COMPUTING TECHNIQUES FOR GAME DESIGN

by

Jonathan Tremblay

School of Computer Science
McGill University, Montréal, Québec, Canada

April 12, 2016

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
DOCTOR OF PHILOSOPHY

Copyright © 2016 by Jonathan Tremblay
Abstract

Game design is the art of crafting interactive content. Game designers build games which are composed mainly of rules describing how its content reacts to the player’s interactions. Designers, in this context, are interested in molding play, the synergy between players and rules. During the creation process designers seek information about this synergy in order to validate their design. Commonly, human testers are required to play the design in order to offer meaningful feedback to the developers.

This dissertation attempts to introduce computing techniques in the game design process, where we seek to design artificial players that can interact with a game developer’s design. These artificial players collect meaningful information about the design, and how they interact with the game provides the game developer with valuable insights. The introduction of mechanisation has potential for reducing game developers’ intellectual reasoning, testing time, development costs, etc.

This thesis discourse presents first a form of game testing where artificial players are described as reactive. This approach consists of letting artificial players use the in-game controls to interact with the design. This approach allows for a fine tuned game testing, where instead of a game abstraction, direct controls are given to the artificial player. In this context, this approach was used to evaluate different non player character behaviours that support the player.

In order to accelerate the collection of interactive data, we explored translating games and their content into formal representations and use search algorithms for interaction analysis. Formal representations consist of describing games as a state space which allows formal modelling and thinking. Using search algorithms we can then explore the state
space to collect meaningful information about the game, possible solutions and attempts in the context of a game level. In greater detail this thesis presents a general game state space representation that takes advantage of time-space constructions. We present three different game genre representations, platformers as they are a popular and easy to design, stealth games for their interesting time-space representation, and combat games to represent the most popular game genre, first person shooters. To these we apply different search algorithms, such as A*, Rapidly-exploring Random Tree, and Monte Carlo Tree Search and show how their existing qualities in terms of search optimization algorithms can be tailored to more general game design purposes. Moreover, we also investigate how computing techniques can be used in order to measure different quantitative values about the game experience, such as the level difficulty.

As a whole this dissertation offers general guidelines and observations to include computing techniques in the design process. These guidelines offer an algorithmic approach for describing the game state space and how to explore it. We also identify techniques for presenting the artificially produced data to developers. Presenting the multi-dimensional data that arises from our design clearly is intricately linked to the developer’s capacity to make informed decisions about his or her design. Comprehensibly, the introduction of mechanisation in the game design process has the potential for increasing the designer’s creative reach as well as reducing his or her symbolic reasoning burden.
La conception de jeu est l’art d’élaborer un système interactif. Les concepteurs de jeux construisent des systèmes qui sont principalement composés de règles. Ces dernières décrivent comment le contenu interagit avec le joueur. Les concepteurs de jeux souhaitent façonner le jeu, cette synergie entre les joueurs et les règles. Durant ce processus, ils cherchent à comprendre et valider cette intéraction. Usuellement, cette information est obtenue par l’introduction de testeurs humains qui testent, jouent et explorent le jeu en conception.

Cette thèse propose d’introduire des techniques de calcul informatique dans le processus de création de jeux, afin de créer des joueurs artificiels qui peuvent interagir avec le contenu en développement. Les joueurs artificiels recueillent de l’information importante à propos de leurs interactions afin d’offrir un point de vue utile et important aux concepteurs. L’introduction de techniques de calcul informatique peut réduire le raisonnement intellectuel des concepteurs, le temps pour tester, les coûts de développement, etc.

En premier lieu, dans cette thèse, nous introduisons des joueurs intelligents qui utilisent directement les contrôles du jeu, nous les décrivons comme agents réactifs. Cette approche consiste à laisser les joueurs artificiels intégrer directement avec la conception. Elle permet de minutieusement tester le jeu, où dans notre autre approche, une représentation est utilisée, or ; certains détails sont perdus. Dans ce contexte, nous avons utilisé cette approche afin d’évaluer certains comportements de personnages non joueurs qui ont pour but d’accompagner le joueur humain dans un monde virtuel.

Afin d’accélérer la collecte de données interactives, nous allons explorer comment traduire un jeu vidéo en représentation mathématiquement formelle et comment utiliser cette

De plus, cette thèse présente nos observations et leçons apprises durant le développement des différents outils informatiques afin d’introduire des techniques de calcul informatique dans le processus de conception de jeu. Nous avons identifié plusieurs techniques afin de présenter des données hautement interactives. Notre capacité à présenter ses données sont directement lié avec celle des développeurs à prendre des décisions éclairées à propos de leur conception. Finalement, l’introduction de techniques de calcul informatique dans le processus de conception de jeu vidéo a le potentiel d’augmenter la capacité créative des concepteurs de jeux, en plus de réduire leurs raisonnements intellectuels.
Lauren Nicole McGann, you, the love of my life, I would like to thank you first. It goes without saying that without you my life would not be as magical and flawless. I am proud and blessed to share my life with you.

_Merci, je t’aime_

Eliana Jade McGann Tremblay, my daughter, in the grand scale of things, I believe you have accomplished more in your first year of life, than I ever have. You make me a very proud father, watching you gave me the drive to continue.

_Je suis fière de toi_

To my parents, _merci à vous pour les sacrifices que vous avez faits pour moi, de m’avoir laissé poursuivre mes rêves et, finalement et simplement de m’aimer. Vous êtes mes modèles de persévérance et détermination. J’apprends tellement de vous._

_Merci Maman et Papa_

Clark Verbrugge, first thank you to have shaped my intellect and provided me with the tools to flourish as a human being. Your guidance through this journey gave me the means to explore my own interest and expand my intellectual horizon (in so many ways).

Clark, you have all my gratitude

Most of this work presented in this thesis would have been impossible without the help of countless people. Thus, I would like to personally thank the following people for their direct help, intellect, commitment, and patience: Perdo Andrade Torres, Nir
Rikovitch, Christopher Dragert, Alexander Borodivski, Qihan Xu, Jonathan Campbell, Muntasir Chowdhury, Shuo Xu, Amir Yahyavi, Claude Gravel, Faiz Khan and Bettina Kemme.

It stands to reason that the friendships I have developed during my years in Montréal helped shape this dissertation. Ipso facto, I would like to thank, Darryel Boucher, Ben Kybartas, Lei Xu, Vladimir Reinharz, Antoine Soulé, Florence Clerc, François Mouillot, Myriam Turenne, Bentley James Oakes, Xing Shi Cai, and Jean-Christophe Guern. Thank you to all of you to have share countless espressos with me.

I would like to thank the government of Québec et les Québécois for having conceived a school system which allows someone of modest means from a small town in the great north of the province to complete a philosophical doctorate. Also, I would like to recognize the financial help provided by the Fonds de recherche du Québec - Nature et Technologies (FQRNT), the Natural Sciences and Engineering Research Council of Canada (NSERC), and Lorne Trottier. Lastly, I would like to thank McGill University and the School of Computer Science, for providing me with an environment where I could learn and grow, both as a computer scientist and as a person.

McGill University, Québec, Canada
April 12, 2016
# Contents

1 Introduction and Contributions 25

1.1 Goal ........................................... 26

1.2 Contributions .................................. 27

1.2.1 Companion NPC ................................... 27

1.2.2 Game Design Mechanisation ................. 29

1.3 Outline .......................................... 31

2 Background 33

2.1 Games .......................................... 33

2.1.1 Combinatorial Games ......................... 34

2.1.2 Digital Games ................................... 35

2.1.3 Game Genre Definitions ................... 39

2.2 NPC Formalisms ................................ 43

2.2.1 NPC Types ...................................... 43

2.2.2 NPC Structure .................................... 44

2.2.3 Decision Trees .................................... 46

2.2.4 Finite State Machine ......................... 47

2.2.5 Hierarchical State Machine ................. 49

2.2.6 Behaviour Tree .................................... 52

2.2.7 Goal Oriented Action Planning (GOAP) .... 61

2.2.8 Behaviour Tree GOAP extension ............... 66
### 8.1 Controller

#### 8.1.1 Controller

- RRT Guard Interactions & Resource Management

### 8.2 Experiments

#### 8.2.1 Controller

- RRT-Combat
- Simple Level Combat RRT
- Non-Trivial Combat Level RRT
- Adding Resources Combat RRT

### 8.3 Chapter Conclusion

#### 9 Content Generation

##### 9.1 Visibility for Content Placement

- Finding Paths
- Visibility Polygons
- Content Locations

##### 9.2 Experiments

- Wasteland
- Human Study

##### 9.3 Chapter Conclusion

### 10 Related Work

##### 10.1 Adaptivity

- Attrition Games: Targeting Enemies
- Game Solvers
- Design Tool
- In-Game Experience Analysis
- Simulating Human Movement
- Generative Content

##### 10.2 Chapter Conclusion

### 11 Future Work and Conclusions

##### 11.1 Architecture

---

201

205

213

213

213

215

215

217

221

223

224

224

226

230

232

232

235

240

242

242

243

245

247

248

250

251

252

253
11.2 Goals & Contributions .............................................. 255
  11.2.1 Companion NPC ............................................. 255
  11.2.2 Game Design Mechanisation ............................... 256
11.3 Future Work ...................................................... 256
  11.3.1 Stealth games ................................................. 257
  11.3.2 Game Design Tool .......................................... 258

Appendices

A Stealth Risk Human Study ........................................... 259

Bibliography ............................................................ 267
List of Figures

2.2 Min-max example where circle states are player 1 (maximizing) and square states indicate player 2’s turn (minimizing). The red arrows show the value returned from the evaluation and the blue arrow represents the move chosen by player 1. Figure taken from https://en.wikipedia.org/wiki/Minimax. .................................................. 36
2.3 Mental state in terms of challenge level and skill level, derived from Csikszentmihalyi’s flow model [Csi90], taken from https://media-mediatemple.netdna-ssl.com/wp-content/uploads/2015/10/02-flow-channel-opt.png .................................................. 38
2.4 Mario Bros platformer game from Nintendo ......................... 40
2.5 NPC Architecture from [MF09] ........................................ 45
2.6 A simple decision tree which govern how a NPC reacts upon seeing an enemy 47
2.7 Finite state machine representing a simple lock with two states and two transitions, where the initial state is locked ..................... 48
2.8 Simple HSM where $s_1$ is the initial state represented by a black circle, $s_4$ is a composite state with $\{s_1, s_2\}$ and $s_{\text{current}} := [s_4, s_1]$ when this HSM is initialized .................................................. 51
2.9 Hierarchical State Machine example from [MF09] .................. 52
2.10 A simple sequencer task node representing the take cover and eat sandwich behaviour .................................................. 54
2.11 Adding a health check guard to the take cover and eat sandwich behaviour.

2.12 This is the exact same behaviour as figure 2.11 behaviour tree with a different tree structure.

2.13 A simple selector task node will try in order to get gold, wood or stone.

2.14 Using the include decorator to encapsulate the take cover and eat sandwich behaviour where the root is an include task node.

2.15 The pick weapon behaviour which uses a random selector.

2.16 Behaviour consisting of an NPC who seeks to scare and/or intrigued the player.

2.17 Sleep behaviour where a sub tree is used as a state representation.

2.18 High-level showing how parallel task can be used to synchronize three different NPCs.

2.19 A generic behaviour tree.

2.20 A generic behaviour tree with look up decorators.

2.21 The general search tree transformation.

2.22 The search tree constructed for the get bone high-level behaviour from table 2.2.

3.1 Simulation game used for testing.

3.2 Base companion behaviour tree.

3.3 Cautious behaviour tree.

3.4 Support behaviour tree.

3.5 Aggressive behaviour tree.

3.6 Adaptive behaviour tree.

3.7 Plot of equation (3.7), showing threat order for different combinations of enemy health and attack.

3.8 Test level.

3.9 Game Intensity over time (s) for the pillars scenario.

3.10 Game Intensity over time (s) for the level scenario.

3.11 Game Intensity over time (s) for the naïve AI player.
3.12 Game Intensity over time (s) for the naïve AI player with adaptive companion and DDA companion
3.13 Top-down view and in-action playtime screenshot of the pillar level
3.14 Simple level uniform
3.15 Simple level medley
3.16 Simple level tank
3.17 Simple level boss
3.18 Pillar Level Uniform
3.19 Pillar Level Medley
3.20 Pillar Level Boss
3.21 Pillar Level Tank
3.22 Mimic Level Uniform
3.23 Mimic Level Boss
3.24 Mimic Level Medley
3.25 Mimic Level Tank
3.26 Cumulative histogram, showing how close to optimal threat ordering performs (discrete context)
3.27 Cumulative histogram, showing how highest attack, lowest health, and random targeting fare in comparison to threat ordering. Note that the x-axis scale differs from figure 3.26.
4.1 A* debug view. The different colours represent different possible actions at each node; the brown square and blue sphere are respectively the initial and goal positions. The final path is defined by the thick brown line.
4.2 MCTS selection
4.3 MCTS extension
4.4 MCTS simulation
4.5 MCTS backpropagation
4.6 MCTS debug view showing a dark green path from the dark green square to the green sphere. The different colour segments represent different possible actions from the search tree.
4.7 An incomplete RRT search tree

4.8 An RRT path solution (green segments and blue nodes) with its search tree (purple segments with red nodes); note that the search was conducted using three dimensions e.g., x, y and time, thus segments can cross over each other

4.9 The debug view for the RRT with 20 allowed actions for local search using MCTS as motion planner. The large blue and green spheres are the initial and goal positions respectively. Green nodes represent the sampled states, while grey nodes are part of the tree structure, linked by the red segments. The thick brown line shows the path found.

4.10 A 1500 platformer paths all drawn with a single colour

4.11 A 1500 platformer paths movement heatmap

4.12 Heat map movement of 1500 RRT paths, with darker indicating more paths. The player starts at the blue sphere on the left and must reach the green sphere on the right, avoiding enemies (yellow spheres). Dashed lines show enemy movements, here consisting of 2 cameras panning left/right, and one straight-line guard patrol.

5.1 The platformer tool interface as a Unity 3D editor panel. This gives the designer fine-grain control over different search options and algorithm choice.

5.2 Parameters for the platformer solvers

5.3 Level 1 outline where the player starts at the blue sphere and has to reach the green sphere

5.4 Level 1 showing the 3D A* debug view; it was not able to find a solution

5.5 Level 2. Black arrows indicate platform movements, and red regions indicate instant death locations. The top red area can be avoided by riding the platform moving horizontally, and the bottom stomper can be avoided by timing a dash below it.

5.6 RRT debug view of level 2 showing a tree configuration that leads to bad timing with both the top platform or the bottom stomper.

5.7 Level 3 with a heap map of RRT_A* results and the A*_2 solution shown in purple.
6.1 The enemy’s (yellow circle) field of view (cropped orange triangle) with an obstacle (white square). ................................................................. 146
6.2 Smoothing algorithm applied to a path solution, where the final path is represented with green segments and blue nodes, the purple segments show and red nodes show the search tree ........................................ 148
6.3 A 2D game space (white rectangle) extruded through time (vertical dimension). A player path (composition of blue cylinders) moves around obstacles (purple prisms). ................................................................. 150
6.4 Different enemy visualizations: a) waiting enemy b) rotating enemy c) moving enemy. ................................................................. 151
6.5 Probability map distribution of getting seen, for the same level shown in figure 4.12. The closer the color is to magenta, the more frequently a cell is observed by enemies, and the green cells are never seen by enemies. Red cells are places the player can never venture, e.g., $\Sigma_{obs}$. The red cells do not perfectly match the obstacle because of the grid granularity used. .......... 153
6.6 (a) Level design proposed by Smith [Smi06]. (b) Movement heat map found from our tool. (c) Moving the north alcove. (d) Changing guard behaviour. Please note that the little white dots are the enemy (represented by yellow circle) waypoints ................................................................. 155
6.7 A 3D overview of the MGS dock level, where (1) is the starting location, (2) collectible items, (3) a jumping point to run away from guards, (4) noise traps, and (5) the exit location. This figure was taken from the Metal Gear Solid: Official Mission Handbook [Hod98]. ................................. 157
6.8 Overview of the major paths the two enemies take in the MGS dock level, whereas their FoV is represented in orange. The player starts at the blue circle and has to reach the green circle location. ....................... 158
6.9 Metal Gear Solid’s dock level’s heatmaps of a 1470 paths found by the tool 159
6.10 Performance analysis, attempts vs. time ........................................ 160
6.11 Performance analysis, attempts vs. probability .................................................. 161
6.12 Performance analysis, grid size vs. probability .................................................. 162
6.13 Performance analysis, time dimension bound vs. probability ......................... 163
6.14 Caption for LOF

6.15 Wasteland 2 level, where the player starts at $\sigma_{init}$ and has to reach $\Sigma_{goal}$. The green curve is a final RRT solution, whereas the blue curve is a raw solution. The yellow points represent non-interactive content locations.

6.16 A preview of the level played by human players.

6.17 Human heatmap movement (please refer to figure 6.15 for the level outline)

6.18 RRT heatmap movement of 500 solutions (please refer to figure 6.15 for the level outline)

6.19 Distribution of time in seconds taken by the RRT solutions to complete the Wasteland level, also including the average (dashed line) and median (dash-dot line). The continuous line shows an equivalent normal distribution for the calculated mean and standard deviation.

6.20 Distribution of time in seconds taken by the human to complete the Wasteland level, also including the average (dashed line) and median (dash-dot line). The continuous line shows an equivalent normal distribution for the calculated mean and standard deviation.

6.21 Distribution of the distance travelled in level units by the RRT solutions to complete the Wasteland level, also including the average (dashed line) and median (dash-dot line). The continuous line shows an equivalent normal distribution for the calculated mean and standard deviation.

6.22 Distribution of the distance travelled in level units by the human to complete the Wasteland level, also including the average (dashed line) and median (dash-dot line). The continuous line shows an equivalent normal distribution for the calculated mean and standard deviation.

6.23 RRT heatmap movement with no smoothing of 500 solutions (please refer to figure 6.15 for the level outline)

6.24 Distribution of time when RRT steps are forced to move as fast as possible towards the goal.

6.25 Heatmap of 500 RRT solutions using a pushing mechanisms to be more concentrated in the hallway center (please refer to figure 6.15 for the level outline)
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.26</td>
<td>Heatmap of 500 RRT solutions using a bias towards non-interactive content locations and all the other improvements proposed (please refer to figure 6.15 for the level outline)</td>
</tr>
<tr>
<td>7.1</td>
<td>Triangulation-based roadmap of a level. Triangles edges are indicated by gray lines, with the roadmap constructed by the dual shown with red segments.</td>
</tr>
<tr>
<td>7.2</td>
<td>Metal Gear Solid’s first level with a heatmap roadmap of the level, as the segments get closer to red, the more time the enemy spends looking at it.</td>
</tr>
<tr>
<td>7.3</td>
<td>Threshold-based angular cost calculation. This figure illustrates how the Cost function maps angles to $[0.5, 1.0]$.</td>
</tr>
<tr>
<td>7.4</td>
<td>A representation of the NM metric time-window. Here enemies can see both near-past and near-future player positions, and so even if the path is successful due to changing enemy FoVs, the player is undertaking a risky movement.</td>
</tr>
<tr>
<td>7.5</td>
<td>Evolution of the presented metrics over time (s) on level 14. The numbers refer to key-frames in figures 7.6, 7.7 and 7.8.</td>
</tr>
<tr>
<td>7.6</td>
<td>Key 1 of level 14, $t \approx 200s$.</td>
</tr>
<tr>
<td>7.7</td>
<td>Key 2, $t \approx 500s$.</td>
</tr>
<tr>
<td>7.8</td>
<td>Key 3, $t \approx 560s$.</td>
</tr>
<tr>
<td>7.9</td>
<td>Level 1 from the human study</td>
</tr>
<tr>
<td>7.10</td>
<td>Level 3 from the human study</td>
</tr>
<tr>
<td>7.11</td>
<td>Level 5 from the human study</td>
</tr>
<tr>
<td>7.12</td>
<td>Level 12 from the human study</td>
</tr>
<tr>
<td>7.13</td>
<td>Metal Gear Solid’s first level with a static probability distribution of getting seen by the enemy. The darker the orange, the higher the chance of getting seen.</td>
</tr>
<tr>
<td>7.14</td>
<td>Clustering of feasible paths within the level.</td>
</tr>
<tr>
<td>7.15</td>
<td>Histogram of the different clusters for the Dist metric.</td>
</tr>
</tbody>
</table>
8.1 Heatmap movement of a trivial example with no controller where players have to move from the blue sphere to the green one. Two static enemies (represented by two yellow circles) are placed in the north hallway facing north. .............................................. 203

8.2 Heatmap movement of a trivial example with a controller Dist of value 8 meters (represented by a red segment) where players have to move from the blue sphere to the green one. Two static enemies (represented by two yellow circles) are placed in the north hallway facing north. ................. 204

8.3 Combat algorithm representation .............................................. 210

8.4 A player path (blue line) displayed in Unity 3D, with a player (blue sphere) fighting (red segment) with an enemy (yellow sphere). ......................... 212

8.5 1470 paths on the Metal Gear Solid dock level, with the Dist controller at 8 meters ................................................................. 214

8.6 Movement heat maps for different enemy and player health parametrizations. The player starts at the blue sphere and has to reach the green sphere. A single enemy is represented by the yellow dot and yellow FoV. ......... 216

8.7 Enemy paths for the non-trivial level. ..................................... 218

8.8 Non-trivial level ................................................................. 219

8.9 Non-trivial level with health pack (green square) near $e_1$ .................. 219

8.10 Non-trivial level with health pack (green square) near $e_4$ ................. 221

8.11 Non-trivial level with 4 health packs (green squares) ...................... 221

9.1 Level geometry with a triangulation-based roadmap (thin gray lines) and the graph representation built from its dual (gray circles and thin purple lines). ............... 225

9.2 The visibility region (light pink) seen by vertex $q$ (green vertex), the region is constructed from the apex vertices (red) and projections (orange), and the non-apex vertices (blue and black). .............................................. 227

9.3 The light pink regions represent two FoVs, $fov_p$ depicts the full, non-obstructed FoV of a point, whereas $fov_q$ results from the intersection of a FoV with its visibility region. .............................................. 227
9.4 Computing weak visibility. A small interval size results in FoV polygons that would merge to fill the corridor (top corridor, left side), but with a larger step size can miss portions (top corridor, right side). The extruded FoV polygon (bottom corridor) is necessarily fully merged.

9.5 Final results including one shared content (yellow) and one unique content (red) per simple path from s to t.

9.6 Wasteland level showing weak visibility region of a path.

9.7 Two shared content (yellow dots).

9.8 One shared content (magenta dots) per visibility path region, using a ratio of 0.5.

9.9 The never seen location (cyan dots).

9.10 Movement heatmap for the shared content level configuration.

9.11 Movement heatmap for the unique content level configuration.

9.12 Movement heatmap for the never seen content level configuration.

9.13 Distribution of time spent (s) in the level for each layout; the shared content (blue), unique content (magenta) and never seen (green) with their averages (dashed lines) and medians (dash-dot lines). The continuous lines show an equivalent normal distribution for the calculated mean and standard deviation.

9.14 Distribution of the distance travelled (m) in the level for each layout; the shared content (blue), unique content (magenta) and never seen (green).

11.1 Structure of our design. The black cells represent the parts provided by the game context, green cells are the parts provided by the user (game developer). Our search process is composed of the blue cells, which generates a visualizable path (red).

A.1 level 1

A.2 level 2

A.3 level 3

A.4 level 4

A.5 level 5

A.6 level 6
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Look up table example for GOAP behaviour tree</td>
</tr>
<tr>
<td>2.2</td>
<td>Look up table example for GOAP behaviour tree which includes prerequisite, please note that higher level prerequisite(s) and update(s) are inferred from lower level behaviours</td>
</tr>
<tr>
<td>3.1</td>
<td>Attack Matrix A, where the entity’s class is given in parenthesis. Swordsman (S), pikeman (P), and horseman (H).</td>
</tr>
<tr>
<td>3.2</td>
<td>Cul-de-Sac scenario results</td>
</tr>
<tr>
<td>3.3</td>
<td>Pillars scenario results</td>
</tr>
<tr>
<td>3.4</td>
<td>Level scenario results</td>
</tr>
<tr>
<td>5.1</td>
<td>Level 1 search results. All times are in ms.</td>
</tr>
<tr>
<td>5.2</td>
<td>Level 2 search results. All times are in ms.</td>
</tr>
<tr>
<td>5.3</td>
<td>Level 3 search results. All times are in ms.</td>
</tr>
<tr>
<td>7.1</td>
<td>Human rankings vs. metrics. 1 refers to blue player and 2 to the red player. 0 represents that no agreement was reached by the humans.</td>
</tr>
<tr>
<td>7.2</td>
<td>Level 1 Metrics</td>
</tr>
<tr>
<td>7.3</td>
<td>Level 3 Metrics</td>
</tr>
<tr>
<td>7.4</td>
<td>Level 5 Metrics</td>
</tr>
<tr>
<td>7.5</td>
<td>Level 12 Metrics</td>
</tr>
<tr>
<td>7.6</td>
<td>Level 14 Metrics</td>
</tr>
<tr>
<td>7.7</td>
<td>Clustering result, showing the average and median for each metric.</td>
</tr>
</tbody>
</table>
8.1 Numeric data for different level designs . . . . . . . . . . . . . . . . . . . . . . 220
List of Algorithms

1  Breath first search algorithm to find a path of behaviours to reach $\Sigma_{goal}$ in the GOAP architecture .......................... 63
2  Forward search which returns true if a valid path exists, false otherwise ........ 112
3  A* .................................................. 114
4  Compute the neighbours of a state .................................. 115
5  Path algorithm to compute a path to a node ....................... 115
6  MCTS .................................................. 123
7  RRT .................................................. 126
8  Path smoothing method ............................................. 148
9  Controller Validation .................................................. 202
10 RRT algorithm that includes combat and resources management. Please refer to algorithm 11 for CollisionEnemy and SolveCombat function definitions. .................................................. 208
11 Extra procedures called by the RRT-Solver, see algorithm 10 ............... 209
Chapter 1

Introduction and Contributions

The process of creating a modern computer game involves multiple components from heterogeneous fields, such as arts, music, computer science, etc. Developers have to create decorative and interactive content to populate the world, design environments that suggest paths for players to follow, compose music that dictates the experience mood, etc. Even though the games are created using digital tools, the methods of creation are not structured formally; most developers follow an ad-hoc process, also described as an iterative process.

The iterative process is described by looping between creating and evaluating. During the creation phase, the developer seeks to generate some art, with a certain experience in mind, for example a video game level where the player needs to fight a skeleton in order to gain a key to a particular door. During the evaluation phase, the developer might play her own level or ask other players. That play information is then analysed by the developer, rather informally in case the designer is the tester, or formally through different data visualizations produced that represent and aggregate the results produced by other players. This results in knowledge being gained about the level in creation with respect to how well it satisfies the design goals. Given that crucial information, the developer decides if she is satisfied or if she wants to alter her design. In the latter case, the iterative process is repeated. This thesis focuses on an exploration of the mechanization of the evaluation phase through computing techniques.

The mechanization of game design entails the introduction of computing techniques
1.1. Goal

into the iterative process of game development in a way that meaningfully reduces a
designer’s symbolic reasoning burdens, extends the testing phase’s information gain, and
increases the designer’s creative reach. This dissertation explores the following questions:
How can computing techniques help explore game design? What does such mechan-
ization imply in terms of representation? How do results from mechanization compare to
humans?

1.1 Goal

The goal of this dissertation is to extend human creative power using computational tech-
niques. We define computational technique as any mean of using an algorithm to gain
information about a design. In this dissertation, we are going to focus on a subset of human
creativity, e.g., digital game design. More precisely we are going to investigate multiple
game genres such as stealth, platformer as well as combat games and how we can incorpo-
rate computing techniques / mechanisation in the process of designing them.

Mechanisation, in this context, can take upon multiple roles; it can make suggestions
about the design, simulate different interactions with the design, judge the quality of the
work, emulate behaviour, learn about the designer preferences, etc. This inclusion in the
creative process is also denoted mixed initiative [NS97, YLA14]. Although since this dis-
sertation’s discourse focuses on the aspect of gaining non-trivial information about the
developers’ design through the usage of mechanisation, it is better described by the field of
computer aided design [SRN08]. In this context machines assist in the creation, modifica-
tion, analysis, or optimization of a design.

In order to aid developers in the context of game design, this dissertation uses tech-
niques borrowed from the field of artificial intelligence. We will investigate different search
algorithms as a mean to simulate in-game players behaviour. We will also explore different
algorithmic metrics heuristically measuring the in-game experience. Using these algorithm-
mic approaches, we developed automation tools in a game design software1. The tool lets
designers create a game level, then our mechanisation simulates players within the level

1Unity 3D — http://unity3d.com/
and produces analysis that describes a player’s possible in-game experience. Using the information provided by our framework, the designer then decides to alter her design or perform further testing using human players.

1.2 Contributions

Contributions of this thesis are focused on solutions and techniques applied to a number of problem areas relevant to modern game design. We divide this into two main areas. First, we investigate problems related to the introduction of companion Non Player Characters (NPC), such as online adaptivity and combat behaviour. These contributions served as initial and motivational exploratory work for mechanisation in the design process. Second, we looked at computation techniques for game design in a more general sense, using formal representations coupled with search algorithms. It is important to note that this work has resulted in multiple publications, which are going to be noted in the following descriptions. Also, the work presented in this thesis only includes Jonathan’s contributions.

1.2.1 Companion NPC

As exploration and early work in our analysis of the realm of digital games, we discuss and address problems related to the inclusion of an online NPC companion. In this context, the player taking the role of a virtual character is given an NPC companion to support her in completing digital game tasks. Examples of game companions are many, and can be found for example in the games *Skyrim* from *Bethesda Game Studios* or *Army of 2* from *Electronic Arts*. In this context we looked at two distinct problems, one involving companion behavioural adaptivity and one involving a formal analysis of a combat selection problem. This work resulted in two conference papers presented at the Foundation of Digital Games (FDG) 2013 and 2014 [TV13, TCV14].
1.2. Contributions

**Adaptive Behaviour**

We observe that companions are given the task of following the player around and offering assistance in goal completion. Making meaningful choices for the companion in this context is not trivial, as the companion’s actions need to at least approximate what a human expects of an assistant in their current context. When the companion fails to understand the situation it leads to player frustration as the companion action is not appropriate.

In this dissertation we present an adaptive algorithm that models the player’s expectation as a function of game intensity. A quantified intensity metric gives us a way to measure the player’s current experience, and use that to modulate companion behaviour. Practical evaluation of the success of that approach required using an artificial player as a proxy for a detailed human study. This work was thus useful in showing a means to improving companion adaptivity, and also in being our first adventure into the realm of introducing AI techniques into the game design process. In this case, an artificial player is able to play a level and produce meaningful data about its experience. This exploratory work motivates most of the contributions in this thesis as in addition to exposing the need for automated analysis, it also gave us an interest in exploring companion stealth behaviour as a major, interesting and under-analyzed aspect of game combat. This work was first presented at the Foundation of Digital Games (FDG) conference in 2013 [TV13].

**Combat Selection**

Combat structure in digital video games present interesting computational problem. In this dissertation we investigate the combat mechanic where two teams are fighting each other. Each entity is given an attack value as well as an health value. The teams take turn where each entity targets (select) an enemy, their attack value is then subtracted from their target’s health value. The problem is then to find a solution where one team survives. The solution consists of finding which entity a team player should target at any given moment. This thesis presents a heuristic approach to this problem allowing real-time computation of NPC behaviours during a combat scenario that achieves at worst 95% of the optimal. In terms of the overall goals of this thesis, this part of our work is another demonstration of the value of using automatic analysis and exploration in the design process. This work was
1.2. Contributions

introduced at the FDG conference in 2014 [TCV14].

1.2.2 Game Design Mechanisation

The dissertation’s main contribution is the development of a unified framework for simulating artificial players in a broad category of game genres. We explore different formal game representations, where we look at formal descriptions of game state spaces and state transitions. Within this context we present a unified framework to describe digital games as a state space in which we can simulate artificial players. We discuss three main topics: search algorithms, extensions to search algorithms, and algorithmic game design analysis for stealth games. We also investigate how we can use computational techniques to generate non-interactive content. This work was presented at the workshop on Games and Software Engineering [TV15b].

Search algorithms

Search algorithms applied to the state space of modern games face multiple challenges. The state space tends to be large, including spatial and time dimensions, as well as dimensions related to specifics of each game, such as enemy states, health, ammunition, and so on. Using naïve search algorithm within this context does not always end up computationally efficient, and/or must be restricted to a subset of solutions. We thus compare several search algorithms commonly applied to game environments, comparing them in terms of efficiency and explorability. This work is composed of multiple conference and workshop papers presented at the Experimental AI in Games Workshop (EXAG 2014), the conference on Computational Intelligence and Games (CIG 2014), and the Artificial Intelligence in the Game Design Process (IDPV 2013) workshop [TTRV13, TTV14a, TBV14]. A second concern is in how well a search result represents a human solution. We thus also include an investigation of the “humanness” of solver solutions, as an additional important factor to consider in using search-based results in lieu of human plays.
1.2. Contributions

**Algorithmic Analysis**

One important aspect to game design is to understand how players play level, *e.g.*, how hard is a certain level arrangement? This thesis proposes different metrics that are function of the design configuration as well as the solution found in the level. This work focuses on the stealth game genre, where geometric algorithms are presented to heuristically investigate the level difficulty. We also propose different visualizations in order to summarize the search space, the solution space, the possible events, *etc.* One major contribution of this section consists of developing tools that are agnostic to the solution’s provenance, artificial or human. This work was presented at the FDG conference in 2014 [TTV14b].

**Extensions**

This section presents two different extensions to the original solvers. First, when the game state space is too large, the presented solvers are not time efficient for finding solutions, for example including combat in a stealth game or using resources. We present a way to manipulate the state space which minimizes computation time. This allows us to extend the solver expressiveness with little computation overhead. Second, we extend the search algorithms in order to allow them to be fine tuned by developers in order to define certain type of artificial player, such as a risk avoiding player or a combat seeking one. This extension allows designers to evaluate how certain type of players might react to their design.

**Content Generation**

We focus some our effort into the realm of procedural content generation, where algorithms are developed to create game content. We developed a technique based on line of sight in order to locate where non interactive content should be positioned in order to assure certain conditions, such as whether this content is seen by all the possible pathing options in this level. This allows a designer to focus on building the level outline and out-source the tedious content placement, or at least gives them a sketch to start working from with formal quality guarantees. This work was presented at the EXAG workshop in 2015 [TV15a].
1.3 Outline

The rest of this dissertation is constructed as follow. Chapter 2 covers background material as well as introducing some important notations. This chapter is organised so as to give the reader a basic understanding of digital games as well as an introduction to tree-search.

Chapter 3 presents our initial work on adaptive NPC helpers/companions. In this we discuss a framework to model player’s behaviour and have her companion change its behaviour accordingly. This work also includes addressing a targeting problem central to good companion behaviour, defining which enemy is the most important to focus on. This chapter represents the oldest work of this thesis, in that while exploring companion problems we encountered the problem of following the player while in stealth (trying to avoid detection) to be quite challenging, and the desire to better explore this problem led us to the theme of this thesis, computing techniques for game design process.

Chapter 4 discusses the search algorithms used in the context of game design. We present the general search process concept using a brute force approach. We then describe three different search algorithms—A*, Monte-Carlo Tree Search (MCTS) [BPW+12] and Rapidly exploring Random Tree (RRT) [LK99]—for the game design context. This chapter also includes simple example of usage in Unity 3D. We also present different approaches used in this thesis as to display solutions found by the different search algorithms.

Chapter 5 presents in greater detail the platformer state space. It also includes an in-depth performance analysis of the different presented search algorithms using different metrics. We discuss the different qualities solver should have in order to be use in the game design process.

Chapter 6 introduces the stealth game genre state space. This chapter also dissects the game state space so as to allow a searchable representation. This chapter concludes with a discussion of the humanness of our randomized algorithm, as an important property of using search to understand possible player behaviours.

Chapter 7 presents a deeper analysis of the stealth game genre where the player needs to avoid detection while moving through the level. In this chapter we present a formal visualization for that state space, this representation is helpful in understanding why some approaches are easier or harder than others, or impossible. The latter is interpreted partly
1.3. Outline

through some novel heuristic metrics that attempt to understand how human players perceive stealth risk. We show how to extend these path-specific metrics to more general level assessments, in order to understand stealth interactions in terms of the overall level design.

Chapter 8 presents extensions to the randomized solver presented in chapter 7. We focus our effort on describing solver controllers, which allow designers to define fixed player behaviours for the artificial players using numerical values. We also spend some time extending the game state representation to incorporate guard interaction, specifically focusing on combat. This gives the solver the ability to choose between avoiding enemies or entering combat. Here we also consider search failure in the solver as another source of meaningful information on how players may experience a level design.

Chapter 9 describes some preliminary work done in the context of generative methods also denoted procedural content generation (PCG). This work falls well under the umbrella of computational techniques for the game design, in that here the machine does preliminary work in placing some content into the design. In our case, we are specifically interested in placing non interactive content within a given level such that the game is visually interesting for a player.

Chapter 10 presents the scientific work related to this thesis themes. We investigate work touching game adaptivity and the attrition game. We describe the research conducted in using algorithmic solvers and computing techniques for the game design process. We also present work which addresses the issue of measuring the player in-game experience through different means. We furthermore present work related to mimicking the players pathing behaviour. At last, we present different related computing techniques used for placing decorative content in a polygon.

Finally chapter 11 presents some software engineering lessons we learned while developing / implementing the different algorithmic approaches presented in this dissertation. We also evaluate the general contributions of this thesis as well as briefly outlining possible future work.
Chapter 2
Background

This thesis describes work that falls into multiple fields of computer science such as artificial intelligence, robotics, planning, and, software engineering with respect to digital games. This chapter first focuses on informally and formally presenting digital games. We also look at formally describing three game genres: platformers for their popularity in design, combat games, and stealth games, with the latter two also serving as primary components of the broadly popular first person shooter style of game. We follow the discussion by presenting different formalisms used by the game industry to describe and implement non player characters.

2.1 Games

Computer scientists and mathematicians have been analysing games for many years. Most of the focus has been put on combinatorial games, for which a specific definition with very clear components exists. We first present a classical definition of games, and then discuss games in the context of mainstream media, in which less well-defined concepts such as fun are a critical factor in designing them.
2.1. Games

2.1.1 Combinatorial Games

Checkers, go, and nim are good examples of combinatorial games, with simple structure that admits a clear formalization and mathematical analysis (although they were not necessarily originally defined with that intention). In general, combinatorial games are sequential (turn-based) games with perfect information, typically involving two players, 1 and 2, who take turns making moves within an abstract game state. A finite number of positions or states exist in the game, and the movement rules are clearly defined from any given board position. Players alternate turns, and a player unable to move loses. In general, each move a player makes brings the state of a game closer to termination, e.g., a player wins or loses. Combinatorial games assume complete information and there is no element of chance (random outcomes) [Guy96].

Within the above mentioned structure, it is possible to ask and answer questions such as, given a certain game state is it possible for player 1 or player 2 to win the game? For example, in the game of tic-tac-toe, if both players play optimally, the game always result in a draw. In order to answer such questions, the game is normally analysed using a tree representation, wherein a node represents a game state and an edge a player game move. Figure 2.1 shows a tree representation for the game tic-tac-toe, where the board starts empty with nine possible, but only three meaningful available choices to the player $x$ (as any other move can be derived using rotation or reflection of the board). As a function of player $x$’s move, player $o$ has two to five different move options, to which $x$ can respond, and so on. Using this approach we could calculate the number of possible states (9!, including duplicates) or find the unique positions where player $x$ wins (91).

Using this representation, an artificial player can search the tree structure using a min-max search algorithm to minimize her possible loss for a worst case (maximum loss) scenarios [vN28]. Min-max consists of evaluating the tree in such way as to find a move leading to a winning state for a player, or the best possible state if winning is not possible. The algorithm consists of recursively exploring the tree using a depth first search approach while avoiding losing branches. The leaf nodes of the tree represents winning or losing states, or gain for player 1. Gain is defined at tree leaves (terminal states), where 1 normally represents player 1’s victory, -1 her defeat, and 0 a draw. The value returned by a leaf
2.1. Games

Figure 2.1: Tic-tac-toe tree search, taken from https://en.wikipedia.org/wiki/Tic-tac-toe

can also be an integer or real value following the same convention; this is normally used to represent heuristic evaluation when the tree is too large. When using a heuristic, infinity is also used to represent player 1 winning with positive infinity (\( \infty \)) or player 2 winning with negative infinity (\( -\infty \)). Gain values are then propagated up from the leaves, computing the gain of an internal node as the maximum or minimum of its children: at even tree height a node keeps the maximum value of its children and at odd tree height a node takes upon the minimum value of its children. Figure 2.2 presents the resulting search tree for a hypothetical 2-player game where there exists at most two moves possible for each player. The min-max approach is effectively a brute-force one which becomes quite expensive as the tree height and branching factor increase. Modern improvements to min-max, such as \( \alpha - \beta \) use techniques to prune the tree, avoiding sub-trees that are less rewarding [KM75]. The algorithm \( \alpha - \beta \) and its many variations are described in many references [RN03, Abr89].

2.1.2 Digital Games

Modern digital games are different from combinatorial games in a number of ways. Digital games often include stochastic elements, e.g., the output of a combat is determined
at random. The most popular game genres use simulated realtime rather than turn-based movement, and thus extend naturally to multiplayer contexts where more than two players can play the game. The information shared by different game entities rely on partial information about the game state. In digital games a move from a player does not necessarily lead the game state to an end. In some cases the game only consists of a virtual environment to explore and interact, behaving as a complex, interactive simulation as much or more than an abstract system.

Given these properties, formal game models are necessarily less precise and this has motivated many, less operational attempts at formalization. Researchers Eric Zimmerman and Katie Salen, for instance, give a general and goal oriented definition of games: “as a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome” [SZ03]. One of the key terms of this definition is the idea of an artificial conflict, or a problem that needs to be resolved, e.g., finding a key to open a door, being the last player to pick an object, the last survivor, etc. The rules refer to a structure that limits player behaviour and defines the game, e.g., an assigned corridor.

Figure 2.2: Min-max example where circle states are player 1 (maximizing) and square states indicate player 2’s turn (minimizing). The red arrows show the value returned from the evaluation and the blue arrow represents the move chosen by player 1. Figure taken from https://en.wikipedia.org/wiki/Minimax.
2.1. Games

for a runner on a race track. Even though this definition lacks mathematical structure and it is interpretive, it encompasses a great range of games. According to Schell, the omission of fun in Zimmerman and Salen’s definition was an over-sight as fun is necessary to understand games [Sch08]. Fun is a form of pleasure that is hard to describe in logical structures. Various attempts have been made to define fun in games, and perhaps one of the most recurrent concepts is flow from the psychologist Csikszentmihalyi [Csi90].

Flow

A key property of player enjoyment in games is the sense of complete immersion into the virtual environment that they provide. This immersion has been likened to Csikszentmihalyi’s sense of flow, wherein an activity provides a (pleasurable) subsumption of the normal thought processes. Csikszentmihalyi defines eight important properties of flow in the context of various activities such as music, sports, work, etc. The task needs: 1) a sense of control over the task, 2) a disconnection with everyday life, 3) the sense of self disappearing and 4) time being altered. In order to experience flow the task has to 5) provide the possibility of completion, 6) allow one to focus on this task, 7) define clear goals, and 8) provide constant feedback regarding the result of one’s actions. Although this was not originally defined specifically for games, finding flow within games has become a major topic of discussion in the digital game scene.

Sweetser and Wyeth [SW05] introduced a model for player enjoyment based on flow theory. They adapted the eight flow criteria to a digital game context. They built a survey to evaluate GameFlow as a model of flow in games, where they mapped game features to the presented criteria. They argued that if a game respects their criteria, it will provide fun to the player and validated their hypothesis by comparing (two) games showing that the game that received a higher score on Metacritic received a higher GameFlow score as well.

How the flow relates to game challenge is also of interest to game developers. Figure 2.3 presents a 2-dimensional representation of how the combination of a user’s skills and the task challenge induces flow. For a player to enjoy a task she needs a challenge that suits her skill level: if the task is too hard it results in anxiety, and if is too easy for her skill level, she will experience boredom. To result in flow the challenge has to be epsilon bigger than
2.1. Games

Figure 2.3: Mental state in terms of challenge level and skill level, derived from Csikszentmihalyi’s flow model [Csi90], taken from https://media-mediatemple.netdna-ssl.com/wp-content/uploads/2015/10/02-flow-channel-opt.png
her skills and thus pose enough difficulty to be interesting while still being feasible. This concept is the basic motivation for multiple digital game difficulty adaptivity structures such as the one presented by Chen [Che07].

### 2.1.3 Game Genre Definitions

This dissertation uses the notions of flow and fun informally to motivate algorithmic choices. Even if these notions are useful, however, we still need structured and rigorous game definitions and representations to investigate computing techniques for the game design process. Thus this subsection will present a generic mathematical model to computationally define and represent digital games. We are also using this notion to describe NPC adaptivity. Using the formal representation, we explore three different game genres that we define in this subsection. We look at platformer games as they are normally an entry point for game developers. We also investigate First Person Shooters (FPS), a popular game genre in the video game industry, using combat and stealth games.

In general we can represent a game as a triple, $(\Sigma, A, f)$, consisting of a set of states $\Sigma$, a set of actions $A$, and an update function $f : \Sigma \times A \rightarrow \Sigma$. We denote a state within the space as follow, $\sigma \in \Sigma$. Actions are normally a user command, such as jump or shoot, which are assumed to have an impact on the state space. The update function takes as input an action and a state, and returns an updated state, e.g., $\sigma' \leftarrow f(\sigma, a)$. In order to complete this basic model we define $\sigma_{init} \in \Sigma$ as the starting state, and $\Sigma_{goal} \subset \Sigma$ as the final objective. The latter uses a region within the space to better represent conditional winning states, e.g., having health greater than zero. Note that we are also assuming the update function is deterministic. We leave the non-determinism of modern games as future work. The rest of this subsection presents specific notation for platformer games as well as stealth and combat games. These presentations extend the basic notation described above.

### Platformer Games

Platformer games are a classic genre, with a relatively simple game structure. The game level is fundamentally a 2-dimensional, Euclidean space, constrained by (virtual) screen boundaries and physical obstacles. A classical example of such a game is the *Mario Bros*
2.1. Games

Figure 2.4: *Mario Bros* platformer game from *Nintendo*

series from *Nintendo* as seen in figure 2.4. Mario has to reach the end of the level, situated somewhere on the right-hand side while avoiding obstacles such as *goombas* (the brown walking mushroom shape figure situated in the center-left of figure 2.4). A designer may also add to her level various other kinds of interactive elements, such as saws, spikes or other kinds of *death* obstacles that kill the player right away. Moving platforms are another common feature, and may repeatedly move horizontally or vertically, acting as dynamic terrain that can be leveraged to reach other areas of the screen. One fundamental feature of platformer games is the presence of a simple physics simulator system. Once the player’s avatar is in the air, it normally falls towards the ground, as per gravitational acceleration. The physics model might also include other elements of basic physics, such as momentum. Other examples of games in this genre are *Super Meat Boy* from *Team Meat* or *Mark of the Ninja* from *Klei Entertainment*.

To formally represent the state of games in this genre, we will assume a subset of features that is representative, although not exhaustive. In most platformer games, the player has basic commands to move her avatar left-right and jump; we also incorporate double-jumps (allowing a second jump while in mid-air), as a popular, if less physically realistic behaviour. Our physics model includes gravitational acceleration downward, but
2.1. Games
does not include player momentum on the horizontal axis—a player’s forward motion depends entirely on the current left/right key-press, and so may be changed arbitrarily and instantaneously, even while in the air—an approach commonly referred as air control by game designers. This gives players fine-grain and highly reactive control, as is common in platformer games.

Formally our state space, $\Sigma$, is composed of the following tuple, $(x, y, t, jump, moving)$ and it is updated using this tuple, $(A, f)$. The first two variables, $x$ and $y$ represent a 2D Euclidean space a player may occupy, $t$ is a non-negative time vector (essential for representing platform movement), $jumping$ is a 3-valued domain to indicate whether a double-jump has been performed, and $moving$ is 3-valued domain to represent motion, as either not moving, moving left or moving right. Then the final state space is composed of the following type,

$$\Sigma \subseteq \mathbb{R}^2 \times \mathbb{R}_{time}^+ \times \{0, 1, 2\}_{jump} \times \{0, 1, 2\}_{moving}$$

The set of actions, $A$, available are moving left-right, jumping straight, double jumping, jump left-right and not moving, which is not part of the state space. Our update function, $f$, is used to update the state of the world using gravity and collision detections with a fixed time update. In section 5 on page 133 we extend and present how this model can be used in the context of game design machination.

**First Person Shooter**

In order to model FPS games we use combat and stealth games, which combine the requirements of traversing a virtual environment with NPC interaction. Viewed abstractly, the primary goal of a player in a combat game is to get from $a$ to $b$ alive: the player has to survive every combat against other NPCs, with combat initiated whenever the player walks into an NPC’s Field of View (FoV). This abstract view includes FPS games such as Half-Life as well as Role Playing Games (RPG) such as Baldur’s Gate. Stealth games can then be seen as a subset of combat games, where combat itself is disallowed, or at least not required: the player has to avoid each NPC’s FoV while getting from $a$ to $b$ [Smi06]. Pure examples of this genre exist, such as in the Thief series, but many and more recent
2.1. Games

games in the stealth genre, such as *Dishonored* allows some amount of combat, and a combined consideration of both combat and stealth behaviours is essential to their design and understanding.

Although distinct game genres, both combat and stealth behaviours are fundamentally based on the task of path-finding from \( a \) to \( b \), and we can understand player behaviour as based on searching a complex space, including enemies for feasible paths. We use the following tuple to represent FPS game state, \( (x, y, t, h, E, E_h) \) and the state is updated using \( (A, f, a, E_a) \). We describe the player position using, \( x, y \), based on a 2-dimensional game terrain, wherein a set of enemies \( E \) interacts over time with the player. Each element, \( e \in E \), is composed of a set of positions—we assume the enemies follow deterministic movement paths, such that a function \( pos_e : \mathbb{R} \rightarrow (x, y, \theta) \) exists to map time to the location and orientation of the enemy, at least in the absence of player interaction. Both enemies, \( E_h \), and the player, \( h \), have scalar health values as well as (fixed, pre-determined) attack values, \( e_a \in E_a \) and \( a \) respectively, to represent the damage done by a single attack. The latter are left out of the state representation as the values are constant. For enemies the health values will be either at maximum or 0, while for the player the health value may be any number—this is meant to allow us to model game-state before and after combat, with the assumption that combat always terminates with either the enemy dead and player at partial health, or the player dead (simulation over), and so there is no need to explicitly model the behaviour of enemies with partial health. Then we have state space composed as follow,

\[
\mathbb{R}_x \times \mathbb{R}_y \times \mathbb{R}_t^+
\]

The available actions, \( A \), to the player are simple movement, modelled as a holonomic system [LaV06]. The update function \( f \) is composed of multiple parts, such as the logic for player movement collisions, combat logic, the selection process during combat, and changing enemies behaviour as a function of the player in-game interactions. Chapter 6 will extend and use this model in introducing computing techniques in the game design process.
2.2 NPC Formalisms

The main purpose of this section is to introduce the reader to the different formalisms used by game developers in order to create believable non player characters (NPC). Although to start this discussion we first present the different NPC types we might encounter in video games. We then describe a general NPC architecture implementation used in digital games to control NPCs. We discuss four formalisms: decision tree, finite state machine, hierarchical state machine, and behaviour tree which are described as reactive in this text. At last, we then present a goal oriented action planning (GOAP) approach for creating NPCs. This section terminates on extending the behaviour tree formalism to embrace concepts of GOAP.

2.2.1 NPC Types

In digital game culture the expression “AI” mainly refers to NPC behaviours and animations, even though AI techniques can be applied to different parts of the game architecture. NPCs are artificial characters that interact with the player in various ways within a digital world. They primarily come in different incarnations: opponents, companions, and minor actors [MA10]:

- **Opponents** are agents that will fight against the player. For example, in a FPS any agent that fights against the player is an opponent.
- **Companions** are agents designed to help the player complete in-game task. In this context, we can picture a team squad or a minion that follows the player in a game.
- **Minor actors** are the rest; they are the NPCs that populate the digital world, with whom the player may or may not interact. Examples exist in terms of merchants, pedestrians, etc.

In the context of developing digital games, these distinctions can be fuzzy, and specific game characters can have varying or multiple roles in a given game. The key property is that NPC behaviour needs to be computer controlled in some fashion, and when developing digital games, NPC behaviours are pre-designed by human designers using formalisms
2.2. NPC Formalisms

such as state-machines or behaviour trees to describe their reactions to the player’s actions. Below we present different NPC formalisms used by game developers to describe such reactivity. For presentation purposes we group them into two families: reactive AI and goal oriented. The former describes NPCs which change state (react) in response to the player’s actions, where their responses are pre determined by human designers, and for which we are going to discuss four formalisms: decision trees, finite state machines, hierarchical state machines, and behaviour trees. For the latter, goal oriented consists of using classic search algorithms in order to determine the best behaviour given the particularity of the situation. We will describe the general architecture and, at last, show how the behaviour tree formalism can be extended to be motivated by by internal goals.

2.2.2 NPC Structure

An NPC is built using different parts and structures, as we can see in figure 2.5, a generic game agent architecture presented by Millington and Funge [MF09]. First, an NPC uses an Execution management to control its thinking time and the components thinking order. This consists of selecting algorithms that compute movement (Movement) or decide on a behaviour (Decision making). NPC interfaces with the game world through a channel to get information about the game state (World interface), using, for example, a blackboard architecture, encapsulating the in-game environment information available to NPCs, allowing simple queries on different game objects. For believability purposes the NPCs actions, reactions, and inactions are visually animated accordingly, including adhering and reacting to constraints of the virtual environment (Physics). When designers want to force reaction or behaviour upon a NPC, they use Scripting which bypasses the agent’s generalized behaviour—the decision making process. Group AI and Character AI refer to particular techniques used to make situational / movement decisions, such as boids (bird-oid object) behaviours or communication [Buc05].

A central feature of game agents is the process of making decisions that will appear intelligent to the player. In particular, decision making refers to the ability of any agent to choose and perform actions. As a general rule game agents can be reactive or goal oriented. Reactive agents are built using formalisms that are developed to react to particular
2.2. NPC Formalisms

![Diagram of NPC architecture](image)

Figure 2.5: NPC Architecture from [MF09]
2.2. NPC Formalisms

events, generating behaviour as a consequence of player actions or specific changes in the environment. For example, a baseball player reacts to the particular event of a ball coming towards her by catching the ball. Intelligent behaviour is often associated with goal oriented agents, who attempt to satisfy high-level, internal goals through a set of pre-defined actions. For example, an enemy planning to craft a better weapon will determine the actions that need to be completed in order to reach her goal, such as getting ore, finding a smith, etc.

In the game context, reactive agent techniques are common, easy to implement, fairly convincing and successful, and developers creating game agents are normally expecting a certain control over the agent’s behaviour through formalisms. On the other hand, goal oriented AI is still an evolving research area, used less frequently in the game industry, as it is harder to implement and has the potential to take longer computing time than reactive agents. Game AI in general has strong efficiency constraints, as well as a need for testability, flexible extension, and to be easily understood and used by content designers, who may or may not be programmers.

The rest of this section is organised as follow, we first discuss the following reactive formalisms: decision trees, finite state machines, and behaviour trees for their popularity. We then present the goal oriented architecture with its different components, such as planning. Using the guidelines describe by the goal oriented architecture, we extend the behaviour tree formalism to include its ideology.

2.2.3 Decision Trees

Decision Trees are quite possibly the simplest form of game AI, and are simply a representation of nested if-else-then structures. When executing behaviour based on a decision tree, we descend down a tree structure assuming each internal node is a condition, the evaluation of which determines which child to evaluate next. Each leaf node consists of an action to perform [Qui87]. For example, figure 2.6 presents a decision tree for a guard in a FPS, if she sees an enemy, then she engages combat, else she is patrolling the area she is assigned too. This formalism allows developers to develop simple behaviours inside a small time frame, and it is, also, possible to build decision trees from data sets using
2.2. NPC Formalisms

See enemy

Engage combat  Patrol

Figure 2.6: A simple decision tree which govern how a NPC reacts upon seeing an enemy

Although quite trivial in design, decision trees are a robust structure and easy to use, care must be taken, however, to ensure efficiency as the tree evolves during the design process. In case the tree is not well balanced, reaching a particular action on a long branch of the tree could take too long for the thinking time-budget. Parts of the tree might also repeat themselves within the tree in order to offer multiple ways to reach the same behaviour. This multiplication raises modularity and code maintenance concerns. For example, the part of the tree enacting combat might occur in different parts of the tree wherever a combat situation may be detected.

2.2.4 Finite State Machine

In order to allow better transitions from behaviour to behaviour, we introduce finite state machine as a formalism that allows for code reliability and maintainability [Wri05]. The intuition behind state machines evolves around defining behaviour as states which characterize the system modelled and defining transitions based on events to reach other states. For example, imagine a simple lock mechanism as a state machine, as shown in figure 2.7. We can define two states such as “locked” and “unlocked”, where we begin in a “locked” state, the transition (event) from “locked” to “unlocked” involves turning a key clockwise, and we turn the key counter-clockwise to return to the initial state.

In general a state machine can be expressed as a tuple, \( \langle S, s_0, s_{\text{current}}, \Pi, \Phi, \delta \rangle \), where \( S \) is the set of reachable states, \( s_0 \in S \) is the initial state (represented by the state pointed at by the black filled circle and arrow in figure 2.7) and \( s_{\text{current}} \in S \) is the current state of the state machine at any particular moment. This assumes that the system can only be in one state at the time. As previously described, state transitions are a function of which events
2.2. NPC Formalisms

![Finite state machine for a lock](image)

Figure 2.7: Finite state machine representing a simple lock with two states and two transitions, where the initial state is locked

are currently occurring. In our notation, we use $\Pi$ to denote the set of all possible events; in terms of implementation, this can be represented by a boolean array, where each position depicts an event. In greater detail, it is initially filled with zeros and when events occur their array positions are switch to 1. In order to avoid concurrent events we are going to only allow one event to occur at a time. Different approaches could be taken in order to allow concurrent events such as using a queue of events. In order to transition from one state to an other, we use the function $\delta$ which takes as parameter a state, the set of events $\Pi$ and the defined transitions $\Phi$. We can trivially represent $\Phi$ as a two dimensional array with rows representing the different states and columns the different events, where $\Phi_{i,j}$ stores a state to transition to (and if a transition does not exist it stores the state $j$). With this structure, the function $\delta$ simply maps the current state with the event using the transition matrix, $s_{\text{current}} \leftarrow \delta(s_{\text{current}}, \Pi, \Phi)$; if no event is defined, the current state stays unchanged.

For game purposes, developers will create unique state machines for each NPC which are normally updated at each game frame by running the state update function. It is also common, when implementing finite state machines for NPCs, to include enter state, update state and exit state functions [Buc05]. The update function runs at each game cycle executing code making NPC behaviour believable; for example, updating the in-game world coordinate to reach certain locations or updating the body skeleton animation to the following frame. Moreover the update code can be used to start new events when particular conditions are met. For example, suppose we define a ‘combat’ state which upon being entered displays a message such as “Charge!” . The update function could then be used to avoid the visual depiction of entering combat by checking whether health is lower than
some threshold, and if so then propagating a ‘low health’ event in order to cause the ‘combat’ state to be abandoned for another state, and otherwise continuing into combat. Upon leaving the state, the exit function is executed; for example, causing the NPC to say “I am hurt!”.

In practice finite state machines tend to be very low-level and limiting. The only design tool available is the usage of transitions from one state to an other. There are no tools to capture higher level behaviour, such as encapsulating states together to describe behaviours. This limits the capability to naturally express behaviour, understand, modify and reuse foreign finite state machines. Finite state machines also do not scale easily—when extending a finite state machine to include new behaviours, developers have to define new events linked from any possible state to the new wanted behaviours. This heavy linking of the same event to multiple states can bring development problems as well as being difficult to debug and understand. Another limitation of finite state machine is the lack of state memory, e.g., NPCs forget what they were doing before entering new behaviours and do not return to the previous behaviour upon completing a new behaviour. The presented model is also equivalent to regular languages, and thus has strong limitations on expressiveness; for example, it is impossible to count without modifying the model. In order to get around these limitations, developers implement unique “hacks” that are fine tuned for each project needs, and can include any or all of the update, enter, and exit functions previously presented. This limits the capability for already implemented finite state machines to be reused in new projects due to the lack of standardization, even within the same game studio [Cha08a].

2.2.5 Hierarchical State Machine

A Hierarchical State Machine (HSM) addresses the problem of incorporating memory into state transitions. It also offers more tools to designers for encapsulating behaviours as well. In order to introduce HSMs, we are going to extend the finite state machine definition previously presented. The intuition behind HSM is to let developers define high-level behaviours and encapsulate them within a state (we denote such a state as composite). Thus
we define an HSM by a tuple, \( (S, S_c, s_0, s_{\text{current}}, \Pi, \Phi, \delta, H) \), where \( S \) is the set of all reachable states. \( S_c \) is the set of composite states\(^1\) such that \( s_c \in S_c \) is a set of states: \( s_c \subset S \). We then define \( S_c \) as a subset of the power set of \( S \) (set of all subsets \( S \)): \( S_c \subseteq \mathcal{P}(S) \). From a top level perspective, any composite state, \( s_c \in S_c \), behaves like a normal state, it reacts to events the same way a state in a finite state machine does, only its internal execution is a little different. A composite state might embed an independent hierarchical state machine which governs its behaviour. Within this recursive structure (a state within a state within a state ...), we redefine \( s_{\text{current}} \) as a linked list in order to keep track of the recursive structure the state machine is in. In essence, as we are going down within composite states we add each recursive composite state to the linked list. Thus the last entry in the list is indeed the HSM current executing state, and the other entries from start to the penultimate entry are composite states. For example, if the HSM is only composed of composite states where \( s_i \) encapsulates \( s_{i+1} \) except for the last one being a simple state, we have that \( |s_{\text{current}}| \leq |S| \).

On the other hand, if the HSM does not have any composite states, then \( |s_{\text{current}}| = 1 \), which is a finite state machine. In this context we can bound the size of \( s_{\text{current}} \) as follow, \( 1 \leq |s_{\text{current}}| \leq |S| \). Also it is important to note that both \( s_0 \) and \( s_{\text{current}} \) are lists, where we want to allow the HSM to start from any possible \( s \in S \). As in our previous definition, we will use the same data structure for the set of possible events, \( \Pi \), using a binary array and the transition set, \( \Phi \), using a 2 dimensional matrix of size \( |S| \times |\Pi| \). The state row does not treat the composite state differently than normal states. The transition function is then applied recursively starting at the first entry in \( s_{\text{current}} \) stopping when a valid transition is found, e.g., an entry not to itself in \( \Phi \). In other words, when the composite state receives an event, if the composite state does not react to the event, it passes the event to its internal states recursively. In order to assure \( s_{\text{current}} \) is valid, the state that reacted to an event has to be removed (including the other elements it points to in the data structure) and the previous object now links to the new state.

For example, figure 2.8 presents a simple HSM where \( s_{\text{current}} \) is defined as follow, \( s_{\text{current}} := [s_4, s_1] \) when initialized. Imagine the event linking \( s_1 \rightarrow s_2 \) is fired. Then we recursively applied the event to \( s_{\text{current}} \) elements and update accordingly; in this case

\(^1\)The literature also uses superstate
2.2. NPC Formalisms

Figure 2.8: Simple HSM where $s_1$ is the initial state represented by a black circle, $s_4$ is a composite state with $\{s_1, s_2\}$ and $s_{current} := [s_4, s_1]$ when this HSM is initialized.

we would remove $s_1$ from $s_{current}$ and update its parent’s next pointer to $s_2$, resulting in $s_{current} := [s_4, s_2]$. If the event $s_4 \rightarrow s_3$ would be fired, then we would operate in the same fashion, recursively applying the event to $s_{current}$; in this case the first state would react and result in $s_{current} := [s_3]$. One limitation of this approach is that state events have to stay within the same composite state level. One way to avoid this would be to use a parent pointer to the composite state it belongs to for each state in the HSM and update how $s_{current}$ is maintained accordingly.

In order to allow history in composite states we use $H$ where each composite state has an associated history state, $H^* \in H$. Upon leaving any composite state, the last internal state executed is stored in $H^*$. When returning to the execution of any composite state, $H^*$ is used to determine which internal state to execute first. The history state is also used to define the initial state of any composite state, whereas we use $s_0$ to denote the initial state of an HSM.

Figure 2.9 depicts a simple example of an HSM as an NPC controller. The model represents the behaviour of a cleaning robot. The robot looks for trash until it runs out of batteries or it does not find trash, in which case it will recharge its battery. The internal state the robot was in prior to recharging is saved in the history state denoted $H^*$ and is used to restore the saved state upon re-entry to the Clean Up mode. $H^*$ points to the initial state upon entering the composite state. The figure denotes its initial state using a solid black circle.
2.2. NPC Formalisms

In general, HSMs address very specific problems with finite state machines, mainly adding memorization (via the history state) and modularity (via the state hierarchy). In the context of game development designing an HSM in such a way that it can be reused in a different context is not a trivial task [Cha08a]. Even with the inclusion of modularity, using any form of state machine forces the developers to think in terms of logic instead of behaviours. This limits most developer’s symbolic reasoning where they think in terms of actions and not which event should cause what. It is not trivial to extend HSMs to encapsulate long term goals as HSMs and finite state machines are reactive, only dealing with events as they occur and are not capable of looking ahead. In order to address such challenges, the game industry has been making a move towards more behaviour-centric formalisms such as behaviour trees and goal oriented.

2.2.6 Behaviour Tree

A behaviour tree is a tree structure formalism with an accompanying execution semantics based on depth-first search to execute nodes in the tree. It was first introduced in 2005 to the game world by Isla for the game Halo 2 [Isl05], even though a similar formalism with the same name was already used by software engineering and robotic domains for
2.2. NPC Formalisms

some years already [Gla04, wTLJXwL04]. In order to allow game developers to develop instances of behaviour trees the semantics is limited to a handful of logical actions (tasks / nodes). Note that here we describe the basic, reactive model of behaviour trees. In the following sub-section we will introduce how the formalism can be extended to plan ahead and be goal oriented.

Formally behaviour trees can be expressed with a tuple \((\Upsilon, \delta, \Omega, \mathcal{R})\), where \(\Upsilon\) is a tree data structure composed of nodes, \(v \in \Upsilon\), from which we assume the following tree functions: obtaining the root, \(v \leftarrow \text{ROOT}(\Upsilon)\), and accessing the children of an node as an ordered set, \(S \leftarrow \text{CHILDREN}(v)\). In order to extend the developer’s modelling power, we define these four basic types of nodes: selector \((s)\), sequencer \((q)\), decorator \((d)\) and task \((t)\) such that \(\Omega := \{s, q, d, t\}\). We also define an accessor function for the task type, \(\omega \leftarrow \text{TYPE}(v)\), where \(\omega \in \Omega\). Please note that \(|\Omega|\) is generally greater than 4 and includes multiple different node types which facilitate behaviour tree modelling usability\(^2\). In order to simulate our behaviour we finally define an evaluation function, \(\delta\), which executes the behaviour tree. This function is applied in a depth first search manner to the tree nodes, starting at the root of the tree. The evaluation function returns one of the following two values: success \((\text{succ})\) and failure \((\text{fail})\), such that we have \(\mathcal{R} := \{\text{succ}, \text{fail}\}\). We then have \(r \leftarrow \delta(v)\) such that \(r \in \mathcal{R}\). It is possible to introduce a running state to \(\mathcal{R}\) allowing the tree to run over multiple evaluations, generalizing the normally binary returned value. Each task, \(\omega \in \Omega\), implements its own evaluation function and we will describe each one of them.

The simplest node is the task node and it normally defines low level actions from which we can infer the overall behaviour. In the context of the tree structure they are the leaves. They are typically implemented by user code such as scripts and execute different actions or check for different conditions, for example, play animation, play sound, start shooting, move to a location, check if there is a player in view, etc. In general, tasks are designed to be reused elsewhere in the tree, for example, a node denoted check clean may return succ if the floor is cleaned and fail otherwise, and can be reused every time the NPC needs to check if the floor is cleaned before continuing onto a specific behaviour. Using a set of developer defined tasks, it is then possible to model complex behaviours using sequencers, selectors.

\(^2\) Although this is often just syntactic sugar.
2.2. NPC Formalisms

and decorators which are the internal tree nodes and are normally denoted as composite tasks [MF09].

Composite tasks standardize the tasks relationships, such as the order of execution. The composite task behaviours are functions of the evaluation function values returned by evaluation of their children. A sequencer task node, normally represented by an arrow, encapsulate a series of tasks that needs to be executed in order. Figure 2.10 shows a simple series of tasks to be executed in logical order (child order), describing a behaviour where an NPC has to first pick a cover location, reach that location and then eat a sandwich. Using this intuition we define a sequencer task as follow,

\[
\delta(q) = \begin{cases} 
\text{succ} & \forall c \in \text{CHILDREN}(q) : \delta(c) \text{ is succ} \\
\text{fail} & \text{otherwise}
\end{cases} \tag{2.1}
\]

The algorithm retrieves the sequencer task node children in order and evaluates each nodes in that order. Upon finishing evaluating each children node, the evaluation function of a sequencer task returns succ if all children returned succ or fail otherwise. It is normally implemented to return fail as soon as a child fails. It is important to note that the order is extremely important as the first node might be a guard to the sequence execution. For example, figure 2.11 extends the previous example to include a low health guard check before executing the cover sandwich behaviour. Note that behaviour trees are not in general uniquely structured; the behaviour tree shown in figure 2.12 presents the exact same behaviour as the one in figure 2.11, but it uses a different tree structure, requiring an extra internal node.

The selector task node tries every one of its children until one returns success and is

---

The sandwich is a reference to the character Heavy in the game Team Fortress 2 from Valve, who eats sandwiches to recover lost health.
2.2. NPC Formalisms

Figure 2.11: Adding a health check guard to the take cover and eat sandwich behaviour

Figure 2.12: This is the exact same behaviour as figure 2.11 behaviour tree with a different tree structure

normally graphically represented by a question mark. Figure 2.13 shows a simple selector which tries to get gold first, respecting the children order, if the previous task fails, then the NPC tries to gather wood, and if this previous behaviour also fails the NPC, finally, mines stone. In case all three behaviour failed, the selector task returns fail. On the other hand, as soon as one of these behaviour succeeds, the selector task returns succ. We then define a selector node as follow,

\[
\delta(s) = \begin{cases} 
  succ & \exists c \in \text{CHILDREN}(q) : \delta(c) \text{ is succ} \\
  fail & \text{otherwise} 
\end{cases}
\]  

(2.2)

Thus we are interested in finding the existence of a children task that is successful. This function is implemented to return succ as soon as one task is found to be successful. The order of the children is also important as a designer might want to prioritize certain behaviours over others.

A decorator task is the last of our modelling tools, but it will allow us to extend the expressiveness of the behaviour tree formalism. A decorator task was inspired by the decorator object-oriented paradigm wherein a decorator task simply wraps a single child node with a new functionality [GHJV95]. The decorator task functionalities tend to be developed for the particular developers needs, although we shall look at a set of the most popular
2.2. NPC Formalisms

Figure 2.13: A simple selector task node will try in order to get gold, wood or stone ones, as described in a popular behaviour tree library [Bao15]:

- **AlwaysFail** always return *fail* even if the child was successful.
- **AlwaysSucc** always return *success* even if the child failed.
- **Inverter** returns success if the child failed or failure otherwise.
- **Include** simply encapsulates a sub-tree allowing for behaviour encapsulation. It returns the same value as its child, and its purpose is to enhance modularity and reusability.
- **Limits** controls the number of time its child can be executed. For example, if a player had locked a door and the NPC does not have the ‘unlock’ behaviour, the NPC might try to open the door infinitely often. We could limit this behaviour to execute 2 times. If the limit is not reach, it returns its child returned value, otherwise it returns *fail* without executing the child task.
- **Repeat** keeps executing the same child task for \( n \) times or even infinitely. Upon reaching its loop limit, it simply returns *succ*. Note that unlike **Limits**, this form of iteration ignores its child returned value.
- **Semaphore** assures that certain sub-tree are not instantiated more than \( n \) times within the NPC population of the same kind. In greater detail, imagine we have 5 NPCs which use the same behaviour tree to dictate their comportments and the cover / sandwich is part of it. The designer uses a **semaphore** task to assure that only one NPC is in the cover / sandwich behaviour. If you are not allowed to execute the child, then the node returns *fail*, otherwise it returns its child value. This task can be implemented using a blackboard architecture to share the behaviour resource.
- **UntilFail** keeps repeating the child task until failure, then it returns *succ*.
- **UntilSuccess** keeps repeating the child task until it succeeds, then it returns *succ*.
2.2. NPC Formalisms

The above list is not meant to be exhaustive but instead to present the breadth of decorators that can be instantiated. For example, figure 2.14 shows how the inclusion of a decorator task (include) can be used to encapsulate the take cover and eat sandwich behaviour previously used to introduce sequencer tasks. This allows the sub-tree to be reused and easily readable by designers.

In order to facilitate the usability of the internal tasks, and to avoid designers repeating the same sub-tree with slight variation, we also introduce randomized sequencer (graphically represented by $\sim\rightarrow$) and randomized selector (graphically represented by $\sim?$). These allow for a greater expressiveness while modelling certain behaviour. Figure 2.15 presents a select weapon behaviour which uses a random selector task. Upon evaluating the selector task its children order is shuffled and then evaluated like a normal selector task. In this case, the uniform distribution makes each weapon equally probable, giving more variety to the NPC’s weapon choice.

Figure 2.16 uses most of the different tasks presented in this section to describe an NPC behaviour. The general behaviour consists of an NPC who seeks to scare and/or intrigue the player. The behaviour starts by evaluating the sequencer task at position 1 which consists of either picking a random weapon and staring at the player or hiding away from her and eating a sandwich, if the NPC can see the player. Please note that ‘pick weapon’ behaviour is presented by figure 2.15 and the ‘Cover sandwich’ is depicted by figure 2.14. Upon failing to see the player, the NPC will check if the player is hiding in a room, (position 2). In case of success the NPC will enter the room to investigate, unlocking the door first if necessary. Finally the NPC will simply wander around if the player is unseen and not

![Figure 2.14: Using the include decorator to encapsulate the take cover and eat sandwich behaviour where the root is an include task node.](image)
2.2. NPC Formalisms

Figure 2.15: The pick weapon behaviour which uses a random selector

Figure 2.16: Behaviour consisting of an NPC who seeks to scare and/or intrigued the player
2.2. NPC Formalisms

In order to handle concurrent behaviours, we also introduce the **parallel** composite task. The parallel composite task starts all of its children at the same time. The update function of a parallel task depends on its flavour:

- **Parallel sequencer** returns `fail` as soon as one of its children return `fail`; if all of its children success, then it returns `succ`. It is graphically represented by three arrows stack on each other, `⇒`.

- **Parallel selector** returns `succ` as soon as one of its children return `succ`, if all of its children fail, then it returns `fail`. It is graphically represented by three question marks (`??`).

The parallel composite tasks can only be used with non conflicting actions; *e.g.*, ‘move to cover task’ and ‘move to resource’ are not compatible, as they would issue conflicting commands to the moving function. The parallel composite task is also useful to extend expressiveness; *i.e.* while behaviour trees do not have a clear state representation, designers can use parallel tasks to check multiple conditions at the same time a certain behaviour is executed, and so simulate conditions for reaching a certain state. Effectively, instead of using transitions to change states as function of the event, we use a sub-tree as transitions. For example, figure 2.17 presents a sleep behaviour where the left sub-tree is used as a state representation allowing the NPC to sleep quietly.

It is also possible to use the parallel composite tasks to control the behaviour of a group of NPCs. In order to accomplish such a goal, we have to define a higher-level behaviour tree which forces the NPCs into using different behaviour trees. For example, figure 2.18...
2.2. NPC Formalisms

shows how parallel tasks can be used to synchronize three different NPCs. We first start to ask each NPC concurrently if they see the enemy, where NPC 1 is the first child of node (1) (and generally NPC i is located at child position i such that i ∈ {1..n}, where n is the number of NPCs). In case one of them sees the enemy, the NPCs concurrently enter into the fight behaviour (task (2)), where NPC 2 will take upon a support role. If the NPCs cannot see the enemy they will take on non-combat roles (task (3)), where NPC 1 patrols around while NPC 2 and 3 are digging. It is important to note that the NPC behaviours such as dig or fight have to be implemented carefully to return the appropriate values. This is where extending the evaluation values, \( R \), to include \textit{running} is important, for example if NPC 1 could not enter in combat at task (2) and return \textit{fail}, it would cause the synchronized behaviour to stop.

Overall, the behaviour tree formalism as presented is similar to a hierarchical state machine such that it reacts following a certain logic to different events and it allows designers to group behaviour together, in this case explicitly using subtrees rather than composite states. Behaviour trees have gained in popularity over the years, as they are practical and intuitive to use, scale more easily than state machines (according to an informal study [Est14]), offer fine grained dynamic control over behaviour and multiple logical tools.
to construct behaviours. Since control is implicit, *e.g.*, there is no ‘real’ state representation, however, bugs can also be difficult to understand and fix. In the following section, we investigate how to extend behaviour tree formalisms to take planning into consideration, as according to Alex Champagnard this is the most valuable aspect of the formalism [Cha08b].

### 2.2.7 Goal Oriented Action Planning (GOAP)

In this subsection we are interested in defining NPCs that are motivated by internal goals and plan actions in order to achieve them. This approach allows for dynamic agents which can adapt themselves to never seen situations, where a GOAP NPC always picks an action to execute, even though it might not be appropriate. Initially GOAP was inspired by the Stanford Research Institute Problem Solver (STRIPS) [FN71], although the action conditions are taken from hierarchical task networks [EHN96]. GOAP architecture has been used in the series *F.E.A.R* from VU Games as well as *Deus Ex: Human Revolution* from Ubisoft Montréal. This section first focuses on defining the formalism architecture and its search process when we are given a goal. We then present different approaches for goal selection, and conclude this section by presenting how the behaviour tree formalism, previously presented, can be adapted to the GOAP architecture.

In a GOAP architecture an NPC picks first a goal to complete, then searches all of its available behaviours, forming a chain of actions that results in achieving its internal goal. In order to understand the implementation details, let's define GOAP using the following tuple $(B, R, f, s, \Sigma_{\text{now}}, \Sigma_{\text{goal}})$, where $B$ is a set of basic, primitive behaviours (short arrangement of actions), $\mathbb{E}$ is the set of possible environments (game states), $\Sigma_{\text{goal}} \in \mathbb{E}$ is a specific goal state, and $\Sigma_{\text{now}} \in \mathbb{E}$ is the current state. In general, we will use $\Sigma \in \mathbb{E}$ for arbitrary states. $R$ is a set of rules used by the function $f$ to determine the goal environment.

We define a **search function** also denoted as planner function, $s$, which takes as input a goal environment, the current environment, $\Sigma_{\text{now}}$, and a set of behaviours, and returns a sequence of behaviours that leads to the goal, *e.g.*, $p \leftarrow s(\Sigma_{\text{goal}}, \Sigma_{\text{now}}, B)$. In order to facilitate this procedure we implement different accessors functions such as $\text{IN}(b, \Sigma)$ and $\text{OUT}(b, \Sigma)$; where the former checks if the environment $\Sigma$ is such that $b$ can be performed, returning *True* or *False* as appropriate. The latter function, $\text{OUT}(b, \Sigma)$, returns the resulting
environment when the behaviour is applied to the environment, $\Sigma' \leftarrow \text{OUT}(b, \Sigma)$. Please note that this approach is similar to the update function presented in section 2.1.3, although for GOAP the state space, $E$, is not explicit, it is defined by developers\(^4\) in a way such that $\text{OUT}(b, \Sigma)$ can easily be optimized through look-up tables (we will extend this notion in the following paragraphs).

Then the resulting path of behaviours from the search process is structured as $p := \{p_0, \ldots, p_{m-1}\}$, where each element, $p_i : (b, \text{parent}, \Sigma)$, is composed of a behaviour, a parent pointer and the environment, $\Sigma$, the behaviour $b$ was applied to and within which we use accessor functions to obtain the different values; for example, $b_i \leftarrow b(p_i)$. Please note that the parent pointer will be use to construct the path after the search process. A valid plan forms a chain of behaviours from our current state to the goal state, guaranteeing that each behaviour can be executed. This can be translated as follow, $\Sigma(p_0) = \Sigma_{\text{now}}$, $\forall i = 0, \ldots, m-2 \text{OUT}(b(p_i), \Sigma(p_i)) = \Sigma(p_{i+1})$, which depicts that each element in a path (expect the last entry) is linked to its following when behaviour $b(p_i)$ is applied, we also have that $\forall i = 0, \ldots, m-1 \text{IN}(b(p_i), \Sigma(p_i)) = \text{True}$, which assures that each behaviour results in a valid environment, and $\text{OUT}(b(p_{m-1}), \Sigma(p_{m-1})) \in \Sigma_{\text{goal}}$, where the last element of the path is within the goal environment.

The search process, $s$, consists of finding such a valid arrangement of behaviours. This can be accomplished using any kind of search process; in discussing it here we will just use breadth first search, as given by algorithm 1. It shows how the breadth first search algorithm can be extended to find paths of behaviour to reach goals. The algorithm starts by defining a queue data structure in order to keep track of the vertexes to unfold next. Lines 3 through 8 initialise the search by applying each valid behaviour in our set of behaviours $B$ to the initial environment, $\Sigma$. Using the queue, we pop the first element, which we denote as $p'$ and then evaluate if this reach our desired environment (lines 12–14). In case we reach the goal, we return the path of behaviours that took us to this environment using the \text{PATH} method. This method starts by initializing a stack with its argument ($p$), then recursively push the parent of $p$ until the parent is undefined. The stack is then returned as an ordered set, $P$. The rest of the search process consists of applying the valid behaviour to $\Sigma'$ defined

\(^4\)This is why we do not use the standard notation we developed for the rest of this thesis.
2.2. NPC Formalisms

**Algorithm 1** Breath first search algorithm to find a path of behaviours to reach $\Sigma_{goal}$ in the GOAP architecture

```
procedure BFS($\Sigma_{goal}, \Sigma, B$)
  queue ← {} 
  for each $b \in B$ do 
    if IN($b, \Sigma$) then 
      $p \leftarrow (b, nil, \Sigma)$ 
      PUSH(queue, p) 
    end if 
  end for 
  while queue ≠ {} do 
    $p' \leftarrow$ POP(queue) 
    $\Sigma' \leftarrow$ OUT($b(\Sigma'), \Sigma(\Sigma')$) 
    if $\Sigma_{goal}$ is $\Sigma'$ then 
      return PATH($p'$) 
    end if 
    for each $b \in B$ do 
      if IN($b, \Sigma'$) then 
        $p \leftarrow (b, \Sigma', \Sigma)$ 
        PUSH(queue, p) 
      end if 
    end for 
  end while 
  return nil 
end procedure
```
by applying the $b(p')$ function to its contextual environment (line 11). Using this new contextual environment we apply each valid behaviour and store the composed node in our queue data structure (lines 15–20). The algorithm terminates upon exhausting the queue, for which it will return nil as it did not find a path of behaviours.

Multiple problems can arise if algorithm 1 is used as is, such as infinite looping and incompleteness of the search. Algorithm 1 assumes there are no cycles (e.g., the graph is a tree) in how the environments are structured and that there is a finite set of environments. Chapter 4 on page 109 introduces algorithms and analysis tools to allow search process in continuous context. Since the search process can be expensive, specially applying $\text{OUT} (\Sigma, b)$ multiple times, most modern implementations precalculate the function, $\text{OUT}$ for most possible environments. Also, in order to decide which node to evaluate next, special heuristics are used to rapidly prune the tree search and to force the planner towards certain paths quickly. As heuristics are fine tuned, the GOAP quickly degenerates towards state machines, where behaviours are predetermined. Moreover, in general the environment space allowed to be searched is restricted by placing strict limitations on the representations. The goal is to force the planner to only take highly probable branches rather than exhaustively search the space (such as with algorithm 1).

As we have now a general idea on how to find a path of behaviours, the second important question is how to select goals. We previously defined the GOAP architecture with the following tuple, $\langle B, R, f, P, s, \Sigma, E, \Sigma_{goal} \rangle$, where we did not discuss $R$ and $f$ yet. For GOAP to be believable, each NPC is given a world representation. Most simply, we can represent that as an element $\Sigma \in E$, although in general it can be extended to include heuristic or incorrect interpretations of the game state. Using this information we use a set of rules, $R$, where any rule, $r \in R$, and the NPC’s current understanding of the world, $\Sigma_{now}$, can be used to find a goal with a function, $f$. One simple approach to implement the goal selector would be to define $R$ as a super set where each element, $r$, would be a set of conditions associated with an heuristic value and a specific goal [MF09]. Then $f$ simply maps the maximized score of $\forall r \in R$ and to its associated goal environment. Every now and then the goal is re-evaluated which causes the planner to find a new path of behaviours if the goal changes. Using rules in this manner relies heavily in fine tuning weights in order to pick the right goal. Also one problem that arises would be how to decide which goal to
2.2. NPC Formalisms

pick when two sets of rules have the same weight. One could also assure that each $r \in R$ are mutually exclusive. Moreover in order to avoid rules based system, developers could look into using state machines or even planning algorithms for goal selection.

Using the different pieces presented in this section we investigate a GOAP example. Please note that this example is inspired by Orkin’s original paper on GOAP [Ork05]. Let’s define the following NPC relevant behaviours: GotoTarget, Attack, Patrol, and TraverseDoor and the following relevant goals: KillEnemy, Investigate and TraverseLink. For the purpose of this example, the NPC is a guard in a bank. The NPC sees the player running away and entering a room. At this moment, the NPC updates $\Sigma_{\text{now}}$ to include knowledge about the player location. The NPC selects the goal KillEnemy, which causes the planner to plan for the following behaviour path, GotoTarget(Room) $\rightarrow$ Attack(Player). Upon reaching the room door, in which the player is, the NPC fails to open it and updates $\Sigma_{\text{now}}$ to include that the door is blocked or locked. This causes the goal to change to TraverseLink(Door) from which the planner defines a new behaviour path, TraverseDoor(Door). Since the player blocked the door with a heavy object, the previous plan failed and the NPC updates his knowledge of the environment to include that the door is non-traversable. Since there exists no other way into the room, the NPC picks as goal to Investigate(Room) and will pick the behaviour Patrol(Room). For this specific example, it would have been possible to construct the same output behaviour using a state machine knowing exactly how we wanted the NPC to react. Using a planner to find solutions to a goal, however, offers more flexibility, as behaviour plans can be constructed even though a designer might have never thought of this specific situation (which is harder with state machines).

In general GOAP has been tried in a number of different research contexts. Still, it is not a standard in the game industry, due to implementation difficulties and efficiency issues [MF09]. Using a general planner as presented, finding a path of behaviours can be an expensive task, where the NPC has to execute expensive backtracking search algorithms to find the best set of actions, in a realtime context. Issues also arise when the set of behaviours is large and the NPC does not choose the path of behaviours imagined by the game developer for a given situation. With these issues in mind we describe how we can relax the GOAP architecture and use it with a behaviour tree formalism.
2.2. NPC Formalisms

![A generic behaviour tree](image)

**Figure 2.19: A generic behaviour tree**

### 2.2.8 Behaviour Tree GOAP extension

When introducing GOAP we omit to describe how the behaviour set, $B$, is implemented. It was assumed that each behaviour was composed of a set of actions that could have been scripted, they had pre-existing conditions to be executed, and could be evaluated describing the resulting state. We now introduce how the **behaviour tree** formalism can be used to express high-level behaviours and goals using a new decorator, look-up tables and a simple tree transformation. Please note that the examples in this section are inspired by Champagnard’s 2007 GDC Europe talk on behaviour trees [Cha08b].

In the previous section we described behaviour trees using task nodes: leaf nodes determining lower level actions and internal nodes describing the execution comportment of the tree. Figure 2.19 presents a generic behaviour tree where each selector node (on the second level) is used as a guard to the sequencer behaviours, and each high-level behaviour represents a generic sub-tree. For example, imagine we want to model a dog chewing a bone. Since it is a sophisticated dog, she always eats her bones in a bowl and would never eat a bone without it. Thus $b_1$ and $b_2$ represent two ways to get a bone, e.g., go to the bone bag or dig one up. The second selector maps to ways the dog could get a bowl, if both selectors return `succ`, we can finally execute the eating a bone behaviour, $b_{f1}$. In order to extend this structure to be goal oriented, we need to include some abstraction to the tree construction, separating the goals from the behaviours.

In order to do so we introduce a look-up table with high and low-level behaviours. Table 2.1 shows an example of a look-up table, where each high-level behaviour is linked to a selector sub-tree which includes all behaviours from that family of behaviours. For example the *idle* high-level behaviour is constructed from multiple lower level behaviours,
Table 2.1: Look up table example for GOAP behaviour tree

<table>
<thead>
<tr>
<th>Behaviour name</th>
<th>Behaviour tree</th>
</tr>
</thead>
</table>
| Idle           | ?
|               | patrol bone eating ...
| Suspicious     | ?
|               | smell search ...
| Alert          | ?
|               | bark show teeth ...
| Look-up(g₁)    | Look-up(g₂) →
|                | b₁ b₂

which can in return be constructed from other entries in the table. In order to use this table in the tree we introduce the **look-up** decorator, which takes as input the wanted behaviour and returns the sub tree it maps to. Figure 2.20 shows how the previous example is extended to include different goals (gᵢ), where g₁ is the get a bone goal and g₂ is the get a bowl goal the instantiation of which would result in the reconstruction of a tree similar to the one previously described. This tree itself could be registered as a higher level goal, such as a *satisfying hunger* goal. Using this approach, a designer can easily define multiple behaviours and hook-then together in a goal oriented approach. This approach is simpler than ‘tweaking’ heuristic values in order to get an NPC to execute the right order of behaviour as previously described.

As we are constructing higher level goal oriented behaviours, our tree might get very unstable, such as starting to seek a bone even though there is no bowl available (remember
that our dog is sophisticated). We now describe how we can build a search tree from our high-level behaviour look-up table. In order to solve this instability problems designers could specify all prerequisites for the completion of the any high-level behaviours and how it will update the environment knowledge. A simpler approach would be to simply define low-level prerequisites and construct a search tree to verify higher level goals as follow. Table 2.2 shows the extension proposed to the look-up table; *get bone* 1 and 2 have different prerequisites but update the world with the same knowledge (having a bone). In this case we do not have higher level prerequisites for the *get bone* behaviour as they are inferred from lower level behaviours. We then transform the behaviour tree to a search tree using the same formalism. Each low-level subtree maps to a sequencer where its first child, $c(b_i)$, checks if all prerequisites are satisfied, which returns *succ* if all conditions are met, *fail* otherwise. In case of success, the second sequencer child, $u(b_i)$, updates the environment knowledge and returns *succ* when the update was successful. This selector structure is then mapped to the higher level behaviour using the same internal composite tasks. It is important to note that knowledge is passed to the following task in a sequencer task but it is reset to the initial entering environment in a selector task. Using the search tree we can then find and return the proper behaviour tree that we are sure can be executed. This search tree transformation (planning) can be repeated periodically or when new knowledge is added to $\Sigma_{now}$. It is important to note that the same results could be achieved without the search tree transformation, but it would need fine-tuned condition definitions from the designers, which introduces a higher chance of mistakes.

In general, this search-tree transformation allows a designer to really focus on high-level behaviour definitions and not worry about the prerequisites of them, assuming that the low-level behaviours are correct. Although, they are also expecting specific behaviour
Table 2.2: Look up table example for GOAP behaviour tree which includes prerequisite, please note that higher level prerequisite(s) and update(s) are inferred from lower level behaviours

<table>
<thead>
<tr>
<th>Behaviour name</th>
<th>behaviour tree</th>
<th>prerequisite(s)</th>
<th>Update $\Sigma_{now}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>get bone</td>
<td><img src="image" alt="Tree" /></td>
<td>bone bag not empty</td>
<td>bone</td>
</tr>
<tr>
<td>get bone 1</td>
<td><img src="image" alt="Tree" /></td>
<td>bone bag not empty</td>
<td>bone</td>
</tr>
<tr>
<td>get bone 2</td>
<td><img src="image" alt="Tree" /></td>
<td>cemetery</td>
<td>bone</td>
</tr>
</tbody>
</table>

Figure 2.22: The search tree constructed for the *get bone* high-level behaviour from table 2.2
2.3. Chapter Conclusion

to specific situation, this significantly increases the complexity of tweaking and tuning the planner results. But, this burden also brings NPCs that can react to a broader range of situations, specially to unknown domain [Cha13].

2.3 Chapter Conclusion

This chapter mainly introduced background work and knowledge central to this thesis subjects. We described AI with respect to digital games, presented different game definitions to help us formally present a mathematical game model for platformer games and FPS. We also investigated the different formalisms used by the game industry to model NPC behaviours in digital games such as decision trees, finite state machines, GOAP, etc. In the following chapter, in the context of introducing computing techniques in the game design process, we discuss different research problems of interest when introducing an NPC companion to the player, such as combat adaptivity and target selection in combat. We use the behaviour tree formalism to describe companion NPC adaptivity in the following chapter. This allow us to describe reactive tester agents, explore the realm of different fixed behaviours and its impact on the game experience.
This chapter focuses on computational techniques used to design NPC companions. In this game context, the companion’s goal is to help the player accomplish in-game goals, simulating the effect of actual co-operative gameplay, and as such the player’s relationship with it is intricately linked to the gameplay, challenge, narrative and experience. Interesting challenges and problems arise from this situation, and even in “triple-A”—high production value—games we frequently find that companions are overly superficial: they may co-exist with the player, but they often fail to appropriately cooperate [BSP05], reducing their value to players, and interfering with immersion. In a fundamental sense, these problems arise from the companion’s lack of understanding of the game world and the player’s dynamic, changing experience. From the perspective of introducing computing techniques in the game design process, the construction of artificial players represents a first step towards automated testing, and provides the initial experience and computing techniques we use in the rest of the thesis to further our goals of improving automation in design.

Our specific goal in this chapter is to present an approach to adaptivity in companion behaviour in order to modulate the player’s in-game experience. In our design the companion is given a basic understanding of the player’s experience and uses that knowledge to change its behaviour accordingly, both in terms of how it assists the player, and how it reacts in combat situations. We want to enable companions to react as clever team-mates, eliminating the need for extra-narrative control of companions during combat, and so giving designers another means of controlling game difficulty.
We show that using our approach to companion adaptivity allows the companion better control over the player’s game experience than a classic game-industry approach to NPC design. We also demonstrate a parallel between behavioural adaptivity and more traditional Dynamic Difficulty Adaptation (DDA), showing how behavioural adaptivity can be embedded seamlessly in narrative, as part of the game design itself, rather than as extranarrative control. We demonstrate how the use of heuristics in combat situation can achieve near-optimal solutions when selecting target. Contributions of this chapter include:

- We develop an online adaptive model for companions in combat games, enabling companions to change behaviour according to basic knowledge of the player’s experience.

- We define a set of metrics to measure the player’s in-game experience, allowing us to gain representative knowledge about the human player–companion relationship.

- We develop a simple but justified target selection strategy through mathematical analysis of a reduced problem space. Our approach is similar to prior analytical work by Churchill et al. and Furtak et al. [FB10, CSB12], but considers scenarios more common to FPS and Role-Playing Games (RPG) contexts. This analysis verifies that a “threat ordering,” prioritizing enemies based on two dimensions (health and attack), is theoretically optimal.

- Through experimental analysis of several representative scenarios, we show that threat ordering outperforms other commonly used target selection strategies, even when considering the additional complexity of real-time gameplay, probabilistic hit and damage, character movement, and physical occlusion.

This chapter is constructed as follows. We first present our game context, describing the different in-game interactions. A presentation of our adaptive model is then discussed, which is followed by an in-depth analysis of the target selection problem. The chapter also presents two sets of experiments, the first exploring the qualities that describe our presented adaptive companion, and a second exploring target selection under different real-time combat scenarios.
3.1 Game Prototype

In order to test our approach to adaptivity, real-time combat decision making, and game design exploration, we develop a game prototype using *Unity 3D*. The purpose of this prototype is to serve as an open platform with which to test companions, gather information about their performance, as well as testing larger game principles such as combat games. We also focus our effort in building tools in Unity 3D which use computational techniques to help designers understand possible game solutions. This prototype implements a basic FPS interface, including a third person view camera, shooting combat, enemies, a player, a companion AI, *etc*.

The in-game goal of the player is to gather blue boxes as seen at the top-left of figure 3.1. The challenge arises from enemies: they walk around in a predetermined path and engage in shooting combat with the player or the companion, if either are seen. If the player or companion runs away from them and they are no longer visible, they will chase them to their last seen position, where that position is stored in their internal game representation. When that position is reached and the player is not found, the enemy goes back to patrolling. In figure 3.1 two enemies can be seen dressed in black; the companion approaches from the right, and the player is bottom center. The companion’s goal is to help the player accomplish the in-game challenge by shooting at enemies and defending the player. The non-controllable companion will be further described in section 3.2 and the adaptive companion is de-constructed in section 3.2.1. The game does not involve any resource management. The player has infinite ammunition and there is no way to recover health. In doing so, issues of adaptation under resource management such as ammo, health packs, *etc*., are avoided in order to focus on the behaviour of the companion.

To facilitate testing, an Artificial Intelligent (AI) player was designed to automatically play the game. The player automation presented here is a first step towards introducing computing techniques in the game design process. A game developer could use the artificial player to test different designs and consult the metrics collected in order to make informed design decisions. In terms of specific behaviour, the AI player walks around collecting blue boxes. If it sees an enemy it will engage in combat using a pattern of shooting then dodging to the left or the right (all agents use this core strategy). This AI player thus
3.2 Adaptive Companion

Figure 3.1: Simulation game used for testing

represents a rudimentary, but straightforward human player, who attempts to clean out the level as rapidly as possible by engaging in every combat and moving to the next goal without pausing. A more complex AI could be used to represent other player strategies, but this does not change the basic approach.

In order to validate our adaptive model, a base companion was developed. Figure 3.2 shows the behaviour tree outlining its behaviour, please note that the behaviour tree formalism is described in depth in chapter 2.

The base companion offers standard companion behaviour but it does not take into consideration the player’s in-game experience. The behaviour tree in figure 3.2 is translated as follow. The companion will fight with any enemy it sees, which is represented by the task node “See and shoot”. Otherwise, the companion will seek to reach the last known position of an enemy (Know enemy loc. – Move to enemy). When the player is immobile (Player not moving), the companion picks a random position near the player and moves to
3.2. Adaptive Companion

Adaptivity in modern computer games happens in multiple forms. The approach used in this chapter was inspired by a component of the DDA system in the well known combat game *Left 4 Dead*. They used a proprietary game “intensity” metric to estimate the player’s level of excitement, and so tailor the game experience to the player’s needs [Boo09]. It is a function of different actions taken, health decay rate, distance to enemies, etc. Based on this intensity measure, if the game intensity is too high, the DDA system will remove enemies from the map to temper the difficulty, increasing the number of enemies again only after intensity has returned to a low level. In our case we use intensity to give guidance to companion behaviour, and so allow it to better adapt to the player’s experience. The adaptive companion uses three distinct behaviours, switching between them based on the intensity measure: *cautious*, *support* and *aggressive*. We first present the three different behaviours and then introduce the adaptive model.

Cautious

Typical AI companions in combat games engage enemies on sight, even though this does not always match player intention. The *cautious* behaviour was thus designed to respect the player choices as to whether it is time to enter combat or not. In particular, if the player...
3.2. Adaptive Companion

Figure 3.3: Cautious behaviour tree

is sneaking around and avoiding combat, she will not attack enemies until she decides it is
time to do so. The cautious behaviour tree in figure 3.3 is translated as follows. The com-
panion first checks if the player is in combat (Combat mode). In this case, it will engage
combat with any enemies it can see (See and shoot), if there are no enemies and the com-
panion does not have any knowledge of a last known position (Know enemy loc. – Move
to enemy), the companion will move to the player’s left or the right (Player left/right). In
case the companion sees an enemy but the player is not in combat, the companion will find
a position where the enemy cannot see her and is close to the player (Move hidden enemy).
If the player is simply moving around, the companion will avoid contact with the player
(Avoid player), otherwise it will follow the player respecting a certain distance (Follow
distance). In case the player moves out of the companion’s line of sight, the companion
will move towards the player’s position (Follow sight).

Support

This behaviour was designed to develop a close companion that offers great support at any
time and efficiency in combat. Its behaviour tree in figure 3.4 is similar to the cautious
behaviour. They differ in how they handle seeing an enemy, primarily in that the support
behaviour will engage combat even if the player is not in combat. If the player is in com-
bat it will keep the same behaviour as previously described, and similarly, if there are no
enemies, the behaviour is also the same as in the cautious behaviour.
3.2. Adaptive Companion

![Support behaviour tree](image1)

![Aggressive behaviour tree](image2)

**Aggressive**

This companion was designed to ease the combat load on the player, mainly by seeking combat and enemies. The *aggressive* behaviour tree in figure 3.5 is translated as follows. The companion will engage in combat with any enemies it sees, and will also chase an enemy to its last known position. If there are no enemies, the companion will explore the level to find enemies, only stopping if they are too far away from the player. The exploration is done through a set of waypoints put in the level; when exploration occurs the companion simply selects the closest previously unseen waypoint as a destination.
3.3 Targeting an Enemy

Adaptation
The different sub-behaviours of our companion NPC are mixed within the same behaviour tree using the structure shown in figure 3.6. This design makes use of a scalar intensity metric which is defined in the following section; when the game intensity is over a threshold, $X$, the adaptive companion will pick the *aggressive* behaviour to aggressively engage the enemy and so try and reduce the intensity. In the situation where the intensity is moderate, over threshold $Y$ and assuming $Y < X$, the companion will support the player, trying to share the combat load more evenly with the player by switching to a *support* behaviour. When the game intensity is low, it assumes that the player is in control. In this case the companion uses the *cautious* behaviour, letting the player take on the bulk of the combat role and decisions. Note that this design embeds adaptivity in a relatively simple, static behaviour tree model. It would also be possible to vary behaviour by creating or modifying trees at runtime; *e.g.*, in the game *Driver: San Francisco* the behaviour tree structure was modified dynamically by rearranging child nodes based on *hints* [Oci12]. Overall the adaptive companion was designed to have a better understanding of the right behaviour to use in different game intensity situations. This is expected to give gameplay improvement over the static, base companion.

3.3 Targeting an Enemy

While it is important for a companion to make good high level behaviour decisions, it is also imperative for her to make good decisions at a low level, such as which enemy
to target first in combat situation. This is a basic concern in many games, where sub-optimal targeting choices by automated companions require a player to spend significant time micromanaging or correcting their behaviours in combat situations.

Abstractly, the target selection problem exists within (Basic) Attrition Games, games wherein two sides, players and enemies, seek to eliminate the other. A solution to attrition games consists of a path of targeting enemies over multiple rounds. At each round, the players pick their targets then their damage values are applied to the selected enemies. For example, in a given round we may have a mapping, \( p_1 \rightarrow e_1, p_2 \rightarrow e_1, \ldots, p_n \rightarrow e_1 \) where each \( p_i \) is a player entity in combat targeting the same enemy \( (e_1) \) and targeting it for this round. A solution consists of \( m \) rounds of targeting entities. It has been previously shown that finding the optimal surviving solution for one team in Basic Attrition Games is exponential (i.e., \( \text{BAGWIN} \in \text{EXPTIME} \)) [FB10], while the decision problem where we decide if a team can win is PSPACE-hard, and is therefore not feasible in real-time. Rather than solve the general form of the problem, we thus aim instead to explore faster heuristics that can easily be computed in polynomial time and which are suitable for the small to medium scale combat situations typically found in FPS/RPG games.

The goal of such heuristics is to find an enemy attack order that maximizes total remaining player team health without evaluating the entire combat tree (state space). A naïve heuristic might be to have all players attack the enemy with the lowest health, or target the enemy with highest attack, or even attack the enemy with the highest health. However, any of these obvious heuristics will fare poorly under different scenarios. For instance, attacking the enemy with the lowest health is a poor choice when there is an enemy with only slightly greater health but much greater attack power. Intuitively, we should target enemies that are easy to kill and which may cause lots of damage first, and enemies which are hard to kill but induce low player damage last. The former represent high threat enemies, while the latter have less priority. Below we demonstrate that this simple model actually has a well justified mathematical basis, describing first a discrete time context, and then extending the result to a more realistic real-time environment. Note that this formulation builds on the mathematical analyses found in work by Furtak and Buro, but deviates in its derivation [FB10].
3.3. Targeting an Enemy

3.3.1 Discrete Time

The following combat scenario will be used to define our basic attrition game. We begin with a set $P$ of players (1 human and some companions) that are fighting a set $E$ of enemies, where $|P| = n$, and $|E| = m$. Each entity $p \in P$ and $e \in E$ has attack $p_a(e_a)$ and health $p_h(e_h)$ with $p_a, e_a, p_h, e_h \in \mathbb{N}^+$, and we use $P_a, P_h, E_h,$ and $E_a$ to represent the total attack and health of player and enemy teams respectively. Fighting occurs in rounds, where the players and enemies each select an opposing entity to attack. A player’s attack is resolved by deducting $p_a$ from an enemy’s health $e_h$. If this leaves $e_h \leq 0$, the enemy is killed and removed from further rounds. Players hit first, meaning that a defeated enemy will not attack during the round in which it is killed. The game plays in turns, which we denote by $T$, e.g., on the very first round $T$ is 0. We can determine the time it takes to kill an enemy, for example $T^\alpha(e) = \lceil e_h/P_a \rceil$, if all the players attack that entity. Any attack exceeding the health of the target is wasted—this is known as overkill, and as we will show below it has an impact on target selection. The game ends when either all players or enemies have been killed. Enemies will choose their targets randomly, and for convenience, $P_h \gg E_a$, simulating role playing games where players typically have an advantage in order to ensure continued gameplay.

An enemy will deal damage each round until it is dead, and so health savings for the player are maximized when the enemy is killed as quickly as possible. We express the maximum health savings $S(e)$ for an enemy $e$ as,

$$S(e) = e_a \cdot (T_{actual} - T^\alpha(e))$$  \hspace{1cm} (3.1)

where $T^\alpha(e)$ is the minimum length of time needed to kill $e$, and $T_{actual}$ is a bound on how long the combat can possibly last (counted in turns). Unfortunately, $T_{actual}$ and $T^\alpha(e)$ vary based on target assignment. We can, however, give a lower-bound on $T_{actual}$ using the total enemies health divided by the total players attack as follow,

$$T_{actual} \geq \left\lceil \frac{E_h}{P_a} \right\rceil$$  \hspace{1cm} (3.2)

However, the possibility of overkill (when $p_a > e_h$) means the real combat length may exceed $T_{actual}$. For instance, consider the situation where we have one player, $n = 1$, with
3.3. Targeting an Enemy

attack 100 and we have \( m \) enemies with each 1 health. Imagine \( m \) is 100, using our previous formulation, \( T_{actual} \) would be 1. This result is illogical, since you cannot spread your attack, you need at least one round per entity, and thus it would take at least \( m \) turns to defeat \( m \) enemies for one entity, regardless of \( E_h \) and \( P_a \). Instead, we approximate \( T_{actual} \) with a summation of each enemy’s health divided by the total players attack as follow,

\[
T \approx \sum_{e \in E} \left[ \frac{e_h}{P_a} \right]
\]

(3.3)

This provides a reasonable estimate since it accounts for the number of enemies. It does allow for overestimation of \( T_{actual} \) \( (e.g., \) in the case where every player can kill any enemy in a single attack and \( n \geq m \)), but this overestimation turns out to be necessary. Consider the situation where \( T = T^\alpha(e) = 1 \). This implies that \( S(e) = 0 \) for all \( e \in E \). If there is overkill, \( T_{actual} \) could be greater than \( T \), yet our savings estimates are all zero, providing no guidance. Overestimating guarantees that we maintain information about enemy attacks and thus can still differentiate targets even in the presence of overkill. In Eq. (3.1), we also need to compute \( T^\alpha(e) \), which is given by

\[
T^\alpha(e) \geq \left[ \frac{e_h}{P_{e,a}} \right]
\]

(3.4)

where \( P_{e,a} \) is the total attack of the subset of players targeting \( e \). We use this subset of attack values to reduce overkill: if for a given \( p \), we have \( p_a > e_h \), then it would not make sense to consider all \( P_a \), so using this reduced attack value allows us to take into account the effects of spreading out attacks among enemies. With values for \( T \) and \( T^\alpha(e) \), we now expand Eq. (3.1) to get our final equation for savings

\[
S(e) = e_a \cdot \left( \sum_{e \in E} \left[ \frac{e_h}{P_a} \right] - \left[ \frac{e_h}{P_{e,a}} \right] \right)
\]

(3.5)

Target selection proceeds by summing \( S(e) \) over all enemies for every possible pairing, \( C \), of \( P \) on \( E \), which has \( m^n \) possibilities since an enemy can be targeted by more than one player:

\[
\max_{c \in C} \left[ \sum_{e \in E} S(e) \right]
\]

(3.6)

The pairing \( c \) that gives us the maximum savings is our target selection. Evaluating Eq. (3.6) takes \( O(m^n) \), and requires no manipulation or transformation from the basic
parameters of the problem. As combat proceeds, we reevaluate each round to determine the optimal savings given that enemies have had their health reduced and may have died.

### 3.3.2 Real-Time Problem

The real-time formulation allows for entities to evaluate the best target at every moment. This means that players can react to changes in game state, such as an enemy dying, and change their attack instead of wasting it. By eliminating the possibility of overkill, Eq. (3.2) becomes exact. Thus, we can evaluate exactly which enemy offers the highest savings. The priority of all targets decreases in linear proportion to time, and so relative priority ranking remains constant over time. Eliminating targets in priority order thus guarantees an optimal outcome in real-time scenarios. We reach the same conclusion as Kovarsky and Buro [KB05], and find that changing targets is suboptimal as it guarantees that the optimal savings will not be reached. In general, all players should always be attacking the same enemy. Using this knowledge we can rewrite Eq. (3.5). Here, $P_{e,a}$ is equal to $P_a$ as the players will all pick the same enemy. Using that knowledge we can drop $P_a$ and get

$$\max_{e \in E} [e.a \cdot (E_h - e_h)]$$

(3.7)

What this means is that targets combining low health and high attack are preferred. We call this strategy threat ordering. Figure 3.7 plots Eq. (3.7) for varying combinations of $e_a$ and $e_h$ while $E_h$ is kept constant. The scale on the right shows relative threat order for different enemy statistics.

### 3.3.3 Augmented problem

In a lot of modern games, attack power depends on the nature of the entities involved. It is not uncommon for example, for entities to follow a rock-paper-scissor structure to the combat, e.g., swordsman (S) beats pikeman, pikeman (P) beats horseman, horseman (H) beats swordsman. We can represent this using an attack table, $A$, where each cell represents a player attack value relative to a particular enemy. An example is shown below, in table 3.1 each entity is given a subscript $e_i$ or $p_j$. 

82
3.3. Targeting an Enemy

Figure 3.7: Plot of equation (3.7), showing threat order for different combinations of enemy health and attack
3.4. Metrics

Table 3.1: Attack Matrix $A$, where the entity’s class is given in parenthesis. Swordsman (S), pikeman (P), and horseman (H).

<table>
<thead>
<tr>
<th></th>
<th>$e_1$ (S)</th>
<th>$e_2$ (P)</th>
<th>$e_3$ (H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$ (S)</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>$p_2$ (P)</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$p_3$ (H)</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

In order to solve this problem, we have to make some adjustment to $P_e, a$ in Eq. (3.5). Rather than assuming a fixed attack value for each agent, the attack value is found in the matrix $A$. When this change is made we use Eq. (3.6) in order to find a target per attacker.

We can think of further augmentation where entities have a choice of different actions in making an attack, e.g., slow attack with lots of damage or fast attack with low damage. This problem can also be reduced to the original problem but with some cost. A simple approach would be to determine the average attack of both attacks over time, as is common practice in game design in order to define difficulty [Sch08]. Using this average will not of course give a precise result in every scenario, but otherwise we would have to work it out with a look-up tree search, which is too expensive for a real-time context.

Other augmentations exist which are more difficult to incorporate into our model. In particular, we are limited to situations where no resources or magic influence the entities’ behaviour or world state, e.g., putting an entity to sleep or reducing an entity’s attack. Shuo Xu extended this work to explore the influence of sleep—making an entity pass on his attack for $n$ turns—in small scale attrition games [Xu15].

3.4 Metrics

The two previous sections presented approaches to improving the in-game player experience, such as a which enemy to target. In order to verify our proposed solutions this section introduces ways to measure the player’s in-game experience and performance, as well as the companion’s performance, which will be used in section 3.5 (Experiences).
3.4. Metrics

3.4.1 Player’s Experience and Performance

In the following two sections, we present different metrics used to measure the player in-game experience. We first present the game intensity metric inspired by a similar metric used in the game *Left 4 Dead* from Valve [Boo09]. Second, we present different statistics such as personal space, number of kills, health ratio, etc.

**Game intensity**

In order to represent the player’s experience we developed a game intensity metric [AV11, Boo09]. The intuition behind this metric is to measure an estimate that represents the in-game intensity as function of the player’s actions. This measure is bound between two values where the higher it is the most stressful the game is or where the cognitive load to play the game is at its highest. On the other hand, when the value is low, the cognitive need is low and the game does not stress the player. We denote the game intensity metric using \( I \). The game intensity is a real number, \( 0 \leq I \leq 1 \), it is measured / updated periodically during gameplay, e.g., every second. The intensity is initialized to 0, where a value of zero represents no intensity (simply walking around). On the other hand, a value of one depicts an experience at full intensity, e.g., difficult boss combat. Between each update, the different entities’ actions are stored in a set of actions, e.g., the set \( A_{enemies} \) stores all game intensity related actions executed by the enemies, such as shooting and dying between two updates. The value is then a function of four components: the player’s health loss between updates \( f(\Delta h) \), the distance to shooting and dying enemies \( g(A_{enemies}) \), the companions actions \( h(A_{companion}) \) and the player’s actions \( k(A_{player}) \).

\[
I_{\text{now}} = I_{\text{prev}} + f(\Delta h) + g(A_{enemies}) + h(A_{companion}) + k(A_{player}) \quad (3.8)
\]

Equation 3.8 presents the overall game intensity update calculation, where \( I_{\text{prev}} \) is the calculated intensity from the previous update, it is 0 when undefined. \( f(\Delta h) \) is defined as follow,

\[
f(\Delta h) = \alpha \times \Delta h \quad (3.9)
\]

Equation 3.9 uses a linear function to update the game intensity; as the health loss increases so does the intensity. We use an \( \alpha \) value to determine the proper rate, this term
is fine tuned by the developer. The perceived intensity for the player is influenced by her proximity to combat which we translated as follow,

\[ g(A_i) = \frac{A_i}{\sum_a \left( |\beta \times \frac{1}{d(a)^2}| \right)} \]

where the function \( d \) returns the distance where the action had taken place relative to the player. We also use \( \beta \) value as a scaling factor, much like \( \alpha \). In the context of the enemies, \( A_{enemies} \), the actions of interest are enemies shooting and dying. These actions represent the most stressful interactions in combat games. We use the same function for the companion action set: the player perceived intensity is influenced by having her companion losing health as well as being in combat. Please note that sets \( A_{enemies} \) and \( A_{companion} \) may be empty if there is no combat.

The last intensity component, \( k(A_{player}) \), represents the different actions the player could do to influence the game intensity, it is very similar to \( g(A) \), with one difference, it returns a negative value when the set of action is empty as to decrease intensity. In other words, if the player is exploring the level without combat, the game intensity has to be updated to reflect on the low cognitive load needed, \( e.g., \) low game intensity. Please note that if the updated game intensity value is outside the interval \( 0 \leq I \leq 1 \), it is clamped to the closest value, \( e.g., \) 2.3 is updated to 1.

**Game Statistic**

In this section we present different statistics that we believe capture essential in-game player experience, such as personal space, participation ratio, number of kills, \( etc \). These metrics are used to unfold the events that happened during the game session. Since we used the game intensity metric in our adaptive model, we use these metrics as an unbiased way to compare the influence the different companion behaviours have over the in-game experience. Here we present a list of the different metrics used.

- **Personal Space** (PS) is used to measure how long the companion spent too close to the player. We measure in seconds the time spent within a radius around the player.
- **Player Health End** (PHE) represents the final player’s health at the end of the level.
3.5. Experiments

- **Companion Health End** (CHE) represents the final companion’s health at the end of the level.

- **Participation Ratio** (R) shows the shared load in terms of health loss. It is calculated as, \( R = 1 - \left( \frac{PHE}{CHE} \right) \). If the ratio is positive, it means the player had the most intense combat. If the ratio is negative it means that the companion did more work.

- **Player Kills** (PK) is the player number of kills at the end of the level.

- **Companion Kills** (CK) is the companion number of kills at the end of the level.

- **Player Successful hits** (PHi) is the player number of successful hits when shooting at the enemies, *e.g.*, shots that touched the enemies.

- **Companion Successful hits** (CHi) is the companion number of successful hits when shooting at the enemies, *e.g.*, shots that touched the enemies.

- **Average Game Intensity** (GI) is the average game intensity recorded for a play through. It uses the previously presented game intensity metric where we keep its average at each second.

### 3.5 Experiments

This section presents the results of our adaptive model and an in-depth analysis of the threat ordering algorithm. We first discuss different companion-player scenarios where we compare a non-adaptive companion, our proposed adaptive companion, and a dynamic difficulty approach. We then present a different set of scenarios which investigate our threat ordering strategy compared to classic strategies such as targeting the highest health or highest attack. We also discuss the threat ordering algorithm relative to full look-up tree search approaches.

#### 3.5.1 Adaptive vs. Base Companion

This subsection focuses on validating our approach to adaptive behaviour using different experiments. The first one consists of comparing the adaptive and base companion
3.5. Experiments

behaviours previously described, where we discuss the testing environment, the different
scenarios it is composed of, and the experimental results comparing both behaviours for
each scenario. Using the same scenarios, we also investigate how the adaptive companion
behaviour can help naïve players achieve a similar in-game experiences to expert ones. We
conclude this subsection with a comparison of classic dynamic difficulty adjustment to our
adaptive companion.

Figure 3.8: Test level

**Scenarios**

In order to evaluate the previously presented companion behaviours, we designed and im-
plemented different scenarios in our game prototype and analysed the resulting play traces
with the previously presented metric suite. These scenarios were designed to reflect differ-
ent combat situations common to FPS games, and were analyzed separately and in aggre-
gation as a full game level.

Figure 3.8 shows a top-down view of the overall level used for most of the testing. The
3.5. Experiments

player and companion start at the green dot and have to reach the magenta dot. The trace
overlay shows the output from a single simulation, where the red and purple lines represents
player and companion movement respectively. The blue Xs show when the player entered
combat and the red splashes mark where an enemy died. The black circles are where the
companion and player entered in collision. The white squares are goal items the player
needs to collect in order to successfully complete the game.

In order to compare the adaptive and base companion, three core scenarios were devel-
oped within the context of a full game level. These scenarios were inspired by common
components of level design found in AAA action games. Below we present the different
scenarios and their results. Note that all scenarios are smaller parts of the full level as de-
scribed above and shown in figure 3.8. Every scenario was run 20 times for each type of
companion, using the artificial player to fulfil the role of the human participant. The artifi-
cial player allows for a mechanised testing, where human testers are not involved. The use
of mechanisation here allows for a non-evolving player, the artificial player, to always play
the scenario in the same fashion, and thus does not learn where the enemies are located or
shoot better at each iteration as a human being would. This facet is important when testing
different companion behaviours, in order to ensure the changes recorded most likely come
from the companions’ behaviour.

**Cul-de-Sac Scenario** - A major level-design feature in war games involves the use of
narrow, dead-end paths wherein the player finds a reward. While common, this induces a
basic problem for companion AI: companions follow the player, but upon reaching the end
and turning around to continue on her quest, the player finds her companion blocking the
way out. A simple way to solve this problem is to increase the distance the companion must
maintain to the player, but this requires limiting the depth of such dead-ends to the follow-
radius of the companion, and can overly separate player from companion. The proposed
adaptive companion understands that when not in combat, it should be out of the way when
the player approaches. In figure 3.8, the yellow section represents the maze part of the
level. For this section, the player starts at the entrance, and then she has to reach a goal at
the end, and then return to the entrance.

Table 3.2 presents the most meaningful metric results from this scenario. Please note
that a cell is composed first of its average value followed by the standard deviation, e.g.,
3.5. Experiments

Table 3.2: Cul-de-Sac scenario results

<table>
<thead>
<tr>
<th>Type</th>
<th>Time (s.)</th>
<th>LS (s.)</th>
<th>PS (s.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>24.9 ± 0.1</td>
<td>5.2 ± 0.1</td>
<td>7.7 ± 0.1</td>
</tr>
<tr>
<td>Adaptive</td>
<td>23.3 ± 0.1</td>
<td>4.1 ± 0.0</td>
<td>6.22 ± 0.0</td>
</tr>
</tbody>
</table>

Table 3.3: Pillars scenario results

<table>
<thead>
<tr>
<th>Type</th>
<th>Time (s.)</th>
<th>PHE</th>
<th>CHE</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>35.1 ± 4.0</td>
<td>83.1 ± 4.0</td>
<td>90.8 ± 4.1</td>
<td>0.3</td>
</tr>
<tr>
<td>Adaptive</td>
<td>30.3 ± 2.4</td>
<td>84.4 ± 3.5</td>
<td>86.7 ± 4.5</td>
<td>0.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PHi</th>
<th>CHi</th>
<th>PK</th>
<th>GI</th>
</tr>
</thead>
<tbody>
<tr>
<td>78.6 ± 8.1</td>
<td>36.8 ± 9.9</td>
<td>3.8 ± 0.8</td>
<td>0.68 ± 0.1</td>
</tr>
<tr>
<td>72.6 ± 4.1</td>
<td>43.2 ± 5.2</td>
<td>3.2 ± 1.2</td>
<td>0.52 ± 0.1</td>
</tr>
</tbody>
</table>

24.9 seconds with a standard deviation of 0.1 seconds. It is noticeable that gameplay with the adaptive companion took less total time than with the base companion. This is explained as the adaptive companion did not have to be pushed by the player in order to move out of the way. The adaptive companion also spent less time in the line of sight (LS) of the player, as well as less time in the player’s personal space (PS). We also observed that in every simulation of this level the base companion collided with the player, whereas the adaptive companion did not.

**Pillars Scenario** - The Pillars scenario was developed to test the companion behaviour inside intense and obstacle-rich combat zones. In figure 3.8, the blue section represents the pillar scenario. The room is filled with six enemies and pillars behind which the player and other agents can hide or use as cover. This design is standard in FPS for combat zones or boss fights.

Table 3.3 gives interesting metrics results for this scenario. As with the cul-de-sac, the time taken by the adaptive companion to finish the scenario was shorter, albeit with more variance due to the randomness of combat resolution. In this case the game intensity caused the adaptive companion to switch to a more aggressive behaviour, whereas the base
3.5. Experiments

Figure 3.9: Game Intensity over time (s) for the pillars scenario

The companion does not adapt. The participation ratio (R) is closer to zero for the adaptive companion, showing a better distribution of the combat load. As the adaptive companion is more aggressive its successful hits (CHi) value was slightly higher than the base companion, which brought the player number of hits down (PHi), although there was less noticeable impact on number of player kills (PK). An important overall result is that the average Game Intensity (GI) was lower for the adaptive companion. The Player Health End (PHE) and Companion Health End (CHE) metrics did not show any meaningful differences.

In general, the data shows that the adaptive agent is able to reduce the intensity load on the player; this can be seen in more detail in figure 3.9, which shows the game intensity (y-axis) evolution over time (x-axis in seconds), where error bars represent the noise recorded in the game intensity evolution, one bar represent 2 standard deviation span. The adaptive companion successfully dampens the more intense sections of the level, where the base agent does not evolve and does not help the player in accomplishing difficult tasks.
3.5. Experiments

Table 3.4: Level scenario results

<table>
<thead>
<tr>
<th>Type</th>
<th>Time (s.)</th>
<th>PHE</th>
<th>CHE</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>142.5 ± 8.8</td>
<td>63.5 ± 5.0</td>
<td>92.5 ± 6.9</td>
<td>0.31</td>
</tr>
<tr>
<td>Adaptive</td>
<td>131.9 ± 5.0</td>
<td>71.2 ± 3.9</td>
<td>83.3 ± 10.1</td>
<td>0.14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PHi</th>
<th>CHi</th>
<th>PK</th>
<th>GI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>238.7 ± 24.3</td>
<td>51.6 ± 24.9</td>
<td>8.55 ± 1.6</td>
<td>0.43 ± 0.1</td>
</tr>
<tr>
<td></td>
<td>182.7 ± 13.2</td>
<td>108.9 ± 14.1</td>
<td>6.8 ± 1.4</td>
<td>0.36 ± 0.1</td>
</tr>
</tbody>
</table>

**Level Scenario** - As a more complete test, we also composed the scenarios into the full level shown in figure 3.8. The AI player starts from the green spot and has to reach the last goal where there is a boss fight (magenta spot). This level was designed to test the companion in a realistic game environment, incorporating a maze/cul-de-sac section, combat situations with varying number of enemies and obstacle density, and a final boss fight.

Table 3.4 gives metrics results for this level scenario. Note that variance is increased here in all factors, primarily due to the non-determinism built into combat aiming and combat positioning. Nevertheless, again gameplay with the adaptive companion can be seen to be more efficient, reducing average level completion time. Player hits (PHi) are higher with the base companion than the adaptive one, showing that the adaptive companion helps the player more than the base companion. This is further shown in companion hits (CHi) and player kills (PK), and also reflected in the health ratio (R), which is closer to zero for the adaptive companion – the adaptive companion worked harder and took more damage, which resonates well with the last argument.

Figure 3.10 shows the game intensity over time for this scenario. At around 80 seconds, the adaptive companion started using the aggressive behaviour, which caused the player to experience less intensity in a shorter period of time. From a flow perspective (see chapter 2), this allows players to improve their skills while avoiding excessive anxiety. Overall the adaptive companion shows a better understanding of the game situation in term of intensity. This understanding has subtle differences in term of player and companion performance, but this difference has an impact on the player’s in-game experience.
3.5. Experiments

Figure 3.10: Game Intensity over time (s) for the level scenario
3.5. Experiments

**Naïve & Expert AI Player**

The value of companion adaptivity can also be seen as a mean to help games adapt to different player skill levels. A naïve player will have weaker aim and be slower in completing a scenario than an expert player, and so player experience may be improved by a companion that can recognize the need for intensity reduction and respond accordingly. The naïve and expert artificial players are using the same behaviour tree, only their shooting skills differ. We design the naïve player to use a high probability miss to target when shooting, whereas the expert player always hits her target.

![Game Intensity over time (s) for the naïve AI player](image)

**Figure 3.11: Game Intensity over time (s) for the naïve AI player**

For this particular test, the game intensity variability is the most interesting metric to look at. The earlier figure 3.10 shows the game intensity produced assuming an expert player, while figure 3.11 shows the game intensity data from a naïve skill level run. In both cases, and as expected, the game intensity level for the AI player is lower when playing
3.5. Experiments

with the adaptive companion than with the base companion. The naïve player scenario does show dramatically larger variance, but the impact of an adaptive companion results in a better moderation of intensity. Note that this impact is partly hidden by the way the intensity measure caps at 1.0; as the game proceeds, a naïve player in conjunction with the base companion reaches the highest intensity levels, while gameplay with the adaptive companion is able to offer some reduction. A further, interesting observation is apparent in the shape of the curves from about 90s onward in figure 3.11. At this point, the base companion is essentially unable to help the naïve player, resulting in a continuously intensive latter third of gameplay, with no calmer periods. This highly stressful gameplay is in contrast to the results shown with our adaptive companion, which at least partially restores the intended, natural up-and-down pace to the level, producing a series of peaks and values much more similar to that experienced by an expert player.

In general, the adaptive model is more successful at helping a player adapt to subtle differences in difficulty. These sorts of differences happen when a player is not paying attention to the game or is learning a new game mechanic [Sch08]. Of course stronger and more universally aggressive companions would also reduce intensity, although that brings concerns of over-trivializing gameplay, which is also not the goal of commercial computer games. The companion adaptivity is there to tweak the difficulty of the game to give a better experience to the player.

Comparison to DDA

Adaptivity within games is most frequently and easily expressed through some form of dynamic difficulty adjustment based on uniformly increasing or decreasing enemy or player power. We thus compare our design with a basic DDA. The DDA system uses the same structure as the proposed model in Section 3.2.1, but modifying only companion fire power according to the game intensity—the higher the intensity the stronger its fire power. Note that as this thesis chapter is interested in adaptive companions, fire power adaptivity was given to the companion only. This particular companion will be denoted DDA companion, and an intensity graph for a naïve player with a DDA companion, an adaptive companion, and a hybrid DDA-adaptive companion (including both behaviours) is shown in figure 3.12.
3.5. Experiments

Figure 3.12: Game Intensity over time (s) for the naïve AI player with adaptive companion and DDA companion
3.5. Experiments

Much like the adaptive, as the DDA companion is given more fire power it takes less time on average to finish the level, slightly compressing the intensity curve over the time axis (and so peaks/valleys line up less well). It is clear that improving fire power has a large impact, and is also able to recover sinusoidal fluctuations in intensity for the naïve player. The hybrid form, however, shows a consistent, further mitigation of intensity, and this leads to the thought that there is an orthogonality between the adaptive companion controlling the game intensity and a more classic form of adaptivity. An appropriate combination of the two techniques may thus enable the reduction in intensity of naïve play to reach the expert levels, and result in a more uniform game experience. This additional impact must of course be measured in relation to the known drawbacks of DDA, in that letting players witness dynamic adjustments that grossly exceed their own abilities or contradict previous experience break immersion. In this sense our adaptive companion provides a much more contextually integrated form of adjustment.

3.5.2 Targeting an enemy

Even though the companion can follow the player around, she is not aware of what makes a good choice as target during combat. This subsection presents an in-depth analysis of her choices and the impact it has on the player’s performance. We first present the four targeting strategies used by our companion. We then introduce four scenarios inspired by actual game situations which were developed in order to compare the different strategies presented in section 3.3. We then present different experiments mixing the scenarios with the targeting strategies. We conclude this subsection by investigating the differences between look-up tree search and our proposed heuristic.

Different Targeting Strategies

In order to fully test NPC targeting actions, we implemented 4 different strategies inspired by modern FPS games. In each case, the selection is constrained to visible enemies, and in our experiments the human player uses the threat ordering strategy. Using the different presented strategies, designers can explore how they would influence combat output and thus understand better how different player might play their design.
3.5. Experiments

- *Closest* strategy will pick the closest enemy using Euclidean distance.
- *Highest Attack* strategy will pick the enemy with the highest attack.
- *Lowest health* strategy will pick the enemy with the lowest health.
- *Threat ordering* strategy will pick the enemy that has the highest priority according to equation 3.7.

Note that the closest strategy is strongly affected by the level design. Through careful placement of enemies, a designer could set up the level in a way that the companion choice will match other strategies, at least initially. We thus randomize enemy starting positions, and so closest acts more like a random selection in our different scenarios.

**Different Scenarios**

We designed four different scenarios, where each one is tailored for a specific strategies. These scenarios try to cover general instance family of combat games and each combination of scenario and strategy was tested in two environment levels:

- *Uniform* scenario is composed of six enemies with the same attack and health value. This is a classic combat scenario, with no surprises.

- *Boss* scenario is composed of five enemies with the same attack and health value, and a boss with high attack and health. This is the classic boss scenario, where minions are more of a nuisance than a threat.

- *Medley* scenario is composed of two enemies with low health and high attack value, two enemies with high health and low attack, and one enemy with medium-high attack and high health.

- *Tank* scenario is composed of five enemies with the same attack and health value, and an enemy with very high health value but only slightly higher attack.

- *Simple level* is an obstacle-free, open field with no geometry blocking NPC vision. This is an optimal situation for our threat ordering strategy, as it was designed with access to perfect information in mind.
3.5. Experiments

Figure 3.13: Top-down view and in-action playtime screenshot of the pillar level

- **Pillar level** is a high-occlusion environment, with pillars blocking vision. Vision constraints increase the problem complexity in ways not accounted for in Eq. 3.7, limiting target choices (in our experiments entities just pick a new target when the lose sight of their initial target), and making movement time a significant cost.

Figure 3.13 shows a top-down view and in-game, playtime screen-shots of the pillar level (simple level is the same with no pillars). Red circles surround the enemies, the blue circle encloses the human player and the green circle the companion.

**Threat Ordering Experiments**

In this section we discuss the different experiments conducted mixing our different scenarios and targeting strategies in order to gain better understanding of the latter. A final set of experiments was also done having the companion making the same target choices as the player (**Mimic Behaviour**), as the player uses different target selection strategies in the **simple level**. This experiment gives insights into how the result could be destructive when the companion does not make her own decisions.

For each combination of strategy, scenario, and level, and for the mimicking situation, we ran 31 simulations. This was sufficient to show trends in the data, while still resulting in a feasible experimental approach. From the data we plotted average final team health and standard deviation. Results can be seen in figures 3.14 to 3.25, where we plot the
3.5. Experiments

![Figure 3.14: Simple level uniform]

![Figure 3.15: Simple level medley]

player+companion health over the duration of combat (in seconds) for the different combinations.

**Simple Level** - Figure 3.14 shows results for all the strategies given the uniform enemy scenario. Since the enemies are identical in terms of attack and health, any target is a good target. Therefore any strategy is good as long as the players do not deviate between the enemies. This scenario gives us a good baseline for variance and sanity check for our simulation. It is interesting to point out that with respect to our theoretical justification, all enemies would sit at one point in figure 3.7, emphasizing the lack of need for specialized strategies.

In the boss and medley scenarios, figures 3.17 and 3.15, highest attack outperforms lowest health as a strategy, and matches our threat ordering approach. This is not always an ideal choice, however, as shown in the tank scenario, figure 3.16. In this case highest attack is actually the worst strategy; it picks the tank (high health enemy) first because it also has slightly higher attack, and thus spends a long time receiving non-negligible damage from the other enemies. Threat ordering, as well as the lowest health strategy, do not fall into the same trap, prioritizing instead enemies that are more quickly eliminated and thus reducing total health loss.

**Pillars** - The addition of pillars to the level design reduces visibility, preventing entities from seeing all targets in general, and dynamically changing the set of available targets as
3.5. Experiments

entities move. In general having subsets of enemies adds more noise to the simulation, and results are largely similar to the simple level, but with larger variance. This is evident in the uniform scenario, figure 3.18, and especially in the medley experiment, figure 3.19. Most evident in the uniform scenario, however, is that total health tends to be higher in the pillar versions. This is due to the enemies queueing behind each other because of the limited room between pillars, thus reducing their visibility and allowing only a subset to shoot at the players. With less shots fired at them, overall player health ends up greater, and this argument holds for every pillar scenario.

For the tank level, figure 3.21, we see a small but interesting change in the relative
3.5. Experiments

![Figure 3.20: Pillar Level Boss](image)

![Figure 3.21: Pillar Level Tank](image)

difference between strategies. The highest attack strategy is still the worst, but the gap between it and other strategies is not as big as in the simple level scenario. Repeated occlusion in the pillar level reduces the ability of the companion to stay focused on their sub-optimal choice, ameliorating the otherwise negative impact of this strategy. This is further verified by measuring and comparing the number of times the companion targets an enemy with respect to the number of times she would target the ideal target for her strategy; in this case the companion is able to choose her intended but sub-optimal target only around 1/3 of the time.

**Companion Mimicking the Player** - Since optimal theoretical solutions suggest that concentrating attacks on one enemy is optimal, a trivial strategy for companions is to simply mimic whatever the human player does. The success of this approach, however, depends very much on the strategy the player chooses. The medley scenario, figure 3.24 betters our understanding of the context. With a more independent companion we had at least one player selecting an optimal choice, decreasing the impact of any wrong choices on part of the other player. With mimicking, however, a wrong choice ends up multiplying the negative impact, and sub-optimal strategies such as lowest health and closest end up with dramatically lower team health values. This is also evident in the highest attack strategy of the tank scenario, figure 3.25.

Of course when the player is an expert at picking the right target, mimicking performs
3.5. Experiments

Figure 3.22: Mimic Level Uniform  
Figure 3.23: Mimic Level Boss

Figure 3.24: Mimic Level Medley  
Figure 3.25: Mimic Level Tank
3.5. Experiments

well, as both players cooperate and use good strategies. However, given that this will also occur if the companion makes an independent choice to use threat order, and that will still imply generally better outcomes if the human player is not an expert, mimicking seems like an overall poor approach. We note that this is not necessarily the case for every game or game context, and mimicking has been explored and shown to an effective approach in more complex contexts where learning from the player is worthwhile, such as in some fighting combat games [SCD11].

Look-up Tree Search

The heuristics we examine offer simplicity and efficiency advantages over look-up tree searches, but even the overall best, threat ordering, due to its inherent abstraction is not always necessarily optimal. We thus also compare performance with search based targeting to see how far from optimal threat ordering ends up being. For this we used a non-graphical, discrete-time simulation, allowing us to explore a large number of scenarios, and avoiding any concerns with perturbing the real-time simulation. Note that even this reduced problem is still NP-hard as shown by Furtak et al. [FB10].

We ran 500000 simulations in the discrete world with 2 players against 2 to 5 enemies. We compared a full look-up tree search (the actual implementation uses depth first search, which is presented in chapter 2) with the discrete form of the threat ordering heuristic, given as equation 3.6. The players in this simulation have attack 3 and health 500 each, with enemy attack varying from 1 to 10 and health from 1 to 9. Although these are fairly arbitrary values, there exists 6 billion ways to arrange the enemies with a large variety of difficulty. Results are shown in figure 3.26. We can see that threat ordering finds the optimal solution 50% of the time, is usually within around 1% of optimal, and never results in a total team health less than 8% of the optimal.

Figure 3.27 compares the behaviour of highest attack, lowest health, and random targeting strategies to threat ordering at each trial; this experiment used 50000 simulations, and 2–10 enemies (other parameters are the same). Note that in this discrete simulation we do not represent geometric constraints, and so random replaces closest. In no cases did these two strategies exceed threat ordering, but highest attack is clearly the better of the two,
3.5. Experiments

Figure 3.26: Cumulative histogram, showing how close to optimal threat ordering performs (discrete context).
Figure 3.27: Cumulative histogram, showing how highest attack, lowest health, and random targeting fare in comparison to threat ordering. Note that the $x$-axis scale differs from figure 3.26.
3.6. Chapter Conclusion

matching threat ordering about 25% of the time, and being within 50% of threat ordering over 97% of the time. Lowest health, however, only barely improves over random.

These discrete, analytical results largely mirror the results shown in the more complex, real-time data given in figures 3.14 to 3.25. In general, focusing on high attack enemies is most important, eliminating weaker individuals is next, and a weighted combination of these results is close to optimal prioritization. Physical proximity has relatively little relevance, although this is likely also an artifact of our combat simulation; as future work it would be interesting to see how close-combat versus distance weapons alter the weighting of proximity.

3.6 Chapter Conclusion

In this chapter we explored an adaptive model for real-time companions that takes the player’s game experience into consideration when making decisions. We showed that this particular model was able to temper the player’s game intensity level when compared to a non-adaptive companion; the resulting gameplay then better respects flow, providing challenge without the anxiety associated with long period of high intensity. We demonstrated that the effect is similar in scale to the use of a more traditional approach to adaptivity based on dynamic difficulty adjustment, but has additional advantages in being both orthogonal to DDA, and of allowing for better narrative justification of any adaptivity.

Adaptivity might not be enough to offer a functional experience for the player, specially for combat situation. Optimal solutions to enemy target selection in combat games are complex, and ideally solved through expensive state-space searches that are not practical in game contexts. Designers thus frequently resort to simple, but fast and easy to compute heuristics such as choosing the closest enemy, strongest enemy, mimicking the player, and so forth. In this work we explored and compared several such common heuristics, showing that a slight variant (threat ordering) can be mathematically justified, and also performs notably better than other simple heuristics in realistic simulation. We also compared this result to a simplified, but exhaustive state space analysis to verify that this
approach is not only relatively better, but also demonstrably close to the theoretical optimum. Understanding and validating these kinds of targeting heuristics is important in terms of building interesting and immersive gameplay where companions behave sanely and can perform effectively.

Implicit in our approach has been the use of mechanized gameplay, simulating player behaviours (as well as NPC) in order to evaluate our solutions. This avoids the need for human evaluation, enabling much more rapid and easily quantified interpretation of the results, and so greatly simplifying the design process. In the following chapters we are going to investigate different game representations and search algorithms for game design purposes, which further, and more generically enables automation in design evaluation. We will argue for different qualities an algorithm should have to be used as a game design tool. In a later chapter we also discuss the similarities and differences in pathing solutions between AI and human players.
Chapter 4
Game Solvers

While implementing and designing an NPC that could take upon the role of the player, we noticed that little research was done in the field of simulating player testers. This chapter explores different candidate algorithms for simulating players solving game problems such as reaching the level exit. This effort falls into the research category of constructing game design tools which offer feedback to game developers, also denoted mixed initiative [NS97]. In our case, the feedback is composed of game solutions from start to finish; in chapter 7 we shall explore other kinds of feedback that solvers allow.

In this chapter, in order to understand and explore the design space of any game genres, we first develop a general state space model that can encompass a variety of them. Using this description, we present different search algorithms that can then be used to discover potential solution paths players may choose. We first describe the state space, followed by a general discussion about search process. We then present the following search algorithms A*, Monte-Carlo Tree Search (MCTS), and Rapidly-exploring Randomly Tree search (RRT). This set of search algorithms is implemented with the purpose of being used offline by game-designers to verify the intent of their designs. We also discuss how to present and analyze solutions found by these solvers.


4.1 State Space

This section refreshes and extends the notation that will be used through the next few chapters of this thesis. Please refer to chapter 2 for an in-depth discussion. Generically, we are interested in finding paths from a starting position to a particular goal within a world, including player and enemy positions, states, and other changing aspects of the game context. The search processes use the following tuple,

\[ \langle \Sigma, \Sigma_{\text{obs}}, \Sigma_{\text{free}}, \sigma_{\text{init}}, \Sigma_{\text{goal}}, A, \mathcal{F} \rangle \]

The set of possible game states is thus denoted as \( \Sigma \), a singular state within this space is \( \sigma \), where \( \sigma \in \Sigma \). We use \( \sigma_{\text{init}} \) as our starting state, and \( \sigma_{\text{goal}} \) or \( \Sigma_{\text{goal}} \subset \Sigma \). We then denote a path, \( p \), as a sequence of single states, \( p(i) = (\sigma_0, \sigma_1, \ldots, \sigma_i, \ldots, \sigma_{n-1}) \), where \( \sigma_0 = \sigma_{\text{init}}, \) and \( \sigma_{n-1} = \sigma_{\text{goal}} \) or \( \sigma_{n-1} \in \Sigma_{\text{goal}} \). We use the function \( p(i) \) to access any a state position in the sequence.

In order to represent a game level state space, such as a specific level a designer is working on, we need to carve out parts of \( \Sigma \). Typically not all positions in a geometric space are valid/reachable. We denote the state space taken by the level obstacles as \( \Sigma_{\text{obs}} \) and assume it is constructed from a set of simple polygons, \( \{P_0, \ldots, P_{n-1}\} \), where one simple polygon is composed from a sequence of points in our 2D state space, \( P(i) = (\sigma_0, \ldots, \sigma_i, \ldots, \sigma_{n-1}) \), such that any pair \( (\sigma_i, \sigma_{i+1}) \), as well as \( (\sigma_{n-1}, \sigma_0) \) is a straight and non-intersecting line segment that are joined pair-wise. Everything falling within that shape is then considered unreachable by the player. The feasible state is defined as,

\[ \Sigma_{\text{free}} = \Sigma - \Sigma_{\text{obs}} \]

In order to explore this reachable space, we define a set of game-specific actions, \( A \), where \( a \in A \) is an action that can be applied to a state using an update function, \( \mathcal{F} \), specifically \( \sigma' \leftarrow \mathcal{F}(a, \sigma, \Sigma_{\text{free}}) \). The application of an action \( a \) changes the state \( \sigma \) to \( \sigma' \). For example, the jump action in a 2D platformer influences the vertical dimension representation. In case the action does not influence the state, the argument state is then returned.

\[ ^{1} \text{This can also be achieved in higher dimensions} \]
4.2. Search Process

Using these definitions the rest of this chapter investigates how search algorithms can use it to compute paths from $\sigma_{init}$ to $\Sigma_{goal}$. Please note that the following presented solvers present different data structures and vector manipulations necessary for computing paths.

4.2 Search Process

This section highlights different challenges that arise in computing a valid path from a starting position to the goal. Let’s use a simple forward search algorithm also known as breadth first search, shown in algorithm 2 and used as a means to explore $\Sigma_{free}$ and introduce the concept of search. The algorithm simply returns $true$ if a path exists or $false$ otherwise. The algorithm starts by initializing 2 data structures, on lines 2 and 3, to keep track of the search process; $Q$ is queue which handles states to be processed—if $Q$ is implemented as a list (first in first out) the algorithm is equivalent to depth first search—as arguments and $E$ is a simple set that keeps track of already explored states. The search process simply pops a state, $\sigma$ from $Q$, and processes it. The following execution is repeated until $Q$ is empty. We apply each available action to the current state (line 10) and process the resulting state, $\sigma'$. We first check if it has reached the goal, and if so the algorithm halts and returns $true$. If the state $\sigma'$ has never been seen before, we add the state to the visited states set (line 12) and to the queue, $Q$, to be extended. On the other hand, if it was already processed, we have to resolve the collision, for example by simply ignoring it, although in practical and real life implementations more thoughts have to be put into handling these state duplicates, as we shall discuss. If we exhaust the states in $Q$, it was impossible to find a path from $\sigma_{init}$ to $\Sigma_{goal}$ thus the algorithm returns $false$.

There are various limitations and concerns that affect this search process. Notably, the search process assumes discrete state transitions, and so approximates the original continuous space; for games this is not usually a major concern, although we will need to consider this further, especially when we address the impact of physics in platformers games, in chapter 5. More importantly, search time is a major factor—large and complex game spaces mean an exhaustive approach can be expensive to implement, and so search is typically limited by a state search budget, expressed either in terms of resources (nodes expanded), or
4.3. A*

In computer science A* is mainly used as a graph traversal algorithm and it is well known for its efficiency and accuracy [DSSW09]. A* is an extension of Edsger Dijkstra’s 1959
4.3. A*

graph search algorithm [Dij59], and it was first proposed in 1968 by Hart et al. demonstrating the incorporation of heuristic information into a formal mathematical theory of graph searching [HNR68]. The rest of this section presents the algorithm in the context of our notation and a trivial example from our platformer domain.

Algorithm 3 outlines the basic A* search procedure. A* starts by creating two sets, open and close, where the former is used to keep track of the states which need to be evaluated and the latter tracks the processed nodes. Using these definitions, the open set is initialised with the initial position, $\sigma_{init}$ and close with the empty set (line 2). A* prioritizes states that are closed to the starting state ($\sigma_{init}$) as computed by $g$ and states that are heuristically close to the goal region ($\Sigma_{goal}$), as computed by $h$, with the two distances summed by $f$. The algorithm selects the state with the lowest $f$ from the open set (line 5) to be processed next.

In order to extent the search, the method GETNEIGHBOURS (algorithm 4) is passed $\sigma_{low}$ as argument. The method applies each action stored in $A$ to the state passed by argument. If the resulting state, $\sigma_{action}$, is within the feasible state space, $\Sigma_{free}$, the state is stored. Moreover we update the resulting state parent field to $\sigma_{low}$ which is essential to retrieve the final path. The method finally returns all states from which was applied a valid action. At this point the algorithm evaluates each returned states stored in $\Sigma_{neighbours}$ (line 11 to 27 of algorithm 3). If the evaluated state, denoted $\sigma_{neighbour}$ in algorithm 3, falls within the goal region we then return a path from that node using algorithm 5. In this context the method PATH computes a sequence of states, $p$, representing a path of $n$ states from $p(0) = \sigma_{init}$ to $p(n - 1) = \sigma_{n-1}$, such that $\sigma_{n-1} \in \Sigma_{goal}$. If the evaluated state is not part of the goal region, its heuristic search fields ($g$, $h$ and $f$) are updated. The first field, $g$, represents the distance from the evaluated state to its parent. The distance is calculate using the distance method, $d$, that takes as arguments two states and returns a numerical value. For example, if our world is a 2 dimensional grid, we could use the following Manhattan distance [MSWH11]\(^2\),

\[d(\sigma_0, \sigma_1) = |x(\sigma_0) - x(\sigma_1)| + |y(\sigma_0) - y(\sigma_1)|\]

The second field to update is our heuristic distance to the goal, denoted $h$ and is computed

\(^2\)Please note that this distance is given as an example and it is not used in the different implementations presented in this thesis.
Algorithm 3 A*

```
procedure A*(σ_init, Σ_goal, Σ_free, A, budget)
    close = {}, open = {σ_init}, m ← 0
    while open ≠ {} and m < budget do
        m ← m + 1
        σ_low ← LOWEST(open)
        open ← open \ {σ_low}
        if σ_low ∈ Σ_goal then
            return PATH(σ_low)
        end if
        Σ_neighbours ← GETNEIGHBOURS(σ_low, Σ_free, A)
        for each σ_neighbour ∈ Σ_neighbours do
            parent(σ_neighbour) ← σ_low
            g(σ_neighbour) ← g(σ_low) + d(σ_low, σ_neighbour)
            h(σ_neighbour) ← d_h(σ_neighbour, Σ_goal)
            f(σ_neighbour) ← g(σ_neighbour) + h(σ_neighbour)
            σ_open ← GETIDENTICAL(σ_neighbour, open)
            σ_close ← GETIDENTICAL(σ_neighbour, close)
            if σ_open ≠ nil and h(σ_neighbour) > h(σ_open) then
                open ← open ∪ {σ_open}
                continue
            end if
            if σ_close ≠ nil and h(σ_neighbour) > h(σ_close) then
                close ← close ∪ {σ_close}
                continue
            end if
            open ← open ∪ {σ_neighbour}
        end for
        close ← close ∪ {σ_low}
    end while
    σ_best ← LOWEST(open)
    return PATH(σ_best)
end procedure
```
4.3. A*

Algorithm 4 Compute the neighbours of a state

procedure \textsc{GetNeighbours}(\sigma, \Sigma_{\text{free}}, A)
\begin{algorithmic}
\State \Sigma \leftarrow \{\}
\For {each \(a \in A\)}
\State \sigma_{\text{action}} \leftarrow \mathcal{F}(\sigma, a)
\If {\sigma_{\text{action}} \in \Sigma_{\text{free}}}
\State \Sigma \leftarrow \Sigma \cup \{\sigma_{\text{action}}\}
\State \text{parent}(\sigma_{\text{action}}) \leftarrow \sigma
\EndIf
\EndFor
\State return \Sigma
\end{algorithmic}
end procedure

Algorithm 5 Path algorithm to compute a path to a node

procedure \textsc{Path}(\sigma)
\begin{algorithmic}
\State \(p = \{\}\)
\State \(i = 0\)
\While {\sigma \neq \text{nill}}
\State \(p(i) \leftarrow \{\sigma\}\)
\State \(\sigma \leftarrow \text{parent}(\sigma)\)
\State \(i \leftarrow i + 1\)
\EndWhile
\State return \(p\)
\end{algorithmic}
end procedure
4.3. A∗

using \( d_h \) (line 14). This function can take multiple forms, it can be defined using the previously presented distance metric or it could encompass more calculation. The method returns an estimate of the cost distance to reach the goal. One interesting characteristic of A∗ assures that the returned path is optimal if the heuristic does not overestimate the distance to the goal. We then update the evaluation field, \( f \), which is used to determined the state with the lowest cost in the open set.

Before adding the discovered state to our open list for further exploration, we have to verify simple facts, such as did we meet or processed this state before, if so, is our new path to reach it better? We first retrieve the identical states from the open and close sets using the `GETIDENTICAL` method. For example, if our state space is composed of two elements, \((x, y)\), then the function returns the state in the set which has \( \epsilon \) similar values, \( d(\sigma, \sigma') < \epsilon \). If a state is found in the set passed as argument, the method `GETIDENTICAL` removes that node from it. Otherwise, when the set does not contain such state, it returns `nil`. On line 18 we consider the case where the path to reach our evaluated state, \( \sigma_{\text{neighbour}} \), is worse than a similar state found in the open set, \( \sigma_{\text{open}} \). If this is true, \( \sigma_{\text{neighbour}} \) is not added to the open set, and \( \sigma_{\text{open}} \) is put back in the open set. The same process is followed for the close set on line 23. In case the path to reach \( \sigma_{\text{neighbour}} \) is better than the one previously found, we add the state to the open set, line 26. This assures that some states can be reached from different set of actions and still be evaluated in the search.

When A∗ is applied to graph search, it might degenerate to evaluating all the graph nodes, and thus is only guaranteed to terminate if the number of nodes is finite. As with our basic search we thus add a search budget term, \( m \), which keeps track of the number of nodes explored. In the event that the budget is exceeded, the algorithm will return the path to the most promising node it reached so far. This allows us to construct a partial solution, and will be useful when discussing RRT (section 4.5).

Our final implementation of A∗ is similar to the Super Mario Bros controller presented by Baugarten for the first edition of the Mario AI benchmark competition [TKB10]. Since A∗ is mainly used to search graph structures (where line 10 of algorithm 3 returns the set of connected nodes to \( \sigma_{\text{open}} \)) using it in a continuous domain, such as implied by real-time control in platformer games poses some challenges. For example, the discretization of the continuous space can be achieved through adding a grid over the space, although
this has limitations for optimality. Nash et al. showed that the returned path from A* is only optimal within the discrete assumptions of their grid representation [NDKF07]. This suggests that a more optimal path might exist within the continuous space that is not found by A*. They focused their effort in “cleaning” the A* resulting path to reach an optimal solution within the continuous domain using an extended A*. The algorithm is denoted Theta* and it propagates information along grid edges without constraining the paths to its representation. Their approach uses line of sight algorithms to find better parent states than the one it came from. In light of this argument please note that the game representation used in this thesis does not always allow for such optimization.

Figure 4.1 shows a resulting path from an A* search for a platformer level where the brown square (on the left) has to reach the blue sphere (on the right). The path is represented by a brown outlined black line. The lime-green blocks represent obstacles, forming the limits and platforms of the level. The player can move left, right, and jump, with each possible action considered in a single A* search illustrated by a segment of a different colour: red for jumping, blue for wait, green for left, magenta for right, yellow for jump-left, and white for jump-right. The collision between two states is done using 2 or 3 dimensions $\langle x, y \rangle$ or $\langle x, y, t \rangle$ using simple Euclidean distance, which is also used as our heuristic to reach the goal state. The 2 dimensional approach allows the search to focus on covering as much as possible of the Euclidean space, while the 3D version will explore time as well. The latter approach is useful when there is a significant time component to the level, such as the need to interact with a moving platform.

As we will show in our experimentation, chapter 5, the algorithm is fast and produces near optimal results. In the context of a level-testing tool, however, it is less interesting, as the deterministic nature of the algorithm does not give a designer a good perspective of the breadth of possible solutions that a level design affords. The following algorithms focus on that particular aspect.
4.4. Monte-Carlo Tree Search (MCTS)

Figure 4.1: A* debug view. The different colours represent different possible actions at each node; the brown square and blue sphere are respectively the initial and goal positions. The final path is defined by the thick brown line.

Figure 4.2: MCTS selection

Figure 4.3: MCTS extension
4.4 Monte-Carlo Tree Search (MCTS)

Much like A*, Monte-Carlo Tree Search (MCTS) is normally used to search graphs. In fact it is specialized in searching trees with large branching factors, like the game Go [BPW+12]. In this section, we first describe the intuition behind the algorithm and then later present the algorithm using our search space notation.

MCTS iteratively builds a search tree, where the root is our initial state, $\sigma_{\text{init}}$. Figures 4.2 through 4.5 depict the main components of MCTS algorithm. Specially for this algorithm we introduce a reward value to the tree node, denoted $r$. Every node in our tree has a reward value, which is used to guide the tree generation. At every iteration MCTS traverses the tree nodes following a heuristic path based on potential reward (figure 4.2). Upon reaching a leaf, MCTS expands it—except if it is terminal—(figure 4.3) and plays a random path from that node, choosing random actions (figure 4.4). Using the information we gain from the random path we update the node reward values which guide our initial traversal (figure 4.5).

Each node is described by the following tuple $(p, \sigma, children, a, r, v)$, consisting of a node parent $p$, its state $\sigma$, a data structure to store the children, the action $a$ used to reach it from the parent, its reward $r$, and how many times it was visited during the selection phase, $v$. Implementing the selection phase is not a trivial task, where we want to maintain balance...
between exploitation of nodes with high rewards (depth) and nodes with little information or which are less rewarding (breadth). The reward function also introduces challenges where we cannot assure a simulation that will terminate. In classical application of MCTS, e.g., solving GO games, the reward function is binary, win (1) and lose (0), and we are assured that the simulation procedure will reach a terminal state as there are a fixed amount of moves available.

We give an implementation of MCTS with algorithm 6. We start by processing states focusing on a promising branch, deviating to another branch when that may lead to a better result. This process is controlled by the TREEPOLICY method that traverses through the tree, initially invoked on line 6. The first step is to make sure that each available action was applied to the state, and if not, to apply an available action and evaluate the result based on a suitable reward function. During the selection phase, if more than one node is given similar weight, we uniformly sample one of them. In standard implementations of MCTS, when a new node is selected the next step is to randomly select child-nodes (random walk) until the process reaches a terminal state (leaf in a tree) or a certain depth; the reached node is then evaluated and reward back-propagated. This random walk assumes that each action taken brings the state of the game towards termination. In the domain in which we are interested, it is hard to make such an assumption as a random walk of different actions might make the player move away from its goal. Hence, we simply evaluate a newly added node to the tree using normalized Euclidean distance and back-propagate it upwards. This step increases our knowledge about the search with data such as the number of times a node was visited and its reward. While testing our final MCTS implementation we noticed that this approach was superior to the random walk.

Once all available actions have been applied to a state, a child is selected based on its reward value, $r$, and the number of times it was visited, $v$. In our implementation the selection phase uses the upper confidence bounds applied to tree (UCT) introduced by Kocsis and Szepesvári [KS06]. UCT is used as an estimation error for different states, where the algorithm balances between testing current best alternatives and current suboptimal subtrees. This assures that better moves are not pruned out because of early estimation errors. This problem formulation is know as the exploration-exploitation dilemma and its simpler form shows up in the multi-armed bandit problem. The multi-armed bandit problem
4.4. Monte-Carlo Tree Search (MCTS)

Figure 4.6: MCTS debug view showing a dark green path from the dark green square to the green sphere. The different colour segments represent different possible actions from the search tree.

consists of maximizing profit of $K$ gambling machines, from which we keep the observed reward $X_{it}$, where $i = 1, \ldots, K, t \leq 1$, and each machine has independent payoff distributions. Using the upper confident bound (UCB), a machine is selected at time, $t$, using the average rewards, $\overline{X}_{it}$, as follow,

$$M_t = \arg\max_{i \in \{1, \ldots, K\}} \left\{ \overline{X}_{i,T_i(t-1)} + \sqrt{\frac{2\ln(t-1)}{T_i(t-1)}} \right\}$$

where $T_i$ returns how many time a machine was played at time $t$ [ACBF02]. Theoretically the probability of choosing the optimal action—winning strategies—converges to 1 as the number of samples grows to infinity. In MCTS each tree node is treated as a separate multi-arm problem using UCB. This step is extremely important as it allows the algorithm to avoid local minimum or maximum. At last, this algorithm is repeated until the resource budget is exhausted or the goal node is found.

Using MCTS in a continuous domain poses challenges. The search process does not take into consideration the proximity to other states as A* does, and naively implemented can sometimes over-search a given state-space location. To avoid this, we added a simple data structure (grid) that keeps track of how many times a grid position was visited. Figure 4.6 shows the MCTS debug view from our Unity 3D implementation—using the same
4.5. Rapidly-exploring Random Tree

Segment colouring for actions as figure 4.1—where in the background we can see the grid view as a red heatmap. As the red intensity increases so does the number of search state in that location, e.g., in figure 4.6 the north part of the level was the most searched. Jacobsen et al. also independently find a similar solution while applying MCTS to the Mario Bros. platformer game for using MCTS with a continuous domain [JGT14]. In order to keep the reward between the 0 and 1 interval the algorithm ranks nodes by the inverse square of their Euclidean distance to the goal, and also penalize nodes where the player dies or which end up being in over-searched positions (lines 18–20).

The MCTS search has some stochastic elements, and thus can be invoked multiple times to find different solutions. This leads to a better understanding of the space of possible solutions for game designers, with the benefit that the search is still very directed, and so the paths found do not wander around the goal.

4.5 Rapidly-exploring Random Tree

Rapidly-exploring Random Tree (RRT) is a well known pathing algorithm in the robotics domain, popular for its suitability to high dimensional and continuous state spaces. In our case, and much like MCTS, the heuristic and stochastic nature of RRT also allows for a broad exploration of possible solutions, more appropriate for modeling potential player behaviours, as we will explore further in subsection 6.4, and in chapter 8.

The intuition behind RRT is to build a search tree of the search space using sampling methods. For example, let’s define \( \Sigma \) as a simple square polygon and \( \Sigma_{\text{free}} \) as the white colour filled polygon (with holes) as seen on figure 4.7. The goal is to find a path from \( \sigma_{\text{init}} \in \Sigma_{\text{free}} \), represented by a green circle to \( \Sigma_{\text{goal}} \subset \Sigma_{\text{free}} \) where the latter is actually a circle within the valid search space and is represented by an unlinked red node in figure 4.7.

For this algorithm we define a tree, \( \Upsilon \), where a node is composed by the following tuple \( (p, \sigma, a) \) where \( p \) defines the parent pointer, \( \sigma \) its search state and \( a \) is the action applied to its parent state to reach it, such that if we have a node, \( v, f(a(v), \sigma(p(v))) = \sigma(v) \). The search tree is represented by purple segments and red circles in figure 4.7. The algorithm consists of sampling \( \Sigma \) and finding the closest tree node to the newly sampled point. The
4.5. Rapidly-exploring Random Tree

Algorithm 6 MCTS

\begin{algorithm}
\begin{algorithmic}
\Procedure{MCTS-UCT}{$\sigma_{\text{init}}, \Sigma_{\text{goal}}, \Sigma_{\text{free}}, A, t, \text{budget}$}
\State $\kappa_{\text{root}} \leftarrow \text{CreateNode}(\sigma_{\text{init}})$
\State $i \leftarrow 0$
\While{$i < \text{budget}$}
\State $i \leftarrow i + 1$
\State $\kappa \leftarrow \text{TreePolicy}(\kappa_{\text{root}}, A)$
\If{$\text{State}(\kappa) \in \Sigma_{\text{goal}}, t$}
\State \Return $\text{Path}(\kappa)$
\EndIf
\EndWhile
\State $\epsilon \leftarrow \text{REWARD}(\text{State}(\kappa), \Sigma_{\text{goal}})$
\State \Call{PROPAGATE}{$\kappa, \epsilon$}
\EndProcedure
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\begin{algorithmic}
\Procedure{REWARD}{$\sigma, \Sigma_{\text{goal}}$}
\State $v \leftarrow \frac{1}{\text{Dist}(\sigma, \Sigma_{\text{goal}})^2}$
\If{$\sigma \in \Sigma_{\text{dead}} \text{ or } \text{GridCount}(\sigma) > x$}
\State $v \leftarrow -\infty$
\EndIf
\State \Return $v$
\EndProcedure
\end{algorithmic}
\end{algorithm}
4.5. Rapidly-exploring Random Tree

The second step is composed of growing a motion segment from the closest node to the sample node. The process of sampling and growing is repeated until a path is found as shown in figure 4.8, where a path is shown by the green segments and blue nodes, or our search budget is reached.

Algorithm 7 gives an overview of our basic RRT implementation which is highly inspired by Lavalle’s description [LKJ00]. The algorithm starts by initializing a tree structure, $\Upsilon$, in order to keep track of the search process. While a resource (time, tree-size) budget is allowed, we randomly sample the collision free state space and extend the search tree by connecting new points to existing ones. Our baseline approach involves sampling only the Euclidean dimensions of $\Sigma$ and then constructing the rest of the sampled state based on the previous state of the connecting point. Connections are made by finding the closest node in our tree structure to our sampled state, measured in terms of Euclidean distance in the plane (line 7).

This Euclidean distance measurement implies a direct line motion, which is not always possible to accomplish. For example, in the case of a platformer game, the connection...
4.5. Rapidly-exploring Random Tree

Figure 4.8: An RRT path solution (green segments and blue nodes) with its search tree (purple segments with red nodes); note that the search was conducted using three dimensions e.g., $x, y$ and time, thus segments can cross over each other.
4.5. Rapidly-exploring Random Tree

Algorithm 7 RRT

procedure RRT(σ\text{init}, Σ\text{goal}, Σ\text{free}, budget)
    \[\begin{align*}
    i &\leftarrow 0 \\
    \text{Init}(Υ, σ\text{init}) \\
    \text{while } i < \text{budget do}
    \end{align*}\]
    \[\begin{align*}
    5: \quad &i \leftarrow i + 1 \\
    \quad &σ\text{rand} \leftarrow \text{Sample}(Σ\text{free}) \\
    \quad &σ\text{near} \leftarrow \text{Nearest}(σ\text{rand}, Υ) \\
    \quad &σ\text{motion} \leftarrow \text{Motion}(σ\text{near}, σ\text{rand}, Σ\text{free}) \\
    \quad &\text{if } σ\text{motion} \notin \text{nill then}
    \quad &\quad Υ \leftarrow (σ\text{near}, σ\text{motion}) \\
    \quad &\text{if } σ\text{motion} \in Σ\text{goal then}
    \quad &\quad \text{return Path}(Υ, σ\text{motion})
    \quad &\text{end if}
    \quad &\text{end if}
    \text{end while}
    \text{end procedure}
\]
between an existing point in the tree and a newly sampled point must respect the game physics, such as gravity pulling the player towards the ground. To model this we thus use a motion planner (line 8), in our case based on A* (section 4.3) or MCTS (section 4.4), to find a subpath from the nearest node to the randomly sample node, or as close as we can get—in this way we ensure that the tree grows and that all newly added points are actually reachable. If and once the goal region is reached, a path back to the origin is traced in the tree, giving us a final solution.

Figure 4.9 shows a visualization of the RRT tree in the platformer domain. The edges are indicated by red lines, sampled points by small green dots, and the actual tree nodes added by grey dots. For this particular example, we used MCTS as a local planner, thus explaining the jumpy resulting path. Also, RRT does not generally result in optimal solutions. We consider this beneficial, as this process allows the game designer to explore the variety of different possible paths players may take to solve the level. With a few changes to the algorithm, however, and enough computation time, it would be possible to show all reachable states, as shown by Morgan et al. [MB04]. Combined with clustering, it is also possible to use this technique to create probabilistic road-maps, such as shown by Bauer and Popović [BP12]. RRT, as presented, returns suboptimal solutions, where optimality is defined by the best possible path given an evaluation function, e.g., the fastest path to reach a destination (where we want to minimize time taken)\(^3\). Karaman and Frazzoli showed an extension to the RRT algorithm presented in this chapter, denoted RRT\(^*\), to assure optimal pathing solutions. The extension replaces the RRT search tree with a graph that is restructured every time a new valid sample is added, keeping the motion within the graph optimal [KF11]. The restructuring process is similar to the tree rewiring of A\(^*\)(lines 18 through 26 of algorithm 3).

### 4.6 Visualizing

MCTS and RRT have the capacity to produce multiple different game solutions which introduces the challenge of visualizing them. Communicating the output from the search

\(^3\)We believe this is a strength of RRT as game solver for game design.
4.6. Visualizing

Figure 4.9: The debug view for the RRT with 20 allowed actions for local search using MCTS as motion planner. The large blue and green spheres are the initial and goal positions respectively. Green nodes represent the sampled states, while grey nodes are part of the tree structure, linked by the red segments. The thick brown line shows the path found.

algorithms is a key to building a successful game design tool. Thus the presented visualizing tools of this section focus on depicting major trends in the solutions.

In normal usage of such a tool, a designer will be interested in visualizing thousands of solutions at the same time. This section addresses the problem of understanding thousands of different solutions to a level using different tools. One simple approach would be to draw each path with a single colour segment as seen by figure 4.10. This approach scales poorly, however, and large sets of different paths make it hard to see major solution trends. We thus also offer different approaches based on heat maps that show aggregate behaviours, using colour intensity to indicate frequency of state repetition. We first focus our effort onto projecting the time dimension as 2D heatmaps. We also explore aggregating methods which consider time, as 3D heatmaps.

4.6.1 2D Heatmaps

Given a set of paths $P$, we can draw a projection of each $p \in P$ onto the plane $(x, y)$. When $|P|$ is quite large, however, information on relative density is easily lost. We thus construct a heat map by discretizing the 2D game world into cells, giving each cell a colour
4.6. Visualizing

Figure 4.10: A 1500 platformer paths all drawn with a single colour
4.6. Visualizing

Figure 4.11: A 1500 platformer paths movement heatmap
that is a function of the proportion of paths that crosses them. Let $u$ be a 2-dimensional integer array, each cell of which contains a count of the number of paths in $P$ that pass through it. We then compute $\text{Dark}(x,y) = \frac{u[x,y]}{\max(u)}$, where $\max(u)$ is a function that returns the highest value in $u$. This view gives a global vision of the paths and helps designers to see places where players more or less frequently go. Various parameters can be used to control the visualization, and both linear colour mapping (figure 4.12) and logarithmic maps, see figure 6.9 (page 159) can be useful. A limitation of this approach of course is that by aggregating all results the choices made within specific paths are no longer obvious.

### 4.6.2 3D Heatmaps

A 2D heatmap aggregates path data over the entire span of time. It is also possible to base the aggregation on the portion of each path found within a specific span (or subset) of time, and so allow a designer to more easily follow path progression. For this we simply use a 3-dimensional array instead of a 2-dimensional one, with the third dimension mapping to some discretization of time. A heat map can then be computed and presented for each slice, showing the state of paths within a given time span. Moving or animating between slices then lets the designer observe how paths change and progress over time.

### 4.7 Chapter Conclusion

This chapter presented the key search algorithms used in the rest of this thesis. We discussed the general search process using a generic depth first approach. We then presented three different search algorithms, $A^*$, MCTS and RRT, which have different strengths such as search efficiency or randomness. For each algorithm we presented small examples and discussed their capacity to be used during the game design process. We also explored different means to visualize the traces produce by solvers, where we focused our effort onto using heatmaps.

In the following chapter we are going to explore the application of the presented solvers in better depth for the platformer genre. We will compare different performance metric values for each solver and discuss the different needs for the game design purpose.
Figure 4.12: Heat map movement of 1500 RRT paths, with darker indicating more paths. The player starts at the blue sphere on the left and must reach the green sphere on the right, avoiding enemies (yellow spheres). Dashed lines show enemy movements, here consisting of 2 cameras panning left/right, and one straight-line guard patrol.
The platformer genre is popular among indie game developers, mainly due to the easy access enabled by tools like Unity 2D or GameMaker. Designing meaningful levels for platformers which are challenging and interesting for players is not a trivial task, and it is important to avoid bad designs where players could make use of undesirable short-cuts to solve levels, or which result in overly obscure or difficult solutions. Verifying game solutions, however, is difficult for small-scale developers without the resources to perform large-scale human testing.

This chapter introduces special application of the previously presented search algorithms in the context of platformer games. We first discuss the platformer state space and its particular features. We then present a comparative study evaluating the effectiveness of the different algorithms on representative platformer levels. This chapter concludes on a discussion about the usability of the different game solvers for the game design process.

### 5.1 Platformer State Space

This section represents the state space representation used for the platformer domain from chapter 2. Our representation is aimed at the classical platformer game genre, where the game level is fundamentally a 2-dimensional, Euclidean space. The space is constrained by screen boundaries and physical obstacles, such as shown by the green rectangles in figure 4.9 (page 128). A designer may also add to her level various other kinds of obstacles,
5.2. Experiments

such as saws, spikes or other kinds of death obstacles that kill the player right away, shown as red rectangles in figure 5.5 (page 141). Moving platforms are another common feature, and may repeatedly move horizontally or vertically; in figure 5.7 (page 143) platform movement is indicated by the grey arrows.

Within the level the player has basic commands to move left-right and jump; we also incorporate double-jumps (allowing a second jump while in mid-air), as a popular, if less physically realistic behaviour as previously describe in chapter 2. Our physics model includes gravitational acceleration downward, but does not include player momentum on the horizontal axis—a player’s forward motion depends entirely on the current left/right key-press, and so may be changed arbitrarily and instantaneously, even while in the air—an approach commonly referred as air control by game designers. This gives players fine-grain and highly reactive control, as is common in platformer games. From this context we can build a formal model of the game state. As a refresher we represent our state space $\Sigma$ as a combination of subspaces:

$$\Sigma \subseteq \mathbb{R}^2 \times \mathbb{R}_{time}^+ \times \{0, 1, 2\}_{jump} \times \{0, 1, 2\}_{moving}$$

Where we encoded the 2D Euclidean space a player may occupy, a non-negative implicit time vector (essential for representing platform movement) a gravity vector to model falling velocity, a 3-valued domain to represent vertical activity as normal/falling (0), jumping (1) and whether a double-jump has been performed (2), and a 3-valued domain to represent motion, as either not moving, moving left or moving right. We designate the set of player actions defined by the game as $A$. In our case we have 6 actions, $A = \{jump, jump-left, jump-right, left, right, wait\}$. We then are interested in finding paths from $\sigma_{init}$ to $\Sigma_{goal}$ using any solver described in chapter 4.

5.2 Experiments

The algorithms described in the previous section were implemented as part of an open-source design tool within Unity 3D [BT14]. This section has for goal to first evaluate the performance of the three solvers presented in chapter 4 using different metrics such as computation time. We first start by introducing details about our implementation in Unity
5.2. Experiments

3D, followed by a discussion on the performance of each search algorithm. We also include discussion on the use of these solvers as part of the game design process.

5.2.1 Methodology

We implemented the different search algorithms as plug-ins for Unity 3D, which could be used in any Unity 3D projects. An important design concern in implementing such a tool in the editor of Unity is the loss of the physics simulator—although it is possible to move GameObjects around and check for collisions, the full, built-in physics is not available in editing mode, and thus we were required to implement our own physics simulator for gravity, drag, etc. For this we used a discretized simulation with limited collision resolution; for example, while the player is falling, the simulator applies gravity forces to the vertical velocity vector until the player collides with a platform, at which point movement stops. This limited physics adequately describes many platformer games, and so we leave more advanced collision handling to future work.

Our design allows individual control of the many parameters available to each search algorithm. It is also possible in our design to choose which motion planner (A* or MCTS) the RRT is going to use. Figure 5.2 shows some of the different parameters available, their impact, and to which search they apply; individual control of these values allows us to experiment with search behaviour within a wide range of parametrizations. Figure 5.1 presents some of the interface controls available to game developer. Some of these controls are shared by different search algorithms, such as the maximum number nodes allowed in a search.

5.2.2 Performance Tests

To evaluate the behaviour of the different search algorithms, especially with respect to their value in design evaluation, we implemented different levels that are meant to stress the algorithms in different ways. We first experimented on a few test levels to determine appropriate and practical parameter settings for each algorithm, and then explored algorithm success in terms of searching time, and success ratio, and number of states explored,
5.2. Experiments

Figure 5.1: The platformer tool interface as a Unity 3D editor panel. This gives the designer fine-grain control over different search options and algorithm choice.
5.2. Experiments

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Effects</th>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min distance</td>
<td>Minimum distance allowed between nodes</td>
<td>RRT, A&lt;sup&gt;*&lt;/sup&gt;</td>
</tr>
<tr>
<td>Max distance</td>
<td>Maximum distance allowed between nodes</td>
<td>RRT</td>
</tr>
<tr>
<td># nodes</td>
<td>Maximum allowed explorable nodes</td>
<td>RRT, A&lt;sup&gt;*&lt;/sup&gt;, MCTS</td>
</tr>
<tr>
<td>Time action</td>
<td>Duration of each action</td>
<td>A&lt;sup&gt;*&lt;/sup&gt;, MCTS</td>
</tr>
<tr>
<td>2 or 3 dimensions</td>
<td># of dimensions used in the state collision</td>
<td>A&lt;sup&gt;*&lt;/sup&gt;</td>
</tr>
<tr>
<td>Density</td>
<td>Grid size used by the reward function</td>
<td>MCTS</td>
</tr>
</tbody>
</table>

Figure 5.2: Parameters for the platformer solvers

as well as solution quality in terms of the resulting solution path length, and number of key
presses required (as a measure of player effort). These values are interesting as they allow
us to gain a better understanding of the different qualities of each search algorithms. For
each presented level we ran 1000 searches, using a 16GB Intel-i7 machine, and Unity 3D
version 4.3.4f1. Numeric results for all tests are given in table 5.1. For our aggregated per-
formance test—which includes averages and one standard deviation—we focused on the
search time in ms, success ratio describing the percentage of path found out of the search
attempts, the time it took for successful searches to complete, the path length in frames, the
number of key pressed on the path found showing complexity of execution and the number
of states explored by the search process. We refer to different implementation of A<sup>*</sup> with
subscript defining the number of dimensions used for state collision, e.g., 3 refers to using
the planar coordinates and time dimensions.

5.2.3 Level 1

This level is designed to require the player follow a non-trivial path of jumping between
platforms, including some amount of vertical ascension as seen in figure 5.3. A<sup>*</sup> with 2
dimensions performs extremely well, directly heading to the goal, and with search time
5.2. Experiments

Figure 5.3: Level 1 outline where the player starts at the blue sphere and has to reach the green sphere.

Table 5.1: Level 1 search results. All times are in ms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Search time</th>
<th>Success ratio</th>
<th>Successful search time</th>
<th>Path length (frames)</th>
<th>Key presses</th>
<th>States Explored</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aᵰ₂</td>
<td>392.1 ± 7.3</td>
<td>1.00</td>
<td>392.1 ± 7.3</td>
<td>572.0 ± 0.0</td>
<td>17.0 ± 0.0</td>
<td>785 ± 0</td>
</tr>
<tr>
<td>Aᵰ₃</td>
<td>2251.7 ± 10.8</td>
<td>0.00</td>
<td>0.0 ± 0.0</td>
<td>0.0 ± 0.0</td>
<td>0.0 ± 0.0</td>
<td>0 ± 0</td>
</tr>
<tr>
<td>MCTS</td>
<td>1077.6 ± 246.7</td>
<td>0.97</td>
<td>1062.3 ± 230.1</td>
<td>656.9 ± 44.5</td>
<td>69.5 ± 10.1</td>
<td>4976 ± 1081</td>
</tr>
<tr>
<td>RRTₐᵰ</td>
<td>2514.7 ± 1441.5</td>
<td>0.96</td>
<td>2354.6 ± 1178.0</td>
<td>759.1 ± 75.8</td>
<td>54.8 ± 9.5</td>
<td>5009 ± 2369</td>
</tr>
<tr>
<td>RRTₐᵰ-MCTS</td>
<td>13410.9 ± 5921.6</td>
<td>0.84</td>
<td>11703.0 ± 4034.1</td>
<td>1692.9 ± 243.0</td>
<td>109.7 ± 16.7</td>
<td>21179 ± 6488</td>
</tr>
</tbody>
</table>

...dramatically less than the others such as seen in table 5.1. The Aᵰ with 3 dimensions was not able to solve this level as it reached the maximum allowed number of explored states before reaching the goal; through debug views we were able to see that inclusion of the time dimension resulted in the search failing to make level-related geometric progress as it over-searched the time dimension, see figure 5.4. The MCTS search took longer than 2D Aᵰ by a large factor, reflecting the greater cost inherent in MCTS of having to backpropagate rewards through the tree every time it expands the search.

Using RRT as a higher-level search adds a lot more “noise” to the paths: the average number of keys pressed implied by the found solutions is significantly higher than Aᵰ. This is expected, since our RRT does not use any heuristic to bias its search, and so explores...
5.2. Experiments

Figure 5.4: Level 1 showing the 3D A* debug view; it was not able to find a solution more states than the biased searches. RRT\textsubscript{MCTS} has the greatest diversity in this sense, although it is slow and suffers from a lower success rate; RRT\textsubscript{A*} seems to represent a better trade-off between breadth and performance while still including many random paths.

5.2.4 Level 2

This level provides two options to the player: going straight along the bottom of the level and avoiding the stomper, or climbing up the platforms and using the horizontal moving platform to reach the goal as seen in figure 5.5. The biggest challenge with this level is the time component: a player has to either wait for the horizontal platform to traverse the level or time her movement to avoid death by the vertical stomper.

In this case both implementations of A* did extremely well as seen in table 5.2. The A* using three dimensions found a marginally faster path, but it was also simpler in terms of keypresses, while the 2D version had to jump around in order to get the timing right. This level is well constructed for A* as the heuristic (Euclidean distance to the goal) leads the search almost directly to the goal. The MCTS search did surprisingly well, given that there is no explicit consideration of time in the search heuristic. The usage of RRT in this level, however, shows a limitation in our sampling heuristic: the tree structure tends
5.2. Experiments

Table 5.2: Level 2 search results. All times are in ms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Search time</th>
<th>Success ratio</th>
<th>Successful search time</th>
<th>Path length (frames)</th>
<th>Key presses</th>
<th>States Explored</th>
</tr>
</thead>
<tbody>
<tr>
<td>A$^*$$_2$</td>
<td>228.2 ± 5.1</td>
<td>1.00</td>
<td>228.2 ± 5.1</td>
<td>342.0 ± 0.0</td>
<td>12.0 ± 0.0</td>
<td>542 ± 0</td>
</tr>
<tr>
<td>A$^*$$_3$</td>
<td>487.1 ± 6.5</td>
<td>1.00</td>
<td>487.1 ± 6.5</td>
<td>340.0 ± 0.0</td>
<td>8.0 ± 0.0</td>
<td>1246 ± 0</td>
</tr>
<tr>
<td>MCTS</td>
<td>897.8 ± 326.9</td>
<td>0.82</td>
<td>791.6 ± 257.8</td>
<td>346.6 ± 33.7</td>
<td>24.3 ± 6.0</td>
<td>4530 ± 1468</td>
</tr>
<tr>
<td>RRT$_A^*$</td>
<td>1464.6 ± 1621.1</td>
<td>0.67</td>
<td>531.3 ± 800.2</td>
<td>333.4 ± 46.8</td>
<td>7.7 ± 6.4</td>
<td>1230 ± 1763</td>
</tr>
<tr>
<td>RRT$_{MCTS}$</td>
<td>11391.3 ± 5097.1</td>
<td>0.35</td>
<td>8167.1 ± 3873.7</td>
<td>904.7 ± 178.1</td>
<td>47.0 ± 15.4</td>
<td>15889 ± 6935</td>
</tr>
</tbody>
</table>

to reach a certain time configuration where the node closest to the moving platforms very likely leads to a dying position. Any nodes sampled nearby end up connecting to it and so lead to death, and this is evident in the low success rates for either form of RRT. Figure 5.6 shows an example of a configuration that makes the RRT fail at finding a path. Since our implementation does not take time into consideration when sampling, if the closest state, $\sigma_{near}$, to the sampled one, $\sigma_{rand}$, is not in a good configuration, e.g., the platform to jump on is at its farthest from $\sigma_{near}$ in the $(x, y, t)$ dimensions. Thus any motion from $\sigma_{near}$ to $\sigma_{rand}$ will result in a failure. We expect that a further tuning of the RRT, including a 3D sampling where we consider points in time as well as space, would produce better results, as we will explore in the following chapter.

5.2.5 Level 3

This level also gives two options to the player, going upwards and then right until reaching the end involves few time-based actions (and is the preferred path for both A$^*$s) as seen in figure 5.7. Alternatively, the player may move to the right and then ride the vertical platform to reach the goal.

The level figure is overlaid with a heat map of the RRT$_A^*$ paths results. As the heat map indicates, most of the RRT solutions follow the same path as the A$^*$ (purple path), climbing up and then going across the level. Some, however, either by falling from the top or by moving there from the start do end up using the vertical platform. In the numeric data we can see that A$^*$ did well, as the heuristic pushed the search to reach the top right as fast as possible. Both paths given by the two implementations of A$^*$ were similar, although the number of states explored is greater by a factor of almost six for the 3D A$^*$. MCTS
5.2. Experiments

Figure 5.5: Level 2. Black arrows indicate platform movements, and red regions indicate instant death locations. The top red area can be avoided by riding the platform moving horizontally, and the bottom stomper can be avoided by timing a dash below it.

Figure 5.6: RRT debug view of level 2 showing a tree configuration that leads to bad timing with both the top platform or the bottom stomper.
5.2. Experiments

Table 5.3: Level 3 search results. All times are in ms.

| Algorithm | Search time | Success ratio | Successful search time | Path length (frames) | Key presses | States Explored |
|-----------|-------------|----------------|------------------------|----------------------|-------------|******************|
| A\(^2\)   | 1128.0 ± 158.4 | 1.00           | 1128.0 ± 158.4         | 427.0 ± 0.0          | 23.0 ± 0.0  | 806 ± 0         |
| A\(^3\)   | 5710.1 ± 447.2 | 1.00           | 5710.1 ± 447.2         | 416.0 ± 0.0          | 25.0 ± 0.0  | 4771 ± 0        |
| MCTS      | 3672.1 ± 952.9 | 0.95           | 3593.8 ± 887.1         | 438.4 ± 34.0         | 36.9 ± 6.5  | 5203 ± 1109     |
| RRT\(_A^*,\) | 3122.4 ± 2563.5 | 0.69           | 1792.9 ± 1430.1        | 516.1 ± 56.9         | 33.5 ± 6.5  | 4116 ± 3080     |
| RRT\(_{MCTS}\) | 16690.3 ± 7365.7 | 0.49           | 11261.3 ± 5449.1       | 1001.9 ± 253.0       | 64.8 ± 15.6 | 21502 ± 9723    |

does not shine as much as A\(^*,\), being generally slower, but does have the useful ability to show more human-like (suboptimal) solutions. We note that MCTS also has a drawback in being a less “out of the box” solution, and required some amount of experimentation with parametrization, as well as a customized reward function in order to perform reasonably well. The random process of RRT shows very interesting results in terms of exploring the space; it tends to find highly variant solutions, and is the most effective algorithm for finding both routes to the goal in our final test. It is, however, by far the slowest, and success can depend strongly on the particular motion planner it uses to connect states. An A\(^*\) planner seems perhaps generally better, as it tends to ameliorate the randomness. In the following chapters we shall explore how RRT can be used efficiently used as a solver for stealth games.

5.2.6 Discussion

From these experiments we can see that A\(^*\) is fast, converging to a solution quickly, and is clearly the better approach if a high quality solution is required, which is not surprising. Incorporating the time dimension into the search can be helpful if the level has a strong timing component, but also greatly magnifies the state space and so requires more resources.

From a game-design perspective A\(^*\) is useful for game design analysis—demonstrating that a solution exists and finding a minimal solution. However, it is not sufficient, as it finds exactly one solution, and so is not helpful in exploring the potential space of solutions in a level. MCTS is then an attractive option as a search process known to be effective in complex domains, and which is also able to find different solutions with multiple searches but requires lots of manipulation to implement. From our short study it does not shine as much as A\(^*\), being generally slower, but does have the useful ability to show more human-like (suboptimal) solutions. We note that MCTS also has a drawback in being a less “out of the box” solution, and required some amount of experimentation with parametrization, as well as a customized reward function in order to perform reasonably well. The random process of RRT shows very interesting results in terms of exploring the space; it tends to find highly variant solutions, and is the most effective algorithm for finding both routes to the goal in our final test. It is, however, by far the slowest, and success can depend strongly on the particular motion planner it uses to connect states. An A\(^*\) planner seems perhaps generally better, as it tends to ameliorate the randomness. In the following chapters we shall explore how RRT can be used efficiently used as a solver for stealth games.
5.3 Chapter Conclusion

In this chapter we investigated the different qualities and implementations of different search algorithms for the game design purpose. From the performance analysis, $A^*$ showed the better results as expected, although it was limited in finding a breadth of different paths. Using $A^*$ as a local planner was useful for RRT as it was able to find a larger breadth of paths.

Heuristic search approaches have the important property of showing a range of solutions, giving more insight into the potential ways players may solve a game. Both MCTS and RRT are thus perhaps the most interesting search algorithms in terms of understanding player behaviour. In the following chapters we will focus our efforts on RRT exclusively, which although sometimes slower has advantages in simplicity of implementation and modularity, as well as its ability to easily include controllers (tree extension decision process). The following chapter presents the application of RRT to the stealth domain as well as a discussion comparing RRT solutions to human ones.
Chapter 6
Stealth Games

The requirement to move unseen through a level is a core mechanic of the stealth game genre; the problem also occurs in more combat-oriented games, where the existence of a stealthy path provides strategic choices to players, enabling them to reach better combat positions or avoid combat altogether. The existence of stealthy paths, however, depends on a complex interplay between enemy senses, locations, and movements, the structure and occlusions provided by virtual objects, and player starting and goal states. A designer thus needs to not only solve this problem, but also ensure that solutions continue to remain available and suitably challenging throughout the iterative design process typical of modern game development.

Based on the different evaluations from the previous chapter (chapter 5), this chapter explores how RRT can be used to search valid stealth paths, as well as exploring the humanness of the solutions produced. The introduction of such a tool allows designers to explore their level designs in terms of the existence and qualities of stealthy paths. This chapter discusses the specific state space used by our RRT solver with non-trivial visualization of the former, as well as presenting different case studies showing the usability of our computational techniques. We conclude our experiments with a human study comparing the humanness of RRT produced solutions, in which we also propose algorithmic improvements that reduce differences.
6.1 Stealth Games State Space

Our state space model for stealth games is based on simplification of a basic single-player stealth, FPS, or RPG level representation, consisting of a 3D world flattened to two dimensions, and wherein a set of AI-controlled $k$ enemies $E$ interact with a player over time. (Please note that this section re-presents some notation previously described in chapter 2.) When not affected by player interaction enemy movement is assumed to be deterministic, and thus we can retrieve an entity’s position and orientation as a simple function of time. This simple model encompasses the core structure of a wide range of modern games in our genre-space. We use the following state space for the stealth domain,

$$\Sigma \subseteq \mathbb{R}^2 \times \mathbb{R}_{\text{time}}$$

which is simpler than the platformer domain. Within this space we carve out the obstacles, $\Sigma_{\text{obs}}$, and the Field of View (FoV) occupied by the enemy, $\Sigma_{\text{FoV}}$.

The state space occupied by the enemies FoV is not as trivial to represent as the one occupied by the obstacles. In stealth games enemies typically patrol a given area using a pre-determined path through the level from an initial position to various points in the level space, and eventually returning to their origin to restart the patrol. For this we assume a function exists mapping time to a particular position in $\Sigma$ for each enemy, e.g., $\text{pos}_e : \mathbb{R} \to (x, y, \theta)$. Enemies have a fixed field of view (FoV) they use to detect the player, which can be defined as a cone, or more simply as an isosceles triangle, projecting forward from the enemy, and reduced appropriately by geometric occlusions. Figure 6.1 shows an example enemy’s FoV in orange and its construction as a function of $\Sigma_{\text{obs}}$ (represented as a white square), assuming an enemy cannot see through or behind an obstacle such as a wall. This FoV always exists at various locations depending on the position of the enemy. Note that this model of enemies and FoVs also accommodates static enemies, such as cameras, which are then simply enemies that are unable to actually move. A rotating camera with the same FoV as shown in figure 6.1, for example, can be modeled by applying a rotation matrix with step $t$ and angle $\theta$ to the points defining the basic outline FoV in two dimension. It is also possible to make the FoV length and angle vary as well by as describing it as a static function of other features of the domain, such as time or location, or probabilistically. We
6.1. Stealth Games State Space

Figure 6.1: The enemy’s (yellow circle) field of view (cropped orange triangle) with an obstacle (white square).

decided to use a small cone similar to figure 6.1 as a simple abstraction that allows for reducing the enemies FoV footprint onto $\Sigma_{free}$.

The goal of the player is then to find a path through the state space, avoiding both obstacles and all enemy FoVs. To define the resulting feasible space, we can construct $\Sigma_{FoV} \subseteq \Sigma$ as the union of all enemy FoVs over time, giving us a final search space defined by $\Sigma_{free} = \Sigma - (\Sigma_{obs} + \Sigma_{FoV})$. This is also the space where the player is free to move. The problem of stealth is purely a motion one, where the player has to avoid enemies’ FoV, thus we do not explicitly describe the action space and will leverage linear algebra to describe motion in the following section.

Path Finding

In order to find solutions in the stealth space we search $\Sigma_{free}$ for a path from $\sigma_{init}$ to a point in $\Sigma_{goal}$. For this we use the Randomly Rapidly exploring Tree algorithm (RRT), preferring it over a deterministic algorithm such as A*, as the randomness better simulates a range of possible player behaviours, and we are also able to easily include controllers that express different player's behaviour, as we will discuss in chapter 8. During preliminary implementation for the stealth domain, we found A* and MCTS were inefficient at solving
6.1. Stealth Games State Space

any stealth level since the branching factor was too large given our voxel representation, and for the reasons given in the previous chapter.

Our design uses the RRT implementation presented by algorithm 7 (page 126) with some changes. In the stealth case, motion in the 3D space representing geometry and time only has to respect a maximum velocity; that is, the angle created between the proposed connection segment with the \( \langle x, y \rangle \) plane of the existing tree point has to be larger than some threshold (maximum velocity). Note that an angle of \( \pi/2 \) is simply the player waiting, while an angle of 0 represents infinitely fast motion. We also have to assure that the motion stays within \( \Sigma_{\text{free}} \), e.g., the segment does not collide with any walls or enemies FoV. Thus, if a feasible motion exists—in algorithm 7 line 8—to connect \( \sigma_{\text{near}} \) to \( \sigma_{\text{rand}} \), the segment \( (\sigma_{\text{near}}, \sigma_{\text{motion}}) \) is added to the tree structure. The process is repeated until a state in \( \Sigma_{\text{goal}} \) is encountered, or a resource limit \( M \) is reached in terms of number of sampled states (or time). If successful, the path solution can be found as a walk through \( \Upsilon \) from \( \sigma_{\text{init}} \) to the discovered goal state.

Results from RRT in the plaformer domain (chapter 5) did not need much “cleaning” as the motion was from a local solver. For the stealth domain, since we are using segment motion, the resulting paths have a drunken behaviour as seen by the purple search tree in figure 6.2. In order to produce simpler paths, as seen by the green path in figure 6.2, we use algorithm 8. This process reduces the number of sharp turns often generated from the random search process. Starting at the first path state, \( i = 0 \), we apply the following. Given three points \( \sigma_i, \sigma_{i+1} \) and \( \sigma_{i+2} \) on a path \( q \), which are in path order, we check if the segment \( (\sigma_i, \sigma_{i+2}) \) is within the feasible space. In algorithm 8 we used the DOUBLENEXT method to get the \( i + 2 \) path state. In case the segment is feasible, we update \( \sigma_i \) next pointer to \( \sigma_{i+2} \), thus removing \( \sigma_{i+1} \) from the path \( q \), and keep \( i \) fixed and reapply the previous computation. Otherwise we increment \( i \) and repeat this process. The result is a more optimized and less meandering path, although this also has implications in terms of the humanness of the result, which we will describe in section 6.4.
Figure 6.2: Smoothing algorithm applied to a path solution, where the final path is represented with green segments and blue nodes, the purple segments show and red nodes show the search tree

Algorithm 8 Path smoothing method

```
procedure SMOOTH(q, Σ_free)

σ_eval ← first(q)

while σ_eval ≠ last(q) do
    while CollisionFree(σ_eval, DoubleNext(σ_eval), Σ_free) do
        Next(σ_eval) ← DoubleNext(σ_eval)
    end while
    σ_eval ← Next(σ_eval)
end while

return q
```

10: end procedure
6.2 State Space Representation

In order to gain a better intuition about the search process and the problem space this section investigates different state space visualization. By treating time as a third dimension we gain the ability to visualize both $\Sigma_{\text{obs}}$ and $\Sigma_{\text{FoV}}$ as surfaces in the space. This approach is not limited to the stealth domain even though it is the main focus of this section. The space taken by static obstacles is the easiest to represent: as an obstacle is in the same space at all times, it is drawn by extending the 2D footprint of the obstacle through the entire time dimension. Figure 6.3 shows two rectangular obstacles extended into rectangular prisms; movement is impossible for the player within these regions.

A player path can also be represented in this space. While conceptually a point, we draw the player position as a small circle for easier visualization, again extruding the area through time in order to represent the player’s history of movement. A non-moving player then becomes a simple cylinder, a moving player implies an angled cylinder, and limits on acceleration define the sharpness of the transition. Figure 6.3 shows a path from a player walking around the left obstacle, assuming instantaneous ability to accelerate (and thus sharp transitions between cylinder segments).

Representing $\Sigma_{\text{FoV}}$ results in the most complicated of our visualization objects. If the enemy is not moving then we can represent the behaviour as triangular prism extruded over time, such as in figure 6.4a. When the enemy moves, the prism is then tilted towards the direction the enemy goes, figure 6.4c. Rotations are described by a curved surface; figure 6.4b shows an example of an enemy performing full, 360° rotations in place (a rotating camera), producing a spiral-staircase effect. Note that we render these shapes as stacks of independent, triangular slabs in order to simplify how the shapes respond to obstacle occlusion (as per figure 6.1).

Using our aggregate heatmap method previously presented in chapter 4, (section 4.6) we can construct a probability density map depicting the probability of getting seen at any position in the level. This has the advantage of also indicating safe spots, as regions which receive no heat map colouration are never observed, and are thus areas that the player can spend arbitrary time within without the threat of discovery. Using such a tool a designer can build simple mental models of unseen movement. More generally, we can present a
Figure 6.3: A 2D game space (white rectangle) extruded through time (vertical dimension). A player path (composition of blue cylinders) moves around obstacles (purple prisms).
6.2. State Space Representation

Figure 6.4: Different enemy visualizations: a) waiting enemy b) rotating enemy c) moving enemy.
6.3 Experiments

probability distribution of getting seen, with locations of 0 probability indicating areas that are never observed. An example is shown in figure 6.5. In this visualization it is clear which grid squares the enemies may observe and which they may not. This gives the designer a good overall understanding of the play space, allowing her to determine whether the safe places are there on purpose or not.

6.3 Experiments

This section explores different use cases of our framework applied to the stealth domain. In order to test our search approach to the stealth genre, we implemented a plug-in tool for Unity 3D where we could easily design a level and use RRT to search that particular level. For this discussion we look at a simple example inspired by the creator of Thief in order to demonstrate the usability of such a tool, followed by an analysis of the first level of Metal Gear Solid 2 as an example of a more complex and real-game context. We follow this some data on our metrics, validating the specific metrics through a short human study, and then showing application of the distance controller.

Our Unity 3D implementation uses a voxel based representation as it allows for quick prototyping and ease for exploration. We leave as future work to implement the search using the continuous representation. Each discrete time×space coordinate maps to a discrete cube. We then construct a data structure with a specific voxel granularity (grid size) and a limit of the time dimension (time sample size). The cubes are then tagged as obstacle, free or FoV based on the level construction. This representation is then used to determine if a motion is valid or not.

6.3.1 Randy Smith’s design

In order to show how the RRT can be used in the game design process, we will analyse a proposed level design by Randy Smith—a well known stealth-game designer known for his work in the Thief series from Eidos Interactive. During his 2006 GDC presentation on stealth gameplay, he described a number of basic level designs for stealth [Smi06]. Figure 6.6 (a) shows one such situation, consisting of a corridor with a guard moving back
Figure 6.5: Probability map distribution of getting seen, for the same level shown in figure 4.12. The closer the color is to magenta, the more frequently a cell is observed by enemies, and the green cells are never seen by enemies. Red cells are places the player can never venture, e.g., $\Sigma_{obs}$. The red cells do not perfectly match the obstacle because of the grid granularity used.
6.3. Experiments

and forth (red path). The expected player path is shown in blue, and Smith argues that players will wait in the alcoves to avoid the guard. We redesigned this space in *Unity 3D*; figure 6.6 (b) shows the resulting space and heat map of possible player movements. As in Smith’s design, the guard starts on the right (yellow dot), moving right to left and back (and repeating); the player starts on the left with the goal to exit on the right. During the design phase, we decided to opt for a FoV that would block the entire hallway when placed in the middle of it, thus forcing the player to use the alcoves. We also used a guard walking and rotational velocity slower than the player in order to get more interesting solutions. Chapter 9 presents how different FoV parameters influence solution qualities.

The heat map solution shows that the expected behaviour does not actually occur. While a majority of paths did head into the first south alcove and then to the north, a large number went directly to the north alcove. None of the paths actually entered the rightmost south alcove. Detailed exploration with our tool showed that once the guard has passed in front of the hidden player there is no need to hide anymore, and the player can proceed directly to the goal. A designer might not have thought that it was possible to reach the north alcove without having previously reached the first south alcove, and so this may indicate weaknesses in the level design. In order to force the players to move into the first south alcove, the north alcove can be moved to the right, to the point where it is too far away to reach without being seen (figure 6.6 (c)). This achieves the goal of forcing the player into the first alcove, but that ends up obviating both the second and third alcoves. We experimented with a number of other variations in the level design and guard movement, and were only able to force the player to visit all 3 alcoves by modifying the guard’s initial position and orientation, and having her stop and rotate/scan at multiple points in the patrol route (figure 6.6 (d)). The solutions found are strongly connected with the parameters used to define the guard’s behaviour, e.g., FoV aperture. Given our assumptions about the guard parameters, our tool to determine, correct, and validate the intended behaviour shows the value of such design exploration, resulting in a better level design and avoiding an expensive play testing cycle.\footnote{For an interactive demonstration of the first part of this investigation see https://www.youtube.com/watch?v=b8gSCG0Xs_k.}
6.3. Experiments

Figure 6.6: (a) Level design proposed by Smith [Smi06]. (b) Movement heat map found from our tool. (c) Moving the north alcove. (d) Changing guard behaviour. Please note that the little white dots are the enemy (represented by yellow circle) waypoints.
6.3.2 Metal Gear Solid

*Metal Gear Solid* is a 1998 game highly regarded as one of the first real stealth oriented games. We used the very first level of the game. Figure 6.7 presents a 3D overview; in the game narrative the player is dropped in the enemy base’s cargo dock (1) and has to sneak around the enemies to get to the elevator (2). We recreated this particular level in Unity 3D using a fan sketch-up file [Ano07] and we added the guards movement from observing the game. The level contains two enemies represented as orange dots in figure 6.8, who follow the red segments they sit on. Their discretized FoV is also represented by the orange squares. The player starts at the blue circle and has to reach the green circle.

This is a complex scenario with multiple path choices for the player. Using the previously presented RRT search algorithm we were able to find thousands of different paths shown by the heatmap in figure 6.9. It is interesting to point out that no path was found to go in the lower left corner, and very little on the right corner. These locations are safe spots that the player may aim for as intermediate goals, but which are not strictly necessary to visit in order to reach the final goal; in the game they added objects for the player to collect at these locations, making these spots more desirable. This is also perhaps an artifact of how RRT searches a space and constructs a final path: even if a branch of the search tree explores these areas, they do not make good progress to the solution, and actual solution paths will tend to branch off more productive parts of the search tree. This result is further interesting for a game designer as it shows that all hallways can be used to reach the goal as none is blocked. As a starting level of the game, it is important that the player can make progress without requiring much backtracking or replay.

6.3.3 Stealth Implementation Performances

The interactive nature of our design hinges on the tool performing well for a reasonable range of parameters. We thus explore the influence of three parameters on the RRT efficiency such as time took to find a path and the probability of finding a solution. We ran our test in the MGS level as it represents a real-life usage case. For every performance data point, we ran 150 iterations and for the probability tests every data point is represented by the number of tries the algorithm needed in order to find 150 good paths. Results were
Figure 6.7: A 3D overview of the MGS dock level, where (1) is the starting location, (2) collectible items, (3) a jumping point to run away from guards, (4) noise traps, and (5) the exit location. This figure was taken from the Metal Gear Solid: Official Mission Handbook [Hod98].
Figure 6.8: Overview of the major paths the two enemies take in the MGS dock level, whereas their FoV is represented in orange. The player starts at the blue circle and has to reach the green circle location.
6.3. Experiments

Figure 6.9: Metal Gear Solid’s dock level’s heatmaps of a 1470 paths found by the tool
6.3. Experiments

calculated on an *Intel i5* at 3.00 GHz with 8 GB of RAM memory computer within *Unity 3D* 4.1.2f. While a given parameter is varying it uses fixed values for other parameters: 30 000 attempts, 1 200 samples, and a grid size of 60×60×1500.

**Number of attempts**

Varying the number of attempts, state budget $m$ in algorithm 7, the RRT does before stopping has a huge influence over the performance and the probability of finding a path. In figure 6.10, one can see that the time taken to find a path increases more or less linearly with an increase in the number of attempts, although only up to a point, after which it levels off. This result is intuitively correct, in that when the search is allowed to explore more states before being terminated it naturally takes longer as more work is done, at least up until a level of resource sufficiency is reached. When the search is given a lower budget, the probability of finding a path also decreases as seen in figure 6.11. When the budget is around 40 000 states for the MGS level, the probability of finding a path is really high, although it might take 1500 ms to find a solution. Biasing the search toward the final or other heuristically determined goals would likely improve the performance significantly.
6.3. Experiments

Grid size

This parameter represents a trade-off between granularity of the simulation and size of the search space. Fine-grain is naturally preferred, but as the space gets bigger picking a point within the goal region $\Sigma_{\text{goal}}$ gets less probable with the uniform distribution sampling algorithm. Please note that for this test we only find a path when we sample a point in the goal region. This cost is clearly seen in figure 6.12, the probability of finding a path decreases as the grid size increases. Very reasonable grid sizes turned out to be effective in our experiments, so this is not necessarily a critical cost factor in practice, but using a grid is cumbersome, and although it greatly simplified the initial tool design, future development will explore the use of a continuous representation.

Time bound

This defines the upper bound on the dimension $t$ within the search space. The main impact of this parameter is on the probability of finding a path, as shown in figure 6.13. The probability of finding a path is higher with a lower time bound, this is caused by searching a smaller volume and constructing a tree with nodes closer to each other. On the other hand as we increase the search volume, the explored states are more spread through the space lowering the probability of finding a path. More generally, we can see an increase in

Figure 6.11: Performance analysis, attempts vs. probability

![Graph showing probability of finding a path vs. number of attempts.](image-url)
6.4 RRT solutions humanness

Figure 6.12: Performance analysis, grid size vs. probability

the time dimension as simply an increase in the size of the state space, and thus requiring
greater resources for a successful RRT search.

6.4 RRT solutions humanness

The capacity for including mechanisation in the game design process has the potential to
replace human testers. This assumes that the presented solvers can find solutions similar to
humans. This section investigates this question for RRT, which has shown great potential
for randomness and diversity in solutions. Thus we evaluate whether the baseline RRT
(algorithm 7) solutions are reasonably similar to human paths. In order to conduct such
a test, we implemented a game level taken from Wasteland 2 as a first person exploratory
game, where the player was given the task to simply find an exit. Below we describe the
process of collecting human traces, followed by a comparison of the RRT and human data.
Based on our observations we then propose different algorithmic improvements to RRT
that aim to achieve more human-like solutions.
6.4. RRT solutions humanness

Figure 6.13: Performance analysis, time dimension bound vs. probability

Figure 6.14: This is the Wasteland Temple of Titan overview map. For our testing purpose we only kept the level layout abstracting away any symbol on this figure. http://game-maps.com/W2/img/Wasteland-2-Arizona-Temple-of-Titan.jpg
6.4. RRT solutions humanness

Figure 6.15: Wasteland 2 level, where the player starts at $\sigma_{\text{init}}$ and has to reach $\Sigma_{\text{goal}}$. The green curve is a final RRT solution, whereas the blue curve is a raw solution. The yellow points represent non-interactive content locations.
6.4. RRT solutions humanness

6.4.1 Experimental Setup

We used the *Temple of Titan* level from Wasteland 2 as we wanted to focus our effort on analysing simple pathing without the stealth component. Thus our developed level only has non-interactive objects. Figure 6.14 shows the outline of this level, from which we took the basic geometry and location of entrance/exits, but abstracted away any interactive symbols denoted on that figure. For our human study we implemented a space theme level where the level outline was used to create a first-person representation as seen in figure 6.16. For our RRT search process, which did not need a human interface, we used the simpler abstraction seen in figure 6.15. It is an overview interpretation of the level geometry, where the player starts at the green node and has to reach the red one. The figure also contains two solutions from a RRT search, the purple one is a raw solution, whereas the green path is its smoothed solution. The yellow nodes represent non-interactive content locations, which was also included in the human study version; their positioning was determined by a generative content placement algorithm which is presented in chapter 9. It is important to note that the presence of visual content is important to understanding human pathing choices, even in the absence of any direct connection to level solutions.

Figure 6.16: A preview of the level played by human players.
6.4. RRT solutions humanness

6.4.2 Analysis

In this part of the discussion we highlight the differences of RRT solutions to the human ones. We first describe the general trends in the human data set followed by a similar analysis for the RRT solution data set. Further analysis and algorithmic improvement are also discussed that should reduce the human–RRT gap in the following section.

In the human study players were given only one task, to reach the exit (without any specific time constraint) from a given starting position. We had 73 participants drafted from the internet (Unity 3D sub-reddit). While they were playing we collected position and orientation data every half-second. Figure 6.17 shows a heatmap movement of humans in our study. The human players took 86 seconds on average to complete the level while travelling an average distance of 680 level units. Interestingly most humans walked in the middle of the hallways, and since content was put at the end of hallways, some humans actually explored most dead-ends.

Using RRT we computed 500 paths using the randomized solver presented in the chapter 4. Note that in order to allow us to do many trials, the RRT solutions were computed on the simplified geometry presented in figure 6.15, which has the advantage of using just a few straight lines. Figure 6.18 presents the RRT-movement heatmap. Based on simple qualitative observation, one can see clear differences. At a fine-grain level, RRT solutions
6.4. RRT solutions humanness

have a noticeable rectilinear property, making sharp turns and with all movement highly linear. Part of this is due to slight differences in the level representation which gave a little more movement latitude in the human-played game.

In terms of overall path choices, there is an arguable tendency for RRT solutions to prefer going into the south hallway, and a clear difference in exploration of the far right side of the level, which is less for RRT than is seen for the human players. The human paths also seem to have less variance within a corridor, whereas the RRT tends to uniformly distribute over the width of the hallway. These differences can be partly attributed to the way RRT works, with a uniform sampling approach necessarily tending to fill out the corridor volume, and the RRT search terminating as soon as the solution is found. Brogan & Johnson showed that humans have a tendency to stay away from corridor walls, and so do not fill the corridor volumes as uniformly [BJ03].

We can quantitatively examine these differences in path distribution from several perspectives. Figures 6.19 and 6.20 show the distribution of time taken by the solutions for RRT and humans respectively. Here we can see striking differences between RRT and humans, with the latter both much quicker to reach the goal and showing dramatically
6.4. RRT solutions humanness

Figure 6.19: Distribution of time in seconds taken by the RRT solutions to complete the Wasteland level, also including the average (dashed line) and median (dash-dot line). The continuous line shows an equivalent normal distribution for the calculated mean and standard deviation.

Figure 6.20: Distribution of time in seconds taken by the human to complete the Wasteland level, also including the average (dashed line) and median (dash-dot line). The continuous line shows an equivalent normal distribution for the calculated mean and standard deviation.
6.4. RRT solutions humanness

less variance in doing so. This difference is mainly due to the RRT algorithm sampling behaviour for the time-dimension. We sample uniformly, and thus other than respecting maximum speed bounds the connection between two points does not try to be as fast as possible. This makes the algorithm tend to uniformly explore the time dimension, and so give a wide and relatively even range of solution times. Humans tend to move at maximum speeds to known goals, and so demonstrate much less variance.

Figures 6.21 and 6.22 present the distribution of distance travelled by the solutions to reach the goal from the start in level unit. Although not as dramatic a difference as with time, RRT solutions tend to be minimal, biasing toward shorter paths, and spending less time wandering around exploring the level than humans. This is part of the nature of RRT of course, with search terminating once the goal is reached. We also note in this that our RRT does not model points of interest in the search process. Since our game level did include a few non-interactive art assets placed around the level (to make it into a more realistic game level), these may also have influenced player movement, and inspired different pathing choices.

6.4.3 Algorithmic Improvements

To further improve how well RRT mimics human paths, we here propose and analyse three simple algorithmic variations on the basic RRT. We then address the way RRT’s uniform sampling in time and space induces differences in both time and how solutions cover corridors, and also introduce a content bias that inspires RRT to more fully explore the space as the humans do.

No-Smoothing

We attribute some of the early difference between human and RRT to the use of path smoothing, which necessarily results in more long, straight path segments. We thus ran new traces considering the raw solutions produced by the RRT, shown in figure 6.23. Without the smoothing step the heatmap looks a little more natural, with gentler curves around corners. This difference might be further improved still, by using a local planner that imitates human movement [BJ03]—although avatar motion did not need to respect a steering
6.4. RRT solutions humanness

Figure 6.21: Distribution of the distance travelled in level units by the RRT solutions to complete the Wasteland level, also including the average (dashed line) and median (dash-dot line). The continuous line shows an equivalent normal distribution for the calculated mean and standard deviation.

Figure 6.22: Distribution of the distance travelled in level units by the human to complete the Wasteland level, also including the average (dashed line) and median (dash-dot line). The continuous line shows an equivalent normal distribution for the calculated mean and standard deviation.
Figure 6.23: RRT heatmap movement with no smoothing of 500 solutions (please refer to figure 6.15 for the level outline)

model in the human traces nor in the RRT traces, it is possible that human players have a bias toward mimicking a vehicle motion model.

**Time Coverage**

Significant differences in the solution length between humans and algorithmic solutions can cause a designer to misjudge game length and resource requirements, both of which are important in game design. In order to better model the fact that players tend to move at maximum speed between locations, the RRT computation can be modified so the time-dimension in a sampled point is contracted to the minimum necessary to reach the same geometric position. Figure 6.24 shows the time distribution paths so constrained; here we not only compact the distribution, but also achieve an average and medium of 81s, a value much closer to that found in the human solutions. Of course as this approach voids our time sampling structure, this could be problematic in game situations where time is an important part of game interactions e.g., stealthy paths, moving platforms, etc. as we have seen in chapter 5. Moving some nodes on the solution to different time locations might let
6.4. RRT solutions humanness

Figure 6.24: Distribution of time when RRT steps are forced to move as fast as possible towards the goal.

the path collide with an enemy’s FoV, and thus would need an additional check to ensure the point remains valid. Another approach to this issue could be to use less uniform time distributions in sampling such as Poisson, where smaller numbers have a higher probability than larger ones.

Corridor Coverage

By comparing the solution from figures 6.17 and 6.23, one can notice that humans have a highly uneven distribution of positions within the width of a hallway: humans prefer walking in the middle of the hallway, whereas RRT solutions again tend to uniformly fill the available dimensions.

There are various ways we can address this within the RRT algorithm. We chose to add a probability of dropping the motion state ($\sigma_{motion}$) as it gets closer to a wall. This probability is scaled as an inverse square of the closest distance from $\sigma_{motion}$ to any wall, dropping to 0 at a constant $dMax$, selected to be approximately half the width of a narrow corridor.
The resulting heatmap for 500 paths is presented as figure 6.25. Solutions in this figure are noticeably less uniform in corridor coverage, although the effect is less pronounced in generally more open areas. Use of a balancing force that attempts to make the distance between opposite walls equal, would likely improve this, as would human-like steering models that attempt to avoid walls when computing motion [ZPPD09]. Cost is a concern here, however; and our simple approach already increases the cost of computing a single solution to 6-10s, and so more sophisticated approaches need to be weighed against the reduction in design interactivity.

**Content Bias**

In figure 6.15 the yellow nodes represent the position of non-interactive visual content, such as statues, skeletons, chests, *etc.* that the human players encountered. Although we added that just to better mimic a game level and make the gameplay less tedious for our experimental subjects, content has an influence on the human choices while wandering through the space [WZ14] that is not present in our RRT paths. In order to achieve a human-like solution, we thus impose a bias to sample the content locations (as well as the
6.4. RRT solutions humanness

Figure 6.26: Heatmap of 500 RRT solutions using a bias towards non-interactive content locations and all the other improvements proposed (please refer to figure 6.15 for the level outline)

goal) with a certain probability, instead of using uniform distribution on all dimensions. In this case, with probability $p$ we uniformly sample one of the content locations, e.g., one of the yellow points in figure 6.15. As the probability increases, the more the solutions will path towards these locations. This approach encourages solutions to visit the content locations, and the result can be seen in figure 6.26. We can see that the artificial players walked closer towards the content locations, which qualitatively matches more the human traces (see figure 6.17 on page 166). This approach has little impact on the performance.

Our results in this investigation suggest that there are significant differences in the paths generated by heuristic search (RRT), and actual human paths, but that it is also possible to mitigate these differences through careful tuning of the search heuristic. Further work in this area would be to develop a formal metric for measuring the similarity of generated to human paths that can be used to optimize this tuning. Our solution-time and distance measures imply a possible direction for constructing a compound metric that may be suitable. Path-similarity measures, such as Hausdorff [Hau14], or Fréchet [Fré08] distance can also
be used to measure geometric similarity, although in the constrained geometry of a typical 3D game, geometric similarity can be more a function of which solution is found than of detailed characteristics of the style of finding it. Other forms of game interaction may also be important to factor into any similarity measurement: reaction to combat, puzzles, and the impact of having to backtrack when searching an unknown environment also likely affect movement patterns, and thus would need to be considered, or separately measured.

6.5 Chapter Conclusion

This chapter presented an extensive description of the stealth game search space with non-trivial visualizations. We also introduced multiple demonstrations for using RRT in the context of stealth game design such as solving Randy Smith’s stealth level construction. We also discussed in depth the RRT humanness using a simple game context in order to compare human solutions to the solver’s ones, as well as including different algorithmic enhancements to improve the humanness of RRT solutions.

This chapter (and the previous two) introduced key concepts for understanding the following two chapters. In the next chapter we will focus on analysing and designing different metrics for quantitatively measure game design qualities and, later, extend our different solvers to include broader game genres and improve their usability.
This chapter presents a stealth level analysis tool that offers quantitative measurements about the level in creation. In the previous chapters we established basic analysis algorithms/mechanisms, which included both qualitative and quantitative observations. To enable further mechanisation of game design it is important to develop multiple, detailed quantitative metrics that can then be used to measure and/or constrain game design. Thus, the presented approaches, in this chapter, allow developers to further their understanding of their designed level, such as with heuristic data on how the player might play a certain level, or enforce or measure certain bounds on the level difficulty. More precisely we present two approaches to stealth level analysis, the first one is based on level geometry analysis (section 7.1.1), and a second approach that uses path analysis (section 7.1.2).

In the context of analyzing level geometry we propose further different visualizations to better understand the solution space as well as heuristics to help determine the in-game experience. From the perspective of path analysis, we looked at measuring the perceived risk level of a solution for the player, where we are interested in quantifying how stressful it might be to play that solution. We use a human study to evaluate multiple proposed metrics, showing that while our measures all have a reasonably close correspondence to human perception, the (conceptually) simple measure of path-distance correlates best with human judgement of relative risk. Like previous chapters, the presented algorithms are integrated into a non-trivial, Unity 3D-based design tool, illustrating technical feasibility of our metric evaluation, and allowing us to further demonstrate application of a measurement-based
7.1. Method

approach to a non-trivial stealth scenario taken from a realistic computer game. Specific contributions for this chapter include:

- We also develop heuristics to quantify level stealth difficulty.
- We propose and describe 3 different metrics for measuring player perception of risk in a stealth game context. These metrics consider intuitively appropriate factors such as distance to enemy (Dist), line of sight (LOS), and the presence of “near misses” in being seen (NM).
- Using a human study, we evaluate how well our metrics correlate with a human ranking of the relative risk of different generated stealth paths. In this we find that (path) distance to an enemy is likely a dominant factor in evaluating risk, more important than more complex line-of-sight oriented features.

7.1 Method

Understanding the range of possible player behaviour is interesting, but it is also useful to know properties of the game solutions that relate specifically to player experience. For example, the amount of “danger” or “risk” inherent in a stealthy path has a close relationship to the potential for discovery by the enemy, and different level designs and solution paths will imply greater or lesser risk to a player. In this section we will present two approaches for computing such metrics, one based purely on the level geometry, and a second approach based on solution measurements.

7.1.1 Level Geometry Analysis

Heuristic measures of level difficulty can be calculated directly from the geometry found in the state space representation. We will first explore the volume taken by $\Sigma_{	ext{FoV}}$, then we shall explore relationship between roadmaps and $\Sigma_{	ext{FoV}}$. 
7.1. Method

**Volume analysis**

Intuitively, the denser enemy FoVs are in the state space, the more difficult it will be for a player to find a path from one point to another without encountering enemies. A trivial approach to evaluating the difficulty of a stealth level is then to check the ratio of $\Sigma_{FoV}$ to the full (non-obstacle) state space, a measure we denote $V_{ratio}$. This value is computed as,

$$V_{ratio} = \frac{V(\Sigma_{FoV})}{V(\Sigma - \Sigma_{obs})}$$

where $V(.)$ refers to the Euclidean $\langle x, y, t \rangle$ volume taken by the space, given a bounded maximum $t$. The larger the ratio, the more space is taken by the enemies’ FoV, and so the greater the difficulty. This measurement has the advantage of being relatively simple to compute, but is limited in only giving a single, scalar value to characterize the dynamics of the entire level. There is also subtlety in choosing the maximum $t$, as many games are not time-bounded; we use the longest periodicity in guard movement. For example, we obtain a ratio of 0.075 for the MGS level previously described in chapter 4, whereas the level 1 (figure 7.9 on page 192) from our human study got 0.08. Naïvely, this suggests that level 1 is as or more difficult than the MGS to solve, but this is not the case, as level 1 is pretty simple. The difference is explained by the proportion taken by $\Sigma_{FoV}$, which in level 1 is larger than the MGS level. In general, this metric is intended to give a very high level understanding about the search state space in terms of overall density of enemy and obstacle presence, but does not necessarily well measure level difficulty for stealth movement—e.g., a guard blocking an essential corridor would make a level unsolvable, but need not have a large ratio. Thus in the following section we discuss a visual approach to get a better understanding of a stealth level difficulty.

**Roadmap analysis**

A finer grain view of map difficulty can be obtained by specializing the heat map approach presented in chapter 4 to individual segments of a roadmap. Any roadmap algorithm (such as taking the dual of a polygon-triangulation) can be used—the point is less to construct an ideal roadmap and more to just identify distinct, interesting areas of the map that a player
Figure 7.1: Triangulation-based roadmap of a level. Triangles edges are indicated by gray lines, with the roadmap constructed by the dual shown with red segments.
7.1. Method

Figure 7.2: Metal Gear Solid’s first level with a heatmap roadmap of the level, as the segments get closer to red, the more time the enemy spends looking at it.

may traverse. In our case we use a triangulation based roadmap construction [MF09]. This is constructed by flattening $\Sigma - \Sigma_{\text{obs}}$ into the plane, triangulating the resulting 2D geometry, and constructing a roadmap by creating a node at each triangle center and joining it to the midpoint of each of its edges. An example is shown in figure 7.1.

With the roadmap constructed we can then identify the segments of the roadmap that are harder or easier to traverse in the sense of being more or less observed. We give each segment an epsilon width which allows us to create a volume for each segment. We then look at how much of that volume is occupied by the enemies FoV (ratio). Then we create a colour linear interpolation between 0 and the maximum volume ratio value to determine the segment colour. This gives a visual indication of relative difficulty encountered along different possible player paths and path segments.

Figure 7.2 depicts such a road map for the MGS level. We can notice 4 red segments
7.1. Method

that will be harder to cross as the enemy spend more time observing them. A more in-depth analysis of these segments can be achieved using the road map graph, where we used depth first search to recursively find all possible paths from starting position to end position (these paths do not take into consideration the enemies FoV). Using this process, we can then identify paths that are going to be easier than other ones. For example paths that only use green segments such as paths using the top hallway are heuristically easier than the ones crossing the middle section of the level. According to our analysis, crossing the middle section should be hard for the player. This conclusion is not exact, this location is harder only because this is the only point where both enemies walk, so for the whole time dimension this section is more blocked than any other section. There are, however, more openings (windows) in the middle part relative to any other parts of the level, which explains why this route is still a very popular option, as will be shown in figure 8.5 on page 214.

A disadvantage to these area-based metrics is that when we project the time component we always lose some information about the level, and although these aggregate level metrics are intuitively interesting and perhaps good for broad generalizations, they can also be misleading. It is thus perhaps more useful to consider metrics that measure player paths (solutions) themselves.

7.1.2 Path-Solution Analysis

As we saw in the previous section, using heuristic metrics based on level analysis is insufficient for reaching concise conclusions about the level difficulty. This section investigates how we can use existing level solutions to determine its difficulty. Our assumption in this is that the amount of danger or risk inherent in a stealthy path has a close relationship to the potential for discovery by the enemy. Relative proximity of enemies is thus important, as is the direction in which enemies face—if an enemy directly looks at a player the risk of failure is also increased. We first describe a metric that relies on path distance, followed by one that focuses on the enemy’s relative angle of sight, and finally a more complex metric that tries to measure how close a player came to being discovered. The different metrics presented should be looked as propositions for a formal definition of risk and will
7.1. Method

be validated with a human study.

**Dist: Distance to enemy**

Heuristically, a player close to an enemy risks discovery more than one far away. Since discovery is typically predicated on visual contact, however, this distance measure also needs to take into account game obstacles—an enemy behind a thin wall represents less of a danger than one equally close but not occluded. Our distance to an enemy measure (Dist) is thus defined in terms of path-distance using an A* search within Σ_free rather than simple Euclidean or Manhattan distance. In order to compute this metric we need to consider proximity to each enemy at each point in time. We thus define $d^*(\alpha, \beta)$ to be the path-distance between two planar position, $\alpha$ and $\beta$. This gives us the equation,

$$\text{Dist}(p) = \frac{1}{\sum_{e \in E} \left[ \sum_{t=1}^{T} \frac{1}{d^*(g(p, t), g(e, t))^3} \right]}$$  \hspace{1cm} (7.1)

In equation 7.1 $p$ represents a player’s path, and $e$ is an enemy path from the set of all enemy paths $E$. $T$ is defined as the maximum $t$ value in our path $p$. The function $g$ returns the tuple $(x, y)$ in the plane at time $t$ for any given path. In the above equation we use the reciprocal of the distance cubed as a non-linear means of scaling intensity as the player gets closer to an enemy. This is based on our need to have the function weight closer enemies as much more dangerous than distant ones, and our observations during prototyping that other exponents tend to either over-value or under-value proximity.

Variable movement speeds imply a further scaling factor must also be applied to produce meaningful values. Suppose there are two paths, initially identical, and deviating only during the last unit of distance, where despite being well away from any enemies one path ends up taking twice as much time as the other. Intuitively, such paths should have very similar values in terms of relative danger, but the accumulation of terms over time in equation 7.1 can give the slower path arbitrarily higher danger values, depending on the relative movement speeds.

To reduce the impact of these extra factors, we normalize the Dist value by the total length, $L$, of the path in three dimensions (2D×time).

$$\overline{\text{Dist}}(p) = \frac{\text{Dist}(p)}{L}$$
A drawback to Dist is that it is relatively expensive to calculate. Even though A* is an efficient search algorithm for path planning, an A* search is done for every enemy in $E$ and repeated for every time step $t$. Incremental approaches and cached searches would improve this, as agents typically only move small distances between time steps. Since our focus is on evaluating the metric itself rather than optimizing efficiency this remains future work.

**LOS: Considering enemy view**

The risk of discovery is also increased when enemies look toward a player—eye-contact is major indicator of being observed, and so the more directly a player is in the line of sight of an enemy the higher the risk. Our second metric focuses on this factor, emphasizing the relative angle between the enemy’s direct line-of-sight (LOS) and the player.

The first step in this computation is to know if there exists a direct LOS from the player to any enemies. If so, the angle, $\text{Angle}(v_1, v_2)$, between the vector formed by $r = g(p, t) - g(e, t)$ and the direction the enemy is facing, $f(e, t)$ is calculated. For these we define helper functions,

$$Vis(p, e, t) = \begin{cases} 
1 & \text{if } r \subseteq \sum_{\text{free}} \\
0 & \text{otherwise}
\end{cases}$$

$$\theta(g(p, t), g(e, t)) = \text{Angle}(r, f(e, t))$$

This angle is then weighted according to a Cost function that considers how far outside the field of view of the enemy the player is. Computation of the cost function is illustrated in figure 7.3. In greater detail, if we take the dot product of $f(e, t) \cdot r$, where $f(e, t)$ refers to the look at vector of the enemy and $r$ to the vector joining the two entities (see figure 7.3), and that value is 1, the player is in front and the value the cost is 1. On the other hand, if the dot product returns -1, then the player is behind the player, thus the cost is 0.5. Considering just the angular proximity of the player to the enemy is of course not enough in itself. An enemy that is very far away, outside the range of vision, is not much of a threat, even if they look directly at the player. We also have the same concern as with Dist, that the variations in duration of the path have a significant impact on the metric value. Thus we scale the accumulated angular values by dividing by both the Euclidean distance (cubed), and the
Figure 7.3: Threshold-based angular cost calculation. This figure illustrates how the Cost function maps angles to $[0.5, 1.0]$. 
7.1. Method

Figure 7.4: A representation of the NM metric time-window. Here enemies can see both near-past and near-future player positions, and so even if the path is successful due to changing enemy FoVs, the player is undertaking a risky movement.

total path length, $L$. Note that in this case Euclidean distance ($d$) rather than path-distance ($d^*$) is usable, as we have already determined a straight line-of-sight exists.

$$
\text{LOS}(p) = \frac{\sum_{t=1}^{T} \sum_{e \in E} \frac{\text{Cost}(\theta(g(p,t),g(e,t)))}{d(g(p,t),g(e,t))^3} \text{Vis}(p, e, t)}{L} 
$$

(7.2)

**NM: Measuring nearly misses**

The last two metrics miss some important information about the player’s behaviour, in that they do not account for the player’s past or future behaviour. A situation that involves a near-miss in terms of discovery, barely avoiding being seen, is more risky than one where the player has ample latitude to easily avoid detection.

To measure near-misses, at each given point in time we look at the states of the last $n$ positions of our player, as well as the next $m$ positions in the future using a fixed time-step $\Delta t$. If these prior or future positions are exposed to enemy view, as shown in figure 7.4, the player experienced (or will experience) a near-miss in terms of detection, suggesting a risky, stressful movement.

The full, window-based risk calculation is shown in equation 7.3. The function
7.2 Results

Seen(α, τ) take as arguments a planar tuple and a specific time τ. In order to avoid continuous analysis, we discretized the path with a fixed time step, which is described by the first summation in equation 7.3. The whole equation 7.3 is intended to capture the idea that the closer a player passes (or will pass) to an enemy’s FoV, the greater the risk. Note that unlike the previous two metrics NM is an unscaled value, not normalized to the total length of the path. The risky, near-miss behaviours we are trying to capture in this calculation imply singularly stressful events that make a path dangerous, even if the rest of it is relatively safe.

\[
\text{Seen}(\alpha, \tau) = \begin{cases} 
1 & \text{if } (\alpha_x, \alpha_y, \tau) \in \Sigma_{\text{FoV}} \\
0 & \text{otherwise}
\end{cases}
\]

\[
W^-(p, t, n) = \sum_{i=1}^{n} (n-i)^2 \cdot \text{Seen}(g(p, t-i), t)
\]

\[
W^+(p, t, m) = \sum_{i=1}^{m} (m-i)^2 \cdot \text{Seen}(g(p, t+i), t)
\]

\[
NM(p) = \sum_{t=1}^{T} (W^-(p, t, n) + W^+(p, t, m))
\] (7.3)

An obvious final direction for these metrics would be consider a hybrid form, combining the individual metrics in some fashion. Our interests in this work, however, is to first understand the efficacy of the our metrics in isolation before addressing the significant complexity of tuning the weights of each metric’s contribution in a hybrid. Efficiency is also expected to be a concern, and a hybrid form would need to demonstrate a sufficient trade-off between cost and improvement. The experimental work we now present shows that the individual metrics already have good predictive power.

7.2 Results

In the context of using computing techniques for the game design process, the different techniques used should propose meaningful quantitative values, e.g., the number should represent human behaviours and/or perceptions. Thus, in this section we look at a specific experiments that validate our risk metric using human judgement. Our experimental work is aimed at demonstrating and comparing the value of our metrics with respect to measuring
stealth difficulty or danger, which will allow us to gain deeper knowledge of the definition of risk. We then explore how the metrics can be used to evaluate a given stealth level.

### 7.2.1 Metric Behaviours

Our metrics have interesting qualitative differences in how they measure danger, and thus tend to identify risky behaviour at different points. Consider an example taken from level 14 of our human study from section 7.2.2. Please note that the human study will be described further in the following section. Figures 7.6, 7.7, and 7.8 show three key-frame moments of that level. The player is represented by a small blue sphere, and proceeds along the gray trace from the the large blue sphere (bottom right of the center) to the large green sphere (bottom left of the center) positions, while the black bars indicate walls, and the orange areas the (discretized) FoVs of 3 mobile enemy guards.

Figure 7.5 shows the evolution over time of each of our metric values. This way we can clearly see at which key moment the metrics are activated. Each metric indicates points of maximal danger at different times in the player’s progress. Although the Dist metric shows some activity in key-frames 7.6 and 7.8, the peak of the Dist metric occurs in key-frame 7.7, where proximity to an enemy is greatest, and the overall accumulated path-distances from the player to all the enemies ends up maximized. The LOS metric, however, considers the point of maximal danger to be at key-frame 7.6, when the player moves directly through the line-of- sight of an enemy, if just outside the visual range. The NM metric finds risky behaviour only at key-frame 7.8, when the enemy has just turned to look at the earlier path of the player.

Arguments can be made for each of these that they accurately represent the element of risk in this stealthy path. Maximal points, however, clearly differ. In the following subsection we thus look at how these metrics compare to human rankings, in order to determine which best corresponds to human judgement.
Figure 7.5: Evolution of the presented metrics over time (s) on level 14. The numbers refer to key-frames in figures 7.6, 7.7 and 7.8.
7.2. Results

Figure 7.6: Key 1 of level 14, $t \approx 200s$.

Figure 7.7: Key 2, $t \approx 500s$.

Figure 7.8: Key 3, $t \approx 560s$. 
7.2. Results

7.2.2 Human Study

Our human study asked participants to evaluate different player paths (produced by the algorithm presented in chapter 4) and determine which one was the safest. The degree of correlation between participant choices and which paths our metrics determined safer would thus show whether our metrics matched human perception, and thus tell us which factors most contributed to what players felt was risky.

Participants were presented with an animated gif image on endless loop showing the movements of two players paths generate by our RRT search described in chapter 4. The animation included enemy movements and FoVs. The level designs were based on different, but game-realistic geometries and enemy arrangements, attempting to provide a cross-section of level complexity and path safety. Figures 7.6 through 7.12 show different examples of the 15 levels presented—the small red and blue spheres represented the two players, both beginning and ending at the same points (choice of red or blue for the different paths was randomized). After viewing a level, the participant had to click on a button to select which path was the overall safest, allowing us to compare the resulting ranking with our individual metric ranks. The study was designed to take about 15–20 minutes to complete (avoiding participant burn-out). The study was conducted under unsupervised conditions (i.e., on the participant’s web browsers, at their leisure) and consisted of 27 anonymous participants, mainly drawn from the graduate and undergraduate population of our department.

Table 7.1 summarizes the raw data from our study. For each of the 15 levels, and for humans and each metric, we indicate a 1 if the participants or metric selected player 1’s path as safest, 2 if they selected player 2’s path as safest, and 0 if there was no consistent choice. The latter value only occurred for human players, where we imposed a 75% threshold on agreement between participants to establish a ranking. This count can be recuperated from the first line of table 7.1.

As an initial and basic observation, we observe that all metrics have quite good agreement with the humans. Out of the 10 levels for which human agreement reached our 75%

---

1The levels and study are fully available at http://goo.gl/fGg3pR. The levels outlines are also included in appendix A
7.2. Results

Table 7.1: Human rankings vs. metrics. 1 refers to blue player and 2 to the red player. 0 represents that no agreement was reached by the humans.

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td># voted</td>
<td>1</td>
<td>22</td>
<td>18</td>
<td>27</td>
<td>23</td>
<td>12</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>22</td>
<td>4</td>
<td>22</td>
<td>8</td>
</tr>
<tr>
<td>Human</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Dist</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LOS</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>NM</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

threshold LOS agrees 7 times, and NM agrees 7. Surprisingly, given that it is conceptually the simplest, the Dist metric stands out as achieving perfect agreement with the humans. Even with just 10 of our 15 human judgements considered definitive this is very unlikely if due to random chance, suggesting that path-distance might be a more important factor than others.

Our expectation after examining the paths found in the levels we had designed was that levels 2–4 and 9–13 represented situations in which one path was clearly safer than the other, while levels 1, 5–8, 14, and 15 were more ambiguous. We thus now explore levels 1, 3, 5, 12, and 14 as example situations where the outcome either did not match our expectation, or where our metrics disagreed.

### Level 1

As shown in figure 7.9, this level consisted of a central occlusion, with a single guard blocking one route around the obstacle and two paths going the other way. Both paths easily avoid the enemy guard, with the only significant difference being that the red path arrived at the goal at the same time as the enemy was rotating on the right.

The metrics for level 1 shown in table 7.2 indicate that Dist and LOS ranked the blue path safer than the red path, although with a very small difference in value. We will revisit this concept for larger values, but in this case, since we are interested in ranking the paths, this is a valid ranking. LOS measures some danger for the red path whereas no line of sight
7.2. Results

Figure 7.9: Level 1 from the human study

Table 7.2: Level 1 Metrics

<table>
<thead>
<tr>
<th></th>
<th>Dist</th>
<th>LOS</th>
<th>NM</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue path (1)</td>
<td>0.00002</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>red path (2)</td>
<td>0.00003</td>
<td>0.00003</td>
<td>0</td>
</tr>
</tbody>
</table>

exists between the blue player and the enemy. In the case of NM, since the players’ paths do not cross the enemy’s path, they both were measured as zero. Since the humans clearly identify the blue player being the safest, this suggests our NM metric may be too coarse, and that a focus on unusually dangerous events is not sufficient.

**Level 3**

This is a tricky level, in that there are two possible ways to get to the goal, and the players proceed through one or the other; see figure 7.10. The north corridor has a fast moving enemy walking north/south, and the south corridor has a slow moving enemy walking from east to west. Here the blue player quickly raced through the north corridor, while the red player slowly sneaked behind the enemy on the south corridor and then dashed to the goal once near the end. Metric data for this level is shown in table 7.3.

This represents a corner case for NM. The red player received a small value, mostly because despite closely following the enemy, the player is far enough away that little cost
7.2. Results

Figure 7.10: Level 3 from the human study

Table 7.3: Level 3 Metrics

<table>
<thead>
<tr>
<th>Path</th>
<th>Dist</th>
<th>LOS</th>
<th>NM</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue path (1)</td>
<td>0.00115</td>
<td>0.00362</td>
<td>428203</td>
</tr>
<tr>
<td>red path (2)</td>
<td>0.00242</td>
<td>0.01243</td>
<td>204</td>
</tr>
</tbody>
</table>

was attributed as she dashed out and was walking outside the cost window. The distance-based weighting of the other two metrics, however, tends to give a higher value to the red player. Since humans did not achieve good agreement themselves, however, we can see this as a trade-off in risk—a brief, close call in the blue player’s path is roughly equivalent to the long, slow, moderately dangerous progress by the red player.

Level 5

This level only has a rotating camera with a long, narrow FoV; see figure 7.11. The blue player walks fully outside of this field of view, whereas the red player waits for an opening in the rotation to dash directly to the goal. Metrics for this level are shown in table 7.4.

The humans here again prioritized the distance to enemy, even though the blue player actually spent more time within the enemy’s viewpoint. Our different metrics of course prioritize this differently: LOS gives greater weight to the angular proximity, while the path-intersection of the red player results in higher NM.

193
7.2. Results

Figure 7.11: Level 5 from the human study

Table 7.4: Level 5 Metrics

<table>
<thead>
<tr>
<th>Path</th>
<th>Dist</th>
<th>LOS</th>
<th>NM</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue path (1)</td>
<td>0.00002</td>
<td>0.01798</td>
<td>0</td>
</tr>
<tr>
<td>red path (2)</td>
<td>0.00003</td>
<td>0.00223</td>
<td>6155</td>
</tr>
</tbody>
</table>

**Level 12**

In this level, shown in figure 7.12, the blue player goes around the long way, but at one point, walks in front of the FoV of the east enemy. The red enemy takes more risks by taking the short cut through the middle of the level.

The results in table 7.5 show that the LOS metric ranks the red player safer than the blue player. This is mainly because the blue player walks in front of FoV of the east enemy, causing the metric to add a high cost to that path. Dist and NM assess the situation better and more like the humans, attributing greater danger to the red player who both comes

Table 7.5: Level 12 Metrics

<table>
<thead>
<tr>
<th>Path</th>
<th>Dist</th>
<th>LOS</th>
<th>NM</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue path (1)</td>
<td>0.00127</td>
<td>28.565</td>
<td>276620</td>
</tr>
<tr>
<td>red path (2)</td>
<td>0.00191</td>
<td>0.005677</td>
<td>682589</td>
</tr>
</tbody>
</table>
7.2. Results

Figure 7.12: Level 12 from the human study

Table 7.6: Level 14 Metrics

<table>
<thead>
<tr>
<th></th>
<th>Dist</th>
<th>LOS</th>
<th>NM</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue path (1)</td>
<td>0.00025</td>
<td>2.36677</td>
<td>1196323</td>
</tr>
<tr>
<td>red path (2)</td>
<td>0.00242</td>
<td>0.127234</td>
<td>645539</td>
</tr>
</tbody>
</table>

closer the enemy, and much closer to being seen.

Level 14

Similar to level 12, the blue player in this level follows a roundabout route, trying to avoid contact with enemies, as shown in the key-frames of figures 7.6, 7.7 and 7.8. The red player (not shown) walks through the enemies’ FoV space quite quickly.

As opposed to level 12, however, the results in table 7.6 show that both LOS and NM rank the red player as being the safest, since despite the proximity to enemies the path manages to almost never cross in front of the enemy, even within a window of past and future positions. Dist agrees more with the humans, ranking blue safer since its route takes it well away from the enemies.

Each of our metrics tries to measure different properties of what might be considered risky behaviour in a stealth context. Comparison with human perception verifies that these factors are indeed important concerns for players as well, and while confirmation of our
7.2. Results

Figure 7.13: Metal Gear Solid’s first level with a static probability distribution of getting seen by the enemy. The darker the orange, the higher the chance of getting seen.

results in a larger human study is necessary, suggests there may be a useful ranking of these factors. Human results best correspond to a pure (geometric) distance measure, and our measures based on near-misses and angular proximity match less well. This may also depend on context, however, and a similar exploration using a first-person perspective visualization (rather than overhead view) would be interesting—near-misses and being apparently within an enemy’s FoV may be more important concerns if the player is more immersed in the game context, and is also less able to easily determine the extent of enemy FoV.

7.2.3 Risk Metrics for Level Analysis

This section investigates the previously evaluated metrics used as a mean for level analysis. Measurement of the different paths through a level gives us an overall measure of level difficulty, as well as distribution of solution difficulty, both of which can then be used in tuning level design.

As an initial demonstration of the technique, we applied our metrics to the first level of Metal Gear Solid [Kon98], described in chapter 4. In figure 7.14 we show a sample of 1500 paths found by our stealth design tool presented in section 6.3 on page 152 using
7.2. Results

Figure 7.14: Clustering of feasible paths within the level.

a basic RRT (algorithm 7 on page 126) that we clustered into 4 groups (colours), based on which corridor a path traverses (indicated by the circles in Figure 7.14). Note that this clustering is heuristic and based on our qualitative observations as in general solutions tended to favour specific sets of corridor connections. Please note that it is possible to have one path associated with more than one cluster.

Table 7.7 shows the average and median metric values of the different clusters. From this simple data we can see that the red cluster seems to have safer paths than the rest, especially if we look at the Dist metric. This was not obvious just from the simple, time-flattened probabilities, as shown in figure 7.13, but does mirror our informal observations, as paths in this cluster seemed to do quite well at avoiding enemies.

We present both average (denoted Avg) and median (denoted Med) since outlier paths can easily skew these simple statistics. It is, however, important to investigate these since they may represent extremal or unusual strategies. Figure 7.15 shows a detailed view of the distribution of the Dist metrics for all the clusters as the average and mean are widely different (please note that the blue cluster is hard to see). From the distribution, one can observe that even though it is not the trend, it is possible to achieve the same level of safety in all the clusters, although only the red cluster is strongly peaked at the low end. Smaller peaks in the green cluster suggest sub-clusters of higher risk, likely due to differences in timing that result in more closely encountering enemies.
7.2. Results

Figure 7.15: Histogram of the different clusters for the Dist metric.

Table 7.7: Clustering result, showing the average and median for each metric.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Red</th>
<th>Blue</th>
<th>Green</th>
<th>Magenta</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Med</td>
<td>Avg</td>
<td>Med</td>
</tr>
<tr>
<td>Dist ($\times 10^{-3}$)</td>
<td>0.6</td>
<td>0.2</td>
<td>3.7</td>
<td>0.9</td>
</tr>
<tr>
<td>LOS ($\times 10^{-2}$)</td>
<td>0.7</td>
<td>0.02</td>
<td>13.8</td>
<td>0.4</td>
</tr>
<tr>
<td>NM ($\times 10^5$)</td>
<td>2.0</td>
<td>1.6</td>
<td>2.8</td>
<td>2.4</td>
</tr>
</tbody>
</table>
7.3. Chapter Conclusion

Informed by this kind of data, a designer has a better understanding of her level design, and can use the relative risk value, as well as the distribution of risk values over different path clusters as a guide for manipulating the level design to better balance overall risk, or introduce interesting variation in strategy choices for players.

7.3 Chapter Conclusion

In this chapter we explored different approaches for evaluating stealth level difficulty. We first focused our discourse on static analysis of the stealth level space. We investigated probability density maps of getting seen as well as looking at road map analysis in order to give a visual difficulty understanding for possible in-game path solutions. Limitations to this level-aggregate approach motivated a focus on individual path behaviours. We defined three novel metrics for risk, and applied these to solutions found by our previously presented solver (see chapter 4). These metrics were validated using a human study and we showed that the Dist metric matched the human judgement more closely than the rest. From the game developers perspective these metrics extend their set of tools for understanding their design. We showed that using the Dist metric, it is possible to understand the general trends of different families of paths, a process further simplified through the use of easily generated path data derived from automatic search. Using this information a designer has the capacity to evaluate the risk distribution of her level and make informed design changes.

The following chapter will present different extensions—to the previously presented—RRT solver. First we present an extension allowing the search to include combat within the stealth domain without adding to the sampling dimensions. We also introduce an RRT controller which uses the Dist metric to validate segments before adding them to the search tree. These extensions will allow game developers to fine tune search process.
Chapter 8
Solver Extensions

The solvers presented in chapter 4 were able to show solvability of a level in the first place. Using the RRT search algorithm, we showed that we can find multiple solutions to a given level, and then use those solutions to gain knowledge about the player’s experience, as shown in chapter 7. During the design phase of the game, however, a game designer might be interested in describing the type of players that would play the level. As a first extension, we thus present a modification to the RRT-solver to let developers fine tune the game solver, e.g., forcing the solver to respect certain rules or thresholds. In other words, the designer is allowed to define player types to test her levels.

An additional change to RRT addresses limitations in the basic RRT stealth solver, attempting to make it more aware of its environment. In chapter 4, the stealth solver only used the base, geometric knowledge about the space to find solutions. This static representation limits the expressiveness of the found solutions, as in the stealth games genre the player is normally given a much richer interaction space with the game, such as by being allowed to interact with the guards, e.g., fighting, attracting, chasing, etc. In this chapter we thus also introduce an other extension to the space representation and RRT solver that allows us to incorporate guard interactions (combat) as well as using resources such as health packs. This chapter presents the overall structure of both extensions as well as tailored experiments to show the usability of each extension. Specific contributions of this chapter include the following,
8.1 Extensions on RRT

- We extend the RRT solver to include controllers in order to express a wide-range of player types and behaviours.

- We demonstrate how the analytical domain and RRT search technique can include player combat and resource management without adding search complexity. This non-trivial addition includes a modular representation of combat resolution and results in a unified context for stealth and combat analysis.

- We reveal the value of our approach on a representative game level, showing how the extensions can expose useful statistics during the game design process.

8.1 Extensions on RRT

This section presents our two algorithmic RRT extensions. We first discuss the introduction of a controller to the solver, which directs the RRT search based on the metrics presented in chapter 7. We also introduce the algorithmic components needed to extend the RRT to include guard interactions.

8.1.1 Controller

The existence of quantitative metrics (see chapter 7) on paths implies the possibility of generating paths that attempt to respect limits on these metrics: if we can measure risk post facto, then perhaps we can add a logical controller to our RRT-based path search to try and guarantee a path with a particular level of risk. In this way we can model specific player types. As an example use, our proposed approach focuses at making artificial players risk-averse, and does not allow risk-seeking players.

RRT readily admits different controllers. Our basic RRT design already includes a simple controller, guaranteeing that segments added to the tree respect a maximum travel velocity. Respecting a set of controllers is achieved by adding a validation step on line 9 of algorithm 7 of chapter 4 (page 126). The additional pseudo-code is shown as algorithm 9 below. The new approach iterates through a set of controllers, each controller receiving the proposed point and edge to be added to the tree, and returning a boolean value to indicate
whether it is acceptable or not. In this way we can guarantee that if a motion exists from
\( \sigma_{\text{near}} \) to \( \sigma_{\text{motion}} \), then that segment has to respect each controller instantiated. We now explore
the inclusion of additional, more complicated controllers based on our three path metrics.
This approach presumes that controllers are satisfied by local inspection of segment in the
tree. For global properties of a path rather than just of each segment individually we could
also pass each controller the full search tree before agreeing on a new edge.

Algorithm 9 Controller Validation

\[
\text{procedure VALIDATION}(\sigma_{\text{near}}, \sigma_{\text{motion}}, C) \\
\quad \text{for all } c \in C \text{ do} \\
\quad \quad \text{if } c(\sigma_{\text{near}}, \sigma_{\text{motion}}) \text{ is false then} \\
\quad \quad \quad \text{return false} \\
\quad \quad \text{end if} \\
\quad \text{end for} \\
\quad \text{return true} \\
\text{end procedure}
\]

Dist controller

Based on our human study we present an extension controller which wraps the Dist met-
ric. Intuitively, this controller seeks to respect a certain distance to the enemy, attempting
to ensure that the player will not cross an invisible boundary around any enemies. This
controller is inspired by the definition of the Dist metric presented in chapter 7. We verify
limits on Dist by computing a boolean function on a path segments,

\[
C_{\text{dist}}(\sigma_0, \sigma_1) = \begin{cases} 
\text{True} & \text{if } d'(\sigma_0, \sigma_1, e) > \alpha \quad \forall e \in E \\
\text{False} & \text{otherwise}
\end{cases}
\]  

(8.1)

where \( d' \) is the shortest path distance function between a segment \( \sigma, \sigma' \) and an enemy
path, \( e \), on the plane \( (x, y) \). The controller is defined by an \( \alpha \) value which gives a threshold
to respect. If the distance calculated is smaller than the threshold, the proposed segment is
dropped and the RRT search will need to try a different sampled point. Once the segment
8.1. Extensions on RRT

Figure 8.1: Heatmap movement of a trivial example with no controller where players have to move from the blue sphere to the green one. Two static enemies (represented by two yellow circles) are placed in the north hallway facing north.

has been validated against all enemies, the segment may be added to the tree structure, and the RRT search continued. The resulting path is assured to respect a certain distance to the enemies.

Figures 8.1 and 8.2 show a trivial level application of our controller. The level is composed of two static enemies placed in the north hallway (represented by two yellow circles) and they are also facing north. The player has to move undetected from the blue sphere to the green one. Figure 8.1 shows a movement heatmap of a 1000 successful paths found from our search tool using no controller. This results is in line with previously presented results, in that we see the paths are mostly uniformly distributed in covering both options, going below the obstacle and going above the obstacle closer to the enemies. Figure 8.2 shows a movement heatmap using a Dist controller of 8 meters (the relative distance is represented by a red segment). The results clearly show that no solutions took the northern route above the obstacle. In section 8.2.1 we will investigate results produced by this controller for the MGS level.
8.1. Extensions on RRT

Figure 8.2: Heatmap movement of a trivial example with a controller Dist of value 8 meters (represented by a red segment) where players have to move from the blue sphere to the green one. Two static enemies (represented by two yellow circles) are placed in the north hallway facing north.
8.1. Extensions on RRT

8.1.2 RRT Guard Interactions & Resource Management

In the previous section we presented the addition of a controller to the RRT-solver. In this section we are interested in extending the search space to include guard interaction-for stealth games. Basic path-finding using just the 3D \((x, y, \text{time})\) subset of our domain is already useful for modelling stealthy behaviours, as seen in chapter 4. The extension proposed in this section wants to incorporate guard interactions as well as resources management into the stealth search process. The intuition behind this extension is to extend the tree representation without adding dimensions to the search process; \textit{i.e.}, we keep the RRT sampling dimensions unchanged, and we simply construct the tree search from segments using new dimensions such as health.

In order to incorporate combat interactions and resource management we extend the stealth game genre tuple presented in chapter 6 to the following,

\[ \Sigma \subseteq \mathbb{R}^2 \times \mathbb{R}^{\text{time}} \times \mathbb{R}^{\text{health player}} \times (|E| \times \mathbb{R}^{\text{health enemy}}) \times (|R| \times \{0, 1\}) \]

where we explicitly add the player’s health value \((\mathbb{R}^{\text{health player}})\), which we will access using the function \(h()\), the enemy health values \((\mathbb{R}^{\text{health enemy}})\) for each of \(|E|\) enemies), and boolean existence of \(|R|\) resources.

Rather than naïvely apply RRT to this extended space, where the additional dimensions greatly complicate the search process as we would need to consider the existence of enemies, combat state, and player health in selecting and attaching new nodes to the search-tree, we use a variant on the basic RRT, where the node attached to the tree at each iteration is determined computationally from the initial, random choice. We show the resulting pseudo-code in algorithms 10 and 11, which are further described below. Note that to reduce complexity in this exposition, we initially assume that only one enemy can fight the player at a given time and that the player cannot lose a fight; these constraints will be relaxed in the following subsection.

Our approach retains the core strategy of randomly sampling and extending reachable points in the state space in order to grow a search-tree. To make this selection we ignore player health and enemy liveness, so this process starts off the same as for the basic RRT search as shown in algorithm 10. In determining whether \(\sigma_{\text{motion}}\) can be connected, however,
we may discover that the connecting segment intersects an enemy FoV as seen on line 11. If this enemy is still alive in \(\sigma_{\text{near}}\) then combat will occur. The algorithmic process is depicted by figure 8.3, where \(\sigma_{\text{motion}}\) is represented by a green circle and \(\sigma_{\text{near}}\) by a blue one. The \texttt{COLLISIONENEMY} function called on line 11 is defined by algorithm 11. This function (lines 1–13) use a helper method \texttt{LINEPRISMCOL} to compute the closest point in our time space dimension to \(\sigma_{\text{near}}\) in the intersection of \((\sigma_{\text{near}}, \sigma_{\text{motion}})\) and each living enemy’s FoV structure. In other words, this finds the intersection of a line segment (motion segment) with a polyhedron (an enemy’s FoV volume in the space-time dimension). We also use \texttt{pos( )} to access the time-space dimension tuple of a state, e.g., \((x, y, t)\) as well as \(\|v\|\) to compute the magnitude of a vector with \(n\) dimensions. The function returns the earliest state at which combat will initiate. We denote that state as \(\sigma_{\text{combat}}\) and it is represented as a red circle in figure 8.3.

Having found a state where combat occurs allows us to simulate combat in different ways. We saw in chapter 3 that simulating combat optimally for a simple attribution game is an expensive task. Thus, in our case, as a proof of concept we used a simplification of combat representation using a high level abstraction, which consists of damage per second (dps). The idea is to assign a fix value of dps to each entity (player and enemies), and using the entities’ health and dps values it is possible to infer the winner and how long the combat is. This calculation is performed by the \texttt{SOLVECOMBAT} function (algorithm 11) shown on lines 15–24. Here we simply divide the enemy health by player attack to give a duration for the combat, and use that duration to also determine how much health the player loses, which could lead the player to her death. Combat is assumed to occur in a single location, giving us a result state, \(\sigma'_{\text{rand}}\), as a state projected upward in time from \(\sigma_{\text{combat}}\) with updated player health and enemy sets, shown as the purple node in figure 8.3. Please note that the presented algorithm uses some additional accessors, such as \(E(\ )\) to access the set of enemies at a given state, and \(dps(\ )\) to access the entities’ attack values. The search-tree is then updated by adding \(\sigma_{\text{combat}}\) and \(\sigma'_{\text{rand}}\) to \(\Upsilon\) through segments \((\sigma_{\text{near}}, \sigma_{\text{combat}})\) and \((\sigma_{\text{combat}}, \sigma'_{\text{rand}})\), representing the player reaching the point of combat, and engaged in combat respectively. Note that this adds two nodes to the tree, although combat-initiation nodes such as \(\sigma_{\text{combat}}\) cannot be nearest neighbours for future \(\sigma_{\text{rand}}\) choices. In cases where no enemies can be encountered we follow the basic RRT algorithm, adding valid and reachable
8.1. Extensions on RRT

states to the tree. Since an enemy entity is fully alive or dead at tree nodes we can model
the enemies health dimension as a binary variable.

Finally, when a goal state is reached, the \textsc{Path}(\mathcal{Y}, \sigma_{\text{rand}}) method call on line 21 of
algorithm 10 retraces a shortest path, $\rho$ from the last node added to the initial state, and we
add $\rho$ to our collection of paths $P$. The entire process is iterated and controlled through
two user-defined parameters to specify how many paths the user wants to see ($M$), and how
large she wants the tree to grow ($N$).

**Player Death and Overlapping Combats**

In using the above algorithm in the context of design exploration, it is extremely useful to
know not just where and how a player may succeed, but also where players may be un-
successful and have died. This allows a designer to gain richer knowledge about solutions
and difficulties of her creation. To retain this information, we drop the requirement that the
player always wins combat, and allow states with $h(\sigma) \leq 0$ to be added to the tree. This
adds a minor complication, in that when finding the nearest state to $\sigma_{\text{motion}}$ we also have to
make sure that the player is still alive. We will present results using this augmentation in
section 8.2.2.

Given the basic design, integration of combat with multiple enemies is conceptually
trivial. Multiple enemy combat occurs when another enemy is able to observe the combat
between the player and an enemy, and so joins in the attack on the player. In our model
this is true if the segment $(\sigma_{\text{combat}}, \sigma'_{\text{motion}})$, representing combat with enemy $e$ intersects
the FoV of a different, living enemy, $e'$. If so, we assume the player continues to fight $e$
to completion, queueing the combat with $e'$ until the fight with $e$ completes. This has the
effect of revising $\sigma'_{\text{motion}}$, extending combat to last the full duration of $(h(e) + h(e'))/a$. As
well as projecting $\sigma'_{\text{motion}}$ further in time, we also update $\sigma'_{\text{motion}}$ to reflect the death of the
additional enemy, and to include the additional health loss to the player of being attacked
by both enemies during the overlap of combat with $e$ and $e'$, and which was suffered during
the queued combat with $e'$. This process has a straightforward extension to accommodate
an arbitrary number of enemies within the same combat sequence. It is also an approach
that is quite modular in the detail of combat representation, and although we have used a
8.1. Extensions on RRT

Algorithm 10 RRT algorithm that includes combat and resources management. Please refer to algorithm 11 for CollisionEnemy and SolveCombat function definitions.

1: procedure COMMPATHS($\sigma_{init}, \sigma_{goal}, \Sigma_{free}, N, M$
2:     for $i = 0$ to $M$ do
3:         Initialize($\Upsilon, \sigma_{init}$)
4:             for $j = 0$ to $N$ do
5:                 $\sigma_{motion} \leftarrow$ Sample($\Sigma_{free}$)
6:                     $\sigma_{near} \leftarrow$ Nearest($\sigma_{motion}, \Upsilon$)
7:                         if $h(\sigma_{near}) < 0$ then
8:                             break
9:                         end if
10:                     if CollisionFree($\sigma_{near}, \sigma_{rand}, \Sigma_{free}$) then
11:                         $\sigma_{combat}, e \leftarrow$ CollisionEnemy($\sigma_{near}, \sigma_{motion}$)
12:                             if $\sigma_{combat} \neq$ nil then
13:                                 $\sigma'_{motion} \leftarrow$ SolveCombat($\sigma_{combat}, e$
14:                                     $\Upsilon \leftarrow (\sigma_{near}, \sigma_{combat}$
15:                                         $\Upsilon \leftarrow (\sigma_{combat}, \sigma'_{motion}$
16:                                             $\sigma_{motion} \leftarrow \sigma'_{motion}$
17:                             else
18:                                 $\Upsilon \leftarrow (\sigma_{near}, \sigma_{motion}$
19:                             end if
20:                     if $\sigma_{motion} \in \sigma_{goal}$ then
21:                         $\rho \leftarrow$ Path($\Upsilon, \sigma_{motion}$
22:                                 $P \leftarrow \rho$
23:                             break
24:                         end if
25:                     end if
26:     end for
27:     end procedure
Algorithm 11 Extra procedures called by the RRT-Solver, see algorithm 10

1: procedure COLLISIONENEMY($\sigma_{\text{near}}, \sigma_{\text{motion}}$)
2:    $\sigma_{\text{combat}} \leftarrow \text{null}$, $e \leftarrow \text{null}$, $d \leftarrow \infty$
3: for each $e_i \in E_{\text{alive}}(\sigma_{\text{near}})$ do
4:    $v \leftarrow \text{LinePrismCol}((\sigma_{\text{near}}, \sigma_{\text{motion}}), g(e_i))$
5: if $v \neq \text{nil}$ $\land$ $\left\lVert \text{pos}(\sigma_{\text{near}}) - v \right\rVert < d$ then
6:    $\sigma_{\text{combat}} \leftarrow \sigma_{\text{near}}$
7:    $\text{pos}(\sigma_{\text{combat}}) \leftarrow v$
8:    $d \leftarrow \left\lVert \text{pos}(\sigma_{\text{near}}) - v \right\rVert$
9:    $e \leftarrow e_i$
10: end if
11: end for
12: return $\sigma_{\text{combat}}, e$
13: end procedure

14:
15: procedure SOLVECOMBAT($\sigma_{\text{combat}}, e$)
16:    $\sigma_{\text{motion}}' \leftarrow \sigma_{\text{combat}}$
17:    $E(\sigma_{\text{motion}}') \leftarrow E(\sigma_{\text{motion}}') \setminus \{e\}$
18:    $\Delta t \leftarrow h(e)/\text{dps}_{\text{player}}$
19:    $t(\sigma_{\text{motion}}') \leftarrow t(\sigma_{\text{combat}}) + \Delta t$
20:    $h(\sigma_{\text{motion}}') \leftarrow h(\sigma_{\text{combat}}) - \text{dps}(e) \cdot \Delta t$
21:    $h(e) \leftarrow h(e) - \text{dps}_{\text{player}} \cdot \Delta t$
22:    $E(\sigma_{\text{motion}}') \leftarrow E(\sigma_{\text{motion}}') \cup \{e\}$
23: return $\sigma_{\text{motion}}'$
24: end procedure
8.1. Extensions on RRT

Figure 8.3: Combat algorithm representation
very simplistic calculation for combat outcome, integration of more elaborate models that optimize combat choices, such as we considered in chapter 3 is part of our future work.

**Visualization and analysis**

Appropriate representation and analysis of the paths found is important for practical use of this approach in level analysis and design. Examining any single, given path is straightforward: every state has a time component, so we can visualize the path by replaying the player and enemy motions in real-time. Figure 8.4 shows a snapshot of this approach from our implementation, using a blue sphere for the player, yellow spheres for living enemies, line segments to indicate the player’s historical path (blue) and current combat engagement (red), and terrain colourization to show enemy FoVs (based on a discretized space model). In this way we can see the exact progress and behaviour of a possible player path.

Visualizing multiple paths is complicated by the fact that paths in general have different sets of live enemies, making it difficult to determine which set of enemies is relevant to each path. For understanding multiple paths, we thus again make use of simple heat maps, flattening all paths in the time dimension, and shading terrain cells according to the density of path points with the same $x, y$ coordinate. Applying this technique to show player movements can be quite illustrative, revealing locations or enemy positions which successful player paths avoid or commonly target.

Of course, and by intention, the paths generated by our search process constitute only successful gameplays—the player reaches the goal state. As mentioned earlier, designers are also interested in where combat occurs, and in particular where players are likely to fail (die). To display the latter information we need to extract data from the nodes attached to all search-trees, and not just the successful paths. When a state is added to the tree, we thus check if $h(σ'_{motion}) \leq 0$, and if so we save $pos(σ'_{motion})$ in a data structure. We then display this information as a separate heat map, showing the relative density of player deaths at each location. Since we have time data in these points as well, this heat map can also be examined as a real-time replay, displaying deaths as animations over an interval $t \pm \alpha$ for some $\alpha$. A similar process can be applied to create visualizations of combat or enemy deaths.
Figure 8.4: A player path (blue line) displayed in Unity 3D, with a player (blue sphere) fighting (red segment) with an enemy (yellow sphere).
8.2 Experiments

This section presents non-trivial examples application of our extensions. We first discuss using the Dist controller in the first MGS level used in chapter 4. We then present a new set of experiments to show the potential of our guard interaction RRT-solver.

8.2.1 Controller

Based on the human-study discussed in chapter 7, the Dist metric matched better the human perception of risk. This particular controller allows a designer to generate and visualize the behaviour of a simulated player that respects a certain distance to the enemy. The type of player we will investigate in this section is a player that acts carefully and does not want to get too close to the enemy.

Using the first Metal Gear Solid level, previously described, we used different distances to control our search. Using a distance of 8 meters to be respected by the player to the enemy shows the most interesting data. The found paths do not go in the south-left and north-right corridors any more as we can see on figure 8.5 compared to the heatmap of figure 6.9 on page 159, which did not use any controller. Using the south and and north corridors implies that the artificial players have to walk closer to the enemies. This tool allows designers to fine tune how the search is going to unfold, helping them understand certain type of behaviour such as risk-averse players. The main purpose of this experiment is to demonstrate the feasibility of incorporating a controller. The process, however, can be applied to any relevant metric, and it would be interesting in future work to include more complex game player models such as the one used by Thue et al. [DT08].

8.2.2 RRT-Combat

In this section we will explore different applications of the presented RRT-Combat framework. We first present a simple case scenario that demonstrates basic usage and results, and then apply the technique to a more complicated context with multiple enemies, and where we introduce health packs. For every test, 1500 searches were performed, with each search capped at sampling 25000 states; i.e., \( N = 1500 \) and \( M = 25000 \) in algorithm 10.
Figure 8.5: 1470 paths on the Metal Gear Solid dock level, with the Dist controller at 8 meters
8.2.3 Simple Level Combat RRT

Our first test illustrates how the search process can traverse combat situations and help identify situations in which combat is possible or must be avoided. For this we use three different parametrizations of the same level, heat map results for which are shown in figure 8.6. In this level the player starts at the blue sphere and has to reach the green sphere, pathing around an obstacle and possible encountering a single, statically positioned enemy on the north side.

In figure 8.6a the enemy has a health of 10, while the player has infinite health. This makes combat trivial, and we see that paths both pass through the enemy FoV, and thus engage in combat, and bypass the enemy: both combat and stealth solutions are viable. An interesting variation is shown in figure 8.6b, where we give the enemy 200 health. This results in a longer combat sequence (average path length increases from ~1000 time steps to ~1200), but since our search process does not favour or optimize for faster paths no significant difference in the resulting movement heat maps is evident.

If player health is limited we expect the stealthy option to be strongly preferred. This is shown in a limiting sense in figure 8.6c, where we give the player just one health unit, and the enemy 10. This makes successful combat impossible for the player, and indeed the heat map of movement shows only paths that avoid the enemy. Interestingly, these are the fastest paths at just ~500 time steps on average.

8.2.4 Non-Trivial Combat Level RRT

In this subsection we explore a non-trivial level with four enemies, including a boss. Figure 8.7 maps out the setup; the player must traverse a fairly large space containing 3 patrolling enemies with 33 health and 3 attack each; enemies $e_1$ and $e_3$ have the most complicated paths, covering a significant part of the level. Enemy $e_4$ blocks the exit of the level and acts as a boss fight, with 99 health and 10 attack. The player has 100 health and 10 attack. Note that paths do sometimes overlap, and so it is possible that the player may need to fight two enemies simultaneously. This level is designed in such a way that the player can only reach the exit if she has full health, and fighting any of $e_1, e_2, e_3$ will not leave enough health to complete the boss fight (and vice versa, fighting the boss will not leave
8.2. Experiments

(a) Enemy with health 10  
(b) Enemy with health 200  
(c) Player with health 1

Figure 8.6: Movement heat maps for different enemy and player health parametrizations. The player starts at the blue sphere and has to reach the green sphere. A single enemy is represented by the yellow dot and yellow FoV.
8.2. Experiments

enough health to fight any minion).

The search process found 386 successful paths out of our 1500 searches, and figure 8.8a shows the resulting movement heat map. Numeric data in table 8.1 verifies that these paths all involved a single enemy combat (with $e_4$). The location of player deaths and combats also supports this interpretation. Figure 8.8b shows where players died, which was as expected primarily against $e_4$, while figure 8.8c shows that in general combat happened along the most straightforward route used to reach the goal.

8.2.5 Adding Resources Combat RRT

From a game designer perspective, the non-trivial level might seem overly biased to stealth, as the player has to avoid all moving enemies in order to reach the goal. In many FPS games the player is given more flexibility in choice through the inclusion of supportive resources, such as health packs or ammo. Our framework easily extends to include such basic resources. For example, in order to incorporate health packs, we simply add another dimension to our state $\sigma$ to specify the existence (or location) of available health packs. We also modify movement to represent player decisions to seek out health packs—when adding a state $\sigma$ to the tree, we check if $h(\sigma) < 100$ and a health pack is reachable without requiring combat. If so, we create a new node $\sigma_{health}$ with the player’s health updated by the health pack’s value and the used health pack removed from the set of all health packs. Then the new node $\sigma_{health}$ and segment $(\sigma, \sigma_{health})$ are added to $\Upsilon$. A similar process could be applied to other resources such as ammo.

Inclusion of health packs modifies the search results, although the resulting behaviour also depends on where the packs are located. In figures/extension 8.9 through 8.11 we show results for different placements and numbers of health packs, indicated visually by green squares. Table 8.1 gives the corresponding numeric data.

Far Health Pack

In order to improve the level design, we first included a health pack near $e_1$. This is relatively far from the player’s typical path, but allows a player to fight even all 3 minions ($e_1, e_2, e_3$) and then pick up the health pack before fighting the boss. Although this scenario
8.2. Experiments

Figure 8.7: Enemy paths for the non-trivial level.
8.2. Experiments

(a) Movement of 386 paths found  
(b) Death locations of 9262 states  
(c) Combat locations

Figure 8.8: Non-trivial level

(a) Movement of 474 paths found  
(b) Death locations of 1276 states  
(c) Combat locations

Figure 8.9: Non-trivial level with health pack (green square) near $e_1$

is unlikely, introducing the health pack facilitates the completion of the level as only 1276 dead states were found compared to 9262 in the base non-trivial level. In figure 8.9a we can see 474 successful paths found out of 1500, where about 60% of them used the health pack. Comparing the death locations of figure 8.8b to figure 8.9b, we can observe that the player still tended to die near $e_4$, although there is a significant reduction in quantity (by a factor of $\sim 7$). Including a health pack also shifts the combat behaviour. In figure 8.9c we now see more combat around $e_1$’s position, and from the data in table 8.1 we can also see that most of the paths now involve fighting 3–4 enemies, although almost half of the solutions still fight only one enemy, and so are not making good use of the health pack.
8.2. Experiments

Table 8.1: Numeric data for different level designs

<table>
<thead>
<tr>
<th></th>
<th>No health pack</th>
<th>Far health pack</th>
<th>Near health pack</th>
<th>4 health packs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total paths</td>
<td>386</td>
<td>474</td>
<td>969</td>
<td>982</td>
</tr>
<tr>
<td>Enemies killed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>42 %</td>
<td>7 %</td>
<td>5 %</td>
</tr>
<tr>
<td>2</td>
<td>100%</td>
<td>5 %</td>
<td>56 %</td>
<td>23 %</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>25 %</td>
<td>30 %</td>
<td>32 %</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>28 %</td>
<td>7 %</td>
<td>40 %</td>
</tr>
<tr>
<td>Player deaths</td>
<td>9262</td>
<td>1276</td>
<td>1616</td>
<td>713</td>
</tr>
</tbody>
</table>

Close Health Pack

A designer is most likely going to put a health pack where players tend to fail. In figure 8.10 we thus place it near $e_4$. This positioning results in 969 paths out of 1500 being successful, a significant improvement over the no and far health pack approaches. Interestingly, the number of player death states is greater than with the far health pack; this is explained by the limit on searchable states, $N$, which in the case of the far health pack was reached sooner since it encouraged greater area exploration. Here the distribution of enemies killed changes again, with most of the paths fighting 2 enemies, $e_2$ and $e_4$ as indicated by figure 8.10c. Again, however, most player deaths were found in front of the boss enemy.

Four Health Packs

Having only one health pack was perhaps restrictive, as the player had to plan when to use the health pack, which was best done just before fighting $e_4$. To simplify this, we added four health packs in the middle of the level. This reduces the need for a singular strategy, and also makes the level easier to win. As table 8.1 shows, the number of player death states is again reduced, while maintaining the large number of successful paths found with the near health pack. Visual results are shown in figure 8.11. Fights are now more likely to happen around the locations of the health packs, and in figure 8.11b, we can observe that $e_4$ is no longer the primary source of player failure, switching instead to a location where
8.3. Chapter Conclusion

(a) Movement of 969 paths found  (b) Death locations of 1616 states  (c) Combat locations

Figure 8.10: Non-trivial level with health pack (green square) near $e_4$

(a) Movement of 982 paths found  (b) Death locations of 713 states  (c) Combat locations

Figure 8.11: Non-trivial level with 4 health packs (green squares)

combined combat with $e_1$ and $e_3$ is likely. Adding the four health packs also lets the player move more freely; table 8.1 shows that over 70% of paths kill three or four enemies.

8.3 Chapter Conclusion

This chapter presented possible extensions to the RRT solver. We first investigated the incorporation of controllers, which have the task to validate any segment added to RRT search tree. We focused our effort in describing a controller that allows a designer to define risk averse players based on the previously described risk measurement metric. This
chapter also described a way to generalize the RRT search process to incorporate combat and resource management without changing the overall algorithmic design presented in chapter 6. We then demonstrated how these techniques could be used by designers to better understand their design and iteratively improve it. Again, this effort is greatly facilitated by the use of a large set of algorithmically generated paths, which allows for rapid and easy exploration of the effect of different design decisions.

In order to composed a broader range of computing techniques for the game design process the following chapter will investigate how computers can be used to generate content. We will focus our effort on placing decorative content into a game level.
In order to create an interesting in-game experience a level typically includes a variety of non-essential, *decorative* content, ranging from textures to interactive objects. The placement of this content is still important to the player’s experience in terms of inducing immersion and inspiring exploration, but is less well defined, and typically placed through manual designer effort. In this chapter we explore a computing approach for generating this kind of content.

Our focus in this chapter, again, is to demonstrate the use of algorithmic analysis in the design and evaluation. Our approach considers possible solutions (paths) through a level, computing a region of *weak visibility*, such that every point in that region is visible from somewhere on the path. We use that to determine what players may or may not encounter, and thus where content should be located so as to either be seen on all paths, be (relatively) unique to a path, or not typically be encountered on any common path. We validate our design through a human study demonstrating that player choices are materially affected by our placement strategies, and thus can serve as a mechanism to encourage exploration or replay.
9.1 Visibility for Content Placement

In this chapter we are interested in describing an algorithmic approach for placing decorative content with specific characteristics. We focus our effort into structuring our approaches to respect conditions such as placing content where every player in the level sees it (shared content), content only seen by unique pathing options (unique content), and content never seen by any simple pathing options (never seen). These different conditions allow for simple automation during the design process. Developers can then use this knowledge to select specific content to be placed accordingly. It is also possible to use these approaches within systems producing end-to-end levels. Thus, in this section we present the different algorithms used to produce unique, shared, and never seen content in a level using visibility polygons. The method is composed of three major components: finding all path solutions to the given level, describing a path as a weak-visibility polygon, and finding the content locations.

9.1.1 Finding Paths

The user of this technique provides a level, \( L \), which we assume is composed of a 2D polygonal geometry (an overhead view). In figure 9.1, for example, the lighter area describes a polygon, with the central obstacle defining a single large hole. Within the level the user also provides starting, \( s \), and goal, \( t \), vector positions, as well as the desired number of shared, \( m \), and unique, \( n \), content locations. In order to find all the possible paths from \( s \) to \( t \), we proposed two approaches based on either using a roadmap and depth-first search, or using individual level traces\(^1\) along with clustering.

Roadmap traces

We treat the problem of finding possible paths from \( s \) to \( t \) as equivalent to enumerating all simple paths from a corresponding node \( v(s) \) to a corresponding node \( v(t) \) in a graph \( G \), generated from the roadmap. Note that this excludes cyclic routes, and so approximates direct path solutions rather than exploratory ones. Arbitrary roadmaps may be used, as

\(^1\)produced by agent or human players
9.1. Visibility for Content Placement

Figure 9.1: Level geometry with a triangulation-based roadmap (thin gray lines) and the graph representation built from its dual (gray circles and thin purple lines).

long as they result in a connected graph. We presented such approach to create roadmaps in chapter 7, we use a similar approach for this work. A roadmap is constructed from a triangulation of the level geometry, building a graph $G$ by associating the centres of triangles with nodes and adding edges between nodes if their triangles share an edge. Figure 9.1 shows such a triangulation and the roadmap retrieved from the dual-graph. For presentation purposes, the roadmap segments are drawn passing through the midpoint of each shared edge. In our implementation, the user can provide his own roadmap or use the presented technique to produce simple paths.

Given $G$, we can then proceed in finding all simple paths from $s$ to $t$. As $s$ and $t$ are arbitrary 2 dimensional vectors, we begin by first locating the triangles enclosing $s$ and $t$ respectively, giving us appropriate starting and ending graph nodes $v(s)$ and $v(t)$. 

225
9.1. Visibility for Content Placement

A recursive search is then conducted, using DFS to generate the set of all simple paths between \( v(s) \) and \( v(t) \). For games that allow for multiple entry and exit points this process can be repeated for all combinations of entry and exits. Although the problem of finding all simple paths between two (sets of) nodes has a worst-case exponential complexity, game roadmaps are relatively small and sparse (and here planar), and our experience is that a brute force approach is sufficiently fast at the scale of reasonable game-levels.

**Clustering traces**

In some situations possible player behaviours might be too complicated to be generated by a simple roadmap graph. Guard interactions in a stealth level, time-dependant actions, combat with enemies, etc., tend to require players make careful choices in movement strategy, resulting in a much less exhaustive and less straightforward set of possible paths between start and goal than the underlying geometry may itself allow, as we saw in chapter 8. An alternative solution is then to build possible paths based on movement traces from actual gameplay. This could be achieved by recording human players as they go through the level, or mimicked by using a randomized game solver such as a rapidly exploring random tree to algorithmically generate a set of solutions as seen in chapter 4. Human and randomized traces tend to show significant fine-grain variability, so when adopting this approach it would also be necessary to group similar traces together using path clustering algorithms, such as the one proposed by Campbell et al. [CTV15]. Once we have a set of path clusters, we can then construct representative paths from the centroid trace of each cluster.

9.1.2 Visibility Polygons

Using the collection of paths from \( s \) to \( t \) previously computed, we build a path weak-visibility region collection. This stage is separated into two steps: (1) constructing the visibility region from a single source, and (2) building the visibility polygon for the path.
9.1. Visibility for Content Placement

Figure 9.2: The visibility region (light pink) seen by vertex $q$ (green vertex), the region is constructed from the apex vertices (red) and projections (orange), and the non-apex vertices (blue and black).

Figure 9.3: The light pink regions represent two FoVs, $fov_p$ depicts the full, non-obstructed FoV of a point, whereas $fov_q$ results from the intersection of a FoV with its visibility region.
9.1. Visibility for Content Placement

Visibility region

The study of visibility is one of the cornerstones of computational geometry as it arises naturally in computer graphics, robotics, digital games, etc. Many forms of visibility problems have been extensively studied in two or higher dimensions. Given a polygon \( P \) of \( n \) vertices with \( h \) holes, the method to define a visibility polygon from a single point, \( q \), is a well established problem \([Gho07]\), of time complexity \( \Theta(n + h \log(h)) \). We use the well known angular plane-sweep algorithm \([Asa85]\) to construct a visibility region \( V(q) \), giving us a star-shaped polygonal region defined by the existing edge set, filtered according to visibility from \( q \). Figure 9.2 shows such a region in light purple for point \( q \).

We are actually more interested in building the weak visibility polygon \( V(s) \) of a segment \( s \), where every point in \( V(s) \) is visible from at least one point on \( s \). Computing weak visibility has a surprisingly high theoretical complexity, and pathological cases can be constructed where \( V(s) \) has \( \Omega(n^4) \) vertices. Suri and O’Rourke gave a worst-case optimal algorithm, showed that weak visibility from a segment can be computed in \( O(n^4) \) time \([SO86]\). To reduce the implementation complexity, however, we resort to a heuristic, discretized form of weak visibility, which we will present as part of our algorithmic methodology.

Point visibility assumes a 360°, infinite range field of view (FoV). In many games, the player’s ability to see is described by a more limited FoV, forming a cone of finite length and angle. This can be modelled in terms of straight-line polygons by a single triangle, or with more precision by multiple triangle strips. In our case we use two triangles as an efficient but still acceptable approximation, as shown in figure 9.3, as the pink area on the right. In order to construct the final visibility region we then intersect this polygon with the visibility region, giving us the final FoV region for a given point. The light pink region in the top left of figure 9.3 shows an example of the result.

Path visibility

Full path visibility implies a weak visibility calculation, computing the visibility region discernible from any point on the path. As previously discussed, weak visibility from even a single segment has high theoretical complexity. Our approach to path visibility is thus...
9.1. Visibility for Content Placement

Figure 9.4: Computing weak visibility. A small interval size results in FoV polygons that would merge to fill the corridor (top corridor, left side), but with a larger step size can miss portions (top corridor, right size). The extruded FoV polygon (bottom corridor) is necessarily fully merged.

A heuristic, based on combining a set of discrete, point-visibility computations, calculated at constant intervals along the path. This under-approximates actual weak visibility, but has a much lower implementation effort, and the accuracy versus cost trade-off can be easily controlled by altering interval size. For each point we build the point-visibility polygon, intersecting it with the player’s FoV assuming they were looking towards the next point, or keeping the orientation from the last point. The resulting FoV regions are merged for each point in the path, giving us the final path visibility.

Naively done, and depending on the interval granularity, this calculation can result in a stepping effect, where portions of the actual visibility region are missing in our result. The top corridor of figure 9.4 shows two path visibility regions with different intervals. The movement on the left side uses a short interval, and the union of the two FoVs thus covers the full corridor. On the right side, a long interval is used, resulting in multiple missing triangles that actually would have been seen by the player walking along the segment.
Determining an optimal granularity is difficult—outside of simple corridor situations, ensuring sufficient overlap of individual player FoV polygons to avoid these missing areas can require arbitrarily small interval granularity. We notice, however, that the union of visibility polygons computed from the set of player positions does in fact include the missing areas, and they become excluded only when we intersect with each individual FoV. To avoid this problem, we thus compute the fully merged set of point-visibility polygons, intersecting it with a single polygon constructed by sweeping or extruding the player FoV from the starting position at the beginning of the segment to the ending position at the end of the segment, as shown in the bottom corridor of figure 9.4.

9.1.3 Content Locations

To insert content items we can now assume a set of paths, each of which is associated with a visibility region. On each path we must then allocate $m$ content items so they are all encountered (seen) on every path, and $n$ content items that will be unique to each path. This allocation process is accordingly separated into two sections, one to place shared items, and one for unique content.

Shared Content

Shared content is placed within the visibility region(s) seen on all our input paths. Given the visibility regions, this common polygonal environment is easily computed as the intersection of all such regions. Note that while having common starting and ending positions implies some overlap must exist, this can in general result in multiple, disconnected polygons. To find a suitable location within these shared polygons, we first sort the set of shared polygons by area. This allows us to prioritize placement within relatively large regions, and so (heuristically) better ensure the item will be seen on each given path. Our polygons are not necessarily convex, and so placing an item within a polygon is not trivial. We thus find a representative point, a specific location guaranteed to lie within the polygon, by triangulating the polygon and selecting the centroid of the largest triangle. Subsequent placements of our $m$ items descend down our polygon priority (size) list, and may also make use of different triangles within the same polygon. Figure 9.5 shows the final results for 2 shared
9.1. Visibility for Content Placement

Figure 9.5: Final results including one shared content (yellow) and one unique content (red) per simple path from \( s \) to \( t \).

content items, represented by two yellow circles.

**Unique Content**

Unique content needs to be placed such that each of \( n \) items is only encountered on one of our paths. For this we can compute the difference between each visibility region and all others, and apply the same process as for shared content to locate items within the resulting set of polygons. Figure 9.5 shows the results for one unique content item (red circle) per path.

**Content Collision Ratio**

In more complex levels with many paths of interest, path overlap can easily be such that locations for content unique to each path may not exist. To deal with this, we also define a more flexible content collision ratio (CCR). This scalar value represents the maximum ratio of other paths on which a given unique content may be encountered. Given \( P \) total paths and a content which we intend to locate on one of those paths, a \( CCR \in [0...1] \), allows
9.2. Experiments

a maximum of \( \lceil \text{CCR} \times (P - 1) \rceil \) other paths to also cover the content. A CCR of 1 then allows the content to be placed in a location that may be covered by all paths, a CCR of 0 will require the content be placed in a location that no other paths cover and hence is truly unique, and values between are proportionally permissive.

It is important to note that these content placement requirements are strict, and may thus fail for given geometries, path sets, and values of \( m \) and \( n \). The content locations will also of course be related to the FoV used, with different FoV assumptions, such as infinite range and/or 360° visibility, producing different results and making it more or less likely that satisfactory content placements exist.

9.2 Experiments

In this section we analyze the effectiveness of our approach on a non-trivial example level. We first verify that we can place shared, unique, and “never seen” content (the inverse of shared). Then we investigate the impact of these approaches through a small-scale human study. We implemented the method described in the previous section using python 2.7, the Shapely library which wraps the geometric engine - open source (GEOS) [The15], and the poly2tri which provides Delaunay triangulation of polygons with holes [HGÅ12].

9.2.1 Wasteland

In order to test our presented approach we modelled the Temple of Titan level from Wasteland 2 [inX14]. Figure 9.6 shows this level with its multiple entry and exit points indicated by red dots. Once entering this level the player has to reach the green position in order to advance the narrative and then exit the level through any of the red points. Note that we only modelled the general level structure, and did not represent game or puzzle elements such as doors, enemies, and collectables. In order to show the potential of our method we explored placing content for a single entry and exit point (\( s \) and \( t \) respectively in figure 9.7, although technically \( s \) is not an entry/exit). Use of multiple starting and ending points increases analysis effort and can reduce opportunities for sharing content between paths, but
9.2. Experiments

Figure 9.6: Wasteland level showing weak visibility region of a path
9.2. Experiments

does not change the basic approach. We consider placement of shared content, unique content using different ratios, and never seen content as a means of encouraging exploration. Please note that this level, and parts of the human study, was also used in chapter 6.

**Shared Content**

In designing a level, a developer may want all players to encounter some key content in the game. This only happens at the intersection of visibility regions of all the simple paths, which for a single start and goal minimally includes the start and goal nodes. Figure 9.7 shows 2 locations in yellow which all the player paths between $s$ and $t$ necessarily see. Note that the exact starting location was not selected as the resulting polygon was too small for the example content we considered.

**Unique Content Ratio**

When a level offers multiple overlapping simple paths, as shown in figure 9.6 it can be impossible to use our simple approach for finding truly unique content locations. In order to assure some uniqueness, but with more latitude and thus more likely to be successful, we use the CCR ratio previously presented. For this we used a brute-force approach where we check all possible combinations. For example, if the ratio is 0.5, a unique content is allowed to collide with half of the path visibility regions. We thus look at all possible combinations that use half of the paths or less until we have a candidate with an area region large enough to place the content item. In order to avoid having always the same region polygon returned for unique content, these combinations are shuffled for every request. This is in general an expensive process overall, but could be made more efficiency with a less brute-force approach, and capped to fail in place of excessive computation time. An example of the result of this process can be seen in figure 9.8, where unique content is placed for each path region option using a ratio of 0.5.

**Never Seen**

In order to encourage exploration of the level, we can also look at the difference between the level polygon and the union of all the path visibility regions. The resulting polygons
Figure 9.7: Two shared content (yellow dots).

represent regions a player does not necessarily need to explore in order to solve the level. Figure 9.9 shows the results for the Wasteland level with never seen content (cyan points) placed in various dead-ends and near unused exit points.

9.2.2 Human Study

In order to evaluate the different presented methods, we ran an online study that we also used in chapter 6. We created a simple exploration game as a first person, 3D simulation, with the player required to find the exit from a fixed starting point. Each player was randomly assigned one of the three presented layouts representing different content distributions (figure 9.7, 9.8 or 9.9). Each content location had a unique art asset, e.g., an angel statue, barrel, well, etc.. We had 73 participants drafted from the internet (Unity3D subreddit), where 20 played the shared content level, 26 the unique content with ratio 0.5, and 27 the never seen content level.
Figure 9.8: One shared content (magenta dots) per visibility path region, using a ratio of 0.5.
9.2. Experiments

Figure 9.9: The never seen location (cyan dots).

Figure 9.10: Movement heatmap for the shared content level configuration
9.2. Experiments

Figure 9.11: Movement heatmap for the unique content level configuration

Figure 9.12: Movement heatmap for the never seen content level configuration
9.2. Experiments

Figure 9.13: Distribution of time spent (s) in the level for each layout; the shared content (blue), unique content (magenta) and never seen (green) with their averages (dashed lines) and medians (dash-dot lines). The continuous lines show an equivalent normal distribution for the calculated mean and standard deviation.

Figure 9.14: Distribution of the distance travelled (m) in the level for each layout; the shared content (blue), unique content (magenta) and never seen (green).
Figures 9.10, 9.11 and 9.12 show the movement heatmaps for each layout. Qualitatively, these results show interesting differences. In the never seen layout (figure 9.12), we can see that players explored much more of the space than in the other layouts. This suggests that our design for placing content outside the main path options encourages exploration.

For a more quantitative view, we also looked at two other measurements: distance travelled and time spent in the level. Figures 9.13 and 9.14 show the distribution for each measurement. In general, players with the never seen layout spent more time and travelled a longer distance than the others. Interestingly, the unique content case also seems to influence the players to move slightly faster towards the goal, as seen by a more skewed distribution and a median of 66s compared to 71s for the shared content layout. A similar observation can be made for the distance travelled. It is possible that unique content items acted as signposts, leading players along an expected path toward the goal. A larger scale study would help verify this, and additional experiments would also be useful to rule out the influence of other factors, such as the number of content items, which may also be influencing pathing decisions. Overall, though, we can see that all three layouts achieve different outcomes in terms of players' movement behavior, and so our process for placing content has a controllable and potentially meaningful impact on the player experience.

9.3 Chapter Conclusion

Observing game content, even narratively unimportant elements, is a major player motivation in most game genres. Leveraging that desire is thus an important aspect of game design, and processes to locate decorative content relative to likelihood of being seen are useful tools in encouraging exploration. Our visibility-based design provides an elegant algorithmic approach to placing such content, with the human-study validation demonstrating that it has a measurable impact on player movement.

We are interested in further investigating the human motivations for pathfinding, as this information is essential in developing better positioning algorithms as well for movement simulations. Informed by human models, architects have multiple school of thoughts about
9.3. Chapter Conclusion

where you should put circulation paths as well as content locations [Fre07, AIS77], and this
general approach could be further exploited to make an improved, methodically defined
game experience.

In the context of computing techniques in the game design process, this chapter pre-
sented useful techniques to gain deeper knowledge about the level configuration and how
players might interact with it through weak visibility polygons. It is also possible to include
the presented algorithms in an end-to-end system for producing levels, where the system
produces the level polygon, places interactive and non interactive content and assures solv-
ability. Generative methods are important for designers as they can provide them with an
early draft level, from there they can easily extend it to meet their visions.
Chapter 10
Related Work

In this chapter we will discuss and present the related work touching the different subjects presented in this thesis under the concept of game design mechanisation. We first discuss subjects related to in-game interactive content adaptivity (section 10.1), followed by the research related to attrition games (section 10.2). We also investigate how solvers have been applied to games, the game design process and in finding stealthy paths (section 10.3). We present different game design tools which use computational techniques as a mean to help designers gain insights about their creations (section 10.4). We discuss the different approaches that have been taken to measure the in-game player experience (section 10.5). We then investigate different research work related to simulating human movement using computational techniques (section 10.6). We conclude this chapter discussing the different work established in generating decorative content for games (section 10.7).

10.1 Adaptivity

The video game adaptivity consists of changing and tweaking the interactive content to fit a player needs. One popular game aspect that fits well adaptivity is in-game difficulty, where the standard approach to Dynamic Difficulty Adjustment (DDA) is to monitor the player, and based on the observations make decisions on how to improve the game experience, apply the changes, and repeat [LB11]. There have been multiple examples of this
10.2 Attrition Games: Targeting Enemies

basic approach in game analysis, using different mechanisms to control difficulty. Hunicke
was interested in building an adaptive system for a level in the FPS Half-life [Hun05]. She
argues that the dynamic economic system—between the player and the game—can be
controlled in order to produce smooth adaptivity using techniques from inventory theory
(supply vs. demand). Ocio presented adaptive behaviour trees to change the order of action
nodes based on in-game knowledge. This tree was then used to build levels tailored to the
player. Their goal was to accommodate casual players by decreasing the difficulty [Oci12].

The focus of companion adaptivity research, as presented in this thesis, has been to-
wards understanding the player’s actions or improving performance. Macindoe et al. used
a POMDP model game structure to develop a companion that understands how its actions
influence human intentions in puzzle games [MKLP12]. This kind of reasoning enables
the companion to interact indirectly with the player and thus solve game puzzles. Tan and
Cheng proposed a neural network approach to improve survival chances of a group of NPCs
and a player in an apocalyptic zombie world [TC07]. By adapting action selections based
on previous outcomes, their system demonstrated an approach that was more efficient in
terms of survival than pre-scripted behaviour. In the game industry, companion design is
taken very seriously, but formal techniques to do so are still lacking. For example, in the
game Bioshock Infinite, level design was specially made to allow one simple path from
start to the end of the game where the companion can take for granted on which graph
segment the player will travel [anonymous14]. This allows them to make sure the player’s
companion is always in her FoV.

10.2 Attrition Games: Targeting Enemies

The general target selection problem consists of two teams attacking each other, with each
entity selecting an opponent to fight as seen in chapter 3. The goal is to maximize the
total team health at the end of the combat. We find this problem in FPS, real-time strategy,
adventure, and other games. Work by others has closely examined the case of 1 player
against \( n \) enemies, showing that the problem of minimizing health loss for even a single
player is NP-Hard [FB10]. This problem is normally solved using look-up tree search. The
10.2. Attrition Games: Targeting Enemies

A naïve way to do it would be to explore every possible strategy at every state, reaching all possible end-game states [MF09]. From there the optimal choice is the one propagating back from the leaf with the best solution. Even for small scenarios, however, exponential growth in the size of the state space makes such an exhaustive search unrealistic in practice, at least within the context of real-time commercial games.

Look-up tree search typically assumes that players play in discrete turns. In a real-time environment this does not hold, as entities take time to perform actions and may do multiple actions between opponent moves, magnifying the branching factor in a tree search. Churchill et al. explored ways to reduce the space of exploration in real-time strategy games by using an alpha-beta search variant [CSB12]. They were able to solve an 8 vs. 8 combat using a game abstraction model that reduces the search space by focusing on important game features. Although this was done in real-time (50ms), the relative cost of this approach is still expensive for the rapidly changing context of FPS games, where high frame-rates are paramount, and CPU is limited.

Heuristic valued approaches offer a more efficient solution by attempting to approximate enemy value through a weighted function of observed data. The enemy with the highest aggregate score is then selected as a target. In the game Uncharted 2, for instance, they used a target selecting system that computed, for each enemy, a weighted function of distance, cover, whether the enemy shot the player last time, who the enemy is targeting, close-range radius, and so on [Rus10]. In general, their NPC would try to target different enemies by adding a negative weight when multiple entities target the same enemies, while staying on that enemy using a sticking factor. This approach can be effective, but as a complex and highly heuristic measure it must be closely tuned to a given context, and does not necessarily result in overall better combat results.

Finally, we note that many games offer some level of manual control over target selection. In Drakensang, a player may override a companion’s target choice, and direct her toward a specific enemy. Such extra-narrative control avoids the need for optimal target selection, but requires invoking an out-of-game interface, and if frequently necessary reduces companions from teammates to tools. Middle ground is found in games such as Dragon Age 2, which lets the player choose very high-level strategies for her companion(s), toggling between aggressive or defensive mode. This reduces the interaction complexity by
hiding detail, but also makes it less obvious to the player what the companion will do, without giving confidence that the best results will be achieved.

10.3 Game Solvers

Effective analysis of game levels implies solving them to some degree at least, and a significant amount of work on level design and analysis has made use of different AI search-based techniques for this purpose. The first edition of the *Mario AI Competition* [TKB10] provides a useful benchmark for learning and search techniques applied to the popular series *Mario Bros.*, particularly with respect to player simulation. Baumgarten’s controller, for example, won the competition using an implementation of A*. The solver used a physics simulator to project the resulting position of player actions. The output of the solver is then used to figure out which state is the closest to the goal (reaching furthest to the right of the screen), and thus which to evaluate next. *Cloudberry Kingdom* by Jordan Fisher uses a similar approach that defines a player AI, which is then used to procedurally generate different platformer levels [Fis12]. Monte-Carlo is also a popular and efficient choice when it comes for general game playing, and schemes can be implemented to learn different heuristics for evaluating tree states for a variety of games [FB08].

In a more general context, game solvers allow designers to explore various aspects of level design as we explored in chapters 4, 5 and 6. Nelson, for instance, proposes different fundamental questions game designers might ask an AI tool about their designs, such as whether and how something might be possible in the game [Nel11]. Our work shows how a tool could answer some of these questions, such as through the use of heat map visualizations that summarize a range of possible game behaviours. Other criteria for AI tools has emphasized the need for game-independent frameworks, that could be reused within different games genres [Smi13]. In this sense, the general search techniques we examine are highly relevant. RRT in particular has been previously shown to be a powerful search algorithm that can be made game-independent, requiring quite little modification to accommodate specific game mechanics, at least for reasonably simple movement models [BCP13]. Shaker et al. presented another, search-based approach to solving continuous-time game
10.3. Game Solvers

constraints in the puzzle-game Cut the Rope [SST13]. They discretized time, finding the
next game state by using a physics simulator to update the world, and then exploring the
state space through a depth-first search of the available actions. This allowed them to
present a solution to a game level as a path in a tree of game actions. Our search imple-
mentations have a similar challenge in discretizing a complex state space, but are applied
to larger state spaces, and aim at finding a range of game solutions representative of human
exploration. Other work, such as by Scales et al., lets AI programmers implement their
own controllers, using an API within the Spelunky GameMaker implementation [Sca14],
as we saw in chapter 8. Spelunky is a richer context than the platformer context we saw in
chapter 5, where the player has to survive multiple enemies, break walls to move, gather
different resources, etc. Scales designed different bots for this context, with different objec-
tives such as gold-digger, or explorer. The controllers developed for this project were based
on decision trees, using A* search for simple path planning, but not for actions. Neural net-
works can also be used to learn play policies using the game pixels as input. Mnih et al.
successfully implemented convolutional neural networks that learn how to play seven Atari
arcade-style games to competitive level [MKS+13]. In order to learn the playing policies
they had to use more than 10 million game frames multiple times, making the learning
phase computationally expensive. This time limitation might not suit the game developers
needs as adding new rules can require that the artificial player relearn how to play the game.

Chapter 6 presented work in finding stealthy paths in an abstract polygon which has
been studied in different contexts and with different flavours. This includes military inter-
est, and Bortoff presented an algorithm for unmanned aerial vehicle planning, computing a
stealthy path through a set of static enemy radar detectors [Bor00]. The presented solution
constructs a rough path using a Voronoi graph representation of the radar field and then
uses ordinary differential equations to determine a valid flying solution. Stealth path con-
struction can be seen as a variant on other search and interception-related pathing problems,
often considered from more theoretical perspectives. The famous Lion and Man problem,
for example, involves both seeking and avoiding behaviours [Cro64]. In the original for-
mulation of this problem, a lion and a man are in a circular area where each is given a maximal
speed. Given the parameters, the central question posed is whether the lion can catch the
man in finite time. This problem has been analysed with many different constraints, such
as giving the lion a line of sight visibility, wherein she can only observe the man with direct line of sight in an non circular geometry [NI14]. A variety of other approaches to finding or intercepting targets have also been developed. Lavalle et al., for instance, introduced a visibility-based motion planner that coordinates a multi-robot system to find a hidden target [LLG+97].

In the context of more specifically finding stealth paths, Ravela et al. presented a stealth planning system that uses visibility analysis on digital terrain elevation maps while minimizing the probability of getting seen [RWD+94]. Geraerts and Schager introduced an algorithm using a corridor map abstraction to find “pseudo”—which does not assure not getting seen—stealthy paths in real time [GS10]. A corridor map consists of finding a road map for the geometry, normally using median skeleton, and updating it for collision free movement. Tews et al. presented an approach that uses a potential field to build a path using a shadow map [TMS04]. Where a potential field consists simply as attraction or repulsion towards certain locations, pushing the stealth agent towards specific goals. Birgersson et al. discussed an approach to stealth pathing based on potential fields applied to situations where a robot is unaware of its environment; once it identified the observer (guard), it will move to a cover location [BHS03]. None of the approaches discussed here assure a fully stealthy solution, however, whereas the approach we develop in chapter 6 uses a time-space construction allowing us to guarantee stealthy paths as long as the observer behaviours are deterministic.

10.4 Design Tool

The work we describe in this thesis can be seen in the context of several research efforts which aim at building design tools that incorporate AI techniques into the game creation process. These typically build on customized forms of search within a specific game domain. Jaffe et al., for instance, explore the concept of fairness/game-balance in games based on win-rate [JMA+12]. Using different search algorithms, such as greedy search, they showed how strategies in a card game could change with different card values, and
so give designers important balancing information. More specifically aimed at simulating stealth games, Pizzi et al. abstracted the level structure and gameplay of the popular video game *Hitman: Blood Money* into discretized events that could be assembled into storyboards [PLWC10]. These storyboards represent major actions such as walking to a point, taking down an enemy, procuring an item, etc. This abstraction allowed the designer to build a restricted level (placing enemies and items), where a planner would find a path to the winning position. Following the sketching tool concept, Liapis et al. presented a tool where designers could draft levels using a high-level terrain editor which included the previously mentioned metrics of symmetry, area control, and exploration [LYT13].

A core problem in tool design is in how to present the complex information the tool computes to the designer. Once gameplay traces are obtained, for instance, whether artificially generated or not, some form of visualization is essential to the design process. A basic approach to this problem is to use different heat maps or influence maps, such as is commonly done to model player death positions [HJ09]. Understanding the design space is also achievable using feature-based state projections, where high dimensional states are mapped to lower dimensions and clustered together [LAS11]. Our work incorporates the latter, and includes multiple forms of visualization as an essential part of making the use of our search techniques practical and useful to game designers.

10.5. In-Game Experience Analysis

Our work is intended to better understand and be able to model player behaviour in digital video games. In chapter 7 we investigated how we can use different metrics to model the stealth game difficulty. A significant amount of previous work has been directed at analyzing and modelling player behaviours. Early informal work was done by Bartle as a categorization of player types [Bar96], he defined four basic kinds of players, Achievers, Socializers, Explorers, and Killers, with each type having different motives to interact with a game. A game’s success can then be partly explained by how much it appeals to the different types, as well as the ecology of types a game attracts. Subsequent, more formal studies have since confirmed this rough categorization [Ali05]. A different approach is
In-Game Experience Analysis

taken by Sweetser and Wyeth [SW05], who introduced the Gameflow model to evaluate player enjoyment in games regardless of type. This model borrows from the flow theory in psychology originally proposed by Csikszentmihalyi, on which they added a level of abstraction specifically oriented to games. Like Bartle’s work and other statistical studies, however, this does not offer a clear formal model at how a machine could, without the help of a human being, evaluate a human player’s experience/trace. In the game series Left 4 Dead 1 and 2 from Valve, an “intensity” metric is used to represent player game experience. The metric varies positively when a player is injured by an opponent, is incapacitated by an opponent, pushed or pulled off a ledge by an opponent, and when nearby opponents die, and is reduced by less stressful stretches of gameplay. The AI Director uses this knowledge about the player’s experience to create a tailored game [Boo09]. Measuring player experience in this fashion has since been applied in a number of different contexts, including an analysis of difficulty in World of Warcraft [AV11] and to this work, see chapter 3. Holmgärd et al. incorporated utility functions in MCTS to simulate different types of players in a simple dungeon game [HLTY15]. For example, you can give the utility function high values for kills, which causes MCTS to find solutions that are risk seeking.

Geometric models have also been applied to game analysis as a means of evaluating how players (may) experience a game level. Liapis et al., for example, translated some game design patterns [BH04] into simple algorithms that describe symmetry, area control and exploration [LYT13]. They showed how these different metrics could be used to optimally evolve different levels. Metrics algorithms are often closely linked to generative methods for game content [COM13]; many generative processes follow a core structure of generating some content, e.g., a game level, followed by using a metric-based utility function on the content to determine the resulting quality [TYSB11]. For example, in the work of Togelius and et al., they presented generative methods to evolve race tracks, measuring the result using neural network-based agents [TDNL07]. This allowed them to evaluate the quality of the track, measuring amount of progress, variation in progress, and difference between maximum and average speeds.

Perhaps the closest prior to the work presented in chapter 7 is by Shi and Crawfis, who presented a design tool that computes metrics on the optimal path a player may find to get
10.6. Simulating Human Movement

through a level, given obstacles and enemy distribution [SC13]. They considered properties such as the minimum damage cover, longest path, and standard deviation of cover points. We are concentrating on metrics relevant to the stealth games genre, in chapter 7, but incorporation of these kinds of FPS metrics would be interesting in more complex situations, where stealth and combat combine. Both their work and ours fall under Nelson’s state space characterization strategy [Nel11]. This characterization consists of probing a game’s state-space from various perspectives, e.g., consisting in reducing the representation for a better understanding.

10.6 Simulating Human Movement

In chapter 6 we examined how path solutions generated by RRT differ from actual human paths. Ensuring synthetic paths are similar to human results is an important factor in mechanising game design. The idea of trying to algorithmically generate player movements is of course not new. Social agent studies in particular aim directly at mimicking human movements. In this context there is a large body of work simulating human crowds as well as defining low level human-like motion planners or movement algorithms. Karamouzas et al., for example presented a collision avoidance model for simulating pedestrians [KHvBO09]. Olivia et al. presented an approach to crowd simulating that includes social simulation between agents [OV14]. With respect to low-level control of motion, seminal work by Reynolds presented a particle system for multi-agent contexts that emulates complex behaviour such as birds flocking [Rey87]. His base model has been greatly extended in various ways, such as in recent work aimed at simulating humans following each other [BDP14]. Use of potential field approaches have also been quite successful in modelling movement, and included in more general search algorithms [KNS+13, GO07].

In chapter 6, we compared the solutions produced by RRT to human ones in terms of overall path choices and coverage properties, assuming single-player game environments. Previous work in this specific area can be found in Brogan and Johnson’s study presenting a human inspired model for path finding [BJ03]. They presented three similarity metrics to evaluate the realism of the approach, which are interesting, but aim at specific path-to-path
10.7 Generative Content

In chapter 9 we presented an approach to generate non-interactive content in a given level. We were generally interested in placing game content in a polygon such that it respects certain properties. The most well studied form of this problem can be expressed as placing single or multiple polygons $P$ (content) inside a polygon $Q$ while minimizing the Euclidean distance from the vertices of $P$ to the vertices of $Q$ [AIIT90]. A problem more similar to the one of interest in this paper is the chromatic art gallery problem, aiming to ensure that landmarks are distinguishable [EL11]. This is itself similar to the well known art gallery problem, but aiming at the minimum number of colours needed when guards are given different colours and their visibility regions overlap. This approach could be extended to apply to content positioning, although their technique does not take into consideration player movement within a level.

The research problem of generating high-level game-level maps is a well known problem; decorating a level with non-puzzle content, however, seems to be neglected. Dormans and Bakkes presented a grammar-based approach that generates missions that are mapped to a polygon representing a level (similar to the level we show in figure 9.9) [DB11], although the levels created are not populated with content. Smith et al. and Horswill & Foged used constraint solvers in order to assure that level puzzle structures were kept when
populating a level with content [SAMP12, HF12].

Player motion is an important aspect of content decoration, as any choice taken by
designer or in generative methods can influence the player’s movement. Winters and Zhu
presented a study that investigated the influence of level design paradigms on player path
choices [WZ14]. They showed that shifted elevation such as stairs and directional lines
had the most influence when making movement choices. Milam and El Nasr presented 5
design patterns used by developers to orient players based on their sight [MEN10]. These
patterns could be used in the work presented in chapter 9 to help orient the player around a
given polygon level.

10.8 Chapter Conclusion

In this chapter we investigated different related work on the subjects this thesis touched
on, including companion behaviour, game adaptivity, game solvers, content generation,
mimicking human pathing, etc. In the following chapter, we will present the different
principles we found while developing the different models presented in this thesis. We will
conclude this thesis by presenting a general framework to follow when using solvers in the
game design process.
Chapter 11
Future Work and Conclusions

The introduction of computing techniques in the game design process is a non-trivial task. This thesis explored different mechanisations such as reactive agents, search algorithms, state space definitions, etc. in order to help designers gain insight on their creations. This conclusion describes the principles found while developing the different tools presented in this thesis. These principles describe how search algorithms can be used to empower game developers in designing digital games. We also discuss the different goals presented at the beginning of this thesis and the contributions they produced. We finally present different directions for future research inspired by the work presented.

11.1 Architecture

While developing the different game design tools presented in this thesis such as for stealth game or platformer genres, we noticed that various problem addressed could be generalized, and thus turned into a framework that abstracts the overall process of integrating search-based mechanisms into the game design process. This framework is intended to be relatively agnostic to the game genre. In this sense it works as a service that game developers and designers could use to test their games and receive meaningful data from the artificial game tester as a proxy for real player data showing possible in-game solution. To do this a game designer would need to provide the framework with a formal game representation, and model of the game rules as this thesis did.
11.1. Architecture

Figure 11.1 presents the general architecture. The design is greatly inspired by the Randomly Rapidly exploring Tree search (RRT) algorithm [LKJ00], although it accommodates other search algorithms, and is integrated into a level analysis tool. The user needs to provide the game representation, including a level with start and goal positions to initialize and terminate the process, and a state budget for bounding the search costs. A motion model, informed by game rules that define allowable behaviours and measure state differences must also be provided. In this dissertation we also defined the motion model as the update function that determines the result state of applying action $a$ to the state $\sigma$.

The process starts by sampling a state from the game representation; for a simple 2D movement game (e.g., pacman), this is any point position within a level. The validation step checks if it is possible to reach the sampled state from already explored states or move towards it, using a motion planner or motion verification to check if the sampled point is indeed reachable. This motion system can be provided by the game rules, or directly derived from the game engine physics. Which we normally express as a set of actions available within the game modeled. If the sampled node is non-reachable, the process returns to the sample step and repeats. If the sampled state is reachable, it is added to a data structure to keep track of what is already explored in the expand step. At this point
11.2. Goals & Contributions

the process verifies if a terminal state was reached, either by reaching a goal position or exhausting the state budget. This process encompasses multiple search algorithms using a given state space definition, and can be called multiple times to generate a collection of solutions. The framework also accommodates different visualizations, such as movement heat maps, which can be incorporated into the game engine.

Designing video games can be intensely iterative, with designers refining levels and mechanics based on human play-test data. A generic framework could leverage a variety of state-of-the-art search techniques to find random, or deterministic game solutions can effectively eliminate most of the labour-intensive aspect of play-testing, improving the design process. Our experience in this thesis strongly suggests such a framework would be feasible and effective, and the development of one would make excellent future work, which could also have significant practical value to game designers.

11.2 Goals & Contributions

This dissertation has the goal of improving the game design process through the use of mechanised processes, allowing designers to receive more immediate and quantitative feedback on their creations. We began with a study of problems related to companion NPCs, and used our approach to that problem area as motivation to further investigate the use of algorithmic, computational techniques in the design process.

11.2.1 Companion NPC

In different game genres, companions support players to complete in-game tasks. We investigated problems related to combat load, exploring which behaviour a companion should follow and its influence over the in-game playing experience using quantitative metrics. This section also demonstrated the value of using mathematical properties to model even modern game problems, letting us represent and then experimentally validate solutions to the problem of improving companion targeting. We introduced the concept of using a reactive agent as a means of artificially testing a game design; the in-game controls given to an artificial player based on a pre-determined formalism such as a behaviour tree then dictates
11.3. Future Work

how the agent reacts to in-game situations.

11.2.2 Game Design Mechanisation

While a significant improvement over beta-testing, using reactive agents for testing purposes is still tedious, as it takes long period of time to get meaningful data—the game still runs in real-time and the solutions found are limited to being a function of the agent’s behaviour reactions. Given these restrictions we explored using mathematical representations in order to explore a developer’s design for different game genres, including platformer, stealth games, and combat games. This approach allows the designer to use formal thinking about the state space, such as in proving a solution exists to her creation. We developed different algorithmic measurements which allows a designer to identify the level’s difficulty, its possible solutions, etc. We also investigated different search algorithms and their applications in the context of game design. We compared them in terms of efficiency and explorability, where we argued that designers are interested in both. We believe that this work is the first to investigate stealth pathing from a perspective of space-time, offering a particular view on the problem. We also investigated how the rapidly-exploring random tree algorithm compares to human paths and proposed different algorithmic solutions to mimic human paths better. The last piece of work presented in this thesis investigated computing techniques in order to place non-interactive content in a given level.

Our various experiences and approaches to formalizing analysis and integrating it into the design process also suggest the potential for a more general framework, as shown in the previous section. The specific search algorithms, metrics, and game contexts we used could be abstracted into a template approach to mechanising game design. Implementation and validation of this general approach remain future work, but we feel this kind of system would have wide applicability to a range of game design problems.

11.3 Future Work

The presented work in this thesis can be extended in multiple directions to further the mechanisation of game design. We will focus our argument on specific improvements to
the expressiveness of the presented computational techniques. Proposed augmentations can be divided into two categories, stealth games, and a game design tool.

11.3.1 Stealth games

We presented different computational models in addressing stealth games, in which we argued that a space-time construction allows for in-depth analysis and search. Stealth games, however, often have a number of additional features and mechanisms we did not consider in our abstraction. We are thus interested in exploring the influence of different game mechanics such as noise, light, walking vs. running, etc., all of which are technically straightforward to incorporate, and would make the stealth tool even more relevant to the reality of modern digital games. We are, also, interested in extending the representation used to incorporate guard interactions, such as use of distractions. For example, a guard might be blocking the only possible path to the goal state, a human player might attract that guard to a given position using distractions that create a noise, such as by throwing stones or breaking bottles. Once the guard leave his position, it becomes possible for a player to sneak towards her goal. Since the guard reaction is deterministic, it is feasible to include such a mechanism in the state space representation. A first challenge, however, arises from finding a valid position to attract the guard; a Monte-Carlo approach could be used where different positions are tried and the search continues from there, although this approach will also create a state explosion. A better approach might be to investigate the level configuration using geometric analysis, to determine a discrete set of abstract locations that are worth considering in this context.

In chapter 3 we presented an adaptive approach to companion behaviour. One of the motivations behind that work was to respect the player intention, especially while sneaking behind an enemy. An interesting application of the presented solver on the stealth space would be in extending it to be online, allowing the companion to find a stealthy path dynamically, in response to a player action. This online context brings a new set of challenges such as the need to be especially efficient in computing stealthy paths to avoid frame-rate drop in gameplay. Outside of the use of explicit stealth modes, this also implies the need to algorithmically determine the player’s goal, which is not a trivial task.
11.3. Future Work

11.3.2 Game Design Tool

This dissertation investigated different approaches to incorporate computing techniques in the design process. These all relied on the definition and computation of specific metrics that measure aspects of the game experience, and there are many ways both the metrics and their computation could be extended. We are interested, for example, in extending the general in-game experience measurement, which in chapter 3 was represented by a game intensity metric. The measure could be made genre agnostic, with the developer defining the events that can influence it. Better, deeper validation would also be useful. An investigation of different game contexts and presentations, for instance, would help determine whether the importance of the distance metric considered in chapter 7 is indeed general, or whether different game contexts and representations may imply other metrics are better. Optimizing the efficiency of metric calculation is also a concern; performance requirements of most games mean that path-distance, and other computationally expensive calculations cannot be done with real-time granularity. Our main interest, however, is in applying metrics to improve game design, providing interactive input to the designer, using metrics to quantify design patterns or principles, and in using dynamically computed risk values to identify player styles for improved adaptivity in gameplay and NPC behaviours. To do this an appropriate visualization of metric results is quite important. The main process used to describe results in this dissertation is a heatmap. Heatmaps are relatively common, but still represent an under-studied tool for understanding multidimensional data in games. A useful extension would be in generically representing abstract events of interest to designers, such as combat or puzzle solving.

Finally, we also investigated the humanness of RRT produced solutions. It would also be interesting to consider other search processes, such as Monte Carlo Tree Search. For this MCTS has an advantage over RRT in that a heuristic is embedded in the evaluation of each step, and this may simplify the process of enforcing human-like path choices.
Appendix A
Stealth Risk Human Study

In this appendix we include all of the levels layout used in our human study to investigate human perception of stealth risk presented in chapter 7.

Figure A.1: level 1
Figure A.2: level 2

Figure A.3: level 3
Figure A.4: level 4

Figure A.5: level 5
Figure A.6: level 6

Figure A.7: level 7
Figure A.12: level 12

Figure A.13: level 13
Figure A.14: level 14

Figure A.15: level 15


Bibliography


Bibliography


273


Bibliography


Bibliography


Robert Daniel Scales. Create a set of AI tools for Spelunky that will allow users to program their own player bots. *Discovery, Invention & Application*, 2014.


Conference on Computational Intelligence and Games, CIG, pages 1–8, August 2014.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>