Procedural Guard Placement and Behaviour Generation

Qihan Xu

Master of Science

Department of Computer Science

McGill University

Montreal,Quebec

2015-4-15

A thesis submitted to McGill University in partial fulfillment
of the requirements of the degree of Master of Science

DEDICATION

This document is dedicated to the graduate students of the McGill University.

ACKNOWLEDGEMENTS

There are number of people without whom this thesis might not have been written. First of all, I would like to express my sincere gratitude to my supervisor Prof. Clark Verbrugge for the continuous guidance of my research and the writing of this thesis. Also I would like to thank my fellow labmate Jonathan Tremblay in Computer Games Lab for the stimulating discussions and all kinds of help that triggered my inspiration and enthusiasm to the research topic. I am also grateful to my parents Jian Xu and Qiji Hu, for giving birth to me at the first place and supporting me throughout my life. This research was also supported by the Fonds de recherche du Quebec - Nature et technologies, and the Natural Sciences and Engineering Research Council of Canada.

ABSTRACT

Enemy observers, such as cameras and guards, are common elements that provide challenge to many stealth and combat games. Defining the exact placement and movement of such entities, however, is a non-trivial process, requiring a designer balance level-difficulty, coverage, and representation of realistic behaviours. In this work we explore systems for procedurally generating both camera and guard placement in a stealth game context. We first investigate a Monte-Carlo approach to optimize randomized enemy positions and motions based on Voronoi-regions. We then perform an automatic roadmap construction, generating more specific patrol behaviours through a grammar-based technique and a rhythm-based technique. We evaluate both approaches with a non-trivial implementation in Unity3D, and apply quantitative metrics to demonstrate how different parametrizations can be used to control level difficulty without sacrificing believability.

ABRÉGÉ

Les observateurs ennemis, comme les cameras et les gardes, sont des éléments courants qui apportent du challenge à de nombreux jeux d'infiltration et de combat. Définir la position et le mouvement exacts de telles entités, néanmoins, n'est pas trivial, requérant un design équilibré de niveau de difficulté, une couverture spatiale, et une représentation de comportements réels. Dans cette recherche, nous examinons des systèmes visant à générer avec des procédures le placement de caméras et de gardes dans un jeu d'infiltration. Nous étudions tout d'abord une approche de type Monte-Carlo pour optimiser de façon aléatoire les positions et mouvements des ennemis basés sur des régions Voronoi. Nous réalisons ensuite une construction automatique de feuilles de route, en générant des comportements de patrouille plus spécifiques à travers des techniques basées sur la grammaire et des techniques basées sur du rythme. Nous évaluons les deux approches avec une implémentation non triviale dans Unity 3D, et nous appliquons des mesures quantitatives pour démontrer comment des paramétrisations différentes peuvent être utilisées pour contrôler le niveau de difficulté sans sacrifié la crédibilité.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
Introduction

A stealth problem in a game requires a player to move between locations, undetected by entities such as security cameras or AI-controlled guards. Defining the location, routes, and other parametrization of enemy agents, however, is a non-trivial task, made especially complex by the need to ensure a solution exists, provides an appropriate challenge to the player's ability to sneak, and that agent behaviours appear suitably realistic for the game context. This design is normally part of a long iterative process that can include much human effort before a designer reaches a desireable form of gameplay—designers repeatedly modify game levels and subject them to human-based alpha/beta-testing. To alleviate the burden, the idea of replacing some part of playtesting with automatic generation/evaluation naturally emerges.

In this thesis, we investigate two methods for procedural placement of enemy agents. We first consider a region-based guard and camera placement, and then a roadmap-based construction of mobile guard routes. For the former, we present an approach to automating simple guard and camera placement in a stealth game context. Our design is based on first decomposing a level-space into *Voronoi regions*, and then using that region geometry to define appropriate observer locations, selecting either basic line segments as patrol paths for simple guards, or static locations for rotational camera positioning.

Mobile guards introduce the additional complexities of planning reasonable movement routes that tour a space, along with specific behaviours guards may use to inspect the space at different points. For this we first build a simplified roadmap map based on a reduced and straightened *medial skeleton* [2]; this enables us to select appropriate end-points and paths between them that traverse the space without appearing random. Additional behavioural complexity is then introduced through two different ways: a grammar-based model that partitions the route into different kinds of movement and scanning segments, and a rhythm-based model that allows more flexibility in path length and the distribution of intra-route behaviours for an agent.

These automated approaches have obvious value in reducing the burden of manual level design and labour-intensive playtesting. They are also flexible, in the sense that the region-based and the roadmap-based approaches have trivial reductions that can ensure solutions exist. We then use a heuristic, Monte-Carlo approach to scale up the complexity, and enable a variety of interesting results. For these more complex situations we preserve solvability by constructing first an initial, arbitrarily complex arrangement of enemy agents, and then verifying that a solution exists through a search-based analysis technique. We demonstrate this process on three game level designs, two synthetic and one based on a commercial game, exploring different configurations to give some sense of appropriate parametrizations for our techniques.

This is the first work on procedural guard placement and behaviour generation. Specific contributions consist of

- an introduction to the overall workflow along with basic algorithms for region-based approach to automating mobile guard and camera placement in a 2D stealth level,
- the fundamental procedures of retrieving a robust roadmap for a given 2D level based on a generated *reduced straightened medial skeleton*,
- the design of a flexible, grammar-based model for defining intra-route activities,
- the design of a rhythm-based model for manipulating agent behaviours within the roadmap-based guard patrol routes, and
- application of quantitative metrics that demonstrate how different parametrizations affect the existence of level solutions and player perception of difficulty.

This thesis includes 6 chapters. For an easy reference, the chapters followed are organized as below:

- Chapter 2 provides the background and related work concerning procedural guard placement and behaviour generation for a stealth game. It basically involves three techniques: geometric visibility, procedural generation and *pathfinding* algorithms.
- Chapter 3 introduces a region-based approach to automating guard and camera placement in a stealth level. Guards move along a straight line and cameras rotate with an angle. The method uses the knowledge of a *Voronoi diagram* to determine areas that are well-observed by guards and cameras.
- Chapter 4 mainly talks about a roadmap-based approach that can generate more complex paths and behaviours for mobile guards. Paths, specifically, are a subset of the roadmap that is constructed by extracting the *straightened*

medial skeleton of level geometry through a *grassfire algorithm*. Behaviours are then generated by either following grammatical rules or a rhythm controller.

- Chapter 5 shows a number of experiments and results on multiple testing levels in order to demonstrate the validity and efficiency of our methods in terms of tweaking difficulties.
- Chapter 6 gives a conclusion on our topic based on the results from our experiments. We also discuss some foreseeable future work.

CHAPTER 2
Background and Related Work

Stealth problems can be loosely defined as tasks wherein the player must move from a start to a goal position, while avoiding enemy Non-Player Character's (NPC) Fields of View (FoV). This problem is presented to players in many combat games, and is central to games in the stealth genre, such as *Mark of the Ninja*, *Thief* or the *Metal Gear Solid* series.

Designing such a game or level involves careful orchestration of enemy positions and behaviours, obstacles, and other positive or negative factors that affect stealth (light, shadow, sound, etc.) [17] so as to present an interesting but feasible challenge to the player.

Within our exploration of stealth games, the design space is limited to placing geometries (occlusions) and two flavours of NPC: cameras and guards, both of which have limited FoVs. Cameras do not move, although they can rotate, sweeping a given subset of the level. Guards refer to entities moving along a repeating, predetermined path (patrol route) where they are also able to perform rotating and pausing activities similar to cameras. Guards can thus be considered as cameras with movement mechanics.

2.1 Relevant Topics

The fundamental task of generating stealth levels combines problems in geometric visibility along with procedural generation. Additionally, the quality of a stealth

level is typically measured through playtesting where we determine the difficulty of our levels based on a pathfinding algorithm that mimics a player moving from start to goal. Here, an incremental sampling and searching algorithm—*Rapidly-exploring Random Tree* (RRT) is introduced to facilitate our measurements. Abstract solutions to related problems have been explored in more theoretical contexts.

Visibility - In a stealth game, placement of enemy entities such as cameras and guards should permit a solution to exist, and we can view the core problem as one of finding a distribution of *n cameras* and *m guards* such that a path exists from *start* to *goal* that does not intersect any enemy FoV. Previous work has investigated the problem of finding such a solution as a pathfinding problem [21]. The problem of creating the initial camera/guard placement, however, is more closely related to visibility problems in computational geometry. Placing fixed entities (with infinite FoV) in a given n-vertex polygonal geometry is well known as Klee's 1973 "art gallery" problem, and there have been many different algorithms and lower bounds demonstrated for variations of this problem [13]. More complex formulations also exist; Erdem *et al.*, for example, presented a realistic expression of the art gallery problem where guards do not have 2π rotational and infinitely long view [5]. Using a discrete (grid-based) world they optimally place cameras that can ensure every point is observed with period less than t. For this they reduced the problem to a *Set Coverage Problem*, solving it using a special case of integer programming. Cohen-Or *et al.* give a general survey of other, real-life application of camera network placement [3]. Stealth games extend this problem, introducing time/movement-models, as well as

the additional complexity of ensuring the placement is imperfect, admitting a solution, with some degree of challenge for a human player. In this context, Xu *et al.* [23] found a heuristic approach to camera placement based on weakening a solution to the "art gallery problem" for simple polygons.[1] By both limiting individual camera coverage (FoV distance or FoV angle) and by reducing the number of cameras, they managed to provide a parametrized method for ensuring an incomplete coverage, and thus a potentially solvable level design. With a better understanding of static entities, a combined approach to guard/camera placement is being explored in this thesis. We present a space-decomposition approach that locates cameras with (heuristically) maximized coverage, and can also be used to define simple, straight-line guard patrol paths. Then we improve our work by generating significantly more complex guard routes and behaviours, formulating the generation with specific rules, and providing a series of experiments to prove the impact of our approaches on game difficulty. The following chapters are going to explain this in-game procedural content generation problem in details.

Procedural Generation - Our camera/guard placement approach is intended to help in procedurally generating stealth levels. Procedural Content Generation

[1] Note that I am a co-author of this publication. This camera placement solution, however, refers to an approach described in this paper that was primarily developed by another of the co-authors, which builds on Fisk's solution to the art gallery problem [13]. By triangulating and 3-colouring the level geometry, this design achieves a minimal camera arrangement, using $\lfloor \frac{n}{3} \rfloor$ cameras for a polygon with n vertices. Two camera properties—field of view distance and angle were studied and compared.

(PCG) has been used for a wide variety of tasks in games: room/obstacle generation (*Spelunky*) [24] [8], weapon generation (*Borderlands*) [18], vegetation filling (*Skyrim*) [19], *etc.,* and continues to be a popular research topic. Shi and Crawfis, for instance, presented a design tool that tries to optimize distribution of objects within a level, based on different properties of player paths, such as the minimum-damage cover, longest path, and standard deviation of cover points [15]. In our work we enable flexible, high-level structuring of the content generation through use of a grammar-based rewrite system, generating increasing complexity by applying rewrite rules. Grammar-based approaches have been used for a number of other PCG applications, including vegetation [20], abstract structure for platformer levels [16], narratives [9], and even as a general tool that creates whole story, missions and levels [4].

Pathfinding - Measuring our approaches requires building multiple paths that mimics a player's attempts to move from an initial position to a goal position. This path is not necessarily a successful one since a player may actually fail. The *A* algorithm* [7] is normally considered one of the best ways to find a *feasible* and *collision-free* path in game industry. However, it only provides one successful solution, that is, the shortest path from start to goal. Additionally, A* is deterministic and cannot appropriately represent a variety of paths (even failed ones) that a player may attempt. Thus, a more heuristic algorithm is sought. Rapidly-exploring random tree (RRT) is an incremental, sampling-based search algorithm [11]. It builds a search tree that gradually improves its resolution in the search space, branching

out from an initial configuration in an attempt to reach the goal state. This incremental construction is guided by a dense, random sequence of samples. In the limit, the tree densely covers space, avoiding all the obstacles [10]. The RRT has proved extremely useful in exploring configuration spaces and ultimately generating trajectories for robotic systems, with some recent applications in games [1]. In our work, RRT offers us a flexible and inexpensive way to explore a state space. Given an upper bound on the total number of samples per search, RRT could also fail, and combined with its random behaviour this allows us to easily represent a wide range of player behaviours.

CHAPTER 3
Region-based Guard and Camera Placement

In this chapter, we present a region-based approach to automating guard/camera placement in a stealth game context. We start from introducing the workflow of our method, then move on to describing the essential technique of selecting regions that is common to both approaches, and finally conclude by explaining the unique processes that either populate guards or cameras.

3.1 Workflow

Our region-based approach to solving this guard/camera placement problem is based on first decomposing a level-space into separate regions, and then using that region geometry to define appropriate observer locations, selecting either basic patrol paths for mobile guards, or static locations for rotational camera positioning. An overall workflow is illustrated in Figure 3–1. Note that while we present the generation of guards and cameras as distinct tasks, it is trivial to combine these steps and produce a population of both moving guards and rotational cameras.

In order to meet a specific level of stealth gameplay, the design applies a series of Monte-Carlo algorithms to content generation, iteratively optimizing an enemy distribution. Therefore, the general process is procedural and stochastic, allowing for high variability in the generated output, scalable time complexity, and easy integration into an iterative design process.

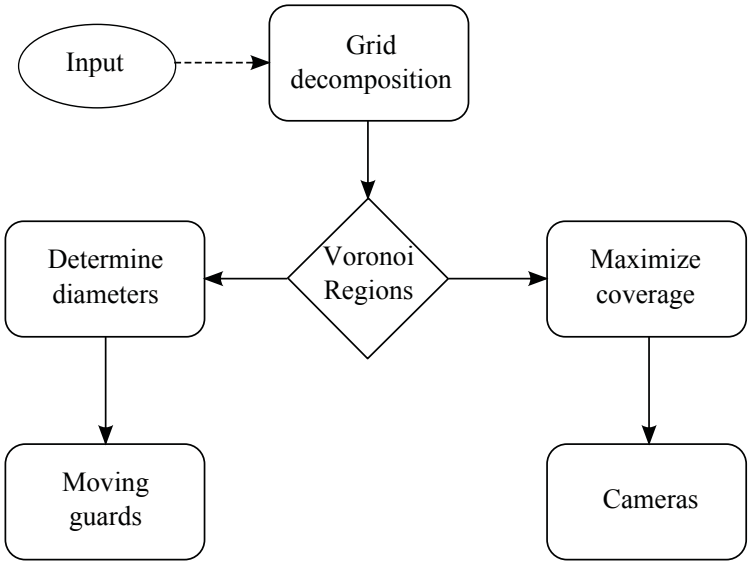


Figure 3–1: Workflow and main stages for automating guard and camera placement where generating *Voronoi regions* serves as a basis of the subsequent content generation.

3.2 Discretization and Regions

Although 3D elements are sometimes used in stealth games, the stealth pathing problem is most typically a 2D one, and thus can be addressed in terms of its underlying floor-plan. As illustrated at the beginning of Figure 3–1, we start off by introducing a 2D level-space as the *Input* to our method. In Figure 3–2 a 2D floor-plan acquired from the top view shows a non-trivial but typical space, featuring boundary edges, obstacles, a start position, and a goal position.

Placement of both moving guards and rotational cameras is based on a coarse-grained region decomposition. This ensures a good separation of our entities, and guides the actual definition of positions/movements. As shown in Figure 3–1, we refer to this procedure as *Grid decomposition*. In this step, regions are created

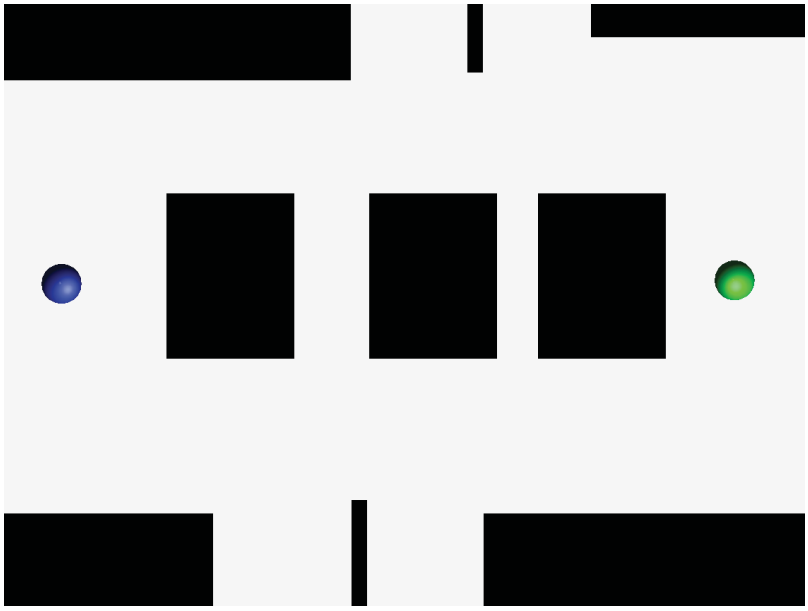


Figure 3–2: Input: 2D floor-plan wherein obstacles in black, walkable regions in white. The green sphere is the *goal* and blue the *start* position.

based on an initial discretization of the space into (fine-grained) grid cells which are squares of equal size. This allows us to apply simple flood-fill algorithms in further decomposition and analysis. Figure 3–3 shows the result after applying this first step to the original 2D level from Figure 3–2, with green cells indicating walkable areas and red (covered by black) indicating obstacles. Note that an entire cell is considered unwalkable if any portion within its boundary is covered by obstacles.

Region decomposition is created based on an initial seed location for each region. We randomly choose k non-obstacle cells as the starting seed points, separated by a minimum distance of d. Note that the k starting points are considered as region centers and the separating distance d necessarily avoids them being too close to each other. Additionally, for flexibility in guard/camera placement, and to also help

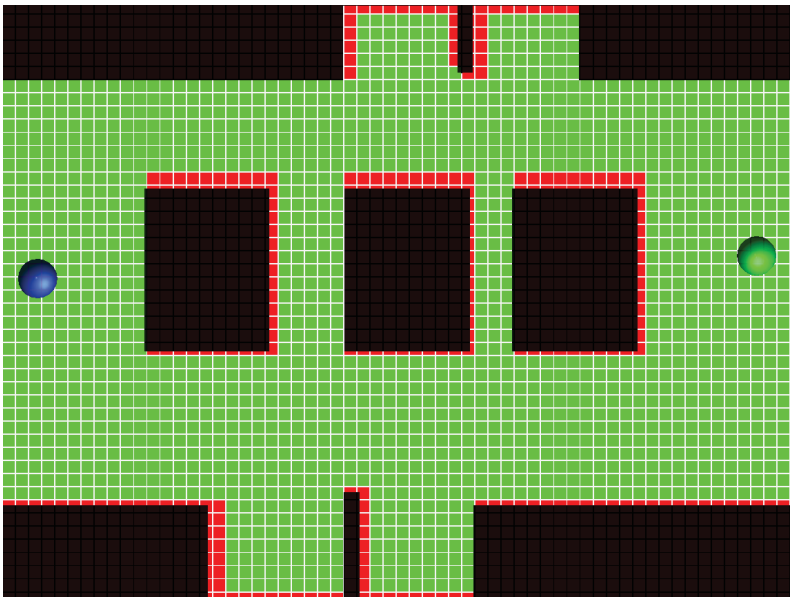


Figure 3–3: Discretization of the input: walkable in green, unwalkable in red. The green sphere is the *goal* and blue the *start* position as mentioned previously. The red cells around obstacles appear offset due to the way obstacles boundaries intersect the underlying grid, and the perspective view of the Unity client.

ensure a level remains solvable, we actually generate more regions than necessary, using $k = 2n - 1$ regions for n guards or cameras. In other words, having $k \geq n$ opens up more possibilities for stealth solutions.

From these k initial points we then compute a discrete, constrained Voronoi diagram. In Fig 3–1, this step is referred to *Voronoi regions* in the center diamond. To begin with, a Voronoi diagram is a plane partitioning according to the *nearest-neighbor rule*: each point in the plane is associated with the region whose center is closest to it. Let R being a set of k points in the same plane denoted by $r_1, r_2, ..., r_k$. Given R, a Voronoi diagram divides a plane into k Voronoi regions with the properties below:

- Each point r_i lies in one region at least.
- If a point $p \notin R$ lies in the same region as r_i, then the Euclidean distance from r_i to p will be shorter than or equal to the Euclidean distance from r_j to p, where r_j is any other point in R.
- If the Euclidean distance from a point $p \notin R$ to $r_i \in R$ is equal with its distance to another region center $r_j \in R$, then point p is on the *boundaries* of the regions centered around r_i and r_j.

According to this definition, we thus refer to k starting points as region centers, find the set of cells closer to each center than to any other and build the Voronoi diagram for our discretized level.

A naive and straightforward method of computing a Voronoi diagram has complexity $O(nk)$ where n denotes the number of total cells and k the number of region centers. Intuitively, we can simply iterate every walkable cell, calculate its Euclidean

distances to all region centers and then compare the results one against the other. Once we find out a region center $center_j$ that the cell $cell_i$ is closest to, we would then assign the tag of it to $cell_i$. Note that we initialize centers with unique tags and leave other cells blank at the beginning of the algorithm. As a final output, all blank cells should have their tags identical to the tag of center which they are nearest to. This algorithm is described in Algorithm 1.

Algorithm 1 Compute Voronoi Diagram

```
1: procedure CalculateVoronoiRegions(n, k, N, R)
2:     InitializeTags(k, R)
3:     for each cell_i ∈ N do
4:         minDist ← MAX, tempDist ← 0
5:         for each center_j ∈ R do
6:             tempDist ← Dist(cell_i, center_j)
7:             if minDist > tempDist then
8:                 minDist ← tempDist
9:                 tag_cell_i ← tag_center_j
10:            end if
11:        end for
12:    end for
13: end procedure
```

This Voronoi diagram, however, cannot fully represent a stealth level given the fact that a Euclidean distance becomes invalid when the line segment between two points encounters any obstacles. We thus proposed a constrained version of Voronoi diagram instead. By "constrained" we mean an actual *path distance* is sought when we make distance calculation. In this way, we are able to take obstacles/barriers into account. Figure 3–4, for example, shows a scenario where Euclidean distance is less preferable than path distance. Suppose the *red* and *blue* circles represent two regions centers while the *yellow* and *green* squares represent (the centres of) two

grid cell unassociated with any region. If we use the Euclidean distance, the *yellow* cell is closer to *red* than to *blue*, and the *green* is closer to *blue* than to *red*. The line segments with shorter distances are shown with black-dotted lines. As these distances both pass through the gray wall, however, this represents an infeasible connection. In this case we must compute the path distance, detouring around any such obstacles. The solid lines shown in Figure 3–4 show minimal paths that reach the closest region center, which ends up different from the center used in Euclidean distance.

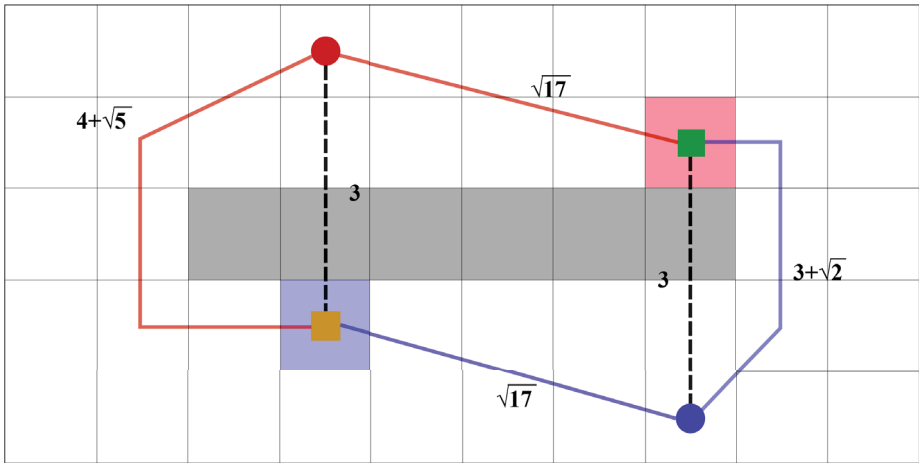


Figure 3–4: Euclidean distance vs. path distance: an obstacle is indicated by the *gray* rectangle, region centers by *red* and *blue* circles, grid centers by *yellow* and *green* squares, inappropriate connections for calculating Euclidean distances by *black-dotted* lines, and minimal paths by solid *red* and *blue* line segments. The *light red* and *light blue* squares indicate which region center the grids should belong to under path distance. Note that one grid cell is 1 unit in length.

In order to generate a constrained Voronoi diagram, we find a flooding solution that is suitable for a discretized level-space, and which can be accomplished in a reasonably fast manner—within $O(n)$ time where n denotes the total number of cells.

Inspired by the "ripple effect," we flood outward from different region centers and produce the boundary of each Voronoi region where wavefronts meet. This flooding approach is resolved by keeping a queue of cells that changes its size until there are no elements in it. Specifically, we assign different tags (from 1 to k) to k region centers and initialize the queue with those centers. Then we dequeue the first node, assign its untagged neighbours with the same tag and enqueue these tagged neighbours. Note that 8 direct neighbours are traversed, and thus (unlike our abstract example in Figure 3–4) we assume integer steps in every adjacency direction. And if a cell does not have any untagged neighbours, we simply drop it from the queue and move on to the next cell. Therefore, the time when the queue is empty is when a Voronoi diagram is constructed and all tags indicate which region a cell belongs to. The algorithm is described by Algorithm 2.

Algorithm 2 Compute Voronoi Diagram, Flooding

```
1:  procedure CalculateVoronoiRegionsUsingFlooding(n, k, N, R)
2:      InitializeTags(k, R)
3:      InitializeQueue(Queue)
4:      for each cell_i ∈ R do
5:          Enqueue(cell_i)
6:      end for
7:      while Queue.size ≠ 0 do
8:          Dequeue(p)
9:          if p.Neighbors ≠ NULL then
10:             for each cell_i ∈ p.Neighbors do
11:                 tag_{cell_i} ← tag_p
12:                 Enqueue(cell_i)
13:             end for
14:         end if
15:     end while
16: end procedure
```

Figure 3–5 shows an example of the result for $k = 5$. Seed points are indicated by the small brown dots, and the distinct regions by different colors.

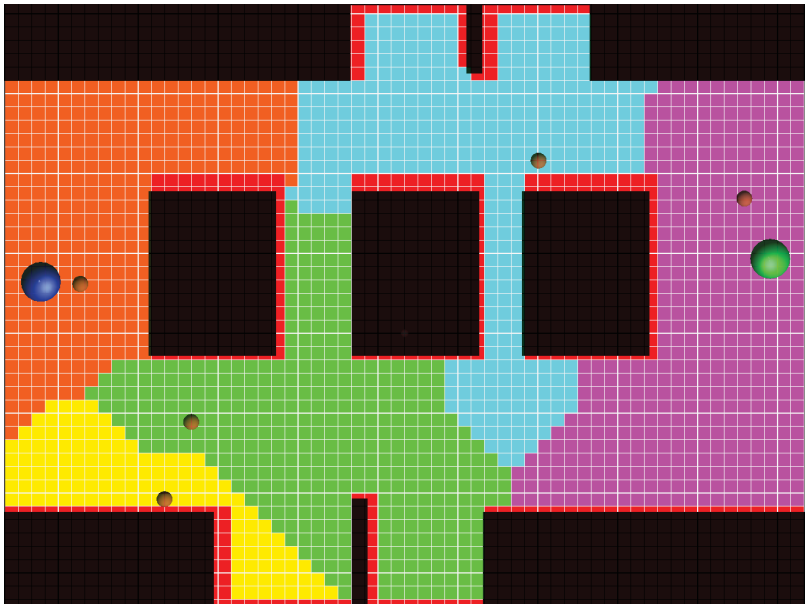


Figure 3–5: Discrete Voronoi regions constructed from our $k = 5$ seeds: regions are colored by *orange*, *yellow*, *green*, *cyan* and *purple*, respectively.

3.3 Moving Guards

Generating arbitrary enemy motions is a very difficult task. We split our research into two types of enemy entities: moving guards and cameras as shown by the two branches after computing *Voronoi regions* in Figure 3–1. In this section we investigate moving guards and restrict them to patrolling a straight line within a region, with FoVs always facing forward. This is not a complex movement strategy, but even simple straight line patrolling is representative of guard movements found in multiple stealth genre games, including *Mark of the Ninja*. It also constitutes a non-trivial

problem, where we need to ensure that patrol paths span the bulk of a region, and thereby heuristically provide a good non-player character coverage of that area.

To fulfil such a patrol task for moving guards, we assume there exists a relatively long line segment within each Voronoi region. We refer to this step as *determining diameters* in Figure 3–1. If our region were convex, we could calculate a *diameter* of polygon with the well-known rotating calipers algorithm, a conceptually simple and efficient ($O(n)$) algorithm [14]. A diameter of a convex polygon is the largest distance between any pair of vertices. An approximation to it, either in terms of distance or direction, suffices to provide a path for moving guards that should also leave open space for players to explore. Thus, an *approximate diameter* would produce such satisfying line representative for patrolling use.

Our situation, however, is more complex due to the fact that obstacles could make a Voronoi region non-convex. In a non-convex scenario, an intersection calculation is required to see if two points can be joined without going outside of the boundary. Figure 3–6 gives an example of a non-convex shape where the basic idea for computing the diameter of a convex polygon does not apply, and so cannot be directly used for defining a patrol path. A possible solution to this level is shown in the rightmost polygon.

Our heuristic procedure for finding such a patrol path works as follows. First, we begin by selecting the n biggest (in terms of area) regions from the k generated regions. Use of distinct regions avoids overlap in guard paths, and thus better coverage. Within each of these n regions we then randomly pick two points until we obtain two points visible to each other in the region. This process is repeated several

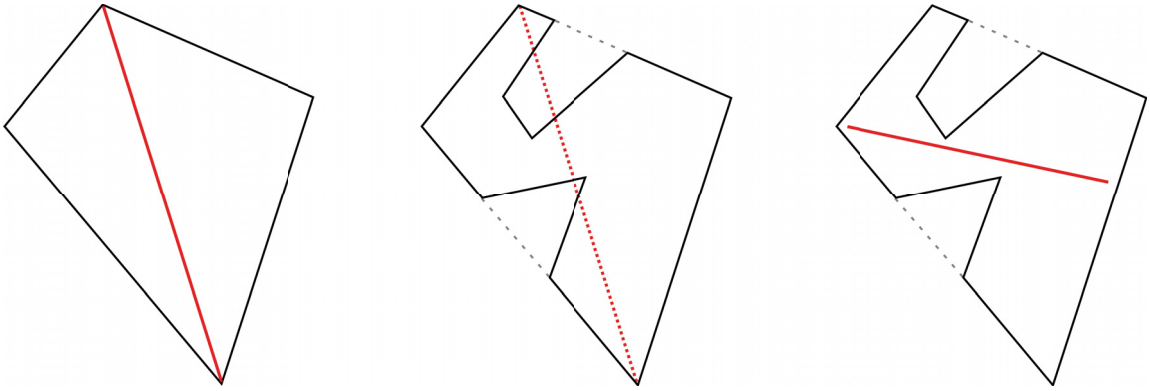


Figure 3–6: The diameter of convex polygons does not apply on non-convex polygons.

times, looking for two points with the largest separation, and so computing an approximate *diameter* for the region. After some number of iterations, the maximally separated pair of points found is then used to define the patrol path for a guard assigned to that region. By this means, our algorithm gives us a well-exploited space that moving guards should cover in game.

Figure 3–7 shows an example of three guards and their patrol paths as generated by our approach. Each guard path consists of two waypoints shown as white dots, with the actual route/direction represented by the orange arrows that span them.

As an additional requirement, we also ensure no patrol path results in a guard initially looking at the player's start position, and that guards do not constantly observe the goal position, both of which lead to an unsolvable level. As a final step in this branch, we define the movement of the guard, from one end of the patrol path to the other and back to the beginning. *Moving guards* (in Figure 3–1) are now well-generated, equipped with meaningful tasks and actions.

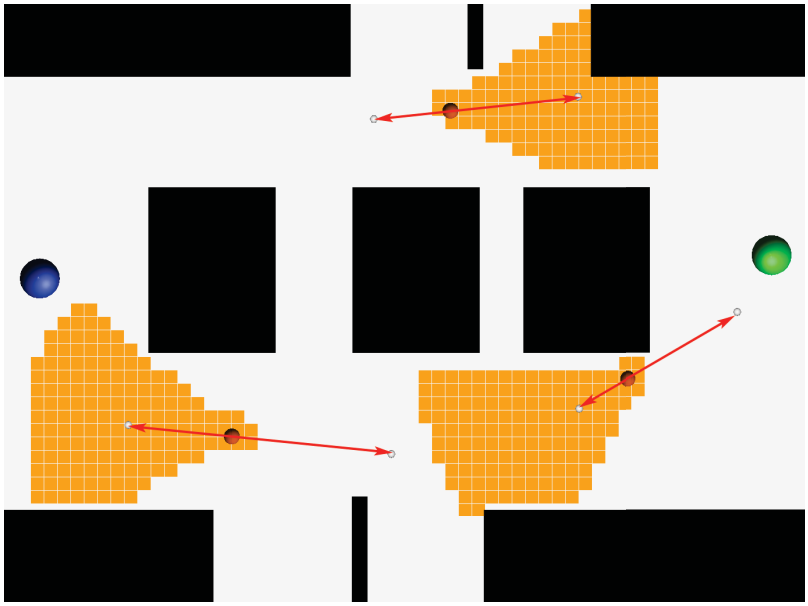


Figure 3–7: Patrol paths for three moving guards.

3.4 Rotational Cameras

In the following, we introduce another type of non-player character—rotational cameras. These agents, static in position yet dynamic in action, also provide visual coverage of the region of interest, which would therefore influence the stealth gameplay of a player. As we did for moving guards, we impose specific constraints on camera behaviours to limit the generational complexity. We assume cameras are usually fixed on walls, sweeping their FoV back and forth between two angular extremes.

We again begin by choosing the n largest regions. We randomly select a point (cell) along the boundary of the region as a camera location; ideally, this is also an obstacle boundary, although that is not possible in all cases, as a region may be entirely defined by common boundaries in sufficiently open area. For each point, we

iterate through 8 possible directions (*i.e.,* 45° intervals) and keep track of the two longest lines of sight. This process is repeated by some number of times and the candidate pair with a largest total line of sight is kept: similar to moving guards, we are choosing the largest line of sight in order to maximize the coverage of every camera.

The angle formed between these two lines of sight, directed inside the polygonal region, then defines the camera's sweep. Note that we apply an algorithm in determining which angle the camera should sweep given the fact that the two lines of sight can be swept either clockwise or counter-clockwise. Specifically, we generate two rays that both divide the acute and non-acute (obtuse) angle into two, extend them to outward until they encounter an obstacle, and pick the rotation direction that has the longer ray. This is illustrated in Figure 3–8 below—the angles chosen have been highlighted by shadows. This procedure results in a good camera placement because it maximizes visual coverage of a camera at its position, focusing on open area that is crucial to be utilized by a player, rather than trivial area near the obstacles that is less likely to be used by a player. Figure 3–9 shows a more complete example: here the camera first iterates through 8 directions, then determines two lines of sight with longest distances (indicated by black arrows) and finally selects the region in between as its main surveillance area.

Again, we ensure the player's start point is not initially covered, and the goal point is not continuously observed. You may refer to this step as *Maximize coverage* in the workflow chart shown by Figure 3–1.

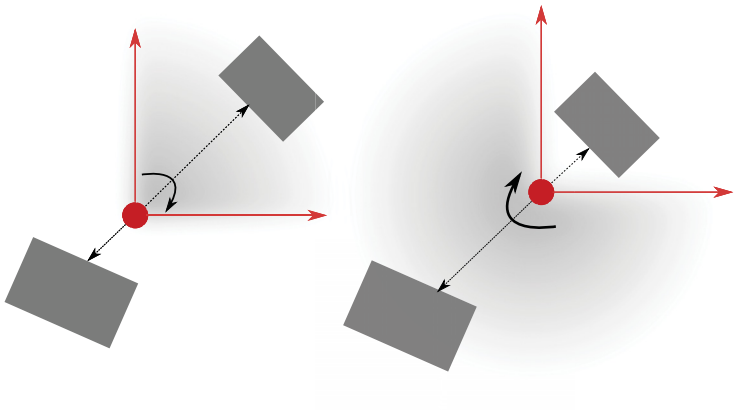


Figure 3–8: Acute angle vs. obtuse angle.

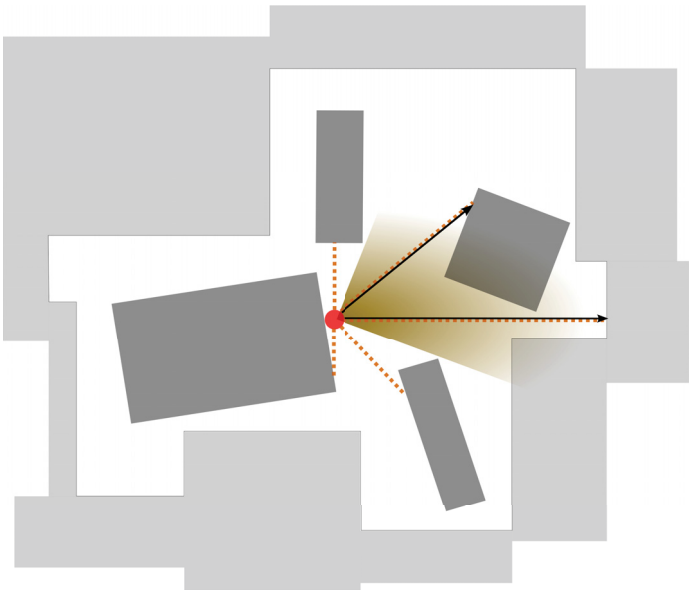


Figure 3–9: Inspecting area generated by choosing two longest lines of sight. The camera is shown by a *red* circle, possible directions are indicated by *orange* dotted-lines, the selected directions with two longest distances by *black* arrows, and the final viewed area by the *yellow* shading.

Figure 3–10 shows an example distribution of cameras and their rotational behaviours as generated for our example level. Cameras are fixed in positions near the obstacles, rotating by following the directions shown by the orange arcs. The leftmost camera is rotating with an angle of 180° which is sometimes, but not initially or always inspecting the spawning position of the player. The topmost rotates with an angle of 90°, and the bottom camera with an angle of 45°. They effectively prevent the player from reaching his/ her goal too easily through the two main hallways. Thus, this example of effort of in-game guard/camera generation implies that the camera placement can result in a reasonably challenging level design.

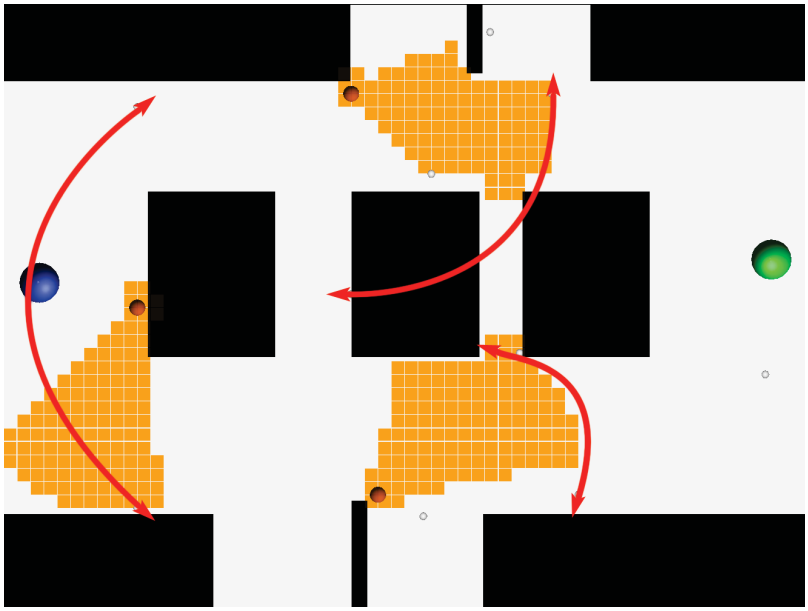


Figure 3–10: Three cameras and their rotational behaviours.

CHAPTER 4
Roadmap-based Guard Placement

In the following chapter, we discuss a roadmap-based technology for procedural generation of enemy agents, considering the mobile guards and their patrol routes in particular. We first give an overview of the roadmap-based approach, comparing it with the region-based one. This is followed by fundamentals of the approach, describing the reduced and straightened *medial skeleton* along with its abstract graph simplification, which is then used to build a roadmap for the game level. Based upon the roadmap, we then investigate different generative methods for constructing patrol routes and intra-route activities.

4.1 Overview

The Monte-Carlo approach for guard/camera placement in the previous chapter gives freedom over the choice of agent type, the number of enemy entities and the coverage of level-space. Using this method, a game designer is able to effectively tweak the level difficulties. Yet some noticeable drawbacks also exist, and for guards it may require many iterations before we obtain a satisfying coverage of a Voronoi region. It also lacks variety in terms of patrol route and intra-route activities.

A roadmap-based approach should ensure guards patrol a space with better individual coverage and allow for more complicated movements as well. This approach introduces a number of additional complexities to defining the movements of enemy agents. A patrol route, for instance, can be more complicated than a straight line, and

so require a geometrical path defined by multiple, connected line segments. Guards are also usually expected to make occasional observations as they patrol, sometimes looking around like cameras rather than just staring forward as they move. To solve these issues, we explored a roadmap-based method for constructing more meaningful, human-like and realistic guard movement. Within a roadmap, guards are able to choose a complex multi-segment path, wander around and present various behaviours during their patrol.

Our approach is based on two techniques that can easily and efficiently ensure an appropriate result for guard simulation, while still providing ample, flexible control for difficulty adjustment. We perform this in three stages, first generating a *graph representation* of the roadmap, then constructing a basic *patrol route*, and finally introducing *intra-route activities* to add further complexity and interest.

4.2 Medial Skeleton and Roadmap

Generally, guards move in a repetitive fashion through some subsets of the level geometry, concentrating on large areas, and avoiding obstacle features. Similar behavior patterns are widely used in games such as *Commandos* series. By repeatedly following a deterministic route, guards mimic fixed patrols, while giving a player the ability to observe and learn enemy movements. By inspecting wide areas, guards indicate a casual, lower level of vigilance, which also opens chances for players. Therefore, from a player's perspective, one can predict and prepare for guard movements, and find a solution to the level. To help with an appropriate patrol route construction, we thus generate a reduced and straightened subset of the *medial skeleton*, and use it to build a roadmap from which we can extract patrol routes.

A *medial skeleton* [6] basically provides a simplified view of a shape by approximating its local symmetry axis. Formally, each point of medial skeleton is defined as the center-point of an inscribed circle that maximally touches the geometry outline. Informally, it is defined as the a thin version of the shape that is equidistant to its boundaries. A straight skeleton, as it literally means, avoids curves that can be present in the basic medial skeleton; this is useful in our game context, as path waypoints assume straight-line connections. Figure 4–1 shows an example of straightened medial skeleton being produced: it serves as a nice topological representation of the walkable area because it forms a connected structure, interior to the geometry, the subsets of which contain information of many possible paths. In Figure 4–1, for instance, one noticeable patrol route worth investigating can be identified, shown as the green line segments in the rightmost picture. This is a subset of the (straightened and reduced) medial skeleton extracted from that level. Such routes have the characteristic that they do not connect directly to corners of our level space, and so put more emphasis on touring the guards through hallways rather than walking to and from corners, which would be less natural.

Our input level space consists of a discretized grid of cells, each marked as either walkable or not, but not specifically identifying individual obstacles. In order to build a skeleton, a first step is necessary to identify different obstacle boundaries as well as the level boundary. To clarify, the level boundary of our floor-plan is the edges of a bounding polygon the that can minimally include every walkable cell, and supposing there is a guaranteed path between any two cells within these walkable areas. The boundaries of interior obstacles then define non-traversable "holes" in this outer

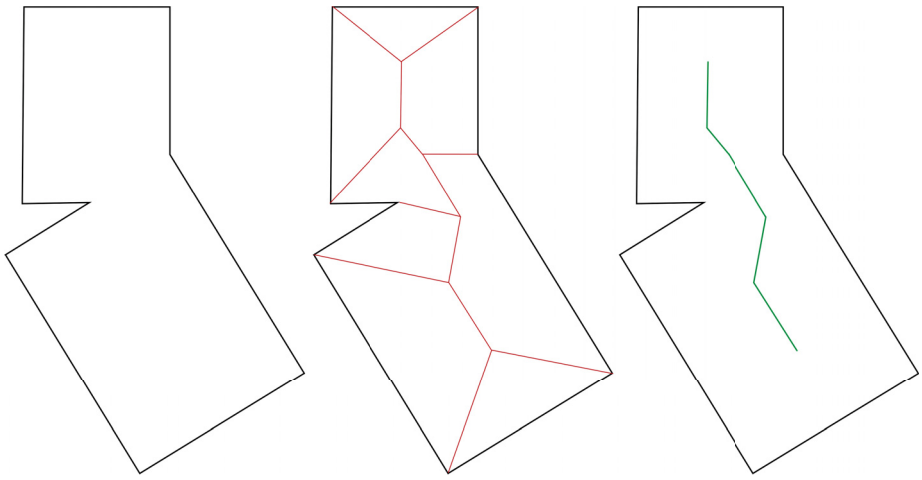


Figure 4–1: An example of straightened medial skeleton. The left image shows a simple level with two rooms connected. The red segments in the middle are *straightened medial skeleton* of the geometry. In the right image, lines connected to the outer polygon's vertices are removed, generating a *reduced straightened medial skeleton* (green) as the final roadmap.

polygon. Therefore cells C within the level can be divided into three categories—(level or obstacle) boundary cells, inner-obstacle cells and non-obstacle cells. A boundary consists of a list of boundary cells, and to keep track of every boundary, we build a list, *boundaryList*, containing all the different contours. Meanwhile, a global *boundaryCnt* (starting from 0) is used to indicate the number of boundaries being found. Once a new boundary is identified, we add it into *boundaryList* and increase both *boundaryList.size* and *boundaryCnt* by 1. We also need a variable to distinguish cells belonging to different boundaries as well as to distinguish boundary cells from other cells (inner-obstacle cells and non-obstacle cells). We thus assign to all cells a *boundaryBelongTo* with an initial value of −1, indicating that they do not belong to any boundary for the moment. This variable eventually turns into a meaningful value from 0 to n representing either a unique boundary index association

or an unreachable cell—note as we assume all non-obstacle spaces are reachable, the 0 index means the cell is inside the boundary of an obstacle.

We use a depth-first search to construct the boundary contour. Before we start, we initialize each cell with two booleans: *isVisited* and *isObstacle*. The former attribute avoids duplicated visits during depth-first search and the latter indicates whether a cell belongs to an obstacle (including the outer boundary). Within a depth-first search, if a cell belongs to an obstacle or it belongs to an edge of the level, we set its *isVisited* as true. Then, we check all its neighbours to decide whether it is also a boundary cell. If it is, we create a new boundary and add the cell to it or simply add the cell to the current boundary. Afterwards, we take *boundaryCnt* as an argument of the recursive function, pass it to its neighbours and thus search deeper in order to find the profile of different boundary contours. This algorithm ends up with all cells obtaining indices from -1 to n. Figure 4–2 shows 5 different obstacles (3 tilted and 2 orthogonal) and their boundary edges in 5 different colors, with the yellow surround produced by the level boundary. Each color here corresponds to a boundary index as indicated in the figure. Also note that due to the overlay between obstacles and boundary cells, the boundaries appear to be variant in thickness, although they are actually all one-cell width. The algorithms for computing this are depicted in Algorithm 3 and Algorithm 4.

After retrieving all the boundaries, we use an approach resembling the grassfire algorithm proposed by Blum [2] to compute the straightend skeleton. Conceptually, our cells are equipped with the following properties:

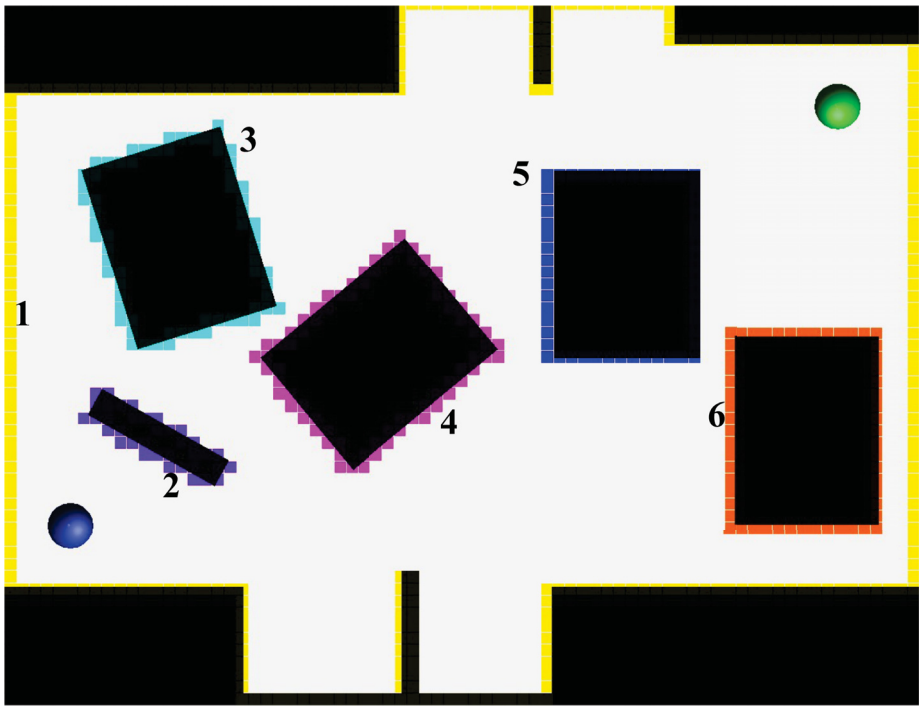


Figure 4–2: An example of boundary/obstacles edges and their representations in different colors. The blue sphere (lower-left) indicates the player's *start* position, and the green sphere (upper-right) the *goal*.

Algorithm 3 Identify Boundaries Part 1

```
1: procedure IdentifyBoundaries(C)
2:     new boundaryList
3:     for each cell_ij ∈ C do
4:         cell_ij.boundaryBelongTo ← −1
5:     end for
6:     boundaryCnt ← 0
7:     for each cell_ij ∈ C do
8:         boundaryCnt ← boundaryCnt + 1
9:         DepthFirstSearchForBoundaries(i, j, boundaryCnt)
10:        ▷ Undo the increasing of boundaryCnt if depth-first search does not find
11:        any boundary contours
12:        if boundaryList.size ≠ boundaryCnt then
13:            boundaryCnt ← boundaryCnt − 1
14:        end if
15:    end for
16: end procedure
```

- excitation—each cell can have a value ranging from −1 to n where −1 means unexcited and the rest excited;
- propagation—each excited cell can excite an unexcited neighbouring cell with a delay proportional to the distance;
- dead time—once excited/fired, an excited cell will no longer be affected again.

In practice, this approach requires expanding all edges, flooding outward from obstacles and inward from the level boundary. When different boundary wavefronts meet, the flooding stops and defines a portion of the skeleton. By executing the above operations synchronously, we are able to generate the medial skeleton for the walkable geometry.

The flooding algorithm (see Algorithm 5) is similar to Algorithm 2 in Chapter 2. We first gather all the unvisited cells (*boundaryBelongTo* = −1) and maintain an

Algorithm 4 Identify Boundaries Part 2

17: **procedure** DepthFirstSearchForBoundaries(i, j, $boundaryCnt$)
18: **if** $cell_{ij}.isVisited$ **then**
19: **return**
20: **end if**
21: **if** $cell_{ij}.isObstacle$ *OR* $cell_{ij}$ *belongs to the level boundary* **then**
22: $cell_{ij}.isVisited \leftarrow true$
23: **if** *one of* $cell_{ij}$*'s neighbours is not an obstacle cell* **then**
24: $cell_{ij}.boundaryBelongTo \leftarrow boundaryCnt$
25: **if** $boundaryList.size \neq boundaryCnt$ **then**
26: Add($boundaryList$, $boundary$)
27: Add($boundary$, $cell_{ij}$)
28: **else**
29: Add($boundaryList_{boundaryCnt-1}$, $cell_{ij}$)
30: **end if**
31: DepthFirstSearchForBoundaries($i-1$, j, $boundaryCnt$)
32: DepthFirstSearchForBoundaries(i, $j+1$, $boundaryCnt$)
33: DepthFirstSearchForBoundaries($i+1$, j, $boundaryCnt$)
34: DepthFirstSearchForBoundaries(i, $j-1$, $boundaryCnt$)
35: ▷ Eliminate searches accordingly if an index is out of scope
36: **end if**
37: **if** *all of* $cell_{ij}$*'s neighbours belong to an obstacle* **then**
38: $cell_{ij}.boundaryBelongTo \leftarrow 0$
39: **end if**
40: **end if**
41: **return**
42: **end procedure**

unvisitedCellsList for them. Then we resolve the problem by propagating excited cells, passing boundary indices to neighbourhoods and removing cells that are excited/fired from *unvisitedCellsList* until there are no elements in it. As an output, each cell will hold a value of *boundaryBelongTo* from 0 to n, and no cell will contain *boundaryBelongTo* = −1.

Algorithm 5 Grassfire/Flooding Algorithm

```
1:  procedure BoundariesFlooding(C)
2:      for each cell_ij ∈ C do
3:          if cell_ij.boundaryBelongTo == −1 then
4:              Add(unvisitedCellsList, cell_ij)
5:          end if
6:      end for
7:
8:      while unvisitedCellsList.size ≠ 0 do
9:          for each boundary ∈ boundaryList do
10:             numOfCells ← boundary.size
11:             for cnt ← [1, numOfCells] do
12:                 cell ← First(boundary)
13:                 if one of cell's neighbours cell_k.boundaryBelongTo ≠ −1 then
14:                     cell_k.boundaryBelongTo ← cell.boundaryBelongTo
15:                     Add(boundary, cell_k)
16:                     Remove(unvisitedCellsList, cell_k)
17:                 end if
18:                 Remove(boundary, cell)
19:                         ▷ Expanding the current boundary and discard dead cells.
20:             end for
21:         end for
22:     end while
23: end procedure
```

After flooding, we can easily obtain an initial profile of roadmap edges: the collision between colors indicates a segment of the roadmap with the help of coloring.

Figure 4–3 shows the result of flooding and Figure 4–4 shows the basic roadmap we extract from this, as indicated by purple line segments.

According to this roadmap prototype, one is able to gain insights of strategic passages and vital crossroads, placing a guard around both of which can potentially impact the difficulty of the level. However, the discretized space we use in our construction tends to result in large numbers of small segment pieces that are not individually useful in defining patrol routes. We thus need to perform a final smoothing step to remove zig-zags and also reduce the number of nodes in the roadmap in order to generate a more usable roadmap graph. For this, we look at each roadmap edge segment separately, assembling them together into longer sequences.

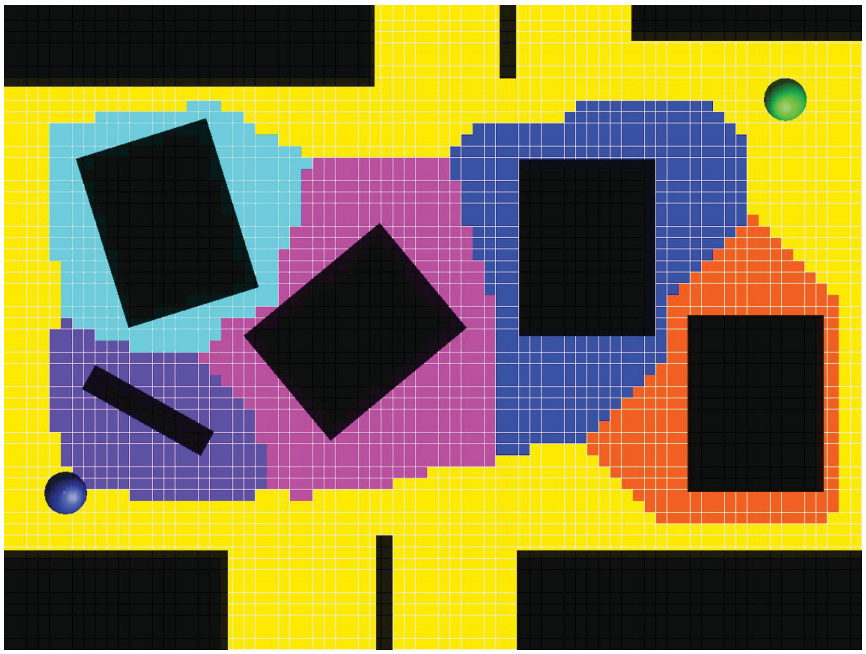


Figure 4–3: Regions generated by flooding.

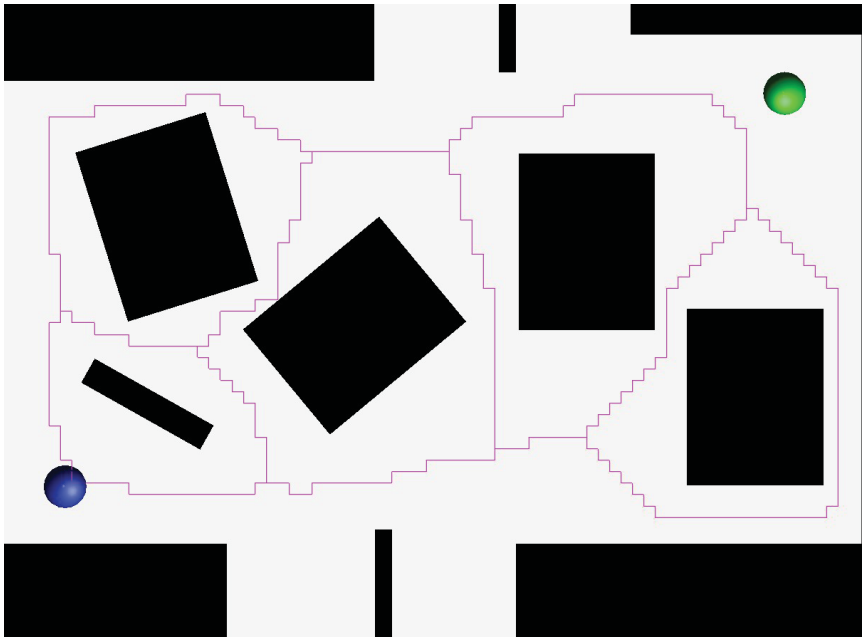


Figure 4–4: Test level with an initial skeleton roadmap.

Since the whole space is divided into grids, an edge segment, either horizontal or vertical represents a common boundary shared by two neighbouring cells. Given this fact, we represent each segment with a unique 4-tuple, indicating the coordinates of the grid cells on either side of it. A horizontal edge is denoted as $(i, j, i, j+1)$ while a vertical edge $(i, j, i+1, j)$, where (i, j) indicates the index of lower/left cell and the remaining two indicate the index of upper/right grid. The 4-tuple representation $(i, j, i, j+1)$ for a horizontal edge is illustrated in Figure 4–5.

Next, we create a new class *RoadmapNode*, representing each segment piece and its properties. We include the 4-tuple coordinates in it. Since every 4-tuple is unique, we use them as parameters for constructing a roadmap node. We also add a boolean variable to avoid duplicated visit. Most importantly, we build up parent/children

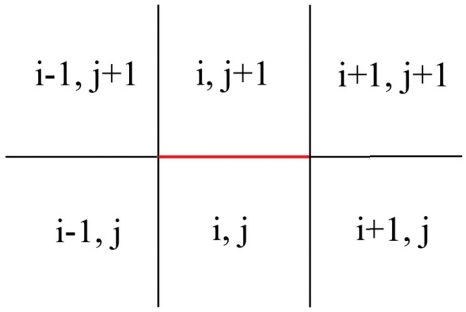


Figure 4–5: A horizontal edge (red) and its neighbours.

relations. These relations imply information in terms of connectivity—if we know the parent and children of one segment, then we know the neighbouring segments connected to the current segment. Therefore, we are able to trace the whole patrol path.

Note that in the following algorithms we use a recursively defined, custom data structure with parent/children relationship, but which can contain cycles, and so is not in general a tree. The final roadmap graph may be represented more simply in a standard graph structure, but this is a pragmatic way of representing the roadmap given our recursive construction.

In our method, segments in the level are stored into a hash map *rnDict* where all 4-tuples are set to be keys and corresponding roadmap nodes values. This gives us an easy access to whichever segment in the level. As mentioned, we then establish ascendant/descendent relations over the roadmap nodes through another depth-first search shown in Algorithm 6 and Algorithm 7. Within each recursion, we first determine if a segment is horizontal or vertical. Then we check two cells on either sides of it in order to see if they belong to different boundaries. If they do, we construct a parent/children relation to the edge in the previous recursion. In the

end, we move on to all 6 neighbouring segments (as per Figure 4–5) to deepen our search. This depth first search enables us to keep track of every specific segment and its adjacencies, and we could update the ascendant/descendant relations later on.

Algorithm 6 Initializing Roadmap Nodes

```
 1: procedure InitializingRoadmapNodes(C)
 2:     rnDict ← ConstructDict(C)
 3:     for i ← 0, imax do
 4:         for j ← 0, jmax do
 5:             if cell_ij.boundaryBelongTo == 0 then
 6:                 continue
 7:             end if
 8:             root ← new RoadmapNode (−1, −1, −1, −1)
 9:             DFSForRoadmapNodes(i, j, i + 1, j, root)
10:             DFSForRoadmapNodes(i, j, i, j + 1, root)
11:             if root.children.size ≠ 0 then
12:                 new rnList
13:                 Add(rnList, root)
14:             end if
15:         end for
16:     end for
17: end procedure
```

Our next step is to collect "super nodes," that abstract zig-zags. By zig-zags we refer to the step-like segments with an interval of one grid's width. We remove them in a recursive fashion, only keeping the remaining roadmap nodes and generating an intermediate as shown in Figure 4–6. Here, we consider the remaining roadmap nodes as candidates for *graph nodes*. Class *GraphNode* is the type corresponding to a single graph vertex after we finalize our raw roadmap to an abstract graph. By candidates we mean each graph node shares its location with a specific roadmap node that is kept. A graph node is aware of its direct neighbours for the sake of building

Algorithm 7 Extracting/Connecting Roadmap Nodes

procedure DFSForRoadmapNodes($a, b, c, d, parent$)
  if $rnDict(a,b,c,d).isVisited$ **then**
    return
  end if
  ▷ horizontal segment
  if $c - a == 0$ **AND** $d - b == 1$ **then**
    if $cell_{ab}.boundaryBelongTo \neq cell_{cd}.boundaryBelongTo \neq 0$ **then**
      $rnDict(a,b,c,d).isVisited \leftarrow true$
      $rnDict(a,b,c,d).parent \leftarrow parent$
      Add($parent.children$, $rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a+1, b, a+1, b+1, rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a-1, b, a-1, b+1, rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a-1, b+1, a, b+1, rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a, b+1, a+1, b+1, rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a-1, b, a, b+1, rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a, b, a+1, b+1, rnDict(a,b,c,d)$)
    end if
  end if
  ▷ vertical segment
  if $c - a == 1$ **AND** $d - b == 0$ **then**
    if $cell_{ab}.boundaryBelongTo \neq cell_{cd}.boundaryBelongTo \neq 0$ **then**
      $rnDict(a,b,c,d).isVisited \leftarrow true$
      $rnDict(a,b,c,d).parent \leftarrow parent$
      Add($parent.children$, $rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a, b+1, a+1, b+1, rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a, b-1, a+1, b-1, rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a, b, a, b+1, rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a, b-1, a, b, rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a+1, b, a+1, b+1, rnDict(a,b,c,d)$)
      DFSForRoadmapNodes($a+1, b-1, a+1, b, rnDict(a,b,c,d)$)
    end if
  end if
  return
end procedure

an adjacency matrix. Afterwards, we can initialize an abstract graph with those nodes and connect the graph nodes accordingly. In Figure 4–7, we turn roadmap nodes into graph nodes and they are illustrated as grey spheres. It shows an initial representation of abstract graph for the test level.

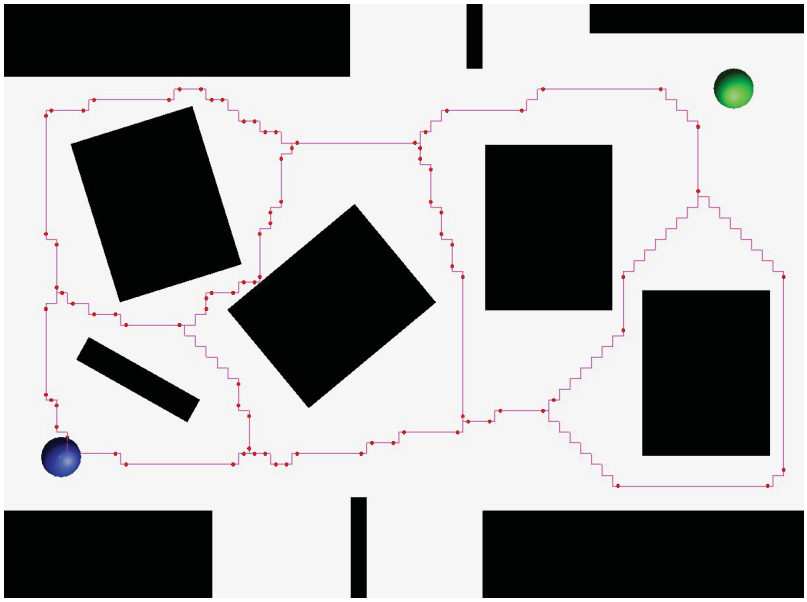


Figure 4–6: Trivial zig-zags are ignored and crucial roadmap nodes (red dots) are identified in the raw skeleton roadmap.

Lastly, we notice in Figure 4–7, that there are a lot of nodes that are clustered heavily or located almost colinear in the same walkable passage. These are considered redundant nodes. Basically, we have a final cleaup phase to remove unnecessary nodes and leave nodes that are crucial: corners, crossroads and nodes that connect two segments with a significant slope change. A cleaning/merging algorithm is shown in Algorithm 8. This cleaning/merging step, generally, can be interpreted by the three

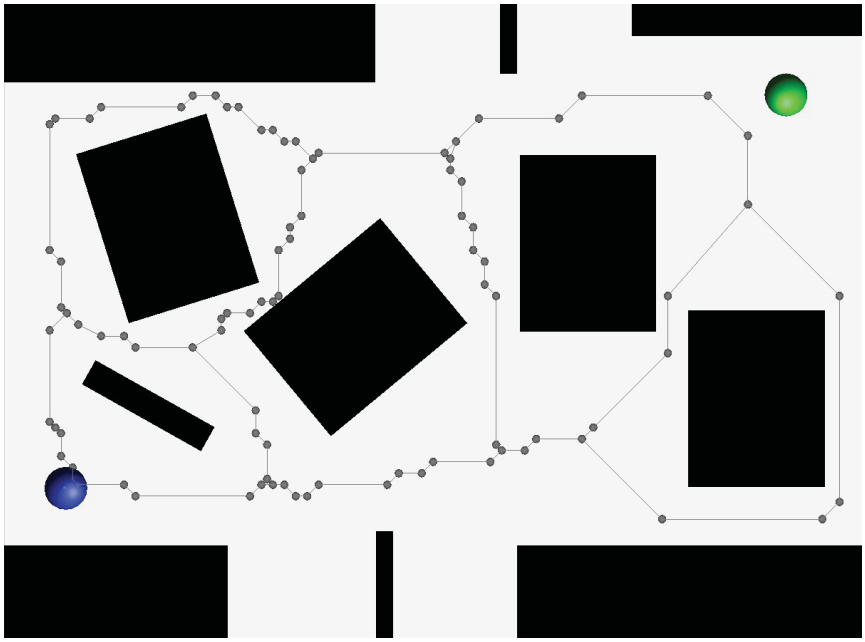


Figure 4–7: An initial representation of abstract graph for test level.

rules that govern whether we keep a node or not. Consider a chain of 3 nodes, a, b, and c. Then,

1. if a node a has a unblocked line of sight towards c across by b, then node b is considered candidate of redundancy;
2. if the node b is connected by more than 2 edges, it must be kept;
3. if obstacles exist within the triangle formed by node a, b and c, then we must keep b.

Rules above are illustrated in Figure 4–8. Solid lines indicate an original connectivity between node a and b, b and c, while dotted lines indicate the attempts to remove b and connect a to c directly, and nodes and lines that are kept after applying the merging rules are shown in red. Only in the leftmost scenario do we succeed in

deleting b; the others fail since in the middle node b has three edges and in the rightmost there is a triangle within $\triangle abc$.

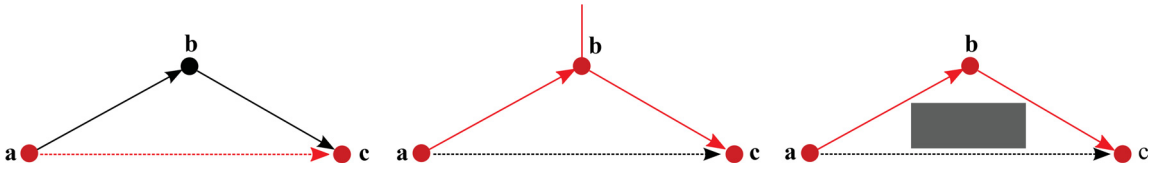
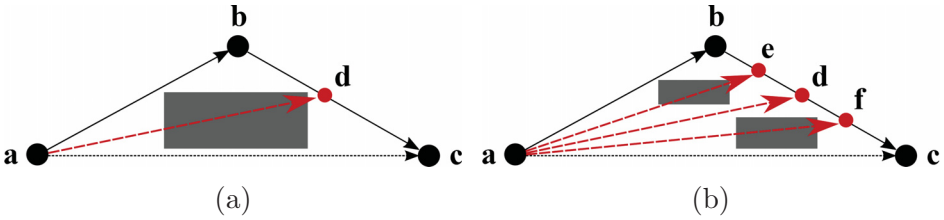


Figure 4–8: Rules to eliminate a node.

The last case of this cleanup involves a potentially complex computation to verify the inner triangle is free of any obstacle. In practice, we experimented heuristically by casting a line of sight connecting node a and the midpoint between b and c. As shown in Figure 4–9a, if the ray ad intersects with the obstacle, we consider an obstacle exists within $\triangle abc$ and thus keep b; otherwise, we connect a to c and remove b. This was effective in our case since obstacles tend to be large, and so easily identified. To achieve a more reliable result, this method can be extended to testing rays between a and each quarter point, as shown in Figure 4–9b, or an accurate result may be obtained by testing for obstacle/triangle inclusion and intersection, for every triangle and every obstacle.



(a) (b)

Figure 4–9: Experimental methods of determining whether any obstacle exists within triangle $\triangle abc$.

In Algorithm 8, we iterate through each graph node in *gnList* which contains all the existing graph nodes in the initial abstract graph. Tracing along the path, we check the descendants of node a. Once a very first node c is found satisfying the all rules above and thus being kept, all intermediate nodes are removed by setting *isRemoved* as true, and reconstructing parent/children relations. To clarify, we use the variable *seen* to indicate whether a node can be seen by its current ascendant a. Also note that raycasting provided by Unity 3D is used for the sake of determining whether the line of sight between two nodes are blocked by any obstacle.

The end result is close to a simplified straightened skeleton, but absent edges that would connect to polygon vertices, or which would otherwise normally be generated by interaction of wavefronts from different edges of the same polygon (obstacle or boundary). This approach avoids the potential for guard paths that would walk to or from corners and concentrates guard behaviour on touring around obstacles. Figure 4–10 shows the result after smoothing and node reduction.

4.3 Patrol Routes

Given a roadmap network, we select patrol routes by randomly choosing starting and ending nodes within the network (different for each guard), and constructing a relatively long path between them through a simple depth-first search. Patrol routes are assumed to be cyclic, with the final position matching the starting position: given two distinct end-points, a and b, the route consists patrols from a to b and then back to a. Note that it is possible that $a = b$, in which case the route will consist of a random tour through the network that returns to a.

Algorithm 8 Clean up Redundant Graph Nodes

```
1: procedure CleanUpNodes(gnList)
2:     for each a ∈ gnList do
3:         if !a.isRemoved then
4:             seen ← TRUE
5:             while seen do
6:                 seen ← FALSE
7:                 Copy(tmpNeighborList, a.neighbors)
8:                 for each b ∈ a.neighbors do
9:                     if b.neighbors.size == 2 then
10:                        ▷ make sure a and c do not duplicate
11:                        if a == b.neighbors_0 then
12:                            c ← b.neighbors_1
13:                        else
14:                            c ← b.neighbors_0
15:                        end if
16:                        if !Raycast(a, c) then
17:                            seen ← TRUE
18:                            Remove(tmpNeighborList, b)
19:                            Add(tmpNeighborList, c)
20:                            Remove(c.neighbors, b)
21:                            Add(c.neighbors,a)
22:                            Clear(b.neighbors)
23:                            b.isRemoved ← TRUE
24:                            break
25:                        end if
26:                    end if
27:                end for
28:            end while
29:        end if
30:    end for
31: end procedure
```

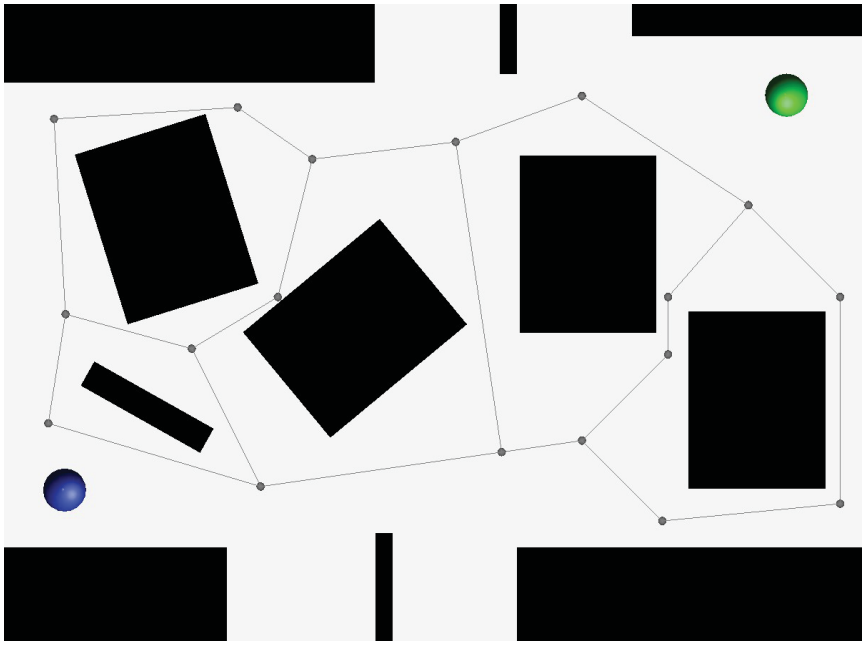


Figure 4–10: Test level with a generation of simplified roadmap.

Given a basic patrol route, we can then investigate different ways to generate intra-route activities. In the following sections, a grammar-based approach and a rhythm-based approach are introduced respectively.

4.4 Grammar-based Generation of Intra-route Activities

The basic route generation results in guard patrols that have reasonably good level coverage, but are overall somewhat robotic in behaviour. More human-like behaviours are created by breaking up the patrol movement, introducing points at which a guard stops and turns to scan the space in different fashions. For this reason we use a grammar-based construction, shown in Figure 4–11.

However many roadmap edges are traversed, a patrol route is initially considered a single segment of continuous movement ($\vdash\!\!\dashv$). The first few rules of this grammar

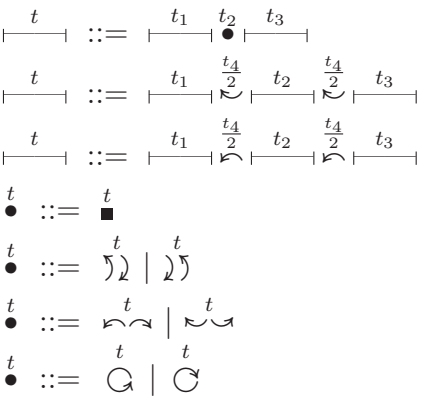


Figure 4–11: Guard patrol grammar

break up a continuous segment of movement into multiple segments: rule 1 rewrites a segment into two, not necessarily equal length segments separated by an activity (•), while rule 2 rewrites a segment into 3 sections with 180° turns between (rule 3 is symmetric, turning in the other direction), and so causes the guard to go back and forth over the same segment. Rule 4 rewrites the abstract activity designator into a pause (■), while the remaining rules rewrite an activity into different kinds of scanning: a 90° turn and return, left or right (rule 5), a 180° turn and return (rule 6), or a full 360° rotation (rule 7).

Applying these rules to the initial guard path gives a more varied and interesting patrol behaviour to a guard. Two factors can be used to control the rule application. Rewriting depth is a core concern, in the sense that overly applying these rules will densely cluster behaviours and make motion again unrealistic. We can also create different classes of behaviours by introducing different probabilities for choosing each rule. By controlling the probability distribution, one can generate an objective-oriented sequence of behaviours aiming for a specific task. For example,

a great probability of scanning 90° or pausing might simulate some *lazy* guards who tend to rest/smoke a lot. Likewise, a great probability of moving and patrolling indicates rigorous, well-trained, disciplined guards. In this case, a parametrization for probabilities of intra-route activities actually leads to a more satisfying results for a game designer.

4.5 Rhythm-based Approach for Intra-route Activities

The grammar-based approach also has potential to be less useful if the rules are applied too many times while the path/time bound is limited—each event takes time, and the whole patrol time could either grow unbounded. Alternatively, if time is capped, a guard may be required to move or rotate at unrealistic speeds. We thus also explored a "rhythm-based" approach that is capable of exploring the whole roadmap while allowing a designer more control over the timing of motions. In this approach, a designer is able to control the *frequency* of introducing extra motions and intra-route activities inbetween. This frequency, in other words, can be interpreted as the *rhythm* of patrol behaviors.

Our implementation is based on first assigning a *finite state machine* to each guard being populated, then building a game manager controlling all these finite state machines, finally triggering all the finite state machines to run and adding activities to guards by a timer. A finite state machine is conceived as an abstract machine that can be in one of a finite number of states. Here, states are represented by three agent behaviour types: moving, pausing and rotating. The machine is in only one state at a given time; by default the state is moving. A game manager will use a controller manipulating all the finite state machines: it automatically checks

each time interval t, inviting a finite state machine to switch its state at that time. By changing a state, new waypoints are generated and behaviours modified. State changes are invoked based on different probabilities in our model—there are odds for triggering all finite state machines and there are probabilities for any activity state switches. For example, every t interval, there is a probability p_i for the ith finite state machine to run, at which point it will consider probabilities p_m, p_r or p_p to change a guard's behaviour. Note that a newly introduced behaviour does not necessarily have to wait until the previous one finishes. Components and workflows for this method are shown in Figure 4–12.

This approach has advantages in not requiring a full set of intra-route activities be computed and be stored ahead of time, and will in general produce a non-deterministic set of behaviours. By adjusting probabilities and timing of this model, we are able to tweak the frequency of introducing any new agent behaviours, or even setup a deterministic schedule.

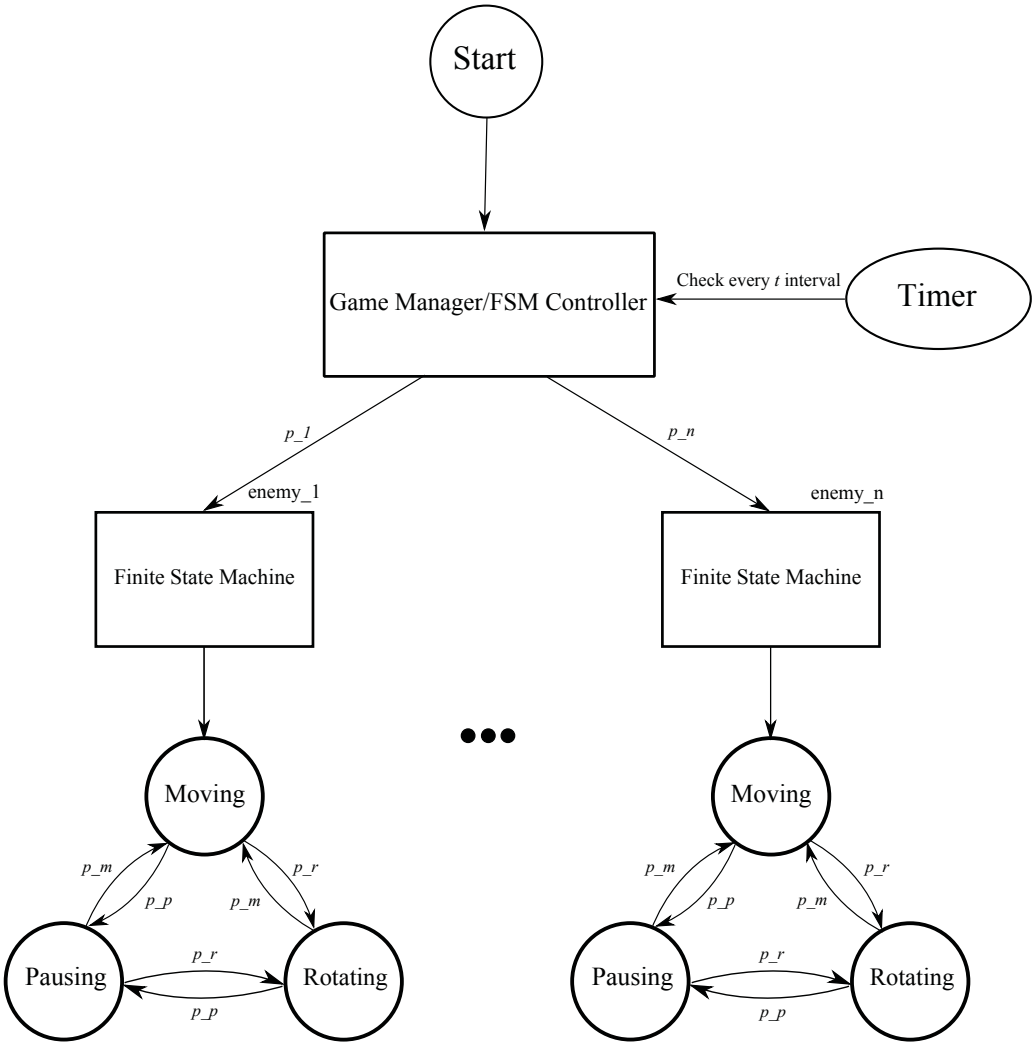


Figure 4–12: Basic components and workflow of rhythm-based model.

CHAPTER 5
Experiments and Results

Recalling from previous chapters, we divided our solutions to the problem of procedural guard placement and behaviour generation into two classes: a region-based one and a roadmap-based one. In this chapter we provide experimental results to validate the effectiveness of our approaches, and to show how choice of different strategies affects game difficulty. We measured the impact on game challenge through quantitative metrics in order to determine how we can relate varying parametrization to different measures of player difficulty. Additionally, we solidified our results with three representative test levels.

Below we start by introducing our test levels, followed by a description of our metrics, and finally we graph the results obtained from two techniques, first the region-based approach and then the roadmap-based approach. Note that there are also two alternatives for the roadmap-based approach: one based on grammatical rules, and one based on rhythms. We thus also made quantitative comparisons among these approaches.

5.1 Test Levels

In order to validate and compare our approaches, we examined the effects of camera and guard placement on multiple game-levels as follows:

- *Test Level 1*: containing few large obstacles, varying in size and alignment; a roadmap-based version of it is shown in Figure 4–10;

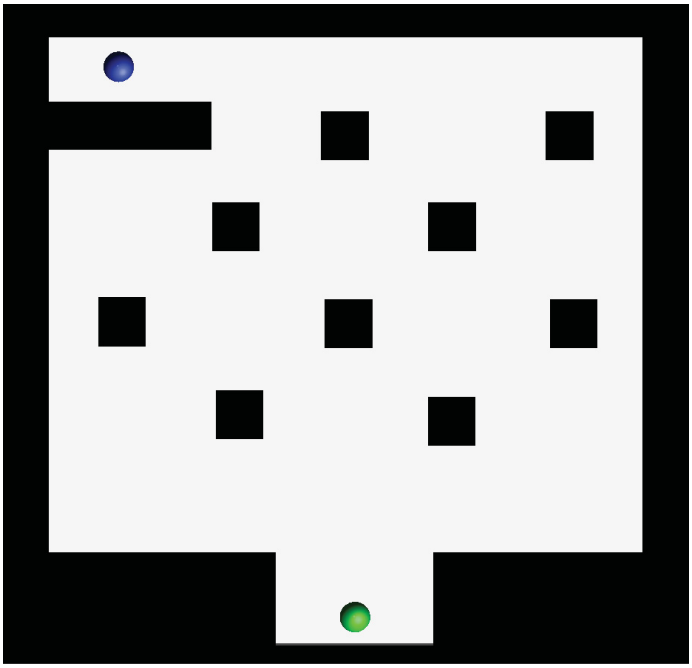


Figure 5–1: *Test Level 2* with orthogonal obstacles and latticed passages.

- *Test Level 2*: consisting of many small obstacles, equally-sized, and well-aligned; the level is depicted in Figure 5–1;
- *Test Level 3*: a mock-up of the "Dock" level from the *Metal Gear Solid* (MGS) game, shown in Figure 5–2.

The first two test levels we chose are representatives of two typical types of game level containing FPS and stealth content and the third one is based on a real game:

- *Test Level 1*: indoor maze with narrow tunnels to explore, little space for travelling but sufficient room to shelter;
- *Test Level 2*: outdoor battlefield with large area to move, multiple sources of cover but inadequate place to hide;
- *Test Level 3*: an actual in-game level from a very popular stealth game.

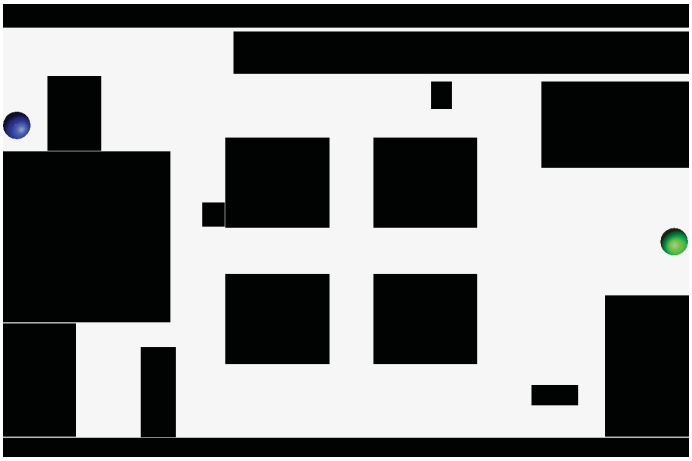


Figure 5–2: A mock-up of the first level from Metal Gear Solid (MGS), the Dock.

5.2 Metrics

To measure the difficulty of our stealth levels, we make use of two different metrics, one that measures relative feasibility, and another that measures the perception of risk.

We determine the degree of feasibility by computing the success rate of a heuristic path exploration technique. Path-finding techniques are able to solve stealth levels, at least with deterministic enemy behaviours, by modeling the game state as a geometric space extruded over time. A path that reaches the goal position, while continually advancing in time and avoiding all enemy FoV at each point in time is then a solution to the stealth level [21]. We compute such paths using a Rapidly Exploring Random Tree (RRT) search [12] rather than a more traditional A* or other deterministic approaches, since the RRT paths may fail, tend to have a significant, sub-optimal randomness to the result, and so better approximate a variety of human

gameplay behaviours. We thus use the *success ratio* of RRT searches as a proxy for overall level difficulty.

Other metrics have also been proposed and evaluated for measuring properties of stealth levels. Tremblay *et al.* describe 3 different path quality metrics that attempt to measure the player sense of risk or danger [22]. Through comparison with a results from a human study they argue that a conceptually simple path-distance measure, considering the relative distance from a player to enemies at each point best approximates human perception. We thus also apply this metric in order to determine the degree of risk our levels present to players. Note that this metric is inverted, and higher values in the distance metric indicate riskier paths that tend to be closer to enemy agents.

Both metrics are applied to game levels modeled in a non-trivial Unity3D game development framework, and so consider realistic player movements, including physical and visual constraints typical of modern first or third-person 3D environments.

5.3 Region-based Camera and Guard Placement Analysis

Previous work presented a basic analysis of camera, which offers clear and easy flexibility in difficulty adjustment by modifying the properties of a single camera, namely, the FoV angle and the FoV distance [23]. An understanding of FoV can be further applied to guards–the mobile version of cameras. We used a FoV angle of 33° and a FoV distance of 5 unit length in a 75×75 units floor-plan as they turned out to be appropriate choices in our experiments—we were able to avoid excessive and unrealistic vision overlaps, as well as to be sure that both hard and easy solution end-points exist.

Knowing how to optimize the properties of a single camera/guard does not suffice of course. We also need to understand how the choice of different numbers and/or types of observer agents affects game difficulty. That is, we measured difficulty for different distributions of entities. In order to quantitatively measure the level difficulty, we thus looked at the *success ratio* of RRT searches mentioned above. Since the solver uses a random sampling approach to find a search, this gives us a sense of the likelihood of finding a path in a level. Note that we attempt to sample 25,000 nodes before we finish one RRT search. This effectively increases reliability and accuracy of our results for each RRT search [21].

Here, we used *Test Level 1* and *Test Level 2* for analysis (*Test Level 3* does not involve cameras). For each scene, we ran 50 trials where each trial consisted of 100 agents sent to find a path from *start* to *goal* within one uniquely generated distribution of entities. The entities are either rotational cameras or moving guards, whose amount grows from 0 to 8 in *Test Level 1* and from 0 to 10 in *Test Level 2*. The relative number of agents that succeed then represents the probability of finding a path. Averages and errors bars from the 50 trials for the two test levels are plotted in Figure 5–3 and Figure 5–4 respectively. We are interested in analysing how difficulty scales with the number of entities, as well as whether the choice of different entity types has any impact.

Clearly, the number of moving guards and cameras does influence the ratio of successful paths found—the more entities are introduced into the level, the more difficult it is to find a successful path. This change is not necessarily linear, however, and despite the high variance in this study there is a much more noticeable increase

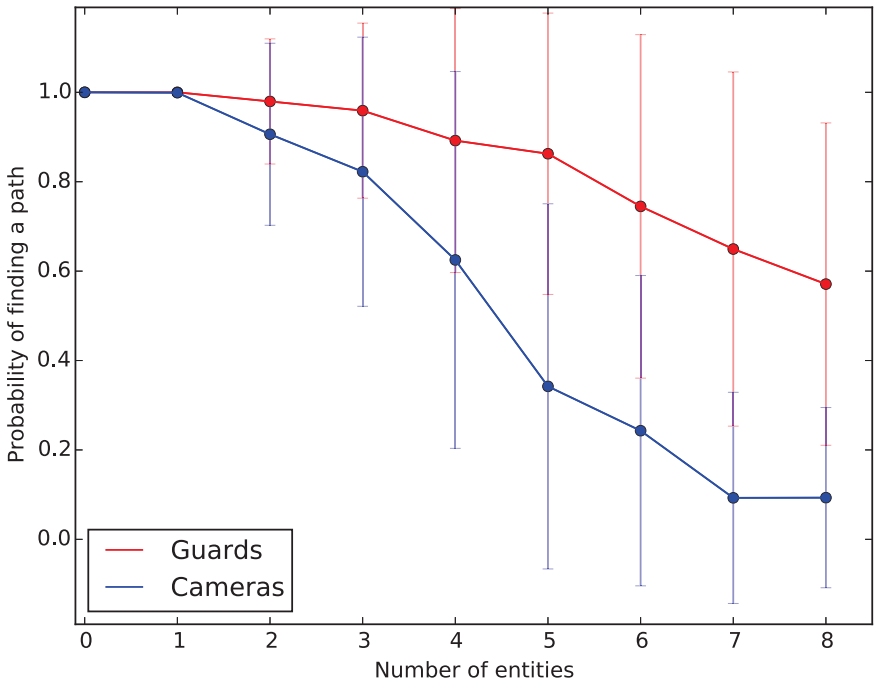


Figure 5–3: Probability of finding a path as the number of entities increases for *Test Level 1*.

in difficulty as the number of entities grows. In addition, the slopes in *Test Level 1* are steeper than those in *Test Level 2*, and we see that the probability of finding a path in *Test Level 1* decreases faster. This is due to the fact that *Test Level 2* provides more possible passages from start to goal, which is less likely being blocked by few entities. Another perception is that a significant decline of probability can be observed between 3 and 5 cameras in *Test Level 1*, whereas the decline is more gradual in *Test Level 2*. This is likely correlated with the level geometry. In *Test Level 1*, an even distribution of 3 to 5 cameras tends to either place cameras near the start or goal positions, or more effectively block the paths between start and goal, causing a drastic decline of success ratio. *Test Level 2*, however, is capable of containing more cameras without pushing the probability to 0 so that the slope changes are not obvious in the graph.

More striking in our results is that Figure 5–3 shows a strong separation in difficulty between cameras and guards at the same number of entities. This can be understood by considering the different behaviours from the two types of entity. Given the same FoV, a patrolling guard will cover a larger area than a sweeping camera, and generally spend less time observing any given spot. Use of mobile guards thus allows players to more easily find opportune times to bypass the guard unseen. Meanwhile, the level geometry becomes a contributing factor for these results again since the separation is not as obvious in Figure 5–4: *Test Level 2* gives more freedom to a player even with cameras since the combination of abundant geometric path choices and many obstacles mean cameras cannot block off areas all the time

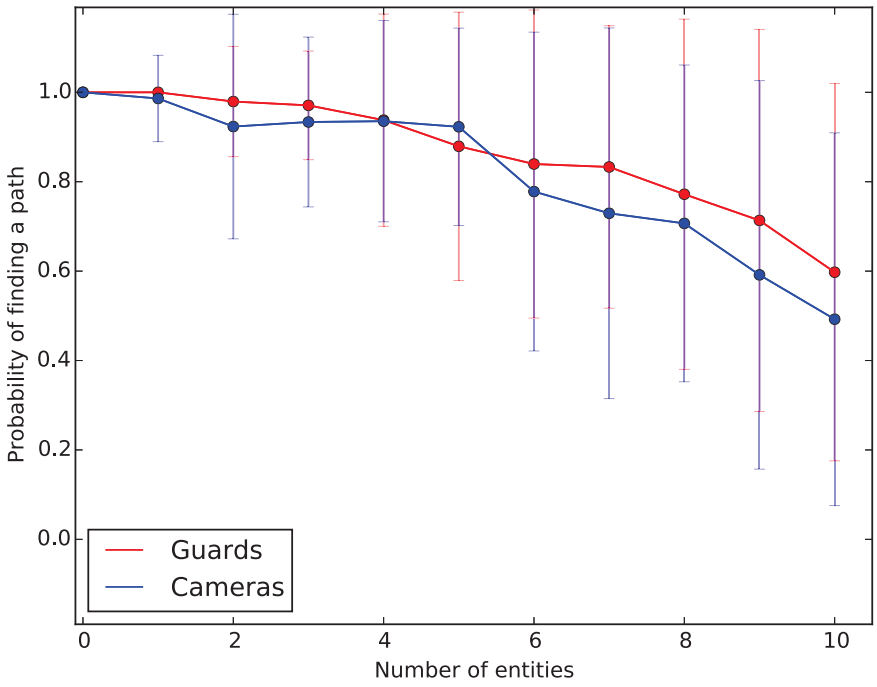


Figure 5–4: Probability of finding a path as the number of entities increases for *Test Level 2*.

any more than mobile guards. This preliminary experiment shows that cameras and guards are meaningfully different and are useful for different design purpose.

5.4 Roadmap-based Guard Placement Analysis

In this section, we run several experiments and have an in-depth analysis on the problem of roadmap-based guard placement. In the previous section, region-based camera and guard placement has demonstrated influence led by different agent numbers and agent types. Yet mobile, human-like guards moving along a roadmap provide more player interest and increase the potential design space. As such, the parameter space is too large to consider exhaustively, and so we analyse different features separately.

We begin by exploring the impact of number of guards on level quality. For this we first use *Test Level 1* shown in Figure 4–10. After computing the simplified roadmap we assign guards different, random patrol routes, but without any intra-route activities. Figure 5–5 shows the results for both the RRT success rate and the path-distance metrics. For each number of guards, we applied 20 RRT searches to each of 30 unique choices of guard paths, giving us an RRT rate /20 and distance metric values on up to $20 \times 30 = 600$ different situations (we did not include failed solutions in the distance metric). Following this, a similar experiment was also carried out for *Test Level 2* and the results are plotted in Figure 5–6. Note that the data we gathered from benchmarking the time of executing RRT searches indicates that it took 6.5 hours (4 hours longer than computing success ratio only) on average to finish a single batch for each number of guards (ranging from 1 to 8)—computing the distance metric is much more expensive than computing the success ratio. This

motivated the reduced batch size, shrinking to 600 (20×30) from 5000 (50×100) in the last section.

Despite the fact that slopes in *Test Level 1* are much steeper than those in *Test Level 2*, again likely due to the difference in level geometry, both test levels show consistent trends in success ratio and distance metric—a decrease for the former and an increase for the latter. Also shown in both figures, the randomized path selection results in a large variance. As expected, though, increasing the number of guards in an otherwise fixed level increases the relative coverage of space, and so makes it harder to find a path that traverses the level unseen from start to finish.

Unlike cameras, however, the cut-off for infeasibility is not nearly as sharp: a moving guard may retard player progress, but by virtue of moving rarely tends to permanently block off an entire corridor, and even with 8 guards (which is quite dense) the RRT is still easily able to find solutions. In this sense the greater slope of the distance metric is perhaps a better representation of game difficulty for players—with more guards, player paths necessarily come closer to guards, and thus closer to being seen.

5.5 Grammar Influence

The choice of and amount of application of grammar rules potentially affects the level as well. Experimentally, however, we found that even fairly large application of the grammar rules does not have a significant impact on RRT success—whether solutions exist is driven primarily by the degree of coverage induced by the number of guards and their paths, and while interrupting a guard path to scan around and

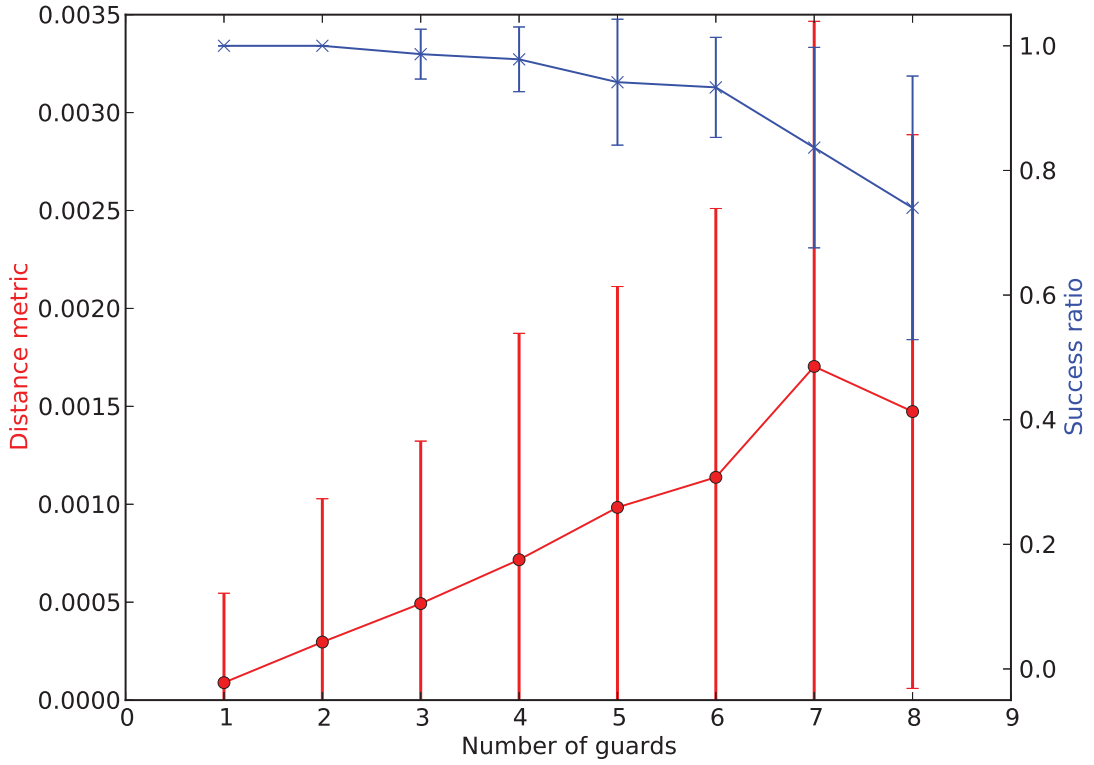


Figure 5–5: Success ratio (×'s, blue) and distance metric (∘'s, red) *vs.* number of guards in *Test Level 1*.

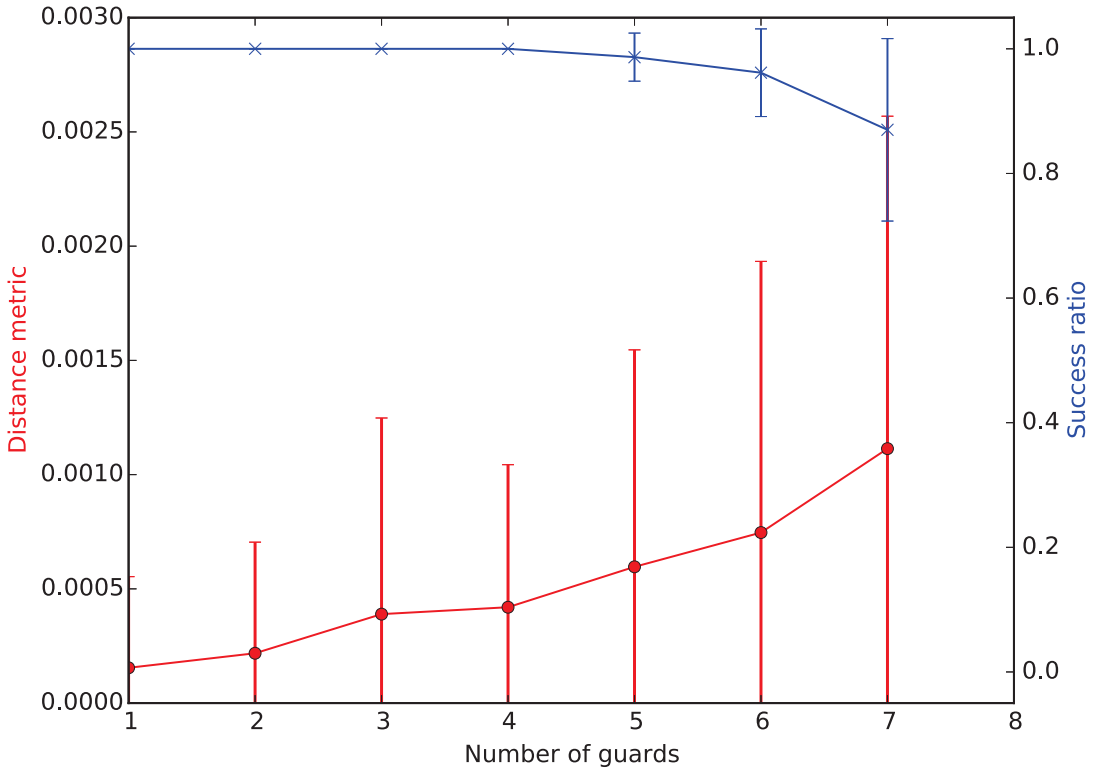


Figure 5–6: Success ratio (×'s, blue) and distance metric (∘'s, red) *vs.* number of guards in *Test Level 2*.

so forth is visually interesting, it has limited impact on whether a level is actually solvable or not.

This does not mean, however, that the grammar has no useful effect on solution paths. As guards spend more time scanning around, they temporarily block possible player paths, and/or require players skirt around them. We thus expect that by increasing and changing guard activities we can influence a player's experience in terms of perceived risk in terms of the amount of time a player spends in close proximity to guards, and this is reflected in the distance metric.

Figure 5–7 shows the distance metric calculations for different kinds and amounts of rewriting, for 4 guards on *Test Level 1*. We apply our grammar rules to introduce each of the 3 kinds of rotations, the zigzag repetition of a segment (rules 2 and 3 in figure 4–11), and the rewriting rule of pausing. For each of these kinds of rewriting, we considered rewriting to different recursive depths, where higher depth means more intra-route activities.

The result of this experiment shows a high degree of variance, but nevertheless separates behaviours into 3 main categories of statistical significance ($p < 0.05$). The distance metric value mainly increases due to use of 180° rotations and the zigzag movement, while others have a lesser impact, and all are different from a baseline of no rewriting. We attribute the stronger influence of 180° rotations and zigzag movements to the way these rules alter the ability of a player to follow a guard: if a guard rotates 90° or does a full 360°, it is relatively easy to follow a guard from a distance or get past them as their point of view rotates. With 180° rotations and zigzagging, however, guards end up looking backwards for some time, and a solution

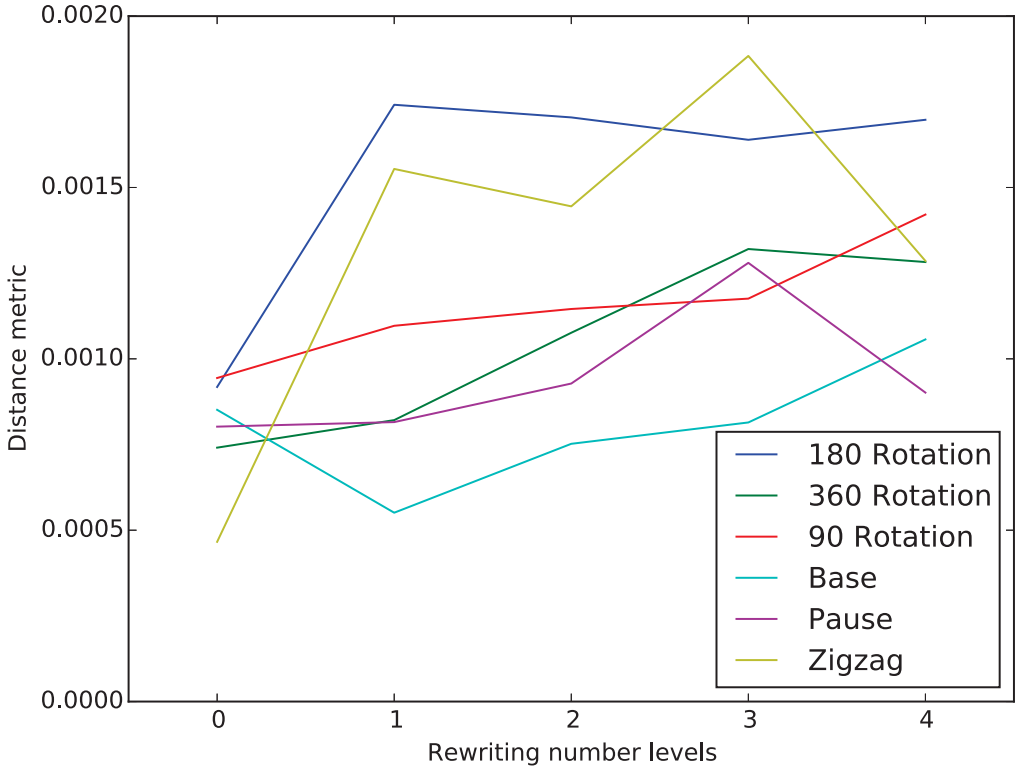


Figure 5–7: Average distance metric values on the *Test Level 1* with different kinds of rule choice and depth of rewriting.

path nearby will need to follow the guard movements closely for longer in order to avoid detection.

Test Level 2, on the other hand, introduces other interesting results. As shown in Figure 5–8, despite the high variance, the lines for different behaviours can also be divided into at least three classes. 1) The average values of distance metric for 90°, 180° and 360° rotations are larger than others. For each, there is a noticeable value increase just by a rewriting level of 1, and the value, generally, becomes stabilized

afterwards. 2) The yellow plot of rewriting into zig-zags lies between rotations and pausing/base; it gradually increases from a rewriting depth of 0 to 2 and becomes stable in future rewritings. 3) Pausing has the least impact on distance metric, which locates slightly above the baseline and barely changes as the rewriting depth goes deep. We strongly consider the reason behind this graph is related to the level geometry. Given *Test Level 2* where a player has ample passages to select from start to goal and each passage is kind of wide, a guard moving in zig-zag fashion does not block the player's pathfinding effectively. Thus, it has less influence than rotations. Also, the level difficulty influenced by depth of rewriting gradually becomes stable and even declining since rewriting with rotation behaviours may cost the guards moving less, which in turn leaves more space open for player passing through in this scenario. Like the noticeable drop for 90° rotation, we can actually predict similar drops for 180° and 360° in further rewritingsl; indeed, the figure shows the start of a declines from rewriting depth of 4 for 180° and 360°.

5.6 MGS Comparison

In this experiment we show that our grammar-based approach can achieve a similar distribution of guard behaviours to a manual design in a commercial game. For this we examine the first level of the Metal Gear Solid, as shown in Figure 5–9. This level has just 2 guards, and a fairly stable 100% success rate under RRT search. Of course this would be easy to make more difficult by adding more guards, but this relatively easy level design is also mitigated by varied guard behaviour that provides interest and perceptual challenge to the player as their first experience in the game.

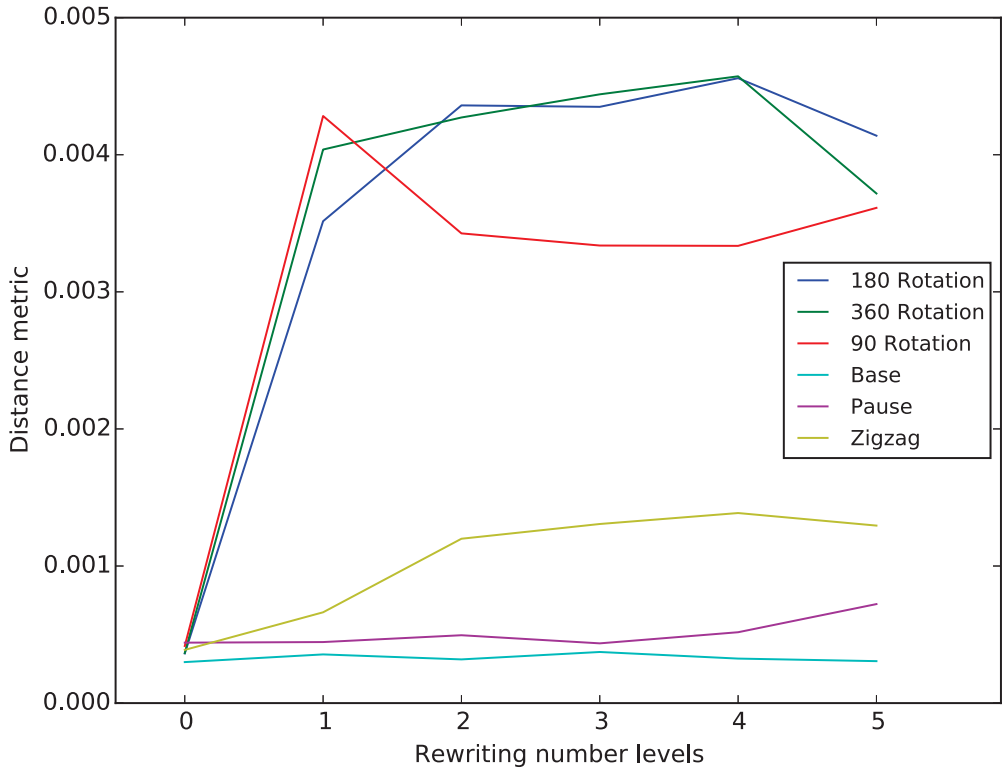


Figure 5–8: Average distance metric values on the *Test Level 2* with different kinds of rule choice and depth of rewriting.

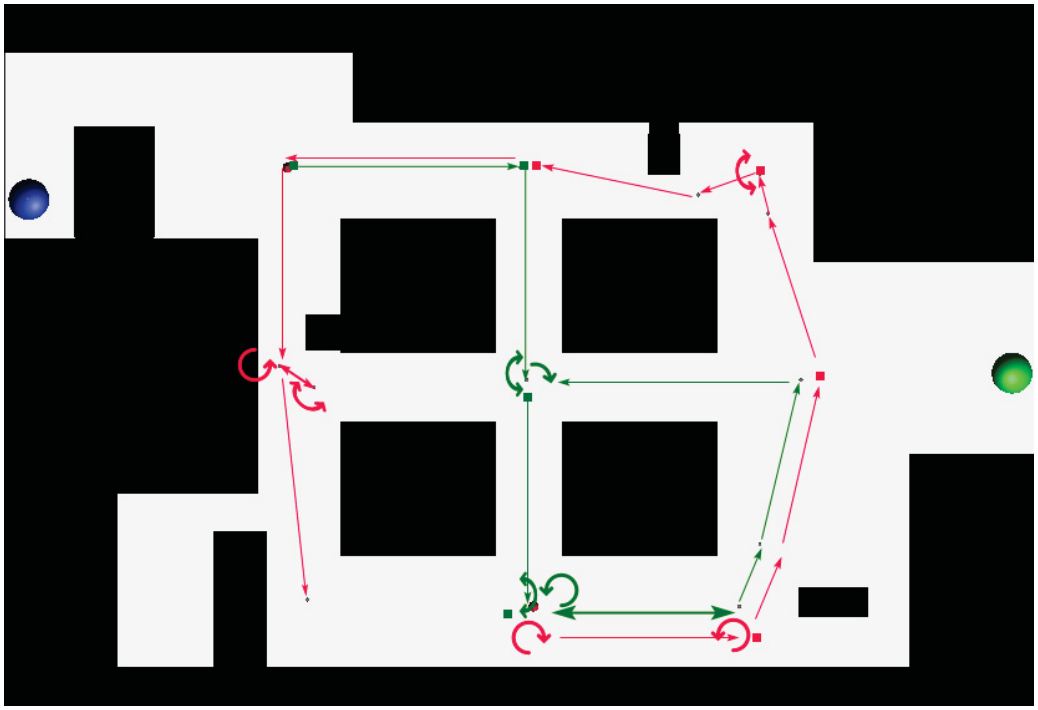


Figure 5–9: The “dock” level from Metal Gear Solid, with two guards route overlay produced by the grammar.

In Figure 5–10 we analyze the original guard movements and show metric data for the distance metric based on 500 successful paths found by RRT searches. Overlaid with this we show the distributions of the same metric data produced by applying our grammar-based approach to the same basic guard routes. We used a rewrite depth of 2, and biased our rule selection to favour zigzag rewrites over segment splits, and pauses over rotations. This produced the 2 paths presented in Figure 5–9, The red guard patrols the outside most of the level, whereas the blue guard follows a more complex route within the level. Double-sided arrows represent zigzag behaviour, and the rest follows the notation presented in Figure 4–11. Given the randomization inherent in our grammar application and RRT choices the distributions do not match perfectly, but this data does suggest we can algorithmically produce a desired style of guard behaviours.

5.7 Rhythm Experiments

In this section, we try out a different experiment to indicate the flexibility that our rhythm-based approach can bring to our game design and its influence on level difficulty. We investigate the relationship between the frequency that we introduce a new behaviour (moving/pausing/rotating) to a guard and the level difficulty reflected by both quantitative measurements—success ratio and distance metric. In order to mimic a game level that allows us to effectively utilize the whole level space and gives a player enough time to explore, we set the number of guards in *Test Level 1* to be 4 and the total time length that a guard can travel to be 1200 time units. We then gradually decrease the time interval that triggers a new random behaviour in the finite state machine from 600 time units to 50 time units. To clarify, the frequency

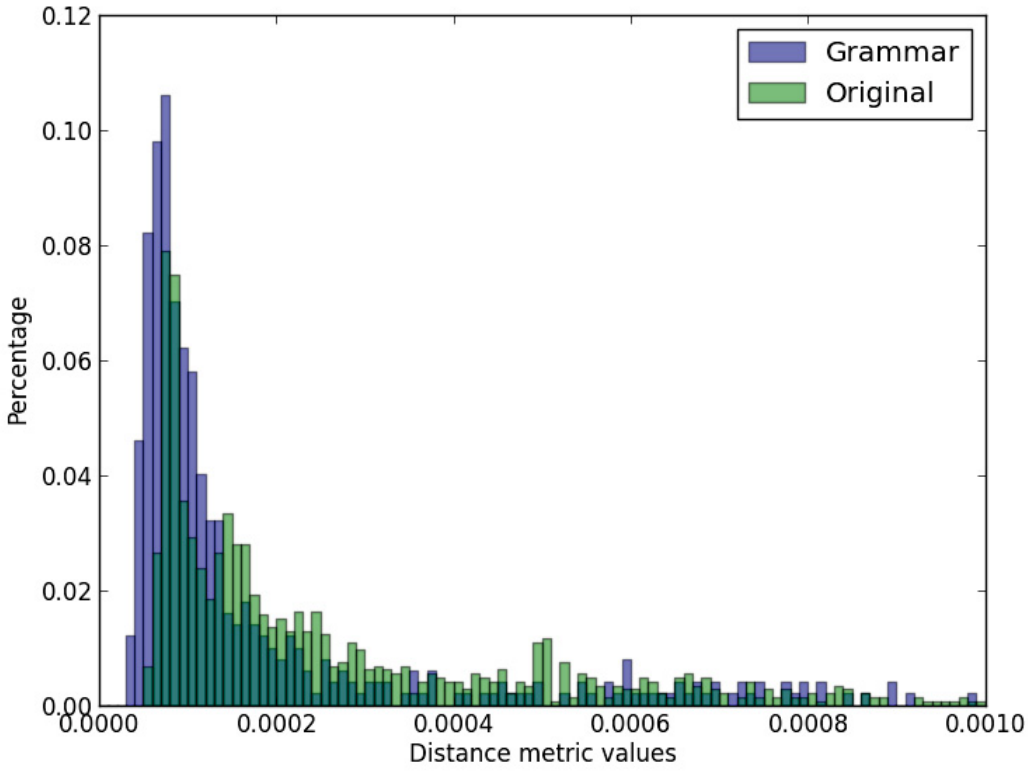


Figure 5–10: Histogram of distance metric values for MGS level.

changes from 0 to 6 following the rule: $frequence = \lfloor timeLength/(4 \times timeInterval \rfloor)$. For each frequency, we applied 20 RRT searches to each of 30 unique choices of guard paths, also giving us up to $20 \times 30 = 600$ different situations (and again, we did not include failed solutions in the distance metric).

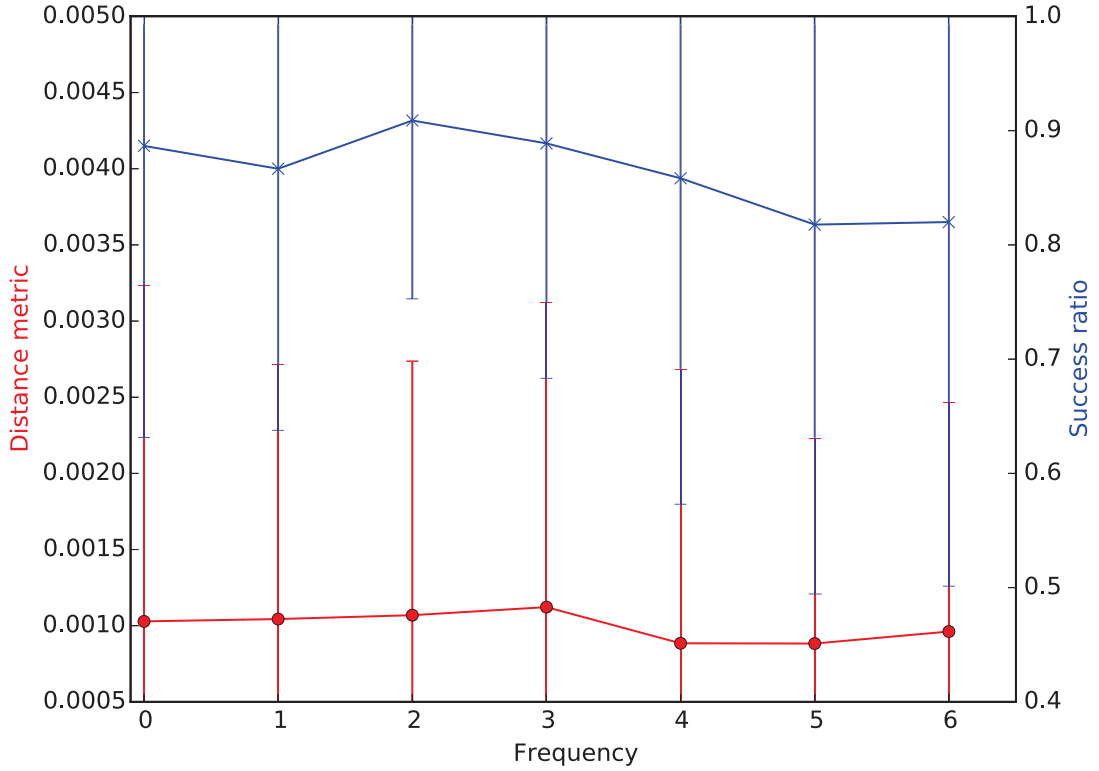


Figure 5–11: Average distance metric values and success ratio on the *Test Level 1* with respect to an increasing frequency, using the rhythm-based approach.

As Figure 5–11 shows, the value changes for both measurements due to how often we switch among behaviours are limited. This means that increasing the frequency of introducing new behaviours is not a substantial factor that can impact the

game difficulty since we are uncertain about which type of behaviour is generated. According to previous results, a pausing or rotating behaviour can actually increase level difficulty while a moving behaviour reverses the result. Therefore, even though we trigger the finite state machine more frequently, if the newly generated behaviours are mostly moving, the game difficulty is unlikely to increase significantly.

The rhythm-based approach thus does not seem to have as controllable an impact on difficulty. Unlike its region-based and roadmap-based counterpart, however, this relative immunity to trends in difficulty gives the rhythm-based approach an easy way to add complexity of interest to a design level, without changing the level difficulty too much. This design may thus still be worth exploring in further work, trying to more formally show that by tweaking the combination of different types of behaviours, such as the probability of the appearance of each behaviour and the time interval of introducing new behaviours, a human designer/tester would be able to easily build a level prototype with high interest. Of course for this we need a quantitative way of measuring interest.

CHAPTER 6
Conclusions and Future Work

Stealth games or levels are popular with players for posing interesting, puzzle-like challenges different from the typical combat-intensive scenarios found in many modern games. A creative puzzle challenge, however, requires a creative puzzle design, ensuring solutions exist, but having appealing and appropriate complexity. Such design often involves careful orchestration of enemy positions as well as behaviours, which can be related to the problem of difficulty adjustment. Normally, game industry uses manual level constructing and iterative playtesting to make a stealth game desirable. The techniques we propose for procedural generation help with the design challenge by providing distinct mechanisms for scaling difficulty in camera placement, generating guard routes and improving player experience through variety in guard behaviours. More importantly, the techniques of generating entities and evaluating difficulty are highly automated, allowing us to alleviate the burden from human testing and to reduce the amount of time for testing iterations.

Understanding how guard/camera placement and behaviours affects level design is an important step towards full procedural generation of stealth problems in games. Our region-based guard/camera placement introduces a simple method that generates straight paths for patrolling guards and static locations for rotational cameras. We first decompose a level-space into separate Voronoi regions, and then use region geometry to define appropriate observer locations. This provides a fast way

of covering the level-space with reasonable guard/camera placement—heuristically, larger, main areas will be under surveillance, while less open spaces remain for players to sneak through. Using our techniques, we can already generate levels with parametrized difficulty. We discover that numbers of enemies and type of enemy agents are two important factors. Specifically, in this region-based method, as the number of enemy entities grows, the level becomes more difficult to solve; also unlike a guard that potentially allows a player to sneakily follow, a camera plays a more strict role in inspecting a certain area—it does not leave as much area unobserved for as long. Overall, we are especially pleased that despite the strong constraints we rely on in this work, the generated levels actually look quite convincing, suggesting the approach can be both practical and effective.

Our roadmap-based guard placement approach, on the other hand, generates more complex paths and behaviours for mobile guards. Starting from the basic grassfire algorithm, we can generate a straightened and reduced version of the medial skeleton to serve as a roadmap. These roadmaps turn out to be robust and reasonable in that we obtain satisfying results on several game levels. We then extend this method by formulating certain grammar rules, rewriting a sequence of behaviours into certain amount of depths—the more we rewrite, the more intra-route activities are introduced. Although we find that applying grammatical rules does not have as significant impact as the number of guards does, it is still interesting that we manage to separate different behaviours by statistical significance, and behaviours like 180° rotations and zigzag movements can be seen to have a stronger influence on game

difficulty than others, depending on level geometry. Finally, we consider a rhythm-based approach for intra-route activities. We build a finite-state machine to control the frequency of introducing new behaviours and the variety of behaviour types, and thus the rhythm of intra-route activities of guards. This provides an alternative way of generating guard activities, that while highly variable, does not impact difficulty as much, and so is useful for different purposes.

Although facilitating the game production significantly, our approach would never take over the whole process. Procedural placement has its limitations—it lacks an understanding of detailed design goals, and it can easily generate agents that move irrationally, and in general requires manual validation and tuning to avoid such flaws. An ideal way to use our work is to use it to first populate agents, controlling level difficulty through our automated approaches, followed by manual adjustments at a micro level. Human effort and procedural placement thus complement each other in order to ensure a good game result.

6.1 Future Work

A simple variation in our approach we hope to investigate as immediate future work is to look at, and better control for the impact of patrol time on our generated behaviours. Recalling Chapter 4, our grammatical rules offer expressiveness to game entities, introducing an interesting population of guard behaviours, but each rewrite potentially increases the amount of time a guard spends on its route, and so grows unbounded if too rules are applied too many times. A simple improvement is to impose equal time bounds to each side of a rewriting rule in our grammar. This has the additional complexity of needing to select a good mix of time durations

for each rewritten component, and may result in guards moving at different speeds throughout the patrol, which is not necessarily realistic in all situations, but has the benefit of ensuring that total patrol time is always fixed, and thus the cyclic patrol pattern more easily made clear to players.

There are many other facets of stealth game behaviours we have not covered, and which we hope can be explored in more distant future work. Further, more dynamic properties of enemy behaviours (speed, non-continuous observation) would give us a richer game model, as would including the full gamut of stealth-related game features, such as sound and footprints. Going beyond pure stealth games, we are also interested in extending the work to account for combat scenarios, defining scenarios wherein players may be required to use combat of various forms in order to solve the level design. This latter approach would let us extend the generation idea into other game genres that use stealth as one mechanism among many rather than as the only form of gameplay.

References

[1] Aaron William Bauer, Seth Cooper, and Zoran Popovic. Automated redesign of local playspace properties. In *FDG'13: Proceedings of the 8th International Conference on Foundations of Digital Games*, pages 190–197, 2013.

[2] Harry Blum. A transformation for extracting new descriptors of shape. In *Models for the perception of speech and visual form*, pages 362–380. MIT Press, 1967.

[3] Daniel Cohen-Or, Yiorgos L. Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, July 2003.

[4] Joris Dormans. Adventures in level design: Generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1:1–1:8, 2010.

[5] Uğur Murat Erdem and Stan Sclaroff. Automated camera layout to satisfy task-specific and floor plan-specific coverage requirements. *Computer Vision and Image Understanding*, 103(3):156 – 169, 2006.

[6] Polina Golland, W Eric, and L Grimson. Fixed topology skeletons. In *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, volume 1, pages 10–17. IEEE, 2000.

[7] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[8] Darius Kazemi. Spelunky's procedural space. `http://tinysubversions.com/2009/09/spelunkys-procedural-space/index.html`, Sept 2009.

[9] Ben Kybartas and Clark Verbrugge. Analysis of ReGEN as a graph-rewriting system for quest generation. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2):228–242, June 2014.

[10] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[11] Steven M. Lavalle and Jr. James J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2000.

[12] S. Morgan and M.S. Branicky. Sampling-based planning for discrete spaces. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 2, pages 1938–1945, 2004.

[13] Joseph O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.

[14] Michael Ian Shamos. *Computational geometry*. PhD thesis, Yale University, 1978.

[15] Yinxuan Shi and Roger Crawfis. Optimal cover placement against static enemy positions. In *Proceedings of the 8th International Conference on Foundations of Digital Games*, pages 109–116, 2013.

[16] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. Rhythm-based level generation for 2D platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 175–182, 2009.

[17] Randy Smith. Level-building for stealth gameplay. Presentation at the Game Developers Conference. `http://www.roningamedeveloper.com/Materials/RandySmith_GDC_2006.ppt`, 2006.

[18] Gearbox Software and Feral Interactive. *Borderlands (XBox 360)*. 2K Games, 2009.

[19] Bethesda Game Studios. *The Elder Scrolls V: Skyrim*. Bethesda Softworks, 2011.

[20] Julian Togelius, Noor Shaker, and Joris Dormans. Grammars and L-systems with applications to vegetation and levels. In Noor Shaker, Julian Togelius, and Mark J. Nelson, editors, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2014.

[21] Jonathan Tremblay, Pedro Andrade Torres, Nir Rikovitch, and Clark Verbrugge. An exploration tool for predicting stealthy behaviour. In *Proceedings of the 2013 AIIDE Workshop on Artificial Intelligence in the Game Design Process, IDP*, volume 2013, 2013.

[22] Jonathan Tremblay, Pedro Andrade Torres, and Clark Verbrugge. Measuring risk in stealth games. In *FDG'14: Proceedings of the 9th International Conference on Foundations of Digital Games*, April 2014.

[23] Qihan Xu, Jonathan Tremblay, and Clark Verbrugge. Generative methods for guard and camera placement in stealth games. In *Proceedings of the Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2014, October 3-7, 2014, North Carolina State University, Raleigh, NC, USA*, 2014.

[24] Derek Yu. Spelunky. (Game) `http://www.spelunkyworld.com/`, 2009.