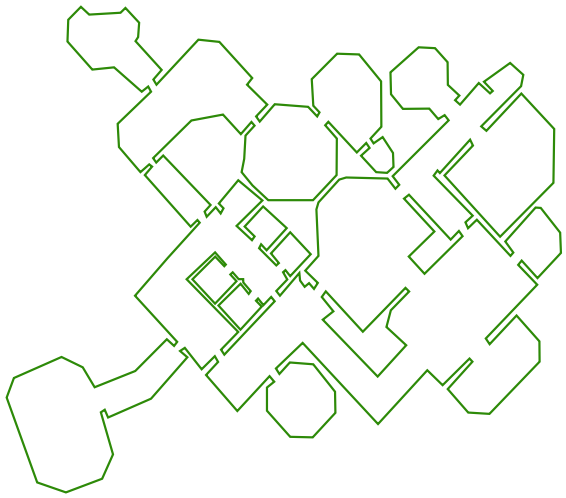

Figure 5.1 Pillars of Eternity: Raedric's Hold Duneons

Pillars of Eternity is a fantasy based role-playing game (RPG) where the player controls a group of characters on quests through the game world. The map is slowly unveiled to the player as the characters move to new areas, an effect that is also known as *fog-of-war*. In fact, players gain points for discovering new regions rather than for killing enemies. This and the availability of many sprawling maps makes this an exploration oriented game. There is a focus on making NPC interaction varied and dependent upon a players previous actions in the game. For these reasons, maps from Pillars of Eternity serve as a good testing ground for analysing NPC exploration behaviour. Our first map (*Fig 5.1*) is called *Raedric's Hold Dungeons*, and it represents a typical indoor environment in an RPG. There

are many individuals rooms and hallways with multiple ways of reaching the same space giving characters many options for charting out an exploration path.

Map 2 - Pillars of Eternity: Copperlane Catacombs

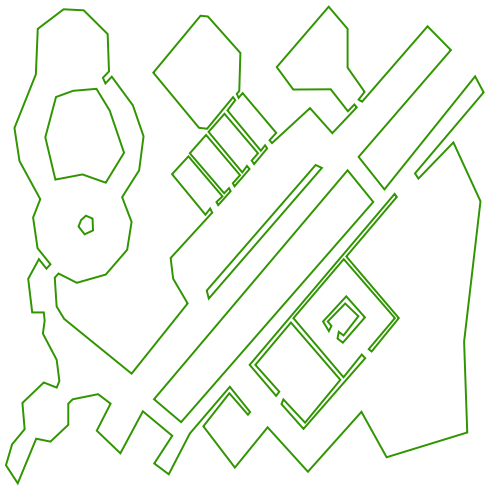

Figure 5.2 Pillars of Eternity: Copperlane Catacombs

Copperlane Catacombs (*Fig 5.2*) is another indoor map from Pillars of Eternity. In addition to having a labyrinth-like structure, this map has some long stretches of space separating different areas of interest in the map.

Map 3 - Witcher 3: Palace of Vizima

Witcher 3 is a role-playing game that is somewhat more action oriented than Pillars of Eternity. The massive size of the game world makes it quite exploration friendly. We chose the map, *Royal Palace in Vizima* (*Fig 5.3*), from Witcher 3 to bring variety to the RPG games chosen for the experiment.

Map 4 - Pillars of Eternity: Doemenel Manor

In *Section 2.2.3* we mentioned that maps with multiple floors could be reproduced in our model through a bit of compromise. To demonstrate this point we chose *Doemenel Manor* (*Fig 5.4*), a map from Pillars of Eternity with two separate floors. The two floors were

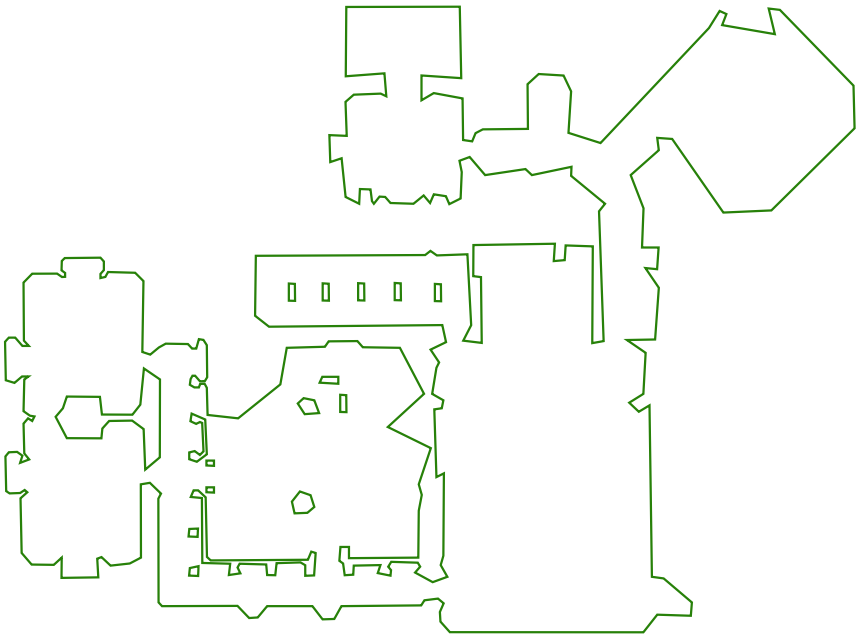

Figure 5.3 Witcher 3: Royal Palace in Vizima

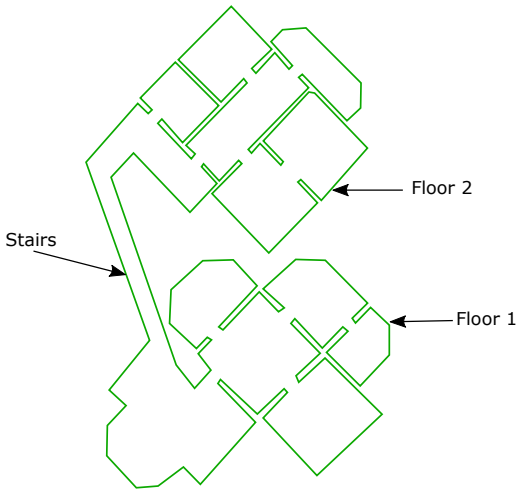

Figure 5.4 Pillars of Eternity: Doemenel Manor

connected at the point where the stairs are, with a straight passage that tries to emulate the area covered when climbing the stairs. The stairs can be created in several ways. In ours a certain part of the other floor is exposed when standing near the stairs. This might not be true in the actual game, and is an issue that can be better handled by the level designer.

Map 5 - Call of Duty: Crash

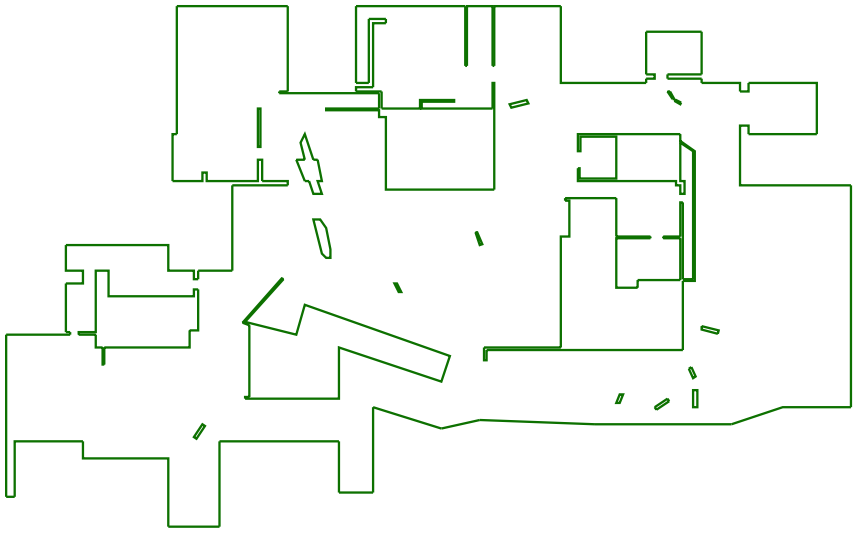

Figure 5.5 Call of Duty 4: Crash

Call of Duty is series of first-person shooting games. Such games are usually very mission oriented and the focus is on killing enemies and moving quickly through a map rather than exploration. However, it makes sense for hostile NPCs to be moving about the map in their own way and attacking players when there is a chance confrontation. The map, *Crash* (*Fig 5.5*), from Call of Duty 4 was chosen because of its large size and how it introduces complexity through many scattered obstacles.

Map 6 - Arkham Asylum: Arkham Mansion

Our last map is a game from the action-adventure genre. *Arkham Mansion* (*Fig 5.6*) is a map from Batman:Arkham Asylum. Players have to complete missions through combat and stealth. The significant use of the latter in the game, and the complexity of the map itself, makes it an interesting choice.

5.1.3 Test Environment

During development and testing, the ideas and algorithms presented in this thesis were implemented entirely in Unity, a popular game engine and integrated development environ-

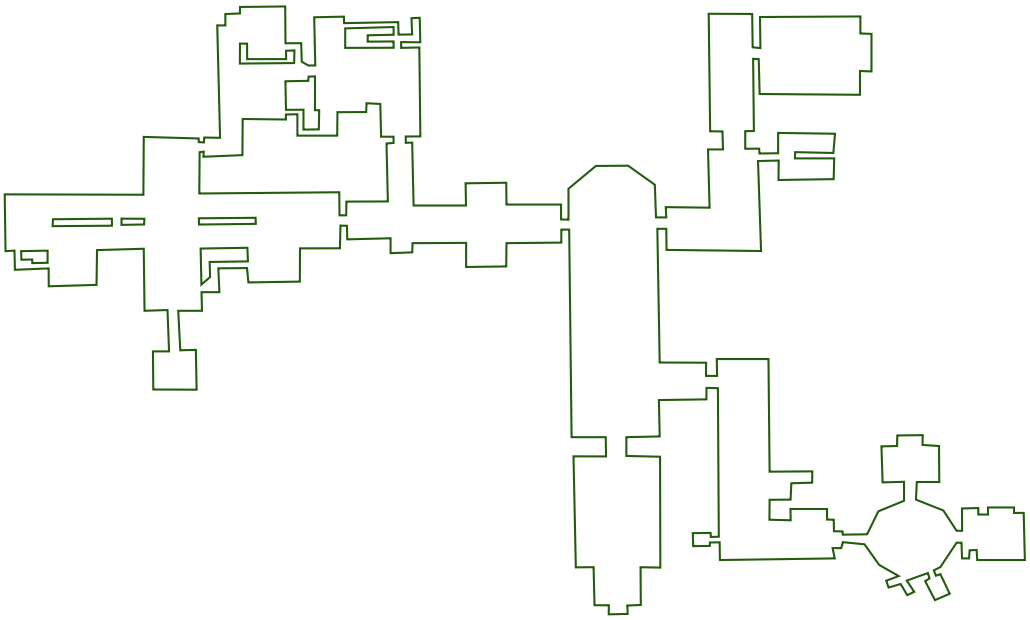

Figure 5.6 Arkham Asylum: Arkham Mansion

ment used to build video games. We wanted to exhibit our tools inside an unified framework that is already optimized for game development and level design, thereby, strengthening the applicability of our work for developers. Unity is also great for visualizing work related to computational geometry, a feature that was immensely helpful when we drew our own test maps and were debugging issues related to numerical stability. However, Unity also adds a good amount of computational overhead, due to many additional environment elements that it manages and that are unrelated to our experiments.

When switching our tests to real game maps, the issue of computation time became more apparent, since real maps were larger than our own and required a lot more processing. Therefore, to ensure a timely execution of the experiment phase of this thesis we used a combination of our fully developed environment and a few third-party libraries dedicated to solving specific problems. This spread the functionality of our tool across different platforms. As Unity matures these functionalities maybe brought back in, or bindings can be developed. The libraries and platforms we have utilized are listed below along with a brief mention of the hardware the programs were run on.

Libraries

The individual computational geometry problems we have had to deal with, such as visibility polygons, are simply a means to solve the larger problem of exploration behaviour. Although we have put considerable time and effort into developing quality solutions, third-party libraries that are entirely dedicated to solving a particular computational geometry problem will naturally have more optimized approaches.

Triangle - Triangle [33] is a triangulation and mesh generation library developed by Jonathan Shewchuk. It was written in C++, and has won the Wilkinson Prize for Numerical Software that is given out every four years.

VisiLibity - VisiLibity [30], is a fast and accurate C++ library for computing visibility polygons. It was developed by K. J. Obermeyer and other contributors.

Shapely - Shapely is a package written in Python that is helpful in manipulating and analysing geometric objects [19]. We have used it to merge visibility polygons. Functions for doing so in Shapely are more optimized than our own. It has been very useful in constructing tours that use coverage as a heuristic, since these require many quick polygon merging operations. Shapely was developed by Sean Gillies and other contributors.

Platforms

The full platform architecture is shown in *Fig 5.7*. Our map parser works in Python. The converted map, which is in a CSV format, is then read into Unity using C#, which is the language we have used to write all our logic in that environment. In Unity, we create a spanning tree and output this to a file. This and the map is then read in C++, where we use *Triangle* to triangulate the map. Once the triangulation is computed, we switch back to Unity and C#. Here the colouring, camera computation, construction of the two roadmaps, and the Dijkstra shortest-path precomputation follow. Next, we switch to C++ and use *VisiLibity* to precompute the visibility polygons for every roadmap node. For the final phase, all relevant information is read into Python, where we have written the logic for our tours. Here we make use of *Shapely* for all polygon mergers and area calculations. All results are stored in CSV files. We wrote scripts in R, a programming language that

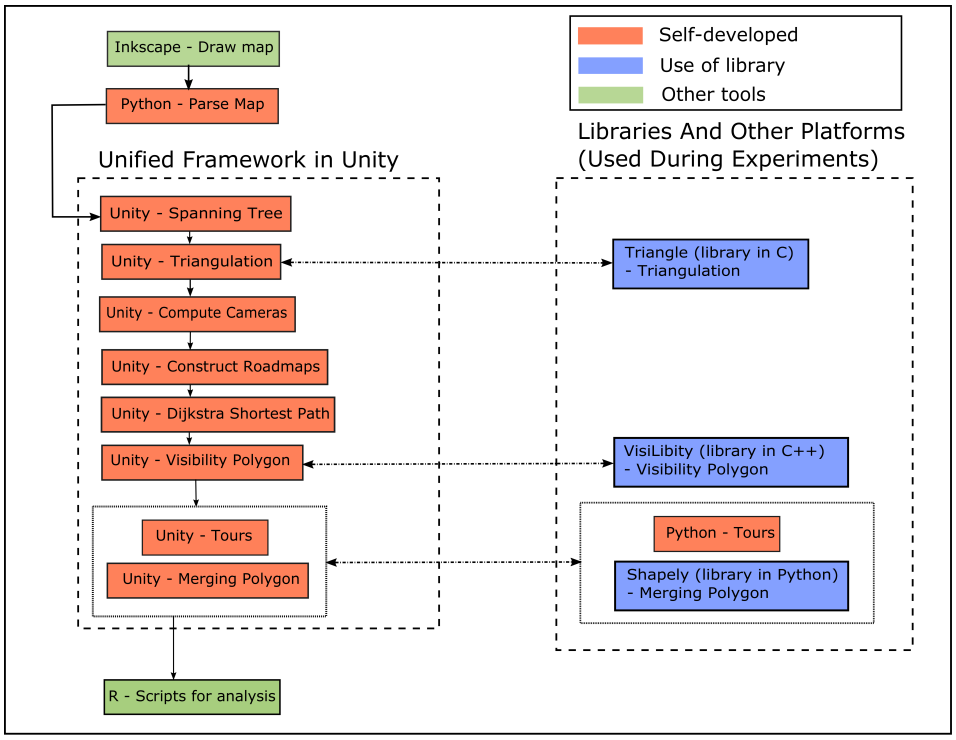

Figure 5.7 Platform architecture

specializes in statistical computing, to analyse our findings.

Hardware

All work in Unity was run on a machine with a dual core Intel Core i3 CPU (@2.27 GHz), 6 GB of RAM, and a Windows 7 operating system. The rest was run on an Amazon EC2 machine with a single core Intel Xeon CPU (@2.50 GHz), 2 GB of RAM, and an Ubuntu 14.04 LTS operating system.

5.2 Results

This section presents the findings of the experiments described previously. The number of maps, and the number of possible combinations in which the features in *Table 5.1* can be set, means that the total number of possible experiments is large, and hence the data that can

Table 5.2 Tour Abbreviations

Abbreviation	Tour Type
N	Nearest
F	Farthest
NNV	Nearest Non-Visible
MU	Maximum Union
MUD	Maximum Union:Distance

be produced and presented is large as well. We have tried to focus on the most interesting avenues of inspection, and aggregated the data where needed and where possible, in order to keep the presentation of results concise and relevant. We begin with a comparison of the different tour types, then move on to the effects of individual features (granularity, target and starting position), and, lastly, experiments with triangulation roadmaps. All tests are run under default settings unless otherwise mentioned. The default settings involve a shortest path roadmap, regular granularity, and using cameras as targets. For the sake of legibility in graphs, and conciseness in discussions, the tours will each be referred to in their abbreviated format as shown in *Table 5.2*.

5.2.1 Tour Comparison

This section will compare our five tour types under normal conditions. The primary metric in concern here is the length of a tour or the total distance travelled. A lower overall distance is considered better. We begin by analysing a single N tour, then compare instances of each tour type, before moving on to an aggregated analysis.

Analysing a Single Tour

Let us first consider the basic information related to any tour undertaken by an NPC in the game map. In our algorithms, a tour is represented as a sequence of points $(p_1, p_2, .., p_{n-1}, p_n)$. These points are all nodes of the roadmap being used. At the very first point, p_1, of the tour, the distance travelled by the NPC is zero. However, the NPC can already see at least some portion of the map, and therefore its coverage is the area of the visibility polygon generated from p_1. As the NPC moves to the next point, p_2, in the sequence, the distance travelled

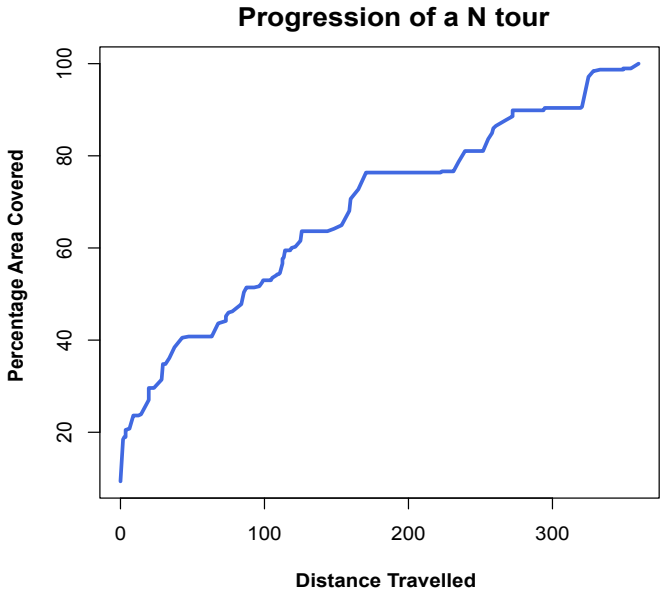

Figure 5.8 A nearest tour starting from a randomly chosen camera on *Map 1*

increases. If any new area of the map is uncovered while moving to p_2, then the coverage increases as well. This process continues until, p_n, when the coverage is at its maximum possible value.

Fig 5.8 shows a Nearest (N) tour that starts from a randomly chosen camera in *Map 1* (Dungeons). The graph is a plot of the distance travelled versus the area covered by an NPC as the tour progresses. The distance is in Cartesian units. Coverage is represented as a percentage of the map's full area. A steep rise in the curve indicates a relatively large increase in coverage, and a flat region indicates that no new area was uncovered during that part of the tour.

Instances of Different Tour Types

Fig 5.9 shows a plot of all five different type of tours when run from the same starting camera on *Map 2*. All the tours are shown in full except for the Farthest (F) tour, which takes considerably longer to finish. This is done to make the graph more legible. The final length of the F tour is labelled on the curve. Initial observations show that the Maximum Union:Distance (MUD) tour is the shortest. The N tour works comparatively well coming

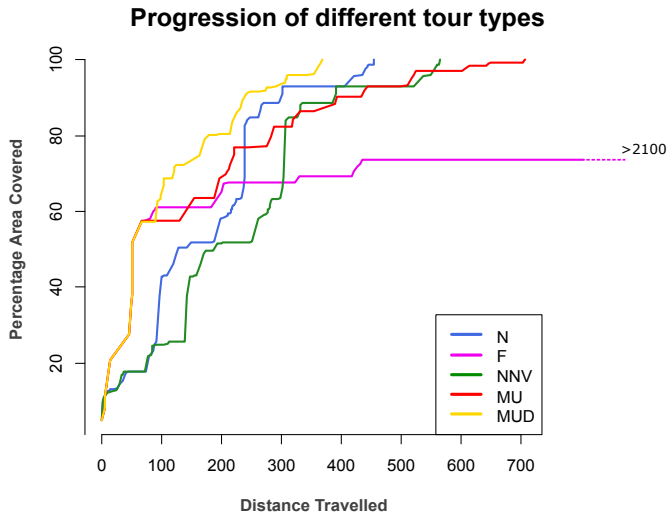

Figure 5.9 Each tour type starting from the same fixed camera on *Map 2*

in as the second most efficient. As noted, the F tour is the worst. Nearest Non Visible (NNV) is better than Max Union (MU), which might imply that when used on its own, distance is a better heuristic than coverage. However, if all the tours are run from a different

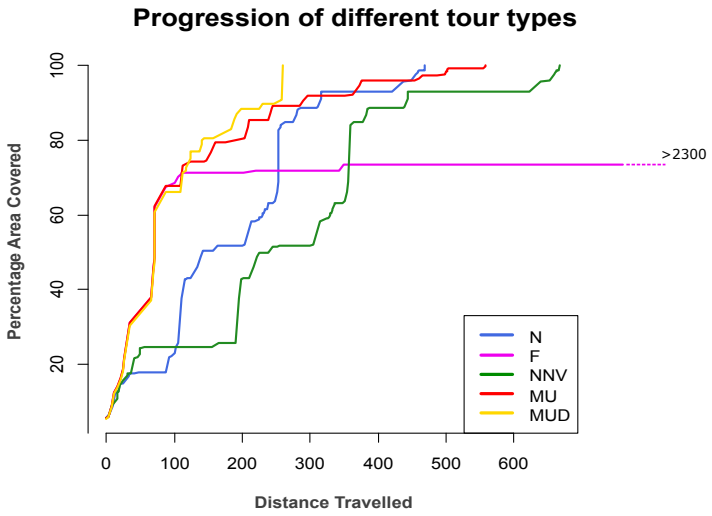

Figure 5.10 Comparison of tours starting from a different fixed camera on *Map 2* shows performance ranks of tour types can vary (MU outperforms NNV).

camera on the same map this observation changes. The plot in *Fig 5.10* shows a case where a MU tour is shorter than a NN tour. Where a tour starts has an effect on its overall length.

Therefore, it is difficult to comment on the comparative efficiency of different types of tours by looking at single instances.

Tour Comparisons Over All Cameras

Since starting points affect performance, we can run each type of tour from every camera in the map, and use the data to plot a probability distribution of total tour lengths. Consider a NNV tour. A probability distribution curve, for the class of NNV tours, will show the range of possible lengths that a NNV tour can have, along with the likelihood of particular values. We do this via a violin plot. *Fig 5.11* (Map 2) and *Fig 5.12* (Map 1) each show violin plots of the data gathered by running different tour types on every camera in a map. The F tour is not shown as its full set of values lie far outside the range of the other tours, visually flattening the other tours and making the chart less comprehensible. This happens in the case of all the maps we have tested. Besides the probability distribution, a violin plot also shows the median (the white circle) and the interquartile range (the black box) for a set of values. The interquartile range represents the range of the values that lie between the first quarter and the last quarter of the sorted data set. Please note that the scales for the different plots we present vary.

General Trend - As seen in *Fig 5.11*, which shows the violin plot for *Map 2*, the MUD tour performs better than other tours for all starting cameras. N is always better than MU, and both overlap with NNV. NNV seems to have the widest variety in tour lengths, a fact that maybe helpful if a developer wants different NPCs to perform at different kinds of efficiency over a wide range. The relationships between the tour types are similar in *Map 4* (Manor), *Map 5* (Crash), and *Map 6* (Mansion).

Other Cases - The violin plots for *Map 1* (*Fig 5.12*) and *Map 3* showed slightly different behaviours. MUD is still the most efficient in both. For *Map 1*, MU appears to be the second best overall, although N is not too different from it. In *Map 3*, both NNV and MU sometimes perform better than N, in some cases working as well as MUD. But their medians are above N's median and interquartile range, implying they perform worse overall.

Inferences

MUD appears to be the most efficient tour type in all maps. This suggests that distance and coverage work better as heuristics when used in combination rather than individually. This makes sense as we are trying to optimize both metrics, and an approach that quantitatively assesses both in its decision process is likely to work best.

In trying to get to the camera with the best coverage increase, MU likely goes to points that are far away, ignoring nearby cameras that were crucial to achieving complete coverage in the local area. This means it will have to come back to the same area later, thereby increasing redundancy. N is likely to ensure the local area is covered, since it is trying to move to the nearest points, but will have some redundancy since it will keep doing this even when the local area has been covered. But this causes less overhead than MU's back and forth across the map. NNV's approach ignores cameras that would complete the coverage of the local area simply because they are visible from the current location, making it incur overhead that is more similar to MU. This makes NNV more likely to be between N and MU in performance rank. Lastly, F suffers from the problems of MU as it treks back and forth across the map without making any of the guaranteed coverage gains, resulting in the worst performing tour.

5.2.2 Target

When analysing the target feature, we considered each map and compared an individual tour's performance under the two settings ("Camera" and "All"). Once again, this is in the form of violin plots showing the distance distribution of the tour run over all starting points (cameras). *Fig 5.13* shows the results for *Map 5*.

The results for 5 of the 6 maps displayed common patterns. When using cameras as the target, N and NNV perform better, sometimes only slightly so. MU and MUD generally worked better when all nodes were targeted, otherwise they performed similarly under both settings. In the map where results were slightly anomalous, the four tours were either similar or worked better with the "All" setting. In every map, the F tour always did significantly worse in the "All" setting.

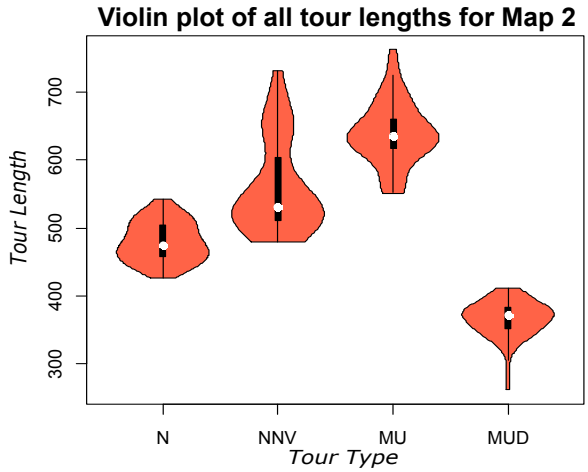

Figure 5.11 *Map 2*'s plot is representative of the general trend for the relative performance ranks of different tour types

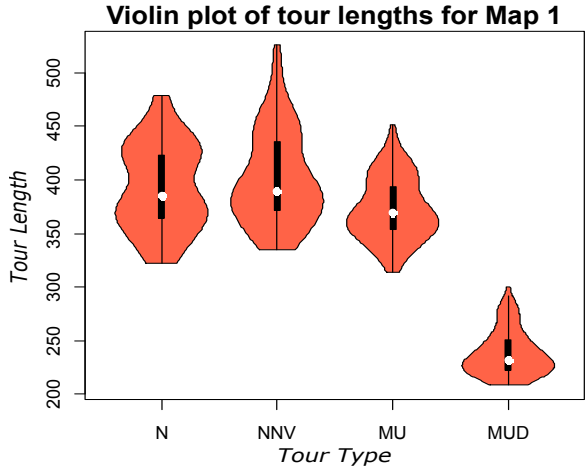

Figure 5.12 *Map 1*'s plot shows aggregated performance ranks can differ from the general trend

Inferences

Distance appears to be a better heuristic when cameras are used as the sole target, whereas coverage is better when all nodes can be tested. Since NNV performs similar to N in these experiments, NNV tours likely rely more on distance than coverage. This makes sense as its strategy of going to a non-visible node is a guess about coverage and not a quantified approximation. Lastly, MUD, the most efficient tour we have, is observably improved when all nodes are considered. Unlike points of good proximity, points of improved coverage are

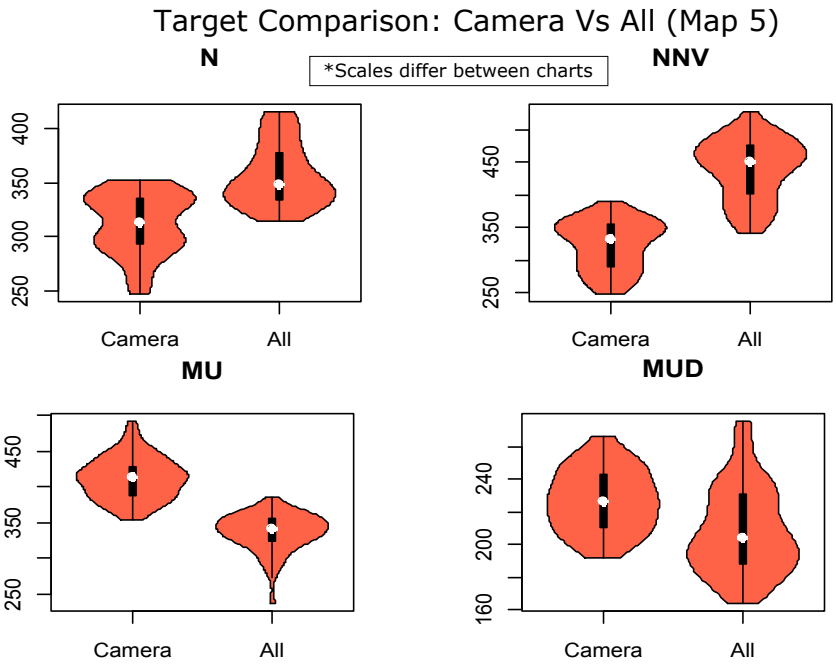

Figure 5.13 Effects of the target feature on different tour types

not spread about symmetrically. Hence, more choices increases the likelihood of a better decision.

5.2.3 Starting Positions

For each map we ran two instances of every tour type, the first on a centrally located camera and the other on a camera located at one of the corners of the map. The length of these tours were then marked on the corresponding violin plots showing tours over all cameras. The results for four of the maps are shown in *Fig 5.14*. A camera at the centre is marked with a green line, and one at an extreme position by a blue line. Random cameras were not picked as the most likely value for those are signified by the points where a violin plot is widest.

Inferences

The markers for the two types of starting position do not seem to show any fixed pattern. Neither one is explicitly better than another overall, for any particular tour type, or a par-

ticular map. For *Map 4* and *Map 6* (except its MUD) the centre camera seems to hover around the widest region of a violin plot. In those maps and in *Map 3*, this observation is true for the MU tour. What that would suggest is that a tour run on a centre camera can be indicative of the most probable length of any tour run on the map. All things considered, a starting position's effect on tour length does not appear to be predictable, and is likely affected primarily by specific geometric properties of individual maps.

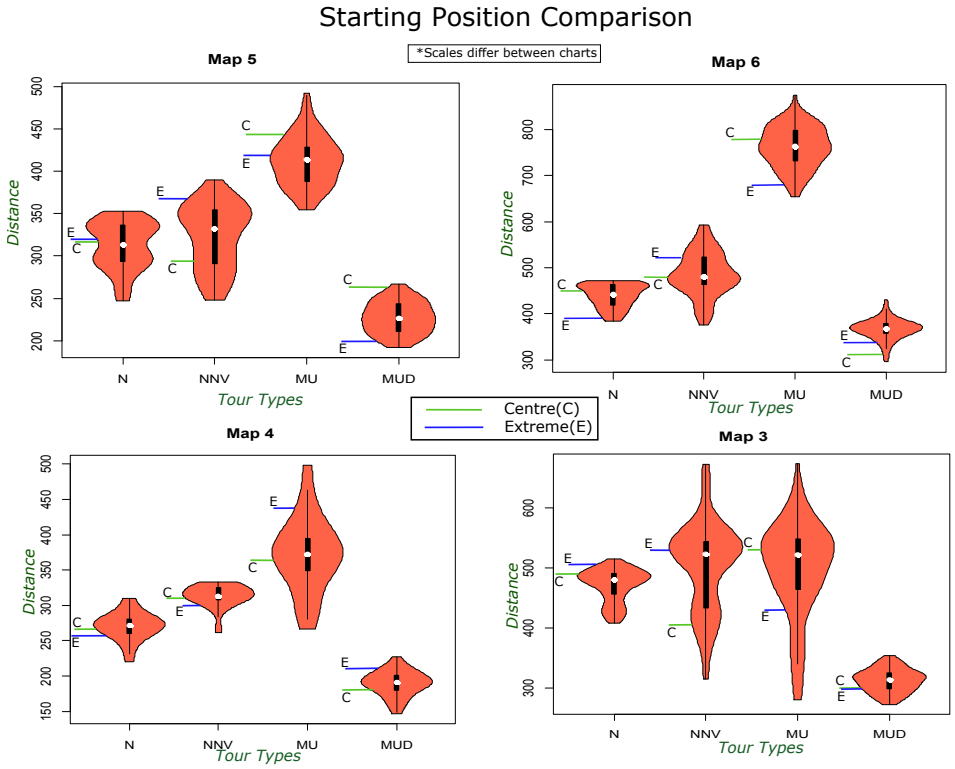

Figure 5.14 Locations of results from different starting positions on the violin plot

5.2.4 Granularity

In most cases there are improvements in tour distance when checking visibility polygons for points at a higher granularity or frequency. The relative ranks of the tours remain the same i.e. F remains the worst, and MUD remains the best. MU, however, tends to cause enough of an improvement to beat N and NNV. Interestingly, F has the largest percentage decrease in tour length, sometimes coming close to the performance of other tours but never

surpassing them. N has the least percentage decrease. Sometimes improvement in the tours will happen only on triple granularity, and sometimes double and triple will have similar performance compared to regular granularity. *Fig 5.15* shows the violin plots of each tour type for *Map 3* at three different granularity levels. Number of points being considered in the roadmap increased significantly. For example, in *Map 3* the number of unique points whose visibility polygon was precomputed was 252, 3457, and 5749, for regular, double, and triple granularity, respectively. The results for all the maps, when higher granularity is used, are summarized below.

N - Markedly better on 3 maps, marginally better on the other 3 maps

F - Markedly better on all maps

NNV - Markedly better on 5 maps, similar on 1

MU - Markedly better on all maps

MUD - Markedly better on 5 maps, slightly better on 1

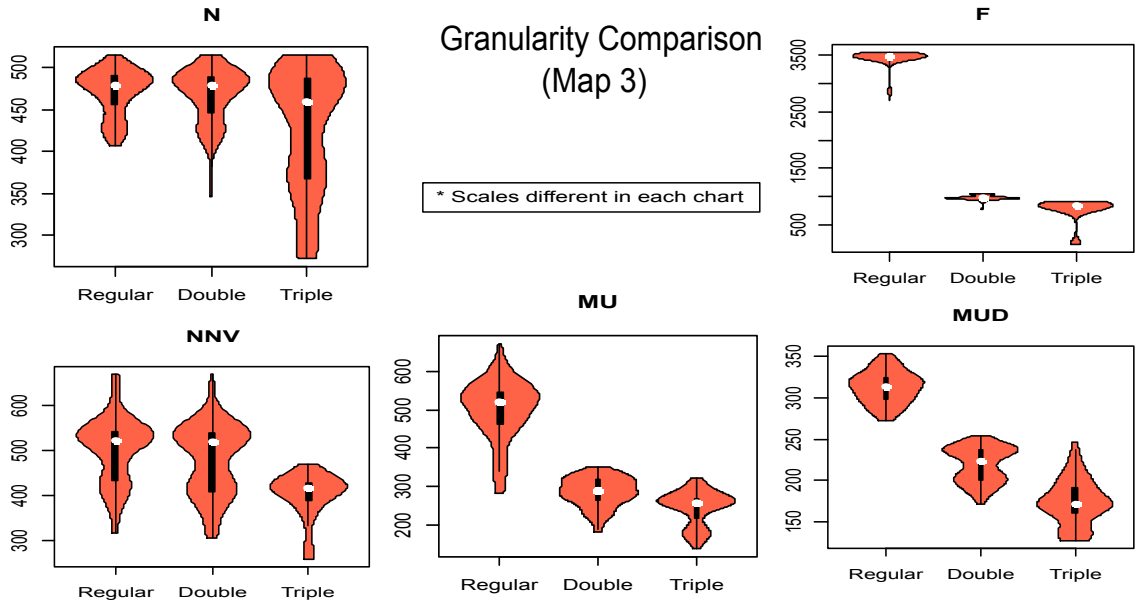

Figure 5.15 Effects of different levels of granularity on tour types

Inferences

As mentioned, F improves the most. This improvement is probably due to paths between cameras in F being longer overall. A longer path could be because of 1) a higher number of total edges, or 2) longer edges. If each edge has some incremental improvement then a higher number of edges would imply a higher aggregated improvement. Edges that are longer cut across larger regions, and therefore are likely to be surrounded by more complex occlusions. This means that the endpoints see less of the area that is visible to points along the edge.

This line of reasoning would explain why N has the least improvement. The heuristic that N uses is in a way the opposite of F, as paths between cameras in an N tour will be of the shortest possible length. Therefore, those paths will have the least number of edges or shorter edges, reducing the chances of overall improvement. Once again, note that N still has a better performance than F. It is simply less affected by granularity.

To be clear, paths between cameras in MUD are never shorter than N. The reason MUD has a shorter tour is because it ends up visiting a lower number of cameras. Therefore, its improvement with higher granularity does not contradict our reasoning.

5.2.5 Triangulation Roadmap

For 3 of the maps (*Map 2*, *Map 4*, *Map 5*) we repeated all the experiments stated above (except granularity) replacing the shortest path roadmap with a triangulation roadmap. We did not do this for all 6 maps as triangulation roadmaps are comparatively larger, and, therefore, experiments take noticeably longer. The results for overall tour comparisons were similar. MUD had the best performance followed usually by N (2 out of 3). As before, N, NNV, and MU overlap, and sometimes swap performance ranks. When all nodes are targeted the computation slows down significantly as triangulation roadmaps have a lot more nodes. For example, on *Map 5* a shortest path roadmap has 194 nodes whereas a triangulation roadmap has 534 nodes. As for the results, N and NNV are clearly worse in the "All" setting, whereas MUD improves slightly or is similar. MU hovers around similar areas in both settings. So overall the observations are similar to those in shortest path roadmaps.

There was only one perceivable trend in experiments with starting positions. The centre camera tends to produce a tour whose length is near the median value of the distribution i.e. representative of a tour type's most likely value in a map. Earlier, this had occurred in some cases on a shortest path roadmap but was not strong enough to be considered a pattern. We conducted the granularity experiments on *Map 2* and *Map 4*. Results improved slightly on the latter and were similar on the former. This could be due to the small size of the map. Improvements were less remarkable than shortest path roadmap due to the size of edges in triangulation roadmap being smaller on average.

Chapter 6
Related Work

This chapter briefly discusses some of the existing research related, both directly and indirectly, to this thesis. Work in video games often ties into the field of computational geometry, and this is certainly true in our case. We begin with a history of algorithms for computing visibility polygons in *Section 6.1*, since measuring visual coverage is a non-trivial task upon which this study heavily relies on. *Section 6.2* talks about two well known problems in the world of computational geometry, both of them dealing with guaranteeing complete coverage of a space. In *Section 6.3*, we take a look at work related to the concept of exploration, primarily in the field of robotics.

6.1 Visibility Polygon

The notion of point visibility in its current form was first introduced in 1979 by Davis and Benedikt [13]. They referred to the set of all points visible from a point inside a polygon as an "isovist." Soon after, Gindy and Avis presented an optimal $O(n)$ algorithm for computing point visibility in a simple polygon [16], where n is the number of vertices of the polygon. Lee [27] reworked that algorithm to make it conceptually simpler. Joe and Simpson [23] showed that both of these failed for polygons that winded sufficiently and proposed corrections. These algorithms do not involve ray casting, and instead use a stack, and do not work for polygons with holes.

Suri et al. [36] and Asano [2] independently developed an optimal $O(n \log n)$ algorithms for point visibility in a polygon with holes. The lower bound of $\Omega(n \log n)$ was proved by reducing the problem to that of sorting n integers. Heffernan and Mitchell [22] improved upon this with an $O(n + h \log h)$ approach, where h is the number of holes in the polygon. More recently, there has been interest in methods that involve a preprocessing stage. For polygons with holes, Pocchiola and Vegter [32] introduced the notion of visibility complex, k. k is a measure of the geometric complexity of the region visible to a query point q. The preprocessing in their method takes $O(n^2 \log n)$ time and the query time is $O(k \log(n/k))$. Zarei et al. [44] increased the time of the preprocessing to $O(n^3 \log n)$ in order to reduce the query time for each point to $O(1 + min(h,k) \log n + k)$. An authoritative account of visibility polygon research can be found in a book by Ghosh [18].

In 2014, Bungui et al. [5] stated that there are no available complete, correct and efficient implementations of the above algorithms. They then provided results of their own implementations, stating these would be available through the C++ graphics library CGAL, which as of this writing is still pending. Algorithms for computing edge visibility are even more complex, and their worst case lower bound is $\Omega(n^4)$ [36]. In our case, we have approximated the visibility of individual edges, and overall paths, by using point visibility at multiple points.

6.2 Guaranteeing Coverage

This section expands upon a topic that is closest to our overall work, the Watchman Route Problem, and its precursor, the Art Gallery Problem.

6.2.1 Art Gallery Problem

The Art Gallery Problem (AGP) requires finding the minimum number of stationary guards or cameras needed to make sure the interior of an n-walled art gallery is fully covered [31]. The problem was first posed by Klee to Chvátal in 1973. Chvátal [9] produced a proof that stated $\lfloor n/3 \rfloor$ guards are sufficient and sometimes required to fully cover a simple polygon of n vertices. This is known as the "Art Gallery Theorem." The proof was later simplified by

Fisk [17] using triangulation. Avis and Touissant [3] proposed an $O(n \log n)$ algorithm for determining the guards via triangulation and 3-coloring. In *Section 3.1* we took advantage of these ideas to compute our camera points. More details about the history of AGP can be found in Rourke's book [31].

6.2.2 Watchman Route Problem

There are many variations of AGP, and the one most relevant to us is the Watchman Route Problem (WRP). The WRP uses a single moving guard instead of multiple stationary guards, and asks us to device a path that ensures the guard sees all parts of the gallery. It is a combination of AGP and the travelling salesman problem. In formal terms, WRP requires computing the shortest route in a polygon, such that every point inside the polygon is visible to some point on the route. It should be apparent that this is the same problem we are trying to solve with regards to exhaustive exploration for an NPC, except that in our case we are more interested in reducing redundancies in a route and not necessarily in finding the optimal route. There are two versions of the problem: fixed WRP and floating WRP. In the former the guard has a fixed starting point, whereas no such restrictions apply to the latter.

The fixed WRP was first discussed by Chin and Ntafos [7] in 1986. They showed that the problem is NP-hard for polygons with holes. As [28] points out, starting from 1991 onwards [8], there were several solutions proposed for the fixed WRP in a simple polygon, followed by observations that these did not work perfectly, along with flawed corrections. Currently, the best known solution for the problem is an $O(n^3 \log n)$ algorithm by Dror et al. [15]. For the floating WRP (i.e. WRP without restrictions) in a simple polygon, the first polynomial-time solution was published by Carlsson et al. [6]. Tan improved this with an $O(n^5)$ solution [37], and also provided a linear time 2-approximation algorithm for the problem in [38]. There have been variations of both AGP and WRP using limited visibility [24][41]. These theoretical approaches are quite complex, and as mentioned is NP-hard in the context of polygons with holes. However, it would be interesting to examine if ideas from such research can be used to improve the performance of practical approaches like ours.

6.3 Exploration

While the theoretical results of AGP and WRP are interesting, they usually do not consider coverage within the practical framework of an autonomous entity exploring a space. In this section we will touch upon research that does so in the fields of video games and robotics.

6.3.1 Games

To the best of our knowledge, there has not yet been any formalized work related to exhaustive exploration in the context of video games. There has been a lot of work done with getting NPCs to navigate spaces, and an useful summary of the various path finding methods can be found in this article by Klingensmith [25]. LaValle's book on planning algorithms is also useful [26].

We found one example related to "exploring" a space in research related to Real Time Strategy (RTS) games. RTS games involve complex strategic manoeuvering of groups of characters at once. These strategies are highly affected by a player's knowledge of the game map, and the location of resources and enemies therein. Hagelback and Johansson [21] use a well known model where the map is divided into tiles i.e. discretized, and then lead NPCs to unvisited tiles using a navigation strategy called potential fields, and give the main AI information related to those tiles only after an NPC has visited them. The strategy is only applicable for games that use the previously mentioned limited visibility technique known as *fog-of-war* (see *Section 5.1.2*), and does not use a visibility polygon, which is a tool that can account for more complex and varying frameworks. Lastly, their work is focused on how AI engages human players, and not the exhaustive exploration of a map.

As mentioned earlier, games like *Civilization* [11] and *Dungeon Crawl Stone Soup* [14] use a trivial greedy approach for automated exploration. These are widely discussed games whose source code is publically available. Their maps are tile-based, and from both observation of gameplay and analysis of the code, it has been seen that the greedy approach can be quite unsatisfactory, often failing to explore important areas, and taking more time than a manual approach [10].

6.3.2 Robotics

Robotics is an area that sometimes overlaps with the world of game AI. Therefore, it is not surprising that it has already produced a good deal of research on the subject of exploration. Much of this work has been focused around what is called the Simultaneous Localization and Mapping (SLAM) problem, also known as Concurrent Mapping and Localization (CML). The problem consists of getting an autonomous agent to simultaneously map an unknown environment (mapping), and to figure out its position within that environment (localization) [39].

In 1990 Smith et al. [35] first introduced a probabilistic solution for the problem, and an experimental analysis of different strategies that followed can be found in [1]. Certain strategies employ moving the agent in a fixed path such as concentric circles [34], while others use some sort of evaluation function to determine the next best point to move to [40]. Unlike exploration in a video game context, these algorithms have to account for hardware issues, and often focus on problems created by faulty sensor readings or a sensor's information extraction capabilities. The other major difference is that in our case the problem is simplified, since we are already aware of the structure of the map, and use processes that utilize that information.

There has been some work done on coverage of known environments as well by Rekleitis and other contributors [29]. Xu et al [42] uses the boustrophedon cellular decomposition technique to break down a region with obstacles, and then create a path that covers the map by visiting all the cells. The latter is done by solving the Chinese postman problem, which is a graph theory problem that asks for an optimal traversal of all edges in a graph. In contrast, our approach involves an attempt to visit all members of a subset (cameras) of all the nodes of a graph (roadmap), and can terminate before all are visited if coverage is complete. Like our approach theirs also tries to reduce redundancy in the planned path, albeit by using techniques in robotics. They also provide experimental results using an aerial device. It would be interesting to study the work in more detail and compare the two approaches.

Chapter 7
Conclusion and Future Work

This chapter provides some concluding remarks about our study. *Section 7.1* briefly summarizes the overall work and offers some final observations. In *Section 7.2*, we discuss the different avenues that this research can take in the future.

7.1 Conclusion

In this thesis, we developed a methodology by which NPCs can exhaustively explore game maps. Our methodology represents maps as 2D polygons with holes, and uses the Art Gallery Theorem to compute a set of camera points that collectively guarantee full coverage of the polygon. We then constructed graphs (roadmaps) to connect these camera points, and proposed different heuristic strategies to visit the points efficiently. To experimentally compare our strategies, we analysed how quickly they explored maps on average, making the necessary measurements with point visibility and shortest path algorithms. We concluded that the Maximum Union:Distance tour, a strategy which quantitatively assesses both distance and coverage as part of its heuristic, performs the best. Through a series of experiments, we were also able to determine the effect of different factors on such strategies in general.

Simplified Map Collection

Experiments were carried out on half a dozen maps from commercial games. We devised a map collection process (using our own parser) that only requires an overhead screenshot of the map and some manual tracing. The final output of the collection process is a list of easily readable geometric coordinates. This process can be used in any variety of experiments that require the essential geometric structure of a game level. The screenshots were sourced from consumer created walk-throughs and posts by players on gaming forums. These proprietary maps were not available in any form elsewhere, showing that crowd-sourced data can be beneficial to video game research.

Visibility Polygon Implementation

A significant amount of time was spent correcting numerical stability issues, particularly for the computation of visibility polygons. This fact, along with the dearth of robust visibility polygon libraries, and the difficulty of implementing the algorithms listed in *Section 6.1*, suggest that there is a need for work related to the practical aspects of visibility polygons.

Overhead in Unity

Unity served as a good tool for prototyping in large part due to its visualization capabilities. We were able to bolster the applicability of our work by presenting it in the form of integrated tools in this popular game development environment. However, for our experiments we needed to carry out larger batches of computation. And that is when Unity's computational overhead forced us to switch certain functionality over to libraries and other platforms. Therefore, Unity may not be the right choice for studies that do not require much visualization, or where the sole focus is on obtaining experimental results. To reiterate, one of the purposes of using Unity was to keep all tool development contained in within a single software framework.

7.2 Future Work

Below we list several ideas for the next steps in exploration research. Some have a more direct connection to the work implemented in this thesis than others.

Qualitative Testing

The need for automated exploration algorithms, their applications, and the lack of sophisticated methods have been thoroughly discussed. To further bolster the usefulness of our methods, it would make sense to carry out a qualitative assessment. This would involve implementing the algorithms in actual games, and using human testing to gauge impact on player experience.

More Complex Metrics

As mentioned in *Section 5.1.1*, the task of creating human-like exploration strategies for NPCs is difficult, simply because "human-like" is hard to define. A study into how human players explore game maps will likely produce insights that make the idea less nebulous. That in turn may lead to metrics that help us judge exploration strategies for NPCs in a more nuanced manner. Metrics may also be developed from examining relations between a map's specific geometric properties (e.g. perimeter, average edge length, curvature) and the average tour length. Such a relation is not obvious and would likely require a sophisticated method of structural analysis.

Curve Analysis and Progression of Tour Types

In *Section 5.2.1* we initially showed curves representing the progression of single instances of tour types. We then concluded that it is difficult to comment on a tour type by looking at the results of a single instance. However, a method that aggregates the curves (not simply the final tour length), for all instances of a tour type, would be able to show the general progression of that tour type. We would then be able to analyse the rate at which coverage increases, and be able to tackle questions such as whether new areas are discovered at an uniform rate or in bursts, and if there are long periods during which no new area is seen.

Even more interesting would be to compare such results with results for human players, which could help the development of new metrics.

Partial Visibility and Partial Exploration

Human players cannot process a 360 degree view of the game world. To make NPCs more human-like it would make sense to limit their field of view. This can be done by considering visibility polygon's rays within a certain angle, and if required allowed up to a certain distance. Exploration could be treated partially too. Even players that tend to focus on map exploration may not always be interested in exhaustive exploration. This can be due to the map not being interesting enough or its size. In such cases we could use "sufficient exploration." This would imply deeming an NPC's tour complete, once a designated percentage of the map has been covered instead of 100 percent. The level designer could also designate the regions that are important in the map, and exploration could be considered within the context of a separate map that only consists of those areas.

True Discovery

Even though we do not give our NPC visual knowledge of any area of the map before it has seen the particular area, we still give it instructions that are based on knowledge of the map. To make the NPC truly discover the game world in real time, we would have to avoid using such information. Here, the problem becomes similar to that of exploration in robotics (see *Section 6.3.2*), and it would make sense to consider the ideas present there. A randomized approach or fixed patterns of movement could be used, but some sort of spatial reasoning would be required to make the NPCs movement seem human-like. If we can incorporate strategies based on tests analysing how humans explore game maps, then such an experiment may provide insights that can even be useful for the exploration problem in the context of robotics.

Discretised Map

If we simplify the model further by using a discretised map that is divided into grids, the computation of visibility polygons could become slightly easier. It would certainly make

the merging of polygons much simpler, and potentially more efficient as well, at least in obstacle dense environments. This, of course, comes at a loss of accuracy with respect to visibility, although the tradeoff may be preferred in games where the precision of an NPC's vision is of a lesser priority.

Explorability

Once more complex metrics have been developed using insights from human testing, it would be possible to comment on the quality of exploration a map provides i.e. its "explorability." For example, further data may indicate that there is a correlation between an NPC's average tour length in a map and the time it takes for a human player to explore the same map. Level designers could then judge the explorability of maps by testing them with NPCs instead of human testing. This would save cost, and allow designers to build better maps at a faster rate.

Bibliography

[1] Francesco Amigoni. Experimental evaluation of some exploration strategies for mobile robots. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 2818–2823. IEEE, 2008.

[2] Tetsuo Asano. An efficient algorithm for finding the visibility polygon for a polygonal region with holes. *IEICE TRANSACTIONS (1976-1990)*, 68(9):557–559, 1985.

[3] David Avis and Godfried T Toussaint. An efficient algorithm for decomposing a polygon into star-shaped polygons. *Pattern Recognition*, 13(6):395–398, 1981.

[4] Richard Bartle. Hearts, clubs, diamonds, spades: Players who suit muds. *Journal of MUD research*, 1(1):19, 1996.

[5] Francisc Bungiu, Michael Hemmer, John Hershberger, Kan Huang, and Alexander Kröller. Efficient computation of visibility polygons. *arXiv preprint arXiv:1403.3905*, 2014.

[6] Svante Carlsson, Håkan Jonsson, and Bengt J Nilsson. Finding the shortest watchman route in a simple polygon. *Discrete & Computational Geometry*, 22(3):377–402, 1999.

[7] Wei-Pang Chin and Simeon Ntafos. Optimum watchman routes. In *Proceedings of the second annual symposium on Computational geometry*, pages 24–33. ACM, 1986.

[8] Wei-Pang Chin and Simeon Ntafos. Shortest watchman routes in simple polygons. *Discrete & Computational Geometry*, 6(1):9–31, 1991.

[9] Vasek Chvatal. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory, Series B*, 18(1):39–41, 1975.

[10] Various contributors. Automated exploration versus manual exploration. https://www.reddit.com/r/civ/comments/18wtqz/what_is_better_auto_explore_or_doing_it_yourself/. *Archived discussion on community forum for the game Civilization*.

[11] Various contributors. Civilization modding wiki. http://modiki.civfanatics.com/. *A wiki for the Civilization source code*.

[12] Various contributors. Unity. https://unity3d.com/. *Game development engine and platform*.

[13] Larry S Davis and Michael L Benedikt. Computational models of space: Isovists and isovist fields. *Computer graphics and image processing*, 11(1):49–72, 1979.

[14] DCSS Devteam. Dungeon crawl stone soup. https://crawl.develz.org/. *Open-source roguelike game*.

[15] Moshe Dror, Alon Efrat, Anna Lubiw, and Joseph S.B. Mitchell. Touring a sequence of polygons. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 473–482. ACM, 2003.

[16] Hossam El Gindy and David Avis. A linear algorithm for computing the visibility polygon from a point. *Journal of Algorithms*, 2(2):186–197, 1981.

[17] Steve Fisk. A short proof of chvátal's watchman theorem. *Journal of Combinatorial Theory, Series B*, 24(3):374, 1978.

[18] Subir Kumar Ghosh. *Visibility algorithms in the plane*. Cambridge University Press, 2007.

[19] Sean Gillies. Shapely. https://github.com/Toblerity/Shapely. *Python package for manipulation and analysis of geometric objects in the Cartesian plane*.

[20] Branko Grünbaum. *Convex Polytopes*. Springer-Verlag, New York,, 2003.

[21] Johan Hagelback and Stefan J Johansson. Dealing with fog of war in a real time strategy game environment. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, pages 55–62. IEEE, 2008.

[22] Paul J Heffernan and Joseph S.B. Mitchell. An optimal algorithm for computing visibility in the plane. *SIAM Journal on Computing*, 24(1):184–201, 1995.

[23] Barry Joe and R.B. Simpson. Corrections to lee's visibility polygon algorithm. *BIT Numerical Mathematics*, 27(4):458–473, 1987.

[24] Giorgos D Kazazakis, Antonis Argyros, et al. Fast positioning of limited-visibility guards for the inspection of 2d workspaces. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, pages 2843–2848. IEEE, 2002.

[25] Matt Klingensmith. Overview of motion planning. http://www.gamasutra.com/blogs/MattKlingensmith/20130907/199787/Overview_of_Motion_Planning.php. Accessed: 2015-11-03.

[26] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[27] D.T. Lee. Visibility of a simple polygon. *Computer Vision, Graphics, and Image Processing*, 22(2):207–221, 1983.

[28] Fajie Li and Reinhard Klette. Watchman route in a simple polygon with a rubberband algorithm. In *Proceedings of the 22nd Canadian Conference on Computational Geometry (CCCG2010)*, pages 1–4, 8 2010.

[29] Raphael Mannadiar and Ioannis Rekleitis. Optimal coverage of a known arbitrary environment. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 5525–5530. IEEE, 2010.

[30] K. J. Obermeyer and Contributors. The VisiLibity library. `http://www.VisiLibity.org`, 2008. *A C++ library for floating-point visibility computations*.

[31] Joseph O'rourke. *Art gallery theorems and algorithms*, volume 57. Oxford University Press Oxford, 1987.

[32] Michel Pocchiola and Gert Vegter. The visibility complex. *International Journal of Computational Geometry & Applications*, 6(03):279–308, 1996.

[33] Jonathan Richard Shewchuk. Triangle. `https://www.cs.cmu.edu/~quake/triangle.html`. *A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator*.

[34] Robert Sim and Gregory Dudek. Effective exploration strategies for the construction of visual maps. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 4, pages 3224–3231. IEEE, 2003.

[35] Randall Smith, Matthew Self, and Peter Cheeseman. Estimating uncertain spatial relationships in robotics. In *Autonomous robot vehicles*, pages 167–193. Springer, 1990.

[36] Subhash Suri and Joseph O'Rourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In *Proceedings of the second annual symposium on Computational geometry*, pages 14–23. ACM, 1986.

[37] Xuehou Tan. Fast computation of shortest watchman routes in simple polygons. *Information Processing Letters*, 77(1):27–33, 2001.

[38] Xuehou Tan. A linear-time 2-approximation algorithm for the watchman route problem for simple polygons. *Theoretical Computer Science*, 384(1):92–103, 2007.

[39] Sebastian Thrun et al. Robotic mapping: A survey. *Exploring artificial intelligence in the new millennium*, pages 1–35, 2002.

[40] Craig Tovey and Sven Koenig. Improved analysis of greedy mapping. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 4, pages 3251–3257. IEEE, 2003.

[41] Pengpeng Wang, Ramesh Krishnamurti, and Kamal Gupta. Generalized watchman route problem with discrete view cost. *International Journal of Computational Geometry & Applications*, 20(02):119–146, 2010.

[42] Anqi Xu, Chatavut Viriyasuthee, and Ioannis Rekleitis. Efficient complete coverage of a known arbitrary environment with applications to aerial operations. *Autonomous Robots*, 36(4):365–381, 2014.

[43] Qihan Xu, Jonathan Tremblay, and Clark Verbrugge. Generative methods for guard and camera placement in stealth games. In *Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2014)*, October 2014.

[44] Alireza Zarei and Mohammad Ghodsi. Query point visibility computation in polygons with holes. *Computational Geometry*, 39(2):78–90, 2008.